Upravljanje mobilnim robotom primjenom robotskog operativnog sustava (ROS)

Šebalj, Karlo

SVEUČILIŠTE U ZAGREBU
FAKULTET STROJARSTVA I BRODOGRADNJE

DIPLOMSKI RAD

Karlo Šebalj

Zagreb, 2020.

SVEUČILIŠTE U ZAGREBU
FAKULTET STROJARSTVA I BRODOGRADNJE

DIPLOMSKI RAD

Mentor: Student:

Prof. dr. sc. Bojan Jerbić, dipl. ing. Karlo Šebalj

Zagreb, 2020.

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

Zahvaljujem mentoru prof. dr. sc. Bojanu Jerbiću i doc. dr. sc. Marku Švaci što su mi dali priliku raditi na ovom radu kroz suradnju s tvrtkom Greyp Bikes te na svim savjetima i pomoći koju su mi pružili tijekom izrade ovog rada. Izuzetno sam zahvalan voditelju dr. sc. Gabrijelu Smoljkiću što me cijelo vrijeme usmjeravao stručnim savjetima i sugestijama te kolegi Mati Mihaljeviću koji mi je pomogao pri integraciji baterije sa sustavom za upravljanje radom baterije. Htio bih se zahvaliti tvrtki Greyp Bikes koja mi je omogućila izradu diplomskog rada s potrebnom opremom te svim zaposlenicima koji su mi svojim profesionalnim savjetima pomogli u rješavanju problema na koje sam naišao. Također, htio bih se zahvaliti prof. dr. sc Ivanu Markoviću na korisnim savjetima i uputama za upravljanje mobilnim robotom.

Zahvaljujem svojim roditeljima, bratu i djevojci što su mi bili najveća podrška tijekom cijelog studija te svim svojim prijateljima i kolegama koji su bili uz mene i na savjetima koje su mi dali.

Karlo Šebalj

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

Sveučilište u Zagrebu Fakultet strojarstva i brodogradnje	
Datum:	Prilog:
Klasa: 602 - 04 / 20 - 6 / 3	
Ur. broj: 15 - 1703 - 20 -	

DIPLOMSKI ZADATAK

Student: **KARLO ŠEBALJ** Mat. br.: 0082049890

Naslov rada na hrvatskom jeziku: **Upravljanje mobilnim robotom primjenom robotskog operativnog sustava (ROS)**

Naslov rada na engleskom jeziku: **Controlling a mobile robot using a Robot Operating System (ROS)**

Opis zadatka:

Razvoj upravljačke podrške za autonomne robote moguće je ostvariti na različitim softverskim platformama od kojih je robotski operativni sustav (eng. Robot Operating System – ROS) standard koji je postao opće prihvaćen u akademskoj zajednici, a sve se više se koristi i u industriji i uslužnoj robotici. Robotski operativni sustav pruža okolinu za razvoj modularne upravljačke programske podrške, komunikacijsku infrastrukturu koja povezuje programske komponente, te otvorenu biblioteku implementiranih algoritama. U svojoj osnovi ROS se sastoji od niza jednostavnijih računalnih programa tj. ROS čvorova, koji se međusobno povezuju i razmjenjuju informacije.

U okviru ovog rada potrebno je:
• Integrirati sustav ARIA – ActivMedia Robotics Interface for Applications na postojeći Pioneer 2DX mobilni robot
• Istražiti i eksperimentalno validirati povezivanje robota s ostalim komponentama za upravljanje: kamera, senzori, kontroleri za motore, upravljačko računalo
• Izraditi programsku podršku koristeći ROS za upravljanje kamerom, kretanjem robota, prikupljanje informacija sa senzorskih sustava
• Implementirati i evaluirati ručno upravljanje mobilnim robotom
• Istražiti mogućnosti autonomnog upravljanja mobilnim robotom s ciljem praćenja definiranih objekata
S ciljem poboljšanja i nadogradnje pojedinih modula Pioneer 2DX mobilnog robota potrebno je istražiti i prema mogućnosti eksperimentalno evaluirati mogućnosti integracije kompatibilnih komponenata iz tvrtke Greyp bikes d.o.o.:
• integracija Greyp baterije s odgovarajućim sustavom za upravljanje radom baterije
• integracija i testiranje odometrijskih podataka dobivenih s Greyp integrirane razvojne pločice
• integracija Greyp 2D kamere s odgovarajućim softverom
• Istraživanje mogućnosti integracije ARM razvojnog sustava za udaljeno upravljanje
U radu je potrebno navesti korištenu literaturu i eventualno dobivenu pomoć.

Zadatak zadan:	Rok predaje rada:	Predviđeni datum obrane:
30. travnja 2020.	2. srpnja 2020.	6. srpnja do 10. srpnja 2020.

Zadatak zadao: Predsjednica Povjerenstva:

prof. dr. sc. Bojan Jerbić prof. dr. sc. Biserka Runje

SADRŽAJ

POPIS SLIKA

POPIS TABLICA

POPIS OZNAKA

Oznaka	Jedinica	Opis
x	mm	Koordinata u smjeru x osi
y	mm	Koordinata u smjeru y osi
θ	rad	Kut orijentacije mobilnog robota
v	mm/s	Translacijska brzina mobilnog robota
ω	rad/s	Rotacijska brzina mobilnog robota
v_r	mm/s	Vektor translacijske brzine desnog kotača
v_l	mm/s	Vektor translacijske brzine lijevog kotača
ω_r	rad/s	Vektor rotacijske brzine desnog kotača
ω_l	rad/s	Vektor rotacijske brzine lijevog kotača
l	mm	Duljina robota
R	-	Matrica rotacije
$\dot{x}$	mm/s	Brzina mobilnog robota u smjeru x osi
$\dot{y}$	mm/s	Brzina mobilnog robota u smjeru y osi
$\dot{\theta}$	rad/s	Brzina rotacije mobilnog robota
K_p	-	Pojačanje P regulatora translacijske brzine
K_h	-	Pojačanje P regulatora rotacijske brzine

SAŽETAK

Zadatak diplomskog rada je razviti upravljanje mobilnim robotom primjenom robotskog operativnog sustava (ROS). Upravljanje robotom pomoću robotskog operativnog sustava pruža korisniku okolinu za razvoj modularne upravljačke programske podrške, komunikacijsku infrastrukturu koja povezuje programske komponente te otvorenu biblioteku implementiranih algoritama. Potrebno je ispitati sve komponente sustava mobilnog robota, validirati njihovu funkcionalnost te ih zamijeniti s odgovarajućim novim komponentama. Nove komponente potrebno je povezati sa starim komponentama u jedinstven sustav te izraditi programsku podršku koristeći ROS za njihovo upravljanje. Primjenom jednostavnog upravljanja mobilnim robotom treba razviti složeno upravljanje. Razvijeno složeno upravljanje treba omogućiti dva načina, ručno i autonomno upravljanje.

Ključne riječi: mobilni robot, robotski operativni sustav, ROS, čvor, upravljanje, softver, integracija, razvoj

SUMMARY

The task of this Master's thesis is to develop the control of a mobile robot through the application of Robot Operating System (ROS). Robot Operating System enables robot control and provides the appropriate environment for development of modular management program support, communication infrastructure which connects program components as well as the open source library of implemented algorithms. In order to complete this assignment, it is necessary to review all components of a mobile robot system, test their accuracy and replace them with suitable new components. Furthermore, it is required to connect new components with the old components in order to make a unique system and to create program support using ROS for its control. Complex control, which should be developed using simple control of a mobile robot, enables two types of control, manual and autonomous.

Key words: mobile robot, Robot Operating System, ROS, node, control, software, integration, development

1. UVOD

Industrijska i mobilna robotika doživjela je veliku transformaciju u posljednjem desetljeću. Do sve većeg razvoja robotike došlo je zbog potrebe za ubrzavanjem i olakšanjem načina proizvodnje pojedinih dijelova, ali i razvojem dijelova koji omogućavaju sve bolji nadzor i upravljanje. Trenutno je to jedno od najbrže rastućih područja znanstvenog istraživanja kod kojeg se očekuje još i veći napredak u idućim desetljećima. Zbog svojih sposobnosti roboti mogu zamijeniti ljude u različitim dijelovima industrije. Područje primjene robotskih sustava vrlo je opsežno i obuhvaća nadzor, automatizaciju i intervenciju na teško dostupnim mjestima, a primjenjuje se i u izvođenju operacija, pružanju medicinske njege, osobnih usluga te u brojnim drugim industrijama.

Robotski operativni sustav (ROS) je skup alata, biblioteka i niz računalnih programa koji su međusobno povezani u jedan sustav i omogućuju jednostavnije izvođenje funkcija poput kontrole, manipulacije, navigacije pružajući korisnicima infrastrukturu i najbolju praksu za izradu poboljšanih i novih aplikacija u svijetu robotike. Razvoj ROS-a predstavlja revoluciju u robotskoj zajednici i doprinosi sve češćoj primjeni robotike kod velikog broja korisnika. Promjenom načina pisanja programa pojednostavljena je izrada robotskog upravljanja te je tako korisniku približio robotske sustave bez znanja o izradi složenih robotskih programa.

U ovom diplomskom radu opisana je integracija postojećih dijelova mobilnog robota s novim poboljšanim komponentama kako bi robot mogao normalno funkcionirati. Prikazana je implementacija načina programiranja i kontrole procesa u robotu pomoću robotskog operativnog sustava (ROS). Objašnjeni su i prikazani na primjerima načini upravljanja robotom. Isto tako, prilagođeni su i integrirani dijelovi iz tvrtke Greyp bikes d.o.o kojima je dokazano da mogu funkcionirati na drugom sustavu kao što je mobilni robot. Na kraju su objašnjeni i definirani dijelovi kojima bi se mogla napraviti potpuna nadogradnja mobilnog robota s novim i boljim komponentama.

2. MOBILNI ROBOTI

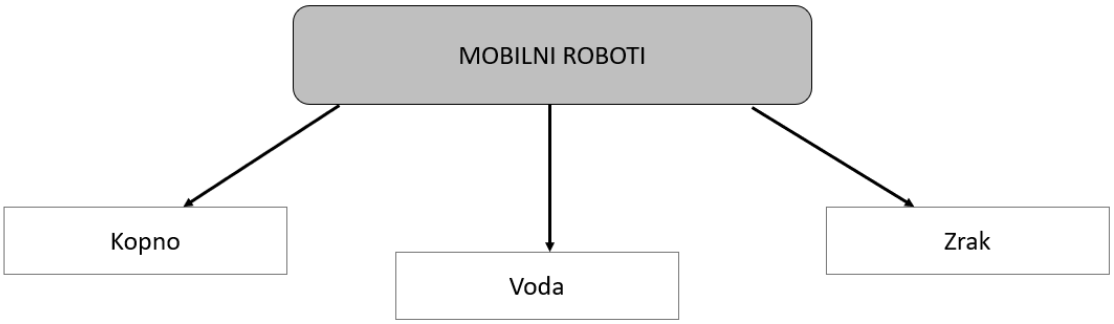
Mobilni robot je upravljivi sustav koji koristi senzore i drugu tehnologiju za prepoznavanje svoje okoline i kretanje kroz nju. Mobilni roboti funkcioniraju koristeći kombinaciju upravljačkih algoritama i fizičkih robotskih elemenata, poput kotača, staza ili nogu. Koriste se za primjenu u različitim industrijskim sektorima i čak za izvršavanje zadataka koji su za ljudske radnike nemogući ili opasni. [1]

Humanoidni roboti, dronovi, bespilotni roboti kopači, roboti kućni ljubimci i brojni drugi neki su od primjera mobilnih robota. Razlikuju se od ostalih tipova robota po svojoj sposobnosti da se mogu kretati samostalno, s dovoljno inteligencije za reakciju i donošenje odluka na temelju percepcije koju dobivaju iz okoline. Kako bi mogli percipirati informacije iz okoline i odgovoriti na okoliš u kojem se nalaze, mobilni roboti moraju imati neki izvor ulaznih podataka, način dešifriranja i obrade tog unosa te način poduzimanja radnji (uključujući i vlastiti pokret). Potreba za osjećajem i prilagođavanjem nepoznatom okruženju zahtijeva snažan kognitivni sustav. Razvoj tehnologije omogućio je da kretnje mobilnog robota, poput trčanja, hodanja, skakanja, budu što sličnije ljudskima. Dolazi do razvoja novih trendova kod mobilnih robota, kao što su umjetna inteligencija, autonomna vožnja, mrežna komunikacija, kooperativni rad, nanorobotika, sigurna interakcija čovjeka i robota te primjena izražavanja i percepcija emocija, koji se primjenjuju u području medicine, zdravstvene zaštite, sporta, ergonomije, industrije, distribucije robe i uslužne robotike. Postoji tendencija daljnjeg razvoja u narednim godinama. Pojavilo se nekoliko polja robotike koja uključuju različita tehnološka područja poput elektronike, mehanike i računalne znanosti. [2]

2.1. Vrste i klasifikacija mobilnih robota

S obzirom na to da se mobilni roboti mogu definirati i kao vozila koja se mogu gibati se kroz okoliš razlikujemo nekoliko vrsta mobilnih robota prema okolišu kroz koji se mogu kretati. To su mobilni roboti koji se mogu kretati na/po kopnu, preko ili kroz vodu ili kroz zrak što je prikazano na slici 1. Time se prikazuje raznolikost koja se smatra pod pojmom mobilna platforma. Međutim, između navedenih vrsta mobilnih robota postoje brojne sličnosti u predmetu i načinu rada. Jedna od najvažnijih funkcija mobilnih robota je mogućnost kretanja i dolaska na određeno mjesto. Osim prema okolišu kroz koji se mogu kretati, razlikujemo i mobilne robote u smislu podataka koje primaju iz okoline kao što je na primjer kretanje prema svjetlu, prema definiranom objektu ili u smislu referentne geometrijske koordinate. U oba

slučaja, robot će krenuti putem do svojeg cilja te pritom izbjegavati prepreke s kojima je suočen kako bi uspio uspješno doći do cilja. [3]

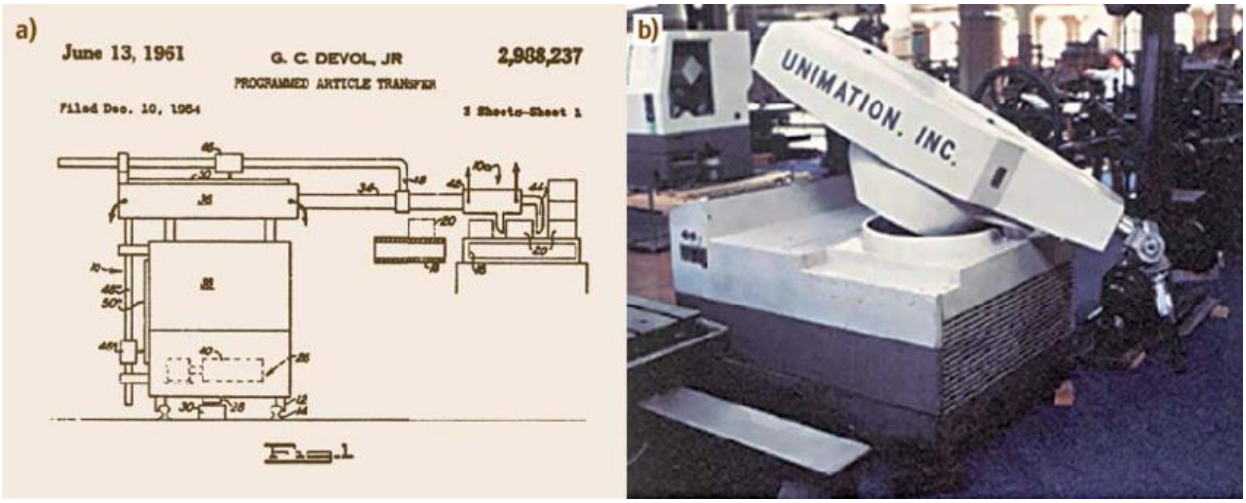


Slika 1. Klasifikacija mobilnih robota s obzirom na okoliš u kojem se kreću

2.2. Povijest mobilnih robota

Pojam robota prvi puta se pojavio u češkoj znanstveno-igranoj predstavi 1920. „Rossum's Universal Robos" koju je napisao Karel Čapek. Izraz je smislio njegov brat Josef što na češkom jeziku znači kmetski rad, ali kolokvijalno marljiv rad. Roboti u igri bili su umjetni ljudi ili androidi i slično mnogim današnjim pričama o robotima koje slijede ovu, roboti se pobune i to završi loše za čovječanstvo. [3]

Prvi patent za ono što bismo danas smatrali robotom napravio je 1954. godine George C. Devol i izdan je 1961. godine. Uređaj se sastojao od mehaničke ruke s hvataljkom koja je bila montirana na stazu i kodiranog redoslijeda pokreta kao magnetski uzorak koji se pohranjivao na rotirajućem bubnju. Prvu tvrtka za robotiku Unimation osnovali su Devol i Joseph Engelberger 1956. godine i njihov prvi industrijski robot prikazan na slici 2. lijevo. Izvorna vizija Devola i Engelberger za robotsku automatizaciju postala je stvarnost i mnogi su milijuni robota kao na slici 2. desno izgrađeni su po uzoru na njega. Roboti slični prvoj verziji danas rade na zadacima poput zavarivanja, lakiranja, utovara i istovara stroja, elektroničke montaže, pakiranja i paletiranja. Upotreba robota dovela je do povećanja produktivnosti i poboljšanja kvalitete proizvoda. Prva generacija takvih robota je učvršćena u mjestu i ne može se micati po postrojenju što znači da nisu bili mobilni. Sljedeća generacija robota počinje dobivati različite načine mobilnosti. Oni se mogu podići s poda koristeći noge ili kotače, letjeti iznad zraka koristeći krila ili im je dizajn prilagođen za gibanje kroz vodu. [3]



Slika 2. Izum industrijskog robota. Lijevo - patent robota G. Devola i J. Engelbergera koji su osnovali kompaniju Unimation. Desno - prvi instalirani robot u tvrtki General Motors [4]

2.3. Glavni sustavi mobilnog robota

Za normalnu funkciju mobilnog robota potrebno je promatrati i integrirati nekoliko tehnoloških područja. Svaki mobilni robot sastoji se od nekoliko podsustava i možemo ih podijeliti na mehanički, aktuatorski, senzorski i upravljački sustav. Ovi podsustavi međusobno komuniciraju kako bi robot mogao funkcionirati kao sustavna cjelina.

2.3.1. Mehanički sustav

Robot teško može funkcionirati u okolišu bez svojeg fizičkog dijela. Kada govorimo o fizičkom dijelu mislimo na kućište mobilnog robota koje drži sve ostale komponente na jednome mjestu. Mehanički dio odgovoran je da robotu osigurava određenu stabilnost i nosivost čitavog robota.

2.3.2. Aktuatorski sustav

Postoje dvije glavne aktivnosti koje aktuatorski sustav mora obavljati, a to su lokomocija i manipulacija. Lokomocija je sustav koji pretvara energiju propulzije u kretanje mobilnog robota što znači da robot mora moći doći iz točke A u točku B u prostoru. Lokomotorni sustav je vrlo bitan u svijetu mobilne robotike jer omogućuje glavnu svrhu mobilnog robota, a to je gibanje. S druge strane manipulacija je postupak premještanja jednog predmeta s jednog mjesta na drugo. Manipulacija se većinom razmatra kod robota kao što su industrijski roboti u obliku ruke čiji je glavni cilj premještanje predmeta ili obavljanje neke

radnje s određenim proizvodnim dijelovima. Kod mobilnih robota mogu se koristiti različite vrste aktuatora, od istosmjernih motora sa ili bez četkica pa sve do koračnih motora. Ovisno o vrsti problema primjenjuju se odgovarajući aktuatori.

2.3.3. Senzorski sustav

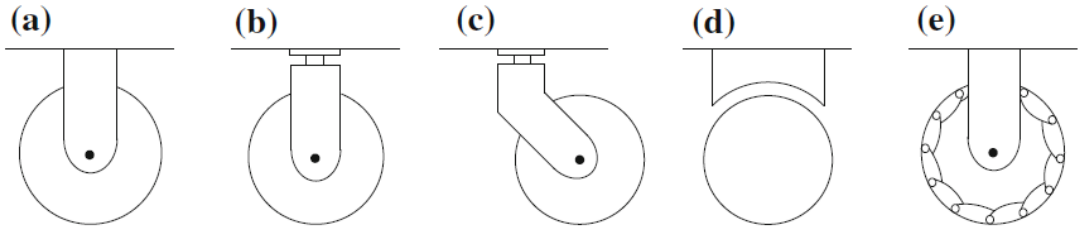
Sustav percepcije ili senzorski sustav odgovoran je da robot prikupi dovoljnu količinu informacija iz okoline kako bi na kraju imao pravu sliku o svome okruženju. Senzorski sustav se smatra najvažnijim dijelom robota jer omogućuje robotu da percipira i izbjegne razne prepreke te nepredvidive utjecaje iz okoliša na koje može naići pri gibanju kroz okoliš. Količina informacija iz okoline definira stanje robota koje u osnovi znači opis svih njegovih parametara u bilo kojem trenutku. Postoji široki raspon senzora koji se koriste u mobilnim robotima. Neki mjere jednostavne vrijednosti, kao što je unutarnja temperatura elektronskog dijela robota, a neki služe za mjerenje kutne brzine robota. Drugi, sofisticiraniji senzori se mogu koristiti za mjerenje pozicije robota u prostoru. Senzorski sustav možemo podijeliti na dva dijela. Razlikujemo eksteroceptivne i propriocepsijske senzore. Eksteroceptivni senzori uzimaju informacije iz okoliša mobilnog robota npr. senzori za mjerenje udaljenosti do objekta kao što su ultrazvučni senzori, IR senzori, laserski senzori, različite vrste kamera. Propriocepcijski senzori mjere unutarnja stanja robota kao što su brzina robota, opterećenje, kutovi zakreta zglobova robotske ruke, napon baterije.

2.3.4. Upravljački sustav

Posljednji sustav koji je dio mobilnog robota kao funkcionalne cjeline je upravljački sustav. Da nema odgovarajućeg kontrolera, robot ne bi mogao nikada biti potpuno autonoman. On uzima podatke sa senzora kako bi odlučio što će učiniti sljedeće te nakon toga poduzeo neke radnje koristeći aktuatore. Na prvu to djeluje jako jednostavno, ali upravo ovaj sustav je najsloženiji jer je ponekad parametre regulatora jako teško odrediti. Za većinu mobilnih robota i robota koji se koriste kao neka vrsta hobi robota, kontroleri su mikroprocesori koji su programirani u nekom programskom jeziku niske razine. Isto tako, nije neobično da robot koristi više od jednog kontrolera. S druge strane time se povećava kompleksnost sustava te integracija više od jednog kontrolera zahtijeva puno više vremena i puno je skuplje nego jedan kontroler.

2.4. Konfiguracije mobilnih robota

Najjednostavniji mehanizam koji omogućava prijenos gibanja kod mobilnih robota je kotač. Oni omogućavaju jednostavno tegljenje predmeta i dovoljna su samo tri kotača kako bi se dobila dovoljna stabilnost i ravnoteža mobilnog robota. Kotači se mogu dizajnirati na različite načine koji su prikazani na slici 3. koja prikazuje konfiguracije kotača mobilnih robota. [5]

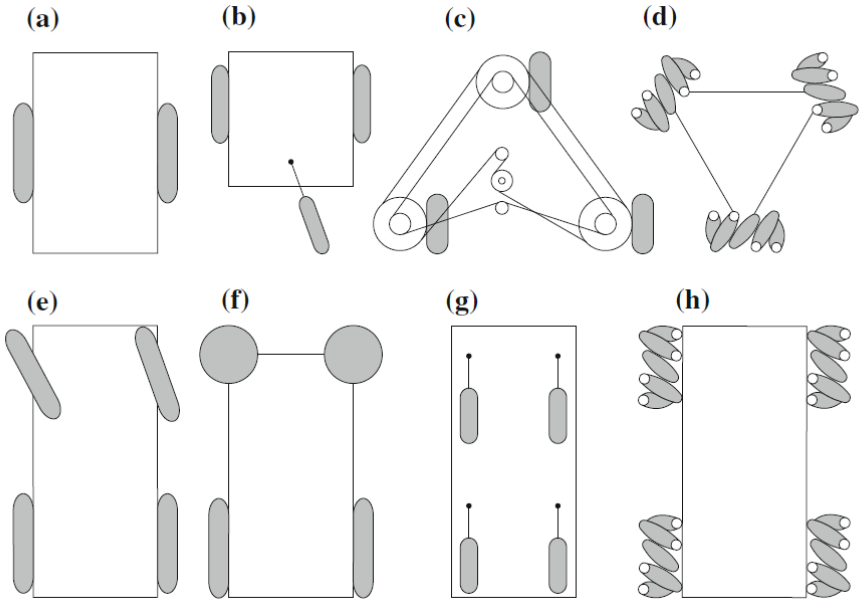


Slika 3. Dizajn kotača: a) standardni fiksni kotač, b) standardni rotirajući kotač, c) kutni rotirajući kotač, d) sferni kotač, e) švedski kotač

Fiksni kotač, standardni rotirajući kotač i kutni rotirajući kotač imaju primarnu os rotacije koje je određena usmjerenjem kotača. Gibanje u drugom smjeru nije moguće bez da se prvo kotač rotira oko vertikalne osi. Sferni kotač omogućuje gibanje u više smjerova bez da se prvo okrene oko osi rotacije. Švedski kotač pokušava postići višesmjerno gibanje sa pasivnim valjcima koji se nalaze na opsegu kotača stoga se on može gibati po različitim trajektorijama, kako naprijed tako i nazad.

Odabir vrste kotača, broja kotača i njihovo mjesto pričvršćivanja na konstrukciju robota značajno utječe na kinematski model mobilnog robota. Minimalni broj kotača koji se mogu staviti na robota je dva, dok maksimalan broj se preporučuje 4 jer povećavanjem broja kotača, povećava se i složenost kinematskog modela, a time i načina upravljanja mobilnim robotom. Različite konfiguracije mobilnih robota prikazani su na slici 4. Slovom a) označen je robot s dva fiksna kotača, slovom b) mobilni robot s dva fiksna kotača i jednim kutnim rotirajućim kotačem, slovom c) tri kotača koji su sinkronizirani s mogućnosti upravljivosti, slovom d) je primjer mobilnog robota s tri kotača u obliku trokuta koji su po tipu kotača švedskog tipa, slovom e) četiri kotača gdje su prednja dva upravljiva što je klasični dizajn kod automobila, slovom f) četiri kotača gdje su prednji kotači sferni, a stražnji klasični fiksni, slovom g) četiri

pogonjena kutna rotirajuća kotača i slovom h) četiri višesmjerna kotača koji su svi švedskog tipa. [5]



Slika 4. Različite konfiguracije mobilnih robota [5]

2.5. Kinematski model mobilnog robota

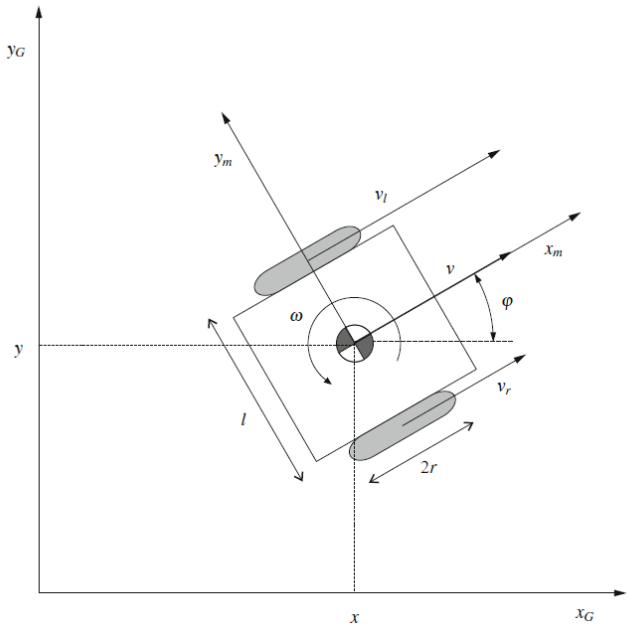
Pretpostavke za olakšanje izrade kinematskog modela rade se kako bi se pojednostavila definicija modela mobilnog robota, a to su:

- Mobilni robot promatramo kao kruto tijelo na kotačima
- Može se gibati samo u horizontalnoj ravnini
- Podloga je uvijek ravna i ravnomjerno raspoređena
- Kotači nikad ne proklizuju
- Kotači se uvijek okreću kada za to dobiju naredbu

S ovim pretpostavkama pozicija robota se može definirati s tri koordinate u prostoru. Prve dvije predstavljaju poziciju u horizontalnoj ravnini (x i y koordinata), a treća opisuje orijentaciju oko vertikalne osi (rotacija oko z osi). Jednostavni kinematski model mobilnog robota s diferencijalnim pogonom prikazuje Slika 5. gdje su globalne osi označen sa x_G i y_G, dok su lokalne koordinatne osi označene slovima sa x_M i y_M. Koordinatna os x_M pokazuje u smjeru gibanja robota prema naprijed. Pozicija i orijentacija robota je definirana sljedećim vektorom

$$x = \begin{bmatrix} x \\ y \\ \theta \end{bmatrix}, \tag{1}$$

gdje *x* i *y* definiraju poziciju robota relativnu na globalni koordinatni sustav te kut θ definira orijentaciju odnosno zakret oko vertikalne osi, osi *z*. Robot s diferencijalnim pogonom prikazan na slici 5. ima jednostavnu mehaničku strukturu. Njegovo gibanje je definirano s dva pogonska kotača pričvršćenih sa svake strane tijela robota. Robot mijenja smjer tako da smanjuje ili povećava brzinu rotacije pojedinog kotača i za to mu nije potrebno dodatno upravljanje vezano uz promjenu smjera kretanja. Ako se motori okreću u istom smjeru s jednakom brzinom, robot će ići prema naprijed ili prema nazad i gibat će se pravocrtno. Ako se pak motori okreću u suprotnom smjeru pri istoj brzini, robot će se rotirati oko središnjeg centra rotacije i doći će do okretanja robota u mjestu. Općenito gledano, centar rotacije robota može biti bilo gdje na pravcu koji prolazi kroz dva kotača odnosno njihove koordinatne osi i ovisit će o tome kako se okreću kotači i pri kojoj brzini se okreću.



Slika 5. Kinematski model mobilnog robota s diferencijalnim pogonom [5]

Ovakva jednostavna kinematika prikladna je za proučavanje idealnog modela gibanja robota. Koristeći širinu robota, koja je označena slovom l na slici 5, izražena je udaljenost dodira kotača robota s podlogom te radijusom kotača robota označena slovom r. Sada se lako može analizirati gibanje robota. Kotači se okreću s kutnom brzinom koju označavamo slovom ω. Svaki kotač ima svoju kutnu brzinu gdje desni kotač ima oznaku ω_r, a lijevi kotač oznaku

ω_l. Tako se dobivaju brzine na svakom kotaču označeni vektorima v_r na desnom kotaču i v_l vektor brzine lijevog kotača. Iz toga možemo izvesti jednadžbe za brzine kotača

$$v_r = r\omega_r \,, \tag{2}$$

$$v_l = r\omega_l \,, \tag{3}$$

Dvije istovremene rotacije kotača rezultiraju translacijskom brzinom robota duž osi x_M i kutnom brzinom oko okomite osi *z*. Iz Slike 5. može se tako izvesti da je kutna brzina jednaka

$$\omega = \frac{v_r - v_l}{l}, \tag{4}$$

Translacijska brzina duž osi x_M onda slijedi iz jednadžbe

$$v = \frac{v_r + v_l}{2}, \tag{5}$$

Jednadžbe (4) i (5) definiraju relaciju između kutne brzine i brzine robota. Međutim, iz perspektive upravljanja robotom pogodnija je obrnuta relacija koja definira kutnu brzinu robota i željene brzine robota. Kada se povežu jednadžbe (4) i (5) dobije se sljedeće

$$\omega_r = \frac{2v + \omega l}{2r}, \tag{6}$$

$$\omega_l = \frac{2v - \omega l}{2r}, \tag{7}$$

Orijentacija robota može se opisati matricom rotacije R

$$R = \begin{bmatrix} \cos\theta & -sin\,\theta & 0 \\ sin\,\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}, \tag{8}$$

Brzina robota determinirana kao par v i ω je definirana relativno na lokalni koordinatni sustav mobilnog robota x_M i y_M. Brzina robota u globalnom koordinatnom sustavu x_G i y_G definirana je kao derivacija pozicije robota vektora x i može se izračunati tako da se rotira lokalno definirana brzina na koju je primijenjena matrica rotacije R kao translacijski dio

$$\begin{bmatrix} \cos\theta & -sin\,\theta & 0 \\ sin\,\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} v \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} v\cos\theta \\ v\,sin\,\theta \\ 0 \end{bmatrix}, \tag{9}$$

te rotacijski dio

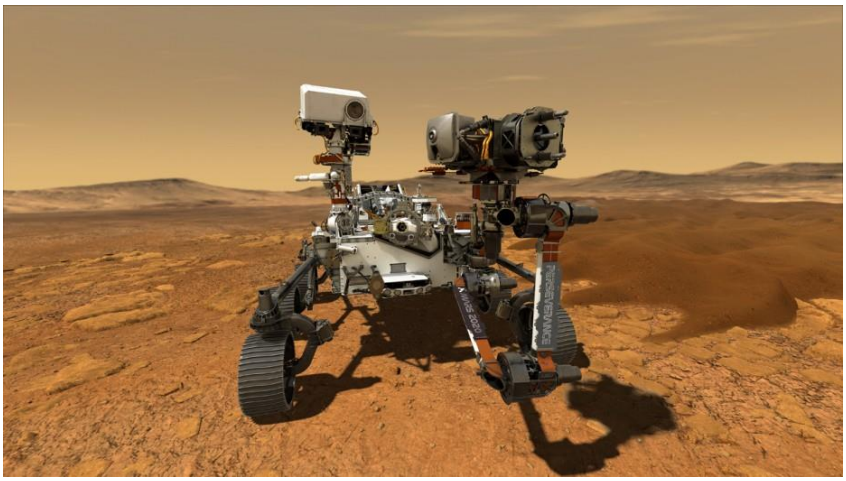
$$\begin{bmatrix} \cos\theta & -sin\,\theta & 0 \\ sin\,\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ \omega \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ \omega \end{bmatrix}, \tag{10}$$

Kada se spoje rotacijski i translacijski dio gornjih jednadžbi te se izostave elementi koji su nula, brzina robota u globalnom koordinatnom sustavu se dobiva kao

$$\dot{x} = \begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \end{bmatrix} = \begin{bmatrix} v\cos\theta \\ v\,sin\,\theta \\ \omega \end{bmatrix}, \tag{11}$$

2.6. Primjena mobilnih robota

Sukladno gore navedenome, mobilni roboti su vozila koja se mogu kretati po kopnu, zraku, u ili na vodi. Najčešće se susrećemo s vrstom mobilnih robota koji se kreću po kopnu. S obzirom na vrlo veliku zastupljenost u svakodnevnoj upotrebi, takve mobilne robote susrećemo u blizini ljudi kao što su stanovi, klinički i bolnički centri, dućani ili muzeji, no sve su više prisutni i u obliku autonomnih vozila koja su se sve počela pojavljivati na cestama i autocestama. Mobilne robote kao vozila koristimo i u obliku autonomnih kućnih usisavača, samostalnih kosilica za travu, vodiča kroz skladišta ili dućane pri čemu surađuju s djelatnicima ili radnicima u proizvodnji. Također, takvi roboti se koriste i pri provođenju svemirskih misija. Najveći dio mobilnih robota pogonjenih kotačima ima stabilnost osiguranu s minimalno tri kotača. U tijeku je i razvoj mobilnih robota u obliku automobila koji imaju određenu autonomiju. Iako se radi još o razvojnoj fazi, vidljivi su početni rezultati i potencijal za daljnju razradu i razvoj. Osim toga, počeo je i razvoj robota koji nalikuju životinjama i pokušavaju imitirati kretanje životinja, na primjer psa ili mačke. Postoje i razne manje verzije mobilnih robota koje studentima pomažu pri učenju i omogućuju sastavljanje vlastitog mobilnog robota za kojeg student može razviti sustav upravljanja.



Slika 6. Mobilni robot tvrtke NASA za misiju na Marsu[6]

Slika 7. Poslužni mobilni robot tvrtke Savioke Relay [7]

Slika 8. Robot koji imitira psa tvrtke Boston Dynamics [8]

Najzastupljenija vrsta mobilnih robota koji se mogu kretati po zraku, odnosno imaju mogućnost letenja su dronovi. Ova vrsta robota je relativno jeftina jer ima jednostavnu mehaničku strukturu. Dronovi koriste četiri motora i opremljeni su senzorima koji se koriste i kod kopnenih mobilnih robota kao što su akcelerometar, žiroskop, kamera. Na slici 9. je prikazan jedan jednostavni dron.

Slika 9. Dron s kamerom za snimanje

Roboti koji se kreću u vodi najčešće se koriste kao neka vrsta podmornica s udaljenim upravljanjem što omogućuje istraživanje područja koja su za čovjeka preopasna i nedohvatljiva, kao što su istraživanja oceana, dna mora ili različitih brodskih olupina u morima. U većini slučajeva takvi roboti imaju ugrađenu robotsku ruku za prihvat i dohvaćanje dijelova ili predmeta koji se pronađu. Autonomni roboti koji plutaju na površini vode zasad se koriste u ekološke svrhe za skupljanje nečistoća iz mora ili oceana.

Slika 10. Podvodno autonomno vozilo u obliku podmornice tvrtke Bluefin Sandshark [9]

3. PIONEER MOBILNI ROBOT

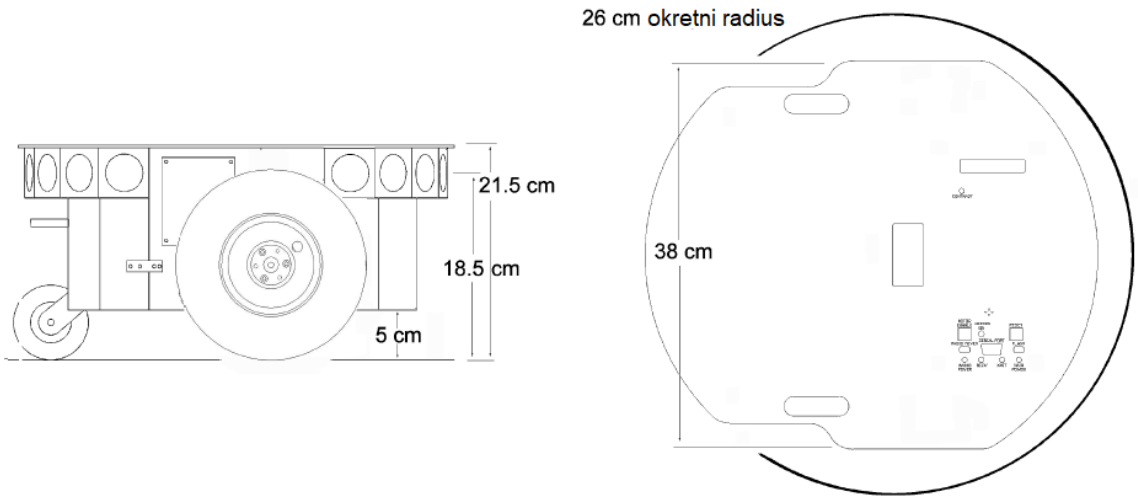
Mobilni robot korišten u radu je Pioneer 2-DX koji je proizveden u tvrtki ActivMedia Robotics. Pioneer je grupni naziv za mobilne robote tvrtke ActivMedia koji imaju dva ili četiri pogonska kotača. Svi roboti se sastoje od server-klijent arhitekture, koju je razvio profesor Kurt Konolige sa sveučilišta Stanford, te su razvijeni u nekoliko verzija od kojih je ovo druga verzija proizvedena 1999. godine. Mobilni roboti proizvedeni su tako da se u budućnosti mogu zamijeniti komponente kako bi se mogao poboljšati sustav što će se pobliže objasniti nakon opisa osnovnih komponenti od kojih je izgrađen mobilni robot. Na slici 11. može se vidjeti Pioneer 2-DX mobilni robot.

Slika 11. Pioneer 2-DX mobilni robot

3.1. Hardverske komponente

Roboti tvrtke ActivMedia su inteligentni, skladno ukomponirani mobilni roboti koji sadrže sve osnovne komponente potrebne za dobivanje informacija iz okoline i navigaciju u realnom svijetu, uključujući bateriju, upravljive motore i kotače, enkodere za pozicioniranje i mjerenje brzine te integrirane senzore. Upravljanje pojedinim komponentama sustava izvodi se preko mikrokontrolera i softvera koji je napravljen u obliku serverske aplikacije. Svaki robot sastoji se i od adresiranih ulazno izlaznih pinova za do šesnaest dodatnih uređaja, dva RS-232 serijska priključka, osam ulazno izlaznih priključka, pet analogno digitalnih pretvornika, kojima se može pristupiti kroz aplikacijsko sučelje na robotski server softver koji se zove P20S (*eng. Pioneer 2 Operating System*). Težina robota iznosi devet kilograma, a maksimalna dodatna nosivost robota je dvadeset kilograma. [10]

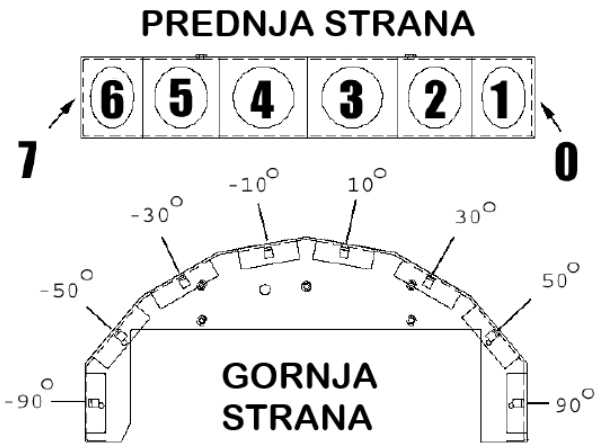
Glavne komponente Pioneer 2-DX mobilnog robota su kućište, tijelo i nos u kojem se nalazi osam sonarnih senzora koji omogućuju detektiranje objekta i navigaciju. Robot je opremljen s dva istosmjerna povratna motora s dva optička enkodera za mjerenje pozicije i brzine vrtnje robota motora. Robot je opremljen i s vlastitim računalom, diskom za pohranu, priključkom za internet te linux operativnim sustavom.



Slika 12. Dimenzije mobilnog robota Pioneer 2-DX

3.1.1. Sonarni senzori

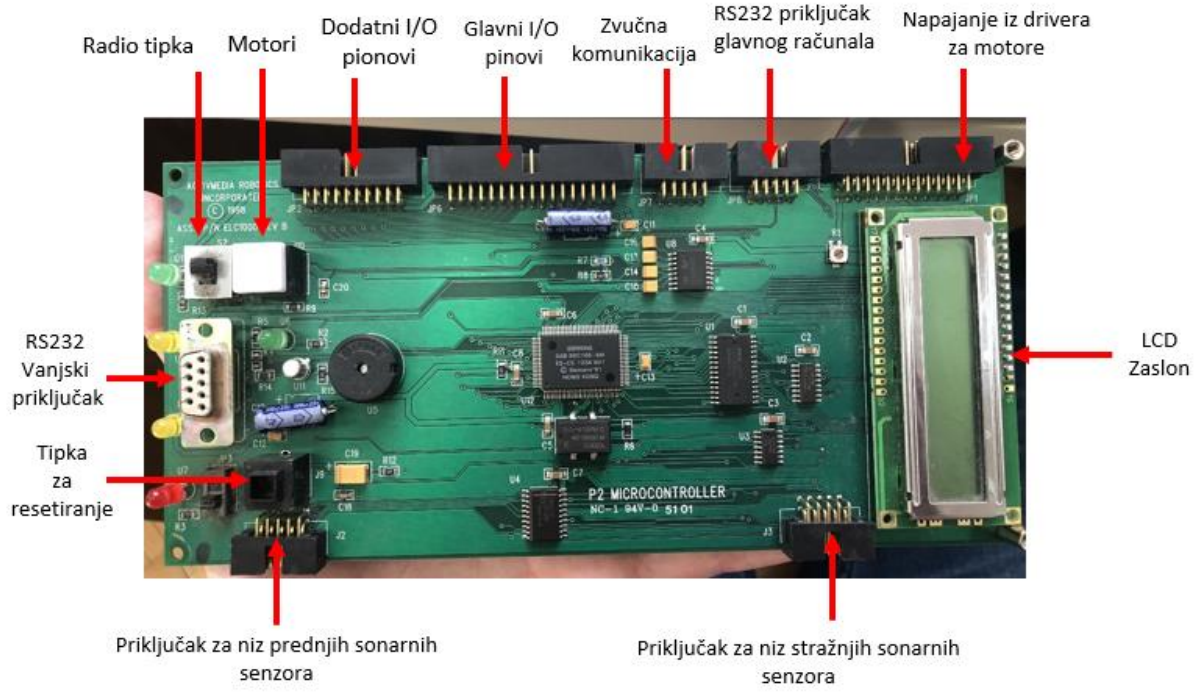
Mobilni roboti tvrtke ActivMedia Robotics opremljeni su sonarnim senzorima. Pioneer 2-DX sadrži osam senzora koji se nalaze na prednjoj strani. Pozicija sonarnih senzora je fiksirana tako da se po jedan senzor nalazi sa svake strane, označeni brojevima nula i sedam na slici 13, te šest senzora koji se nalaze raspoređeni na nosu robota u intervalu razmaka od 20º, koji su na slici 13. označeni brojevima od jedan do šest. Sonarni senzori dolaze s vlastitom elektronikom za neovisno upravljanje. Frekvencija primanja informacija je 25Hz, što znači da svakih 40 milisekundi jedan senzor pošalje i primi informaciju. Osjetljivost senzora je od deset centimetara pa čak sve do skoro pet metara. Osjetljivost senzora lako se može podešavati pomicanjem potenciometra koji se nalazi na pločici za upravljanje senzorima. Senzori se mogu podesiti tako da se smanji mogućnost detektiranja malih objekta, što je u određenim uvjetima poželjno. Ako je puno šumova i buke u okolini ili se od poda događa nejednaka refleksija, senzori će biti preosjetljivi i neće raditi kako treba. Povećavanjem osjetljivosti senzora omogućuje se da prepoznaju male objekte ili objekte na većoj udaljenosti. Takvu razinu osjetljivosti možemo primijeniti kada radimo u relativno tihom i mirnom okolišu.



Slika 13. Raspored niza sonarnih senzora

3.1.2. Mikrokontroler

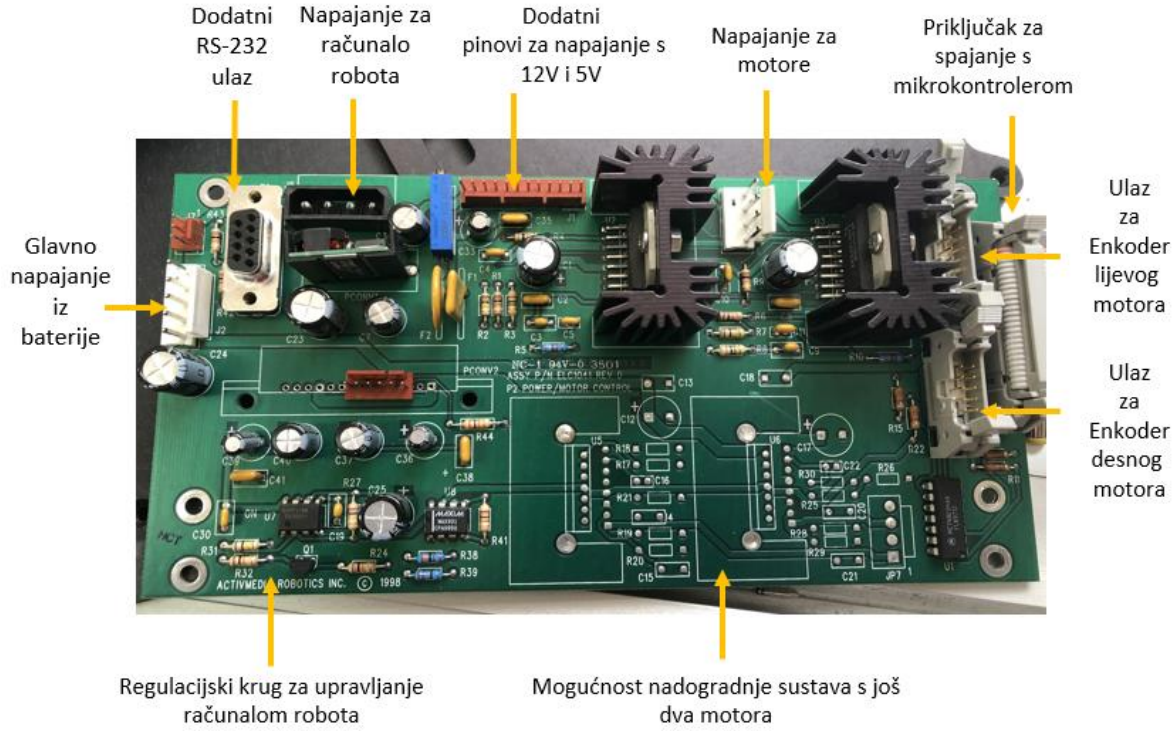
Glavna upravljačka jedinica robota je mikrokontroler sa Siemens 88C166 mikroprocesorom s integriranom memorijom od 32K Flash-ROM te dinamičkim RAM-om od 32K. Informacije o stanju i povezivosti robota prikazuju se na LCD zaslonu koji ima mogućnost prikazivanja do 32 znaka u dvije linije. Zaslon prikazuje stanje komunikacije s računalom kao i trenutni napon baterije. Serijska komunikacija između mikrokontrolera i klijenta se ostvaruje preko RS-232 priključka. Na slici 13. označeni su dijelovi mikrokontrolera.



Slika 14. Mikrokontroler Pioneer 2 mobilnog robota

3.1.3. Driver za motore

Unutar robota, na postolju za baterije, nalazi se upravljačka pločica za motore i napajanje cijelog robota. Direktno iz baterije dolazi napajanje na pločicu s driverima za motore koje se onda regulatorima na pločici regulira u potrebne vrijednosti za pojedine komponente na robotu. Iz pločice izlazi istosmjerna struja s mogućim vrijednostima napona od 12V i 5V. Standardni driver za motore ima dvanaest pinski konektor koji podržava set od četiri priključka za 5V te četiri priključka po 12V gdje je maksimalna struja 1.5A. Ovi priključci se mogu koristiti za razne dodatne komponente za koje je potrebno napajanje te da ga oni dobiju direktno iz robota. Pokraj dodatnog RS232 priključka nalazi se regulator koji napaja integrirano računalo robota. Uključuje specijalni regulacijsku krug koji omogućava spuštanje napona uz konstantnu veću struju koja je potrebna za napajanje računala. Na slici 14. je prikazana pločica s driverima za motore koja regulira napajanje svih komponenti mobilnog robota.



Slika 15. Pločica s napajanjem i driverima za motore

3.1.4. Motori i enkoderi

Pioneer 2-DX mobilni robot sadrži dva istosmjerna motora velike brzine i momenta koji se mogu rotirati u oba dva smjera. Svaki motor je opremljen s dva dvokanalna inkrementalna optička enkodera za mjerenje pozicije i brzine vrtnje robota motora. Podaci o motorima, kotačima i enkoderima prikazani su u tablici 3.1.

Tablica 3.1 Tehničke karakteristike motora robota

Broj kotača	2
Vrsta kotača	ispunjena guma
Dijametar kotača [mm]	165
Širina kotača [mm]	37
Širina pomoćnog kotača [mm]	75
Prijenosni omjer	19.7:1
Maksimalna translacijska brzina [mm/s]	1600
Maksimalna rotacijska brzina [º/s]	300
Broj enkodera	2

3.2. Softverske komponente

Svaki robot tvrtke ActivMedia opremljen je softverskim paketom po principu komunikacije server klijent. Robot je server u ovoj komunikaciji i on se bavi svim detaljima na niskoj razini uključujući održavanje brzine robota, smjera na neravnom terenu, dohvaćanje podataka iz sonarnih senzora te omogućuje upravljanje različitim dodacima kao što je hvataljka. Kako bi se ostvarila arhitektura klijent server, Pioneer roboti zahtijevaju povezivanje s klijentom. Klijent može biti računalo koje dolazi s robotom ili vanjsko računalo kao što je laptop. Softver koji se pokreće na računalu, koje je povezano s robotom serijski direktno na mikrokontroler robota, omogućuje kontrolu više razine, inteligentno robotsko upravljanje te izbjegavanje objekata. Bitna prednost ovakve arhitekture je da se različiti serveri robota mogu upravljati koristeći samo jednog klijenta koji radi upravljanje na višoj razini. Dakle, ako imamo više robota tvrtke ActivMedia, može se složiti upravljanje svih njih koristeći samo jedno vanjsko računalo koje je klijent.

3.2.1. P2OS

Robotski server za upravljanje je program koji se pokreće u mikrokontroleru robota i zove se Pioneer 2 Operacijski Sustav (eng. *Pioneer 2 Operating System – P2OS*). On je zadužen za svu kontrolu sistema mobilnog robota na najnižoj razini. To se odnosi na upravljanje motorima, prikupljanje podataka sa sonarnih senzora i enkodera motora. Naredbe za prikupljanje tih podataka te za primanje podataka dobivaju se iz neke udaljene klijent aplikacije kao što je ARIA. Uz ovakvu klijent server arhitekturu, razvojni robotski programeri ne trebaju

znati sve detalje o serveru na pojedinom robotu jer se za upravljanje na nižoj razini brine server robota smješten unutar mikrokontrolera robota. [11]

3.2.2. ARIA

Sučelje za aplikacije tvrtke ActivMedia Robotics (*eng. ActivMedia Robotics Interface for Applications* – ARIA) je okruženje za razvoj programa za upravljanje ActivMedia robotima. Programska biblioteka ARIA napisana je u programskom jeziku C++ i omogućuje robusnu klijent server komunikaciju s mikrokontrolerom robota. ARIA s klijent strane je zadužena za upravljanje robotom na višoj razini jer komunikaciju i upravljanje na nižoj razini obavlja P2OS. To znači da je njena uloga da omogući nesmetanu komunikaciju između klijenta i servera pomoću klijent softvera koji je baziran na ARIA sučelju i robotskih servera koji se nalaze u sklopu kontrolera robota. Klasa *ArRobot* je središnji dio sustava ARIA koji se ponaša kao vrata komunikacije između klijenta i servera i zadužen je za prikupljanje podataka kao što su pozicija u prostoru u obliku koordinata x i y, brzina robota (translacija i rotacija) te smjer gibanja robota. Klasa *ArSerialConnection* zadužena je da otvori serijski priključak te omogući komunikaciju s robotom. ARIA je sastavljena od velikog broja ovakvih klasa koje se koriste da bi se upravljanje moglo izvoditi na višoj razini. [10]

Jedan od glavnih zadataka svih robotskih aplikacija je da omoguće klijent server komunikaciju između softvera koji se nalazi u sklopu ARIA-e te servera i uređaja koji se nalaze na robotu.

3.2.3. MobileSim

Mobile sim je softver za simulaciju mobilnih robota i njihovog okruženja. Koristi se za otklanjanje grešaka i testiranje pomoću ARIE. Razvijen je unutar Stage simulatora kojeg je napravio Richard Vaughan, Andrew Howard i ostali kao dio Player/Stage projekta kojeg su modificirali programeri iz tvrtke MobileRobots. MobileSim pretvara ActivMedia mapu u Stage okuženje te stvara simulacijski model robota u to okruženje. Nakon toga ostvaruje se upravljano povezivanje sa simulacijskim modelom preko TCP priključka, slično kako bi se povezali i u stvarnosti na pravog robota. Prednost ARIA-e je da se može na ovaj način povezivati sa simulacijskim modelom umjesto da se povezuje s robotom preko serijskog priključka te se tako može testirati kretanje robota u virtualnom okruženju.

4. ROBOTSKI OPERATIVNI SUSTAV

Robotski operativni sustav (ROS) fleksibilan je okvir za izradu robotskog softvera. To je zbirka alata, knjižnica i konvencija kojima je cilj pojednostaviti zadatak stvaranja složenih i robusnih robotskih ponašanja kroz široki raspon robotskih platformi. Zašto? Jer je teško stvoriti univerzalan robotski softver. Iz perspektive robota, problemi koji čovjeku na početku izgledaju jednostavno često se znaju lako zakomplicirati ovisno o zadataku i okruženju. Suočavanje s tim problemima je toliko teško da se niti jedan pojedinac, laboratorij ili institucija ne mogu nadati da će to učiniti sami. Kao rezultat, ROS je izgrađen od temelja kako bi potaknuo zajednički razvoj softvera za robotiku. Na primjer, jedan laboratorij možda je bio stručnjak za izradu navigacijskih karata u zatvorenim prostorima i mogao bi pridonijeti sustavu svjetske klase za izradu karata. Druga skupina ljudi su stručnjaci za navigaciju, a možda neka treća skupina je razvila metodu korištenja računalne vizije koja funkcionira za prepoznavanje malih objekta u prostoru. Stoga je ROS dizajniran posebno za grupe koje vole surađivati i nadograđivati se na međusobnom radu. [12]

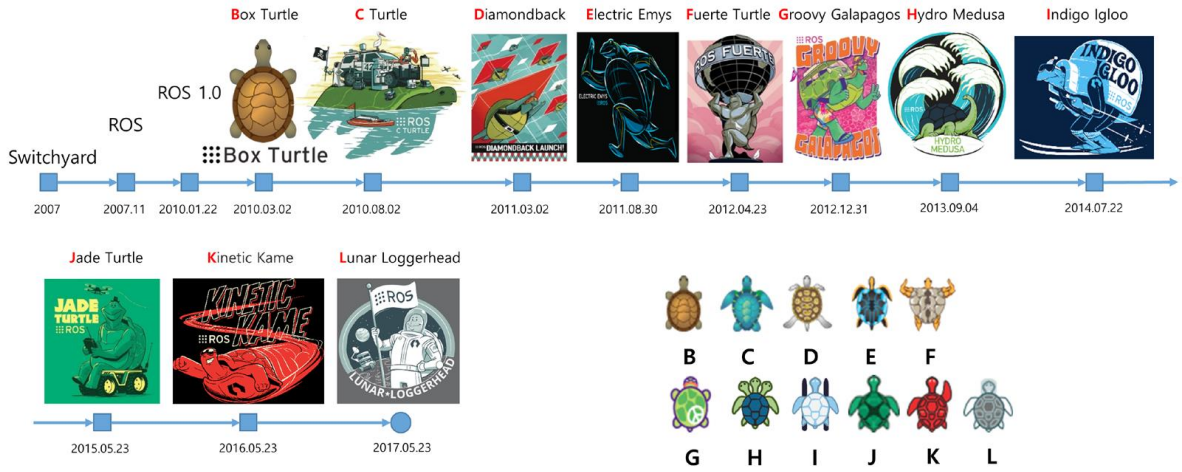
Drugim riječima, ROS uključuje sloj apstrakcije hardvera slično operativnim sustavima. Međutim, za razliku od konvencionalnih operativnih sustava, može se koristiti za brojne kombinacije hardverske implementacije. Nadalje, to je robotski softver koji pruža različita razvojna okruženja specijalizirana za razvoj robotskih programa.

4.1. Povijest ROS-a

ROS je veliki projekt na kojem je sudjelovalo više ljudi i suradnika iz područja robotike. Ljudi iz robotske istraživačke zajednice osjetili su potrebu za razvojem sustava koji će biti otvoren za sve ljude te će se lako moći koristiti. Stududent sveučilišta sa Stanforda Morgan Quigley pokrenuo je projekt razvoja robotskog sustava koji je otvoren za sve u svibnju 2007. godine pod imenom „Switchyard“ kao podrška projektu STAIR (*eng. Stanford Artificial Intelligence Robot*). U studenome 2007. godine, američka robotska tvrtka Willow Garage osigurala je značajna sredstva za širenje ovih koncepata softvera da postane dostupan svima. Od početka, ROS je razvijan na više različitih institucija i za različite robote. Prvo je izgledalo kao nešto što neće uspjeti jer bi bilo jednostavnije da su svi stavljali svoje dijelove programa na jedan server. Ironično, kroz godine, ovo se ispostavilo kao jedna od glavnih snaga ROS sustava jer svaka skupina može napraviti vlastiti ROS repozitorij koda na svojim serverima i zadržat će potpunu kontrolu i vlasništvo nad njim. [13]

4.2. Verzije ROS-a

Prva verzija ROS sustava izašla je u prosincu 2008. godine imenom ROS 0.4 Mango Tango. Ubrzo nakon toga izlazi verzija ROS 1. koja sadrži dijelove softverskog paketa koji se koriste i u današnjim verzijama uz robot imenom Box Turtle. Box Turtle je prva službena distribucija ROS softvera sa setom softverskih paketa za javnu upotrebu. Druga takva distribucija ROS sustava izašla je 2010. godine pod nazivom ROS C Turtle. Do godine 2013. izlaze su još četiri verzije, a to su Diamondback, Electric Emys, Fuerte Turtle te Groovy Galapagos. Sa šestom verzijom službene verzije 'ROS Groovy Galapagos', Willow Garage pokušao je prodrijeti na tržište komercijalnih servisnih robota u 2013. godini, no na kraju su se podijelili u nekoliko start-up tvrtki i ujedinili sa zakladom OSRF(*eng. Open Source Robotics Foundation*). Od tada je objavljeno još pet verzija, Hydro Medusa, Indigo Igloo, Jade Turtle, Kinetic Kame, Lunar Loggerhead. U svibnju 2017. godine OSFR mijenja ime u Open Robotics i razvija, vodi i vrši kontrolu nad ROS distribucijama. Posljednja, 12. distribucija ROS sustava zove se ROS Melodic Morenia. ROS označava prvo slovo svakog imena distribucije abecednim redom i koristi kornjaču kao njihov simbol kao što se vidi na slici 16. [13]



Slika 16. Vremenska crta izlaska ROS distribucija [13]

Imenovanje ROS verzija radi se abecednim redom, isto kao distribuirane verzije Ubuntu i Android sustava. Na primjer, distribucija Kinetic Kane je 11. verzija čime započinjemo sa slovom K i verzijom za 10. službeno izdanje. Osim toga, postoji još jedno pravilo. Svaka verzija ima ilustraciju u obliku plakata i ikona kornjače, kao što je prikazano na slici 16. Ikone kornjača koriste u službenoj ROS simulaciji za mobilne robote koja se zove Turtlesim.

4.3. ROS arhitektura

Arhitektura ROS-a podijeljena je na tri konceptualne razine:

1. Datotečni sustav
2. Razvojna razina
3. Razina zajednice

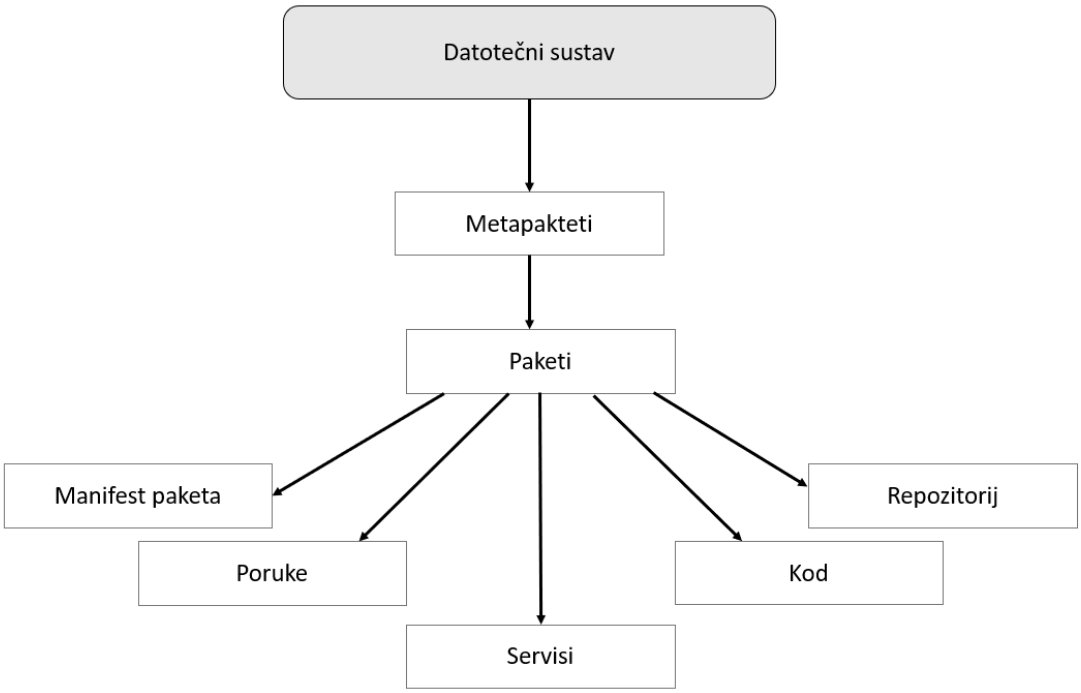
Prva razina je datotečni sustav. Na ovoj razini nalazi se struktura datoteka te minimalan broj datoteka koje su potrebne za normalnu funkcionalnost sustava. Struktura datoteka predstavlja skupinu koncepta koje se koriste kako bi se objasnila funkcionalnost ROS-a.

Druga razina je razvojna razina gdje se odvija komunikacija između procesa i sistema. Na ovoj razini su smješteni koncepti i mehanizmi koje ROS koristi za upravljanje sustavom, način na koji se pokreću ili isključuju procesi te način komunikacije sustava s drugim sustavima.

Zadnja razina je zajednica koju čini set alata i koncepta kroz koje se širi znanje, algoritmi te programski kod između razvojnih programera. Ova razina je od velikog značaja jer je većina projekta otvorena za zajednicu na kojima svi mogu raditi. Postojanje jake zajednice poboljšava i olakšava novim članovima brzo razumijevanje osnovnih koncepata ROS sustava te otklanjanje čestih pogrešaka s kojima su se neki do sada već susreli. Ovakav način zajednice doveo je do brzog prihvaćanja ROS-a kod velikog broja ljudi koji se bave robotikom. [14]

4.3.1. ROS datotečni sustav

ROS datotečni sustav (*eng. Filesystem level*) je skup koncepta od kojih je izgrađen ROS. Glavni mu je cilj da se centralizira proces izgradnje i podizanja projekta, ali i da istovremeno pruži dovoljno fleksibilnosti i alata za decentralizaciju njegovih podsustava. Korisniku koji se prvi puta susretne s ROS-om ovo će možda izgledati jako komplicirano, no rad u ROS-u omogućava i olakšava brže shvaćanje osnovnih koncepata ove razine. Na slici 17. je prikazan strukturalni koncept datotečnog sustava.



Slika 17. Strukturalni prikaz datotečnog sustava

Metapaketi (*eng. Meta packages*) je skupni naziv za grupu paketa s posebnom namjenom. U starijim verzijama kao što su Electric i Fuerte, zvali su se „Stacks“, ali u današnjim verzijama su radi jednostavnosti datotečnog sustava izbačeni. Jedan primjer metapaketa je ROS navigacijski sustav.

Paketi (*eng. Package*) su osnovna jedinica ROS softvera. U njima se nalaze ROS procesi koji se pokreću prilikom pokretanja ROS-a, a koje zovemo ROS čvorovi (*eng. Nodes*). Tu se nalaze još i biblioteke, konfiguracijski podaci te brojni drugi koji su organizirani kao jedinstvena cjelina. Paketi su najviša razina datotečnog sustava u kojima se izdaju dijelovi ROS sustava.

Manifest paketa (*eng. Package manifest*) je vrsta konfiguracijske datoteke unutar paketa koji sadrži sve informacije o paketu, kao što je autor, licenca, funkcijske zavisnosti, kompajlerske podatke. Datoteka *package.xml* je manifest paketa, u kojem se nalaze svi podaci važni uz izgradnju softvera u ROS-u. Njih možemo sami kasnije modificirati ovisno o tome što nam treba pri izgradnji ROS softvera.

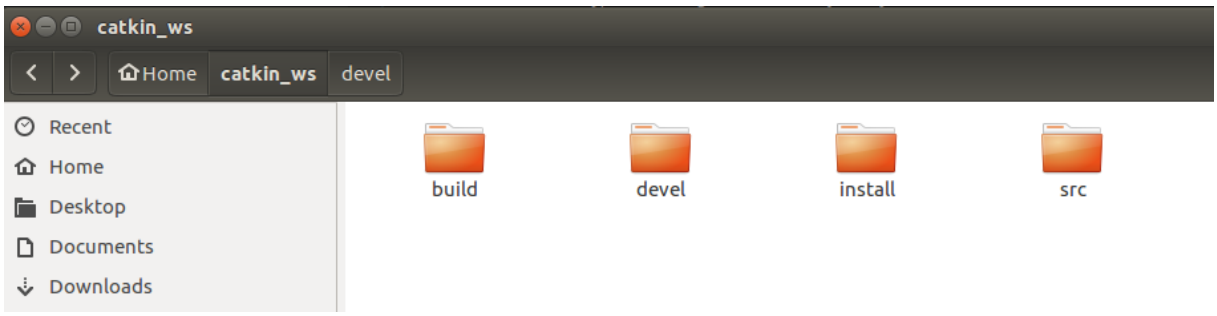
Poruke (*eng. Messages*) su vrste datoteka koje su u formatu *.msg* i predstavljaju tip informacije koja se šalje iz jednog ROS procesa u drugi. Mogu se definirati vlastite poruke unutar datoteke *msg* unutar paketa (npr. *moj_paket/msg/ MojTipPoruke.msg*).

Servisi (*eng. Services*) su dio paketa koji se odnosi na interakciju između dijelova koji rade na principu zatraži i odgovori. Podaci o tome što će se slati i prihvatiti nalaze se unutar *srv* datoteke i definiraju se na način (npr. *moj_paket/srv/MojTipServisa.srv*).

Repozitorij (*eng. Repository*) se odnosi na dio kojim se održavaju verzije paketa. Danas postoje različiti softveri za verzioniranje od kojih je najpoznatiji Git. Skup paketa koji dijele isti način verzioniranja mogu se nazvati repozitorijem.

4.3.1.1. ROS Radni prostor

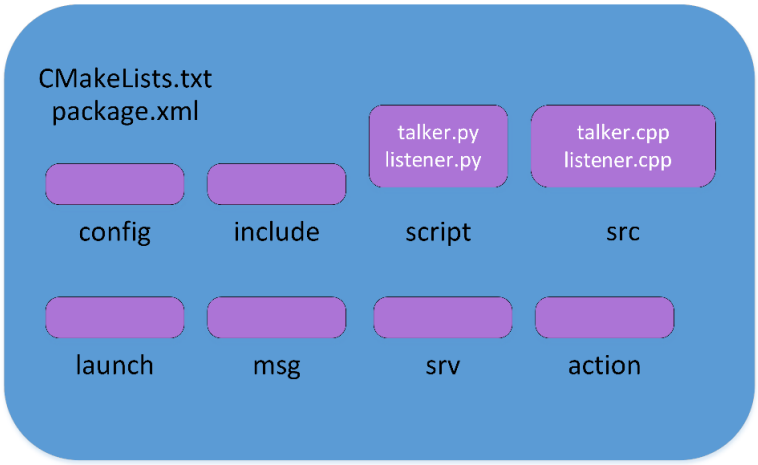
Svaka verzija ROS softvera mora se inicijalizirati unutar jedne vlastite datoteke. Generalno, radni prostor je datoteka koja sadrži pakete gdje su smješteni svi dijelovi opisani u datotečnom sustavu. Unutar radnog prostora definirani su načini kompajliranja i izgradnje ROS softvera. Koristi se kao centralna jedinica razvoja ROSa te sadrži mogućnost kompajliranja više različitih paketa u isto vrijeme. ROS radni prostor nalazi se unutar datoteke koja se najčešće imenuje imenom *catkin_ws* kao što je prikazano na slici 18. gdje se glavni dio razvojnog koda nalazi u datoteci *src*. Unutar radnog prostora stavljamo sve projekte, pakete i ostale dijelove vezane uz razvoj softvera. Datoteka *build* sadrži sve informacije o posrednim datotekama za naše projekte i pakete, dok se unutar *devel* datoteke nalaze naše osobne preference koje koristimo pri pisanju programa. U *devel* datoteci smješten je i naš program nakon napravljenog kompajliranja. *Devel* datoteka se koristi za različite vrste testiranja koji se kasnije mogu premjestiti u *install* datoteku kako bi se podijelio kreirani program s ostalim korisnicima.



Slika 18. Tipični ROS radni prostor

4.3.1.2. ROS paketi

ROS paketi su osnovna jedinica ROS sustava koji predstavljaju procese u ROS-u i ovdje ćemo prikazati njihovu strukturu. Klasična struktura ROS paketa prikazana je na slici 19.



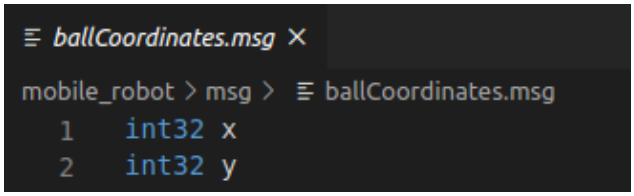
Slika 19. Struktura klasičnog ROS paketa[14]

Struktura svakog ROS paketa objašnjena je kroz organizaciju datoteka na sljedeći način[14]:

- *config*: datoteka je mjesto gdje se nalaze svi konfiguracijski podaci korišteni u ROS paketu. Korisnik kreira sam ovu datoteku i uobičajena ime za nju je *config* kako bi znali drugi korisnici koji će možda koristiti ovaj paket što se unutra nalazi
- *include*: mjesto za biblioteke i podatke koji se koriste unutar paketa
- *scripts*: sadrži sve Python skripte koje su napisane za pojedini paket. Na slici 19. vidimo dvije skripte kao primjer, *talker.py* i *listener.py*
- *src*: datoteka gdje je spremljen sav kod napisan u programskom jeziku C++. Također, na slici 19. vidimo dva primjera koda, *talker.cpp* i *listener.cpp*
- *launch*: podaci za pokretanje odjednom više različitih čvorova
- *msg*: poruke koje korisnik kreira i definira spremaju se u ovu datoteku
- *srv*: svi servisi koji se odnose na definirani paket
- *action*: definicije akcija koje paket koristi
- *package.xml*: manifest paketa koji predstavlja konfiguracijsku datoteku
- *CMakeLists.txt*: tekstualna datoteka koja koristi CMake način izgradnje programa i sastavni je dio svakog ROS paketa

4.3.1.3. ROS poruke

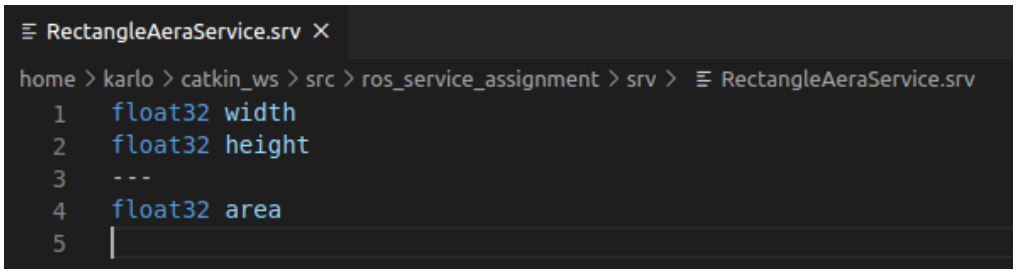
ROS koristi pojednostavljeni tip opisnog jezika za razmjenu poruka kako bi opisao podatkovne vrijednosti koje ROS čvorovi objavljuju. Ovakav način pisanja poruka omogućuje korisnicima da pišu izvorni kod u različitim jezicima, ali pritom da ti čvorovi mogu komunicirati na isti način. Može se smatrati nekom vrstom protokola koji korisnicima pojednostavljuje korištenje ROS-a. Veliki broj poruka je već predefiniran u ROS-u, ali korisnici isto tako mogu napisati svoje vlastite poruke samo je bitno da u nazivu stave ekstenziju *.msg* kako bi ROS mogao prepoznati tu poruku. Svaka poruka ima dva glavna dijela koja se zovu polja i konstante. Polja definiraju tip podatka koji se šalje u poruci i to može biti float64, int32, string ili bilo koji novi tip poruke koji korisnik definira. Konstante su imena za definirana polja kao imena za varijable u programskim jezicima. Primjer vlastite poruke koja se koristi za slanje koordinata prepoznatog objekta smještenog unutar datoteke *msg* prikazan je na slici 20. [14]



Slika 20. Primjer vlastite ROS poruke s podacima koji sadrže koordinate x i y

4.3.1.4. ROS servisi

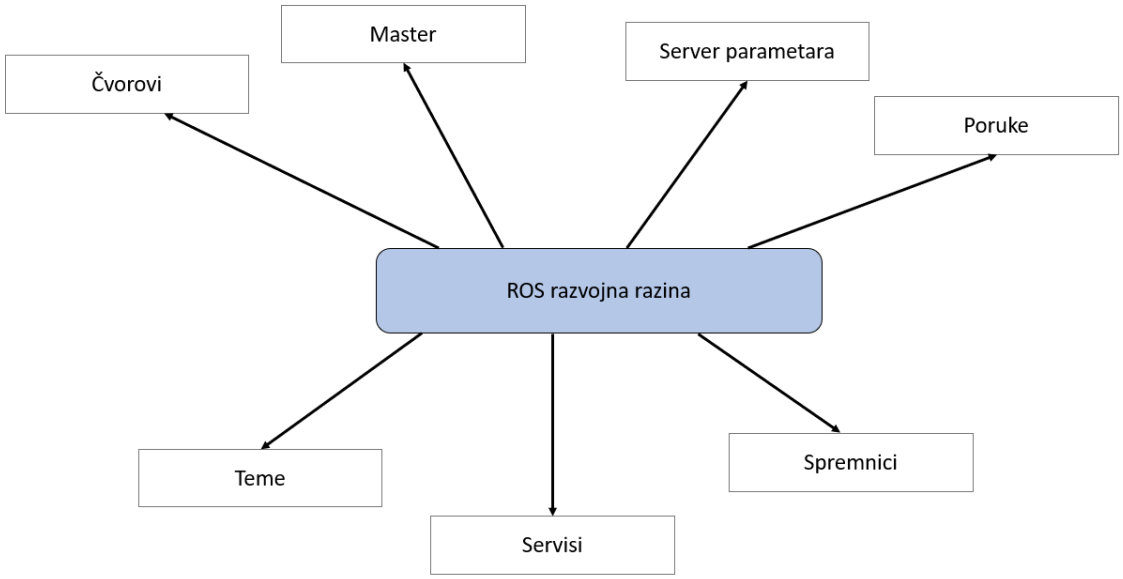
ROS servisi funkcioniraju na sličan način kao i ROS poruke. ROS koristi isti princip opisa vrste servisa deskriptivnim jezikom te temelji komunikaciju na vrsti ROS poruke kako bi omogućio način komunikacije zatraži/pošalji između različitih čvorova. Kod servisa je bitno naglasiti da je potrebno imati dva različita čvora tako da jedan, na primjer prima dvije vrijednosti, obradi ih i vrati njihov rezultat, a drugi čvor čita te vrijednosti. Tip servisa koji računa površinu pravokutnih objekata tako što uzima njihovu duljinu i visinu te vraća površinu prikazan je na slici 21.



Slika 21. ROS servis za izračunavanje površina detektiranih pravokutnika

4.3.2. ROS razvojna razina

ROS razvojna razina (*eng. Computation level*) je dio sustava koji se bavi mrežom procesa koje nazivamo čvorovi. Često se ova mreža naziva razvojnim grafom. Svaki čvor u mreži može pristupiti toj mreži, imati interakciju s drugim čvorovima, vidjeti informacije koje ti čvorovi šalju i prenositi podatke u mrežu. Struktura ROS razvojne razine prikazana je na slici 22.



Slika 22. Struktura ROS razvojne razine

Glavni koncepti razvojne razine su ROS čvorovi, master, server parametara, poruke, teme, servisi i spremnici.

Čvorovi (*eng. nodes*) su procesi koji izvode različite radnje u ROS-u. Svaki ROS čvor je napisan koristeći ROS klijent knjižicu *rospy* ili *roscpp*. Koristeći već postavljene dijelove ROS sustava možemo definirati različite načine komunikacije između čvorova. Odličan primjer prikaza broja čvorova je mobilni robota gdje je puno različitih procesa odnosno čvorova koji moraju komunicirati kako bi sustav mogao normalno funkcionirati. Ideja pisanja procesa na principu čvorova je da se naprave jednostavni procesi koji funkcioniraju kao jedinstveni sustav sa svim funkcionalnostima. Ako treba pronaći i otkloniti grešku u dijelu sustava to je puno lakše ako je sustav podijeljen na puno manjih procesa nego da je upravljanje napisano kao jedan složeni proces.

Master (*eng. Master*) je središnja jedinica ROS sustava koja omogućuje registraciju čvorova i pregled svih čvorova u sustavu. Također, omogućuje komunikaciju između čvorova.

Bez njega nije moguća komunikacija s drugim čvorovima, servisima, porukama odnosno bez mastera ROS ne funkcionira. Glavna osobitost je da može biti definiran samo jedan ROS master i ako se definira više od jednog mastera sustav neće raditi. ROS master je vrsta DNS servera, koji kada se pojedini čvor pokrene omogućuje registraciju imena tog čvora u ROS sustavu. Funkcionira na principu grupe internet protokola TCP/IP i posjeduje vlastitu varijablu *ROS_MASTER_URI* koja sadrži IP adresu i priključak ROS mastera. Koristeći tu adresu, čvorovi pronalaze mastera i on im omogućuje komunikaciju. Ako je varijabla kriva, komunikacija između čvorova nije moguća.

Server parametara (*eng. Parameter server*) je vrsta servera koji ima ulogu rječnika i omogućuje da spremamo različite varijable na principu ključ brava. Time je omogućeno da čvorovi prilikom normalnog rada ROS-a čitaju, mijenjaju ili brišu vrijednosti iz servera parametara. Ovakav način je idealan kod slučajeva kada imamo mobilnog robota kod kojeg je veći broj parametara gdje se parametri izmjenjuju između više različitih procesa.

Poruke (*eng. Messages*) se koriste kod komunikacije jednog čvora s drugim čvorom. Isti princip poruka se koristi kao i u datotečnoj razini. Poruke su strukture podataka koje sadrže vrijednosti koje se izmjenjuju između čvorova. Postoje standardni tipovi poruka koji su već postavljeni u ROS-u (*eng. integer, float, boolean*).

Teme (*eng. Topics*) su vrsta imena koje se pridjeljuju određenom skupu poruka. Kada čvor pošalje poruku preko teme, onda kažemo da čvor objavljuje (*eng. publish*) na određenu temu. S druge strane dok čvor primi poruku iz teme, kažemo da se pretplatio (*eng. subscribe*) na tu temu.Istovremeno može postojati više čvorova koji objavljuju podatke te one koji se pretplaćuju na određenu temu gdje jedan čvor može objavljivati i pretplaćivati se na više tema odjednom. Svaka tema ima jedinstveno ime i svaki čvor joj može pristupiti te poslati podatke ako sadrži poruku koja se nalazi u toj temi u suprotnom neće moći dobiti podatke na toj temi.

Servisi (*eng. Services*) se često koriste u robotskim aplikacijama u kojima model objavi/pretplati se neće biti dovoljan ako mu je potrebna interakcija zatraži i odgovori. Razlog je što princip objavi/pretplati se predstavlja vrstu jednosmjernog transporta i ako radimo s distribuiranim sustavom gdje je potrebna višesmjerna komunikacija taj princip neće biti dovoljan. Zbog toga se koriste ROS servisi. Oni nam omogućuju da napišemo dva čvora tako da je jedan odgovoran za zatraživanje podataka, a drugi za odgovor na zatraživanje. Dakle,

jedan čvor će biti server čvor a drugi klijent. Jednako kao i kod tema, svaki servis mora imati jedinstveno ime.

Spremnici (*eng. Bags*) su način spremanja i reprodukcije podataka koji se nalaze u ROS porukama. Važan su mehanizam za spremanje podataka, kao što su podaci sa senzora, koji se često teško prikupljaju, ali su nužni za razvoj i testiranje robotskih algoritama. Također, vrlo su korisni kada radimo s kompleksnim robotskim mehanizmima.

4.3.2.1. Čvorovi

Čvorovi su jedan od najvažnijih dijelova ROS ekosustava jer su to procesi koji izvode različite operacije koristeći standardne biblioteke *roscpp* ili *rospy*, gdje je roscpp za programski jezik C++, a rospy za Python. Jedan čvor može komunicirati s drugim čvorovima koristeći ROS teme, servise i parametre. Dobar primjer je mobilni robot koji sadrži veliki broj čvorova. Na primjer, jedan čvor je zaslužan za obradu slike s kamere, drugi obrađuje serijske podatke koji dolaze iz robota,a treći čvor obrađuje odometrijske podatke.

Glavna prednost ovakvog razvoja softvera pomoću čvorova, kao manjih procesa, je da ako se jedan čvor isključi to ne mora nužno značiti da cijeli sustav neće raditi. Funkcija, koju je obavljao taj robot, će prestati raditi i lako će se ustanoviti gdje je greška te kako ju otkloniti. Svaki čvor mora imati jedinstveno ime u ROS-u. Ime čvora se koristi kako bi se znalo koji čvorovi razmjenjuje informacije te kako drugi čvorovi ne bi mogli ometati komunikaciju trenutne razmjene podataka između čvorova.

4.3.2.2. Teme

ROS teme su dijelovi sustava kojima imenujemo skup poruka koje se koriste za izmjenjivanje poruka između čvorova. One mogu samostalno objavljivati i pretplaćivati se čime je stvaranje poruka odvojeno od njihovog korištenja. Čvorovi funkcioniraju tako da traže teme i provjeravaju odgovara li tip poruke čvoru koji se objavljuje ili pretplaćuje bez obzira na to koji čvor objavljuje kada je pretplaćen na određenu temu. Kako smo već spomenuli, komunikacija koja se odvija koristeći teme je jednosmjerna. Ako želimo obostranu komunikaciju trebamo koristiti ROS servise. ROS čvorovi komuniciraju koristeći protokol TCP/IP koji se zove TCPROS. Ova metoda je standardni način prijenosa informacija u ROS-u gdje je ROS master zaslužan za odvijanje stabilne komunikacije. Drugi tip komunikacije je

UDP način prijenosa podataka koji se zove UDPROS. Ovakav način ima malo kašnjenje i gubitak informacija te je pogodan za udaljeni prijenos.

4.3.3. ROS razina zajednice

ROS razina zajednice (*eng. Community level*) je razlog zašto ROS danas postaje sve popularniji. Omogućuje korisnicima različite načine razmjenjivanja znanja i softvera. Kako je ROS otvorenog tipa ovakav način je omogućio da se razvije snažna zajednica koja pomaže novim korisnicima pri rješavanju problema, diskusiji o različitim problemima te potiče rast i razvoj bolje zajednice. Koncepti ROS zajednice su različiti resursi koji su dostupni svima [14]:

- Distribucije - slično kao i kod Linux distribucija, ROS distribucije su skup verzija meta paketa koje možemo instalirati. One omogućuju jednostavnu instalaciju i prikupljanje ROS softvera.
- Repozitoriji – ROS se oslanja na mrežu repozitorija programskog koda gdje različite institucije mogu razvijati i pridonositi s vlastitim softverskim dijelovima za različite robote.
- ROS Wiki – je glavni forum za dokumentiranje podataka i informacija o ROS-u. Svatko se može prijaviti sa svojim računom i pridonositi dokumentaciji, predlagati poboljšanja i nove usluge, pisati upute itd.
- Sistem za praćenje grešaka – velika prednost je što ljudi često objavljuju uočene greške u softveru te ih zajedno s ostalim korisnicima onda pokušavaju otkloniti.
- Lista za mailove – primarni komunikacijski kanal je preko maila, gdje se dobivaju različite informacije o novim verzijama te forum na kojem se mogu postaviti pitanja vezana uz ROS softver.
- ROS odgovori – podaci na internetu pomažu postavljanju pitanja vezanim uz ROS. Kada iznesemo svoj problem, drugi ROS korisnici ga mogu pročitati i pomoći nam jer su se možda već susreli sa sličnim problemom.
- Blog – donosi informacije o vijestima, slikama i videima vezanima uz ROS te novim stvarima koje su implementirane unutar ROS-a.

4.4. Razlog korištenja ROS sustava

Glavni cilj ROS-a je zahtjev za modularnost i ponovnu uporabljivost. ROS sustav je potpuno modularan. Pitanje koje se nameće pri korištenju svakog sustava s robotskim aplikacijama je hoće li se zaustaviti aplikacija robota ako se bilo koji od dijelova glavnog koda sruši. Izradom čvorova za različite procese, ROS omogućuje da sustav funkcionira neovisno o pojedinom čvoru. Dakle, ROS pruža robusne metode za nastavak funkcionalnog rada sustava čak iako neki senzori ili aktuatori ne funkcioniraju.

Postoje visoke mogućnosti ROS-a jer ROS dolazi s već predodređenim funkcionalnostima kao što su SLAM (*eng. Simultaneous Localization and Mapping*) i AMCL (*eng. Adaptive Monte Carlo Localization*). Ti paketi mogu se koristiti prilikom autonomne navigacije mobilnih robota te pomoću *MoveIt* paketa za planiranje gibanja robotskih manipulatora. Ove mogućnosti mogu se koristiti unutar našeg robotskog softvera bez ikakvih komplikacija. Također, ove funkcionalnosti su vrlo prilagodljive, što znači da se lako mogu podešavati pomoću različitih parametara unutar samog ROS-a.

ROS pruža jako puno alata za uklanjanje grešaka, vizualizaciju te izvođenje simulacije. Alati kao što su rqt_gui, Rviz i Gazebo neki su od snažnih alata dostupni svima i koriste se za vizualizaciju, uklanjanje pogreška i simulaciju. Softverski paket koji ima ovako puno alata pruža korisnicima različite mogućnosti.

Odlična senzorska i aktuatorska podrška čini ROS kompatibilnim s mnogim sustavima. Unutar ROS-a nalaze se upravljački programi i paketi različitih senzora i aktuatora koji se koriste u robotici. Neki od senzora koje obuhvaća paket su Velodyne-LIDAR, laserski skeneri, Kinect. Komponente se jednostavno povezuju s ROS-om pomoću drivera kojeg su drugi ljudi već napisali, testirali te implementirali unutar ROS-a.

Operativnost među platformama vrši se preko ROS poruka koje služe za komunikaciju različitih čvorova u sustavu. Ti se čvorovi mogu programirati u bilo kojem programskom jeziku koji posjeduje ROS biblioteku. Dakle, možemo napisati čvorove za visoke performanse u jezicima C/C++, a ostale čvorove u programskim jezicima Pythonu ili Javi. Ovakva fleksibilnost nije dostupna u drugim robotskim sustavima.

4.5. Instalirana verzija ROS-a

ROS trenutno postoji za operativne sustave Linux, macOS i Windows. U početku je ROS bio namijenjen samo za distribucije Linux operativnog sustava, no danas je moguć i na ostalim operativnim sustavima. Kada se odabire verzija ROS sustava koja će se instalirati na uređaj bitno je paziti da je ta verzija ROS-a kompatibilna s verzijom operacijskog sustava na računalu. Verzija operativnog sustava na lapopu na kojem je testiran softver je Ubuntu 16.04.06 Xenial Xerus. Za ovu Ubuntu verziju odabran je ROS Kinetic Kame jer je to najstarija stabilna verzija koja se trenutno još uvijek verzionira. Glavni razlog odabira ove verzije je što je većina drivera za komponente napisana i testirana te je najbolje koristiti verziju koju koristi trenutno najveći broj korisnika.

4.6. ROSARIA

ROSARIA je paket u kojem se nalazi čvor RosAria. Čvor Rosaria omogućava povezivanje ROS-a s većinom robota koji su proizvedeni u tvrtkama ActivMedia Robotics i Adept MobileRobots kao što su Pioneer 2, Pioneer 3, AmigoBot, PowerBot, PeopleBot i svi slični koji podržavaju biblioteku otvorenog koda ARIA-u. Informacije o bazi robota, upravljanju brzinom i akceleracijom je implementirano unutar RosAria čvora, koji objavljuje teme s podacima dobivenih iz robotskog ugradbenog sustava kojeg kontrolira ARIA. Način komunikacije s robotom odvija se i dalje preko ARIA-e gdje se naredbe brzine i akceleracije šalju u ROS-u preko ROSARIE koja onda dalje komunicira s ARIA-om. [15]

Glavna prednost ROS-a može se uočiti kroz ROSARIA-u, a to je da su ljudi iz zajednice odlučili napisati ROS čvorove koji će se koristiti kako bi se moglo upravljati mobilnim robotima kroz ROS. Nakon što su uspješno testirali i integrirali taj dio u ROS-u, objavili su ga na stranicama od ROS-a što omogućuje ostalim korisnicima jednostavno konfiguriranje i korištenje.

5. NADOGRADNJA POSTOJEĆIH KOMPONENTI MOBILNOG ROBOTA

Prvi korak izrade diplomskog rada je utvrđivanje funkcionalnosti pojedinih dijelova mobilnog robota. Mobilni robot Pioneer 2-DX proizveden je 1998. godine i zbog vremena od tada do danas, neki dijelovi su izgubili svoju funkcionalnost te ih je bilo potrebno zamijeniti. Jedan od prvih zadataka bio je pronaći validne zamjenske module mobilnog robota kako bi se robota moglo koristiti i omogućiti daljnji razvoj upravljačkog algoritma. Problem je bio što je tvrtka ActivMedia Robotics, koja je proizvela ovaj robot, prestala raditi 2018. godine i nije postojala nikakva mogućnost komunikacije s tvrtkom te dobivanja informacija o pojedinim dijelovima. U korisničkom priručniku nisu postojale električne sheme za proučavanje načina spajanja pojedinih komponenti te je bilo teško shvatiti na koji način su povezani pojedini dijelovi. Nakon detaljne analize i testiranja svih dijelova mobilnog robota, utvrđeno je da integrirano računalo, kamera te baterije mobilnog robota nisu u funkciji. Radi razvoja upravljanja mobilnog robota, navedene dijelove je bilo potrebno zamijeniti novim dijelovima te ih integrirati na pravilan način.

S ciljem zamjene i poboljšanja dijelova robota koji nisu u funkciji dogovorena je suradnja s tvrtkom Greyp Bikes tako da se pojedini dijelovi s električnog bicikla pokušaju implementirati i integrirati na postojeći mobilni robot. Kako se bicikl sastoji od sličnih modula kao i mobilni robot pojavio se zajednički interes istraživanja mogućnosti korištenja komponenti s bicikla na mobilnom robotu. Električni bicikl se, osim klasičnih mehaničkih dijelova kao što su gume, pedale, rama, volan, sastoji i od baterije s odgovarajućim sustavom za upravljanje radom baterije, s dvije kamere smještene na prednjem i stražnjem dijelu bicikla, elektromotora te centralne upravljačke jedinice koja omogućuje funkcionalnost svim ostalim modulima. Upravljački softver na biciklu sastoji se od brojnih procesa koji se moraju istovremeno odvijati, na primjer asistencija na motoru, spajanje na mrežu, odgovornost za sliku kamere. Cijeli sustav baziran je na ROS sustavu i svi ovi procesi napisani su u obliku čvorova koji se pokreću prilikom paljenja bicikla. Isti princip može se primijeniti kod mobilnog robota te je iz tog razloga odlučeno da se s bicikla preuzme kamera, baterija s odgovarajućim sustavom za upravljanje radom baterije te istraže mogućnosti integracije centralne upravljačke jedinice kao središnji sustav za upravljanje mobilnim robotom. U ovom poglavlju su opisane sve komponente koje su uspješno integrirane te je prikazan način na koji su spojene s mobilnim robotom. Na slici 23. prikazan je bicikl G6 tvrtke Greyp Bikes.

Slika 23. Električni bicikl tvrtke Greyp Bikes [15]

5.1. Baterija i BMS

Prva komponenta koju je bilo potrebno integrirati kako bi mogli testirati robota je baterija. Baterija na biciklu je dimenzijom veća i kapacitivno snažnija nego mobilni robot i zato ju je trebalo redimenzionirati tako da stane u kućište robota, ali isto tako trebalo je paziti da će moći funkcionirati s ostalim komponentama na mobilnom robotu.

Pioneer 2-DX je imao tri akumulatorske baterije koje su se mogle lako izvaditi i zamijeniti te odgovarajući sustav za kontrolu napona baterija. Jedna baterija je bila napona 12V i kapaciteta 7Ah što je 85Wh čime su tri potpuno iste baterije dale maksimalan kapacitet od 252Wh. Svojstvo akumulatorskih baterija je da su jeftinije, ali zato dugoročno nisu isplative. Rezultat toga je prestanak rada nakon nekoliko godina radi potrebe za stalnim održavanjem kapaciteta baterija. Prednost akumulatorskih baterija je mogućnost punjenja čime je omogućen rad mobilnog robota i do četiri sata. Nakon što su se baterije potrošile robot je mogao dalje funkcionirati tako da ga priključimo na punjač čime je omogućen rad robotskog sustava dok se on puni. Sustav od tri baterije imao je određenu težinu čime je održavana ravnoteža robota jer bez baterija robot bi se naginjao prema naprijed. Ovi podaci su bili ulazni parametri za proračun baterije koja se sastoji od litij-ionskih ćelija i koja se nalazi u središnjem dijelu bicikla što je prikazano na slici 23. Prednost korištenja litij-ionskih baterija je primjena moderne tehnologije u koje ulažu novce sve više tvrtki u svijetu te se razvila čitava znanost o tome kako pravilno regulirati baterijske ćelije. Jedna od glavnih prednosti povezivanja baterija pomoću baterijskih

ćelija je što baterije nije potrebno održavati već se odgovarajućim baterijskim sustavom prati njihov vijek trajanja. One su puno duljeg vijeka trajanja nego klasične akumulatorske baterije koje se također mogu puniti i na njima se danas razvijaju mnogi električni automobili, bicikli, romobili itd. Negativna strana korištenja baterijskih ćelija je što su u početku puno skuplje od akumulatorskih baterija, ali ako se gleda omjer cijene s obzirom na vijek trajanja onda dolazimo do zaključka da su puno isplativije. Na slici 24. lijevo prikazana je klasična akumulatorska baterija, a na desnoj strani jedna litij-ion ćelija.

Slika 24. Lijevo - akumulatorska baterija 12V i 7Ah [16]. Desno - jedna baterijska ćelija tipa 18650 napona 3.2V i 2.6Ah [17]

Ulazni parametar za dimenzioniranje baterije s baterijskim ćelijama bio je kapacitet baterije s baterijskim ćelijama koji mora biti sličan kapacitetu robota s tri akumulatorske baterije. Također, željeni kapacitet mora biti veći od 252Wh te je baterija morala davati minimalni napon od 12V koji je potreban za normalan rad sustava mobilnog robota.

Baterijski paket sastoji se od baterijskih ćelija koje se spajaju serijski ili paralelno. Spajanjem baterijskih ćelija serijskim spojem dobivamo veći napon baterijskog paketa tako da pritom kapacitet ostaje i jednak je kapacitetu jedne baterije. S druge strane kada slažemo baterije u paralelni spoj dobivamo veći kapacitet, ali napon ostaje isti kao napon jedne baterijske ćelije. Kada povežemo ova dva načina i povećamo broj baterijskih ćelija dobivamo jednu bateriju koja traje puno dulje od tri akumulatorske baterije i možemo ju dimenzionirati ovisno o namjeni i problemu.

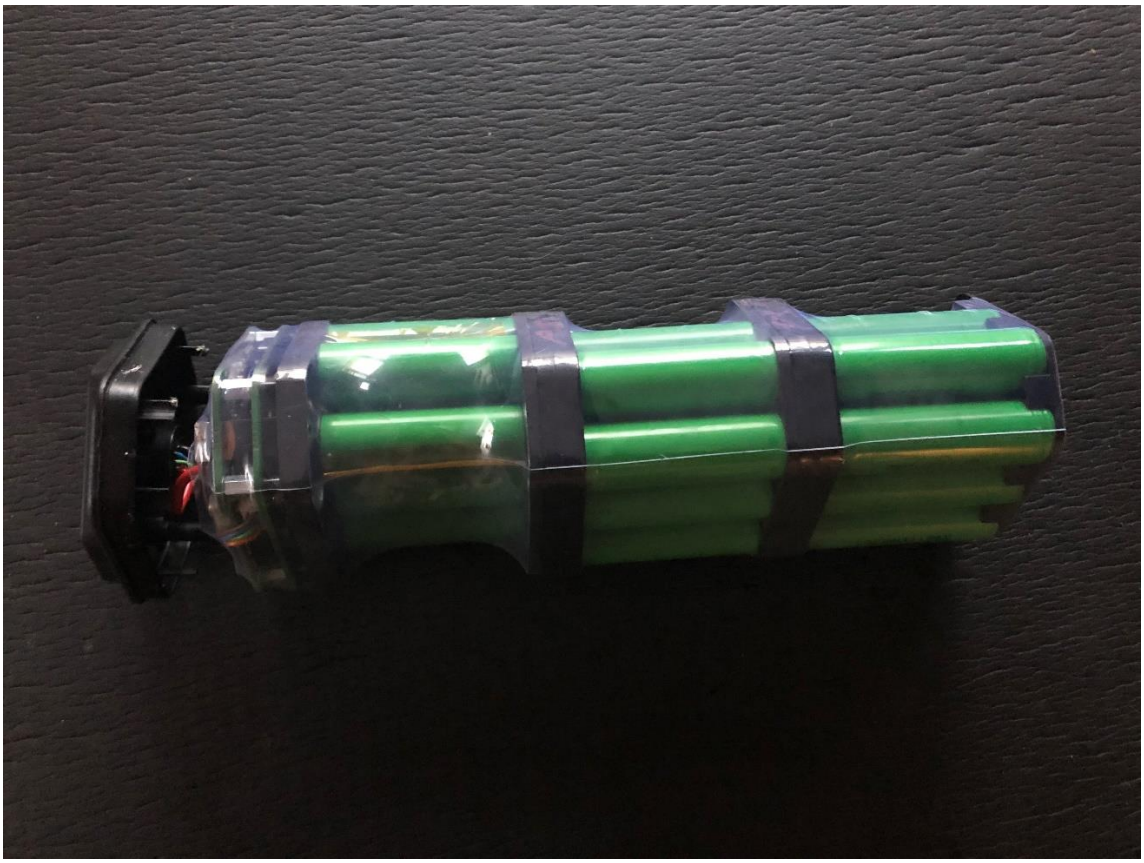
Za mobilni robot u ovome radu napravljen je baterijski paket koji se sastoji od šest baterija u serijskom spoju i pet u paralelnom spoju. Maksimalni napon jedne baterijske ćelije koja je napunjena je 4.2V što znači da kada imamo šest baterija u serijskom spoju dobivamo maksimalni napon baterije od 25.2V. Taj napon neće se nikad ostvariti na bateriji i gleda se prosječni napon tijekom pražnjenja koji iznosi 3.635V. Maksimalni napon bitan je prilikom punjena baterije jer ne smijemo prijeći napon 25.2V, inače dolazi do oštećenja baterijskih ćelija. Kako bi baterija mogla trajati što dulje spojene su baterije i u paralelni spoj tako da se dobije veći kapacitet baterije. U paralelnom spoju stavljene su četiri baterije te se dobiva maksimalni kapacitet od 305.3kW. Baterijski paket je napravljen da ima veći kapacitet nego što je bio prije što znači da će se robot moći dulje koristiti. Potrebno vrijeme punjenja baterije je oko jedan sat. U tablici 5.1 su prikazane vrijednosti jedne baterijske ćelije LG MJI 18650, a u 5.2 baterijskog paketa za mobilni robot. Na slici 25. može se vidjeti kako je sastavljen baterijski paket.

Tablica 5.1 Vrijednosti baterijske ćelije LG MJI 18650

Nominalni kapacitet [Ah]	3.5
Minimalni kapacitet [Ah]	3.4
Prosječni napon tijekom pražnjenja [V]	3.635
Maksimalni napon [V]	4.2
Maksimalna struja pražnjenja [A]	10
Napon isključivanja ćelije [V]	2.5

Tablica 5.2 Karakteristike baterijskog paketa integriranog u mobilni robot

Broj ćelija u serijskom spoju	6
Broj ćelija u paralelnom spoju	4
Ukupan broj baterija	24
Prosječni napon [V]	21.81
Maksimalni napon [V]	25.2
Prosječni kapacitet [Wh]	305.3
Maksimalni kapacitet [Wh]	352.8
Maksimalni kapacitet [Ah]	14

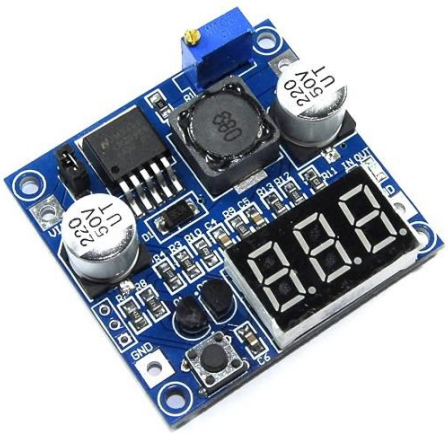


Slika 25. Baterijski paket za mobilni robot

Baterija je opremljena s odgovarajućim sustavom za upravljanje radom baterije - BMS (eng. Battery Management System) koji je zadužen za brojne sigurnosne dijelove vezane uz baterijske ćelije. Jedan od zadataka je da prati napon pojedinačnih ćelija, ali isto tako pojedinih serija i paralela. Baterijski sustav isključuje bateriju ako je napon bilo koje ćelije ispod 2.8V. Razlog isključivanja baterijskog sustava na 2.8V je jer popratna elektronika na BMS sustavu troši energiju, točnije RTC (*eng. Real Time Clock*) čip koji prati trenutno vrijeme. Minimalni dozvoljeni napon ćelije je 2.5V što znači da što je baterija dulje ispod te vrijednost dolazi do značajnog oštećenja kapaciteta baterije. Ako baterijski sustav ne bi gasio bateriju na 2.8V postojala bi velika opasnost od oštećenja kapaciteta baterijskih ćelija. Jedna od značajki unutar sustava je temperaturni sigurnosni dio koji prati temperaturu baterijskih ćelija i baterije jer je ovakav tip baterije vrlo opasan ako se dogodi pregrijavanje te može dovesti do ozbiljnih posljedica. Preko CAN protokola mogu se čitati različiti podaci o bateriji kao što su trenutni napon baterije, napon baterijske ćelije, njihova temperatura i sve ostale pojedinosti vezane uz bateriju. Na slici 26. dolje je prikazana baterija za mobilni robot, a gore za električni bicikl tvrtke Greyp Bikes.

Slika 26. Usporedba baterije s bicikla (gornja baterija) s umanjenom verzijom baterije za mobilni robot (donja baterija)

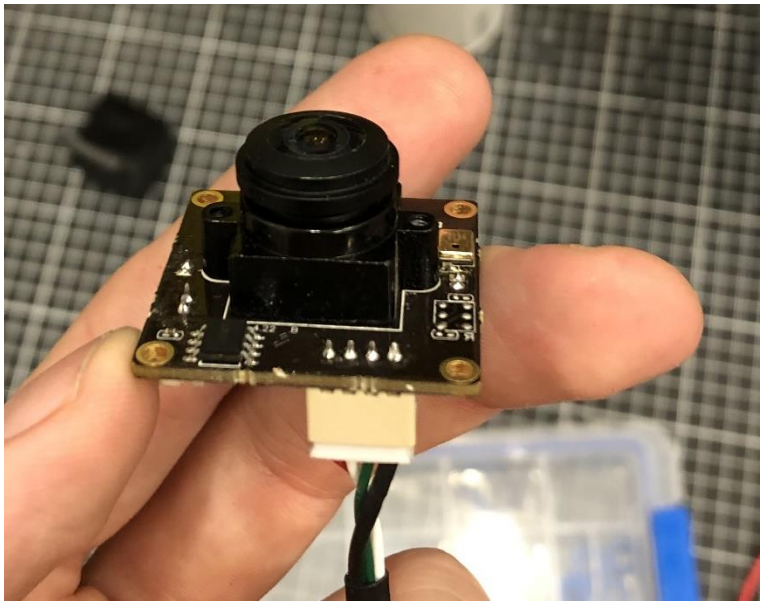
Problem prilikom integracije baterije mobilnog robota bio je napon baterijskog paketa koji se nalazi u rasponu od 15V do maksimalne vrijednosti od 25.2V dok je potrebni napon za upravljanje mobilnog robota 12V. Kako bi iz baterije dobivali konstantni napon bilo je potrebno integrirati dodatnu pločicu koja će služiti za pretvaranje napona s 25.5V na stalnih 12V, ali isto tako da se omogući i mijenjanje napona na izlazu. Ovdje je iskorištena gotova pločica LM2596-2 sa sklopom za spuštanje i regulaciju izlaznog napona. Raspon ulaznog napona je od 4.1V do maksimalno 37V, a izlaza 1.2V do 37V uz maksimalnu struju od 3A.



Slika 27. Modul za spuštanje napona LM2596-2 [18]

5.2. Greyp kamera

Druga komponenta koju je bilo potrebno zamijeniti na robotu je kamera. Kamera koja je došla zajedno s Pioneer 2-DX mobilnim robotom bila je Pioneer Pan-Tilt-Zoom kamera. Ona je slala video u NTSC ili PAL formatu koji su vrsta prijenosa video signala u analognom obliku. Radi poboljšanja vizijskog sustava i načina prijenosa video signala u digitalnom obliku, odlučeno je da se implementira kamera koja se koristi na Greyp električnom biciklu. Greyp kamera ima mogućnost prikaza video sadržaja visoke razlučivosti (1920 x 1080 piksela) s brzinom stvaranja slike od 30 slika u sekundi. Video se prenosi u formatu H.264 MJPEG koji je vrsta modernog digitalnog načina prijenosa signala gdje se svaka sekvenca videa šalje posebno u obliku JPEG slike. Spajanje s računalom omogućeno je preko USB 2.0 priključka i maksimalna potrošnja struje joj je 190mA što je puno manje od prijašnje kamere. Greyp kamera može se vidjeti na slici 28.



Slika 28. Greyp kamera

5.3. Upravljačko računalo

Zadnji modul koji je zamijenjen je integrirano upravljačko računalo. Pioneer mobilni robot imao je ugrađeno svoje računalo, koje je za vrijeme u kojem je razvijeno bilo dovoljno za pokretanje raznih programa i obradu slike. Pokretanje današnjih programa, obrada slike i izvođenje više različitih procesa na njemu jednostavno ne bi bilo moguće. Računalo se nažalost nije moglo upaliti niti testirati kako bi se preuzeli programi i datoteke koje su potrebni za

upravljanje mobilnim robotom kao što je ARIA te jednostavno vidjeti koje datoteke bi se mogle iskoristiti za komunikaciju s mikrokontrolerom.

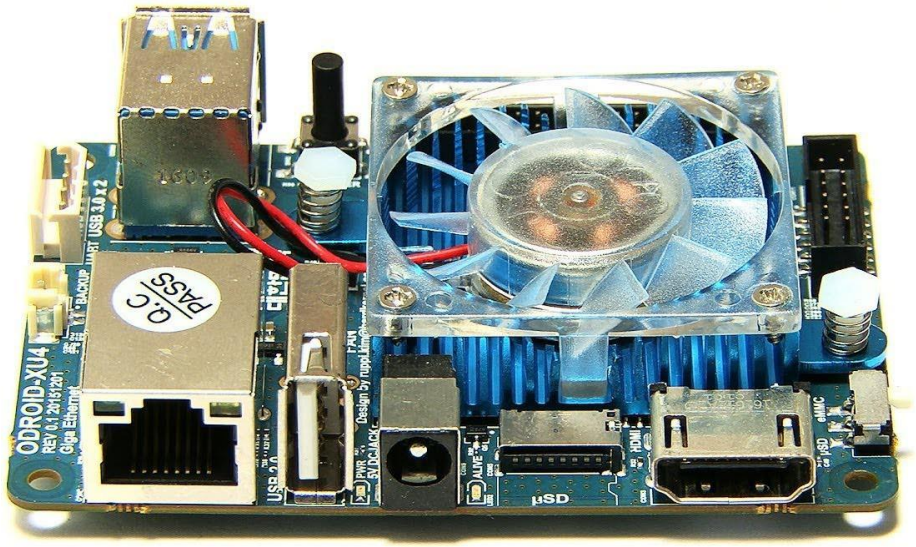
Prva zamisao bila je iskoristiti središnju upravljačku jedinicu – CIM (*eng. Central Intelligence Module)* s Greyp bicikla kako bi se ona pokušala integrirati kao središnji sustav za upravljanje mobilnim robotom. Svi procesi bili bi napravljeni isto kao i čvorovi u ROS-u koji bi se odvijali na isti način kao i procesi na biciklu. Prvi i glavni problem bio je nemogućnost jednostavne integracije sustava ARIA i ROSARIA na CIM. Iako su procesi na biciklu napisani u obliku čvorova unutar ROS-a, njihov način pokretanja je puno drugačiji od načina na koji ARIA funkcionira. Čvorovi su komprimirani kako bi zauzimali što manje memorije na upravljačkom računalu i postupak implementacije ARIA-e tako nije bio moguć. ARIA je skup od nekoliko stotina biblioteka i klasa koje je radila cijela jedna firma posljednjih dvadeset godina. Proces prepisivana i integriranja ARIE u komprimiranom obliku zahtijevao bi jako puno vremena i razumijevanja cijelog načina funkcioniranja sustava na CIM-u i nakon toga još sustava ARIA.

Zbog navedenog, trebalo je pronaći zamjenski sustav koji će biti dovoljno snažan da pokrene sve potrebne procese od upravljanja robota pa sve do obrade slike. Prva opcija bila je industrijsko računalo s brzim SSD diskom i jakim procesorom. Problem kod industrijskih računala je što su cijenom puno skuplji te prilikom normalnog rada troše puno više struje. Druga opcija bila je koristiti laptop na kojem se mogu pokretati ROS i ARIA. Laptop posjeduje vlastitu bateriju i ne bi bio trošio bateriju mobilnog robota. Ovaj slučaj je bio dobar u obliku testiranja i privremenog rješenja, ali je također cijenom neprihvatljiv. Treća opcija bila je primjena mikroračunala koji ima procesor ARM arhitekture (*eng. Advanced RISC Machines*). Ovakva vrsta procesora danas se nalazi u mnogim manjim uređajima npr. mobiteli, pametni satovi, pametni televizori, auti itd. ARM tip arhitekture bazira se na smanjenom setu naredbi, ali zato imaju procesore s visokim frekvencijama koje se mjere u gigahercima i mogu izvoditi milijune instrukcija u sekundi. Performanse su im bolje za određen broj naredbi te prilikom rada troše puno manje snage po satu nego industrijsko računalo. Procesori mogu biti tipa 32-bit ili 64-bit, na koje je bilo bitno paziti prilikom odabira mikroračunala.

Izbor je sveden na tri mikroračunala, a to su Raspberry Pi 3 Model B+, Odroid-XU4 i NVIDIA Jetson Nano. U tablici 5.1 navedene su karakteristike i specifikacije navedenih mikroračunala radi usporedbe i razlog odabira. Prema performansama odabir je skraćen na tri

slična mikroračunala koje proizvode tri velike kompanije. Proizvođač Raspberry Pi je Sony, Odroida je Samsung, a Jetsona Nano tvrtka NVIDIA. S obzirom na brzinu procesora Odroid-XU4 ima najbolja obilježja, dok je što se tiče tipa procesora iza Raspberryja i Jetsona Nano. Najbolju radnu memoriju ima Jetson Nano, nakon toga Odroid i na zadnjem mjestu je Raspberry pi 3B+. S obzirom na broj USB i vrstu priključaka, te I2C, UART, SPI i HDMI priključaka su gotovo istih značajki. Nažalost, niti jedan nema potreban RS232 priključak, a kasnije će se prikazati kako je otklonjen taj nedostatak. Što se tiče ulaznog napona, vidimo da su mikroračunala također istih karakteristika, no što se tiče potrošnje tu je Odroid XU-4 na prvom mjestu jer može povući struju sve do 4A, dok druga dva su duplo manji potrošači i mogu povući maksimalno 2A. Važno je naglasiti da Odroid XU-4 u najopterećenijim uvjetima povlači struju od 4A tako da je to relativno zanemariva činjenica. Što se tiče cijene, najskuplji je Jetson Nano, nakon njega Odroid XU-4 ,a najjeftiniji je Raspberry pi 3B +.

Odabrani uređaj za integraciju je Odroid XU-4 zato što iako ima 32-bitni procesor, ima najbrži procesor s osam jezgri koji će moći obraditi dovoljno procesa kada će se na njemu pokretati ROS i ARIA. Drugi razlog je što podržava Ubuntu verziju 16.04 kao i Raspberry pi, koja je vrlo bitna jer nam je potrebna Kinetic Kame verzija ROS-a na kojemu se odvija softver na biciklu i čiji dijelovi će se onda moći iskoristiti. Iako ima manju količinu RAM memorije od Jetsona Nano te veću potrošnju prilikom najvećeg opterećenja, svejedno je prvi izbor zbog gore navedenih razloga. Kada bi Jetson Nano podržavao verziju Ubuntu 16.04 bio bi prvi izbor, ovako ipak po svim ostalim značajkama Odroid XU- 4 je bolji.



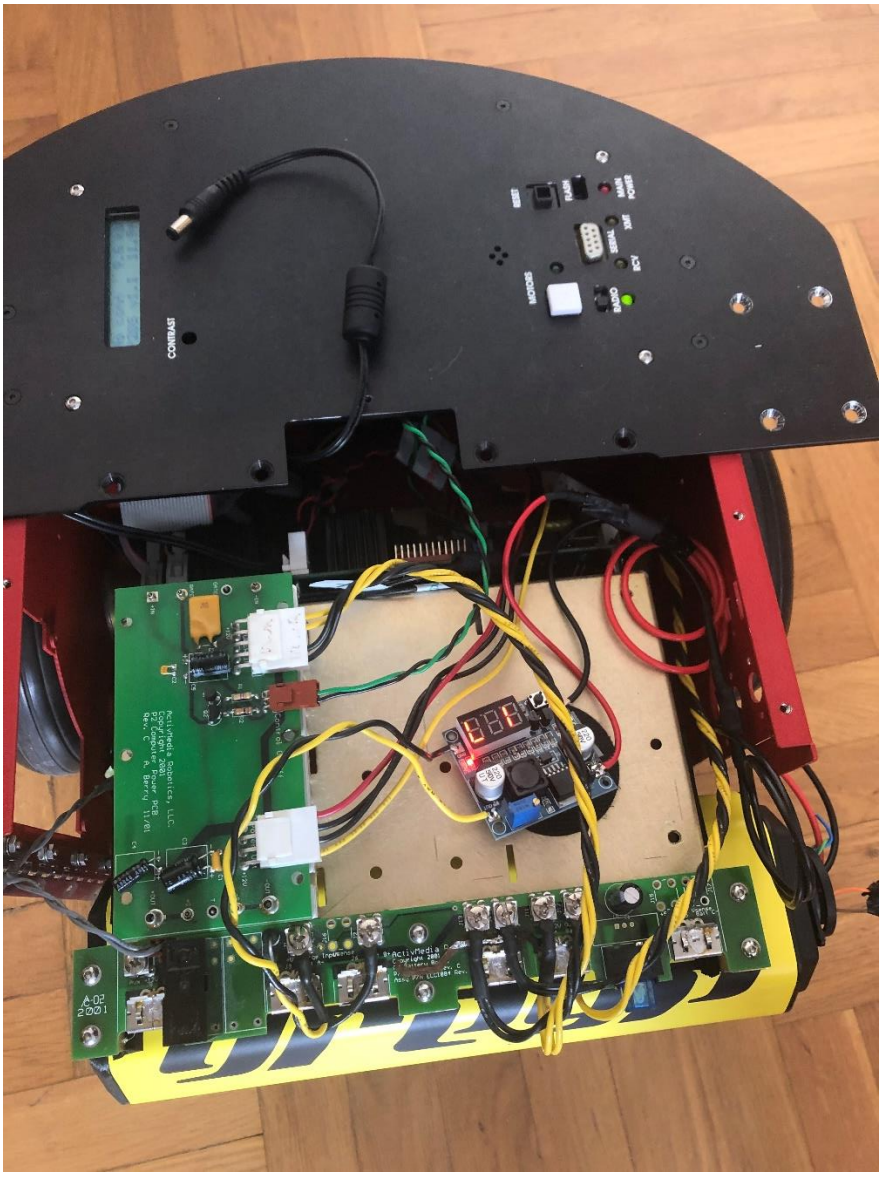
Slika 29. Odabrano mikroračunalo Odroid XU-4

Tablica 5.3 Usporedba mikroračunala sličnih karakteristika

	Raspberry Pi 3 B+	**Odroid-XU4**	**NVIDIA Jetson Nano**
Čip - SoC	Broadcom BCM28370B0	Samsung Exynos-5422	Nvidia Erista
Procesor	ARM Cortex-A53 (64-bit) 1.4GHz quad core	ARM Cortex-A15 / ARM Cortex-A7 (32-bit) 2GHz octa core	ARM Cortex-A57 MPCore (64-bit) 1.43GHz quad core
Veličina RAM	1GB	2GB	4GB
Ugrađeni RAM	DA	DA	DA
Broj USB priključka	4	3	4
Verzija USB priključka	2.0	2 x 3.0 + 1 x 2.0	3.0
Podržava Linux	Raspbian	DA Linux distribucija Ubuntu 16.04	JetPack 4.2 SDK Linux distribucija Ubuntu 18.04
Ulazni napon	4.8V – 5.2V	4.8V – 5.1V	5V
Ulazna struja	600mA - 2.4A	1A - 4A	2A
HDMI priključak	DA	DA	DA
I2C	1	1	1
SPI	1	1	1
UART	1	2	1
RS232	NE	NE	NE
Cijena	330 KN	800 KN	1200 KN

5.4. Promjene na konstrukciji

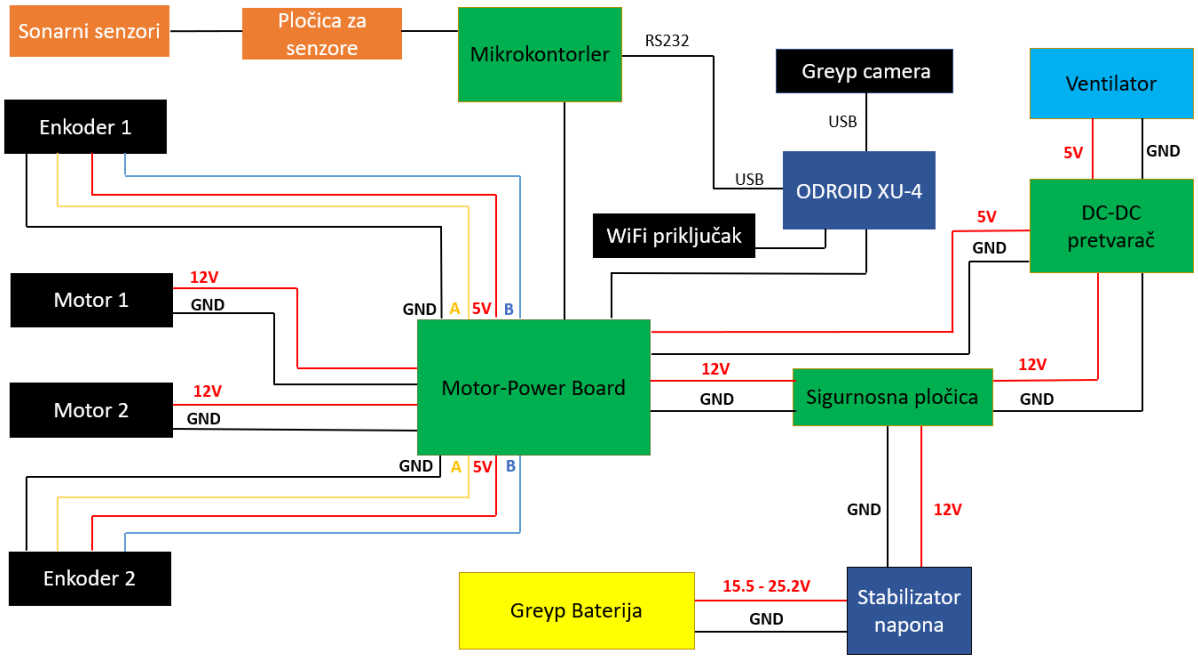
Dijelovi robota koji nisu bili u funkciji su uklonjeni iz robota čime je došlo do smanjena ukupne mase mobilnog robota. Prijašnja masa mobilnog robota iznosila je 10kg, a trenutna iznosi 9kg. Bitne promjene na konstrukciji su napravljene kako bi se mogla baterija s odgovarajućim sustavom sigurno smjestiti u robota te kako baterija prilikom vožnje se ne bi kretala i sudarala s kućištem ili drugim dijelovima robotskog sustava. Rješenje je napravljeno tako da se baterija sa svojim sustavom može lako izvaditi iz robota kao što je to moguće na biciklu. Aluminijsko kućište u kojem su bile tri akumulatorske baterije iskorišteno je za trenutnu bateriju. Na slici 30. je prikazan stražnji dio kućišta robota u koji je smještena baterija.



Slika 30. Integrirana baterija na mobilni robot

5.5. Električna shema mobilnog robota

Električna shema mobilnog robota napravljena je kako bi se lakše prikazao način na koji su spojene pojedine komponente, koliki izvor napajanja im je potreban te kolika je maksimalna struja koju trebaju za normalan rad. Gledajući shemu na slici 26. krenut ćemo od izvora napajanja robota, a to je Greyp baterija. Baterija može maksimalno dati napon od 25.2V, koji kako baterija radi se počinje smanjivati. Kada napon padne ispod 15V onda ju njen sustav za upravljanje gasi kako ne bi došlo do oštećenja baterijskih ćelija. Baterija je povezana sa stabilizatorom napona LM2597-2, koji spušta ulazni napon u rasponu od 25.2V do 15V na napon od 12V koji je potreban za normalan rad sustava mobilnog robota. Iz sigurnosnih razloga ostavljena je zasad pločica koja je služila kao zaštita prijašnjeg sustava od tri baterije tako da se stabilizator napajanja direktno spaja na sigurnosnu pločicu na kojoj se nalazi osigurač. Osigurač je samo dodatna mjera opreza protiv kratkog spoja, preopterećenja ili previsokog napona jer baterijski sustav ima svoju zaštitu i on služi za zaštitu svih ostalih komponenti robotskog sustava. Sigurnosna pločica spojena je na dvije strane. Jedna strana je spojena s pločicom koja služi za spuštanje napona s 12V na 5V uz maksimalnu struju od 1.5A, dok je druga strana spojena s dijelom za napajanje robota. Napon od 5V koristi se za napajanje ventilatora koji se nalazi unutar kućišta i rashlađuje unutrašnjost mobilnog robota. Napon od 12V iz sigurnosne pločice i napon od 5V iz pretvarača povezuju se direktno u pločicu koja se koristi za upravljanje motora, ali isto tako i za napajanje svih ostalih dijelova mobilnog robota. Dva motora i dva enkodera povezana su s driverom za motore tako da dobivaju napon od 12V, odnosno 5V za enkodere uz ograničenje struje od maksimalno 1.5A. Mikrokontroler mobilnog robota povezan je preko sabirnice s pločicom za drivere i od nje dobiva napajanje od 12V i 5V. Skup sonarnih senzora povezan je na pločicu za senzore koja je odgovorna da senzori ne šalju podatke iz okoline u isto vrijeme te da se omogući napajanje od 5V potrebno za senzore. Pločica za senzore je spojena sabirnicom s mikrokontrolerom od kojeg dobiva naredbe o vremenu aktivacije pojedinog senzora.



Slika 31. Električna shema spajanja komponenti mobilnog robota

Zadnja komponenta koja je integrirana u sustav je upravljačko mikroračunalo Odroid XU-4. Karakteristika da može povući struju od 1A do 4A predstavljala je problem kod integracije za ovakav sustav jer je veći dio robotskog sustava ograničen maksimalnom strujom od 1.5A. Rješenje je izvedeno tako da je ispitivan priključak korišten za napajanje prijašnjeg integriranog računala na priključku J5. Priključak J5 ima vlastiti regulator struje te mogućnost napajanja od 5V uz struju od 7A. Nažalost pinovi priključka nisu bili funkcionalni te je spoj zalemljen direktno na pločici regulatora. Prije korištenja regulatora trebalo je provesti određena testiranja kako bi bili sigurni da će moći napajati Odroid mikroračunalo. Provedeno je nekoliko stres testova gdje je Odroid bio opterećen različitim načinima kako bi se pratila temperatura na regulatoru. Optimalna temperatura regulatora pri maksimalnom opterećenju je između 70°C i 80°C. Ako bi došlo do povećanje te temperature, značilo bi da regulator ne radi kako treba i moglo bi doći do oštećenja na regulatoru, a onda i na drugim komponentama robotskog sustava.

5.6. Konačni prikaz robota

Mobilni robot je na kraju sastavljen i spojen prateći shemu prikazanu na slici 31. Baterija je ugrađena u mobilni robot tako da se lako može izvaditi i koristiti za čitanje podataka s baterije. Svi glavni dijelovi su izolirani i ispitani kako bi robot bio siguran za korištenje. Iskorišteno je kućište od prijašnjeg mobilnog robota koje je prekriveno zaštitnim premazom te je prelakirano kako bi se poboljšao vizualni izgled mobilnog robota. Konačni izgled sastavljenog mobilnog robota i povezanim svim komponentama robotskog sustava prikazan je na slici 32.

Slika 32. Konačni prikaz mobilnog robota

6. POVEZIVANJE ROBOTA S KOMPONENTAMA

Mobilni robot, nakon zamjene i poboljšanja pojedinih komponenata, je spreman za proces povezivanja s ostalim komponentama za upravljanje. Prvi korak je instalacija robotskog operativnog sustava – ROS na upravljačka računala, laptop i mikroračunalo Odroid XU-4. Zatim je potrebno podesiti sustav ARIA koji se koristi za komunikaciju s mikrokontrolerom robota te prikladni čvor ROSARIA. Na kraju, prikazat će se kako su spojeni kamera, senzori, kontroleri za motore te kako su one povezane s upravljačkim računalom.

6.1. Instalacija ROS-a

Inicijalni korak za povezivanje svih komponenti u sustavu je instalirati ROS. Robotski operativni sustav može se skinuti i instalirati prateći korake sa službene ROS stranice. Detaljno je prikazan način konfiguriranja ROS-a na različitim operativnim sustavima te ARM razvojnim sustavima koji nam je potreban za pokretanje robota na mikroračunalu. Postupak odabira verzije ROS-a, koji će se instalirati na operativni sustav, ovisi isključivo o vrsti i verziji operativnog sustava. Najstarija stabilna i najbolje testirana verzija ROS-a je ROS Kinetic Kame koja je službeno podržana samo na Ubuntu distribucijama Willy (Ubuntu 15.10), Xenial Xerus (Ubuntu 16.04) te Jessie (Debian 8). Verzija Linux distribucije instalirana na laptopu je Ubuntu 16.04.4 Xenial Xerus čime je omogućeno da se instalira verzija ROS Kinetic. Odorid je podešen na isti način prateći upute sa službene ROS stranice, gdje je instalirana verzija Linuxa s distribucijom Ubuntu Mate 16.04 te je uspješno postavljen ROS.

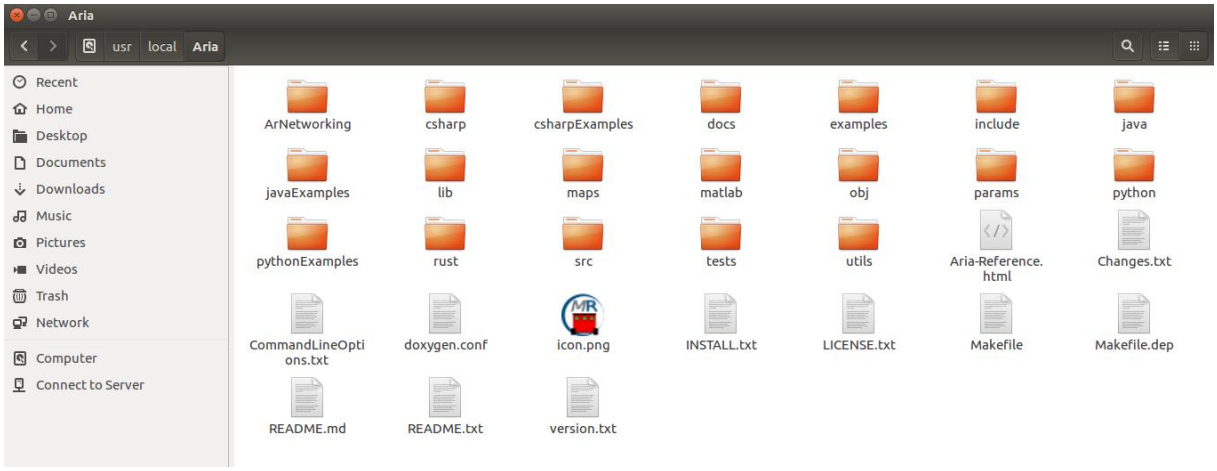
Kada se ROS prvi puta konfigurira na računalu važno je odraditi sve korake koji pišu u uputama na službenoj ROS stranici te je najbitnije da se inicijalizira radni prostor ROS-a. Inicijalizacija ROS radnog prostora radi se tako da se uđe u jednu novu praznu datoteku i upiše se naredba *catkin_init_workspace*. Datoteka u kojoj se kreira radni prostor često se imenuje *catkin_ws* (*eng. catkin workspace*) u kojem će se odvijati kasnije razvoj različitih paketa, čvorova i projekata. ROS softver gradi se od paketa gdje se svaki dio izgrađuje pomoću *catkin* sustava. Sustav *catkin* sve pakete povezuje kao kombinaciju CMake makronaredbe i različitih Python skripti. CMake je sustav otvorenog tipa koji se koristi za povezivanje dijelova programa u jedinstvenu sustavnu i programsku cjelinu. ROS nije vrsta okuženja kao Microsoft Visual Studio gdje se pritiskom na tipku za izgradnju programa sve poveže i kreira se program. Ovdje se od korisnika zahtjeva razumijevanje sustava *catkin* koji je službeni sustav za izgradnju ROS sustava.

6.2. Integracija ARIE i ROSARIE

Drugi korak potreban za povezivanje je integracija sustava ARIA i njena inačica u obliku ROS paketa koji se zove ROSARIA. Normalna funkcionalnost ROSARIE se ostvaruje jedino ako je sustav ARIA instaliran na uređaju na kojem se pokreće čvor ROSARIA. Integracija sustava ARIA i paketa ROSARIA je napravljena na laptopu, a zatim i na mikroračunalu. Problem s instalacijom ARIA-e je što nije postojao postupak instalacije niti mjesto s kojeg se može dohvatiti ARIA paket jer je tvrtka ActivMedia Robotics, koja je proizvodila Pioneer mobilne robote, zatvorena 2018. godine. Nakon zatvaranja tvrtke, ugašena je bila i stranica na kojoj su bili svi potrebni koraci i datoteke za upravljanje sustavom ARIA. Nasreću, korisnici su uspjeli sačuvati dio softverskih paketa i uputa te su ih arhivirali na jednom mjestu.

6.2.1. Integracija na računalo

Integracija ARIE na laptopu je prošla bez velikih problema. Skinuta je instalacijska datoteka te je koristeći Ubuntu instalacijski sustav, ARIA uspješno instalirana. Validiranje instalacije radi se tako da se provjeri prisutnost ARIE unutar datoteke smještene na putanji */usr/local/Aria*. Ako se unutar datoteke na našem računalu nalaze datoteke jednake onima na slici 27. znači da je proces integracije sustava ARIA bio uspješan. Verzija ARIE koja je konfigurirana je 2.9.4.



Slika 33. Izgled ARIA datoteke nakon uspješne integracije

Idući korak bio je preuzeti paket ROSARIA sa službene ROS stranice. Jedna od glavnih prednosti ROS-a je distribucija softverskih paketa među članovima i dobra dokumentacija istih.

Za verziju ROS Kinetic postoje, kao i za instalaciju ROS-a, detaljne upute kako preuzeti, izgraditi i podesiti paket ROSARIA. Jedan od navedenih koraka je da se preuzme ARIA sa službene stranice ActivMedia Robotics i njega treba preskočiti jer smo već integrirali ARIU na naše računalo. Preuzimanje ROSARIE se radi tako da se ode u datoteku *src* i unutra se koristeći naredbu *git clone* preuzme ROSARIA paket. Nakon toga se izgradi paket pomoću naredbe *catkin_make* u novom Terminal prozoru iz datoteke gdje smo inicijalizirali ROS radni prostor što je kod nas na putanji */home/user/catkin_ws*. Ako je povezivanje dijelova softvera prošlo bez grešaka, ali uz neka upozorenja, znači da je ROSARIA uspješno instalirana na računalu.

6.2.2. Integracija na Odroid XU-4

Postupak instalacije na ARM razvojni sustav nije tako jednostavan kao kod integracije na računalo. Prvi problem je što je ARIA bila namijenjena za operacijske sustave Windows i Ubuntu, ali ne i ARM operacijske sustave. Drugi problem je što se nije moglo kontaktirati tvrtku koja je radila na razvoju ARIE, jer je kako je navedeno prije, tvrtka prestala raditi 2018. godine te je svaki pokušaj da se dobije binarna datoteka bio nemoguć. Ako bi imali binarnu datoteku, integracija bi se mogla odraditi ručno tako da se natjera sustav da poveže dijelove ARIA sustava i instalira ju unutar putanje */usr/local/Aria*. Nakon nekoliko neuspjelih pokušaja, primijenjeno je sljedeće rješenje. Korištena je ista instalacijska datoteka kao i prilikom integracije na računalo s verzijom ARIE 2.9.4 te navedeni koraci:

1. sudo dpgk –force-all -i libaria_2.9.4+ubuntu16_amd64.deb
2. cd usr/local/Aria
3. make clean
4. make -j4
5. cd /catkin_ws
6. catkin_make –force-cmake

Prvi korak je da se prisilno instalira ARIA na mikroračunalo. Zatim se uđe u mjesto gdje je instalirana ARIA na putanji *usr/local/Aria* i otvori se novi terminalni prozor. Sada se upiše naredba *make clean* da se očisti od svih prijašnjih verzija ARIE, ako one postoje te se pomoću *make -j4* zada mikroračunalu da pokuša povezati sve dijelove sustava ARIA. Ovaj proces će potrajati neko vrijeme i izbacit će puno upozorenja (*eng. warning*), no bitno je da se ne prikaže niti jedna greška (*eng. error*). Ako se pojavi greška, instalacija će se odmah zaustaviti. Kada taj

dio završi, odlazimo naredbom *cd* (*eng. change directoy*) u ROS radni prostor i tamo moramo upisati naredbu 6. tako da se prisili povezivanje svih dijelova koji do tada nisu povezani s naredbom *make clean.* Ako i ovdje sve prođe bez grešaka uz pokoje upozorenje, znači da je ARIA uspješno integrirana na mikroračunalu. ARIA izvorno nije bila napisana za ARM operativne sustave tako da uvijek postoji mogućnost da neki dijelovi neće funkcionirati kako bi trebali. Na kraju možemo skinuti i integrirati ROSARIU kako smo napravili i kod podešavanja na računalo.

6.3. Kamera

Greyp Kamera povezuje se s USB-om na mikroračunalo mobilnog robota. Upravljanje kamerom preko ROS-a omogućeno je odgovarajućim driverima za USB kameru. Preuzeti driver *usb_cam* sa službene ROS stranice pruža korištenje velikog dijela USB kamera te mijenjanje parametara na kameri. Čvor koji se pokreće u ROS-u kada se krene koristiti kamera iz ROS-a objavljuje na temu */usb_cam/image_raw*. Naredba za automatsko preuzimanje i instalaciju drivera je *sudo apt-get install ros-kinetic-usb-cam.*

6.4. Senzori

Sustav od 8 sonarnih senzora raspoređen je na luku u rasponu od 180º. Senzori posjeduju posebnu elektroniku koja omogućava da svi senzori mogu poslati i primiti informacije iz okoline. Elektronika senzora direktno je povezana mikrokontrolerom preko jedne sabirnice. Softversko povezivanje senzora i mobilnog robota odrađuje se preko naredbi koje se šalju i primaju iz mikrokontrolera, a koji iz ROS-a dobiva naredbe za primanje informacije sa senzora pomoću ROSARIE.

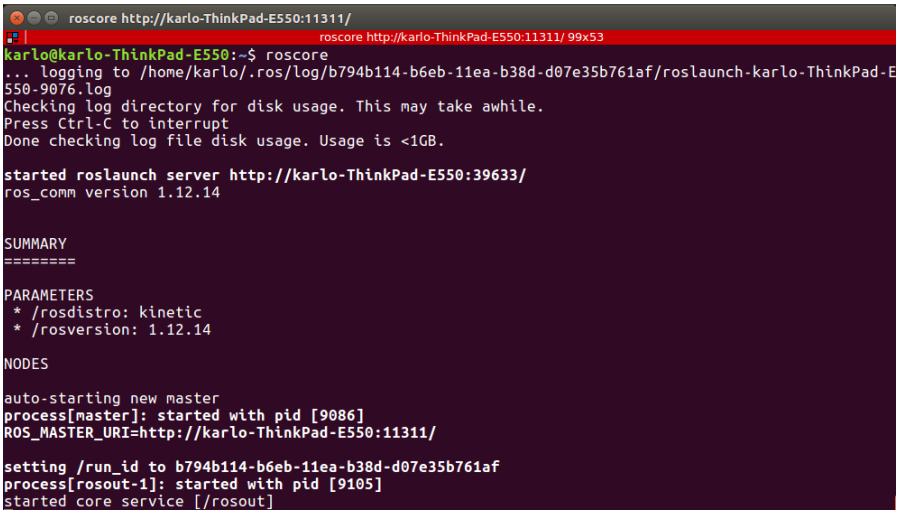
6.5. Kontroleri za motore

Motori i enkoderi povezani su direktno na pločicu za drivere za motore. Kontroleri za motore su isto kao i senzori povezani direktno na mikrokontroler iz kojeg dobivaju naredbe kada da se šalju informacije pojedinom motoru. Podaci s enkodera se čitaju kao i senzori, oni se šalju u mikrokontroler koji je povezan s ROS-om na mikroračunalu preko ROSARIE. Motori dobivaju napajanje od 12V s pločice koja se koristi za upravljanje motorima. Enkoderi imaju dva kanala A i B te pozitivni i negativni dio koji se spaja na istu tu pločicu. Sva regulacija kontrole motore s odgovarajućom elektronikom preuzeta je od prošlog robota kako bi sustav mogao pravilno biti upravljan preko ARIE.

7. IZRADA PROGRAMSKE PODRŠKE

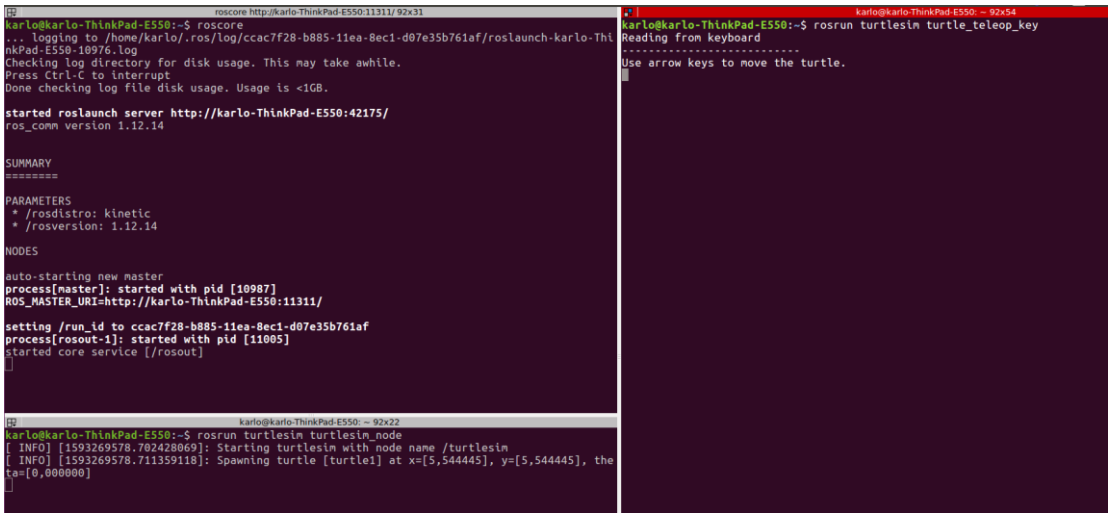
Program za upravljanje dijelovima robotskog sustava razvijan je pomoću ROS-a tako da su procesi napisani u obliku čvorova u ROS-u koji se mogu naizmjenično pokrenuti ili isključiti. Čvorovi se nalaze u jednom ROS paketu gdje su definirane sve njihove zavisnosti o potrebnim bibliotekama da se ostvari željena funkcionalnost i upravljanje mobilnog robota. Čvorovi mogu biti napisani pomoću programskih jezika C++ ili Python. Čvorovi napisani različitim programskim jezicima funkcioniraju na jednaki način. Procesi dohvaćanja i prikazivanja vrijednosti sa senzorskih sustava, upravljanja kamerom i obrada slike te kretanjem robota predstavljaju pojedinačne čvorove. Ovakav način odvajanja dijelova robotskog sustava pomoću čvorova omogućuje lako uočavanje i ispravljanje grešaka koje se mogu pojaviti. Svi čvorovi ROS sustava povezani su preko ROS Mastera koji omogućuje da čvorovi mogu međusobno komunicirati i izmjenjivati informacije. U ovom poglavlju prikazat ćemo prvo kako se prikupljaju informacije sa senzora i odometrijski podaci, nakon toga kako je implementiran vizijski sustav te napravljena obrada slike i na kraju će se testirati osnovne funkcionalnosti kretanja robota na ROS simulatoru Turtlesim što će se primijeniti i za testiranje kretnji na stvarnom mobilnom robotu.

Robotski operativni sustav se pokreće preko komandnog korisničkog sučelja (*CLI, eng. Command Line Interface*) gdje se interakcija računalnog programa i korisnika odvija preko komandnog prozora u obliku uzastopnih linija teksta. Pokretanje ROS-a odvija se otvaranjem novog komandnog Terminal prozora u kojemu se napiše naredba *roscore* i dolazi do pokretanja ROS sustava. Uspješno pokretanje ROS sustava prikazano je na slici 34.



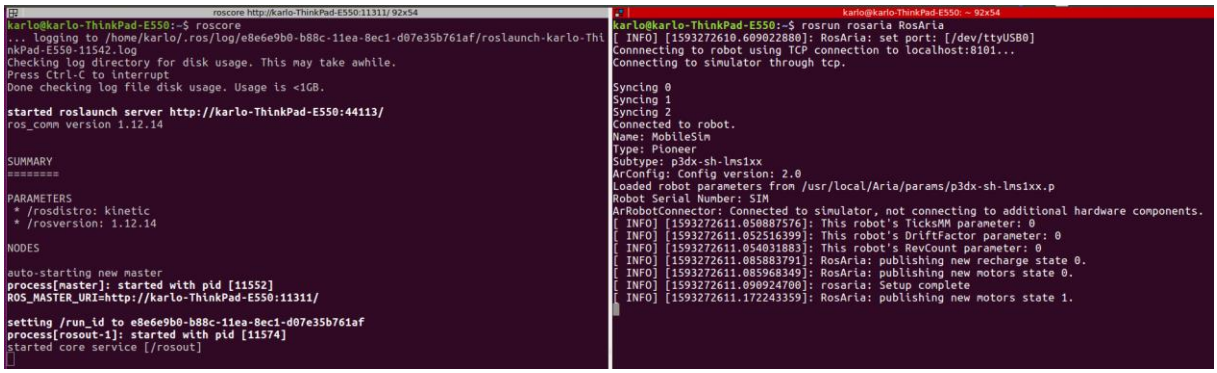
Slika 34. Pokretanje ROS sustava

Na slici 34. vidimo da se ispisuju distribucija i verzija ROS-a te IP adresa ROS Mastera koji se uvijek mora pokrenuti kada pokrene ROS. Ako se ROS Master ne inicijalizira na početku znači da postoji problem s konfiguracijom IP adresa pojedinih čvorova ili da je u nekom Terminal prozoru već ROS sustav pokrenut. Stoga, kada se jednom pokrene ROS s naredbom *roscore* više ga ne trebamo pokretati nego taj terminalni prozor ostavljamo zasebnog i ne smijemo ga izgasiti.



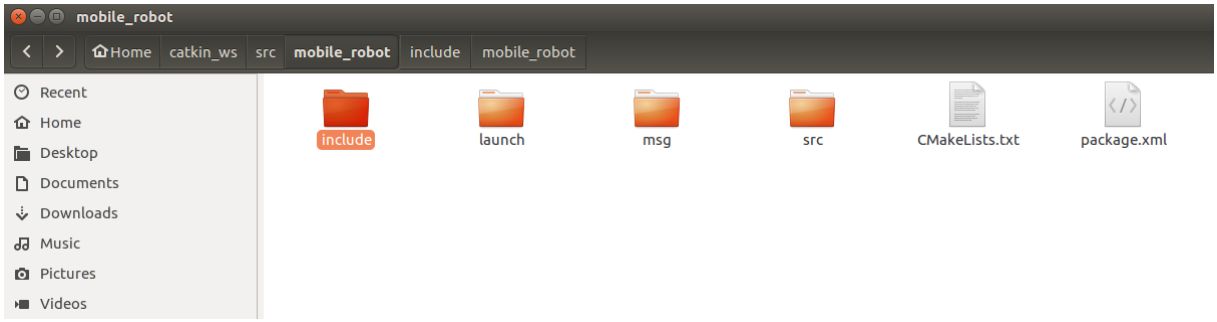
Slika 35. Otvaranje novih Terminal prozora za pokretanje različitih čvorova

RosAria je čvor koji pokrećemo u novom Terminal prozoru nakon što se pokrene ROS sustav. Pokretanje RosAria služi za testiranje ispravnosti instalacije ARIE. Ako je ARIA uspješno instalirana ispisat će se ekran kao na slici 36. gdje se prikazuje pokretanje čvora pomoću novog prozora naredbom *rosrun* nakon koje se navodi ime paketa koji se pokreće *rosaria* te ime čvora koji se pokreće *RosAria*. Također, nakon što se uspješno pokrenula RosAria, dokazano je da je ARIA uspješno integrirana jer se inače čvor RosAria ne bi mogao pokrenuti i prikazala bi se greška.



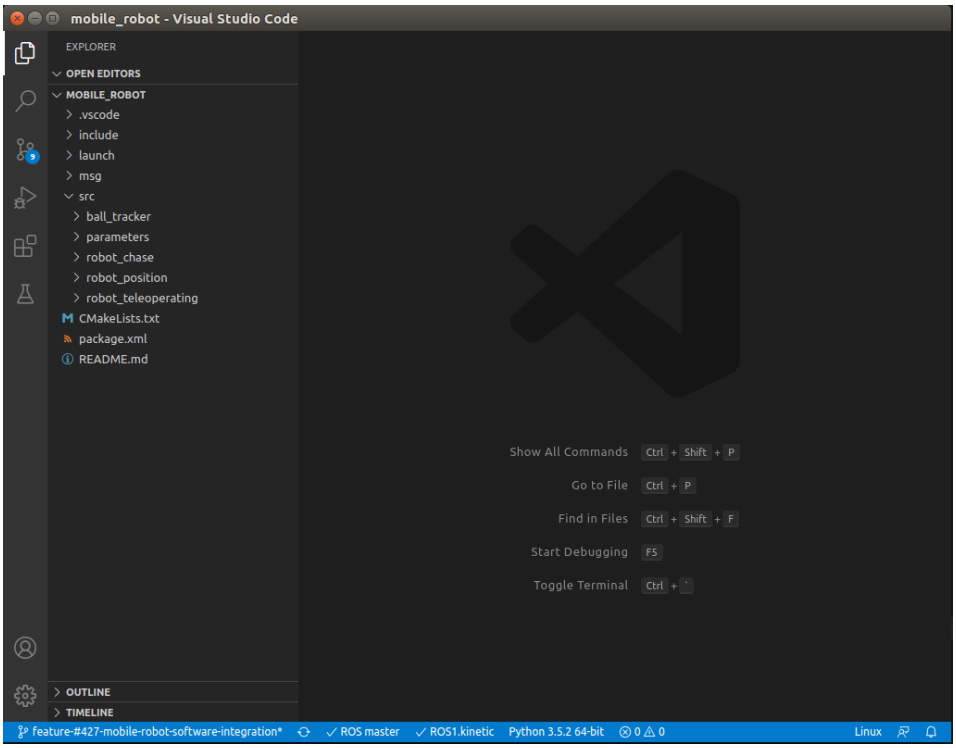
Slika 36. Uspješno pokretanje čvora RosAria

Uobičajeni način kreiranja programskog dijela u ROS-u je da se kreira jedan paket u koji će biti smješteni svi dijelovi sustava potrebni za mobilni robot. Kreiran je paket pod imenom *mobile_robot* u kojem se nalaze svi dijelovi našeg programskog dijela mobilnog robota. Način organizacije paketa jednak je klasičnom načinu koji se primjenjuje u ROS-u. Unutar datoteke *src* stavljeni su dijelovi programskog koda, *msg* kreirane nove poruke potrebne za razmjenu informacija, *launch* datoteka koja služi za pokretanje više čvorova odjednom te *include* datoteka u kojoj se nalaze potrebne biblioteke korištene za uspješno funkcioniranje pojedinih čvorova. Također, imamo i CmakeLists.txt i package.xml koji sadrže informacije o paketu te postupke povezivanje dijelova koda u jedan program. Na slici 37. prikazan je paket *mobile_robot.*



Slika 37. Struktura paketa za izradu programske podrške mobilnog robota

Programski kod pisan je pomoću editora za kod Visual Studio Code – VSCode. Visual Studio Code nastao je zbog potrebe za distribucijom koda na različitim vrstama operativnih sustava poput Windows, Linux i MacOS. Razvijen je iz programskog okruženja Microsoft Visual Studio od strane tvrtke Microsoft. VSCode je vrsta naprednog tekstualnog editora sličan općepoznatom Notepadu koji posjeduje puno više stvari od Notepad-a. Značajke uključuju podršku za otklanjanje grešaka, osjenčavanje sintakse, automatsku dopunu koda, refaktoriranje koda i ugrađeni sustav za verzioniranje koda Git. Korisnici mogu promijeniti temu, dodati razne kratice, preference i instalirati dodatne ekstenzije koje ga čine boljim od osnovne verzije. VSCode je relativno nova vrsta programskog editora i sve se više koristi u širokoj programerskoj zajednici. Može se koristiti za programiranje velikog broja programskih jezika od Jave, Javscripta, Go, Node.js pa sve do Pythona i C++. Navedene prednosti su razlog korištenja VSCode-a za razvoj programske podrške jer posjeduje podršku za pisanje ROS programa, te čvorova koji će biti napisani u C++.



Slika 38. Izgled programskog editora Visual Studio Code

7.1. Senzorski sustav

Senzorski sustav iskorišten je od postojećeg dijela sustava Pioneer 2-DX mobilnog robota. ARIA čita i obrađuje informacije koje se dobiju s pojedinog senzora te ih oblikuje u podatke s kojima se može raditi daljnja obrada. ROSARIA uzima te informacije i objavljuje ih na temu pod imenom */RosAria/sonar*. ROSARIA čvor objavljuje podatke te je za čitanje informacija s navedene teme potrebno kreirati čvor koji će se pretplatiti na tu temu i prikazati podatke koji dolaze sa senzora. Dakle, imamo dva čvora od kojih je jedan koji objavljuje (eng. publisher), a drugi je koji se pretplaćuje na temu */RosAria/sonar* što je prikazano na slici 39. Naredbom *rostopic info /RosAria/sonar* dobivamo informacije koliko je čvorova povezano s temom.

```
karlo@karlo-ThinkPad-E550:~$ rostopic info /RosAria/sonar
Type: sensor_msgs/PointCloud

Publishers:
 * /RosAria (http://karlo-ThinkPad-E550:43293/)

Subscribers:
 * /sensor_test (http://karlo-ThinkPad-E550:40083/)
```

Slika 39. Informacije o temi preko koje čitamo informacije o senzorima

Prije nego se krene u pisanje programskog koda za čitanje s teme, potrebno je detaljno proučiti strukturu poruke u kojoj su sadržani podaci koje želimo pročitati. Koristeći naredbu iz ROS-a *rosmsg show sensor_msgs/PointCloud* prikazuje se detaljna struktura poruke. Podaci koje želimo pročitati su vrijednosti koordinata x i y koje predstavljaju vrijednosti u koordinatnom sustavu referentnom na svaki senzor. Na slici 40. je prikaz izgleda strukture poruke *sensor_msgs/PointCloud*.

```
karlo@karlo-ThinkPad-E550:~$ rosmsg show sensor_msgs/PointCloud
std_msgs/Header header
  uint32 seq
  time stamp
  string frame_id
geometry_msgs/Point32[] points
  float32 x
  float32 y
  float32 z
sensor_msgs/ChannelFloat32[] channels
  string name
  float32[] values
```

Slika 40. Struktura poruke sensor_msgs/PointCloud

Kada se saznaju sve informacije potrebne za čitanje senzorskih podataka, može se krenuti u pisanje programskog koda odnosno kreiranje ROS čvora. Kreiran je čvor pod imenom *sensor_test.cpp* u kojem se nalaze dijelovi potrebni za čitanje informacija s teme na koju se objavljuju senzorski podaci. Prvi korak je uključiti potrebne biblioteke čiji dijelovi će se koristiti za pisanje programskog koda. Korištene su četiri biblioteke, a to su *iostream* koja je standardna knjižica za ulazne i izlazne objekte, *ros/ros.h* koja nam pruža mogućnost korištenja ROS sustava, poruka *sensor_msgs/PointCloud.h* preko koje ćemo čitati podatke i na kraju *math.h* za rad s matematičkim računskim operacijama. Poruka iz koje želimo čitati podatke mora se uključiti na početku pisanja programskog koda kako bi se moglo pristupiti podacima iz te poruke prilikom pokretanja čvora.

```
#include <iostream>
#include <ros/ros.h>
#include <sensor_msgs/PointCloud.h>
#include <math.h>
```

Funkcija *callbackSonar()* prima argument *msg* koji je objekt ROS poruke preko kojeg će se raditi operacije za čitanje vrijednosti sa senzora. Udaljenosti sa senzora se dobiva tako da se

pročitaju koordinate x i y, koje predstavljaju vrijednosti u koordinatnom sustavu referentnom na mobilni robot, te se kvadriraju i korjenuju radi dobivanja udaljenost robota od predmeta u prostoru. Prvo inicijaliziramo varijable na vrijednost nula kako bi svako mjerenje krenulo pravilno, nakon toga u varijable *distance0* do *distance7* spremamo vrijednosti udaljenosti preko x i y koordinate.

```
void callbackSonar(const sensor_msgs::PointCloud::ConstPtr &msg) {
    ROS_INFO("Started reading sensor data" );

    float distance0 = 0.0;
    float distance1 = 0.0;
    float distance2 = 0.0;
    float distance3 = 0.0;
    float distance4 = 0.0;
    float distance5 = 0.0;
    float distance6 = 0.0;
    float distance7 = 0.0;

    distance0 = sqrt(pow(msg->points[0].x, 2) + pow(msg->points[0].y, 2));
    distance1 = sqrt(pow(msg->points[1].x, 2) + pow(msg->points[1].y, 2));
    distance2 = sqrt(pow(msg->points[2].x, 2) + pow(msg->points[2].y, 2));
    distance3 = sqrt(pow(msg->points[3].x, 2) + pow(msg->points[3].y, 2));
    distance4 = sqrt(pow(msg->points[4].x, 2) + pow(msg->points[4].y, 2));
    distance5 = sqrt(pow(msg->points[5].x, 2) + pow(msg->points[5].y, 2));
    distance6 = sqrt(pow(msg->points[6].x, 2) + pow(msg->points[6].y, 2));
    distance7 = sqrt(pow(msg->points[7].x, 2) + pow(msg->points[7].y, 2));
```

Na kraju funkcija ispisuje vrijednost koje su pročitane iz ROS poruke, te preoblikovane za bolje razumijevanje udaljenosti predmeta što je prikazano u sljedećem dijelu koda. Osam je sonarnih senzora na mobilnom robotu i napravljeno je osam istih čitanja sa senzora kako bi se dobili podaci koje očitavaju svaki od osam senzora.

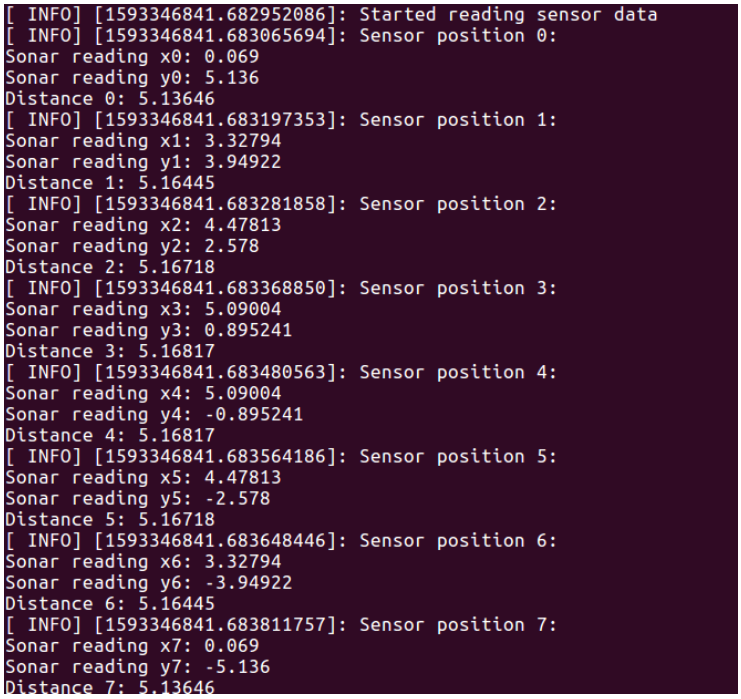
```
    ROS_INFO("Sensor position 0: ");
    std::cout << "Sonar reading x0: " << msg->points[0].x << std::endl;
    std::cout << "Sonar reading y0: " << msg->points[0].y << std::endl;
    std::cout << "Distance 0: " << distance0 << std::endl;
```

Svaki ROS čvor sastoji se od jedne *main* funkcije u kojoj se prvo mora inicijalizirati čvor pomoću ROS funkcije *ros::init* u kojem se navodi ime čvora kako bi ROS kasnije znao koji čvor želimo pokrenuti. Nakon toga se kreira *ros::NodeHandle* objekt koji je središnji dio za komunikaciju s ROS sustavom. Pozivom *ros::Subscriber* je način kako se kaže ROS-u da želimo primati poruke s definirane teme što je ključno kako bi ROS Master kasnije znao koji čvor objavljuje, a koji se pretplaćuje. Na kraju funkcija *ros::spin()* se koristi da se prilikom

pokretanja čvora podaci cijelo vrijeme dohvaćaju te stvara vrstu beskonačne petlje dok god se ne prekine ROS Master ili se prekine čvor.

```
int main(int argc, char **argv)
{
    ros::init(argc, argv, "sensor_test");
    ros::NodeHandle nodeHandle;
    ros::Subscriber sonarSub = nodeHandle.subscribe("/RosAria/sonar", 1, callbackSonar);
    ros::spin();
}
```

Prije testiranja čvora, potrebno je izgraditi čvor tako da se ode u datoteku na putanji *home/user/catkin_ws/*, otvori novi Terminal prozor i napiše naredba *catkin_make*. Nakon uspješno kreiranog čvora, uz otvoren ROS u jednom prozoru i RosAria čvor u drugom, otvara se treći čvor s naredbom rosrun mobile_robot sonar_test. Ako su svi dijelovi dobro postavljeni ispisat će se ispis kao na slici 41.



Slika 41. Prikupljeni podaci sa senzorskih sustava i ispisani u Terminal prozoru

Sa slike 41. vidimo da su prikazani skoro isti podaci očitavanja na svim senzorima te da su to maksimalne vrijednosti senzora. To znači da je robot stavljen u okolinu u kojoj nema predmeta u blizini robota. Vrijednosti koje su ispisane prikazane su u metrima što znači da je maksimalni raspon sonarnih senzora detektiranje predmeta na udaljenosti do pet metara. Regulacija udaljenosti predmeta može se kontrolirati preko potenciometra na elektronici za senzore.

7.2. Odometrijski podaci

Odometrija je postupak korištenja podataka sa senzora pokreta radi utvrđivanja promjene pozicije u proteklom vremenu s obzirom na prijašnju poznatu poziciju. To je osnovna metoda kojom se mjeri udaljenost koju je prešao mobilni robot kako bi se moglo primijeniti željeno upravljanje. Odometrijski podaci izračunavaju se direktno preko mikrokontrolera koji primaju informacije s rotacijskih enkodera. Odometrija je česti senzor pozicije koji se koristi kod mobilnih robota, ali ima svoja ograničenja. Budući da je da se mjeri kumulativno, bilo koja greška će se povećavati kako vrijeme prolazi. Izvori greška koje se mogu pojaviti su nepravilno mjerenje promjera kotača, različita veličina kotača, pogreške koje se javljaju prilikom mjerenja pulsa enkodera te sporo procesiranje odometrije.

Odometrijski podaci s mobilnog robota Pioneer 2DX dobivaju se direktno iz inkrementalnih enkodera koji su smješteni zajedno s motorima. Enkoderi su električno mehanički uređaji koji pretvaraju gibanje vratila motora u informacije koje šalju u mikrokontroler robota. Enkoderi šalju informacije o gibanju vratila motora koje se onda u P2OS mikrokontroleru pretvara u mjerenje pozicije, brzine i udaljenosti. ARIA je odgovorna za prikupljanje podataka iz mikrokontrolera koje onda šalje ROSARIA-i te ih možemo prikazati.

Postupak čitanja i testiranja prikupljenih odometrijskih podataka sličan je načinu koji je korišten prilikom čitanja sa senzorskih sustava. ROSARIA čvor objavljuje na temu */RosAria/pose* za koji je potrebno kreirati čvor koji će se pretplatiti na temu i dohvatiti informacije. Podatke koje možemo pročitati s mobilnog robota su pozicija u obliku x i y koordinata, orijentacija oko vertikalne osi z te linearnu i kutnu brzinu robota. Upisivanjem naredbe *rostopic info /RosAria/pose* dobivaju se informacije o čvorovima koji objavljuju podatke te koji se pretplaćuju na temu */RosAria/pose* prikazani su na slici 42.

```
karlo@karlo-ThinkPad-E550:~$ rostopic info /RosAria/pose
Type: nav_msgs/Odometry

Publishers:
 * /RosAria (http://karlo-ThinkPad-E550:43293/)

Subscribers:
 * /pose_robot (http://karlo-ThinkPad-E550:46041/)
```

Slika 42. Informacije o čvorovima koji objavljuju informacije a koji se pretplaćuju

ROS poruka u kojoj se nalaze odometrijski podaci je predefinirana poruka ROS-a *nav_msgs/Odometry*. Sastoji se od četiri manje poruke, a to su *std_msgs/Header*, string *child_frame_id*, *geometry_msgs/PoseWithCovariance* i *geometry_msgs/TwistWithCovariance*. Poruka *std_msgs/Header* sadrži trenutno vrijeme, *child_frame_id* je ime okvira u kojem se nalaze informacije u vremenu neposredno prije sadašnjeg. Pozicija i orijentacija robota nalaze se u *geometry_msgs/PoseWithCovariance* tako da je pozicija zapisana u drugoj standardnoj poruci *geometry_msgs/Pose pose* u x, y i z koordinatama, a orijentacija u kvaternijskom načinu zapisa. Linearna i kutna brzina dio su poruke *geometry_msgs/TwistWithCovariance* u kojoj su zapisane pomoću vektora. Na slici 43. može se vidjeti struktura poruke iz koje se dobivaju odometrijski podaci.

```
karlo@karlo-ThinkPad-E550:~$ rosmsg show nav_msgs/Odometry
std_msgs/Header header
  uint32 seq
  time stamp
  string frame_id
string child_frame_id
geometry_msgs/PoseWithCovariance pose
  geometry_msgs/Pose pose
    geometry_msgs/Point position
      float64 x
      float64 y
      float64 z
    geometry_msgs/Quaternion orientation
      float64 x
      float64 y
      float64 z
      float64 w
  float64[36] covariance
geometry_msgs/TwistWithCovariance twist
  geometry_msgs/Twist twist
    geometry_msgs/Vector3 linear
      float64 x
      float64 y
      float64 z
    geometry_msgs/Vector3 angular
      float64 x
      float64 y
      float64 z
  float64[36] covariance
```

Slika 43. Struktura poruke za odometrijske podatke

Proces izrade čvora za čitanje odometrijskih podataka sličan je procesu čitanja podataka sa senzorskog sustava. Prvo definiramo sve potrebne biblioteke gdje je i poruka koju želimo čitati. Nakon toga kreiramo strukture podataka u koje predstavljaju skup varijabli u koje ćemo pohranjivati informacije. Ovdje su korištene strukture podataka jer je lakše grupirati varijable s istim imenom jer imamo x, y i z kao elemente vektora kod pozicije, orijentacije, linearne te kutne brzine. Također, kreiramo i objekt koji će služiti kao čvor koji se pretplaćuje.

```
#include <ros/ros.h>
#include <geometry_msgs/Twist.h>
#include <geometry_msgs/PoseWithCovariance.h>
#include <geometry_msgs/Pose.h>
#include <nav_msgs/Odometry.h>
#include <cmath>

ros::Subscriber pose_subscriber;
struct Position {
    double x, y, z;
};
struct Quaternion {
    double w, x, y, z;
};
struct EulerAngles {
    double roll, pitch, yaw;
};
struct LinearSpeed {
    double x, y, z;
};
struct AngularSpeed {
    double x, y, z;
};
```

Kreirana je i pomoćna funkcija *radians2degrees()* kako bi pretvorili radijane u stupnjeve radi lakšeg razumijevanja o tome kako je orijentiran robot. Ova funkcija će se koristiti poslije kod postupka rotiranja robota za željeni odnosno definirani kut zakreta.

```
double radians2degrees(double angle_in_radians){
    return (angle_in_radians *180.0) /PI;
}
```

Iduće je kreiran poseCallback() funkcija koja čita podatke sadržane u objektu *pose_message*. Čitanje podataka iz objekta jednako je čitanju podataka iz poruke sa sonarnih senzora. Unutar funkcije prvo inicijaliziramo objekte u koje ćemo spremati pročitane podatke.

```
void poseCallback(const nav_msgs::Odometry::ConstPtr & pose_message) {
    Position p;
    Quaternion q;
    EulerAngles angles;
    LinearSpeed linear;
    AngularSpeed angular;
```

```
p.x = pose_message->pose.pose.position.x;
p.y = pose_message->pose.pose.position.y;
p.z = pose_message->pose.pose.position.z;

q.x = pose_message->pose.pose.orientation.x;
q.y = pose_message->pose.pose.orientation.y;
q.z = pose_message->pose.pose.orientation.z;
q.w = pose_message->pose.pose.orientation.w;

linear.x = pose_message->twist.twist.linear.x;
linear.y = pose_message->twist.twist.linear.y;
linear.z = pose_message->twist.twist.linear.z;

angular.x = pose_message->twist.twist.angular.x;
angular.y = pose_message->twist.twist.angular.y;
angular.z = pose_message->twist.twist.angular.z;
```

Ovdje je uočljivo zašto su strukture podataka bile korisne jer možemo razlikovati koja komponenta x, y ili z pripada kojem vektoru. Problem kod razumijevanja podataka orijentacije robota je što je u ROS poruci bio kvaternion zapis kutova, a smisao je da imamo Eulerov način prikaza kutova. Kvaternion zapis je dobar kada imo rotaciju oko sve tri osi koordinatnog sustava. Kako mobilni robot ima rotaciju samo oko vertikalne osi, osi z, bolje je koristiti Eulerov zapis. Kvaternion zapis prikazan je sljedećim vektorom:

$$q = \begin{bmatrix} q_0 \\ q_1 \\ q_2 \\ q_3 \end{bmatrix}, \tag{12}$$

Relacija za pretvaranje iz kvaternion u Eulerove kuteve prikazana je gotovom formulom [19]:

$$q = \begin{bmatrix} arctan \frac{2(q_0 q_1 + q_2 q_3)}{1 - 2({q_1}^2 + {q_2}^2)} \\ \arcsin\left(2(q_0 q_2 + q_1 q_3)\right) \\ arctan \frac{2(q_0 q_3 + q_1 q_2)}{1 - 2({q_2}^2 + {q_3}^2)} \end{bmatrix}, \tag{13}$$

Prilikom implementiranja formula za transformaciju treba odvojiti funkcije *arctan* i *arcsin* u dvije jer većina programskih jezika ima mogućnost prikazati rezultate od -π/2 do π/2. Za tri rotacije između tih vrijednosti nije moguće prikazati sve orijentacije. Stoga, u programskom kodu treba implementirati *atan2* funkciju na sljedeći način:

$$q = \begin{bmatrix} \text{atan2}(2(q_0 q_1 + q_2 q_3)),\ 1 - 2({q_1}^2 + {q_2}^2) \\ \arcsin\left(2(q_0 q_2 + q_1 q_3)\right) \\ \text{atan2}(2(q_0 q_3 + q_1 q_2)), 1 - 2({q_2}^2 + {q_3}^2) \end{bmatrix}, \tag{14}$$

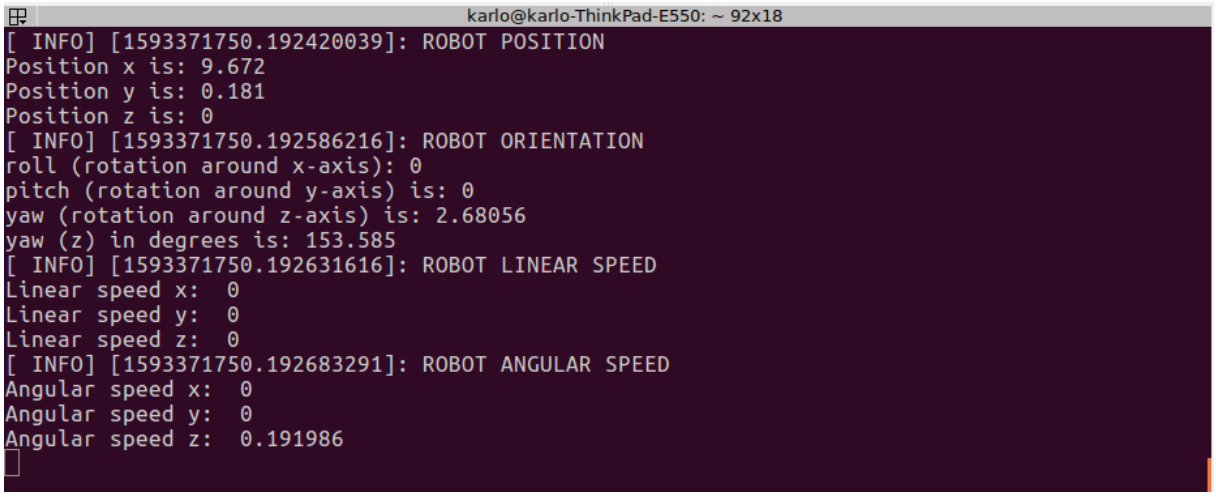
Implementacija navedenih jednadžbi u programskom kodu izgleda ovako:

```
    // roll (x-axis rotation)
    double sinr_cosp = 2 * (q.w * q.x + q.y * q.z);
    double cosr_cosp = 1 - 2 * (q.x * q.x + q.y * q.y);
    angles.roll = std::atan2(sinr_cosp, cosr_cosp);

    // pitch (y-axis rotation)
    double sinp = 2 * (q.w * q.y - q.z * q.x);
    if (std::abs(sinp) >= 1)
        angles.pitch = copysign(PI / 2, sinp); // use 90 degrees if out of range
    else
        angles.pitch = std::asin(sinp);

    // yaw (z-axis rotation)
    double siny_cosp = 2 * (q.w * q.z + q.x * q.y);
    double cosy_cosp = 1 - 2 * (q.y * q.y + q.z * q.z);
    angles.yaw = std::atan2(siny_cosp, cosy_cosp);
```

Kako bi se napravila validacija pročitanih odometrijskih podataka, napravljen je ispis u Terminal prozoru. Na kraju imamo *main()* funkciju u kojoj se naprave isti koraci kao i za čitanje senzora samo se promijeni ime čvora u *pose_robot* i tema */RosAria/pose* na koju će se pretplatiti taj čvor. Ovdje je dokazana jednostavnost uporabe i integracije ROS-a za čitanje podataka pomoću čvorova.

```
int main(int argc, char* argv[])
{
    ros::init(argc, argv, "pose_robot");
    ros::NodeHandle n;
    pose_subscriber = n.subscribe("/RosAria/pose", 10, poseCallback);
    ros::spin();
}
```

Nakon kreiranja svakog novog čvora ili dijela upravljanja, izgradnja programskog sustava radi se pomoću *catkin_make* naredbe. Ispravnost pročitanih odometrijskih podatka napravljena je tako da se robot kreće u slobodnom prostoru, a podaci se proučavaju na Terminalu što je prikazano na slici 44. Može se zaključiti da dobiveni odometrijski podaci prikazuju dovoljno dobre vrijednosti koje će se kasnije testirati kroz upravljanje robota. Također, postoji mogućnost od pojavljivanja grešaka jer prilikom gubitka napajanja inkrementalni enkoderi ne zadržavaju poziciju robota u kojoj je bio prije isključivanja. Jedna od mogućnosti je da se programskim kodom spremaju odometrijski podaci pročitani s mikrokontrolera.



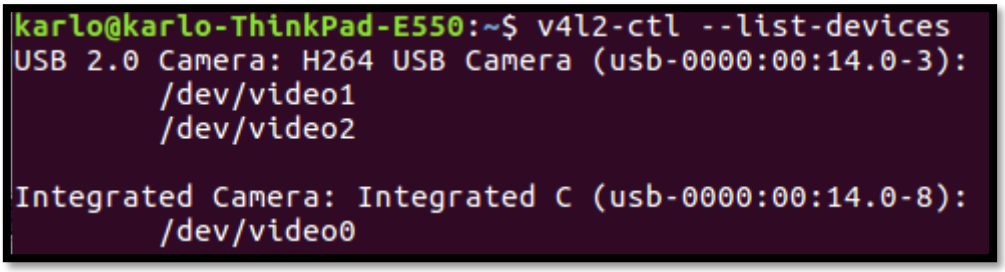
Slika 44. Prikaz prikupljenih odometrijskih podataka

7.3. Vizijski sustav

Vizijski sustav na mobilnom robotu je jedan bitan dio robotskog sustava koji se danas koristi za manipulaciju objektima i navigaciju. Danas na tržištu postoji niz dvodimenzionalnih i trodimenzionalnih vizijskih senzora te ih većina ima potreban softverski driver za interakciju s ROS sustavom. Ovdje ćemo prikazati način integracije Greyp kamere s odgovarajućim softverom u ROS-u. Prikazat ćemo kako se ROS povezuje s bibliotekom OpenCV koja se koristi za obradu slike s kamere. Napravit ćemo čvora pomoću C++ programskog jezika, čime će se testirati funkcionalnost i uspješnost integracije vizijskog sustava.

7.3.1. ROS usb_cam paket

ROS sustav podržava korištenje kamera koje se spajaju preko USB priključka preko paketa *usb_cam.* Paket *usb_cam* je vrsta ROS drivera koji se koristi za Video4Linux (V4L) USB kamere. V4L je skup drivera za različite uređaje za Linux operativni sustav koji se koristi za snimanje videa u realnom vremenu preko web kamera. ROS paket *usb_cam* koristi postojeće V4L drivere i objavljuje slijed videa u obliku slika sadržane u obliku ROS poruka. Možemo napraviti čvor koji se pretplati na temu na koju ROS objavljuje slike videa i istovremeno raditi vlastitu obradu slike. Službeni ROS paket *usb_cam* može se instalirati direktno na računalo preko sljedeće naredbe u Terminalu *sudo apt-get install ros-kinetic-usb-cam.* Nakon toga možemo provjeriti uspješnost instalacije pomoću naredbe *v4l2-ctl –list-devices* koja će nam izlistati sve video uređaje koje je pronašao kernel kao što je prikazano na slici 45.



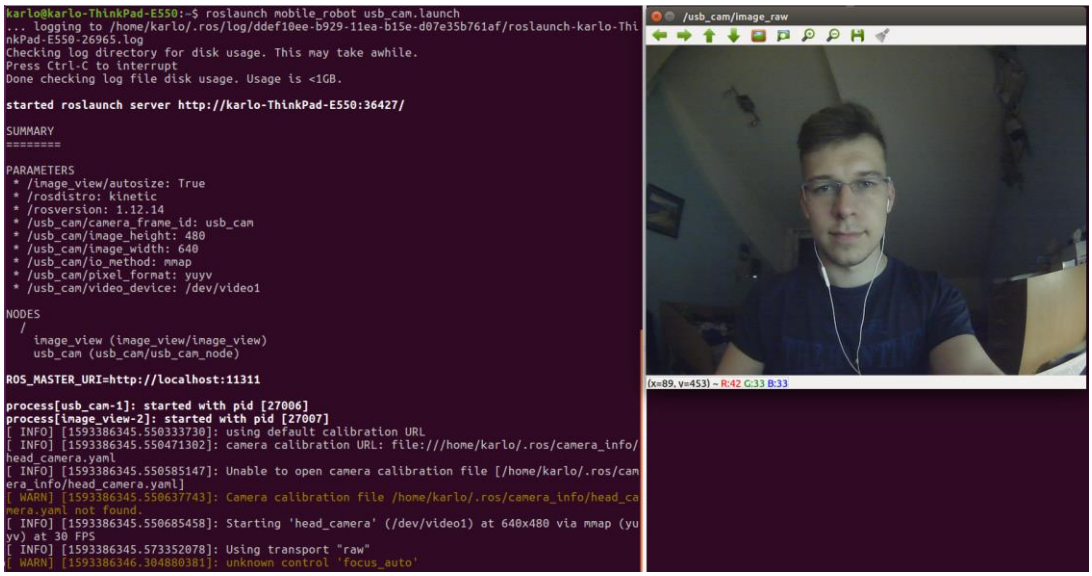
Slika 45. Prikazani video uređaji spojeni na računalo

7.3.2. Testiranje kamere u ROS-u

Sada kada smo instalirali potrebne pakete za integraciju kamere u ROS-u, trebamo ispitati njihovu funkcionalnost. Iskoristit će se *launch* datoteka koja će pokrenuti *usb_cam* čvorove kako bi prikazala slike s kamere i objavila ih na ROS *image* temu. Unutar kreirane launch datoteke možemo mijenjati različite parametre kao što su rezolucija slike, veličina prozora videa, vrstu video uređaja s kojeg želimo prikazivati sliku, ime čvora itd. Vrste launch datoteka smjestili smo u paket *mobile_rob*ot u datoteci *launch* radi pridržavanja organizacije strukture programskog koda u ROS-u. Primjer datoteke za testiranje kamere prikazan je ispod.

```
<launch>
  <node name="usb_cam" pkg="usb_cam" type="usb_cam_node" output="screen" >
    <param name="video_device" value="/dev/video1" />
    <!-- <param name="image_width" value="1280" />
    <param name="image_height" value="720" /> -->
    <param name="image_width" value="640" />
    <param name="image_height" value="480" />
    <param name="pixel_format" value="yuyv" />
    <param name="camera_frame_id" value="usb_cam" />
    <param name="io_method" value="mmap"/>
  </node>
  <node name="image_view" pkg="image_view" type="image_view" respawn="false" output="screen">
    <remap from="image" to="/usb_cam/image_raw"/>
    <param name="autosize" value="true" />
  </node>
</launch>
```

Pokretanje čvorova *usb_cam* i *image_view* izvodi se pomoću naredbe *roslaunch mobile_robot usb_cam.launch* i ako sve funkcionira kako treba prikazat će se ekran kao na slici 46. ROS tema s koje ćemo kasnije učitavati i obrađivati sliku je pretplaćivanjem na temu */usb_cam/image_raw*.

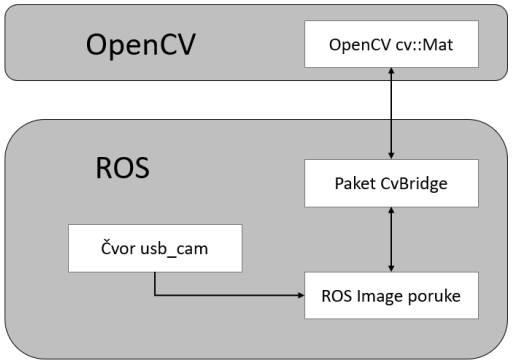


Slika 46. Uspješna integracija usb_cam paketa u ROS-u

7.3.3. Open CV 3.3.1

Obrada slike napravljena je pomoću biblioteke OpenCV. OpenCV je jedna od najpopularnijih biblioteka koja omogućuje računalnu viziju u realnom vremenu. Napisana je u programskom jeziku C/C++ i besplatna je za akademsku i komerijaclnu primjenu. Obrada slike može se programirati koristeći programske jezike C++, Python ili Java i podržana je na različitim operativnim sustavima od Windowsa, Linuxa i MacOS pa sve do Androida i iOS. Sasoji se od niza programske podrške koja se može koristiti za implementaciju računalnog vida.

Biblioteka OpenCV je integrirana u ROS-u kroz paket koji se zove *vision_opencv*. Ovaj paket smo instalirali kada smo napravili instalaciju ROS-a pomoću naredbe *ros-kinetic-desktop-full*. *Vision_opencv* je vrsta metapaketa koji sastoji od dva paketa, a to su *cv_bridge* i *image_geometry*. Paket *cv_bridge* je odgovoran za pretvaranje OpenCV slikovnih podataka vrste *cv::Mat* u ROS *Image* poruku zvanu *sensor_msgs/Image* i obratno. Ukratko, ponaša se kao most koji spaja OpenCV s ROS-om. S druge strane, paket *image_geometry* odgovoran je za kalibraciju kamere. On sadrži biblioteke napisane u C++ i Pythonu koje pomažu ispravljanju geometrijske karakteristike slike koristeći kalibracijske parametre. *Image_geometry* koristi tip poruke imena *sensor_msgs/CameraInfo* za rukovanje s kalibracijskim parameterima i korekcije slike. Između ova dva paketa, najviše ćemo koristiti *cv_bridge* kako bi pretvorili ROS *Image* poruku iz *usb_cam* za oblik pogodan *OpenCV*-u, a to je cv::Mat oblik. Nakon pretvorbe poruke u cv::Mat formu, možemo primijeniti dijelove OpenCV-a za procesiranje slike s kamere. Na slici 47. prikazan je način pretvorbe slike pomoću *cv_bridgea*.



Slika 47. Proces pretvorbe slike pomoću cv_bridge paketa

Drugi paket koji ćemo koristiti je ROS Image poruke između dva ROS čvora je *image_transport*. Ovaj paket se koristi da se na njega čvorovi pretplate te da objavi podatke o slici u ROS-u. Image_transport nam može pomoći da se transportiraju slike na niskom frekvencijskom području koristeći različite tehnike komprimiranja slike. Prilikom potpune instalacije ROS-a, instalira se i ovaj paket.

Ispravnost povezivanja kamere s ROS-om preko cv_bridgea i image_transport paketa ćemo testirati preko koda napisanog u C++ programskom jeziku. Tako ćemo testirati načine pisanja čvorova koji povezuje nekoliko različitih čvorova. Također, testirat će se uspješnost integracije kamere i prikazati način dobivanja slike s kamere u ROS-u

7.3.4. Kod u C++

Prvo moramo uključiti potrebne biblioteke kako bi mogli raditi sa svim dijelovima konfiguriranih paketa.Biblioteka *ros/ros.h* se mora uključiti u svakom čvoru koji se kreira, inače se izvorni kod neće kompajlirati. Koristeći image_transport možemo se pretplatiti i objavljivati na slike u ROS-u te nam je omogućen pristup komprimiranim slikama. Moramo uključiti i *cv_bridge* paket koji sadrži korisne funkcije prilikom enkodiranja slika. I na kraju za obradu slike koriste se dvije biblioteke to su imgproc i highgui.

```
#include "ros/ros.h"
#include <image_transport/image_transport.h>
#include <cv_bridge/cv_bridge.h>
#include <sensor_msgs/image_encodings.h>
#include <opencv2/imgproc/imgproc.hpp>
#include <opencv2/highgui/highgui.hpp>
```

Sljedeća linija koda daje ime slikovnom prozoru koji će se koristiti kroz program.

```
static const std::string OPENCV_WINDOW = "Image window";
```

Zatim je kreirana C++ klasa radi lakše organizacije dijelova ROS čvora. Objekt *ros::Nodehandle* je potreban za rukovanje s ROS čvorovima, *ImageTransport* pomaže ROS-u da kroz ROS sustav šalje slike te *Subscriber* i *Publisher* za slanje ili dohvaćanje podataka s određene teme.

```
class ImageConverter
{
private:
    ros::NodeHandle nh_;
    image_transport::ImageTransport it_;
    image_transport::Subscriber image_sub_;
    image_transport::Publisher image_pub_;
```

Konstruktor se koristi za inicijalizaciju kreiranih čvorova. Čvor image_sub_ kreira čvor koji se pretplati na ulaznu temu s koje dohvaća sliku. Kada god se pojavi slika na temi *image*, čvor pretplatitelj će pozvati funkciju *imageCallback*. S druge strane, čvor image_pub_ je odgovoran da se nakon obrade slike, slika objavi na temu /image_converter/output_video.

```
    ImageConverter() : it_(nh_)
    {
        // Pretplati se na dolazni video preko teme image
        image_sub_= it_.subscribe("image", 1, &ImageConverter::imageCallback, this);
        // Objavi video na temu /image_converter/output_video
        image_pub_ = it_.advertise("/image_converter/output_video", 1);
        cv::namedWindow(OPENCV_WINDOW);
    }
```

Nakon prekida rada čvora, bitno je izbrisati sve kreirane objekte pomoću OpenCV highgui funkcije *destroyWindow()* koji se poziva da kreira ili uništi prozor za prikaz slike s kamere.

```
    ~ImageConverter() {
        cv::destroyWindow(OPENCV_WINDOW);
    }
```

Idući dio koda je definicija funkcije imageCallback() koja prvo pretvara ROS sliku u oblik CvImage koja je potrebna za rad s OpenCV. Ono što funkcija zapravo napravi je da pretvori podatke s teme sensor_msgs/Image u cv::Mat koji je OpenCV tip podatka. Nakon što se dobije slika na nju će se nacrtati krug stoga trebamo koristiti OpenCV funkciju toCvCopy(). Ako se ne uspije dohvatiti slika s kamere, ROS će ispisati grešku i ukazati nam koji je razlog zbog kojeg se ne može dobiti slika.

```
    void imageCallback(const sensor_msgs::ImageConstPtr& msg)
    {
        cv_bridge::CvImagePtr cv_ptr;
        try
        {
            cv_ptr = cv_bridge::toCvCopy(msg, sensor_msgs::image_encodings::BGR8);
        }
        catch(const std::exception& e)
        {
            ROS_ERROR("cv_bridge exception: %s", e.what());
            return;
        }
```

Sada slijedi glavni dio programa. Nakon što se slika pretvori iz ROS *image* poruke u oblik podatka OpenCV-a da se primjene OpenCV funkcionalnosti i nacrta se zeleni krug na prozoru. Na kraju se prikaže taj krug na prozoru te se pretvori OpenCV slika u ROS *image* poruku kako bi se mogla slati dalje kroz ROS sustav. Na kraju se ROS čvor mora imenovati i inicijalizirati u *main()* funkciji.

```
        // Nacrtaj krug na izlaznom videu
        if (cv_ptr->image.rows > 60 && cv_ptr->image.cols > 60 ) {
            cv::circle(cv_ptr->image, cv::Point(50,50), 10, CV_RGB(0, 255, 0));
        }
        // Osvježi GUI prozor
        cv::imshow(OPENCV_WINDOW, cv_ptr->image);
        cv::waitKey(3);
        // Objavi modificirani video
        image_pub_.publish(cv_ptr->toImageMsg());
    }
};
```

Kako bi se čvor mogao pravilno iskonfigurirati potrebno je podesiti i *CmakeLists.txt* datoteku. Bitno je da se tamo uključe sve potrebne biblioteke koje smo dodali u čvoru kao što su *cv_bridge* i *image_transport* inače će se pojaviti greška prilikom izgradnje čvora.

```
find_package(catkin REQUIRED COMPONENTS
  roscpp
  rospy
  std_msgs
  actionlib_msgs
  cv_bridge
  image_transport
  message_generation
)
```

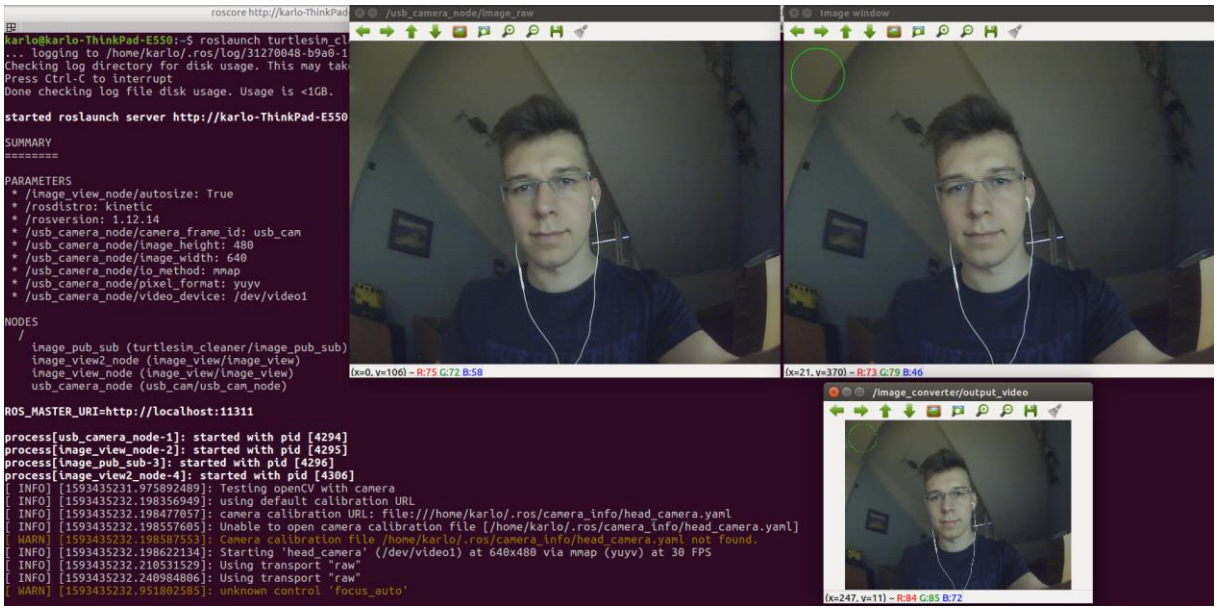
Moramo uključiti i potrebne pakete za izgradnju koju koristi CMake makro za generiranje paketa:

```
include_directories(
  ${catkin_INCLUDE_DIRS}
  ${OpenCV_INCLUDE_DIRS}
)
```

I na kraju da se može kreirati *image_pub_sub* čvor te povezati s catkin i OpenCV bibliotekama:

```
add_executable(image_pub_sub src/robot_perception/image_pub_sub.cpp)
target_link_libraries(image_pub_sub ${catkin_LIBRARIES})
target_link_libraries(image_pub_sub ${OpenCV_LIBRARIES})
```

Ispravnost i funkcionalnost kamere testirana je i prikazana na slici 48. Vidimo tri prozora u kojima se prikazuje slika s kamere. Prvi lijevi prozor je ono što ROS čita s kamere preko usb_cam drivera, drugi prozor je obrađena slika pomoću OpenCV-a s nacrtanim zelenim krugom i treće je izlazni video koji se šalje natrag opet u ROS.



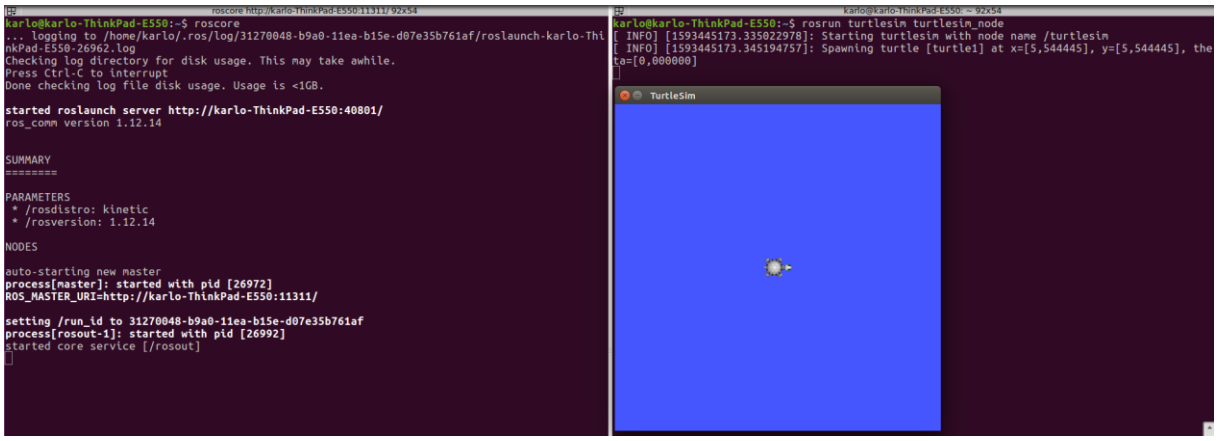
Slika 48. Testiranje funkcionalnosti integrirane kamere u ROS-u

7.4. Testiranje na Turtlesim simulatoru

Prilikom potpune instalacije ROS sustava uključen dolazi simulator koji se zove Turtlesim. Simulator se sastoji od grafičkog prozora koji prikazuje mobilnog robota u obliku kornjače. Na simulatoru se mogu testirati osnovne funkcije ROS-a kao što su ROS čvorovi, parametri, poruke, ali isto tako upravljanje mobilnog robota. Kornjača se može pomicati po

zaslonu tako da se upravljanje radi preko ROS čvorova. Također, mogu se testirati različiti algoritmi koji se primjenjuju i na realne mobilne robote.

Turtlesim je ROS paket koji se pokreće kao i svi ostali ROS paketi tako da se u jednom prozoru pokrene ROS s naredbom roscore, a u drugome turtlesim čvor naredbom *rosrun turtlesim turtlesim_node* kao što je prikazano na slici 49. Pokretanjem turtlesim čvora stvara se kornjača u novom prozoru.



Slika 49. Pokretanje Turtlesim simulatora

Detalji o Turtlesim čvoru mogu se dobiti upisivanjem naredbe *rosnode info /turtlesim* koja se koristi kada se žele saznati sve informacije o pojedinom čvoru unutar ROS sustava. Ispis informacija kao na slici 50. govori nam da Turtlesim čvor objavljuje informacije na dvije teme, a to su */turtle1/color_sensor* koji se koristi za dobivanje informacije o boji pozadine i */turtle1/pose* koji prikazuje poziciju robota u obliku x i y koordinata. Čvor *turtlesim* se može pretplatiti na temu */turtle1/cmd_vel* odnosno mogu se slati reference brzine kao i kod mobilnog robota te se kornjačom može upravljati.

```
karlo@karlo-ThinkPad-E550:~$ rosnode info /turtlesim
--------------------------------------------------------------------------------
Node [/turtlesim]
Publications:
 * /rosout [rosgraph_msgs/Log]
 * /turtle1/color_sensor [turtlesim/Color]
 * /turtle1/pose [turtlesim/Pose]

Subscriptions:
 * /turtle1/cmd_vel [unknown type]

Services:
 * /clear
 * /kill
 * /reset
 * /spawn
 * /turtle1/set_pen
 * /turtle1/teleport_absolute
 * /turtle1/teleport_relative
 * /turtlesim/get_loggers
 * /turtlesim/set_logger_level
```

Slika 50. Informacije o Turtlesim čvoru

Pomoću Turtlesim simulatora testirat će se upravljanje mobilnom robotom, čiji dijelovi će kasnije biti iskorišteni za upravljanje stvarnim mobilnim robotom. Prvo će se testirati osnovne funkcije za gibanje prema naprijed i nazad, te rotacija oko vertikalne osi. Nakon toga će se primijeniti složenije upravljanje koristeći osnovne funkcije kao što su odlazak na cilj te gibanje robota po nekoj zadanoj mreži točaka u prostoru. Kreiran je novi paket imena turtlesim_cleaner u kojemu će se testirati navedene funkcionalnosti. Upravljanje je napravljeno u obliku jednog ROS čvora koji se zove *robot_control.cpp.*

Čvor robot_control.cpp kreiran je tako da su uključene potrebne biblioteke za ostvarivanje željene funkcionalnosti. Prvo je potrebno uključiti klasični ROS paket za rad s čvorovima. Nakon toga dvije predefinirane poruke, a to su za dobivanje pozicije turtlesim mobilnog robota *turtlesim/Pose.h* te poruka za slanje reference brzine *geometry_msgs/Twist.h*.

```
#include "ros/ros.h"
#include "turtlesim/Pose.h"
#include "geometry_msgs/Twist.h"
```

Zatim definiramo klasu u kojoj ćemo definirati sve potrebne funkcije za upravljanje robotom na simulatoru, definiramo imena poruka koje će služiti za pretplaćivanje na temu i objavljivanje na temu. Kreirana su četiri objekta. Prvi služi za rukovanje s ROS čvorovima imena *nodeHandle_*, čvor za slanje reference brzine na temu */turtle1/cmd_vel velocity_pub*, čvor za dobivanje informacija o poziciji robota s teme */turtle1/pose* i na kraju za korištenje informacija o poziciji unutar programa kreiran je *turtlesim_pose* objekt.

```
class RobotControl
{
private:
    ros::NodeHandle nodeHandle_;
    ros::Publisher velocity_pub;
    ros::Subscriber pose_sub;
    turtlesim::Pose turtlesim_pose;

public:
    RobotControl()
    {
        velocity_pub = nodeHandle_.advertise<geometry_msgs::Twist>
("/turtle1/cmd_vel", 1000);
        pose_sub = nodeHandle_.subscribe
("/turtle1/pose", 10, &RobotControl::poseCallback, this);
    }
```

Unutar klase deklarirane su sve funkcije koje ćemo koristiti za upravljanje mobilnim robotom. Ovakav način je napravljen da se kasnije jednostavne funkcije za translacijsko i rotacijsko gibanje mogu implementirati i koristiti u drugim funkcijama. Osim osnovnih vrsta gibanja pokazat će se zakret robota za željeni kut te kako se dobiva pozicija s robota. Također, demonstrirat će se odlazak robota u zadani cilj i na kraju način mogućeg skeniranja definiranog prostora. Ovdje su prikazana samo imena funkcija i njihovih argumenata. Njihova funkcionalnost će se objasniti kroz sljedeća poglavlja.

```
    void moveLinear(double speed, double distance, bool direction);
    void roatateAngular (double angular_speed, double angle, bool clockwise);

    double degrees2radians(double angle_in_degrees);
    void setDesiredOrientation (double desired_angle_radians);

    void poseCallback(const turtlesim::Pose::ConstPtr & pose_message);
    void moveToAPose(turtlesim::Pose goalPosition, double distance_tolerance);

    void gridClean();
};
```

Gibanje robota ostvaruje se tako da se diferencijalni kinematski model mobilnog robota pretvori u gibanje izraženo kao komponenta translacije i rotacije što je prikazano jednadžbama (4) i (5). Takav način upravljanja puno je jednostavniji i lakše se njime rukuje nego zadavanje različitih brzina pojedinom motoru mobilnog robota. Translacijsko gibanje ostvaruje se tako da se zadaju potpuno iste brzine na motorima te robot kreće prema naprijed ili prema natrag. S druge strane, rotacijsko gibanje se izvodi tako da se jedan kotač vrti brže od drugog i dolazi do rotacije robota oko vertikalne osi. Ovisno o tome koji kotač se okreće brže, na tu stranu robot skreće. Na slici 51. je prikazana predefinirana poruka *geometry_msgs/Twist* koja se koristi za gibanje robota. Sastoji se od dva vektora, odnosno dvije komponente brzine, a to su translacijska (*eng. linear*) i rotacijska (*eng. angular*) komponenta.

```
karlo@karlo-ThinkPad-E550:~$ rosmsg show geometry_msgs/Twist
geometry_msgs/Vector3 linear
  float64 x
  float64 y
  float64 z
geometry_msgs/Vector3 angular
  float64 x
  float64 y
  float64 z
```

Slika 51. ROS poruka za upravljanje gibanjem mobilnog robota

7.4.1. Translacijsko gibanje

Prvo i najjednostavnije gibanje je translacijsko gibanje koje se ostvaruje tako da se mobilnom robotu pošalje pozitivna ili negativna referenca brzine preko poruke *geometry_msgs/Twitst*. Komponenta za kretanje robota prema naprijed je *linear.x* komponenta. Ako je pozitivna robot se kreće prema naprijed, a ako je negativna robot se kreće unatrag. Funkcija za translacijsko gibanje robota sastoji se od argumenata za odabir brzine kojom želimo da robot ide, udaljenosti te smjera koji može biti naprijed ili natrag. Bitno je postaviti sve ostale komponente vektora brzine na vrijednost nula kako bi bili sigurni da robot se giba linearno.

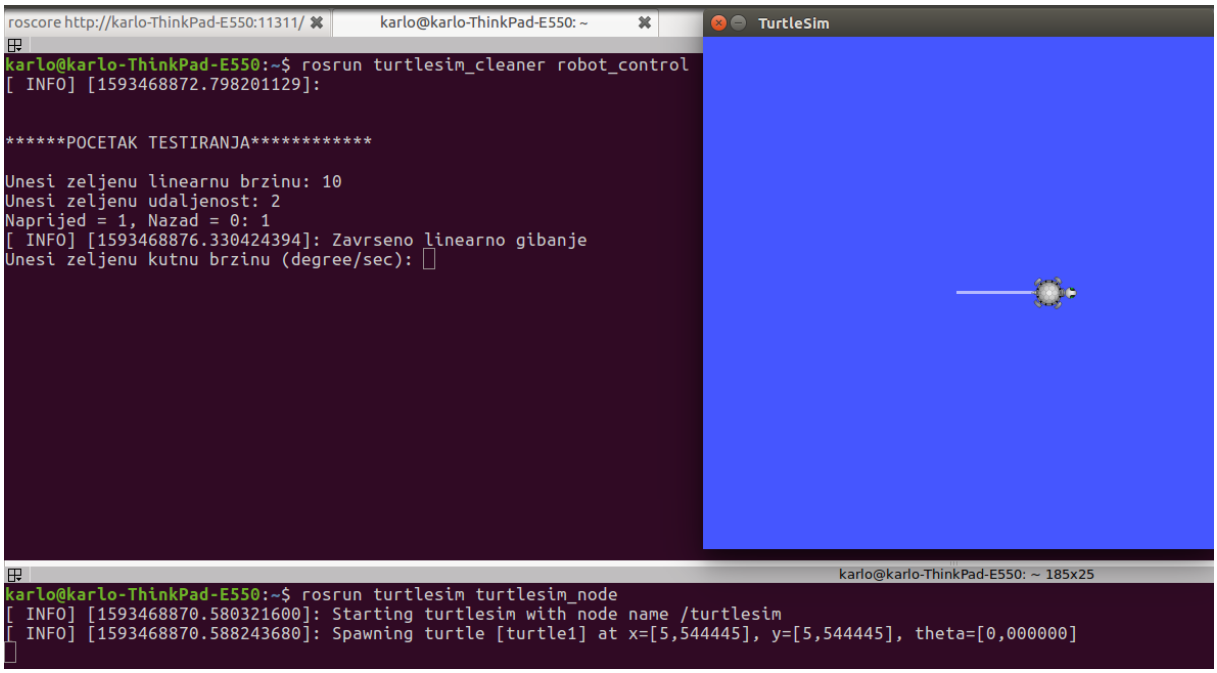
```
void RobotControl::moveLinear(double speed, double distance, bool direction){
    geometry_msgs::Twist vel_msg;

    // Određivanje smjera prema naprijed ili prema nazad
    if (direction)
        vel_msg.linear.x =abs(speed);
    else
        vel_msg.linear.x =-abs(speed);
    vel_msg.linear.y = 0;
    vel_msg.linear.z = 0;
    vel_msg.angular.x = 0;
    vel_msg.angular.y = 0;
    vel_msg.angular.z =0;
```

Nakon toga inicijaliziramo varijablu za spremanje trenutnog vremena preko kojeg će se izračunavati udaljenost koju je prešao robot tako da se množi trenutna brzina s razlikom vremena. Također, inicijaliziramo trenutnu udaljenost na vrijednost nula kako bi kada se unese željena udaljenost bila točna, a ne umanjena za prošlu udaljenost koja je spremljena u toj varijabli. Definira se i frekvencija slanja poruka u ROS-u na 100hz.

```
    double t0 = ros::Time::now().toSec();
    double current_distance = 0.0;
    ros::Rate loop_rate(100);
```

Zatim se kreira beskonačna petlja koja izračunava je li trenutna udaljenost manja od željene udaljenosti. Dok je god udaljenost manja znači da se mobilni robot nije pomaknuo za željenu udaljenost i za to vrijeme će slati robotu naredbu da se giba prema naprijed objavljujući referencu brzine na */turtle1/cmd_vel* temu. Na kraju, kada je robot dosegao željenu udaljenost, trenutna udaljenost postat će veća od željene udaljenosti i robot će se zaustaviti. Rezultat testiranja gibanja robota prema naprijed prikazano je na slici 52. Pokraj prozora u kojem se nalazi simulator Turtlesim, upisuju se vrijednosti željene brzine, udaljenosti i smjera kretanja mobilnog robota.

```
    while(current_distance < distance){
        velocity_pub.publish(vel_msg);
        double t1 = ros::Time::now().toSec();
        current_distance = speed * (t1-t0);
        ros::spinOnce();
        loop_rate.sleep();
    }
    //zaustavi robota
    vel_msg.linear.x =0;
    velocity_pub.publish(vel_msg);
}
```



Slika 52. Gibanje mobilnog robota prema naprijed

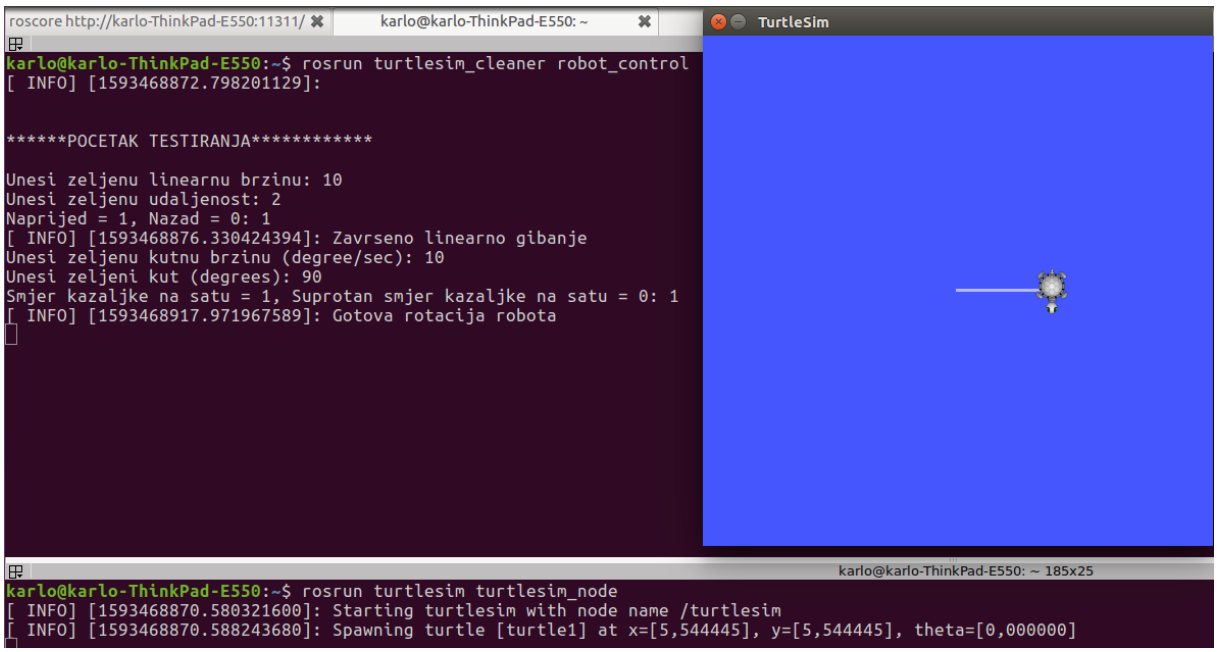
7.4.2. Rotacijsko gibanje

Programski kod rotacijskog gibanja mobilnog robota u simulatoru potpuno je isti kao i za translacijsko gibanje samo se koristi rotacijska komponenta z ROS poruke *geometry_msgs/ Twist*.

```
   if (clockwise)
        vel_msg.angular.z = -fabs(angular_speed);
    else
        vel_msg.angular.z = fabs(angular_speed);
```

Za testiranje rotacijskog gibanja koristi se pomoćna funkcija koja omogućuje korisniku da upiše željeni kut u stupnjevima, a da program radi s radijanima.

```
double RobotControl::degrees2radians(double angle_degrees){
    return (angle_degrees *PI) /180.0;
```



Slika 53. Rezultat rotacijskog gibanja robota

7.4.3. Rotacija za određeni kut

Jedan od ciljeva upravljanja je da se može robota zakrenuti za točno određeni kut. Kako bi se to moglo napraviti potrebna nam je trenutna orijentacija robota. Orijentacija se dobiva tako da se čvor pretplati na temu */tutrtle1/pose* i čita informacije s poruke na koju objavljuje *turtlesim* čvor. Pročitani podaci spremaju se u varijablu *turtlesim_pose* kako bi se mogle te informacije koristiti u drugim dijelovima programa.

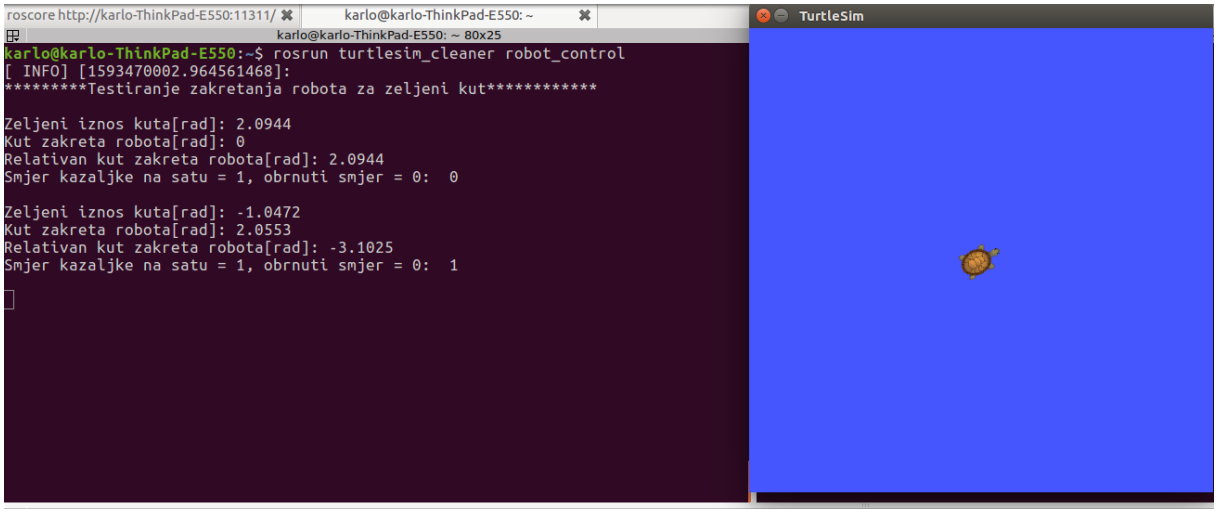
```
void RobotControl::poseCallback(const turtlesim::Pose::ConstPtr &pose_message){
    turtlesim_pose.x=pose_message->x;
    turtlesim_pose.y=pose_message->y;
    turtlesim_pose.theta=pose_message->theta;
}
```

Rotacija za određeni kut napravljena je pomoću funkcije *setDesiredOrientation()* koja prima kao argument kut u radijanima i oduzima ga od trenutnog kuta zakreta mobilnog robota. Kako se robot rotira prema željenom kutu, smanjuje se trenutni kut sve dok ne padne u nulu. Unutar funkcije *setDesiredOrientation()* korištena je osnovna funkcija za rotaciju robota kako se ne bi morala opet pisati čitava logika za rotacijsko gibanje. Istovremeno dok se robot rotira u željeni

kut, ispisuju se u konzoli vrijednosti željenog kuta, trenutnog kuta, relativnog kuta te smjera u kojem se rotira mobilni robot.

```
void RobotControl::setDesiredOrientation (double desired_angle_radians){
    double relative_angle_radians = desired_angle_radians - turtlesim_pose.theta;
    bool clockwise = ((relative_angle_radians<0)?true:false);
    std::cout << "Zeljeni iznos kuta[rad]: " << desired_angle_radians << std::endl;
    std::cout << "Kut zakreta robota[rad]: " << turtlesim_pose.theta << std::endl;
    std::cout << "Relativan kut zakreta robota[rad]: "
              << relative_angle_radians <<std::endl;
    std::cout << "Smjer kazaljke na satu = 1, obrnuti smjer = 0:  "
              << clockwise <<std::endl;
    std::cout << ""  <<std::endl;
    roatateAngular(degrees2radians(10), fabs(relative_angle_radians), clockwise);
}
```

Rezultati testiranja rotacije mobilnog robota mogu se vidjeti na slici 54. U lijevom Terminal prozoru se prikazuju vrijednosti iz funkcije *setDesiredOrientation()*, a desno je prikazano kako se mobilni robot rotira za definirane kuteve od 120º, -60º i 10º.



Slika 54. Rotacija mobilnog robota za definirani kut

7.4.4. Odlazak na cilj

Razmotrimo sada slučaj da se mobilni robot želi dovesti u neku definiranu točku u prostoru. S obzirom na ravninski koordinatni sustav možemo označiti početnu poziciju robota s (x1, y1), a cilja sa (x2, y2). Prvo je potrebno izračunati udaljenost između cilja i početne koordinate, što se može dobiti iz formule za udaljenost između dvije točke u ravnini. To je napravljeno u funkciji *getDistance()* koja uzima koordinate cilja i početne točke te izračunava njihovu udaljenost.

```
double getDistance(double x1, double y1, double x2, double y2){
    return sqrt(pow((x2-x1),2)+pow((y2-y1),2));
```

Pomoću funkcije *getDistance()* kontrolirat će se translacijska brzina robota tako da bude proporcionalna udaljenosti do cilja što je prikazano idućom jednadžbom:

$$v_x = K_p \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2} \ , \tag{15}$$

Implementacija jednadžbe za regulaciju translacijske brzine unutar funkcije moveToAPose prikazana je sljedećim linijama koda. Funkcija prima koordinate cilja i toleranciju udaljenosti kao argumente.

```
        double Kp=1.0;
        double error = getDistance(turtlesim_pose.x, turtlesim_pose.y, goalPosition.x, g
oalPosition.y);
        double tolerance = tolerance+error;
        vel_msg.linear.x = (Kp*error);
```

Rotacijska brzina dobiva se iz sljedeće jednadžbe:

$$\omega = K_h \, atan2 \, \frac{y_2 - y_1}{x_2 - x_1} \ , \tag{16}$$

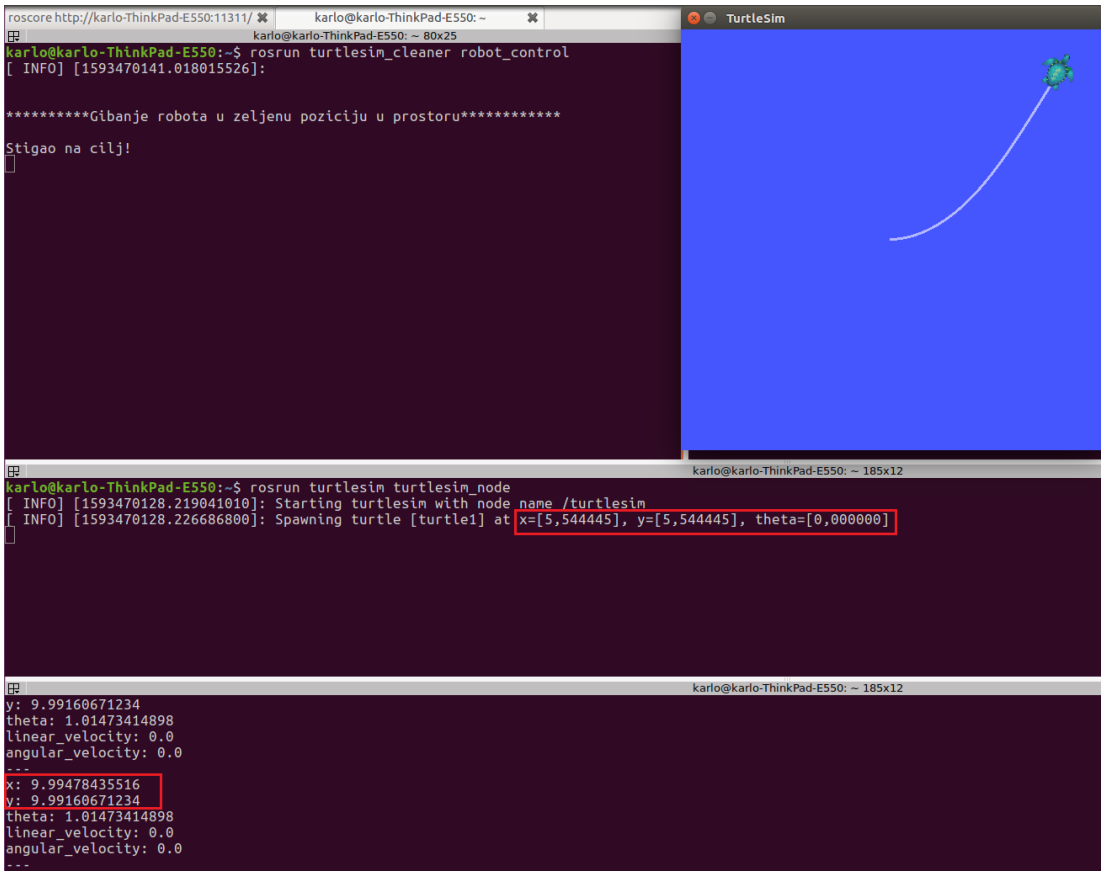
Implementirano unutar funkcije *moveToAGoal()* to izgleda ovako

```
        vel_msg.angular.z =4*(atan2(goalPosition.y-turtlesim_pose.y, goalPosition.x-
turtlesim_pose.x)-turtlesim_pose.theta);
```

Kao i kod rotacije za određeni kut, uspoređuju se trenutna pozicija sa željenom pozicijom. Kako se robot približava cilju, tako će se ta udaljenost smanjivati, a time i translacijska i rotacijska brzina. Kada robot dođe u cilj, udaljenost će postati jako malena te se zato smanjuju i komponente brzine.

```
       }while(getDistance(turtlesim_pose.x, turtlesim_pose.y, goalPosition.x, goalPositi
on.y)>distance_tolerance);
}
```

Vrijednosti parametara proporcionalnog regulatora, gdje je vrijednost faktora translacijske brzine K_p jedan, a rotacijske K_h četiri pokazale najbolji rezultat. Na slici 55. u najdonjem terminal prozoru ispisuju se trenutna pozicija i orijentacija robota. Točka cilja je (10, 10), a konačna pozicija robota je (9.9948, 9.99160). Greška x koordinate je 0.0052, a y koordinate 0.0084 čime je regulacija pomoću proporcionalnog regulatora vrlo prihvatljiva.



Slika 55. Pomicanje robota u ciljanu točku u prostoru

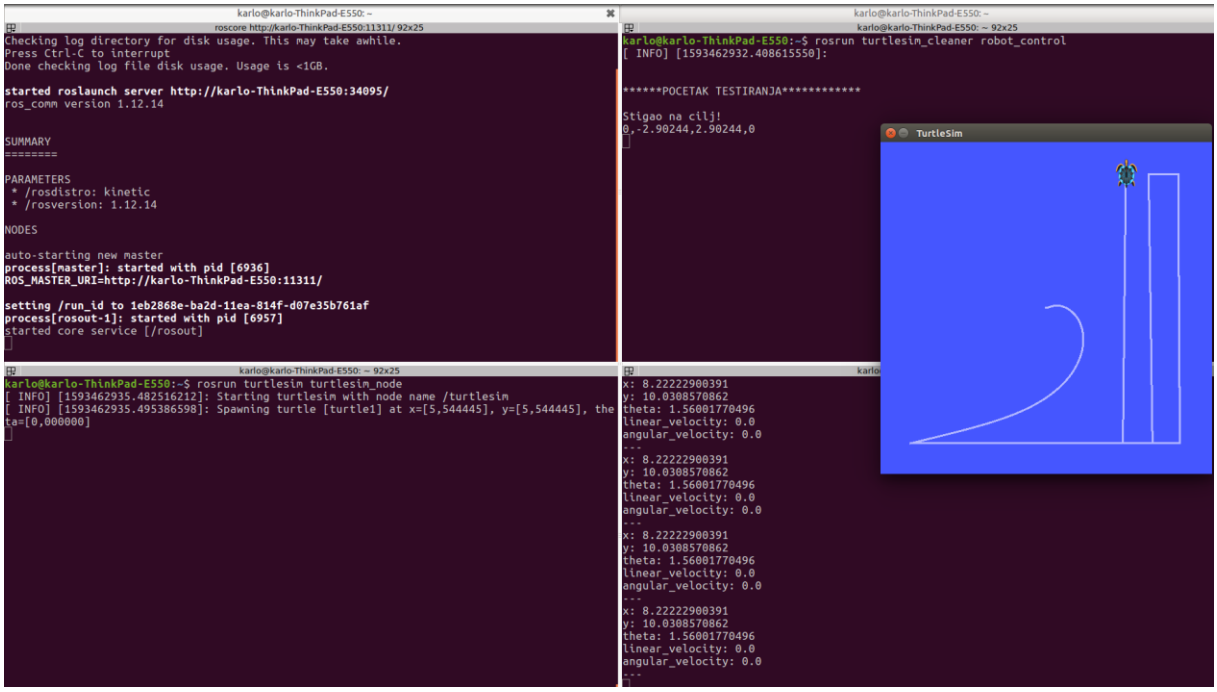
7.4.5. Mapiranje prostora

Koristeći osnovne funkcionalnosti za translacijsko i rotacijsko gibanje, rotacija za definirani kut te pomicanje robota u točku u prostoru, napravljena je simulacija mapiranja prostora. Simulacija je napravljena tako da prvo robot ode u željenu točku u prostoru, a nakon toga se napravi rotacija u smjeru kojem treba ići. Funkcija loop.sleep() se koristi da bi se osiguralo da se radnja prije uspješno završila.

```
void RobotControl::gridClean(){
    ros::Rate loop(0.5);
    // kreiranje početne pozicije
    turtlesim::Pose pose;
    pose.x=1;
    pose.y=1;
    pose.theta=0;

    // odlazak u zeljenu poziciju
    moveToAPose(pose, 0.01);
    loop.sleep();
    setDesiredOrientation(0);
    loop.sleep();
```

Kombinacijom rotacijskog i translacijskog gibanja dobije se rezultat kao na slici 56.

```
// rotacija za 90 stupnjeva i pomicanje gore
roatateAngular(degrees2radians(10), degrees2radians(90), false);
loop.sleep();
moveLinear(2.0, 9.0, true);

// rotacija za 90 stupnjeva i pomicanje lijevo
roatateAngular(degrees2radians(10), degrees2radians(90), false);
loop.sleep();
moveLinear(2.0, 1.0, true);
```



Slika 56. Rezultat simulacije mapiranja prostora

Funkcionalnosti upravljanja mobilnim robotom uspješno su testirane na ROS simulatoru Turtlesim. Greške koje su dobivene na simulatoru su prihvatljive za potrebe testiranja upravljanja. Dio upravljanja sa simulatora pokušat će se testirati i integrirati na realnom mobilnom robotu te će rezultati biti prikazani u idućem poglavlju. Ako je potrebno mogu se podesiti drugi parametri ili primijeniti bolji načini regulacije.

8. UPRAVLJANJE MOBILNIM ROBOTOM

Upravljanje mobilnim robotom može se izvoditi na tri načina[21]:

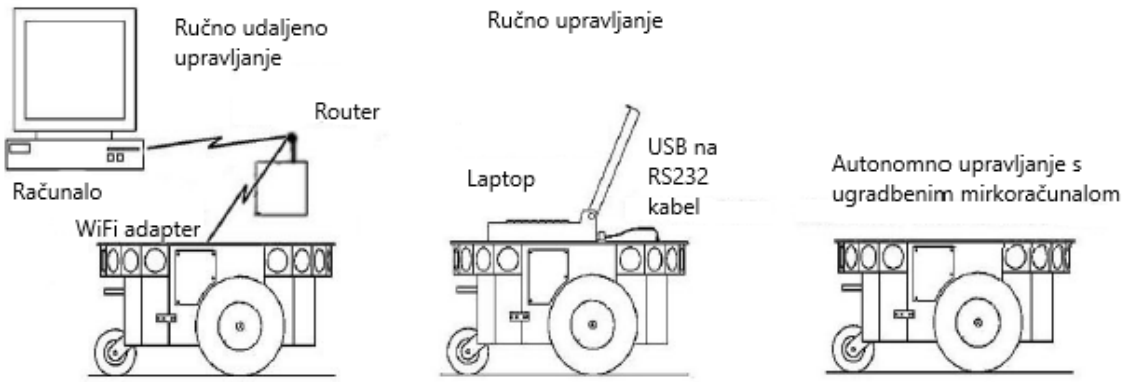
- Ručno upravljanje
- Poluautonomno upravljanje
- Autonomno upravljanje

Ručno upravljanje je način upravljanja pri kojem se kontrola i nadzor nad mobilnim robotom izvodi direktno preko čovjeka. Kontrola se radi preko udaljenog priključka za upravljanje ili daljinskog upravljača.

Poluautonomno upravljanje se nalazi između ručnog i autonomnog upravljanja. Kontrola se izvodi i pomoću ručnog i pomoću autonomnog upravljanja. Za izvođenje jednostavnih zadataka, robot može koristiti autonomno upravljanje, ali kod kompleksnih zadataka mora se promijeniti način upravljanja na ručni jer inače neće moći odraditi zadani zadatak.

Autonomni robot ima potpunu kontrolu nad svojim akcijama i sam donosi svoje odluke. Ima mogućnost učenja i prilagodbe u novim situacijama te se čitavo upravljanje izvodi na samom robotu. Čovjek nema kontrolu nad robotom, nego mu kroz niz uvjeta i logičkih sljedova postavlja načine upravljanja u određenoj okolini. Navedene osobine se primjenjuju na okolinu u kojoj se može napraviti lokalizacija kako bi robot dobio podatke gdje se trenutno nalazi u prostoru i da na temelju toga može planirati svoje kretanje.

U ovom poglavlju prikazat će se ručno i autonomno upravljanje na modificiranoj verziji mobilnog robota. Primijenit će se dijelovi upravljanja koji su korišteni za simulaciju upravljanja mobilnim robotom kako bi se evaluirala ispravnost upravljačkog programa. Prvo će se prikazati načini ručnog upravljanja te ukazati na probleme koji su se pojavili. Kroz ručno upravljanje implementirat će se udaljeno upravljanje preko spajanja računala i mikroračunala u ROS-u na zajednički router. Na kraju će se objasniti način na koji je implementiran algoritam za autonomno upravljanje. Mobilni robot slijedi definirane objekte tako da na temelju obrade slike dobiva informacije kako se treba kretati u okolini u kojoj se nalazi. Vrste upravljanja koje će se testirati na mobilnom robotu prikazane su na slici 57.



Slika 57. Vrste upravljanje mobilnim robotom

8.1. Ručno upravljanje

Ručno upravljanje napravljeno je na temelju simulacijskih rezultata turtlesim simulatora. Funkcionalnosti koje smo tamo testirali, ovdje će biti evaluirane na realnom sustavu mobilnog robota. Prvo će se testirati osnovna funkcionalnost gibanja robota tako da je laptop povezan direktno s mobilnim robotom preko serijskog kabela koji na jednom kraju ima USB priključak, a na drugom kraju RS232 priključak kojim se spajamo direktno na mikrokontroler robota. Nakon toga će se testirati ručno udaljeno upravljanje tako da će mikroračunalo biti spojeno serijski s mikrokontrolerom robota, a primati će naredbe preko laptopa.

Slika 58. Ručno upravljanje mobilnim robotom

Upravljanje se izvodi jednako kao i što su testirani senzorski sustav i odometrijski podaci. U jednom terminal prozoru pokreće se ROS s naredbom roscore, u drugom se pokreće čvor RosAria dok se u trećem otvara paket za testiranje upravljanja koji se zove *mobile_robot*. Čvor kojime se ispituje funkcionalnost kretanja robota zove se *test_robot_movement.cpp*. Provjera ispravnosti načina upravljanja mobilnim robotom izvodi se pomoću već napravljenog čvora za odometrijske podatke iz poglavlja 7.2. Tako smo mogli sigurno odrediti koje i kolike su greške u upravljanju. Problem koji se javio pri testiranju funkcionalnosti upravljanja bio je način pisanja programskog koda. Naime, kod programskog koda za testiranje upravljanja na simulatoru kreirana je klasa u kojoj su definirane sve funkcije te napravljeni konstruktor i destruktor objekata. Ovdje takav način definiranja nije bio moguć. Razlog je što biblioteka ARIA koja je implementirana u ROS-u već sadrži konstruktor u kojem se inicijaliziraju pojedini čvorovi koji objavljuju informacije. Radi jednostavnosti, ovdje su korištene samo funkcije bez primjene klasa i objekata.

8.1.1. Translacijsko gibanje

Funkcija za pravocrtno gibanje napisana je isto kao i funkcija za translacijsko gibanje na simulatoru. Argumente koje prima su brzina kojom želimo da se robot giba, udaljenost te smjer gibanja koji može biti prema naprijed ili natrag. Kod implementacije funkcije za gibanje mobilnog robota, jako je važno sve ostale komponente vektora brzine staviti na nulu jer nekad u mikrokontroleru znaju ostati vrijednosti od prije. Inače bi se prilikom povezivanja robotom sam počeo neželjeno kretati.

```
void moveLinear(double speed, double distance, bool isForward) {
    geometry_msgs::Twist vel_msg;
    // Određivanje smjera prema naprijed ili prema nazad
    if (isForward)
        vel_msg.linear.x =fabs(speed);
    else
        vel_msg.linear.x =-fabs(speed);
    vel_msg.linear.y = 0;
    vel_msg.linear.z = 0;
    vel_msg.angular.x = 0;
    vel_msg.angular.y = 0;
    vel_msg.angular.z = 0;
```

Problem koji se javio prilikom testiranja upravljanja na robotu je što nije bilo moguće pokrenuti robota ako ne prima poruke minimalnom frekvencijom od 2Hz. Stoga je varijablu *loop_rate*, u kojoj se definira vrijednost u Hz, potrebno staviti minimalno na vrijednost 2.

```
double t0 = ros::Time::now().toSec();
double current_distance = 0.0;
ros::Rate loop_rate(2);
```

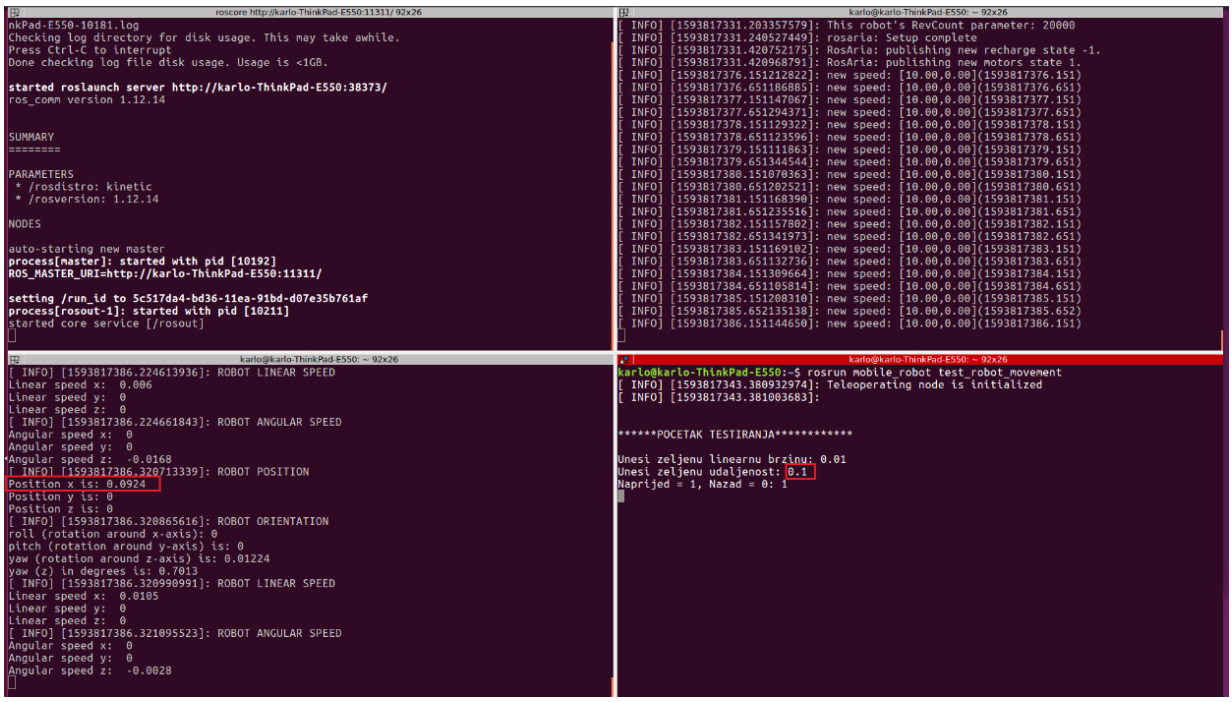
Na kraju da bi se izvršilo gibanje robota kreirana je beskonačna petlja koja objavljuje referentnu brzinu na definiranu temu i koja prestaje kada robot dosegne udaljenost koju smo mu zadali.

```
do{
    velocity_publisher.publish(vel_msg);
    double t1 = ros::Time::now().toSec();
    current_distance = speed * (t1-t0);
    ros::spinOnce();
    loop_rate.sleep();
}while(current_distance < distance);
```

Nakon izlaska iz petlje kao sigurnosni dio stavlja se komponenta linearne brzine na nulu kako robot se ne bi nastavio kretati kada ispuni naredbe koje smo mu zadali.

```
//zaustavi robota
 vel_msg.linear.x =0;
 velocity_publisher.publish(vel_msg);
```

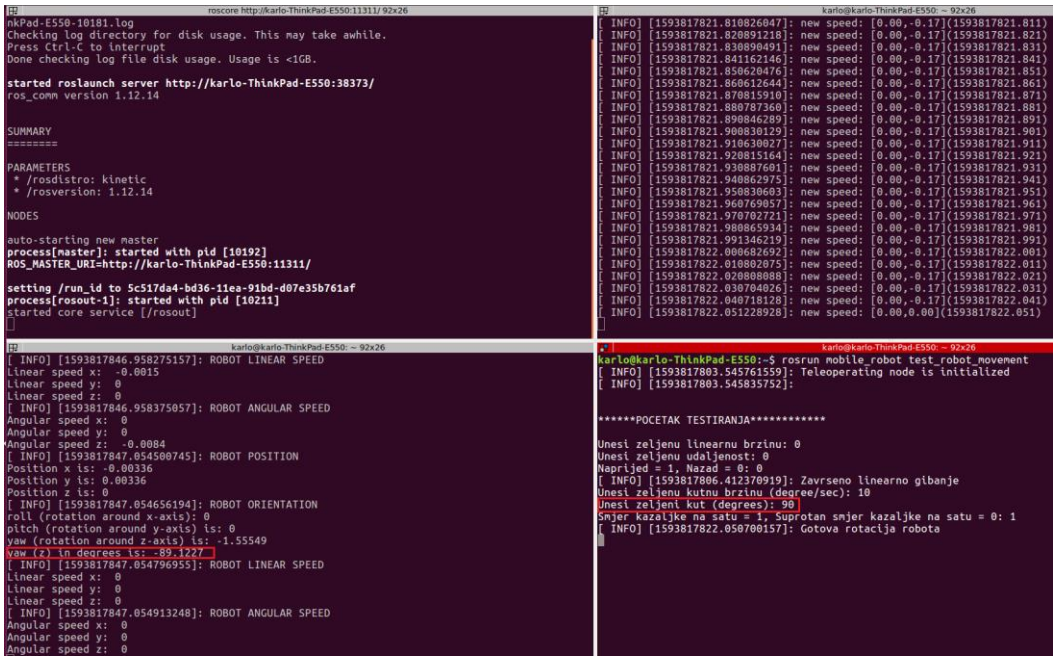
Rezultat ispitivanja gibanja na mobilnom robotu prikazan je na slici 59. Vidimo da smo zadali robotu da se pomakne za 0.1m, a da je došao na x koordinati na 0.0924 čime dobivamo grešku od 0.0076 što je vrlo dobar rezultat.



Slika 59. Rezultat ispitivanja translacijskog gibanja na mobilnom robotu

8.1.2. Rotacijsko gibanje

Programski kod za rotacijsko gibanje potpuno je isti kao i za translacijsko uz razliku da se zadaje komponenta za kutnu brzinu. Također, kod je jednak funkciji kojom je testirano rotacijsko gibanje na Turtlesim simulatoru. Rezultat testiranja rotacijskog gibanja na mobilnom robotu prikazan je na slici 60. Vidimo da je željeni kut zakretanja mobilnog robota 90 stupnjeva, a robot se zakrenuo za 89.1227. Time je greška od čak 0.8773 što je gotovo cijeli jedan stupanj. Rezultati koje smo dobili na simulatoru razlikuju se dosta od rezultata testiranih na mobilnom robotu. Moguće rješenje je da se primijeni druga vrsta regulatora kako bi se smanjila ta greška.



Slika 60. Rezultat ispitivanja rotacijskog gibanja mobilnog robota

8.1.3. Zakret za definirani kut

Prilikom ispitivanja zakreta za definirani kut koriste se odometrijski podaci koji se pročitaju s ROS poruke za odometriju. Iako je ovdje korišten proporcionalni regulator, greška kod zakreta za definirani je dosta velika. Jedan od uzroka takve greške mogu biti netočni podaci koji se šalju s enkodera mobilnog robota. Drugi uzrok može biti primjena lošeg regulatora čime se greška povećava.

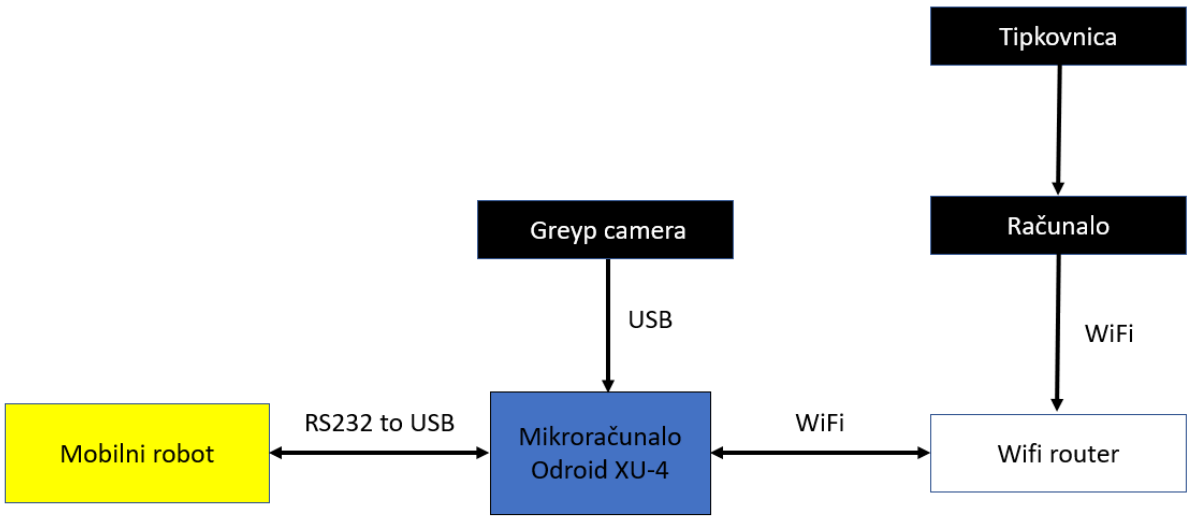
```
void setDesiredOrientation (double desired_angle_radians){
    EulerAngles angle;
    double Kp = 1.04;
    double relative_angle_radians = Kp * (desired_angle_radians - angle.yaw);
```

Jedina razlika između načina pisanja ROS čvorova za upravljanje mobilnim robotom preko simulatora i na realnom robotu je objavljivanje podataka na temu ili pretplaćivanje na određenu temu. Kod simulacije smo objavljivali brzinu na temu */turtle1/cmd_vel* dok se ovdje objavljuje na temu */RosAria/cmd_vel*. Također, razlika je i u frekvenciji kojom se šalju poruke u ROS-u. Na simulatoru je mogući široki raspon frekvencija bez ograničenja dok je kod realnog mobilnog robota bitno da najmanja frekvencija slanja i primanja poruka bude 2Hz.

```
int main(int argc, char* argv[])
{
    ros::init(argc, argv, "test_robot_movement");
    ros::NodeHandle n;
    velocity_publisher = n.advertise<geometry_msgs::Twist>("/RosAria/cmd_vel", 10);
    pose_subscriber = n.subscribe("/RosAria/pose", 10, poseCallback);
}
```

8.1.4. Udaljeno upravljanje

Upravljanje robotom preko računala koje je spojeno s mikroračunalom preko WiFi mreže predstavlja način udaljenog upravljanja. Računalo postaje udaljeni kontroler i robot se giba u skladu s naredbama koje korisnik pritišće na tipkovnici. Takav način upravljanja s određene udaljenosti zove se teleoprativno upravljanje. Iako se danas pod pojmom mobilni robot misli na robote koji su potpuno autonomni i sami donose odluke u različitim situacijama, još uvijek postoji primjena ručnog udaljenog upravljanja gdje je ljudsko usmjeravanje bolje rješenje zbog niza faktora. Upravljanje robotom izvodit će se kao i kod primjene translacijskog i rotacijskog gibanja tako da se šalju reference brzine u obliku poruke *geometry_msgs/Twist* na temu */RosAria/cmd_vel*.



Slika 61. Shematski prikaz ručnog udaljenog upravljanja

Pomoću blokovskog dijagrama ručnog udaljenog upravljanja prikazuje se shematski način izvođenja komunikacije. Računalo prima informacije od korisnika preko tipkovnice i šalje te vrijednosti preko WiFi mreže koristeći TCP/IP protokol na WiFi router. Nakon toga router šalje informacije na mikroračunalo Odroid XU-4 na kojemu je pokrenut čvor za slanje referentne brzine ovisno o tome koja je tipka na tipkovnici pritisnuta. Mikroračunalo zatim šalje vrijednosti u mikrokontroler robota koji pokreće motore i mobilni robot se kreće. Također, dobiva se i slika s kamere koju Odroid XU-4 šalje na router kako bi se slika mogla prikazati na udaljenom računalu s kojeg se odvija upravljanje.

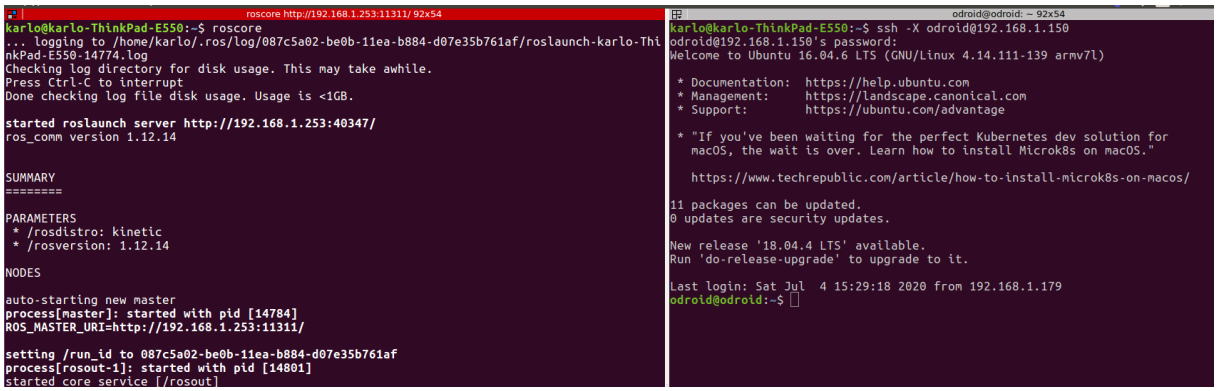
Programski kod za upravljanje preko tipkovnice preuzet je s *rosaria_client* repozitorija, modificiran i uspješno implementiran na mikroračunalo. Program je napisan u obliku čvora *teleop_twist_key.cpp* tako da su prvo definirane vrijednosti tipki koje će se pritisnuti na tipkovnici kako bi se znalo koja tipka je pritisnuta u određenom trenutku. Nakon toga kreirana je klasa u kojoj se nalaze dvije funkcije. Jedna je za inicijalizaciju komponenti brzine mobilnog robota na vrijednost nula te vrijednosti kojima se može povećavati ili smanjivati brzina mobilnog robota. Koristeći *switch case* tvrdnju, gleda se tipka koja je pritisnuta i šalje se referenca brzine u mobilni robot. Na kraju se inicijalizira čvor i pokreće program s *teleop_RosAria* funkcijom. Detaljan kod može se vidjeti na kraju u prilogu.

Prvi korak ručnog udaljenog upravljanja je da se sustav i način komunikacije postavi na odgovarajući način. Prije smo već naveli da je moguće postojanje samo jednog aktivnog ROS Mastera koji je odgovoran za komunikaciju između čvorova. Tako je i kada se ROS upravljanje radi na dva različita uređaja, jedan na laptopu a drugi na mikroračunalu. Dakle, ROS sustav se pokreće samo jednom i to će biti napravljeno na laptopu. Njemu ćemo postaviti varijablu *ROS_MASTER_URI* unutar *.bashrc* datoteke kojom se definira pokretanje ROS-a preko naredbe *roscore*. Moraju se definirati i posebne varijable kako bi ROS znao koji uređaj pokreće koji čvor tako da se definira IP adresa pojedinog uređaja. Prvi dio na slici 62. je dio konfiguracije na laptopu, a drugi dio je konfiguracija na Odroidu XU-4.

```
export ROS_MASTER_URI=http://localhost:11311/
export ROS_HOSTNAME=192.168.1.253
export ROS_IP=192.168.1.253

export ROS_MASTER_URI=http://192.168.1.253:11311/
export ROS_HOSTNAME=192.168.1.150
export ROS_IP=192.168.1.150
```

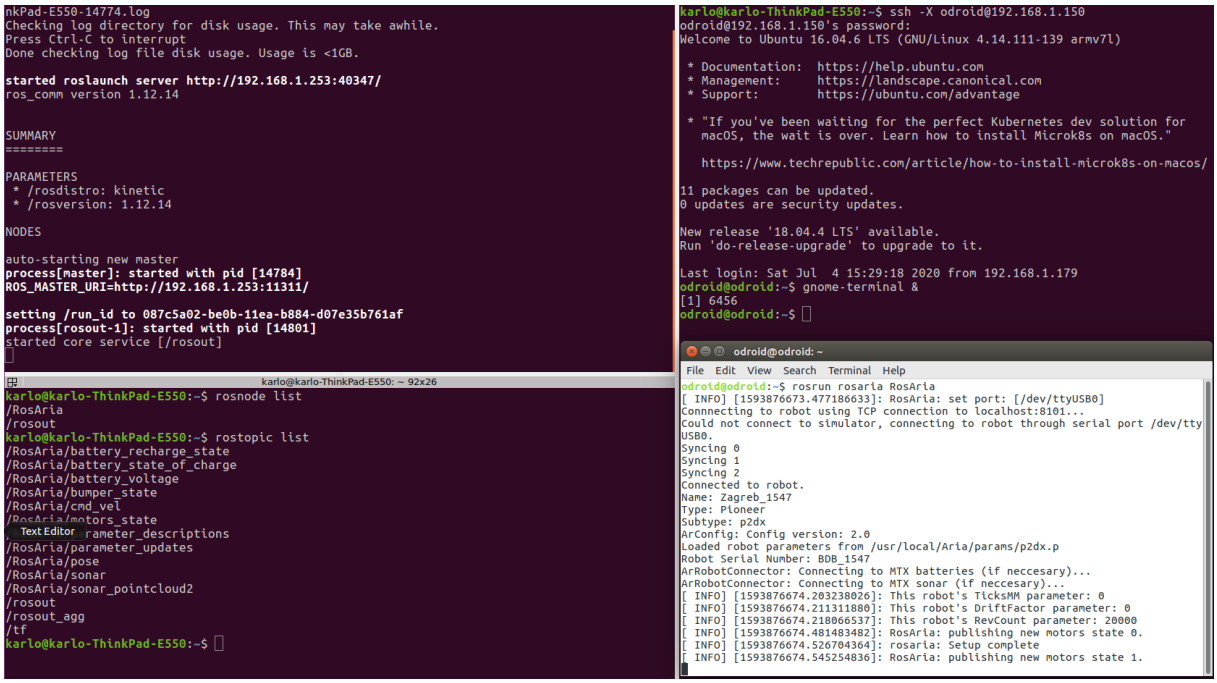
Slika 62. Konfiguracija datoteka za pokretanje ROS-a na dva različita uređaja

Testiranje funkcionalnosti udaljenog upravljanja izvodilo se tako da se prvo pokrenuo ROS na glavnom računalu i spojilo se na mikroračunalo pomoću Linux funkcionalnosti SSH (*eng. Secure Shell*). SSH koristi internet protokol i omogućuje udaljeno upravljanje za različita računala i servere. Naredbom *ssh -X odroid@192.168.1.150* spajamo se na mikroračunalo. Bitno je da se upiše točna internet adresa uređaja na koji se spajamo kako bi povezivanje bilo uspješno. Ako sve funkcionira kako treba dobiti će se rezultat kao na slici 63.



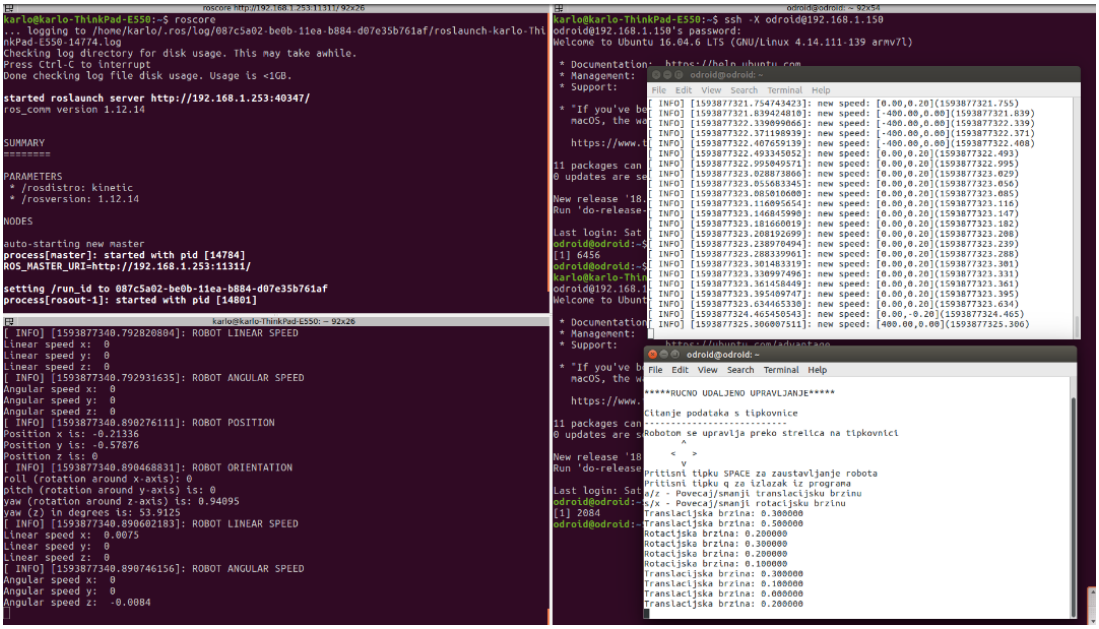
Slika 63. Spajanje na mikroračunalo

Sada kada smo spojeni na mikroračunalo imamo pristup otvaranju različitih terminala pomoću kojih ćemo pokretati različite čvorove. Naredbom *gnome terminal &* otvara se terminal prozor s bijelom pozadinom kako bi se moglo razlikovati procese koji se pokreću na laptopu od onih na mikroračunalu. Sada pokrećemo čvor RosAria i dobivamo rezultat na slici 64.



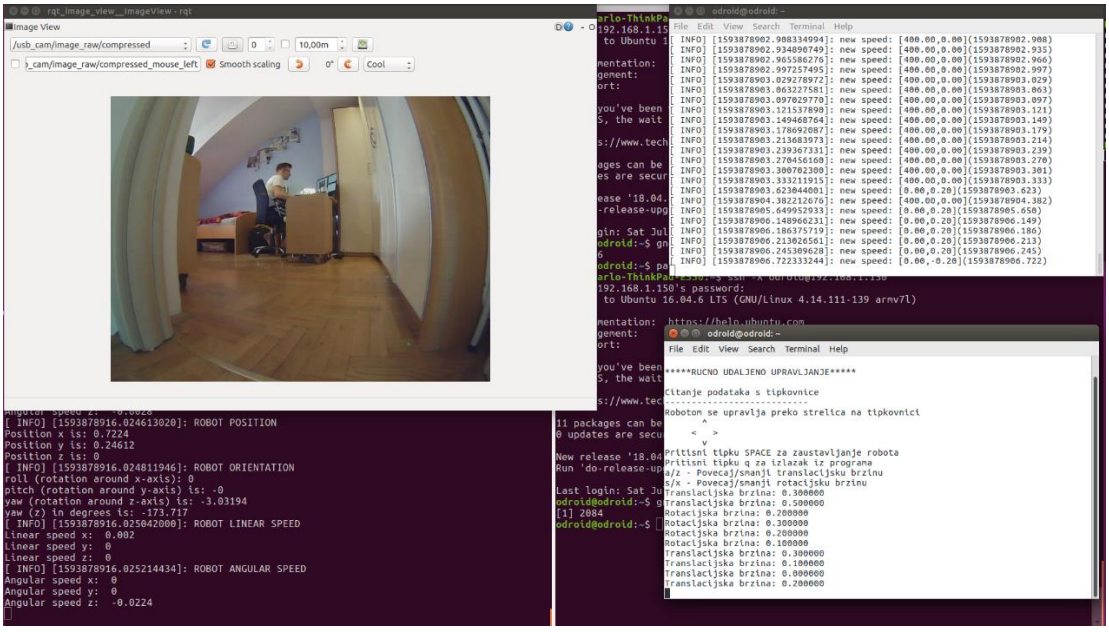
Slika 64. Pokretanje čvora RosAria na mikroračunalu

U desnom donjem prozoru sa slike 64. prikazan je ispis tema koje koristi čvor RosAria što je dokaz da su dva uređaja u ROS-u uspješno povezana. Sada se može pokrenuti čvor za teleoperativno upravljanje s naredbom *rosrun mobile_robot teleop_twist_key*. Na laptopu ćemo pokrenuti čvor kojime će se pratiti odometrijski podaci koji se šalju s mobilnog robota kako bi mogli pratiti poziciju i brzinu robota. Uspješno pokretanja čvorova vidi se na slici 65.



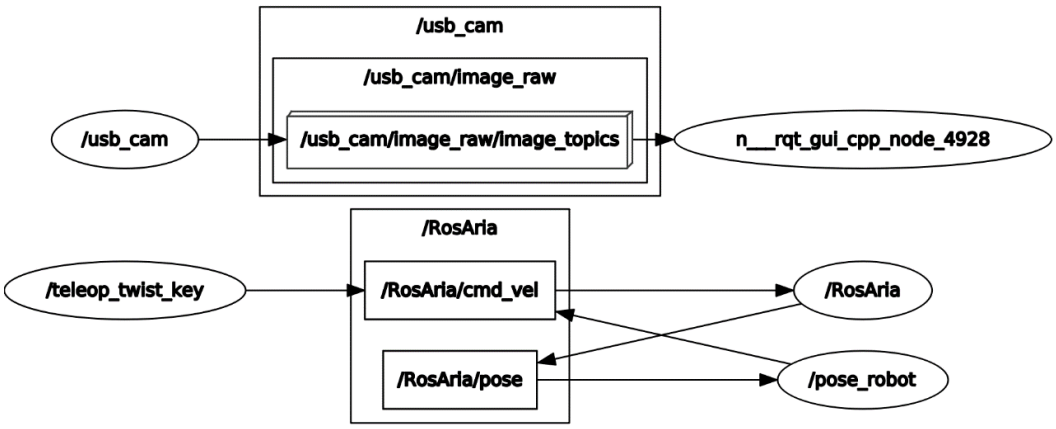
Slika 65. Pokretanje čvorova za udaljeno upravljanje i odometrijske podatke

Pokrenuti ćemo i čvor za dobivanje slike s kamere kako bi cijelo vrijeme vidjeli kuda idemo prilikom upravljanja robota što daje rezultat kao na slici 66. Slika koju dohvaćamo s kamere na laptopu je rezolucije 1920 puta 1080 piksela što je maksimalna rezolucija kamere.



Slika 66. Pokrenuti čvor za dohvaćanje slike s kamere i prikazan na zaslonu od laptopa

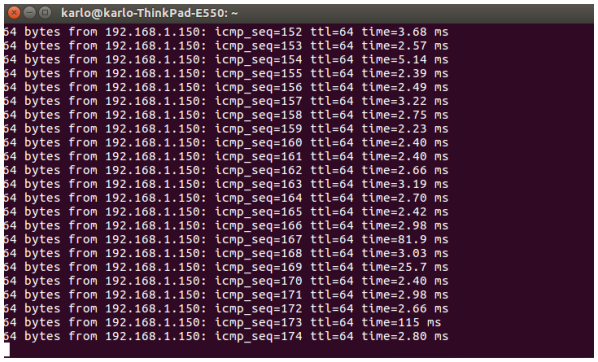
Radi lakšeg razumijevanja način na koji je napravljeno udaljeno upravljanje grafički će se prikazati svi trenutno pokrenuti čvorovi korišteni za ručno udaljeno upravljanje pomoću naredbe *rosrun rqt_graph rqt_graph*. Na slici 67. čvorovi su prikazani u obliku elipsa kao što su čvorovi *teleop_twist_key* i *pose_robot*, dok su ROS teme /usb_cam, /RosAria/cmd_vel prikazane u obliku kvadrata.



Slika 67. Grafički prikaz ROS čvorova i tema za udaljeno upravljanje

Vidimo da čvor teleop_twist_key objavljuje referencu brzine na temu */RosAria/cmd_vel* s koje onda podatke čita RosAria čvor i upravlja robotom. Čvor *pose_robot* se pretplatio na temu */RosAria/pose* i prikuplja odometrijske podatke, ali isto tako može objavljivati i na temu */RosAria/cmd_vel*.

Nakon testiranja ručnog udaljenog upravljanja dobiveni su jako dobri rezultati. Ovako dobar rezultat posljedica je stabilne veze i zanemarivo malog gubitka podataka prilikom korištenja bežične mreže. Ako je bežična mreža dovoljno dobra može se primijeniti ovakav način upravljanja. Na slici 68. vidimo da je tijekom testiranja upravljanja došlo samo dvaput do značajnijeg gubitka podataka što nije značajno utjecalo na komunikaciju.

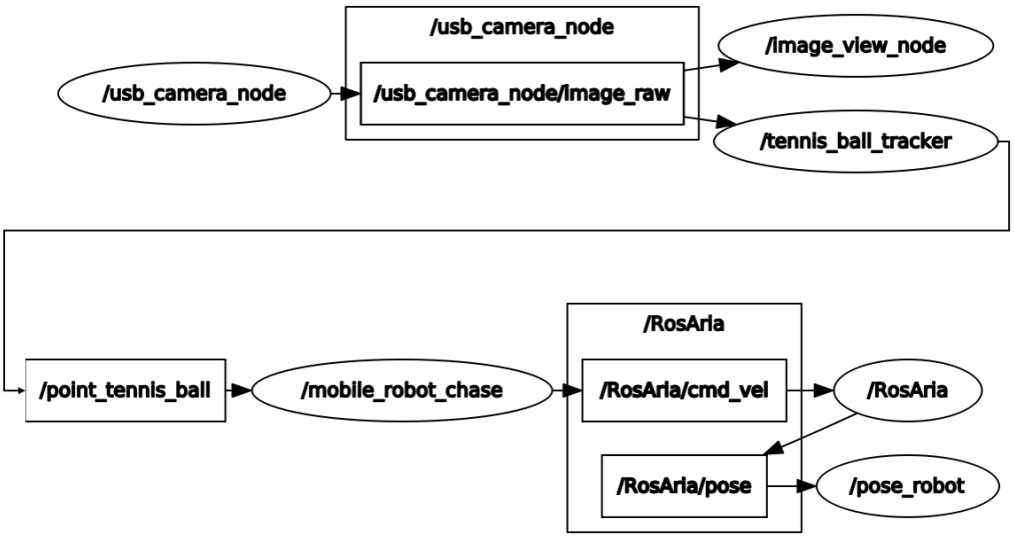


Slika 68. Rezultati prijenosa podataka putem bežične mreže

8.2. Autonomno upravljanje

Autonomni mobilni robot je onaj koji može percipirati okoliš u kojem se nalazi, donijeti odluke ovisno o tome što je percipirao te koji je isprogramiran tako da prepozna i donese odluku o gibanju unutar okoliša u kojem se nalazi. Ovdje će se prikazati implementacija jednostavnog algoritma radi testiranja autonomnog upravljanja mobilnog robota tako da će robot pratiti definirane objekte zadanih dimenzija. Također, iskoristit će se gotovo sve funkcije koje su razvijane i testirane kroz rad bez kojih izrada autonomnog upravljačkog algoritma ne bi mogla biti moguća.

Tok programa izvodi se na sljedeći način. Prvo čvor *usb_camera_node* uzima sliku s kamere i objavljuje ju na temu */usb_camera_node*. Čvor *image_view_node* pretplaćuje se na temu od kamera čvora imena */usb_camera_node/image_raw* kako bi prikazao sliku s kamere, dok čvor *tennis_ball_tracker* se pretplaćuje na istu temu kako bi mogao napraviti obradu slike i dobiti koordinate objekta sa slike. Nakon što dobije koordinate sa slike objavljuje ih u obliku vlastite poruke na temu */point_tennis_ball*. Zatim se čvor *mobile_robot_chase* pretplaćuje na temu */point_tennis_ball* kako bi dobio koordinate loptice te na temelju logike u programu objavljuje referencu brzine na temu */RosAria/cmd_vel*. *RosAria* čvor se pretplaćuje na temu /RosAria kako bi mogao poslati podatke za pomicanje motora u mikrokontroler na mobilnom robotu. Pozicija robota prikazuje se preko čvora *pose_robot* koji se pretplaćuje na temu */RosAria/pose* i prikazuje podatke o poziciji robota i brzini tijekom autonomnog upravljanja mobilnim robotom.



Slika 69. Grafički prikaz ROS čvorova korištenih kod autonomnog upravljanja

8.2.1. Obrada slike

Obrada slike odvija se u ROS čvoru *tennis_ball_tracker* koristeći biblioteku OpenCV i vizijski sustav opisan u poglavlju 7. Prvo je potrebno uključiti sve biblioteke koje će se koristiti kod obrade slike i definirati parametre kamere. Parametre koje definiramo su fokalna duljina kamere, stvarni promjer loptice, udaljenost kamere od podloge te rezolucija slike.

```
#define RGB_FOCAL_LEN_MM 138.90625  // fokalna duljina kamere u mm
#define BALL_DIAM_MM 900.0          // promjer lopte u mm
#define CAMERA_HEIGHT_MM 260.0      // udaljenost kamere od podloge u mm
#define IMG_HEIGHT_PX 480.0         // u pikselima
#define IMG_WIDTH_PX 640.0          // u pikselima
```

Zatim zadajemo ime prozora u kojemu će se prikazivati slika s kamere, varijable za spremanje koordinata x i y, radijus te konstantu π koja će se koristiti za izračunavanje površine prepoznatog objekta.

```
static const std::string OPENCV_WINDOW = "Image window";
static int x, y, radius;
const double PI = 3.14159265359;
```

Radi lakše organizacije programskog koda, kreirana je klasa *BallDetector* u kojoj definiramo čvorove za transport slike putem ROS-a, čvorove za pretplaćivanje na informacije, čvor za objavljivanje na određenu temu, čvor za objavljivanje koordinata robota u vlastitu poruku i na kraju varijabla za vrijeme pojavljivanja loptice na slici kamere.

```
class BallDetector {
private:
    ros::NodeHandle nodeHandle_;
    image_transport::ImageTransport imageTransport_;
    image_transport::Subscriber imageSub_;
    image_transport::Publisher imagePub_;
    ros::Publisher coordPub_;
    mobile_robot::ballCoordinates coordinates;
    double t0_ ;
```

Sada moramo inicijalizirati sve ROS čvorove tako da im definiramo teme na koje će objavljivati određene podatke te teme s kojih će preuzimati podatke.

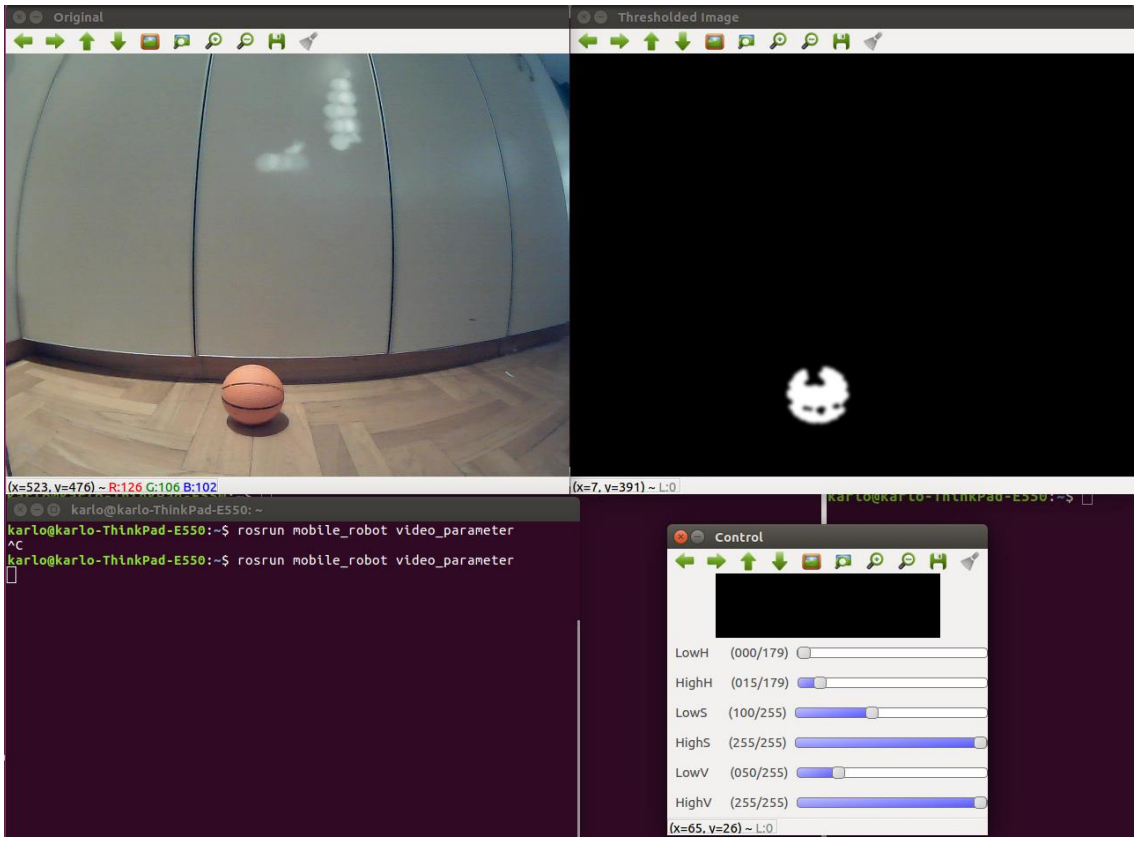
```
public:
    BallDetector() : imageTransport_(nodeHandle_)
    {
        //Pretplati se na ulazni video i objavi sliku na izlaz
        imageSub_ = imageTransport_.subscribe(
            "/usb_cam/image_raw", 10, &BallDetector::imageCallback, this);
        imagePub_ = imageTransport_.advertise("/image_converter/output_video", 10);
```

```
        coordPub_ = nodeHandle_.advertise<mobile_robot::ballCoordinates>
                    ("point_tennis_ball", 1000);
        t0_ = ros::Time::now().toSec();
        cv::namedWindow(OPENCV_WINDOW, 0);
    }
```

Definirani je i destruktor kojime se nakon prekida rada ROS čvora isključuju svi prozori na kojima se prikazuje slika s kamere i obrada slike.

```
    ~BallDetector()
    {
        cv::destroyWindow(OPENCV_WINDOW);
    }
```

Obrada slike odvija se pozivom funkcije *imageCallback()* koja kao argument prima trenutni okvir slike u obliku poruke sensor_msgs/Image. Prvo se instanciraju svi ROS slikovni i OpenCV objekti, te se definiraju parametri za prepoznavanje željene boje loptice. Parametri se određuju pomoću čvora *video_parameter.cpp* u kojemu se prikazuje slika s kamere. Na sliku su primijenjene metode obrade slike kako bi se izolirala iz slike samo loptica određene boje. Također definira se vektor u koji će se spremati x i y koordinate te radijus loptice. Na slici 70. određuju se parametri za obradu slike pomoću čvora *video_parameter.cpp*.



Slika 70. Određivanje parametara za obradu slike

```
void imageCallback(const sensor_msgs::ImageConstPtr &msg) {
    cv_bridge::CvImagePtr cvPtr;
    cv_bridge::CvImagePtr cvGrayPtr;
    cv::Mat rgbIMG, hsvIMG, thresholdedIMG, lowerBoundColor,
            upperBoundColor, reverseMask;

    //orange ball tracking
    int iLowH = 0;
    int iHighH = 60;
    int iLowS = 100;
    int iHighS = 255;
    int iLowV = 54;
    int iHighV = 255;

   // 3 elementa vektora, ovdje će se spremati koordinate x i y, te radijus loptice
   // prosljeđuje se u funkciju HoughCircles()
    std::vector<cv::Vec3f> v3fCircles;
```

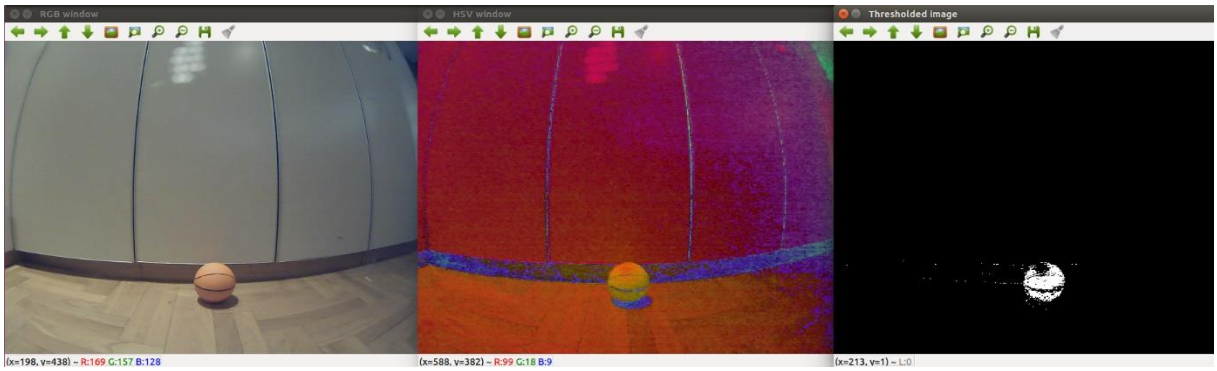
Prije obrade slike potrebno je instancirati kopiju objekta na kojem će se raditi obrada slike kako bi mogli uspoređivati rezultate s originalnom slikom.

```
    try {
        cvPtr = cv_bridge::toCvCopy(msg, sensor_msgs::image_encodings::BGR8);
        rgbIMG = cvPtr->image;
        // cv::imshow("RGB window", rgbIMG);
```

Sada slijedi obrada slike. Prvo prebacimo sliku iz RGB područja boja u HSV područje kako bi lakše mogli napraviti obradu slike. Problem koji se često javlja kod primjene HSV područja je osjetljivost na količinu svjetlosti. Poželjno je da ne postoji niti previše niti premalo količine svjetla, već optimalna količina svjetlosti. Nakon toga upotrijebimo prethodno dobivene parametre pomoću funkcije *inRange()* kojom se definira raspon boje za prepoznavanje.

```
        // Pretvaranje RGB slike u HSV područje
        cv::cvtColor(rgbIMG, hsvIMG,  cv::COLOR_BGR2HSV);

        // Primjena praga kako bi se definirao raspon boje
        // loptice koju zelimo prepoznati
        cv::inRange(hsvIMG,
                    cv::Scalar(iLowH, iLowS, iLowV),
                    cv::Scalar(iHighH, iHighS, iHighV),
                    thresholdedIMG);
```

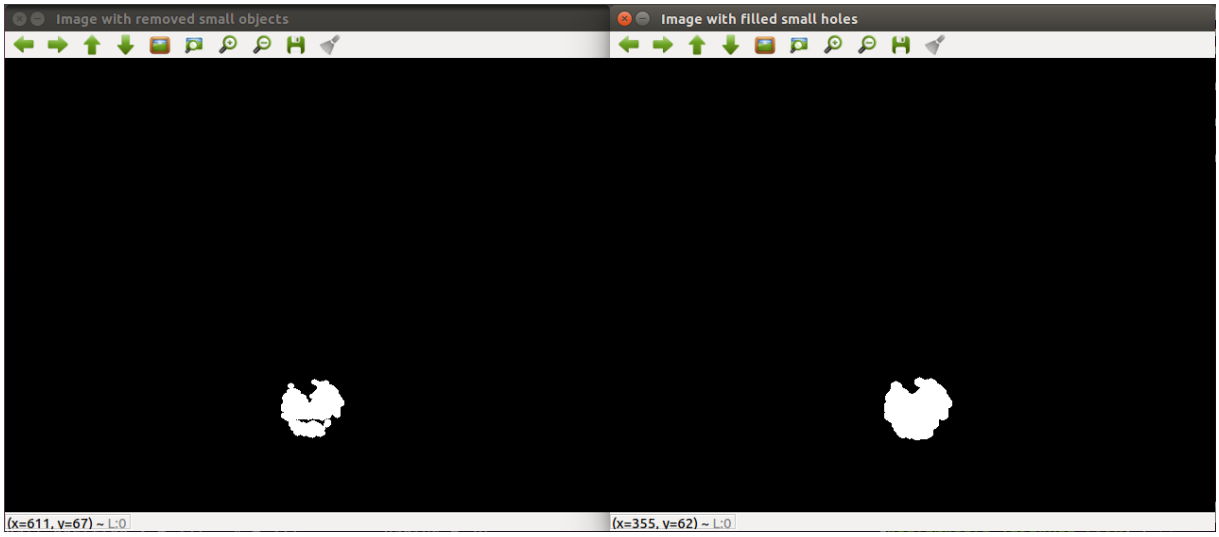
Prebacivanje u slike s kamere u HSV područje i primjena praga raspona boje koji želimo izdvojiti iz slike prikazano je na slici 71.



Slika 71. Prebacivanje slike iz RGB u HSV zapis uz primjenu obrade slike

Implementacijom različitih filtera uklanja se šum sa slike te se jasno definiraju granice lopice. U posebnom prozoru prikazuje se slika s korištenim filterima tako da se prvo uklone manji objekti definirane vrijednosti te se popune praznine preostalih malih objekata što se vidi na slici 72.

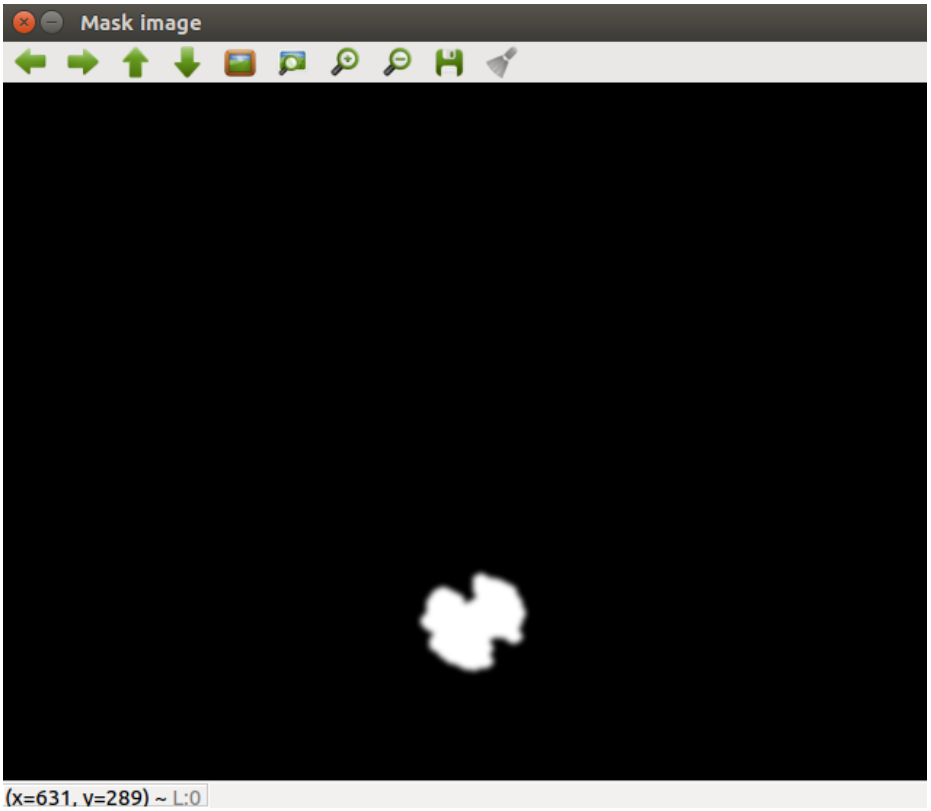
```
// Uklanjanje malih objekta na slici kako bi eliminirali šum
 cv::erode(thresholdedIMG, thresholdedIMG,
          cv::getStructuringElement(cv::MORPH_ELLIPSE, cv::Size(5, 5)) );
 cv::dilate( thresholdedIMG, thresholdedIMG,
          cv::getStructuringElement(cv::MORPH_ELLIPSE, cv::Size(5, 5)) );

 // Popunjavanje preostalih malih objekata radi uklanjanja šuma
 cv::dilate( thresholdedIMG, thresholdedIMG,
          cv::getStructuringElement(cv::MORPH_ELLIPSE, cv::Size(5, 5)) );
 cv::erode(thresholdedIMG, thresholdedIMG,
          cv::getStructuringElement(cv::MORPH_ELLIPSE, cv::Size(5, 5)) );
```



Slika 72. Lijevi prozor prikazuje uklanjanje manjih objekata, dok desni prikazuje popunjavanje praznina koje tvore manji objekti.

Kada se popune sve praznine, primjeni se filter za zamućivanje slike kako bi se lakše mogao ocrtati rub loptice.

```
// efekt zamagljivanja svega osim loptice
 cv::GaussianBlur(thresholdedIMG, thresholdedIMG, cv::Size(9, 9), 2, 2);
 cv::imshow("Thresholded img", thresholdedIMG);
```



Slika 73. Zamućena slika loptice

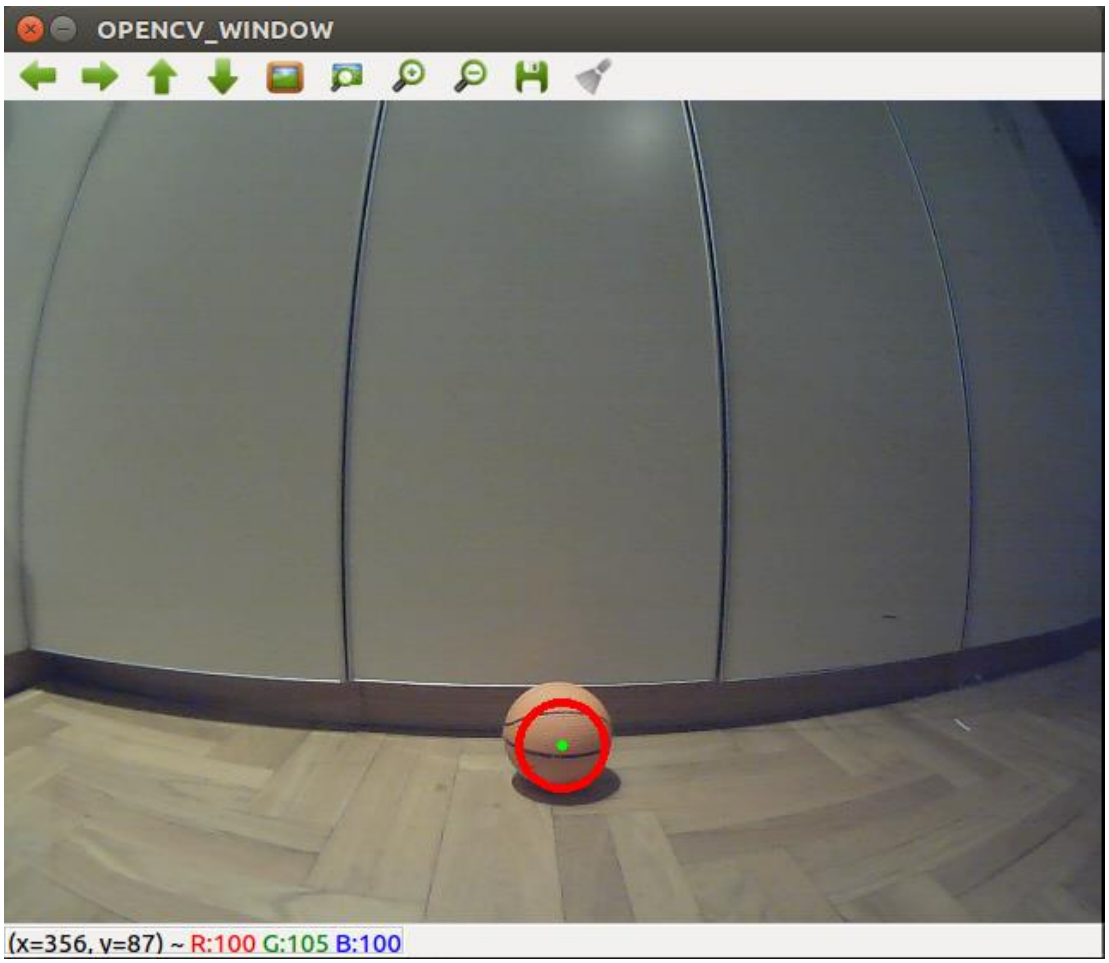
Ocrtavanje ruba loptice na slici izvodi se pomoću gotove metode *HoughCircles()* koja kao argumente uzima obrađenu sliku, vektor u koji će se spremiti podaci o loptici te različite paramatere za skaliranje slike. Nakon detekcije ruba prikazuje se nacrtani rub kao na slici 74.

```
cv::HoughCircles(thresholdedIMG, v3fCircles, CV_HOUGH_GRADIENT,
                 1, hsvIMG.rows/8, 100, 20, 0, 0);
 cv::imshow("Mask image", thresholdedIMG);
```

Kada se dobiju podaci s obrađene slike potrebno ih je odvojiti i spremiti u odgovarajuće varijable koje će se kasnije slati u poruci *BallCoordinates.msg*. Također, kako bi se eliminirale sve moguće manje površine od površine prepoznate loptice izračunava se površina detektirane loptice. Ako je površina detektiranog objekta veća od definirane vrijednosti čvor *tennis_ball_tracker* će objavljivati koordinate na temu */point_tennis_ball*.

```
    for (size_t i = 0; i < v3fCircles.size(); i++) {

            // pozicija loptice na slici u obliku
           // x i y koordinate te radius loptice
            x = static_cast<int>(round(v3fCircles[i][0]));
            y = static_cast<int>(round(v3fCircles[i][1]));
            radius = static_cast<int>(round(v3fCircles[i][2]));
            cv::Point center(x, y);
            double area = 0.0;
            area = (pow(radius,2) * PI);
            // std::cout << "area of ball is: " << area << std::endl;

            if (area > 2000)
            {
                coordinates.x = x;
                coordinates.y = y;
                coordPub_.publish(coordinates);
```

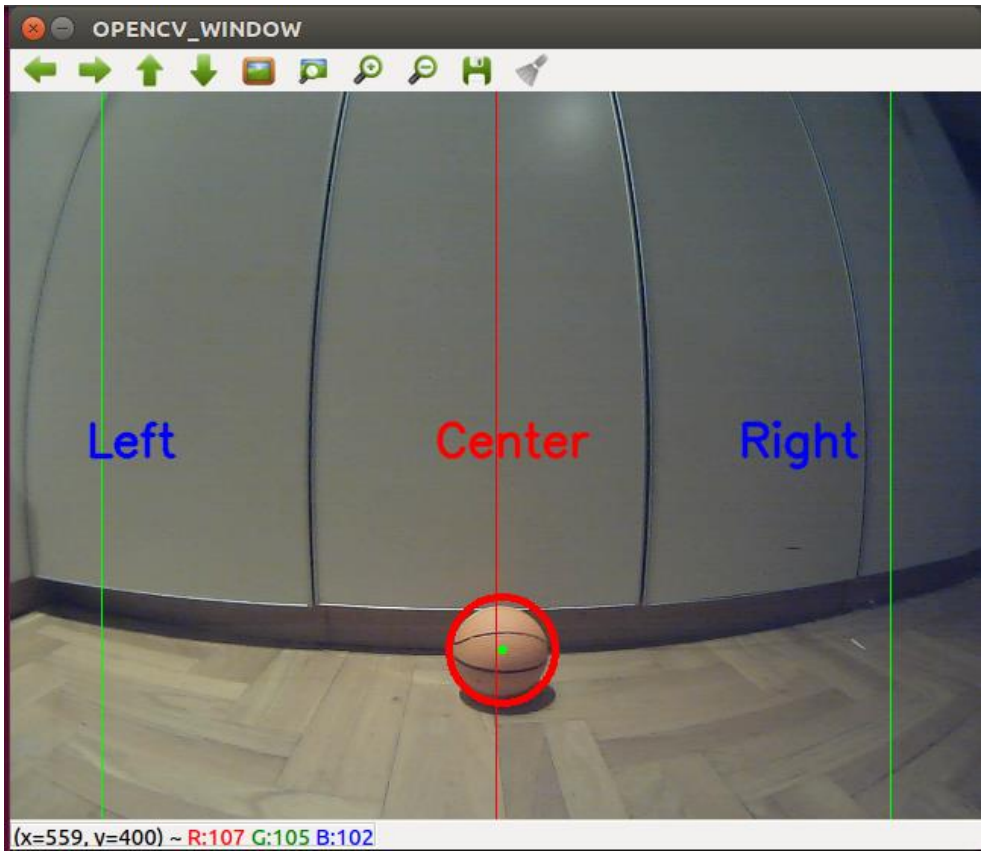
Na kraju se nacrta središte i vanjski rub prepoznate loptice kako bi se prikazala obrađena slika u posebnom prozoru.



Slika 74. Nacrtani i prepoznati rub loptice

Sliku podijelimo linijama tako da se definira područje lijevog, središnjeg i desnog dijela slike kako bi znali u kojem je području loptica.

```
//Podjela slike na lijevi i desni te centralni dio
cv::Point pt1, pt2,pt3,pt4,pt5,pt6;

// Centralno područje
pt1.x = IMG_WIDTH_PX / 2;
pt1.y = 0;
pt2.x = IMG_WIDTH_PX / 2;
pt2.y = 480;

// Lijevo područje
pt3.x = (IMG_WIDTH_PX / 2) - CAMERA_HEIGHT_MM;
pt3.y = 0;
pt4.x = (IMG_WIDTH_PX / 2) - CAMERA_HEIGHT_MM;
pt4.y = 480;

// Desno područje
pt5.x = (IMG_WIDTH_PX / 2) + CAMERA_HEIGHT_MM;
pt5.y = 0;
pt6.x = (IMG_WIDTH_PX / 2) + CAMERA_HEIGHT_MM;
pt6.y = 480;
```



Slika 75. Prepoznata loptica s ocrtanim rubom u središnjem dijelu vidnog polja kamere

Konačno, takva slika se natrag pretvara u ROS poruku i objavljuje na temu Image. Inicijalizira se čvor te mu se daje ime radi pokretanja i rukovanja s njime u ROS-u

```
            imagePub_.publish(cvPtr->toImageMsg());
    }
};

int main(int argc, char* argv[])
{
    ros::init(argc, argv, "tennis_ball_tracker");
    ROS_INFO("Node is initialized");
    BallDetector ball;
    ros::spin();
    return 0;
}
```

8.2.2. Praćenje definiranih objekata

Drugi čvor *mobile_robot_chase* koristi se za primanje koordinata s obrađene slike te na temelju toga donosi odluku o gibanju robota. Dodatni čvor je napravljen kako bi se odvojila obrada slike od logike za upravljanjem gibanja robota. Kod je organiziran u klasu *RobotChase* u kojoj se prvo osigurava da je robot zaustavljen tako što se u konstruktoru klase postavljaju translacijska i rotacijska brzina na nulu.

```
    RobotChase(): linear_(0.0), angular_(0.0)
    {
        positionSub_ = nodeHandle_.subscribe("/point_tennis_ball", 10, &RobotChase::Coo
rdinatesCallback, this);
        x = ballCoordinates.x;
        y = ballCoordinates.y;
        time_detected_ = 0.0;
    }
```

Deklariraju se četiri funkcije. Prva je trenutak kada se prepozna loptica, dok preostale tri koriste osnovna translacijska i rotacijska gibanja koja smo prikazali kod ručnog upravljanja. Ovdje se može uočiti razlog zašto je prije bilo potrebno testirati upravljanje robotom, a to je sigurnost od pojavljivanja grešaka vezanih uz gibanje robota.

```
    void CoordinatesCallback (const mobile_robot::ballCoordinates::ConstPtr &msg);
    double BallDetected();

    void MoveForward();
    void MoveLeft();
    void MoveRight();
};
```

Glavni dio logike upravljanja robotom nalazi se u funkciji *CoordinatesCallback()* u kojoj se prikupljaju *x* i *y* koordinate loptice.

```
void RobotChase::CoordinatesCallback(const mobile_robot::ballCoordinates::ConstPtr &msg)
{
    ballCoordinates.x = msg->x;
    ballCoordinates.y = msg->y;
    x = ballCoordinates.x;
    y = ballCoordinates.y;
    time_detected_ = ros::Time::now().toSec();
```

Ovdje se implementira logika upravljanja lopticom. Prvo se funkcijom *BallDetected()* provjerava je li loptica u vidnom polju kamere. Zatim se provjerava vrijednost *x* koordinate. Ako je ona između vrijednosti 20 i 220 znači da se loptica nalazi na lijevoj strani i robot će skretati ulijevo. Ako se nalazi između 220 i 420 loptica se nalazi u središnjem dijelu i robot se giba prema naprijed te raspon područja između 420 i 640 predstavlja desni dio područja kamere i robot skreće udesno.

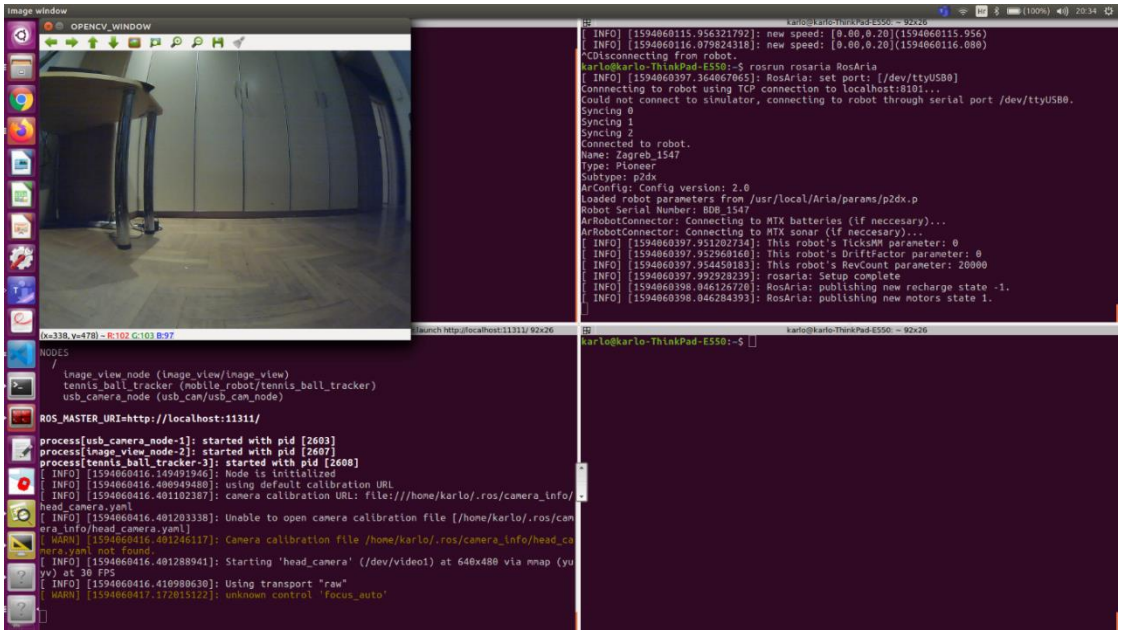
```
    if(!BallDetected()) {
        std::cout << "Object not found" << std::endl;
    }


    // LIJEVO PODRUČJE
    if(x > 20 && x < 220 ) {
        std::cout<<"LEFT TURN"<< " " << std::endl;
        MoveLeft();
    }

    // SREDIŠNJE PODRUČJE
    else if(x > 220 && x < 420) {
        std::cout<<"FORWARD"<< " " << std::endl;
        MoveForward();
    }

    // DESNO PODRUČJE
    else if(x > 420 && x < 640) {
        std::cout<<"RIGHT TURN"<< " " << std::endl;
        MoveRight();
    }
}
```

Bitno je naglasiti da se čvor za objavljivanje reference brzine ne smije staviti u konstruktor klase jer onda program neće funkcionirati. Mora se napraviti tako da se inicijalizira u glavnoj funkciji *main()* na prikazani način.

```
int main(int argc, char **argv) {
    ros::init(argc, argv, "mobile_robot_chase");
    ROS_INFO("Node is initialized");
    ros::NodeHandle n;
    velocityPub_ = n.advertise<geometry_msgs::Twist>("/RosAria/cmd_vel", 10);
    ros::Rate loop_rate(2);
    RobotChase robot;
    ros::spin();
    return 0;
}
```

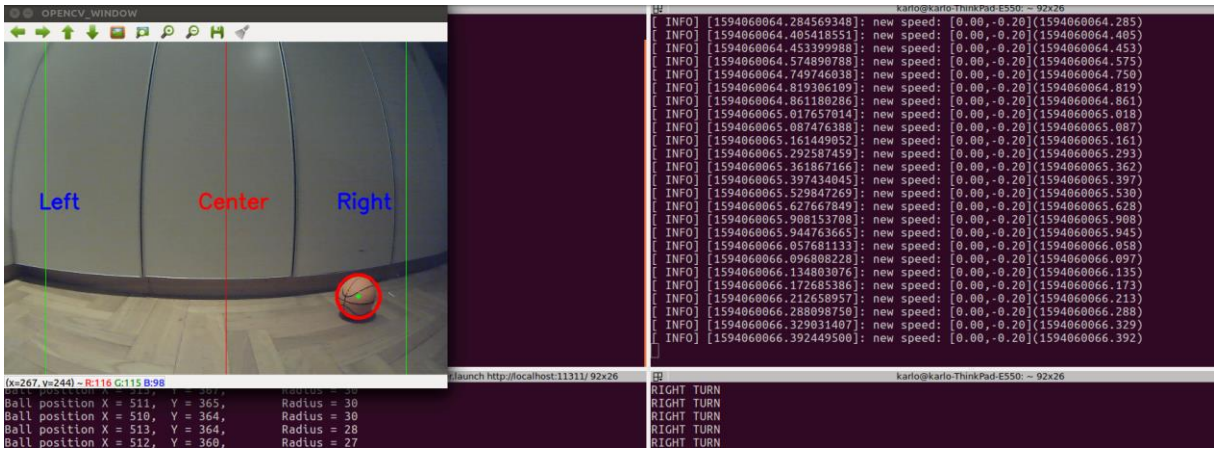
8.2.3. Testiranje autonomnog upravljanja

Autonomno upravljanje je testirano tako da je robot postavljen u okolinu u kojoj se može slobodno gibati. Prvi korak je da se sustav inicijalizira kao na slici 76. Ako loptice nema u vidnom polju kamere on se neće kretati jer je upravljanje napravljeno da mora postojati definirani objekt. Pokretanje vizijskog sustava nalazi se unutar *launch* datoteke zvane *ros_ball_tracker.launch* kojom se odjednom pokreće više ROS čvorova koji sudjeluju u komunikaciji kod vizijskog sustava.

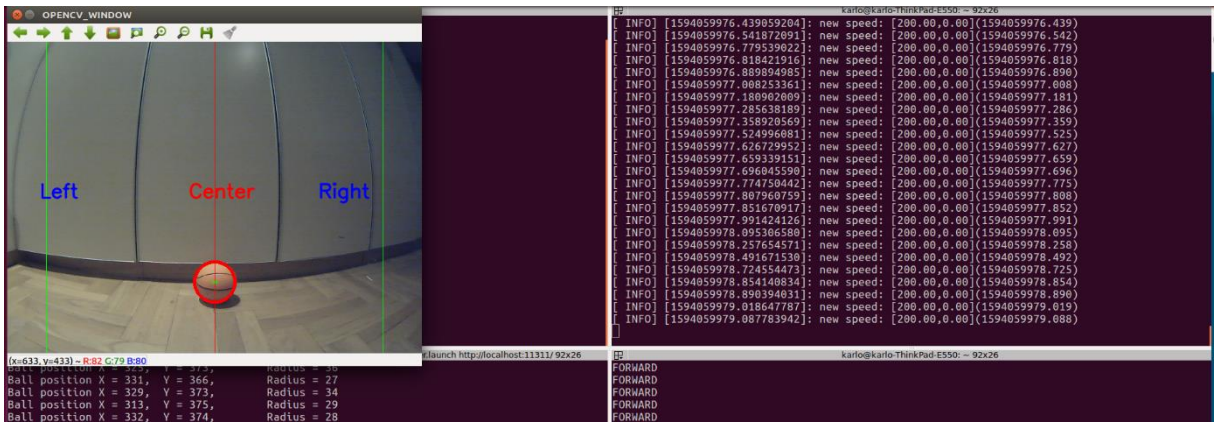


Slika 76. Inicijalizacija sustava za autonomno upravljanje

Autonomno upravljanje je prvo testirano na simulatoru tako da je loptica bila na jednom mjestu s dovoljno svjetlosti te se promatralo praćenje robota. Rezultati su prikazani na sljedećim slikama.



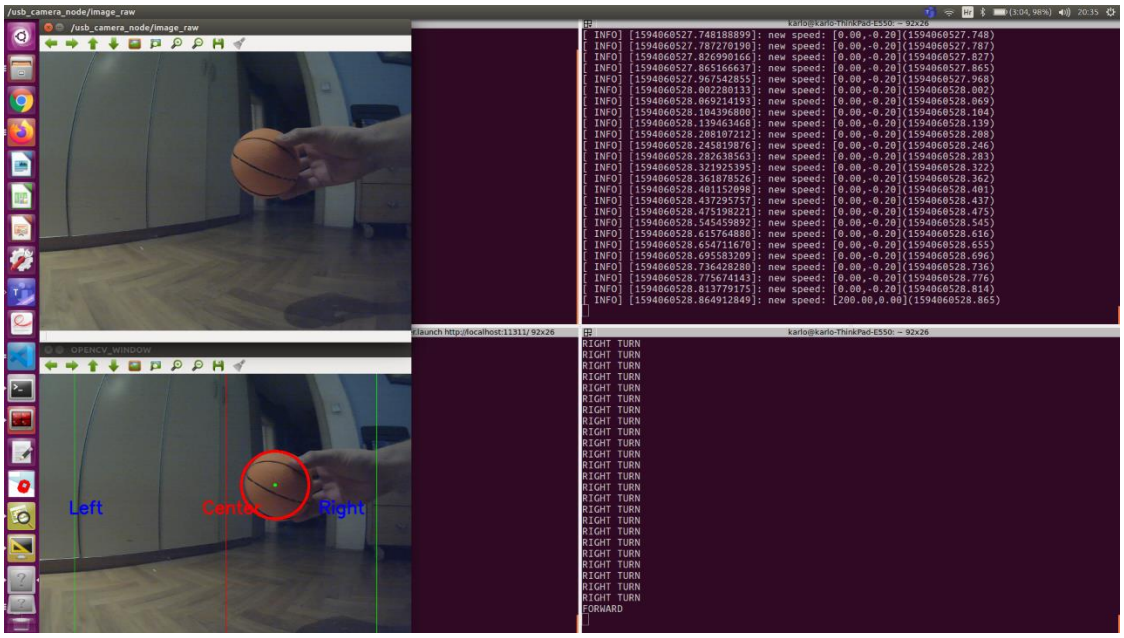
Slika 77. Loptica u desnom području kamere. Robot se kreće udesno



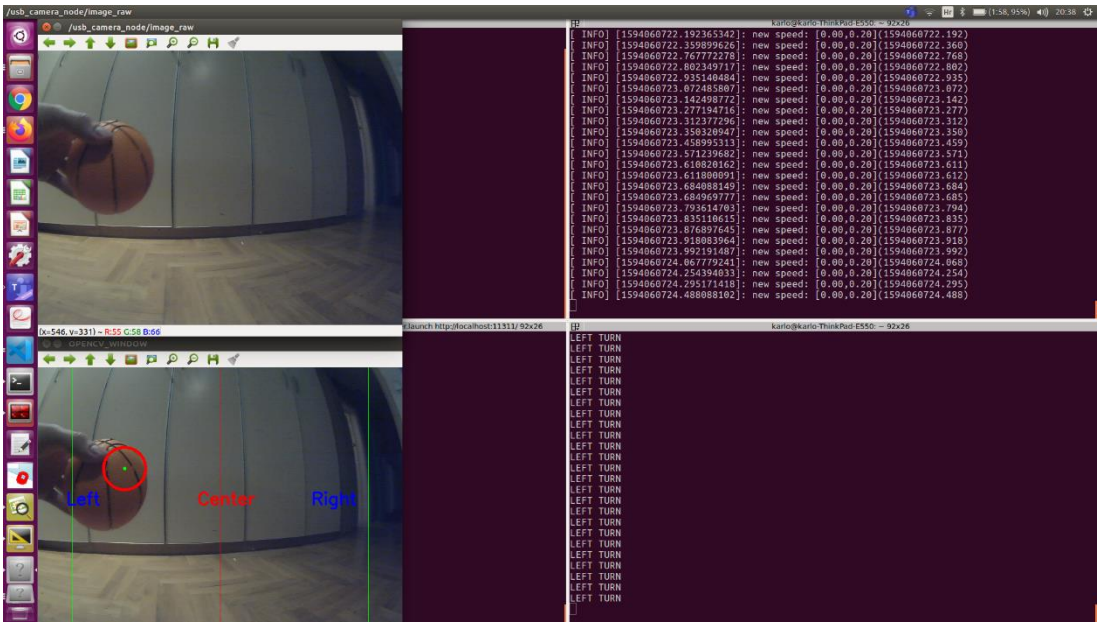
Slika 78. Loptica u središnjem području kamere. Robot se kreće ravno

Slika 79. Loptica u lijevom području kamere. Robot se kreće ulijevo

Rezultati dobiveni testiranjem na simulatoru su jako dobri te su primijenjeni na realnom mobilnom robotu što je prikazano na sljedećim slikama.



Slika 80. Loptica prelazi iz desnog u središnje područje s prilagodbom smjera kretanja robota



Slika 81. Loptica se nalazi u lijevom području i robot skreće ulijevo

Iz prikazanih rezultata može se zaključiti da algoritam za autonomno upravljanje daje dobre rezultate. Ovisno o području u kojem se loptica nalazi izvodi se gibanje robota. Jedini problem ovakvog načina obrade slike je što je potrebna dovoljna količina svjetlosti inače se neće moći prepoznati definirani objekti.

9. MOGUĆNOST POBOLJŠANJA SUSTAVA

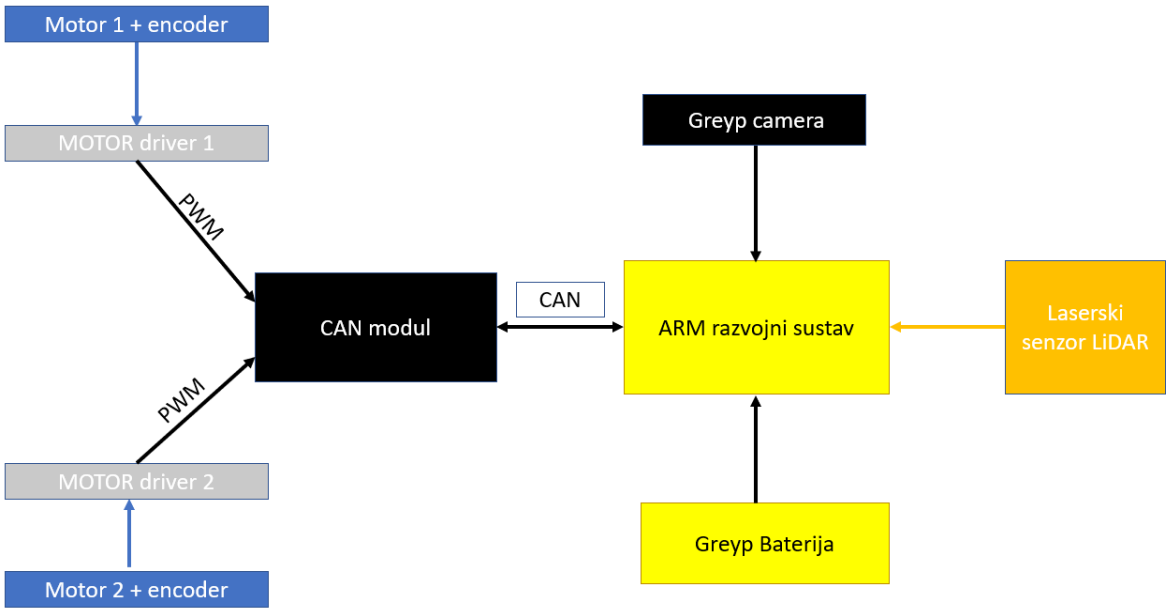
Sustav mobilnog robota koji je nadograđen s novim i boljim trenutnim dijelovima pokazao se kao dobra jedinica za testiranje jednostavnih upravljačkih algoritama mobilnog robota. S ciljem testiranja složenijih algoritama i upravljanja potrebno bi bilo dijelove, koji su ostali od prijašnje verzije mobilnog robota, zamijeniti novim i boljim dijelovima. Novi dijelovi moraju ispunjavati određene uvjete kako bi se trenutno upravljanje moglo izvoditi s novim sustavom. Jedna od ideja je da se potpuno uklone trenutni mikrokontroler i driveri za motore čime bi se potpuno izostavio sustav ARIE i ROSARIE. Tada bi se moralo napraviti potpuno novo programsko upravljanje motorima robota na koje bi se primijenio diferencijalni kinematski model mobilnog robota prikazan u prvom poglavlju. Ovdje će se prikazati idejno rješenje zamjene preostalih dijelova prijašnjeg mobilnog robota s novim i boljim dijelovima.

Prvi korak je uklanjanje mikrokontrolera od Pioneer robota i odabir ARM sustava koji je bolji od odabranog trenutnog mikroračunala Odroid XU-4. Jedan od zahtjeva je da mikroračunalo na sebi ima mikrokontroler čime bi se upravljačka jedinica od trenutne dvije svela na jednu. Kako baterija koristi CAN protokol, cilj je da se pronađe razvojni sustav koji bi podržavao CAN protokol jer bi se preko njega mogle pročitati sve vrijednosti koje su vezane uz podatke za bateriju. Ako bi se ostavili motori koji se trenutno nalaze na robotu, traženi sustav morao bi imati mogućnost slanja PWM signala. Jedna od mogućnosti je da se koristi Greyp integrirana pločica koja posjeduje sve navedene karakteristike. Ona bi predstavljala središnji sustav upravljanja gdje bi se programsko upravljanje napravilo u ROS-u. Jedini problem je što na pločici nema dovoljno ulaznih i izlaznih priključaka pa bi se trebao pronaći dodatni CAN modularni priključak na koji bi se spajale sve ostale komponente.

Drugi korak je da se integriraju bolji senzori. Trenutno se na robotu nalazi osam sonarnih senzora koji imaju jako puno ograničenja. Nemaju mogućnost rada u svim uvjetima okoliša, ne pokazuju dobre rezultate te zahtijevaju jako puno podešavanja. Za primjenu složenijih SLAM (*eng. simultaneous localization and mapping*) algoritama za lokalizaciju i mapiranje prostora potrebno je integrirati, uz postojeću kameru, dodatni laserski senzor. Prijedlog laserskog senzora je LiDAR (eng. *Light Detection and Ranging*) senzor koji koristi pulseve iz lasera kako bi dobio informacije iz okoline. Ovi podaci se kasnije mogu koristiti za kreiranje 3D modela te mapiranje objekata i prostora. Laserski LiDAR senzori danas postaju sve češći kod autonomnog upravljanja jer daju jako dobre rezultate u različitim uvjetima. Razlog zašto je ova vrsta senzora

bolja od sonarnih senzora, koji koriste radio i zvučne valove kod mjerenja udaljenosti predmeta, je jer mjere udaljenost predmeta pomoću brzine svjetlosti te mogu emitirati i do nekoliko tisuća pulseva u sekundi. LiDAR senzori imaju u sebi integriran i IMU (*eng. Inertial Measurement Unit*) senzor čime bi se dobili bolji odometrijski podaci. Odometrijski podaci koji se izračunavaju direktno iz enkodera često nisu dovoljno dobri za navigaciju i mogu sadržavati greške i smetnje. Za kompenzaciju grešaka prilikom gibanja robota koristili bi se žiroskop i akcelerometar s IMU senzora čime bi se mogao ostvariti bolji rezultati upravljanja.

Zadnji korak bi bio odabrati i integrirati nove drivere za motore kojima bi se mogla ostvariti komunikacija s motorima preko PWM signala. Enkoderski podaci bi se obrađivali direktno u driverima čime bi se ostvarila bolja točnost mjerenja pozicije robota. Idejna shema poboljšanja sustava prikazana je na slici 82.



Slika 82. Idejna shema poboljšanja trenutnog sustava mobilnog robota

10. ZAKLJUČAK

Mobilni robot predstavlja složeni mehatronički sustav koji se sastoji od niza podsustava povezanih u funkcionalnu upravljačku cjelinu. Za razumijevanje principa rada mobilnog robota potrebno je poznavanje različitih grana tehničkog područja kao što su mehanika, senzorika, elektronika, upravljanje i regulacija te razvoj programske podrške. Kroz diplomski rad potvrđena su ranije stečena znanja sa smjera mehatronike i robotike te su primijenjena na stvarnom upravljačkom sustavu.

Ideja je bila prikazati način integriranja sustava ARIA, koji se razvijao dugi niz godina s relativno mladim sustavom ROS koji se danas sve češće koristi jer se otkrivaju mogućnosti njegove primjene. Prikazani su načini povezivanja komponenata robota u smislenu cjelinu kako bi se ostvarila mogućnost upravljanja mobilnog robota. Nakon što su stari dijelovi robotskog sustava zamijenjeni novim dijelovima, integrirana je nova baterija koja je povećala vrijeme rada mobilnog robota. Na kraju izrađena je programska podrška u ROS-u za upravljanje pojedinim dijelovima mobilnog robota. Dijelovi osnovnih programa kasnije su povezani u veće cjeline koje su se iskoristile za razvoj upravljanja robotskog sustava. Prvo je upravljanje testirano na simulatoru koje se kasnije implementiralo na stvarnom robotskom sustavu. Kroz dva načina upravljanja, ručno i autonomno, prikazane su mogućnosti i način kako se primjenom robotskog operativnog sustava može upravljati složenim robotskim sustavima.

Upravljanje mobilnim robotom postaje područje sve većeg interesa i razvoja gdje se svakodnevno razvijaju novi načini prikupljanja informacija iz okoline. Razvojem tehnologije i bržeg načina komunikacije načini upravljanja prikazanih u radu postat će općeprihvaćeni. Ideja ovog rada bila je da se pokažu funkcionalnosti robotskog operativnog sustava primjenom različitih načina upravljanja mobilnim robotom. Iako je to relativno mlad i novi sustav, predstavlja prekretnicu u načinu upravljanja i izradi programske podrške za različite robotske sustave. Na početku je bio zamišljen kao razvojna platforma kojom će se izvoditi različita testiranja i simulacije prototipnih modela. Glavni cilj je bio postizanje razine apstrakcije programskog sustava kako bi se mogli dijelovi koda ponovno koristiti na drugim i sličnim sustavima. Trenutno je razvijena i nova verzija ROS sustava zvana ROS2 koja predstavlja bolju i stabilniju verziju sadašnjeg sustava. Nova verzija omogućit će brži razvoj robotskih sustava čime će doći do povećanja produktivnosti u mnogim industrijskim područjima.

LITERATURA

[1] https://internetofthingsagenda.techtarget.com/definition/mobile-robot-mobile-robotics, 01.06.2020

[2] https://journals.sagepub.com/doi/full/10.1177/1729881419839596, 01.06.2020.

[3] Corke, P: Robotics, Vision and Control, Springer, 2017.

[4] https://www.researchgate.net/publication/226964084_Industrial_Robotics, 03.06.2020.

[5] Mihel M, Bajd T, Ude A, Lenarčič J, Stanovnik A, Munih M, Rejc J, Šlajpah S. Robotics. Secnod Edition. Ljubljana: Springer; 2019.

[6] https://english.newsnationtv.com/science/news/nasa-mars-rover-2020-named-perseverance-255724.html, 04.06.2020.

[7] https://svrobo.org/svr-case-studies-savioke-solving-delivery-robots-as-a-service/, 04.06.2020

[8] https://www.google.com/search?q=boston+dynamics+dog&sxsrf=ALeKk03Tzqro2z1655TZh3Epbg4MbZ_hJg:1592145175071&source=lnms&tbm=isch&sa=X&ved=2ahUKEwiS7Nz8woHqAhUPhlwKHdSGDiIQ_AUoAXoECA0QAw&biw=1536&bih=710&dpr=1.25#imgrc=wyxMYTJg7Z1RsM, 04.06.2020.

[9] https://www.hydro-international.com/content/article/the-advancing-technology-of-auvs, 04.06.2020

[10] Patnaik S. Robot Cognition and Navigation. Bhubaneswar, Indija: Springer 2007.

[11] Pioneer 2 / PeopleBot Operations Manual, ActivMedia Robotics, 2001.

[12] Y Pyo, H Cho, R Jung, T Lim. ROS Robot Programming, From the basic concept to practical programming and robot application, 10.06.2020.

[13] https://www.ros.org/, 10.06.2020.

[14] L Joseph, J Cacace. Mastering ROS for robotics programming – Second edition. Packt 2017.

[15] https://zimo.dnevnik.hr/clanak/rimac-predstavio-novi-jos-bolji-greyp-elektricni-bicikl---553107.html, 12.06.2020.

[16] https://www.chipoteka.hr/artikl/79840/baterija-akumulatorska-12v-70-ah-151x65x98-mmfaston-48-yuasa-2400005207, 14.06.2020.

[17] https://www.refill-bar.hr/baterija-punjiva-sony-li-ion-18650-vtc5.html, 14.06.2020.

[18] https://www.amazon.co.uk/LC-Technology-LM2596-Voltmeter-Arduino/dp/B077K4XGVY, 14.06.2020.

[19] https://en.wikipedia.org/wiki/Conversion_between_quaternions_and_Euler_angles, 20.6.2020.

[20] http://wiki.ros.org/vision_opencv, 25.06.2020.

[21] L Joseph, J Cacace. ROS Robotics Projects. Packt 2017.

PRILOZI

I. Programski kod za upravljanje mobilnim robotom

II. CD-R disc

Programski kod za upravljanje mobilnim robotom

sensor_test.cpp:

```
#include <iostream>
#include <ros/ros.h>
#include <geometry_msgs/Twist.h>
#include <math.h>

void callbackSonar(const sensor_msgs::PointCloud::ConstPtr &msg) {
    ROS_INFO("Started reading sensor data" );

    float distance0 = 0.0;
    float distance1 = 0.0;
    float distance2 = 0.0;
    float distance3 = 0.0;
    float distance4 = 0.0;
    float distance5 = 0.0;
    float distance6 = 0.0;
    float distance7 = 0.0;

    distance0 = sqrt(pow(msg->points[0].x, 2) + pow(msg->points[0].y, 2));
    distance1 = sqrt(pow(msg->points[1].x, 2) + pow(msg->points[1].y, 2));
    distance2 = sqrt(pow(msg->points[2].x, 2) + pow(msg->points[2].y, 2));
    distance3 = sqrt(pow(msg->points[3].x, 2) + pow(msg->points[3].y, 2));
    distance4 = sqrt(pow(msg->points[4].x, 2) + pow(msg->points[4].y, 2));
    distance5 = sqrt(pow(msg->points[5].x, 2) + pow(msg->points[5].y, 2));
    distance6 = sqrt(pow(msg->points[6].x, 2) + pow(msg->points[6].y, 2));
    distance7 = sqrt(pow(msg->points[7].x, 2) + pow(msg->points[7].y, 2));

    ROS_INFO("Sensor position 0: ");
    std::cout << "Sonar reading x0: " << msg->points[0].x << std::endl;
    std::cout << "Sonar reading y0: " << msg->points[0].y << std::endl;
    std::cout << "Distance 0: " << distance0 << std::endl;

    ROS_INFO("Sensor position 1: ");
    std::cout << "Sonar reading x1: " << msg->points[1].x << std::endl;
    std::cout << "Sonar reading y1: " << msg->points[1].y << std::endl;
    std::cout << "Distance 1: " << distance1 << std::endl;

    ROS_INFO("Sensor position 2: ");
    std::cout << "Sonar reading x2: " << msg->points[2].x << std::endl;
    std::cout << "Sonar reading y2: " << msg->points[2].y << std::endl;
    std::cout << "Distance 2: " << distance2 << std::endl;

    ROS_INFO("Sensor position 3: ");
    std::cout << "Sonar reading x3: " << msg->points[3].x << std::endl;
    std::cout << "Sonar reading y3: " << msg->points[3].y << std::endl;
    std::cout << "Distance 3: " << distance3 << std::endl;
```

```
    ROS_INFO("Sensor position 4: ");
    std::cout << "Sonar reading x4: " << msg->points[4].x << std::endl;
    std::cout << "Sonar reading y4: " << msg->points[4].y << std::endl;
    std::cout << "Distance 4: " << distance4 << std::endl;

    ROS_INFO("Sensor position 5: ");
    std::cout << "Sonar reading x5: " << msg->points[5].x << std::endl;
    std::cout << "Sonar reading y5: " << msg->points[5].y << std::endl;
    std::cout << "Distance 5: " << distance5 << std::endl;

    ROS_INFO("Sensor position 6: ");
    std::cout << "Sonar reading x6: " << msg->points[6].x << std::endl;
    std::cout << "Sonar reading y6: " << msg->points[6].y << std::endl;
    std::cout << "Distance 6: " << distance6 << std::endl;

    ROS_INFO("Sensor position 7: ");
    std::cout << "Sonar reading x7: " << msg->points[7].x << std::endl;
    std::cout << "Sonar reading y7: " << msg->points[7].y << std::endl;
    std::cout << "Distance 7: " << distance7 << std::endl;

}

int main(int argc, char **argv)
{
    ros::init(argc, argv, "sensor_test");
    ros::NodeHandle nodeHandle;
    ros::Subscriber sonarSub = nodeHandle.subscribe("/RosAria/sonar", 1, callb
ackSonar);
    ros::spin();
}
```

pose_robot.cpp:

```
#include <ros/ros.h>
#include <geometry_msgs/Twist.h>
#include <geometry_msgs/Pose.h>
#include <nav_msgs/Odometry.h>
#include <cmath>

#define _USE_MATH_DEFINES

ros::Publisher velocity_publisher;
ros::Subscriber pose_subscriber;

geometry_msgs::Pose pose;
const nav_msgs::Odometry::ConstPtr pose_message;
const double PI = 3.14159265359;

struct Position
{
```

```
    double x, y, z;
};

struct Quaternion {
    double w, x, y, z;
};

struct EulerAngles {
    double roll, pitch, yaw;
};

struct LinearSpeed {
    double x, y, z;
};

struct AngularSpeed {
    double x, y, z;
};

double radians2degrees(double angle_in_radians){
    return (angle_in_radians *180.0) /PI;
}

void poseCallback(const nav_msgs::Odometry::ConstPtr & pose_message) {
    Position p;
    Quaternion q;
    EulerAngles angles;

    LinearSpeed linear;
    AngularSpeed angular;

    p.x = pose_message->pose.pose.position.x;
    p.y = pose_message->pose.pose.position.y;
    p.z = pose_message->pose.pose.position.z;

    q.x = pose_message->pose.pose.orientation.x;
    q.y = pose_message->pose.pose.orientation.y;
    q.z = pose_message->pose.pose.orientation.z;
    q.w = pose_message->pose.pose.orientation.w;

    linear.x = pose_message->twist.twist.linear.x;
    linear.y = pose_message->twist.twist.linear.y;
    linear.z = pose_message->twist.twist.linear.z;

    angular.x = pose_message->twist.twist.angular.x;
    angular.y = pose_message->twist.twist.angular.y;
    angular.z = pose_message->twist.twist.angular.z;

    // roll (x-axis rotation)
```

```
    double sinr_cosp = 2 * (q.w * q.x + q.y * q.z);
    double cosr_cosp = 1 - 2 * (q.x * q.x + q.y * q.y);
    angles.roll = std::atan2(sinr_cosp, cosr_cosp);

    // pitch (y-axis rotation)
    double sinp = 2 * (q.w * q.y - q.z * q.x);
    if (std::abs(sinp) >= 1)
        angles.pitch = copysign(PI / 2, sinp);
    else
        angles.pitch = std::asin(sinp);

    // yaw (z-axis rotation)
    double siny_cosp = 2 * (q.w * q.z + q.x * q.y);
    double cosy_cosp = 1 - 2 * (q.y * q.y + q.z * q.z);
    angles.yaw = std::atan2(siny_cosp, cosy_cosp);

    ROS_INFO("ROBOT POSITION");
    std::cout <<  "Position x is: " << p.x << std::endl;
    std::cout <<  "Position y is: " << p.y << std::endl;
    std::cout <<  "Position z is: " << p.z << std::endl;

    ROS_INFO("ROBOT ORIENTATION");
    std::cout << "roll (rotation around x-
axis): " << angles.roll << std::endl;
    std::cout << "pitch (rotation around y-
axis) is: " << angles.pitch << std::endl;
    std::cout << "yaw (rotation around z-
axis) is: " << angles.yaw << std::endl;
    std::cout << "yaw (z) in degrees is: " << radians2degrees(angles.yaw) << s
td::endl;

    ROS_INFO("ROBOT LINEAR SPEED");
    std::cout << "Linear speed x:  " << linear.x << std::endl;
    std::cout << "Linear speed y:  " << linear.y << std::endl;
    std::cout << "Linear speed z:  " << linear.z << std::endl;

    ROS_INFO("ROBOT ANGULAR SPEED");
    std::cout << "Angular speed x:  " << angular.x << std::endl;
    std::cout << "Angular speed y:  " << angular.y << std::endl;
    std::cout << "Angular speed z:  " << angular.z << std::endl;
}

int main(int argc, char* argv[])
{
    ros::init(argc, argv, "pose_robot");
    ros::NodeHandle n;
    pose_subscriber = n.subscribe("/RosAria/pose", 10, poseCallback);
    ros::spin();
}
```

image_pub_sug.cpp:

```
#include "ros/ros.h"
#include <image_transport/image_transport.h>
#include <cv_bridge/cv_bridge.h>
#include <sensor_msgs/image_encodings.h>
#include <opencv2/imgproc/imgproc.hpp>
#include <opencv2/highgui/highgui.hpp>

static const std::string OPENCV_WINDOW = "Image window";

class ImageConverter
{
private:
    ros::NodeHandle nh_;
    image_transport::ImageTransport it_;
    image_transport::Subscriber image_sub_;
    image_transport::Publisher image_pub_;
public:
    ImageConverter() : it_(nh_)
    {
        // Pretplati se na dolazni video preko teme image
        image_sub_= it_.subscribe("image", 1, &ImageConverter::imageCallback,
this);

        // Objavi video na temu /image_converter/output_video
        image_pub_ = it_.advertise("/image_converter/output_video", 1);
        cv::namedWindow(OPENCV_WINDOW);
    }

    ~ImageConverter()
    {
        cv::destroyWindow(OPENCV_WINDOW);
    }

    void imageCallback(const sensor_msgs::ImageConstPtr& msg)
    {
        cv_bridge::CvImagePtr cv_ptr;
        try
        {
            cv_ptr = cv_bridge::toCvCopy(msg, sensor_msgs::image_encodings::BG
R8);
        }
        catch(const std::exception& e)
        {
            ROS_ERROR("cv_bridge exception: %s", e.what());
            return;
        }
```

```
        // Nacrtaj krug na izlaznom videu
        if (cv_ptr->image.rows > 60 && cv_ptr->image.cols > 60 )
        {
            cv::circle(cv_ptr-
>image, cv::Point(50,50), 10, CV_RGB(0, 255, 0));
        }

        // Osvježi GUI prozor
        cv::imshow(OPENCV_WINDOW, cv_ptr->image);
        cv::waitKey(3);

        // Objavi modificirani video
        image_pub_.publish(cv_ptr->toImageMsg());
    }
};

int main(int argc, char* argv[])
{
    ros::init(argc, argv, "image_pub_sub");
    ROS_INFO("Testiranje cv_bridge s kamerom");
    ImageConverter ic;
    ros::spin();
    return 0;
}
```

image_pub_sub.launch:

```
<launch>
  <node name="usb_camera_node" pkg="usb_cam" type="usb_cam_node" output="scree
n" >
    <param name="video_device" value="/dev/video0" />
    <!-- <param name="video_device" value="/dev/video1" /> -->
    <param name="image_width" value="1280" />
    <param name="image_height" value="720" />
    <param name="pixel_format" value="yuyv" />
    <param name="camera_frame_id" value="usb_cam" />
    <param name="io_method" value="mmap"/>
    <remap from="image" to="/camera/image_raw"/>
  </node>

  <node name="image_view_node" pkg="image_view" type="image_view" respawn="fal
se" output="screen" >
    <remap from="image" to="/usb_camera_node/image_raw"/>
    <param name="autosize" value="true" />
  </node>

  <node name="image_pub_sub" pkg="turtlesim_cleaner" type="image_pub_sub" outp
ut="screen" >
        <remap from="image" to="/usb_camera_node/image_raw"/>
  </node>
```

```
  <node name="image_view2_node" pkg="image_view" type="image_view" output="scr
een" >
        <remap from="image" to="/image_converter/output_video"/>
  </node>

</launch>
```

robot_control.cpp:

```
#include "ros/ros.h"
#include "turtlesim/Pose.h"
#include "geometry_msgs/Twist.h"

#include <sstream>

const double PI = 3.14159265359;

class RobotControl
{
private:
    ros::NodeHandle nodeHandle_;
    ros::Publisher velocity_pub;
    ros::Subscriber pose_sub;
    turtlesim::Pose turtlesim_pose;

public:
    RobotControl()
    {
        velocity_pub = nodeHandle_.advertise<geometry_msgs::Twist>("/turtle1/c
md_vel", 1000);
        pose_sub = nodeHandle_.subscribe("/turtle1/pose", 10, &RobotControl::p
oseCallback, this);
    }

    void moveLinear(double speed, double distance, bool direction);
    void roatateAngular (double angular_speed, double angle, bool clockwise);

    double degrees2radians(double angle_in_degrees);
    void setDesiredOrientation (double desired_angle_radians);

    void poseCallback(const turtlesim::Pose::ConstPtr & pose_message);
    void moveToAPose(turtlesim::Pose goalPosition, double distance_tolerance);

    void gridClean();
};

// Gibanje robota prema naprijed sa željenom linearnom brzinom
// za definiranu udaljenost pravocrtno prema naprijed ili nazad
```

```
void RobotControl::moveLinear(double speed, double distance, bool direction){
    geometry_msgs::Twist vel_msg;

    // Određivanje smjera prema naprijed ili prema nazad
    if (direction)
        vel_msg.linear.x =abs(speed);
    else
        vel_msg.linear.x =-abs(speed);
    vel_msg.linear.y =0;
    vel_msg.linear.z =0;

    vel_msg.angular.x = 0;
    vel_msg.angular.y = 0;
    vel_msg.angular.z =0;

    double t0 = ros::Time::now().toSec();
    double current_distance = 0.0;
    ros::Rate loop_rate(100);
    while(current_distance<distance){
        velocity_pub.publish(vel_msg);
        double t1 = ros::Time::now().toSec();
        current_distance = speed * (t1-t0);
        ros::spinOnce();
        loop_rate.sleep();
        //std::cout<<(t1-
t0)<<", "<<current_distance <<", "<<distance<<std::endl;
    }

    //zaustavi robota
    vel_msg.linear.x =0;
    velocity_pub.publish(vel_msg);
}


void RobotControl::roatateAngular(double angular_speed, double relative_angle,
 bool clockwise){

    geometry_msgs::Twist vel_msg;
    // postavljanje linearnih brzina na nulu
    vel_msg.linear.x = 0;
    vel_msg.linear.y = 0;
    vel_msg.linear.z = 0;

    // postavljanje x i y komponenata na nulu
    vel_msg.angular.x = 0;
    vel_msg.angular.y = 0;

    if (clockwise)
        vel_msg.angular.z = -fabs(angular_speed);
```

```
    else
        vel_msg.angular.z = fabs(angular_speed);

    double current_angle = 0.0;
    double t0 = ros::Time::now().toSec();
    ros::Rate loop_rate(100);
    while(current_angle<relative_angle){
        velocity_pub.publish(vel_msg);
        double t1 = ros::Time::now().toSec();
        current_angle = angular_speed * (t1-t0);
        ros::spinOnce();
        loop_rate.sleep();
    };

    //zaustavi robota
    vel_msg.angular.z =0;
    velocity_pub.publish(vel_msg);

}

double RobotControl::degrees2radians(double angle_degrees){
    return (angle_degrees *PI) /180.0;
}


void RobotControl::setDesiredOrientation (double desired_angle_radians){
    double relative_angle_radians = desired_angle_radians - turtlesim_pose.the
ta;
    bool clockwise = ((relative_angle_radians<0)?true:false);
    std::cout << "Zeljeni iznos kuta[rad]: " << desired_angle_radians << std::
endl;
    std::cout << "Kut zakreta robota[rad]: " << turtlesim_pose.theta << std::e
ndl;
    std::cout << "Relativan kut zakreta robota[rad]: " << relative_angle_radia
ns <<std::endl;
    std::cout << "Smjer kazaljke na satu = 1, obrnuti smjer = 0:  " << clockwi
se <<std::endl;
    std::cout << ""  <<std::endl;

    roatateAngular(degrees2radians(10), fabs(relative_angle_radians), clockwis
e);

}

void RobotControl::poseCallback(const turtlesim::Pose::ConstPtr &pose_message)
{
    turtlesim_pose.x=pose_message->x;
    turtlesim_pose.y=pose_message->y;
    turtlesim_pose.theta=pose_message->theta;
```

```
}

double getDistance(double x1, double y1, double x2, double y2){
    return sqrt(pow((x2-x1),2)+pow((y2-y1),2));
}

void RobotControl::moveToAPose(turtlesim::Pose goalPosition, double distance_t
olerance){

    geometry_msgs::Twist vel_msg;

    ros::Rate loop_rate(100);
    double tolerance = 0.01;
    do{
        /****** Proporcionalni regulator******/
        //linearana brzina u smjeru x-osi
        double Kp=1.0;
        double error = getDistance(turtlesim_pose.x, turtlesim_pose.y, goalPos
ition.x, goalPosition.y);
        double tolerance = tolerance+error;
        //Kp = v0 * (exp(-alpha)*error*error)/(error*error);
        vel_msg.linear.x = (Kp*error);
        vel_msg.linear.y =0;
        vel_msg.linear.z =0;

        //kutna brzina u smjeru z-osi
        vel_msg.angular.x = 0;
        vel_msg.angular.y = 0;
        vel_msg.angular.z =4*(atan2(goalPosition.y-
turtlesim_pose.y, goalPosition.x-turtlesim_pose.x)-turtlesim_pose.theta);

        velocity_pub.publish(vel_msg);
        ros::spinOnce();
        loop_rate.sleep();

    }while(getDistance(turtlesim_pose.x, turtlesim_pose.y, goalPosition.x, goa
lPosition.y)>distance_tolerance);
    std::cout << "Stigao na cilj! "<<std::endl;
    vel_msg.linear.x =0;
    vel_msg.angular.z = 0;
    velocity_pub.publish(vel_msg);
}



void RobotControl::gridClean(){

    ros::Rate loop(0.5);
```

```
    // kreiranje varijable pozicije
    turtlesim::Pose pose;
    pose.x=1;
    pose.y=1;
    pose.theta=0;

    // odlazak u zeljenu poziciju
    moveToAPose(pose, 0.01);
    loop.sleep();
    setDesiredOrientation(0);
    loop.sleep();

    // idi linearno naprijed
    moveLinear(2.0, 9.0, true);
    loop.sleep();

    // rotacija za 90 stupnjeva i pomicanje gore
    roatateAngular(degrees2radians(10), degrees2radians(90), false);
    loop.sleep();
    moveLinear(2.0, 9.0, true);

    // rotacija za 90 stupnjeva i pomicanje lijevo
    roatateAngular(degrees2radians(10), degrees2radians(90), false);
    loop.sleep();
    moveLinear(2.0, 1.0, true);

    // rotacija za 90 stupnjeva i pomicanje dolje
    roatateAngular(degrees2radians(10), degrees2radians(90), false);
    loop.sleep();
    moveLinear(2.0, 9.0, true);

    // rotacija za 90 stupnjeva i pomicanje lijevo
    roatateAngular(degrees2radians(30), degrees2radians(90), true);
    loop.sleep();
    moveLinear(2.0, 1.0, true);

    // rotacija za 90 stupnjeva i pomicanje gore
    roatateAngular(degrees2radians(30), degrees2radians(90), true);
    loop.sleep();
    moveLinear(2.0, 9.0, true);

    double distance = getDistance(turtlesim_pose.x, turtlesim_pose.y, x_max, y
_max);

}

int main(int argc, char **argv)
{
    // Inicijalizacija ROS čvora robot_control
```

```
    ros::init(argc, argv, "robot_control");

    //instanciranje robot objekta
    RobotControl robot;

    // inicijalizacija varijabli
    double linearSpeed = 0.0;
    double angularSpeed = 0.0;
    double distance = 0.0 ;
    double angle = 0.0 ;
    bool direction = true;
    bool clockwise = true;

    //Program za testiranje upravljanja robota
    // ROS_INFO("\n\n\n******POCETAK TESTIRANJA***********\n");
    std::cout << "Unesi zeljenu linearnu brzinu: ";
    std::cin >> linearSpeed;
    std::cout << "Unesi zeljenu udaljenost: ";
    std::cin >> distance;
    std::cout << "Naprijed = 1, Nazad = 0: ";
    std::cin >> direction;
    robot.moveLinear(linearSpeed, distance, direction);
    ROS_INFO("Zavrseno linearno gibanje");

    std::cout <<"Unesi zeljenu kutnu brzinu (degree/sec): ";
    std::cin >> angularSpeed;
    std::cout << "Unesi zeljeni kut (degrees): ";
    std::cin >> angle;
    std::cout << "Smjer kazaljke na satu = 1, Suprotan smjer kazaljke na satu
= 0: ";
    std::cin >> clockwise;
    robot.roatateAngular(robot.degrees2radians(angularSpeed), robot.degrees2ra
dians(angle), clockwise);
    ROS_INFO("Gotova rotacija robota");


    ROS_INFO("\n*********Testiranje zakretanja robota za zeljeni kut**********
**\n");
    ros::Rate loop_rate(0.5);
    robot.setDesiredOrientation(robot.degrees2radians(120));
    loop_rate.sleep();
    robot.setDesiredOrientation(robot.degrees2radians(-60));
    loop_rate.sleep();
    robot.setDesiredOrientation(robot.degrees2radians(0));

    ROS_INFO("\n\n\n**********Gibanje robota u zeljenu poziciju u prostoru****
********\n");
    ros::Rate loop_rate(0.5);
    turtlesim::Pose goalPosition;
```

```
    goalPosition.x = 10;
    goalPosition.y = 10;
    goalPosition.theta=0;
    robot.moveToAPose(goalPosition, 0.01);
    loop_rate.sleep();

    // ros::Rate loop(0.5);
    // turtlesim::Pose pose;
    // pose.x=1;
    // pose.y=1;
    // pose.theta=0;
    // moveToAPose(pose, 0.01);

    // pose.x=6;
    // pose.y=6;
    // pose.theta=0;
    // moveToAPose(pose, 0.01);

    ROS_INFO("Gibanje robota po zadanoj mreži");
    robot.gridClean();

    ros::spin();
    return 0;
}
```

test_robot_movement.cpp:

```
#include <ros/ros.h>
#include <geometry_msgs/Twist.h>
#include <geometry_msgs/PoseWithCovariance.h>
#include <geometry_msgs/Pose.h>
#include <nav_msgs/Odometry.h>

#include <signal.h>
#include <termios.h>
#include <stdio.h>
#include <cmath>

#define _USE_MATH_DEFINES

ros::Publisher velocity_publisher;
ros::Subscriber pose_subscriber;

geometry_msgs::Pose pose;

const nav_msgs::Odometry::ConstPtr pose_message;

const double PI = 3.14159265359;
```

```
struct Position {
    double x, y, z;
};

struct Quaternion {
    double w, x, y, z;
};

struct EulerAngles {
    double roll, pitch, yaw;
};

struct LinearSpeed {
    double x, y, z;
};

struct AngularSpeed {
    double x, y, z;
};

double degrees2radians(double angle_in_degrees){
    return (angle_in_degrees *PI) /180.0;
}

double radians2degrees(double angle_in_radians){
    return (angle_in_radians *180.0) /PI;
}

double getDistance(double x1, double y1, double x2, double y2){
    return sqrt(pow((x1-x2),2)+pow((y1-y2),2));
}

void poseCallback(const nav_msgs::Odometry::ConstPtr & pose_message) {
    Position p;
    Quaternion q;
    EulerAngles angle;

    LinearSpeed linear;
    AngularSpeed angular;

    p.x = pose_message->pose.pose.position.x;
    p.y = pose_message->pose.pose.position.y;
    p.z = pose_message->pose.pose.position.z;

    q.x = pose_message->pose.pose.orientation.x;
    q.y = pose_message->pose.pose.orientation.y;
    q.z = pose_message->pose.pose.orientation.z;
    q.w = pose_message->pose.pose.orientation.w;
```

```
    linear.x = pose_message->twist.twist.linear.x;
    linear.y = pose_message->twist.twist.linear.y;
    linear.z = pose_message->twist.twist.linear.z;

    angular.x = pose_message->twist.twist.angular.x;
    angular.y = pose_message->twist.twist.angular.y;
    angular.z = pose_message->twist.twist.angular.z;


    // roll (x-axis rotation)
    double sinr_cosp = 2 * (q.w * q.x + q.y * q.z);
    double cosr_cosp = 1 - 2 * (q.x * q.x + q.y * q.y);
    angle.roll = std::atan2(sinr_cosp, cosr_cosp);

    // pitch (y-axis rotation)
    double sinp = 2 * (q.w * q.y - q.z * q.x);
    if (std::abs(sinp) >= 1)
        angle.pitch = copysign(PI / 2, sinp); // use 90 degrees if out of rang
e
    else
        angle.pitch = std::asin(sinp);

    // yaw (z-axis rotation)
    double siny_cosp = 2 * (q.w * q.z + q.x * q.y);
    double cosy_cosp = 1 - 2 * (q.y * q.y + q.z * q.z);
    angle.yaw = std::atan2(siny_cosp, cosy_cosp);

    // ROS_INFO("ROBOT POSITION");
    // std::cout <<  "Position x is: " << p.x << std::endl;
    // std::cout <<  "Position y is: " << p.y << std::endl;
    // std::cout <<  "Position z is: " << p.z << std::endl;

    // ROS_INFO("ROBOT ORIENTATION");
    // std::cout << "roll (rotation around x-
axis): " << angle.roll << std::endl;
    // std::cout << "pitch (rotation around y-
axis) is: " << angle.pitch << std::endl;
    // std::cout << "yaw (rotation around z-
axis) is: " << angle.yaw << std::endl;
    // std::cout << "yaw (z) in degrees is: " << radians2degrees(angle.yaw) <<
 std::endl;

    // ROS_INFO("ROBOT LINEAR SPEED");
    // std::cout << "Linear speed x:  " << linear.x << std::endl;
    // std::cout << "Linear speed y:  " << linear.y << std::endl;
    // std::cout << "Linear speed z:  " << linear.z << std::endl;

    // ROS_INFO("ROBOT ANGULAR SPEED");
    // std::cout << "Angular speed x:  " << angular.x << std::endl;
```

```
    // std::cout << "Angular speed y:  " << angular.y << std::endl;
    // std::cout << "Angular speed z:  " << angular.z << std::endl;
}

void moveLinear(double speed, double distance, bool isForward) {
    geometry_msgs::Twist vel_msg;
    // Određivanje smjera prema naprijed ili prema nazad
    if (isForward)
        vel_msg.linear.x =fabs(speed);
    else
        vel_msg.linear.x =-fabs(speed);
    vel_msg.linear.y =0;
    vel_msg.linear.z =0;
    vel_msg.angular.x = 0;
    vel_msg.angular.y = 0;
    vel_msg.angular.z =0;

    double t0 = ros::Time::now().toSec();
    double current_distance = 0.0;
    ros::Rate loop_rate(2);
    do{
        velocity_publisher.publish(vel_msg);
        double t1 = ros::Time::now().toSec();
        current_distance = speed * (t1-t0);
        ros::spinOnce();
        loop_rate.sleep();
    }while(current_distance<distance);

    //zaustavi robota
    vel_msg.linear.x =0;
    velocity_publisher.publish(vel_msg);
}

void rotateAngular(double angular_speed, double relative_angle, bool clockwise
){

    geometry_msgs::Twist vel_msg;
    //set a random linear velocity in the x-axis
    vel_msg.linear.x =0;
    vel_msg.linear.y =0;
    vel_msg.linear.z =0;
    //set a random angular velocity in the y-axis
    vel_msg.angular.x = 0;
    vel_msg.angular.y = 0;

    if (clockwise)
        vel_msg.angular.z =-fabs(angular_speed);
    else
        vel_msg.angular.z =fabs(angular_speed);
```

```
    double current_angle = 0.0;
    double t0 = ros::Time::now().toSec();
    ros::Rate loop_rate(100);
    do{
        velocity_publisher.publish(vel_msg);
        double t1 = ros::Time::now().toSec();
        current_angle = angular_speed * (t1-t0);
        ros::spinOnce();
        loop_rate.sleep();
    }while(current_angle<relative_angle);

    vel_msg.angular.z =0;
    velocity_publisher.publish(vel_msg);
}

void setDesiredOrientation (double desired_angle_radians){
    EulerAngles angle;
    // double Kp = 1.05;
    // double Kp = 1.0382;

    double Kp = 1.04;
    double relative_angle_radians = Kp * (desired_angle_radians - angle.yaw);
    bool clockwise = ((relative_angle_radians<0)?true:false);
    std::cout << "Zeljeni iznos kuta[rad]: " << desired_angle_radians << std::
endl;
    std::cout << "Kut zakreta robota[rad]: " << angle.yaw << std::endl;
    std::cout << "Relativan kut zakreta robota[rad]: " << relative_angle_radia
ns <<std::endl;
    std::cout << "Smjer kazaljke na satu = 1, obrnuti smjer = 0:  " << clockwi
se <<std::endl;
    std::cout << ""  <<std::endl;

    rotateAngular(degrees2radians(10), fabs(relative_angle_radians), clockwise
);
}

int main(int argc, char* argv[])
{
    ros::init(argc, argv, "test_robot_movement");
    ros::NodeHandle n;
    velocity_publisher = n.advertise<geometry_msgs::Twist>("/RosAria/cmd_vel",
 10);
    pose_subscriber = n.subscribe("/RosAria/pose", 10, poseCallback);

    // inicijalizacija varijabli
    double linearSpeed = 0.0;
    double angularSpeed = 0.0;
    double distance = 0.0 ;
```

```
    double angle = 0.0 ;
    bool direction = true;
    bool clockwise = true;

    ros::Rate loop_rate(2);

    //Program za testiranje upravljanja robota
    ROS_INFO("\n\n\n******POCETAK TESTIRANJA***********\n");
    std::cout << "Unesi zeljenu linearnu brzinu: ";
    std::cin >> linearSpeed;
    std::cout << "Unesi zeljenu udaljenost: ";
    std::cin >> distance;
    std::cout << "Naprijed = 1, Nazad = 0: ";
    std::cin >> direction;
    moveLinear(linearSpeed, distance, direction);
    ROS_INFO("Zavrseno linearno gibanje");

    std::cout <<"Unesi zeljenu kutnu brzinu (degree/sec): ";
    std::cin >> angularSpeed;
    std::cout << "Unesi zeljeni kut (degrees): ";
    std::cin >> angle;
    std::cout << "Smjer kazaljke na satu = 1, Suprotan smjer kazaljke na satu
= 0: ";
    std::cin >> clockwise;
    rotateAngular(degrees2radians(angularSpeed), degrees2radians(angle), clock
wise);
    ROS_INFO("Gotova rotacija robota");

    ROS_INFO("\n*********Testiranje zakretanja robota za zeljeni kut**********
**\n");
    setDesiredOrientation(degrees2radians(120));
    loop_rate.sleep();
    setDesiredOrientation(degrees2radians(60));
    loop_rate.sleep();
    setDesiredOrientation(degrees2radians(0));
    loop_rate.sleep();

    ros::spin();
    return 0;
}
```

teleop_twist_key.cpp:

```
#include <ros/ros.h>
#include <geometry_msgs/Twist.h>
#include <signal.h>
#include <termios.h>
#include <stdio.h>
```

```
#define KEYCODE_R 0x43
#define KEYCODE_L 0x44
#define KEYCODE_U 0x41
#define KEYCODE_D 0x42
#define KEYCODE_A 0x61
#define KEYCODE_Z 0x7A
#define KEYCODE_S 0x73
#define KEYCODE_X 0x78
#define KEYCODE_Q 0x71
#define KEYCODE_SPACE 0x20

class TeleopRosAria
{
  public:
    TeleopRosAria();
    void keyLoop();
  private:
    ros::NodeHandle nh_;
    double linear_, angular_, l_scale_, a_scale_;
    double current_linear_, current_angular_, step_linear_, step_angular_;
    ros::Publisher twist_pub_;
};
TeleopRosAria::TeleopRosAria():
  linear_(0),
  angular_(0),
  l_scale_(2.0),
  a_scale_(2.0),
  current_angular_(0.1),
  current_linear_(0.1),
  step_linear_(0.2),
  step_angular_(0.1)
{
  nh_.param("scale_angular", a_scale_, a_scale_);
  nh_.param("scale_linear", l_scale_, l_scale_);
  twist_pub_ = nh_.advertise<geometry_msgs::Twist>("RosAria/cmd_vel", 1);
}
int kfd = 0;
struct termios cooked, raw;
void quit(int sig)
{
  tcsetattr(kfd, TCSANOW, &cooked);
  ros::shutdown();
  exit(0);
}
int main(int argc, char** argv)
{
  ros::init(argc, argv, "teleop_twist_key");
  TeleopRosAria teleop_RosAria;
  ROS_INFO("teleop is initialized");
```

```
  signal(SIGINT,quit);
  teleop_RosAria.keyLoop();
  return(0);
}
void TeleopRosAria::keyLoop()
{
  char c;
  bool dirty=false;
  // get the console in raw mode
  tcgetattr(kfd, &cooked);
  memcpy(&raw, &cooked, sizeof(struct termios));
  raw.c_lflag &=~ (ICANON | ECHO);
  // Setting a new line, then end of file
  raw.c_cc[VEOL] = 1;
  raw.c_cc[VEOF] = 2;
  tcsetattr(kfd, TCSANOW, &raw);
  puts("\n*****RUCNO UDALJENO UPRAVLJANJE*****\n");
  puts("Citanje podataka s tipkovnice");
  puts("---------------------------");
  puts("Robotom se upravlja preko strelica na tipkovnici");
  puts("       ^    ");
  puts("     <   > ");
  puts("       v          ");
  puts("Pritisni tipku SPACE za zaustavljanje robota");
  puts("Pritisni tipku q za izlazak iz programa");
  puts("a/z - Povecaj/smanji translacijsku brzinu");
  puts("s/x - Povecaj/smanji rotacijsku brzinu");
  for(;;)
  {
    // get the next event from the keyboard
    if(read(kfd, &c, 1) < 0)
    {
      perror("read():");
      exit(-1);
    }
    linear_=angular_=0;
    char printable[100];
    ROS_DEBUG("value: 0x%02X\n", c);
    switch(c)
    {
      case KEYCODE_L:
        ROS_DEBUG("LEFT");
        angular_ = current_angular_;
        linear_ = 0;
        dirty = true;
        break;
      case KEYCODE_R:
        ROS_DEBUG("RIGHT");
        angular_ = -current_angular_;
```

```
        linear_ = 0;
        dirty = true;
        break;
      case KEYCODE_U:
        ROS_DEBUG("UP");
        linear_ = current_linear_;
        angular_ = 0;
        dirty = true;
        break;
      case KEYCODE_D:
        ROS_DEBUG("DOWN");
        linear_ = -current_linear_;
        angular_ = 0;
        dirty = true;
        break;
  case KEYCODE_A:
    ROS_DEBUG("INCREASE LINEAR SPEED");
    current_linear_ += step_linear_;
    sprintf(printable, "Translacijska brzina: %02f", current_linear_);
    puts(printable);
    dirty=true;
    break;
  case KEYCODE_Z:
    ROS_DEBUG("DECREASE LINEAR SPEED");
          current_linear_ -= step_linear_;
    if(current_linear_ < 0)
        current_linear_ = 0;
          sprintf(printable, "Translacijska brzina: %02f", current_linear_);
          puts(printable);
          dirty=true;
          break;
  case KEYCODE_S:
    ROS_DEBUG("INCREASE ANGULAR SPEED");
          current_angular_ += step_angular_;
          sprintf(printable, "Rotacijska brzina: %02f", current_angular_);
          puts(printable);
          dirty=true;
          break;
  case KEYCODE_X:
    ROS_DEBUG("DECREASE LINEAR SPEED");
          current_angular_ -= step_angular_;
    if(current_angular_ < 0)
        current_angular_ = 0;
          sprintf(printable, "Rotacijska brzina: %02f", current_angular_);
          puts(printable);
          dirty=true;
          break;
      case KEYCODE_SPACE:
        ROS_DEBUG("STOP");
```

```
        linear_ = 0;
        angular_ = 0;
        dirty = true;
        break;
      case KEYCODE_Q:
        ROS_DEBUG("QUIT");
        ROS_INFO_STREAM("ILAZAK IZ RUCNOG UDALJENOG UPRAVLJANJA");
        return;
        break;
    }
    geometry_msgs::Twist twist;
    twist.angular.z = a_scale_*angular_;
    twist.linear.x = l_scale_*linear_;
    if(dirty ==true)
    {
      twist_pub_.publish(twist);
      dirty=false;
    }
  }
  return;
}
```

tennis_ball_tracker.cpp:

```
#include "ros/ros.h"
#include "std_msgs/String.h"
#include "std_msgs/Float64.h"
#include <mobile_robot/ballCoordinates.h>


#include <iostream>
#include <image_transport/image_transport.h>
#include <cv_bridge/cv_bridge.h>
#include <geometry_msgs/Twist.h>
#include <sensor_msgs/image_encodings.h>
#include <opencv2/imgproc/imgproc.hpp>
#include <opencv2/highgui/highgui.hpp>
#include <math.h>
#include <nav_msgs/Odometry.h>

#define RGB_FOCAL_LEN_MM 138.90625  // fokalna duljina kamere u mm
#define BALL_DIAM_MM 900.0          // promjer lopte u mm
#define CAMERA_HEIGHT_MM 260.0      // udaljenost kamere od podloge u mm
#define IMG_HEIGHT_PX 480.0         // u pikselima
#define IMG_WIDTH_PX 640.0          // u pikselima
#define _USE_MATH_DEFINES

static const std::string OPENCV_WINDOW = "Image window";
static int x, y, radius;
const double PI = 3.14159265359;
```

```
class BallDetector {
private:
    ros::NodeHandle nodeHandle_;
    image_transport::ImageTransport imageTransport_;
    image_transport::Subscriber imageSub_;
    image_transport::Publisher imagePub_;
    ros::Publisher coordPub_;
    mobile_robot::ballCoordinates coordinates;
    double t0_ ;

public:
    BallDetector() : imageTransport_(nodeHandle_)
    {
        //Pretplati se na ulazni video i objavi sliku na izlaz
        imageSub_ = imageTransport_.subscribe(
            "/usb_cam/image_raw", 10, &BallDetector::imageCallback, this);
        imagePub_ = imageTransport_.advertise("/image_converter/output_video",
 10);
        coordPub_ = nodeHandle_.advertise<mobile_robot::ballCoordinates>("poin
t_tennis_ball", 1000);

        t0_ = ros::Time::now().toSec();
        cv::namedWindow(OPENCV_WINDOW, 0);
    }

    ~BallDetector()
    {
        cv::destroyWindow(OPENCV_WINDOW);
    }

    void imageCallback(const sensor_msgs::ImageConstPtr &msg) {
        cv_bridge::CvImagePtr cvPtr;
        cv_bridge::CvImagePtr cvGrayPtr;
        cv::Mat rgbIMG, hsvIMG, thresholdedIMG, lowerBoundColor, upperBoundCol
or, reverseMask;

        lowerBoundColor =cv::Scalar(30, 100, 50);
        upperBoundColor = cv::Scalar(60, 255, 255);

        //orange ball tracking
        int iLowH = 0;
        int iHighH = 60;
        int iLowS = 100;
        int iHighS = 255;
        int iLowV = 54;
        int iHighV = 255;
```

```
        std::vector<cv::Vec3f> v3fCircles;      // 3 elementa vektora, ovdje ć
e se spremati koordinate x i y, te radijus loptice
                                                // prosljeđuje se u funkciju H
oughCircles()
        try
        {
            cvPtr = cv_bridge::toCvCopy(msg, sensor_msgs::image_encodings::BGR
8);
            rgbIMG = cvPtr->image;
            // cv::imshow("RGB window", rgbIMG);

            // Pretvaranje RGB slike u HSV područje
            cv::cvtColor(rgbIMG, hsvIMG,  cv::COLOR_BGR2HSV);
            // cv::imshow("HSV window", hsvIMG);

            // Primjena praga kako bi se definirao raspon boje loptice koju ze
limo prepoznati
            cv::inRange(hsvIMG, cv::Scalar(iLowH, iLowS, iLowV), cv::Scalar(iH
ighH, iHighS, iHighV), thresholdedIMG);

            // Uklanjanje malih objekta na slici kako bi eliminirali šum
            cv::erode(thresholdedIMG, thresholdedIMG, cv::getStructuringElemen
t(cv::MORPH_ELLIPSE, cv::Size(5, 5)) );
            cv::dilate( thresholdedIMG, thresholdedIMG, cv::getStructuringElem
ent(cv::MORPH_ELLIPSE, cv::Size(5, 5)) );

            // Popunjavanje preostalih malih objekata radi uklanjanja šuma
            cv::dilate( thresholdedIMG, thresholdedIMG, cv::getStructuringElem
ent(cv::MORPH_ELLIPSE, cv::Size(5, 5)) );
            cv::erode(thresholdedIMG, thresholdedIMG, cv::getStructuringElemen
t(cv::MORPH_ELLIPSE, cv::Size(5, 5)) );

            // efekt zamagljivanja svega osim loptice
            cv::GaussianBlur(thresholdedIMG, thresholdedIMG, cv::Size(9, 9), 2
, 2);
            cv::imshow("Thresholded img", thresholdedIMG);

            cv::HoughCircles(thresholdedIMG, v3fCircles, CV_HOUGH_GRADIENT, 1,
 hsvIMG.rows/8, 100, 20, 0, 0);

            // reverse mask makes everything that is black white, and vice ver
sa
            // reverseMask = 255 - thresholdedIMG;
            // cv::imshow("Thresholded Image", reverseMask); //show the thresh
olded image
            cv::imshow("Mask image", thresholdedIMG);
        }
        catch(const std::exception& e)
        {
```

```
            ROS_ERROR("cv_bridge exception: %s", e.what());
            return;
        }

        for (size_t i = 0; i < v3fCircles.size(); i++) {

            // pozicija loptice na slici u obliku x i y koordinate te radius l
optice
            x = static_cast<int>(round(v3fCircles[i][0]));
            y = static_cast<int>(round(v3fCircles[i][1]));
            radius = static_cast<int>(round(v3fCircles[i][2]));
            cv::Point center(x, y);


            double area = 0.0;
            area = (pow(radius,2) * PI);
            // std::cout << "area of ball is: " << area << std::endl;

            if (area > 2000)
            {
                coordinates.x = x;
                coordinates.y = y;
                coordPub_.publish(coordinates);

                // cv::circle(cvPtr-
>image, cv::Point(50,50), 10, CV_RGB(255, 0, 0));
                // cv::line(rgbIMG, center, center, black, 2);

                std::cout << "Ball position X = "<< x                    // x p
ozicija od središta kruga
                          <<",\tY = "<< y                                 // y p
ozicija od središta kruga
                          <<",\tRadius = "<< radius<< "\n";               // rad
ijus kruga

                cv::circle(rgbIMG,
      // nacrtaj na originalnoj slici
                    cv::Point((int)v3fCircles[i][0], (int)v3fCircles[i][1]),
      // središte kruga
                    3,
      // radijus kruga u pikselima
                    cv::Scalar(0, 255, 0),
      // u zelenoj boji
                    CV_FILLED);
      // debljina linije


      // nacrtaj krug oko detektiranih objekata
```

```
                cv::circle(rgbIMG,
        // nacrtaj na originalnoj slici
                    cv::Point((int)v3fCircles[i][0], (int)v3fCircles[i][1]),
        // središte kruga
                    (int)v3fCircles[i][2],
        // radijus kruga u pikselima
                    cv::Scalar(0, 0, 255),
        // u crvenoj boji
                    3);
                // cv::imshow("RGB window", rgbIMG);

                //Podjela slike na lijevi i desni te centralni dio
                cv::Point pt1, pt2,pt3,pt4,pt5,pt6;

                // Centralno područje
                pt1.x = IMG_WIDTH_PX / 2;
                pt1.y = 0;
                pt2.x = IMG_WIDTH_PX / 2;
                pt2.y = 480;

                // Lijevo područje
                pt3.x = (IMG_WIDTH_PX / 2) - CAMERA_HEIGHT_MM;
                pt3.y = 0;
                pt4.x = (IMG_WIDTH_PX / 2) - CAMERA_HEIGHT_MM;
                pt4.y = 480;

                // Desno područje
                pt5.x = (IMG_WIDTH_PX / 2) + CAMERA_HEIGHT_MM;
                pt5.y = 0;
                pt6.x = (IMG_WIDTH_PX / 2) + CAMERA_HEIGHT_MM;
                pt6.y = 480;

                line(rgbIMG,  pt1,  pt2, cv::Scalar(0, 0, 255),0.2);
                line(rgbIMG,  pt3,  pt4, cv::Scalar(0, 255, 0),0.2);
                line(rgbIMG,  pt5,  pt6, cv::Scalar(0, 255, 0),0.2);

                cv::putText(rgbIMG, "Left", cvPoint(50,240), cv::FONT_HERSHEY_
SIMPLEX, 1, cvScalar(255,0,0), 2, CV_AA);
                cv::putText(rgbIMG, "Center", cvPoint(280,240), cv::FONT_HERSH
EY_SIMPLEX, 1, cvScalar(0,0,255), 2, CV_AA);
                cv::putText(rgbIMG, "Right", cvPoint(480,240), cv::FONT_HERSHE
Y_SIMPLEX, 1, cvScalar(255,0,0), 2, CV_AA);
                // cv::imshow("RGB with line", rgbIMG);
            }

        }

        cv::Mat resized;
        cv::resize(cvPtr->image, resized, cv::Size(640, 480));
```

```
        cv::imshow("OPENCV_WINDOW",resized);

        // Osvježi video na izlazu s napravljenom obradom slike
        cv::imshow(OPENCV_WINDOW, cvPtr->image);
        if (cv::waitKey(10) == 27)
        {
            std::cout << "Esc key is pressed by user. Stoppig the video" << st
d::endl;
            cv::destroyAllWindows();
        }

        imagePub_.publish(cvPtr->toImageMsg());
    }
};

int main(int argc, char* argv[])
{
    ros::init(argc, argv, "tennis_ball_tracker");
    ROS_INFO("Node is initialized");
    BallDetector ball;
    ros::spin();
    return 0;
}
```

mobile_robot_chase.cpp:

```
#include "ros/ros.h"
#include "std_msgs/String.h"
#include "std_msgs/Float64.h"
#include <mobile_robot/ballCoordinates.h>

#include <ros/ros.h>
#include <iostream>
#include <image_transport/image_transport.h>
#include <cv_bridge/cv_bridge.h>
#include <geometry_msgs/Twist.h>
#include <sensor_msgs/image_encodings.h>
#include <opencv2/imgproc/imgproc.hpp>
#include <opencv2/highgui/highgui.hpp>
#include <math.h>
#include <nav_msgs/Odometry.h>

mobile_robot::ballCoordinates ballCoordinates;
ros::Publisher velocityPub_;

static int x, y, radius;

class RobotChase
{
private:
```

```
    ros::NodeHandle nodeHandle_;
    ros::Subscriber positionSub_;
    double linear_, angular_;
    double time_detected_;
public:
    RobotChase(): linear_(0.0), angular_(0.0)
    {
        positionSub_ = nodeHandle_.subscribe("/point_tennis_ball", 10, &RobotC
hase::CoordinatesCallback, this);
        x = ballCoordinates.x;
        y = ballCoordinates.y;
        time_detected_ = 0.0;
    }

    void CoordinatesCallback (const mobile_robot::ballCoordinates::ConstPtr &m
sg);

    double BallDetected();

    void MoveForward();
    void MoveLeft();
    void MoveRight();
};

void RobotChase::MoveForward()
{
    geometry_msgs::Twist vel_msg;
    ros::Rate rate(2);

    // gibanje robota naprijed
    vel_msg.linear.x = 0.2;
    vel_msg.linear.y = 0;
    vel_msg.linear.z = 0;
    vel_msg.angular.x = 0;
    vel_msg.angular.y = 0;
    vel_msg.angular.z = 0;
    velocityPub_.publish(vel_msg);
}

void RobotChase::MoveLeft()
{
    geometry_msgs::Twist vel_msg;
    ros::Rate rate(2);

    // gibanje robota u lijevu stranu
    vel_msg.linear.x = 0;
    vel_msg.linear.y = 0;
    vel_msg.linear.z = 0;
    vel_msg.angular.x = 0;
```

```
    vel_msg.angular.y = 0;
    vel_msg.angular.z = 0.2;
    velocityPub_.publish(vel_msg);
}

void RobotChase::MoveRight()
{
    geometry_msgs::Twist vel_msg;
    ros::Rate rate(2);

    // gibanje robota u desnu stranu
    vel_msg.linear.x = 0;
    vel_msg.linear.y = 0;
    vel_msg.linear.z = 0;
    vel_msg.angular.x = 0;
    vel_msg.angular.y = 0;
    vel_msg.angular.z = -0.2;
    velocityPub_.publish(vel_msg);
}

double RobotChase::BallDetected()
{
    return (ros::Time::now().toSec() - time_detected_ < 1.0);
}

void RobotChase::CoordinatesCallback (const mobile_robot::ballCoordinates::Con
stPtr &msg)
{
    // ROS_INFO("Started getting x and y coordinates");
    // ballCoordinates.x = msg->x;
    // ballCoordinates.y = msg->y;
    ballCoordinates.x = msg->x;
    ballCoordinates.y = msg->y;
    x = ballCoordinates.x;
    y = ballCoordinates.y;
    time_detected_ = ros::Time::now().toSec();
    // std::cout << "Time the ball is detected: " << time_detected_ << std::en
dl;

    if(!BallDetected()) {
    //   std::cout<<"Object not found"<< Mat(p) << endl;
        std::cout << "Object not found" << std::endl;
    }

    if(x > 20 && x < 220 ) {
        std::cout<<"LEFT TURN"<< " " << std::endl;
        MoveLeft();
    }
```

```
    else if(x > 220 && x < 420) {
        std::cout<<"FORWARD"<< " " << std::endl;
        MoveForward();
    }

    else if(x > 420 && x < 600) {
        std::cout<<"RIGHT TURN"<< " " << std::endl;
        MoveRight();
    }
}


int main(int argc, char **argv)
{
    ros::init(argc, argv, "mobile_robot_chase");
    ROS_INFO("Node is initialized");
    ros::NodeHandle n;
    velocityPub_ = n.advertise<geometry_msgs::Twist>("/RosAria/cmd_vel", 10);
    ros::Rate loop_rate(2);
    RobotChase robot;
    ros::spin();
    return 0;
}
```

ros_ball_tracker.launch:

```
<launch>
  <node name="usb_camera_node" pkg="usb_cam" type="usb_cam_node" output="scree
n" >
    <param name="video_device" value="/dev/video0" />
    <!-- <param name="image_width" value="1280" />
    <param name="image_height" value="720" /> -->
    <param name="image_width" value="640" />
    <param name="image_height" value="480" />
    <!-- <param name="image_width" value="320" />
    <param name="image_height" value="240" /> -->
    <param name="pixel_format" value="yuyv" />
    <param name="camera_frame_id" value="usb_cam" />
    <param name="io_method" value="mmap"/>
    <remap from="/usb_cam/image_raw" to="/camera/image_raw"/>
  </node>

  <node name="image_view_node" pkg="image_view" type="image_view" respawn="fal
se" output="screen" >
    <remap from="image" to="/usb_camera_node/image_raw"/>
    <param name="autosize" value="true" />
  </node>

  <node name="tennis_ball_tracker" pkg="mobile_robot" type="tennis_ball_tracke
r" output="screen" >
```

```
        <remap from="/usb_cam/image_raw" to="/usb_camera_node/image_raw"/>
  </node>

  <!-
- <node name="image_view2_node" pkg="image_view" type="image_view" output="scr
een" >
        <remap from="image" to="/image_converter/output_video"/>
  </node> -->

</launch>
```

video_parameter.cpp:

```
#include <iostream>
#include "opencv2/highgui/highgui.hpp"
#include "opencv2/imgproc/imgproc.hpp"

using namespace cv;
using namespace std;

int main( int argc, char** argv )
{
    VideoCapture cap(0); //capture the video from USB camera

    if ( !cap.isOpened() )  // if not success, exit program
    {
         cout << "Cannot open the web cam" << endl;
         return -1;
    }

    namedWindow("Control", CV_WINDOW_AUTOSIZE); //create a window called "Cont
rol"

    int iLowH = 30;
    int iHighH = 60;

    int iLowS = 100;
    int iHighS = 255;

    int iLowV = 50;
    int iHighV = 255;

    //Create trackbars in "Control" window
    cvCreateTrackbar("LowH", "Control", &iLowH, 179); //Hue (0 - 179)
    cvCreateTrackbar("HighH", "Control", &iHighH, 179);
    cvCreateTrackbar("LowS", "Control", &iLowS, 255); //Saturation (0 - 255)
    cvCreateTrackbar("HighS", "Control", &iHighS, 255);
    cvCreateTrackbar("LowV", "Control", &iLowV, 255); //Value (0 - 255)
    cvCreateTrackbar("HighV", "Control", &iHighV, 255);
```

```
    while (true)
    {
        Mat imgOriginal;
        bool bSuccess = cap.read(imgOriginal); // read a new frame from video

         if (!bSuccess) //if not success, break loop
        {
            cout << "Cannot read a frame from video stream" << endl;
            break;
        }

        Mat imgHSV;

        cvtColor(imgOriginal, imgHSV, COLOR_BGR2HSV); //Convert the captured f
rame from BGR to HSV

        Mat imgThresholded;
        inRange(imgHSV, Scalar(iLowH, iLowS, iLowV), Scalar(iHighH, iHighS, iH
ighV), imgThresholded); //Threshold the image

        //morphological opening (remove small objects from the foreground)
        // erode(imgThresholded, imgThresholded, getStructuringElement(MORPH_E
LLIPSE, Size(5, 5)) );
        // dilate( imgThresholded, imgThresholded, getStructuringElement(MORPH
_ELLIPSE, Size(5, 5)) );

        // // //morphological closing (fill small holes in the foreground)
        // dilate( imgThresholded, imgThresholded, getStructuringElement(MORPH
_ELLIPSE, Size(5, 5)) );
        // erode(imgThresholded, imgThresholded, getStructuringElement(MORPH_E
LLIPSE, Size(5, 5)) );

        cv::GaussianBlur(imgThresholded, imgThresholded, cv::Size(9, 9), 2, 2)
;   //Blur Effect
        imshow("Thresholded Image", imgThresholded); //show the thresholded im
age
        imshow("Original", imgOriginal); //show the original image

        if (waitKey(30) == 27) //wait for 'esc' key press for 30ms. If 'esc' k
ey is pressed, break loop
        {
            cout << "esc key is pressed by user" << endl;
            break;
        }
    }
}
```