Hvatanje predmeta u slobodnom letu robotom

Kokić, Mia

Master's thesis / Diplomski rad

2016

Degree Grantor / Ustanova koja je dodijelila akademski / stručni stupanj: **University of Zagreb, Faculty of Mechanical Engineering and Naval Architecture / Sveučilište u Zagrebu, Fakultet strojarstva i brodogradnje**

Permanent link / Trajna poveznica: https://urn.nsk.hr/urn:nbn:hr:235:508395

Rights / Prava: In copyright / Zaštićeno autorskim pravom.

Download date / Datum preuzimanja: **2024-07-16**

Repository / Repozitorij:

Repository of Faculty of Mechanical Engineering and Naval Architecture University of Zagreb

SVEUČILIŠTE U ZAGREBU

FAKULTET STROJARSTVA I BRODOGRADNJE

DIPLOMSKI RAD

Mentor:

Prof. dr. sc. Bojan Jerbić

Ime i prezime:

Mia Kokić

Zagreb, 2016.

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

Zahvaljujem se mentoru prof. dr. sc. Bojanu Jerbiću na odabiru teme. Također se zahvaljujem asistentu mag. ing. mech. Filipu Šuligoju na pruženoj stručnoj pomoći i korisnim savjetima tijekom izrade rada.

Mia Kokić

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

Sveučilište u Zagrebu Fakultet strojarstva i brodogradnje	
Datum	Prilog
Klasa:	
Ur.broj:	

DIPLOMSKI ZADATAK

Student: Mia Kokić Mat. br.: 0035182651

Naslov rada na hrvatskom jeziku: **HVATANJE PREDMETA U SLOBODNOM LETU ROBOTOM**

Naslov rada na engleskom jeziku: **CATCHING OBJECT IN FREE FLY BY ROBOT**

Opis zadatka:

Problem hvatanja predmeta u slobodnom letu robotom podrazumijeva razvoj robotskog upravljačkog sustava za hvatanje loptice koja se baca u njegov radni prostor. Problem hvatanja loptice pomoću robota primjenjuje se za razvoj i ispitivanje metoda robotskog upravljanja u dinamičkoj okolini i vizijskih algoritama za prepoznavanje u realnom vremenu. Sustav za hvatanje loptice može se podijeliti u tri podproblema: prepoznavanje loptice, estimacija trajektorije i upravljanje robotom.

Sustav mora temeljem vizijskog prepoznavanja loptice u letu i njezinih prostornih koordinata izračunati predviđenu trajektoriju i koordinate presretanja robotom. Kontinuiranim praćenjem leta loptice vizijskim sustavom osigurati korekciju trajektorije i predviđene pozicije prihvaćanja. Temeljem dobivenih podataka robot mora pozicionirati hvataljku u položaj prihvaćanja u najkraćem mogućem vremenu, odnosno do trenutka dolaska loptice u položaj presretanja.

Zadatak zadan:	Rok predaje rada:	Predviđeni datum obrane:
14. siječnja 2016.	17. ožujka 2016.	23., 24. i 25. ožujka 2016.

Zadatak zadao: Predsjednik Povjerenstva:

Prof. dr. sc. Bojan Jerbić

Prof. dr. sc. Franjo Cajner

SADRŽAJ

POPIS SLIKA

POPIS TABLICA

POPIS OZNAKA

Oznaka	Jedinica	Opis oznake
$\mathbf{R}$		Matrica rotacije
$\mathbf{{}^{k}_{B}T}$		Matrica transformacije iz Kinect-ovog u imaginarni k.s.
$\mathbf{{}^{k}_{B}T}$		Matrica transformacije iz Kinect-ovog u imaginarni k.s.
α	rad	Elevacijski kut

SAŽETAK

Hvatanje objekta u pokretu iznimno je težak zadatak kako za čovjeka tako i za robota. Kako bi izvršio isti robot mora imati sposobnost opažanja, praćenja objekta, predviđanja gibanja, planiranja trajektorije i dobru senzimotornu koordinaciju. Zadatak ovog rada bio je razviti program za upravljanje robotom koji može uhvatiti lopticu u letu. Glavne elemente sustava čine loptica, vizijski sustav, računalo i UR5. U sklopu zadatka razvijen je algoritam za prepoznavanje loptice u letu, predviđanje točke hvatanja i upravljači program koji šalje podatke robotu u koju točku se pomaknuti kako bi hvatanje bilo uspješno. Program je napisan u programskom jeziku C#, a kompletan kod priložen je u prilogu A.

Ključne riječi: Robotika, Prepoznavanje, UR5, Microsoft Kinect, Računalni vid, Hvatanje loptice

SUMMARY

Catching a moving object with a hand is one of the most difficult tasks for humans as well as for robot systems. Smart sensing, object tracking, motion prediction, on-line trajectory planning and motion coordination are capabilities required in a robotic system to catch a thrown ball. The application for catching a flying ball was developed and the main elements of the system are the ball, vision system and UR5. The algorithm for detection and predictions were developed as well as the program for controlling the robot. Program is written in C# programming language and the complete code is in attachment A.

Key words: Robotics, Detection, Prediction, UR5, Microsoft Kinect, Machine vision, Ball catching

1. UVOD

Hvatanje objekta u pokretu iznimno je težak zadatak kako za čovjeka tako i za robota. Kako bi izvršio isti robot mora imati sposobnost opažanja, praćenja objekta, predviđanja gibanja, planiranja trajektorije i odgovarajuću senzimotornu koordinaciju. Roboti mogu biti jednostavno programirani za različite zadatke, ali kada je riječ o ponašanju u nepredvidivim situacijima ili dinamičkoj okolini javljaju se mnogi problemi. Takav primjer je i hvatanje objekta u letu jer robot mora u vrlo kratkom vremenu integrirati nekoliko različitih informacija dobivenih sa svojih senzora i sukladno njima programirati položaj hvatanja.

Sudjelovanje ili obavljanje sportske aktivnosti predstavlja odlično mjerilo za uspješnost percepcije, planiranja i kontrole u stvarnom vremenu, a takvi roboti se stoga mogu koristiti na satelitima za sakupljanje svemirskog otpada, u automobilskoj industriji, kućanstvu ili jednostavno za potrebe igranja. Robot koji u vrlo kratkom vremenu može reagirati na promjene u okolini i nepredvidive situacije mora imati mogućnost prepoznavanja i zaključivanja neovsino o vanjskim uvjetima ili tehničkim karakteristikama robota. Uspješna percepcija znači da robot s vrlo velikom točnošću može prepoznati objekte i njihova gibanja u svojoj okolini, a uspješna kontrola se odnosi na brzinu i točnost reakcije na promjene u okolini uzevši obzir faktore kao što su tehničke mogućnosti robota, položaj drugih objekata i sigurnost čovjeka.

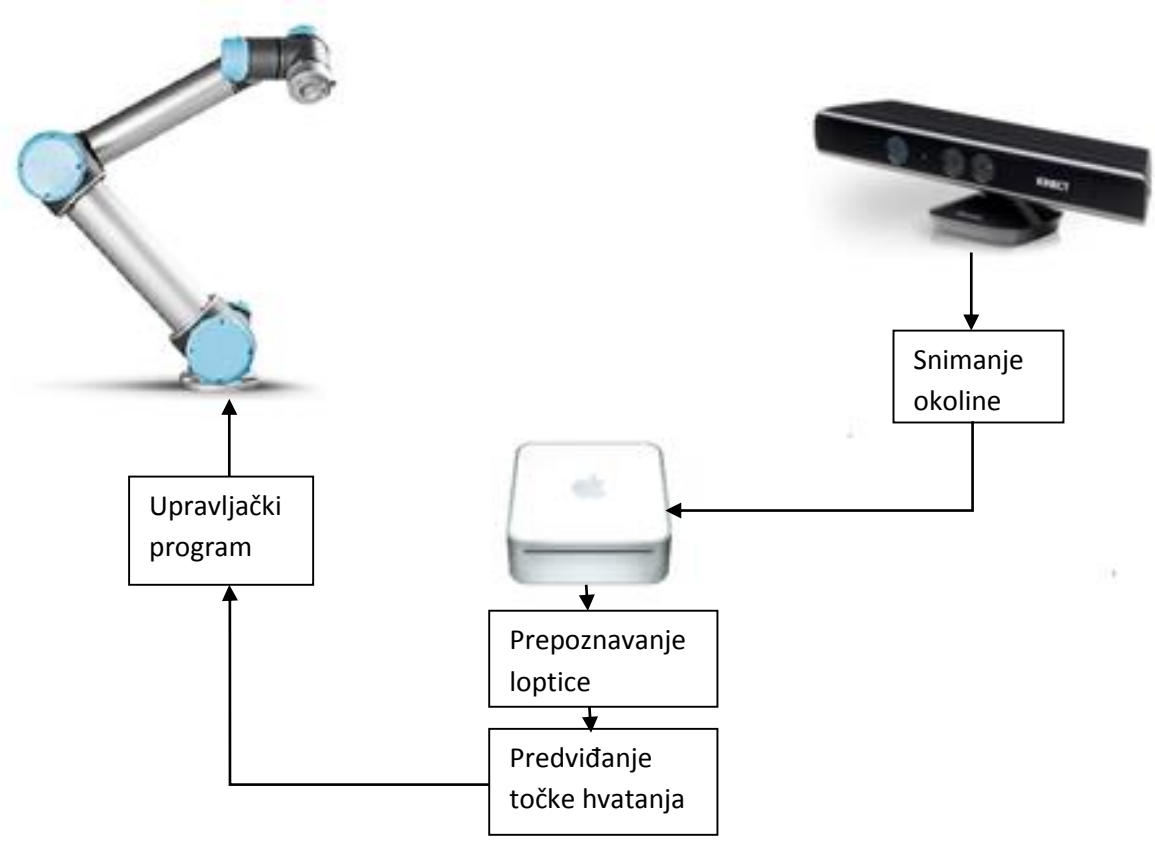
Zadatak je bio razviti aplikaciju za upravljački program za hvatanje loptice u letu gdje glavne elemente tehničkog sustava čine loptica, vizijski sustav, računalo i jedna robotska ruka. Koristeći se Microsoft Kinect-om moguće je vizualno pratiti leteći objekt koji se kreće prema robotu i poslati informaciju robotu da se pomakne u točku u kojoj će moći uhvatiti lopticu. Algoritam za prepoznavanje loptice pokušava naći položaj središta, polumjer i udaljenost loptice od uređaja u svakoj sličici (brzina uređaja je tridesdet sličica u sekundi). Dobiveni skup točaka u prostoru može se opisati krivuljom drugog reda koja zatim služi za predviđanje točke hvatanja. U ovom radu korišteni su već postojeći i razvijeni novi algoritmi za prepoznavanje po boji i obliku, kalibraciju sustava te predviđanje točke hvatanja, a upotreba Microsoft Kinect-a kao senzora značajno smanjuje cijenu opreme. Kao i u ovom radu, u većini drugih upotrebljavaju se statične kamere, a prepoznvanje loptice se temelji na segemtaciji pomoću boje.

Različiti pristupi vezani uz hvatanje objekata u letu i procjenu trajektorije mogu se podijeliti u nekoliko kategorija: upotreba stereovizijskog sustava za potrebe hvatanja, upotreba jedne kamere za hvatanje u ravnini, upotreba jedne kamere za procjenu trajektorije i hvatanje u prostoru, procjenitelji koji se bave Chapmanovom hipotezom, sustavi koji uzimaju u obzir izmjenu sile između hvataljke i objekta, upotreba neuronskih mreža i drugi. Vizijski sustavi koji upotrebljavaju dvije ili više kamera koriste se metodom trijangulacije za rekonstrukciju pozicije loptice u prostoru [1-3]. Ipak, ova metoda zahtjeva točnu kalibraciju i sofisticiranu izradu hardvera. Stereovizijski sustav prošireni Kalmanov filtar i algoritam za predviđanje upotrebljeni su u [4]. Bez posebnog hardvera i koristeći već razvijene algoritme, autori koriste stereovizijski sustav za praćenje objekta i hvatanje mrežicom postavljenom na vrh alata, a segmentacija se temelji na razlici stvarne slike i referentne slike. Osim spomenute metode segmentacije, većina autora u svojim radovima za potrebe prepoznavanja upotrebljava Houghovu transformaciju [5-7], a neki se bave pronalaženjem oblika nakon prebacivanja u HSL ili HSV prostor boja [8,9] kako bi se smanjio utjecaj vanjskih uvjeta. Većina tih algoritama kasnije zahjeva upotrebu nekog filtra za uklanjanje šuma kao što je slučaj i u ovome radu. Sustavi koji koriste jednu kameru jednostavniji su za kalibraciju, ali je teže rekonstruirati pozicije loptice u prostoru. Pozicija u prostoru i brzina izbačenog projektila su procjenjeni u [5] analizom sekvence slika dobivenih sa jedne kamere. Većina ovih sustava pretpostavlja samo jednu lopticu u letu i česti su slučajevi krivih prepoznavanja. U takvim sustavima procjena pozicije loptice temelji se na upotrebi proširenog Kalmanovog filtra [6,7] ili polinomne regresije [8-10] kao što je slučaj ovom radu.

Glavni doprinos ovoga rada je razvijanje algoritma za prepoznavanje objekta u pokretu te algoritma za predviđanje na temelju podataka iz prošlosti. Uspješno riješen zadatak potvrđuje da se kompleksan problem kao što je hvatanje objekta u letu može riješiti na jednostavan način koristeći jeftinu opremu laku za korištenje i programiranje.

2. TEHNIČKI POSTAV

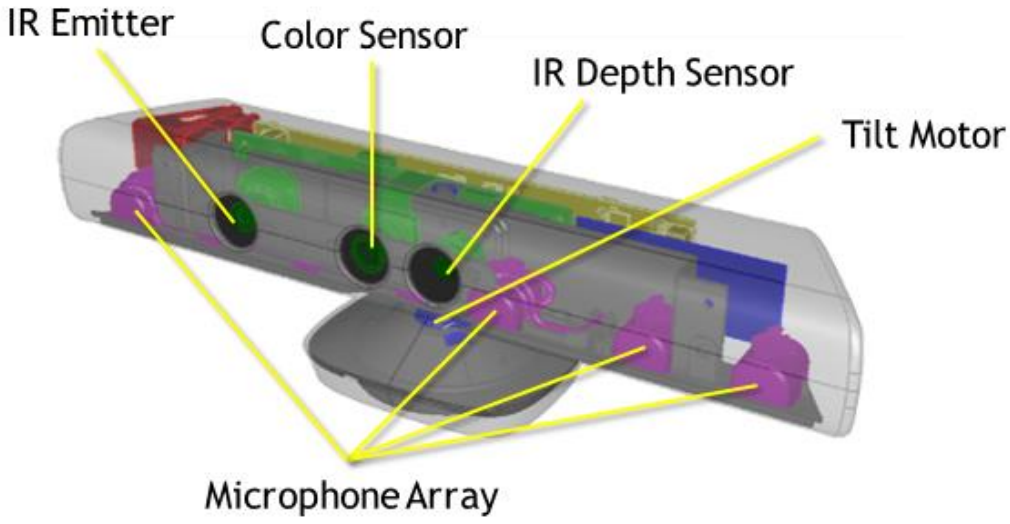
Glavni elementi tehičkog sustava su Microsoft Kinect, UR5 robot, računalo i loptica. Sustav je namješten tako da je kamera okrenuta prema robotu na kojem je postavljena kutija sa rupom u koju bi loptica, u slučaju uspješnog hvatanja, trebala upasti. Loptica je žute boje od spužvastog materijala, promjera šest centimetara. Razvijeni program na računalu u stvarnom vremenu računa položaj loptice u prostoru, a komunikacija sa robotom se vrši putem mreže, odnosno TCP protokola.



Slika 2.1: Tehnički sustav

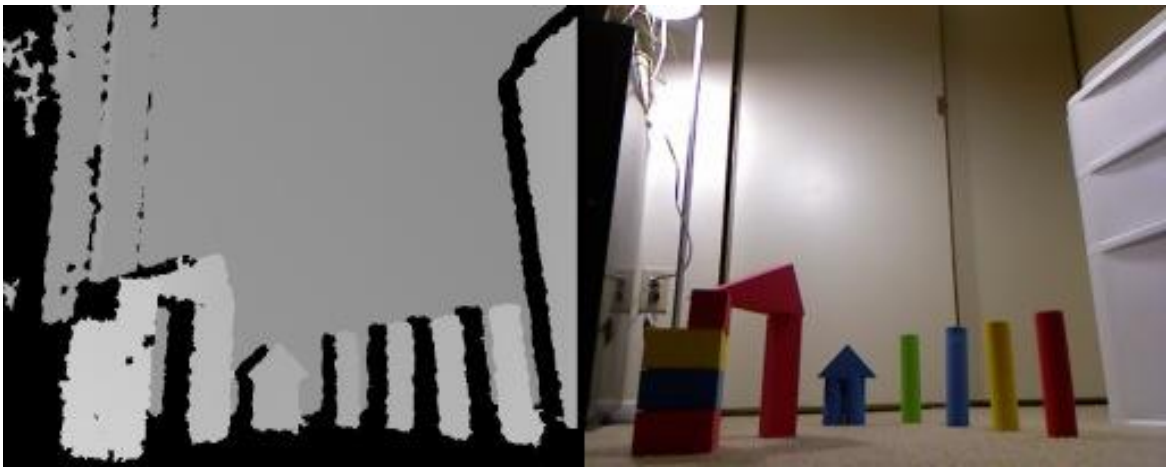
2.1. Microsoft Kinect

Microsoft Kinect je uređaj koji omogućuje prepoznavanje pokreta i ljudskog glasa koji služe za upravljanje Xbox 360 igraćom konzolom ili osobnim računalima s Windows operacijskim sustavom. Službeno je izašao na tržište u Siječnju 2010. i postavio novi rekord kao najbrže prodavani elektronički uređaj od 8 milijuna prodanih uređaja u prvih 60 dana, a do početka 2012. godine prodano je preko 18 milijuna uređaja što je čvrsti dokaz budućnosti koju ima ova tehnologija. Prvo je bio namijenjen samo Xbox igraćoj konzoli kako bi pružio novi način zabave i igranja igara. Ono što u početku nitko nije predvidio su nevjerojatne mogućnosti Kinect-a i njegova široka primjena. Odmah po izlasku na tržište krenula je utrka za probijanjem zaštite uređaja i izdavanjem upravljačkih programa za osobna računala. Open Source zajednica odmah je prepoznala primjenu koju Microsoft nije predvidio i omogućila korištenje Kinect-a na osobnim računalima. Microsoft je izdao službene upravljačke programe, SDK i Kinect uređaj za Windows računala u Veljači 2012.



Slika 2.2: Microsoft Kinect

Kinect omogućuje interakciju putem tri tipa senzora. Sliku prepoznaje uz pomoć video kamere razlučivosti 640x480 piksela i brzine trideset sličica u sekundi, zvuk pomoću četiri mikrofona te dubinu uz pomoć infracrvenog projektora i infracrvene kamere. Mogućnosti koje pruža Kinect garantira upravo dubinska kamera koja može prepoznati objekte ispred nje i izmjeriti udaljenost objekata od Kinect-a.



Slika 2.3: Slika dobivena dubinskom kamerom Kinect-a (lijevo) i slika dobivena RGB kamerom (desno)

Dubinsku kameru čine dva elementa: infracrveni projektor i infracrvena kamera. Infracrveni projektor koji raspršuje infracrvene točke po prostoru ispred senzora. Kada dođu do nekog objekta točke se odbijaju te ukoliko je objekt bliže, točke koje prekrivaju objekt će biti bliže jedna drugoj, dok će za udaljene objekte razmak između točaka biti veći. Drugi element je infracrvena kamera koja snima infracrvene točke raspršene po prostoru. Udaljenost između infracrvenih točaka koju snima kamera omogućuje izračunavanje dubinske mape koja govori koliko je pojedini objekt u sceni udaljen od kamere. Dubinska kamera omogućuje maksimalnu rezoluciju od 640x480 piksela.

2.2. Universal robots

Universal Robots je proizvođač fleksibilnih industrijskih robota sa sjedištem u Odenseu, Danskoj. Esben Østergaard, Kasper Støy i Kristian Kassov, osnivači tvrtke, su tijekom zajedničkog istraživanja na Sveučilištu Syddansk došli do zaključka da tržištem robota dominiraju teški, skupi i nezgrapni roboti. Zajedno su došli do zaključka da postoji potražnja za laganim robotima koji se lako programiraju i prilagođeniji su korisniku. Osnovana 2005. tvrtka je imala za cilj učiniti robotske tehnologije dostupne manjim i srednjim poduzećima. Od 2010. počeli su sa prodajom prvih UR5 robota na danskom i njemačkom tržištu i od tada tvrtka stalno proširuje svoje aktivnosti. UR10 i UR3 su plasirani na tržište 2010. i 2015. godine redom. Sa gotovo 140 zaposlenih i globalnom mrežom od 200 distributera u 50 zemalja diljem svijeta Universal Robots je jedna od najprogresivnijih tvrtki koje se bave proizvodnjom industrijskih robota. Njihova tri glavna proizvoda su roboti UR3, UR5 i UR10,

a brojke u imenu označavaju nosivost svakog robota u kilogramima. Lagani, šestoosni roboti teže 11 kg, 18 kg i 28 kg redom. Svi zglobovi mogu se rotirati ±360º brzinom do 180º/s, a zadnji zglob robota UR3 ima beskonačnu rotaciju. Ta tri robota su kolaborativni roboti što znači da mogu raditi uz osoblje bez sigurnosnih dodataka.

Slika 2.4: UR5 robot

U ovom radu korišten je UR5 robot nosivosti 5 kg. Ovaj lagani, fleksibilni i kolaborativni robot omogućava automatizaciju repetitivnih i opasnih zadataka, a idealan je za optimizaciju kolaborativnih procesa u kojima se koriste predmeti male mase kao što su podizanje i postavljanje te testiranje. Sa polumjerom od 850 mm ovom kolaborativnom robotu je sve na dohvat ruke i time omogućava radnicima da vrijeme posvete drugim fazama procesa. UR5 se lako programira, siguran je i kolaborativan što ga čini idealnim izborom za mnoga industrijska postrojenja. Iako namijenjen za industriju zbog navedenih mogućnosti UR5 se također može koristiti i u medicini ta za potrebe ispitivanja ponašanja robota u dinamičkoj okolini. Tehničke specifikacije navedene su u tablici 2.1.

Tablica 2.1: UR5 tehničke specifikacije

Težina:	18,4 kg
Nosivost:	5kg
Doseg:	850mm
Radni opseg zglobova:	±360º
Brzina svih zglobova:	180º/s
Brzina alata:	1m/s
Ponovljivost:	±0.1mm
Broj stupnjeva slobode gibanja:	6
Veličina upravljačke jedinice:	475mm x 432mm x 268mm
Komunikacija:	TCP/IP 100 Mbit & Modbus TCP
Programiranje:	Grafičko korisničko sučelje PolyScope
Potrošnja energije:	≈ 200W
Materijali:	Aluminij i Polipropilen (PP)
Temperaturno radno područje:	0ºC do 50ºC
Napajanje:	100 – 240 VAC, 50 – 60Hz

2.2.1. TCP protokol

TCP protokol jedan je od osnovnih protokola unutar IP grupe protokola, a naziv je kratica od engleskog naziva *Transmission Control Protocol.* TCP aplikacija korištenjem protokola na nekom od hostova umreženog u računalnu mrežu kreira virtualnu konekciju prema drugom hostu te putem ostvarene konekcije zatim prenosi podatke. Stoga za razliku od bespojnih protokola kakav je primjerice UDP, ovaj protokol spada u grupu tzv. spojnih protokola. TCP garantira pouzdanu isporuku podataka u kontroliranom redoslijedu od pošiljatelja prema primatelju, a osim toga pruža i mogućnost višestrukih istovremenskih konekcija prema jednoj aplikaciji na jednom hostu od strane više klijenata, gdje su najčešći primjeri za to web ili poslužitelji e-pošte. TCP podržava neke od najčešće korištenih aplikacijskih protokola na Internetu, kao što su HTTP(protokol za pregled web

stranica), SMTP (protokol za razmjenu elektroničke pošte), telnet i SSH (protokole za udaljeni rad na računalu) i brojne druge.

TCP upotrebljava određen raspon portova kojima razdjeljuje programe na strani pošiljatelja i primatelja. Svaka strana TCP konekcije ima dodijeljenu 16-bitnu oznaku za obje strane aplikacije (slanje, primanje). Portovi su u osnovi podijeljeni u 3 kategorije: poznati portovi (0 – 1023), registrirani portovi (1024 - 49150) i dinamički/privatni portovi (49151 - 65535).

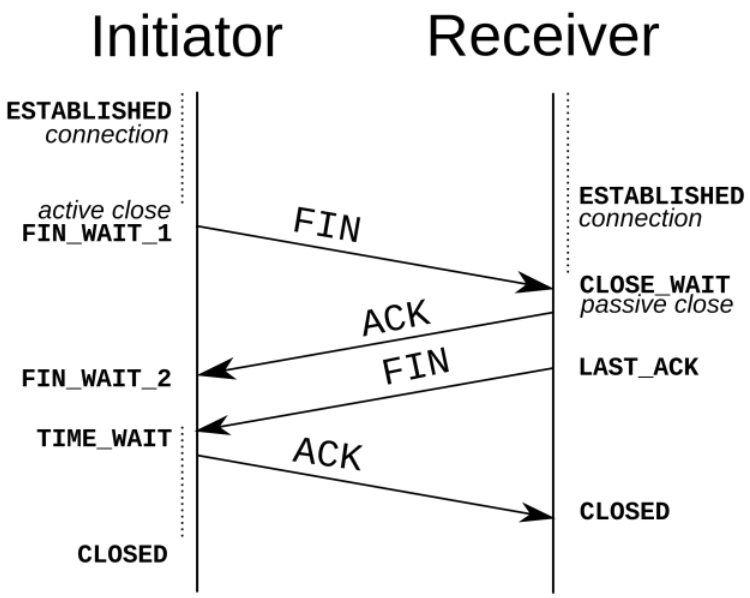
Proces koji se izvodi na jednom računalu prilikom uspostavljanja veze s procesom na drugom računalu naziva se uspostavljanje veze (eng. *Connection Establishment*). Korisnik (eng. *client*) je računalo koje traži supostavu veze, a drugo računalo se naziva poslužitelj (eng. *poslužitelj*). Korisnički proces informira njegov TCP da želi uspostaviti vezu s poslužiteljem. Korisničko računalo tada šalje poslužitelju prvi specijalni segment nakon čega poslužitelj odgovara drugim specijalnim TCP segmentom i konačno korisnik odgovara trećim specijalnim segmentom. Ova procedura se naziva "three-way handshake ". A koraci su objašnjeni u nastavku:

1. SYN (eng. *Synchronize Sequence Number*): Prvo računalo traži supostavu veze s drugim računalom stoga korisnik šalje TCP segment sa SYN-om drugom računalu. Ovaj segment informira poslužitelja da korisnik želi započeti komunikaciju s njim i sa kojim sekvencijskim brojem će započeti svoj segment. Taj segment ne sadrži podatke aplikacijske razine, a u zaglavlju segmenta jedan od bitova zastavnica – tzv. SYN bit je postavljen na 1. Potom korisnik odabire inicijalni redni broj (client_isn) i stavlja ga u polje za redni broj inicijalnog TCP SYN segmenta.

2. SYN-ACK: Poslužitelj će odgovoriti korisniku sa prihvaćanjem (eng. *acknowledgment*) i setom SYN bitova. Poslužitelj zatim prihvaća korisnikov SYN segment i informira korisnika s kojim sekvencijskim brojem započinju njegovi podaci. Taj TCP SYN segment odobravanja veze također ne sadrži podatke aplikacijske razine, ali sadrži tri važne informacije u zaglavlju segmenta. Prvo, SYN bit je postavljen na 1, drugo Acknowledgment polje zaglavlja TCP segmenta se namješta na isn+1 i treće, poslužitelj bira svoj inicijalni redni broj (poslužitelj_sin) i stavlja vrijednost u polje zaglavlja TCP segmenta.

3. ACK: Nakon primanja segmenta odobravanja veze korisnik također alocira spremnik i varijable u vezi. Korisniksko računalo tada šalje poslužitelju još jedan segment koji potvrđuje da je dobio segment odobravanja veze. To radi tako da stavi vrijednost poslužitelj_isn+1 u acknowledgment polje zaglavlja. SYN bit se postavlja na 0 budući da je veza uspostavljena.

Nakon ova tri koraka korisnik i poslužitelj mogu jedan drugome slati segmente koji sadrže podatke, a u svakom budućem segmentu SYN će biti postavljen na 0.



Slika 2.5: „Three-way handshake“

2.2.2. Komunikacija s računalom

Na UR5 robotu su pokrenuta četiri poslužitelja na četiri porta:

1. 29999 – Poslužitelj nadzorne ploče (eng. *Dashboard poslužitelj*)
2. 30001 – Primarni poslužitelj
3. 30002 – Sekundarni poslužitelj
4. 30003 – Real time poslužitelj

Kako bi komunikacija bila moguća i robot i računalo moraju biti u istoj podmreži.

1. Poslužitelj nadzorne ploče: Spajanjem na poslužitelj nadzorne ploče robota nije moguće programirati, ali je moguće upravljati raznim opcijama na njemu. Preko tog porta moguće je slati naredbe kao što su load (učitavanje programa), play (pokretanje programa), stop (zaustavljanje programa), pause (pauziranje programa) i shutdown (isključivanje programa). Korištenje tih naredbi može biti vrlo korisno u industriji gdje je više robota spojeno u jednu mrežu.

2. Primarni poslužitelj: Primarni poslužitelj služi za programiranje i dobivanje informacija o stanju robota. Nakon spajanja na poslužitelj podaci se primaju brzinom od 10Hz. Primarni

poslužitelj šalje stanje robota i dodatne poruke, a robotom je moguće upravljati preko tog poslužitelja tako da mu se šalju naredbe u obliku skripte. To može biti cijeli program ili samo jedna naredba. Pojedinačne naredbe koje se šalju robotu on izvršava trenutno neovisno o tome je li u radu ili miruje, a u slučaju slanja cijelog programa on ga izvršava nakon što primi zadnji red programa. Svaka naredba mora završavati sa „\n“ što predstavlja novi red.

3. Sekundarni poslužitelj: Za sekundarni poslužitelj vrijedi sve kao i za primarni osim što sekundarni poslužitelj za razliku od primarnog šalje samo stanje robota.

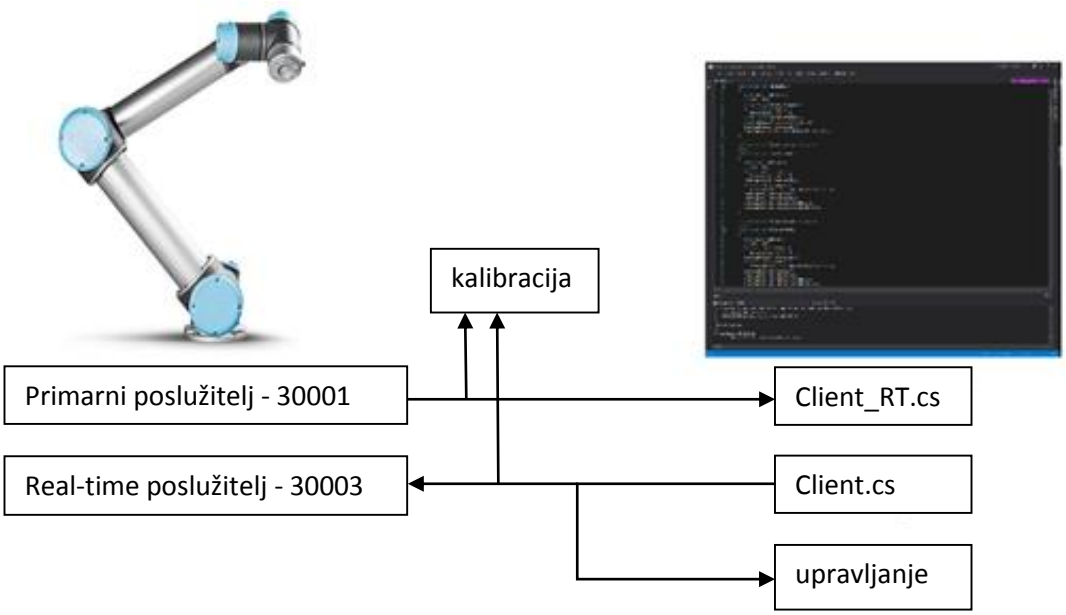
4. „Real time“ poslužitelj: „Real time“ poslužitelj svim spojenim korisnikima šalje informacije o stanju robota brzinom od 125Hz. Ovaj poslužitelj se najčešće koristi za brzo primanje stanja robota. Veličina i redoslijed podataka se razlikuju ovisno o verziji softvera na robotu. Ukupna veličina poruke je 1060 bajtova, a sve numeričke vrijednosti robot šalje u dvostrukoj preciznosti prema IEEE 754 standardu.

Tablica 2.2: Sadržaj poruke koju šalje Real-time sever

Naziv	Veličina [B]	Opis
Veličina poruke	4	Ukupnja duljina poruke u bajtovima
Vrijeme	8	Proteklo vrijeme od početka rada kontrolera
$\mathbf{q_t}$	48	Ciljani položaj svakog zgloba
$\mathbf{\dot{q}_t}$	48	Ciljana brzina svakog zgloba
$\mathbf{\ddot{q}_t}$	48	Ciljana akceleracija svakog zgloba
$\mathbf{I_t}$	48	Ciljana struja svakog zgloba
$\mathbf{M_t}$	48	Ciljani moment u svakom zglobu
$\mathbf{q_a}$	48	Trenutni položaj svakog zgloba
$\mathbf{\dot{q}_a}$	48	Trenutna brzina svakog zgloba
$\mathbf{I_a}$	48	Trenutna struja u svakom zglobu
$\mathbf{I_c}$	48	Napon kontrolera svakog zgloba
$\mathbf{TCP}_a$	48	Trenutne koordinate alata u kartezijevom k.s.
$\mathbf{T\dot{C}P_a}$	48	Trenutne brzine alata u kartezijevom k.s.
$\mathbf{F}$	48	Sile u alatu
$\mathbf{TCP}_t$	48	Ciljane koordinate alata u kartezijevom k.s.
$\mathbf{T\dot{C}P_t}$	48	Ciljane brzine alata u kartezijevom k.s.
Digitalni ulazi	8	Trenutno stanje digitalnih ulaza u bitovima

$\boldsymbol{\vartheta}$	48	Temperatura svih zglobova u ºC
Tajmer kontrolera	8	Vrijeme potrebno za izvršenje thread-a
Ispitna vrijednost	8	Vrijednost koju koristi samo UR softver
Mod robota	8	Način rada robota
Modovi zglobova	48	Način rada zglobova
Sigurnosni mod	8	Sigurnosni način rada
-	48	Vrijednost koju koristi samo UR softver
TC̈P	24	x, y i z vrijednosti ubrzanja alata
-	48	Vrijednost koju koristi samo UR softver
Brzina skaliranja	8	Brzina skaliranja trajektorije
p	8	Količina gibanja
-	8	Vrijednost koju koristi samo UR softver
-	8	Vrijednost koju koristi samo UR softver
$\mathbf{V_m}$	8	Glavni napon
$\mathbf{V}_r$	8	Napon robota
$\mathbf{I}_r$	8	Struja robota
$\mathbf{V}_a$	48	Trenutni naponi u zglobovima
Digitalni izlazi	8	Trenutno stanje digitalnih izlaza u bitovima
Stanje programa	8	Trenutno stanje programa
Ukupno	**1060**	

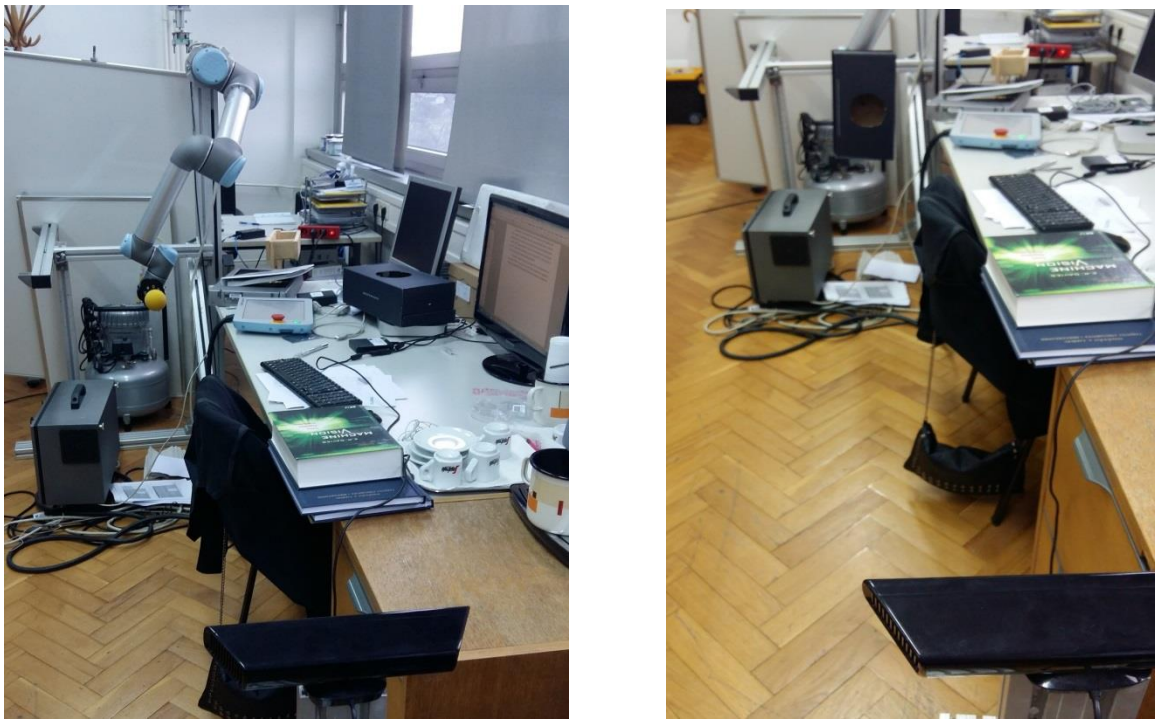
Uspostavljanje komunikacije između robota i računala neizostavno je ako želimo robota poslati u neku točku. U tom slučaju dovoljno je spojiti se na robotov primarni poslužitelj (30001) sa IP adresom: 192.168.0.9. TCP korisnik šalje robotu pojedinačnu naredbu u obliku stringa, a naredba *movej* sadrži podatke o poziciji zglobova, akceleraciji, brzini, vremenu itd. Pozicija robota u baznom koordinatnom sustavu definirana je sa x, y i z koordinatama i rotacijama Rx, Ry i Rz koje uvijek ostaju iste. Kako bi se povećala brzina varijabla vremena je postavljena na 0.12s što znači da se traži od robota da dođe u točku u roku od 0.12s.



Slika 2.6: Shema sustava

U slučaju kalibracije računalo se spaja na robotov „Real time“ poslužitelj na portu 30003. TCP korisnik šalje poruku robotu da dođe određenu točku, a cijelo vrijeme robot šalje nazad računalu vrijednosti svojih zglobova. Kada robot stigne u željenu točku greška se smanjuje približno na nulu i program započinje sa algoritmom za prepoznavanje loptice. Veličina cijele poruke je 1060 bajta, a trenutne vrijednosti koordinata su zapisane od 444. do 492. bajta. Veličina svake vrijednosti je 8 bajtova, a poruka se prima u obliku polja znakova gdje je svaki znak zapisan kao jedan bajt. Iz tog polja izvlači se po 8 znakova od 44. do 492. člana. Svaki znak se zatim pretvara u binarni oblik koji se sastoji od 8 bitova, a isti se zatim pretvara u string koji se sastoji od 64 bita. Dobivena vrijednost se zatim pretvara u numeričku vrijednost zapisanu u dvostrukoj preciznosti prema IEEE 754 standardu.

```
NetworkStream stream = client_RT.GetStream();
Byte[] data = new Byte[8192];
Int32 bytes = stream.Read(data, 0, data.Length);
double val = 0;
for (int j = 1; j <= 6; j++)
{
	string value = "";
	for (int i = 444 + (j - 1) * 8; i < 444 + j * 8; i++)
	{
		string bstr = Convert.ToString(data[i], 2);
		if (bstr.Length == 7)
		bstr = "0" + bstr;
		else if (bstr.Length == 6)
		bstr = "00" + bstr;
		else if (bstr.Length == 5)
		bstr = "000" + bstr;
		else if (bstr.Length == 4)
		bstr = "0000" + bstr;
		else if (bstr.Length == 3)
		bstr = "00000" + bstr;
		else if (bstr.Length == 2)
		bstr = "000000" + bstr;
		else if (bstr.Length == 1)
		bstr = "0000000" + bstr;
		value += bstr;
	}
if (j < 4)
val = BitConverter.Int64BitsToDouble(Convert.ToInt64(value, 2));
else if (j >= 4)
val = BitConverter.Int64BitsToDouble(Convert.ToInt64(value, 2));
robotPos.Add(val);
}
```



Slika 2.7: Tehnički postav za vrijeme kalibracije (lijevo) i bacanja (desno)

3. PREPOZNAVANJE LOPTICE

3.1. Računalni vid

Računalni vid je područje umjetne inteligencije koje se bavi prepoznavanjem dvodimenzionalnih i/ili trodimenzionalnih predmeta – na primjer, ljudskog lica. Bez razvijenog računalnog vida robot se ne može snalaziti u prostoru, što znači da može biti potencijalno opasan u slučaju ljudske prisutnosti u istom području. Računalni vid se fokusira na obradu i analizu 2D slike odnosno kako pretvoriti jednu sliku u drugu sa određenim operacijama nad pikselima kao što su uklanjanje šuma, poboljšanje kontrasta, izdvajanje rubova, zamagljivanje slike itd. Neki od zadataka računalnog vida su: prepoznvanje (objekata, lica, znakova...), analiza pokreta (eng. *egomotion, tracking, optical flow*...), rekonstrukcija događaja i restauracija slike. U području računalnog vida, prepoznavanje objekata opisuje zadatak pronalaženja i identificiranja objekata na slici ili video odsječku. Ljudi su sposobni prepoznati mnoštvo objekata na slici unatoč činjenici da slike objekata mogu varirati u različitim pogledima, dimenzijama, mjerilima, a čak i kada su objekti rotirani ili translatirani. Čovjek lako može prepoznati objekt i kada cijela slika istoga nije dostupna, no taj zadatak još uvijek nije jednostavan za računalo.

Prepoznavanje objekata ili utvrđivanje sadrži li slika određeni objekt, značajku ili aktivnost klasični je problem računalnog vida. Taj zadatak se obično može riješiti trivijalno i bez truda od strane čovjeka, ali još uvijek nije zadovoljavajuće riješen problem računalnog vida za opći slučaj – proizvoljni objekt u proizvoljnim situacijama. Postojeće metode za rješavanje ovog problema se u najboljem slučaju mogu primijeniti samo za pojedine objekte kao što su jednostavni geometrijski oblici, tiskano ili ručno pisani znakovi, ljudska lica itd.

Prepoznavanje loptice na slici dobivenoj od Microsoft Kinect-a se svodi na aplikaciju nekoliko algoritama računalnog vida kako bi se prepoznala kružnica koja omeđuje lopticu i definira položaj te dimenzije loptice u prostoru.

Neka od područja primjene su medicinski računalni vid ili medicinska obrada slike gdje je svrha vađenje podataka iz slike liječnička dijagnoza. Općenito, slikovni podatak je u obliku mikroskopskih slika, rendgenskih slika, ultrazvučnih slika i tomografskih slika. Primjer informacija koje se mogu izvući iz takvih slikovnih podataka je otkrivanje tumora, arterioskleroze i drugih malignih promjena. Drugo područje primjene računalnog vida je u

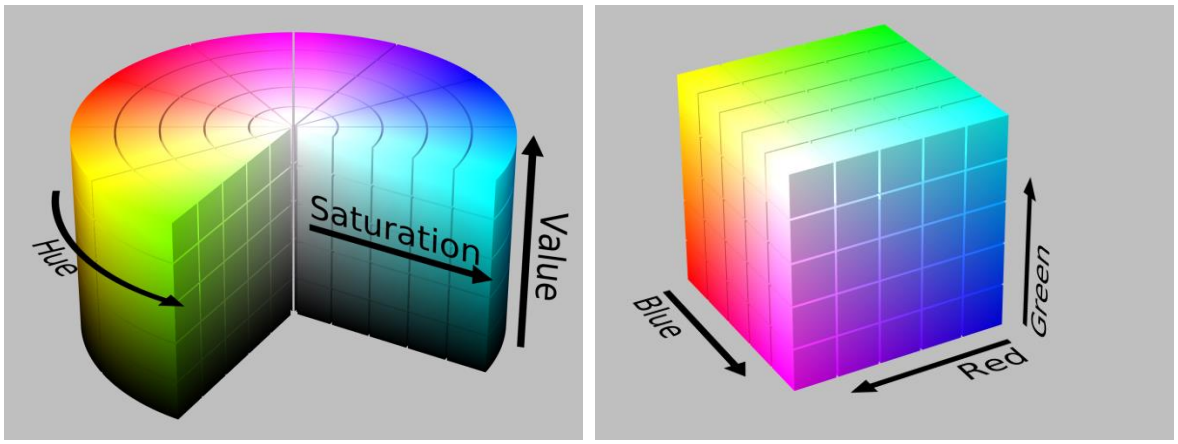
industriji, ponekad se zove strojni vid, gdje je informacija izvađena u svrhu poticanja proizvodnog procesa. Jedan primjer je kontrola kvalitete, a drugi je mjerenje položaja i orijentacije detalja kako bi je robotska ruka izdvojila iz proizvodnog procesa. Vojne aplikacije su vjerojatno jedan od najvećih područja računalnog vida, a očiti primjeri su otkrivanje neprijateljskih vojnika i vozila te navođenje raketnih sustava. Jedan od novijih područja primjene su bespilotna vozila, uključujući podmornice, kopnena vozila i bespilotne letjelice (UAV). Potpuno autonomna vozila obično koriste računalni vid za navigaciju, odnosno snalaženje u prostoru ili izradu karte svoje okoline (SLAM) i za otkrivanje prepreka. Svemirska istraživanja se već rade pomoću autonomnih vozila koja koriste računalni vid, npr., NASA-in Mars Exploration Rover i ESA-ExoMars Rover.

3.2. Razvoj vizijskog programa za prepoznavanje loptice

Program za prepoznavanje loptice implementiran je u Microsoft Visual Studio razvojnom okruženju, a korišteni programski jezik je C#. OpenCV vizijska biblioteka je prilagođena za C i C++ programske jezike stoga je u ovom radu upotrebljen višeplatformni omotač Emgu CV koji omogućava pozivanje OpenCV funkcija u .NET kompatibilnim jezicima kao što su C#, VB, VC++ itd. Biblioteka OpenCV-a sadržava više od 500 funkcija koje obuhvaćaju mnoga područja računalnog vida, uključujući inspekciju proizvoda u tvornicama, medicinsku dijagnostiku, osiguranje i nadzor, kalibraciju kamera, stereo-vid i robotiku. Budući da su računalni vid i strojno učenje bliska područja, OpenCV također sadržava i biblioteku strojnog učenja.

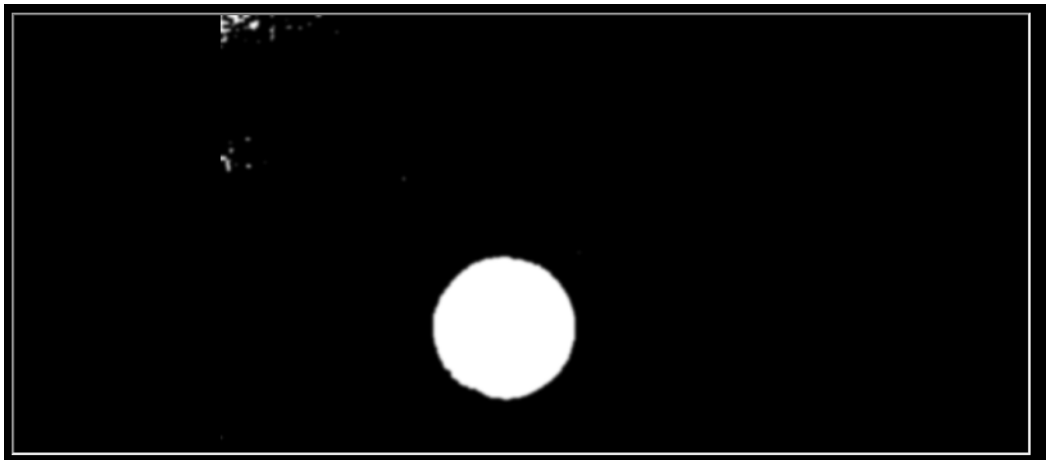
U ovom radu razvijen je algoritam za prepoznavanje loptice na temelju boje i oblika. Snimajući prostor Kinect-ovom RGB kamerom dobivamo informacije o 2D okolini koja je opisana matricom piksela sa određenim vrijednostima intenziteta. U [11,12] autori se služe Kinect-om, ali se prepoznavanje temelji na razlici između dvije dubinske mape. Kako bi postupak obrade slike dao što bolje rezultate prilikom prepoznavanja loptice, istu je potrebno adekvatno pretprocesirati. To u konkretnom slučaju uključuje izdvajanje žute boje i uklanjanje šumova. Dobivenu sliku je prvo potrebno iz RGB (Red Green Blue) prostora boja prebaciti u HSV (Hue Saturation Value) prostor. Svaka boja u RGB prostoru boja nastaje zbrajanjem pojedinih komponenata te tri boje. Sličan postupak primjenjuje se i u [8], gdje se RGB prostor prebacuje u HSL prostor boja, a nakon postupka binarizacije i primjena algortana za uklanjanje šuma pronalaze se oblici koji odgovaraju kružnicama.

RGB model predstavljen je pomoću kocke, gdje crvena boja predstavlja x-os, zelena boja y-os, a plava boja z-os. Taj se prostor boja najčešće koristi u računalima. Svaka boja najčešće je predstavljena sa 8-bita, odnosno vrijednostima od 0 do 255 (256 vrijednosti) što daje ukupno 2563 = 16777216 mogućih boja. Najčešće se taj prostor boja normira na vrijednosti od 0 do 1. RGB prostor boja jednostavan je za računalo, ali nije prikladan za čovjeka. Crvena, zelena i plava komponenta međusobno su korelirane tako da je čovjeku vrlo teško izborom tih komponenata definirati željenu boju u RGB prostoru boja. Stoga se najčešće koriste neki drugi prostori boja kao što su npr. HSV (HSI, HSB) ili HSL prostor boja.



Slika 3.1: Usporedba RGB i HSV prostora boja

HSV prostor boja stvorio je A. R. Smith 1978. Taj prostor boja definiran je s tri koordinate: tonom boje (engl. hue), zasićenjem boje (engl. saturation) i svjetlinom boje (engl. value, intensity, brightness). Ton boje predstavljen je kutom od 0° do 360°. Zasićenost boje ima vrijednost od 0% do 100%. Svjetlina boje ima vrijednost od 0% do 100%. HSV prostor boja predstavljen je pomoću valjaka. Često se taj prostor boja prikazuje kao stožac ili šesterostrana piramida, jer je percipirana promjena zasićenja boje od 0% do 100% manja za tamne boje (one koje imaju manju vrijednost svjetline) nego za svijetle boje (one koje imaju veću vrijednost svjetline). Da bi se nadoknadila ta razlika u percepciji, valjak se izobličuje u stožac. Pokazalo se da je u ovom prostoru boja čovjeku daleko lakše (intuitivnije) definirati i izabrati boju nego u RGB prostoru boja. Često se vrijednosti tona, zasićenja i svjetline boje normiraju na vrijednosti od 0 do 1 [17].



Slika 3.2: Prikaz loptice nakon segmentacija žute boje

Slika se prvo prebacuje u HSV prostor, a zatim se izdvajanje žute boje temelji na podešavanju vrijednosti tri matrice. Konkretno, za žutu boju dovoljno je podesiti vrijednosti matrice tona boja tako da je minimalna vrijednost 20, a maksimalna 30. Te vrijednosti su dobivene ekspreminetalno i ovise o nijansi žute boje, osvjetljenju itd. Standardne vrijednosti za osnovne boje su:

Narančasta: 0 – 22

Žuta: 22 – 38

Zelena: 38 – 75

Plava: 75 – 130

Ljubičasta: 130 – 160

Crvena: 160 – 179

```
Bitmap bmpFrame = ColorImageFrameToBitmap(colorFrame);
Image<Bgr, Byte> colorImage = new Image<Bgr, Byte>(bmpFrame);

using (Image<Hsv, byte> hsv = colorImage.Convert<Hsv, byte>())
{
	Image<Gray, byte>[] channels = hsv.Split();
	try
	{
		CvInvoke.cvInRangeS(channels[0], new Gray(22).MCvScalar, new
	Gray(38).MCvScalar, channels[0]);
		CvInvoke.cvErode(channels[0], channels[0], (IntPtr)null, 2);
		channels[0]._SmoothGaussian(9);
		if (calibration == true)
		{
			calibration
		}
		if (started == true)
		{
			if (pbf == false)
```

```
                    {
                        for (int i = 0; i < 3; i++)
                        {
                            times.Add(time);
                            Vector3D pointBefore = new Vector3D(320.0, 240.0,
                        500.0);
                            CurveFitting.Fit(pointBefore, times[times.Count() -
                        1]);
                            time = time + ((1 / 30.0) * 1000);
                            lastDetected = pointBefore;
                            pbf = true;
                        }
                    }
                    contours
                }
                picVideoDisplay.Image = colorImage.ToBitmap();
            }
        }
```

Za uklanjanje šuma korišten je Gaussov filtar nakon čega dobijemo zamagljenu sliku kao rezultat primjene Gaussove funkcije na matricu slike. Zamagljivanje pomoću Gaussovog filtra je način zamagljivanja koji koristi Gaussovu funkciju (koja predstavlja normalnu distribuciju u statistici) za računanje transformacije koja se primjenjuje na svaki piksel na slici. Jednadžba Gaussove funkcije za jednodimenzijalni slučaj:

$$G(x) = \frac{1}{\sqrt{2\pi\sigma}} e^{-\frac{x^2}{2\sigma^2}} \tag{3.1}$$

Efekt se generira konvolucijom slike sa jezgrom koje imaju Gaussove vrijednosti. Postupak je najbolje podijeliti u dva koraka. U prvom jednodimenzijalna jezgra se koristi za zamagljivanje u jednoj dimenziji, a u drugom u preostaloj. Jedna od najčešćih primjena ovog filtra je za nalaženje rubova. Većina algoritama za pronalaženje rubova je jako osjetljiva na šum, a upotreba Gaussovog filtra smanjuje udio šuma na slici.

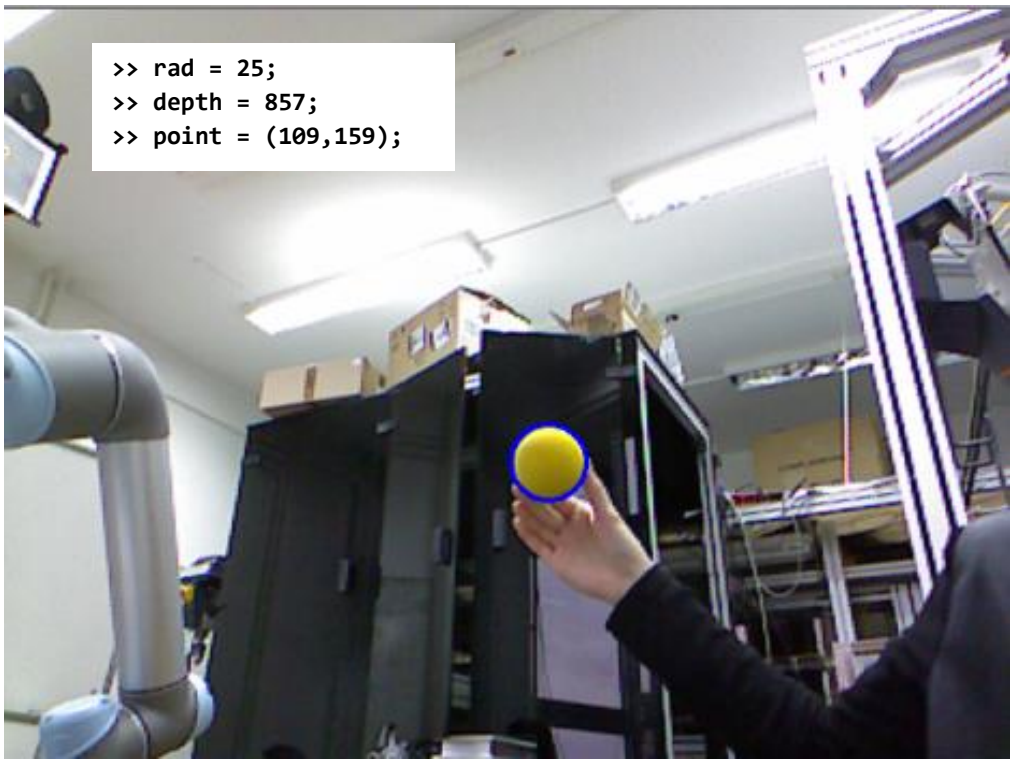
Slika 3.3: Primjena Gaussovog filtra na sliku

Za pronalaženje kružnica na slici najčešće se koristi OpenCV funkcija *HoughCircles*. Houghova transformacija je tehnika obrade slika koja se uobičajeno koristi za pronalazak linija ili kružnica na slici. Za opis rada Houghove transformacije podrazumijeva se kako se na ulaznoj slici prethodno izvelo prepoznavanje ruba. Dobiveni pikseli rijetko predstavljaju stvarne linije. Uzrok tome može biti prisutnost šuma, neravnomjerna osvijetljenost i ostali efekti koji utječu na diskontinuitete u intenzitetu. Hough transformacija se može upotrijebiti za određivanje parametara kružnice kada je poznat broj točaka koje padaju na obod kružnice.

Jednadžba kružnice:

$$(x-a)^2 + (y-b)^2 = r^2 \tag{3.2}$$

Svaka točka (x,y) se u parametarskom prostoru transformira u skup kružnica sa središtem u koordinatama (x,y). Recimo da poznajemo polumjer kružnice, svaka točka te početne kružnice tada se transformira u novu kružnicu. Kada se ovaj postupak provede nad svim točkama kružnice dobiva se skup kružnica istog polumjera koje se sijeku u jednoj točki čije koordinate određuju koordinate središta početne kružnice.



Slika 3.4: Prepoznavanje žute loptice

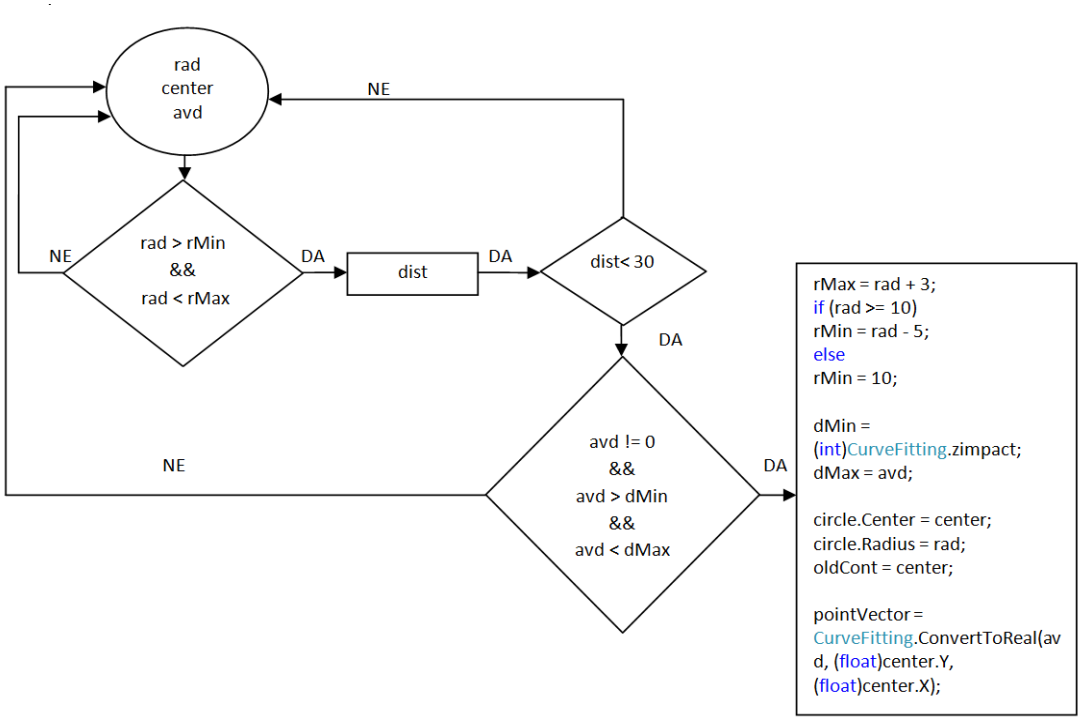
Ipak, u ovom radu korištena je funkcija za pronalazak kontura na slici koja je dala bolje rezultate kod visokih šumova. Ova funkcija nalazi konture svih oblika žute boje, a svaka kontura može biti aproksimirana uokvirujućim pravokutnikom, a potom i kružnicom. Na taj način svaki žuti oblik na slici predstavljen je kružnicom određenog polumjera i središta definiranog x i y koordinatom u pikselima. Za centar svake konture udaljenost (dubina) od uređaja se računa korištenjem dubinske kamere. Kako su vrijednosti dobivene dubinskom kamerom u pikselima potrebno je napraviti konverziju iz piksela u stvarne vrijednosti u milimetrima.

```
public static Vector3D ConvertToReal(int zv, float yv, float xv)
{
        double fh= 0;
        double fv = 0;
        int h = 640;
        int v = 480;
        double yw;
        double xw;
        Vector3D realWorld = new Vector3D();

        fh = h / (2 * Math.Tan(62 * Math.PI / 360));
        fv = v / (2 * Math.Tan(48.6 * Math.PI /360));
        double alpha = Math.Atan((320 - xv) / fh);
        double beta = Math.Atan((yv - 240) / fv);
        double zwh = Math.Cos(alpha * Math.PI / 180) * zv;
        double zwv = Math.Cos(beta * Math.PI / 180) * zv;

        xw = Math.Sin(alpha) * zv;
        yw = Math.Sin(beta) * zv;

        realWorld.X = xw;
        realWorld.Y = yw;
        realWorld.Z = zwh;

        return realWorld;
}
```

Koordinati sustav Kinect-a se definira tako da je ishodište u kućištu uređaja, x-os je paralelna sa duljom stranicom Kinect-a, a z-os ide iz uređaja. Y-os je određena pravilom desne ruke. Program mora u svakoj sličici pronaći lopticu iz skupa kontura i pripadajućih udaljenosti središta konture od uređaja. Filtracija se provodi u više koraka.

U prvom koraku se za svaku konturu u prvom trenutku računa polumjer, centar kružnice i udaljenost središta kružnice od Kinect-a. Zatim se provjerava je li izračunati polumjer veći od najmanjeg i manji od najvećeg za taj vremenski trenutak. Vrijednosti najmanjeg i najvećeg polumjera određene su na temelju veličine loptice za prvi trenutak, a u svakom sljedećem minimalna vrijednost postaje vrijednost zadnjeg polumjera umanjena za pet milimetara, a maksimalna vrijednost postaje vrijednost zadnjeg polumjera uvećana za tri milimetra. To je

zbog toga što se udaljavanjem loptice od kamere, polumjer kružnice u principu smanjuje, ali su uzete u obzir i greške zbog osvijetljenja, drugih mogućih objekata itd. U sljedećem koraku se računa udaljenost između starog središta kružnice i novog. Ta udaljenost treba biti manja od trideset milimetara čime se smanjuje mogućnost prepoznavanja drugih objekata u kadru. U zadnjem koraku se ispituje zadovoljava li izračunata dubina uvjet da je manja od najveće i veća od najmanje vrijednosti slično kao i za polumjer. Zbog nesavršenosti dubinske kamere dodan je još jedan korak u kojem se radi linearna regresija i predviđanje vrijednosti dubine u slučaju kada je izračunata vrijednost nekonzistentna sa prošlim vrijednostima. S obzirom na to da se i polumjer i dubina linearno mijenjaju udaljavanjem od kamere, predviđanje se temelji na odnosu vrijednosti polumjera i dubina u prošlosti. Time se značajno povećao broj točaka za predviđanje, a nije se značajno narušila točnost.



Slika 3.5: Prikaz algoritma za prepoznavanja loptice

3.3. Problemi i zaključak

Zbog nesavršenosti kamere, šumova i raznih drugih problema kao što su veličina loptice, boja loptice, vanjsko osvjetljenje, broj predmeta u kadru itd. glavni problem u radu je bilo upravo prepoznavanje loptice u svakoj sličici. Problem prepoznavanje loptice i njegovo riješenje u ovom radu je samo specifičan slučaj gdje se prepoznavanje objekta na slici temelji na boji i obliku. Kada bi loptica bila neke druge boje ili oblika program ne bi funkcionirao. Da bi se izbjegao taj problem jedna od ideja je također bila koristiti samo infracrvenu kameru Kinect-a koja omogućuje dobivanje dubinske mape za svaku sličicu. U dubinskoj mapi boja svakog piksela predstavlja udaljenost te točke od Kinect-a. Tako bi se svaka sličica mogla uspoređivati sa prethodnim i promatrati koji su pikseli najviše promijenili dubinu. Ako neka grupa piksela zadovoljava uvjet da je promjena dubine veća od neke granične onda se može pretpostaviti da ta grupa piksela označava lopticu. Problem sa ovim pristupom je u tome što je infracrvena kamera Kinect-a sklona šumu i za velike udaljenosti loptice od kamere dobivene vrijednosti su nepouzdane. Zato se ostalo na prvotnom pristupu koji se pokazao primjenjivijim u ovom slučaju.

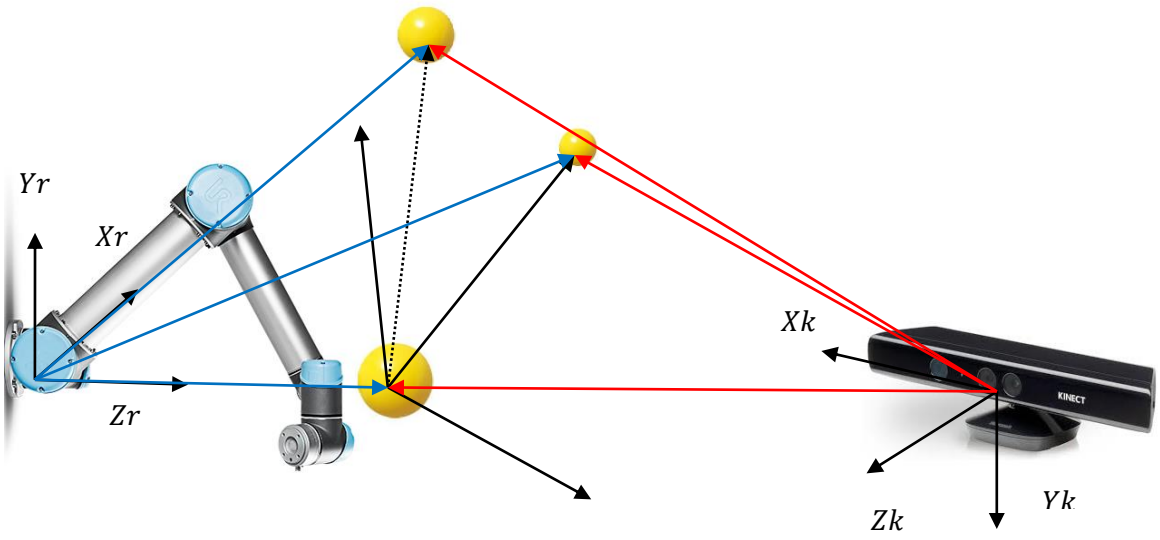
Drugi problem je veličina loptice koja treba biti dovoljno velika za prepoznavanje, ali i dovoljno lagana i mala za hvatanje. Zato je odabrana spužvasta loptica promjera šest centimetara. U budućnosti bi se ta loptica mogla zamijeniti sa nekom većom i lakšom te usporediti rezultate. Tako bi se udaljenost Kinect-a od robota mogla povećati, a da se pri tome ne naruši točnost prepoznavanja, a posljedično i predviđanja. Veća loptica znači i veću točnost prepoznavanja, a posljedično i kalibracije te predviđanja. Tada također ne bi bilo potrebno povećavati brzinu i akceleraciju gibanja robota.

4. KALIBRACIJA ROBOTA I KAMERE

4.1. Kalibracije pomoću tri točke

Položaj Kinect-a u prostoru je proizvoljan, a razvijena metoda kalibracije ne zahtjeva da položaj Kinect-a uvijek bude isti. Važno je samo da uređaj ne bude pomaknut za vrijeme kalibriranja i bacanja. Kinect se postavlja tako da točnost predviđanja bude što veća i na način da se dubina s vremenom povećava linearno. Microsoft Kinect je okrenut prema robotu, a bacanje se odvija tako da se loptica kreće od Kinect-a prema robotu (bacač stoji pored ili iza Kinect-a). S obzirom na to da čak i mali pomak Kinect-a u prostoru znači drugačije vrijednosti u milimetrima, potrebno je svaki put napraviti kalibraciju kako bi robot mogao sa što većom točnošću doći u točku hvatanja.

Kako se u svakom vremenskom koraku spremaju x, y i z vrijednosti u milimetrima predviđanje također daje vrijednosti u milimetrima, odnosno položaj u koordinatnom sustavu Kinect-a. Ta vrijednost ne znači ništa ako robot ne zna koja je to točka u njegovom koordinatnom sustavu te je kalibracija nužna kod svakog bacanja. Nakon kalibracije, Kinect se ne smije pomicati prije i tijekom bacanja. Postavljanje Kinect-a u prostoru je kompromis između točnosti kalibracije i predviđanja, a približno idealne vrijednosti su dobivene eksperimentalno.



Slika 4.1: Shema postupka kalibracije

Kalibracija pomoću tri točke omogućava pronalaženje koordinatnog sustava koji je zajednički robotu i Kinect-u. Imaginarni koordinatni sustav ima ishodište u prvoj točki kalibracije, a proces kalibracije započinje kada robot zajedno s lopticom dođe u tu prije definiranu točku. Kako bilo koju ravninu jednoznačno definiraju tri točke, potrebno je proizvoljno odrediti i druge dvije točke koje ravnina xy imaginarnog koordinatnog sustava definira. Smjer i orijentacija z-osi dobiju se računanjem vektorskog produkta x i y vektora gdje je x vektor usmjeren od prve do druge točke, a y od prve do treće. Računanjem vektorskog produkta između z i x osi dobiva se točan smjer y vektora te ta tri vektora čine trodimenzionalni imaginarni Kartezijev koordinatni sustav.

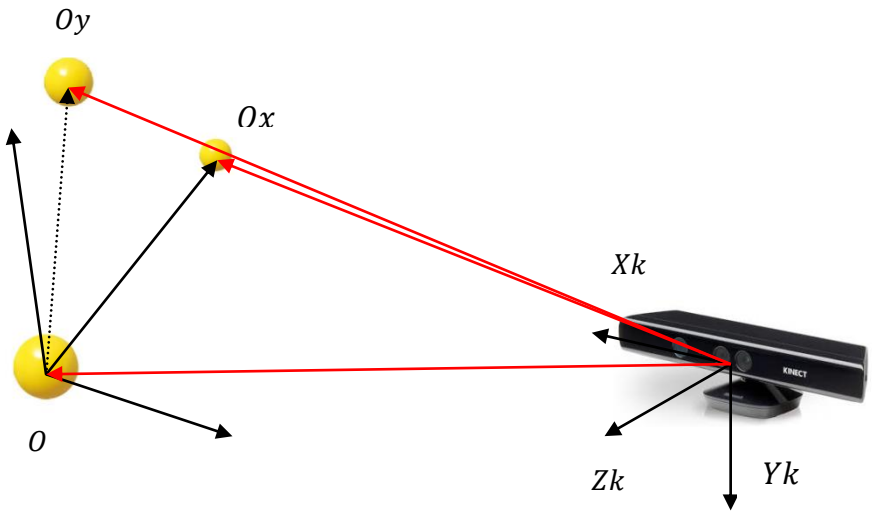
```
calibration = false;
PointF calCenter = new PointF();
CircleF calCircle = new CircleF();

using (MemStorage stor = new MemStorage())
{
	for (var contours =
channels[0].FindContours(CHAIN_APPROX_METHOD.CV_CHAIN_APPROX_SIMPLE,
RETR_TYPE.CV_RETR_EXTERNAL); contours != null; contours = contours.HNext)
	{
		if (robpoint == 0)
		{
			int calRad = (contours.BoundingRectangle.Height +
		contours.BoundingRectangle.Width) / 4;
			calCenter = contours.GetMinAreaRect().center;
			int calDepth = depthPixels[(int)calCenter.X + (int)calCenter.Y *
		depthFrame.Width].Depth;

			if (calRad <= 15 && calRad >= 8 && calDepth < 2000 && calDepth >
		1380 && calDepth != 0)
			{
				calCircle.Center = calCenter;
				calCircle.Radius = calRad;

				Vector3D realCoord =
			CurveFitting.ConvertToReal((int)calDepth, calCenter.Y,
			calCenter.X);

				if (System.Windows.Forms.MessageBox.Show("Ball found!\n" +
			"Coordinates:" + realCoord.X.ToString() + "," +
			realCoord.Y.ToString() + "," + realCoord.Z.ToString() + "\nPlease
			confirm the position:", "Confirm", MessageBoxButtons.YesNo,
			MessageBoxIcon.Question) == System.Windows.Forms.DialogResult.Yes)
				{
					kinectPoints.Add(realCoord);
					robpoint++;
					i++;
				}
			 }
		}
	}
}
```

Postupak prepoznavanja loptice za potrebe kalibracije sličan je onome za prepoznavanje loptice u letu osim što se proces filtracije kontura razlikuje. Kako bi kamera Kinect-a uspješno prepoznala lopticu na nekoj udaljenosti od uređaja potrebno je postaviti zahtjeve na polumjer kružnice, položaj središta kružnice i udaljenost središta od kamere. Za svaku sličicu se pronalaze konture koje zadovoljavaju dva uvjeta određena eksperimentalno. Prvi je da je polumjer kružnice koja opisuje konturu između nekih prije definiranih vrijednosti te da je udaljenost središta kružnice od Kinect-a veća od neke također prije definirane vrijednosti. Ako bi Kinect bio postavljen na drugačiji način, npr. da kamera gleda od robota ili da je postavljena pod nekim kutom, te vrijednosti bi trebale biti promijenjene.

4.2. Koordinatni sustavi

Kinect i robot imaju vlastite koordinatne sustave te je potrebno naći matrice transformacija iz Kinect-ovog u robotov kako bi se mogla odrediti točka hvatanja robota u prostoru.

Kao što je prije spomenuto koordinatni sustav Kinect-a definira tako da je ishodište u kućištu uređaja, x-os je paralelna sa duljom stranicom Kinect-a, a z-os ide iz uređaja. Y-os je određena pravilom desne ruke. Prilikom prepoznavanja loptice vrijednosti u milimetrima označavaju položaj središta loptice u koordinatnom sustavu Kinect-a, a da bi se te vrijednosti prebacile u koordinatni sustav robota definira se novi, imaginarni koordinatni sustav. Imaginarni koordinatni sustav ima ishodište u prvoj točki kalibracije. Položaj imaginarnog koordinatnog sustava u odnosu na onaj Kinecta određuje se u tri koraka. Nakon prepoznavanja loptice u prvoj točki kalibracije, a potom i drugoj, oduzimanjem vrijednosti i normaliziranjem dobije se vektor u prostoru čije vrijednosti predstavljaju položaj osi x imaginarnog koordinatnog sustava u odnosu na onaj Kinect-a. Prepoznavanjem u trećoj točki kalibracije i oduzimanjem vrijednosti treće i prve točke novi vektor, koji označava položaj osi y imaginarnog sustava, služi za definiranje osi z u prostoru. Položaj osi z određen je vektorskim produktom x i y osi, a nakon normalizacije vektorski produkt između osi x i z korigira vektor y u prostoru. Tri imaginarne osi čine desnokretni koordinatni sustav gdje je svaka os jedinični vektor.



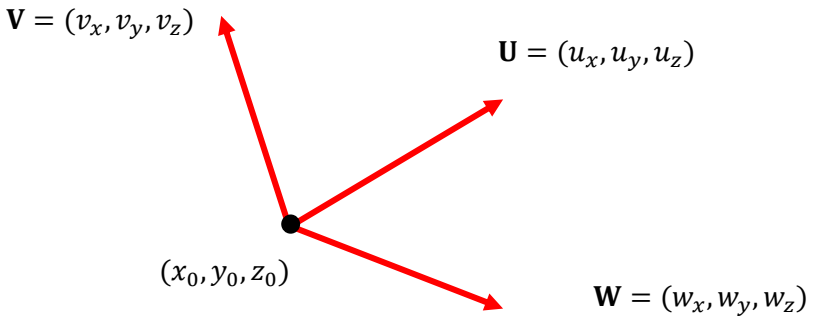
Slika 4.2: Opis računanja matrice transformacije iz Kinect-ovog u imaginarni k.s.

Stupce rotacijskog dijela matrice transformacija iz Kinect-ovog u imaginarni koordinatni sustav čine upravo vrijednosti jediničnih vektora x, y i z osi imaginarnog koordinatnog sustava. Matrica dimenzija 3x3 proširuje se matricom translacije koja predstavlja udaljenost imaginarnog sustava od Kinect-ovog sustava. Udaljenosti u milimetrima dobivene nakon prve prepoznavanja predstavljaju tu translaciju u x, y i z smjeru. Opći oblik matrice rotacije:

$$\mathbf{R} = \begin{bmatrix} R_{11} & R_{12} & R_{13} \\ R_{21} & R_{22} & R_{23} \\ R_{31} & R_{32} & R_{33} \end{bmatrix} \tag{4.1}$$

Ako je koordinatni sustav zadan sa tri trodimenzionalna vektora $\vec{u}$, $\vec{v}$ i $\vec{w}$ tada orijentacija tog koordinatnog sustava glasi:

$$\mathbf{R} = [\vec{u} \quad \vec{v} \quad \vec{w}] = \begin{bmatrix} u_x & v_x & w_x \\ u_y & v_y & w_y \\ u_z & v_z & w_z \end{bmatrix} \tag{4.2}$$



Slika 4.3: Koordinatni sustav opisan s tri vektora

Uobičajene transformacije u računalnoj grafici kao što su translacija, rotacija, skaliranje, projekcije itd. se lakše izvode ako se koriste homogene koordinate, jer se onda te transformacije mogu implementirati u obliku operacija s matricama. Korištenje homogenih koordinata znači dodavanje četvrtog reda te se na taj način svaka točka u prostoru prikazuje sa četiri vrijednosti. Inverzijom prve matrice transformacije i množenjem sa vrijednostima dobivenim iz Kinect-a lako se određuje položaj loptice u imaginarnom koordinatnom sustavu. Tako nakon množenja inverza matrice transformacije sa vrijednostima dobivenim za prvu točku rezultat mora biti (0,0,0,1) budući da je to ishodište imaginarnog koordinatnog sustava. Jednadžbe za kalibraciju su navedene u nastavku:

$$\overrightarrow{OO_x} = \|O_x - O\| \tag{4.3}$$

$$\overrightarrow{OO_y} = \|O_y - O\| \tag{4.4}$$

$$\vec{z} = \overrightarrow{OO_x} \times \overrightarrow{OO_y} \tag{4.5}$$

$$\overrightarrow{OO_x} = \vec{x} \tag{4.6}$$

$$\vec{y} = \vec{z} \times \vec{x} \tag{4.7}$$

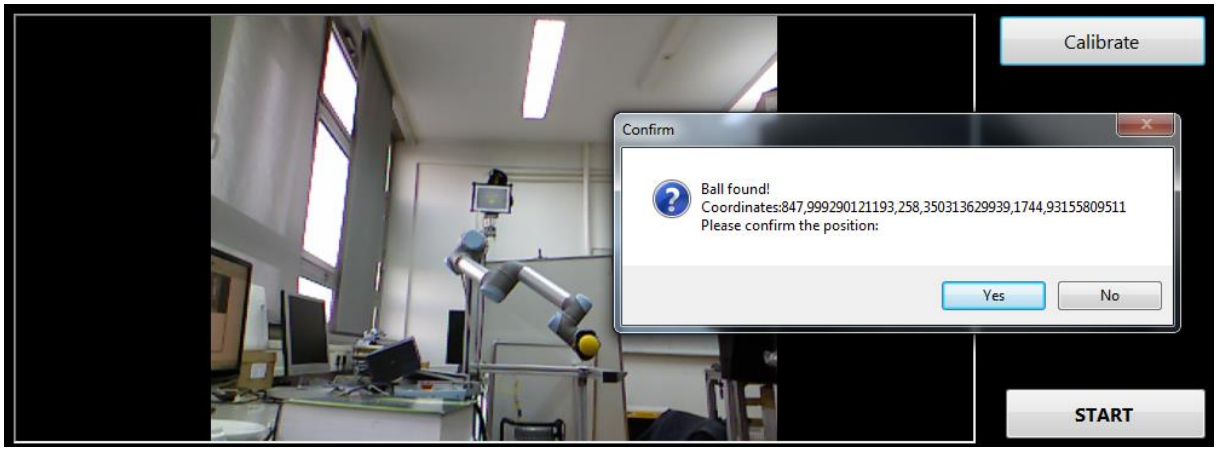
$$\mathbf{{}^{k}_{B}T} = \begin{bmatrix} x_x & y_x & z_x & O_x \\ x_y & y_y & z_y & O_y \\ x_z & y_z & z_z & O_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{4.9}$$

Isti postupak se primjenjuje ponovo kako bi se našla matrica transformacije iz imaginarnog u robotov koordinatni sustav. Analogno, robot se dovodi u tri točke redom i vrijednosti u svakoj točki označavaju udaljenost TCP-a u baznom koordinatnom sustavu robota. Ponavljanjem istog postupka definira se položaj (rotacija i translacija) imaginarnog koordinatnog sustava u odnosu na onaj robota. Razlika između prvog i drugog koraka kalibracije je što se u slučaju prebacivanja iz imaginarnog u robotov koordinatni sustav, matrica transformacije, a ne njezin inverz, množi sa vrijednostima u imaginarnom koordinatnom sustavu. Rezultat su četiri broja, od čega prva tri označavaju položaj neke točke u koordinatnom sustavu robota što nakon predviđanja točke hvatanja u koordinatnom sustavu Kinect-a i želimo dobiti. Za računanje matrica transformacija koristi se biblioteka Math.NET. Math.NET je biblioteka za izgradnju i održavanje alata potrebnih za računanje osnovnih i naprednih matematičkih funkcija .Net razvijača. Funkcije iz linearne algebre Math.NET biblioteke služe za operacije nad matricama kao što su množenja matrica, računanje inverza i množenja matrica s vektorom.

```
#region kinectToBase

double dk1 = Math.Sqrt(Math.Pow((kinectPoints[2].X - kinectPoints[0].X), 2) +
Math.Pow((kinectPoints[2].Y - kinectPoints[0].Y), 2) + Math.Pow((kinectPoints[2].Z -
kinectPoints[0].Z), 2));

imagKinect_X.X = (kinectPoints[2].X - kinectPoints[0].X) / dk1;
imagKinect_X.Y = (kinectPoints[2].Y - kinectPoints[0].Y) / dk1;
imagKinect_X.Z = (kinectPoints[2].Z - kinectPoints[0].Z) / dk1;

imagKinect_Y.X = (kinectPoints[1].X - kinectPoints[0].X);
imagKinect_Y.Y = (kinectPoints[1].Y - kinectPoints[0].Y);
imagKinect_Y.Z = (kinectPoints[1].Z - kinectPoints[0].Z);

imagKinect_Z = Vector3D.CrossProduct(imagKinect_X, imagKinect_Y);
double dk2 = Math.Sqrt(Math.Pow(imagKinect_Z.X, 2) + Math.Pow(imagKinect_Z.Y, 2) +
Math.Pow(imagKinect_Z.Z, 2));
imagKinect_Z.X = imagKinect_Z.X / dk2;
imagKinect_Z.Y = imagKinect_Z.Y / dk2;
imagKinect_Z.Z = imagKinect_Z.Z / dk2;

imagKinect_Y = Vector3D.CrossProduct(imagKinect_Z, imagKinect_X);

double[,] kToB = new double[,] { { imagKinect_X.X, imagKinect_Y.X, imagKinect_Z.X,
kinectPoints[0].X }, { imagKinect_X.Y, imagKinect_Y.Y, imagKinect_Z.Y,
kinectPoints[0].Y }, { imagKinect_X.Z, imagKinect_Y.Z, imagKinect_Z.Z,
kinectPoints[0].Z }, { 0, 0, 0, 1 } };
kinectToBase = MathNet.Numerics.LinearAlgebra.Double.Matrix.Build.DenseOfArray(kToB);

#endregion kinectToBase
```

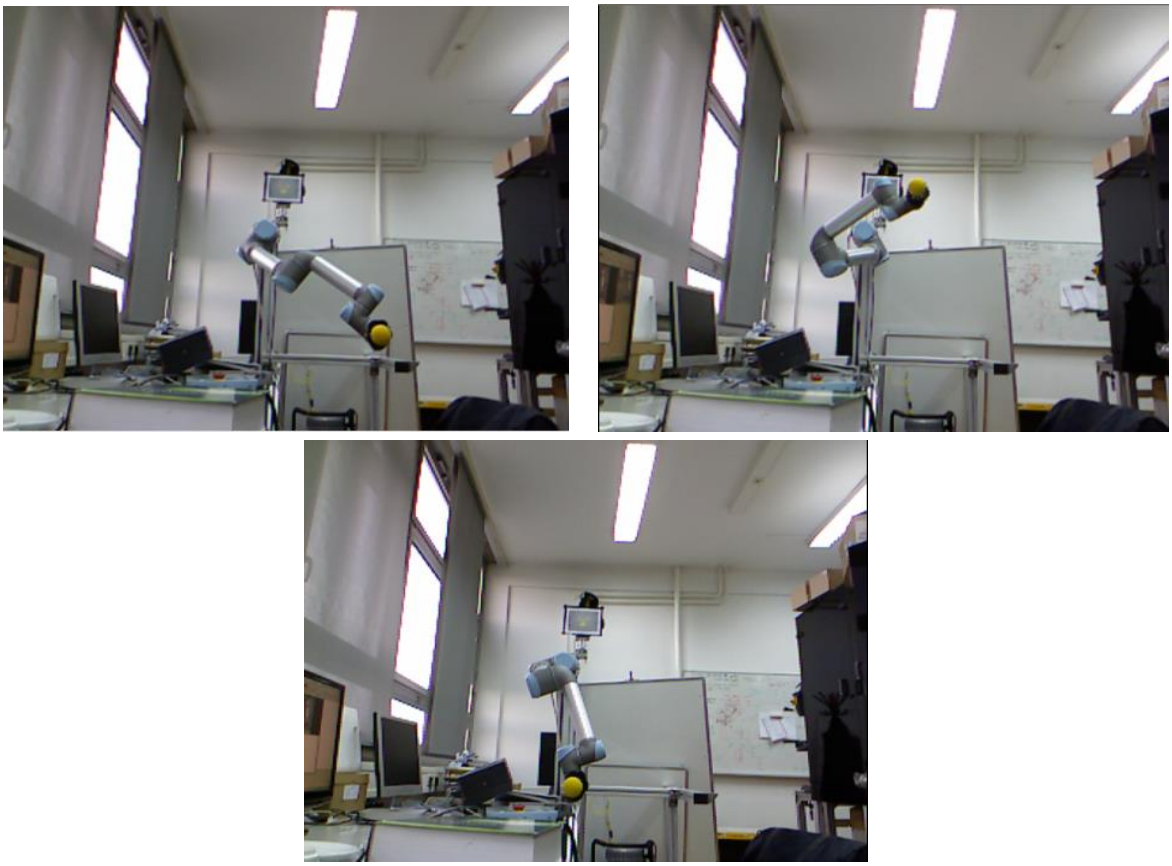
4.3. Postupak kalibracije

Kako kod kalibracije za iste točke u prostoru trebamo vrijednosti i u Kinect-ovom i robotovom koordinatnom sustavu loptica se prilikom kalibracije postavlja na vrh TCP-a i dovodi u svaku točku. Robot, nakon što mu je poslana naredba da dođe u prvu točku kalibracije, zajedno s lopticom dolazi u istu i tek nakon što se greška između vrijednosti zadane robotu i one poslane putem protokola računalu smanji na približno nulu, program za prepoznavanje započinje. Filtracija prilikom kalibracije služi kako bi se uklonili svi šumovi i točno odredio položaj kuglice u prostoru. Postavljanjem vrijednosti za najmanju i najveću dubinu te najmanji i najveći polumjer može se sa sigurnošću naći pozicija loptice u milimetrima. Za svaku kružnicu koja zadovoljava postavljene uvjete program zahtjeva od korisnika potvrdu kako bi bili sigurni da je točna kružnica prepoznata kao što se vidi na Slici 4.4.



Slika 4.4: Postupak kalibracije

Nakon što korisnik potvrdi točnost vrijednosti u milimetrima, robot nastavlja u drugu pa zatim i u treću točku prilikom čega je postupak isti za svaku sljedeću. Po završetku procesa vrijednosti dobivene za tri točke služe za računanje matrica transformacija, a robota se šalje u neku točku u prostoru koja definira prostor hvatanja (robot može izvesti hvatanje samo ako je točka hvatanja u polumjeru od dvadeset entimetara od početne pozicije robota). Kalibracija pomoću tri točke prikazana je na Slici 4.5.



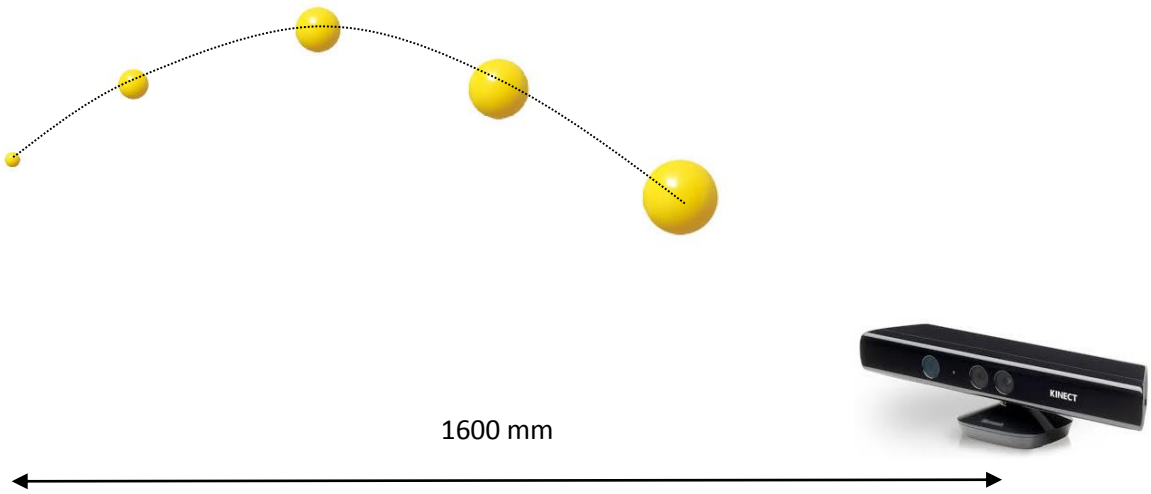
Slika 4.5: Kalibracija pomoću tri točke

Pošto se pretpostavlja da će bacanje biti izvedeno tako da se loptica kreće od Kinect-a prema robotu poželjno je da ravnina koju definiraju tri točke bude približno paralelna sa xy ravninom Kinect-ovog koordinatnog sustava. Također, zbog bolje točnosti predviđanja na većoj udaljenosti između Kinect-a i robota može doći do greške prilikom računanja dubine točaka kalibracije. Zato odabir položaja Kinect-a treba biti kompromis između točnosti kalibracije i točnosti predviđanja.

Eksperimenti su pokazali da je najbolji omjer za to kada se Kinect nalazi na udaljenosti između tisuću petsto i tisuću sedamsto milimetra od robota. Problem računanja udaljenosti središta kružnice od Kinect-a bi mogao biti riješen boljim odabirom tri točke kalibracije tako da su sve tri točke bliže Kinect-u ili odabirom veće loptice koja bi poslužila samo za kalibraciju. Ipak, ako odaberemo točke koje su bliže Kinect-u tada robot ulazi u svoje limite odnosno ne može doći u te točke zajedno sa lopticom zbog svoje strukture. Zato bi se taj problem u budućnosti mogao riješiti odabirom veće loptice koja bi služila ili samo za potrebe kalibracije ili i za cijeli proces hvatanja.

5. PREDVIĐANJE TOČKE HVATANJA

Kako bi uhvatio lopticu u letu, robot mora znati gdje će se loptica nalaziti nakon nekog vremena. Dubina, odnosno z koordinata hvatanja je fiksirana i određena eksperimentalno tako da robot ima dovoljno vremena za hvatanje, ali i da je prepoznavanje loptice u svakoj sličici što točnije. Povećavanjem dubine, smanjuje se točnost prepoznavanja zbog smanjenja polumjera loptice i udaljenosti od Kinect-a, ali se povećava broj točaka u prostoru što u konačnici znači i povećanje točnosti predviđanja. Predviđanje točke hvatanja znači da putanja kretanja loptice u prostoru mora biti poznata. Kako je gibanje loptice u prostoru opisano kosim hicem to znači da je potrebno poznavati barem tri točke kako bi sve jednadžbe gibanja bile jednoznačno određene. Kasnije se na temelju novih točaka radi korekcija parametara i za zadano vrijeme hvatanja računaju vrijednosti koordinata u prostoru.



Slika 5.1: Prikaz loptice u letu u odnosu na Kinect

U ovom radu nisu uzeti u obzir otpor zraka ni rotacija loptice u letu. Zbog toga umjesto korištenja kompleksnijih metoda za računanje predviđanja i estimaciju putanje kao što je Kalmanov filtar, prošireni Kalmanov filtar (Extended Kalman Filter) [4] ili neke od metoda strojnog učenja u radu je korištena jednostavnja metoda za interpolaciju krivulje polinomom, polinomijalna interpolacija kao u [8-10]. Ako znamo vrijednosti funkcije $f(x)$ na nekom skupu točaka tada možemo provući glatku krivulju kroz zadane točke i tako dobiti

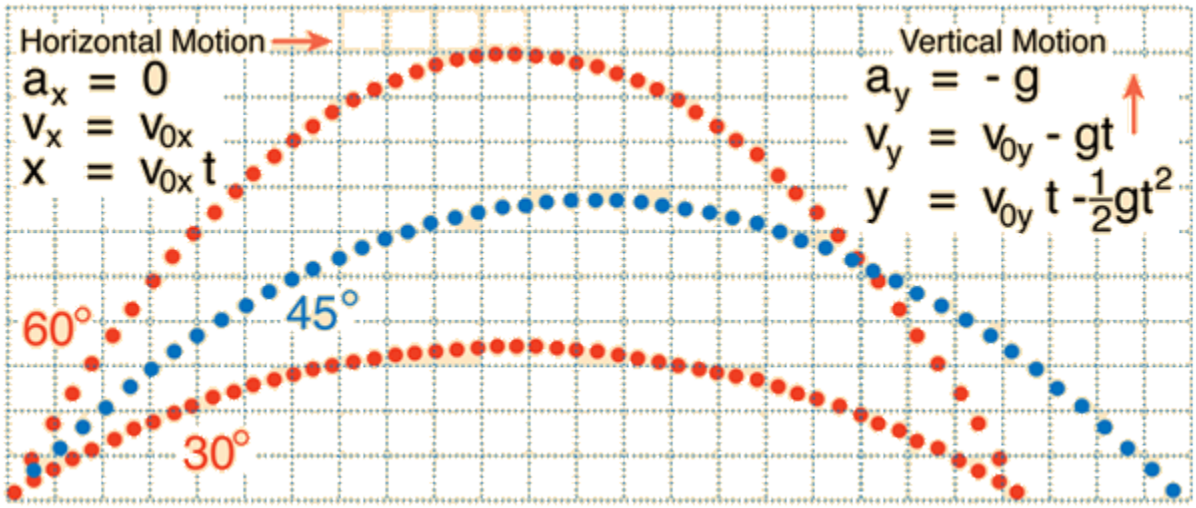
analitički izraz za računanje vrijednosti funkcije u bilo kojoj točki. Najjednostavnija interpolacija je linearna interpolacija kod koje se vrijednosti funkcije između dvije susjedne točke grafa prikazuju kao da leže na pravcu između te dvije točke. Tako je analitički izraz za računanje vrijednosti u bilo kojoj točki funkcije zapravo jednadžba pravca kroz dvije točke. Polinomijalna interpolacija je generalizacija linearne interpolacije. Kod polinomijalne interpolacije linearni interpolat zamjenjujemo polinomom višeg stupnja. Kako bi jednoznačno odredili polinom n-tog reda potreban je $n+1$ podatak odnosno $n+1$ točka interpolacije. Tako ako želimo jednoznačno odrediti polinom drugog reda (parabolu po kojoj se giba loptica) potrebno je imati najmanje tri točke interpolacije. Funkcija za polinomijalnu interpolaciju vrača vrijednosti koeficijenata polinoma u smislu najmanjih kvadrata. Iako jednostavna za korištenje ova metoda ima jedan veliki nedostatak, a to je dugo vrijeme računanja. Kako bi spriječili blokiranje programa i uštedjeli na vremenu algoritam za predikciju započinje prilikom pokretanja programa. Tada algoritam za neke tri ručno definirane točke računa prvu točku predviđanja koja služi samo pokretanju algoritma. Sve vrijednosti se brišu kod prve prepoznavanja loptice, a algoritam nastavlja s računanjem točaka predviđanja.

```
if (x.Count() >= 3)
{
pos_X = MathNet.Numerics.Fit.Polynomial(t, x, 2);
pX = MathNet.Numerics.Fit.Line(t, x);
pos_Y = MathNet.Numerics.Fit.Polynomial(t, y, 2);

coeffsX.Add(pX); //
coeffsY.Add(pos_Y);
t_list.Add(t);

p = MathNet.Numerics.Fit.Line(t, z);
impact_time = (zimpact - p.Item1) / p.Item2;

if (Double.IsInfinity(impact_time) == false)
       Console.Out.WriteLine("Impact time:" + (impact_time - t[0]).ToString() + "ms");

       //predictedPoint.X = pos_X[2] * Math.Pow(impact_time, 2) + pos_X[1] *
impact_time + pos_X[0];
       predictedPoint.X = impact_time * pX.Item2 + pX.Item1;
       predictedPoint.Y = pos_Y[2] * Math.Pow(impact_time, 2) + pos_Y[1] * impact_time
+ pos_Y[0];
    predictedPoint.Z = zimpact;
}
```

5.1. Kosi hitac u prostoru

Izbačaj tijela u prostor na koji djeluje sila teža zove se hitac. Hitac predstavlja složeno gibanje, a može biti horizontalni ili vertikalni (gibanje materijalne točke koja je izbačena vodoravno u polju sile teže), okomiti (gibanje materijalne točke koja je izbačena u polju sile teže okomito prema gore ili prema dolje) i kosi (gibanje materijalne točke koja je izbačena u polju sile teže pod kutom prema vodoravnoj ravnini). Vrsta hica je određena smjerom vektora početne brzine prema sili teži, a ako je otpor zraka zanemariv, putanja gibanja je parabola.

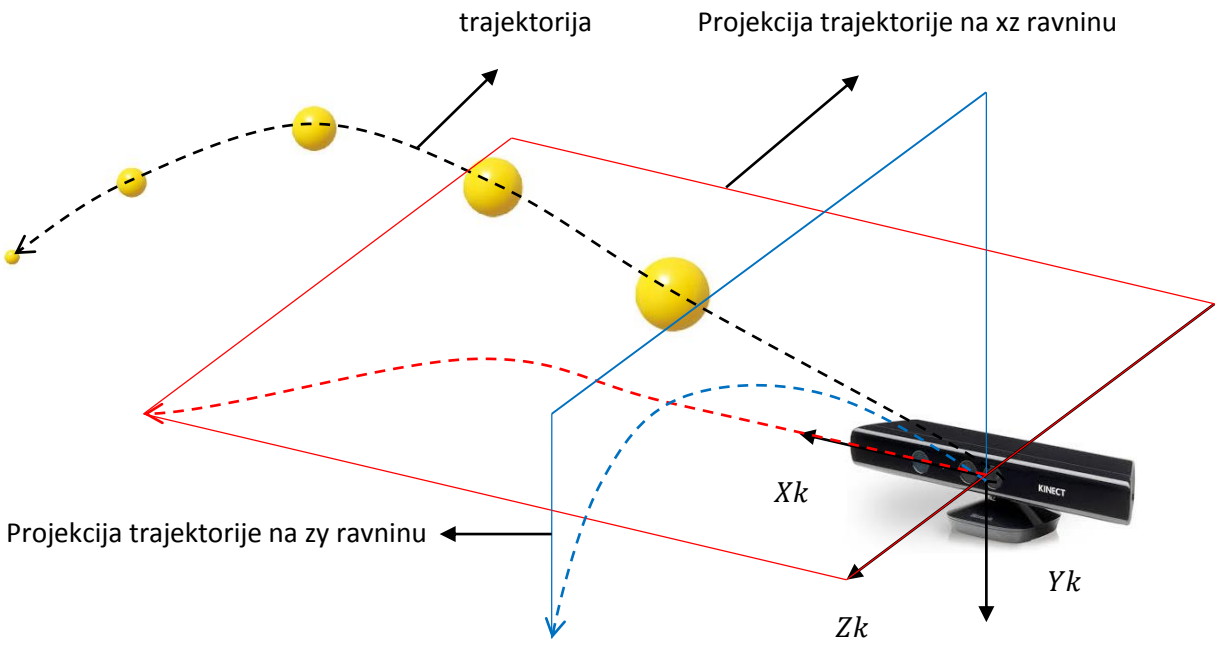
Kada vektor početne brzine izbačenog tijela zatvara oštri kut prema vodoravnoj ravnini nastaje složeno ili krivocrtno gibanje koje nazivamo kosi hitac. Putanja bačenog tijela ima oblik parabole s tjemenom na vrhu. Na izbačeno tijelo djeluje vektor kose početne brzine te ubrzanje zemljine sile teže. Svako tijelo bačeno početnom brzinom v_0, pod nekim kutem α (elevacijski kut) izvodi složeno gibanje, onakvo kao kod kosog hica. U 2D slučaju, početnu brzinu rastavljamo na okomitu i vodoravnu komponentu gdje vodoravna komponenta brzine određuje udaljenost koju tijelo pređe na tlu, dok okomita komponenta brzine određuje visinu na koju će tijelo dospjeti. Slična analogija se primjenjuje i u prostoru samo što početnu brzinu rastavljamo na tri brzine. Onu u smjeru osi x, y i z.



Slika 5.2: Jednadžbe i grafovi kosog hica

Horizontalna komponenta (u smjeru osi z) je konstantna tijekom cijelog gibanja jer akceleracija ne utječe na istu pa se tijelo u smjeru osi z giba jednoliko po pravcu. Uzimajući to u obzir za neki zadani z lako je moguće odrediti vrijeme hvatanja t. U vertikalnom smjeru tijelo se giba stalnom akceleracijom a za brzinu se uvrštava vertikalna komponenta početne brzine. Tako dobivamo tri jednadžbe od kojih su prve dvije drugog reda, a treća prvog reda.

Trajektorija bačene loptice modelirana je kao ona kosoga hica u prostoru. Kako se dubina mijenja linearno u ovisnosti o vremenu, lako se može odrediti vrijeme hvatanja od početka bacanja. Trajektorija u prostoru se projicira na dvije ravnine i opisuje dvjema parabolama drugog reda te se iz toga računaju x i y vrijednosti za definirano vrijeme hvatanja. Iako se vrijednost y računa po paraboli drugog reda, zbog činjenice da je Kinect okrenut prema robotu prilikom bacanja, x koordinata ostaje približno ista, dok se y koordinata mijenja po nekoj paraboli. Zbog toga se u radu i pretpostavlja da se x koordinata mijenja linearno.



Slika 5.3: Trajektorija loptice u prostoru i projekcija na dvije ravnine

5.2. Točka hvatanja

Kao što je prije rečeno, točka hvatanja predstavlja kompromis između točnosti predviđanja i brzine hvatanja. Polumjer kretanja robota je određen da bude dvadeset centimetara kako robot ne bi došao u limite ili pokušao izvesti neku kretnju koja u dinamičnoj radnoj okolini može biti opasna za čovjeka. Kako je svaka krivulja drugog reda određena sa tri točke, prva predviđanje se događa nakon tri prepoznavanja loptice. Za svaku sljedeću prepoznavanje provlače se po dvije parabole drugog reda i računa točka hvatanja. Ne treba napominjati da se povećanjem broja točaka povećava i točnost predviđanja. Ipak, za svaku točku hvatanja se pomoću matrica transformacija računaju koordinate u robotovom baznom sustavu i provjerava zadovoljava li ta točka uvjet da se nalazi unutar dvadeset centimetara od početnog robotovog položaja. Najčešće je taj uvjet zadovoljen tek za zadnjih par točaka hvatanja tako da prvih par predviđanje koje imaju najmanju točnost nisu niti uzete u obzir. Ispod je prikaz ispisa programa sa koordinatama prepoznatih točaka i predviđenom točkom hvatanja:

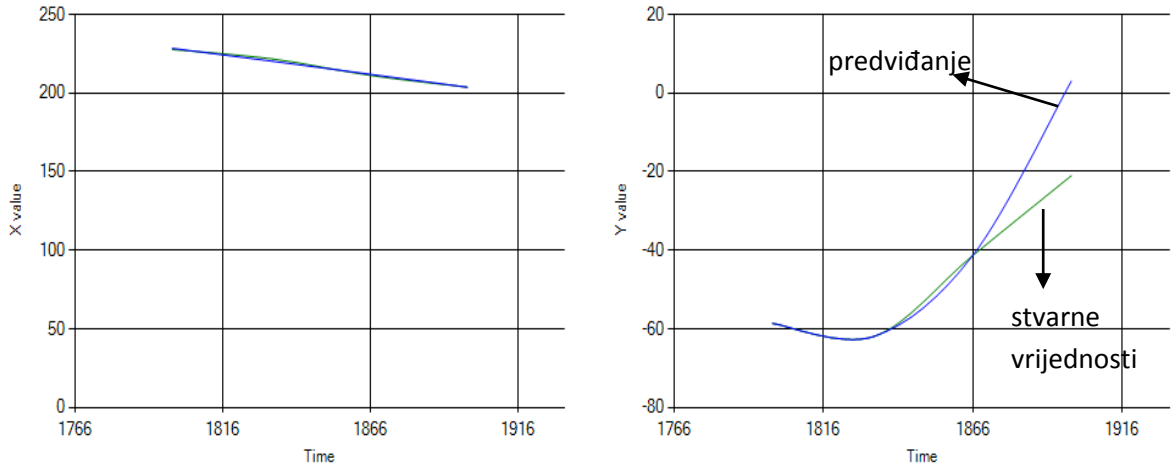
```
Detected point:-93,7407961649075;-128,080003321326;750,998208526493____Radius:24
Detected point:-95,7379369620294;-128,7365541384;766,998170359281____Radius:24
Detected point:-95,9875795616696;-130,457627237996;768,99816558838____Radius:24
Impact time:3321,00223211028ms
Predicted point:-205,955636000217;-5423,76704878341;1650
Detected point:-98,5373063187812;-131,283804773629;777,998088904686____Radius:24
Impact time:3979,58775712838ms
Predicted point:-290,938940793422;257,220385110958;1650
Detected point:-107,849740356858;-137,289484252553;804,997785974792____Radius:23
Impact time:4169,3742577058ms
Predicted point:-351,964020000796;-1344,71090306188;1650
Detected point:-119,060383806516;-147,81275240885;850,997446218847____Radius:22
Impact time:4273,52305896491ms
Predicted point:-347,120940170204;-555,27208997456;1650
Detected point:-120,171846199043;-144,363579755263;878,997481942576____Radius:22
Impact time:4530,15134151234ms
Predicted point:-305,549228793606;883,659523354097;1650
Detected point:-121,100830793798;-138,697349350756;906,997522525074____Radius:21
Impact time:4496,58833994225ms
Predicted point:-273,907396215902;1311,67108881211;1650
Detected point:-125,183917256427;-134,243067374136;945,997462056681____Radius:20
Impact time:4249,92143700796ms
Predicted point:-249,073368568013;1169,32403007263;1650
Detected point:-133,304958577732;-118,027397622594;994,997263409836____Radius:19
Impact time:3912,35256169263ms
Predicted point:-237,60279667298;1146,89344241015;1650
Detected point:-135,716505567078;-113,625159596875;1012,99721390368____Radius:18
Impact time:3790,80051433252ms
Predicted point:-235,305948383367;1118,36668140778;1650
Detected point:-140,193846983666;-94,9221814475755;1049,99713192229____Radius:18
Impact time:3514,52513225125ms
Predicted point:-231,000439949207;1061,69600461914;1650
```

5.3. Točnost

Točnost predviđanja neposredno ovisi o uspješnosti prepoznavanja loptice što znači ako vizijski sustav nije dobro odradio prepoznavanje i predviđanje će biti kriva. Također, povećanjem broja točaka povećava se točnost pa je zadnja predviđanje ujedno i najtočnija. Idealno, predviđanje bi trebala dati približno točn rezultat već nakon tri točke. To ostavlja dovoljno vremena robotu da dođe u točku hvatanja. Eksperimentom je utvđeno da je nakon tri prepoznavanja loptica otprilike na udaljenosti od jednog metra od Kinect-a. To znači da čak i ako je predviđanje točna, a dubina hvatanja je tek nešto veća od jednog metra robot neće imati dovoljno vremena stići u tu točku. Brzina robota se može povećati do neke granice no to za posljedicu ima velike trzaje što u konačnici nije dobro za robota. Prednost male dubine je u tome što je na manjim dubinama prepoznavanje loptice bolje (polumjer je veći).

Ako pak povećamo udaljenost Kinect-a od robota, odnosno postavimo dubinu hvatanja na otprilike dva metra, onda se točnost prepoznavanja smanjuje. Kinect već nakon otprilike jednog metra i šesto milimetara slabo može uočiti lopticu i razlikovati ju od šuma. Ipak, u ovom slučaju možemo povećati minimalni broj točaka za prvu predikciju.

Drugi problem je način gibanja loptice. Grafovi pokazuju da tjeme parabole često dolazi tek nakon tri prepoznavanja. Što znači će greška predviđanja nakon tri točke biti značajnije veća od one za, recimo četiri ili pet točaka. Međutim, čak i ako predikciju radimo nakon tri, četiri ili pet točaka dobivene vrijednosti za točku hvatanja često ne zadovoljavaju uvjet da su unutar dvadeset centimetara od početne pozicije robota. U većini slučajeva, taj uvjet je zadovoljen tek kod zadnje ili predzadnje predviđanja.



Slika 5.4: Usporedba predviđanja i stvarnih vrijednosti nakon bacanja

6. PROBLEMI I IZAZOVI

Uzevši u obzir sve prednosti i mane korištenih i razvijenih algoritama te sam način izvedbe, provedeni eksperimenti u različitim fazama procesa pokazali su ne samo moguće probleme već i otvorili prostor za nadogradnju korištenih algoritama i poboljšanje tehničkog sustava. Uspješnost rješenja zadanog problema ovisila je kako i o odabiru elemenata tehničkog sustava tako i o razvijenom programu. Problemi i izazovi u radu navedeni su u sljedeća tri ulomka.

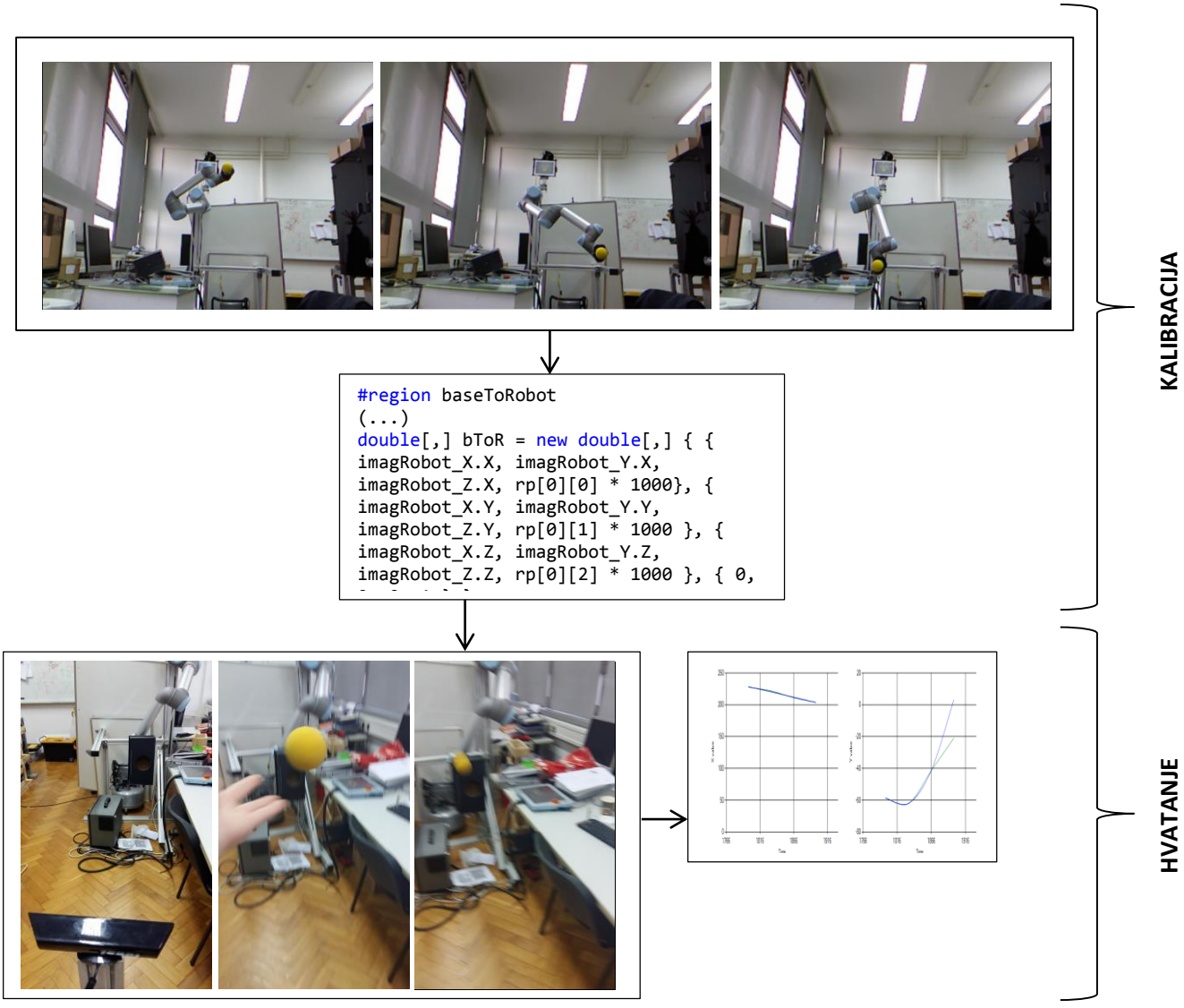
Da uhvati lopticu u letu robot mora imati sposobnost percepcije i praćenja objekta, predviđanja putanje te planiranja hvatanja. Trodimenzionalnost ili stereo vid je sposobnost čula vida da predmete prepoznaje u tri dimenzije, dakle i po dubini. Hvatanje loptice u letu je trodimenzionalni problem stoga je poznavanje dubine nužno za riješenje istoga. Jedna RGB kamera ne može dati informaciju o dubini, ali korištenje dvije kamere ili jedne RGB kamere i jedne dubinske kamere kao u slučaju Kinect-a, može. Također, uspješnost prepoznavanja loptice, a posljedično i predviđanja, ovisi o brzini kamere koja je za uređaje kao što su Kinect ili Bumblebee relativno mala. Neki od čimbenika pri odabiru senzora uključuju jednostavnost izvedbe, cijenu i lakoću programiranja. Unatoč sporoj kameri od svega 30 slika u sekundi te nepreciznost dubinske kamere, jeftina cijena, jednostavnost korištenja i programiranja presudili su pri odabiru Kinect-a kao senzora. U budućnosti bi se sustav mogao nadograditi postavljanjem jedne *„eye-in-hand“* kamere na robotu. Korištenje takve kamere u kombinaciji sa Kinect-om povećalo bi uspješnost prepoznavanja loptice u letu. Nepreciznost dubinske kamere uvjetuje i karakteristike loptice. Zbog činjenice da je domet dubinske kamere Kinect-a od 800 do 4000 milimetara loptica treba biti dovoljno velika da omogući prepoznavanje na udaljenostima većim od jednog metra i manjim od dva metra što je predviđena udaljenost za hvatanje između robota i bacača. Minimalna vrijednost dometa uvjetuje i postavljanje senzora prema robotu umjesto prema bacaču. Loptica također mora biti dovoljno mala, specifične boje te od mekanog materijala kako bacanje ne bi oštetilo robota ili ugrozilo čovjekovu sigurnost. Kako je dobacivanje i hvatanje loptice u ovom slučaju samo sebi svrha, poželjno je da robotska ruka što više nalikuje ljudskoj pa je zbog svoje težine, malih dimenzija i lakoće upravljanja odabrana UR5 robotska ruka.

Korištenje Kinect-a kao senzora daje na izbor dva modela prepoznavanja loptice u prostoru. Jedan je korištenje samo dubinske kamere, a drugi kombinacija RGB i dubinske kamere. Iako bi se korištenjem samo dubinske kamere izbjegli svi problemi vezani uz dimenzije i izgled loptice te vanjske čimbenike kao što su osvjetljenje i okolina robota, šum dubinske kamere i uvjet na nepomičnost okoline predstavljaju veći problem od prije navedenih. Za prepoznavanje objekata na temelju boje i oblika razvijeni su mnogi algoritmi jednostavni za korištenje pa se upotreba istih i razvijanje novih, sličnih algoritama pokazalo dovoljno dobro za problem prepoznavanja loptice. Eksperimentalno se pokazalo da je segmentacija žute boje, odnosno korištenje žute loptice, i pronalaženje kontura na slici dalo najbolje rezultate za problem prepoznavanja u prostoru. Najveći doprinos je postavljanje uvjeta na dimenzije kontura i udaljenost središta kružnice u ovisnosti o vremenu čime je razvijen algoritam za prepoznavanja i praćenja objekta u prostoru. Izlaz čini skup točaka u prostoru koji služi kao ulaz algoritmu za predviđanja točke hvatanja.

Jedan od problema u kojem algoritmi strojnog učenja nalaze uspješnu primjenu jest upravo problem predviđanja (eng. *prediction*). Većina algoritama koristi se za predviđanje budućih vrijednosti na temelju podataka iz prošlosti. U slučaju hvatanja loptice buduće vrijednosti su potencijalne točke hvatanja, a podaci iz prošlosti su točke koje predstavljaju poziciju loptice u prostoru u određenom trenutku. Predviđanje je vrsta učenja s nadzorom koje koristi regresiju kao pristup modeliranja relacija između jedne ili više varijabli označene sa Y, te jedne ili više varijabli označene sa X. U situacijama kao što je hvatanje loptice u letu, u kojima funkcionalan odnos između zavisne varijable y i nezavisne varijable x ne može biti adekvatno aproksimiran linearnim odnosom, odnosno kada je odgovor (nezavisna varijabla) nelinearan, koristi se polinomna regresija. Polinomna regresija je prikladna za ovaj slučaj jer se gibanje lotpce u prostoru može opisati parabolom drugog reda koja se projicira na dvije ravnine. Tako se pronalaženje koeficijenata polinoma svodi na riješavanje tri jednadžbe sa tri nepoznanice. Uzevši u obzir da je gravitacija jedina sila na tijelo i da je ravnina u kojoj se loptica giba okomita na xy ravninu Kinect-a pojednostavljivanja kao što su zamjena druge i treće jednadžbe linearnim jednadžbama daju dovoljno dobre rezultate, a smanjuju vrijeme računanja. Sa povećanjem broja točaka u prošlosti algoritam daje bolji rezultat pa je potrebno izgraditi takav tehnički postav da se već nakon par točaka može vjerno opisati dinamika gibanja. U budućnosti bi bilo zanimljivo ispitati upotrebu drugih algoritama strojnog učenja za regresiju i predviđanje ili pak korištenje Kalmanovog filtra. Kalmanov filtar, kao optimalni

procjenitelj (estimator) i prediktor nepoznate veličine pronašao je veliku primjenu u upravljanju sustavima, navigaciji, praćenju i predviđanju putanje objekta.

Kada je razvijen cijeli program rezultati su pokazali da je najveći problem kašnjenje sustava. Jedno od riješenja je ili povećavanje udaljenosti između Kinect-a i robota ili povećavanje brzine gibanja robota. Zbog problema vezanih uz prepoznavanje, udaljavanje Kinect-a od robota značilo bi manju točnost predviđanja i hvatanja. Zato je odabrano drugo riješenje gdje je udaljenost ostala ista, ali vrijeme za koje robot mora doći iz početne točke u točku hvatanja je 0.12 sekundi. Trzaji su u tom slučaju neminovni, ali za tako male udaljenosti ne od posebnog značaja. Robot je također ograničen na gibanje u polumjeru od dvadeset centimetara od početnog položaja kako bi se povećala kontrola nad sustavnom smanjila mogućnost pogreške. Na Slici 6.1. prikazano je završno testiranje i faze procesa:



Slika 6.1: Završno testiranje

7. ZAKLJUČAK

Ponašanje u nepredvidivim situacijama i dinamičkoj okolini predstavlja problem za čovjeka pa tako i za robota. Roboti danas mogu obavljati puno zadataka, ali većina primjena se odnosi na relativno statično okruženje i predvidive situacije. Za brzu i adekvatnu reakciju u dinamičkoj okolini robot mora imati sposobnost percepcije, predviđanja i planiranja. Jedan od pristupa riješenju je pretpostavka da će se objekt u letu nastaviti ponašati kako se i ponašao u prošlosti, omogućavajući robotu da predvidi što će se dogoditi. Drugi se odnosi na konstantno promatranje i kontinuirano ažuriranje krajnjeg rezultata, u ovom slučaju točke hvatanja u prostoru.

U ovom radu robot je opremljen Microsoft Kinect-om koji se koristi za dobivanje informacija o položaju loptice u prostoru u određenom trenutku. Kinect-ova kamera (RGB i dubinska) ima brzinu od trideset sličica u sekundi i za svaku sličicu se traže koordinate središta kružnice koja omeđuje prepoznatu lopticu. Prepoznavanje se temelji na segmentaciji žute boje nakon prebacivanja u HSV prostor boja i izdvajanju rubova koji definiraju kružnicu. U svakoj sličici filtriranje se radi na principu prošlih vrijednosti uzimajući u obzir da se radijus s udaljenošću od Kinect-a smanjuje, a dubina povećava. Dubina se mjeri za središte kružnice i mijenja linearno kroz vrijeme. Podaci iz prošlosti služe za predviđanje točke hvatanja uzimajući u obzir kinematiku kosog hica. Nakon najmanje tri prepoznate točke moguće je polinomskom regresijom dobiti tri jednadžbe koje definiraju krivulju koja prolazi kroz te točke u prostoru. Kako je udaljenost od Kinect-a zadana i iznosi 1650mm, lako se izračunaju x i y koordinate. Tri vrijednosti je potrebno iz Kinect-ovog koordinatnog sustava prebaciti u robotov. Za to se koriste matrice transformacije izračunate nakon postupka kalibracije. Matrice transformacije daju koordinate točke hvatanja u robotovom koordinatnom sustavu. Točka hvatanja mora biti u radijusu ud dvadeset centimetara od početne pozicije robata, a ako točka zadovolji taj uvjet robotu se putem TCP protokola šalje naredba da se pomakne u istu. Poruka se šalje na robotov real-time poslužitelj koji se najčešće koristi za brzo primanje stanja robota.

Najveći izazv predstavljalo je prepoznavanje loptice. Za to su korišteni i razvijeni prikladni algoritmi. Jedini zadatak robota je pomaknuti kraj svoje ruke u točku hvatanja kada je ona izračunata i zadovoljava uvjete koji garantiraju sigurnost čovjeka i okoline. Razvijeni

program je upotrebljen na UR5 robotu i pokazalo se da uz korištenje relativno jeftinih i jednostavnih komponenata tehničkog sustava problem može riješiti, ali ostavljajući puno prostora za unaprijeđenje i nadogradnju.

Iako je prepoznavanje po boji i obliku najčešće korištena metoda, u budućnosti bi se mogla koristiti samo dubinska kamera za dobivanje informacija o položaju u prostoru. Također, umjesto statičnog uređaja može se koristiti kamera na robotu, samostalno ili u kombinaciji sa Kinect-om. Hvatanje objekata relativno jednostavne geometrije moglo bi se otežati hvatanjem objekata kompleksne geometrije koji su statički i dinamički nestabilni te korištenje robotske šake umjesto robotske ruke sa šest stupnjeva slobode.

LITERATURA

[1] C. Borst, M. Fischer, S. Haidacher, H. Liu, G. Hirzinger, DLR hand II: experiments and experiences with an anthropomorphic hand, in: IEEE International Conference on Robotics and Automation, Taipei, 2003, pp. 702–707.

[2] V. Lippiello, F. Ruggiero, B. Siciliano, L. Villani, Visual grasp planning for unknown objects using a multifingered robotic hand, IEEE/ASME Transactions on Mechatronics 18 (3) (2013) 1050–1059.

[3] Q. He, C. Hu, W. Liu, N. Wei, M. Meng, L. Liu, C. Wang, Simple 3-D point reconstruction methods with accuracy prediction for multiocular system, IEEE/ASME Transactions on Mechatronics 18 (1) (2013) 366–375.

[4] U. Frese, B. Bauml, S. Haidacher, G. Schreiber, I. Schaefer, M. Hahnle, G. Hirzinger, Off-the-shelf vision for a robotic ball catcher, in: IEEE/RSJ International Conference on Intelligent Robots and Systems, Maui, HI, 2001, pp. 1623–1629.

[5] R. O. Duda and P. E. Hart, "Use of the Hough transformation to detect lines and curves in pictures," Comm. of the ACM, vol. 15, no. January, pp. 11–15, 1972.

[6] C. Kimme, D. Ballard, and J. Sklansky, "Finding circles by an array of accumulators," Comm. of the ACM, vol. 18, no. 2, pp. 120–122, 1975.

[7] H. Yuen, J. Princen, J. Dlingworth, and J. W. Kittler, "A comparative study of hough transform methods for circle finding," in Proc. of the Alvey Vision Conf., 1989.

[8] J. Bouguet, Camera calibration toolbox for Matlab, Tech. Rep., California Institute of Technology, 2010. http://www.vision.caltech.edu/bouguetj/.

[9] B. Siciliano, L. Sciavicco, L. Villani, G. Oriolo, Robotics: Modelling, Planning and Control, Springer, London, UK, 2008.

[10] G. Batz, A. Yaqub, H. Wu, K. K ¨ uhnlenz, D. Wollherr, and M. Buss, "Dynamic manipulation: Nonprehensile ball catching," in Proc. of the IEEE Mediterranean Conf. on Control and Automation, 2010.

[11] Curić I., Bronzin T., Kako upravljati sadržajem putem Kinect uređaja, CASE25, Zagreb, 2012.

[12] N. Kirić, Primjena 3D vizijskog sustava za praćenje objekata robotom u realnom vremenu, Diplomski rad, Fakultet strojarstva i brodogradnje, 2015.

[13] M. Smolec, Bušenje nehomogenih materijala pomoću robota, Diplomski rad, Fakultet strojarstva i brodogradnje, 2015.

[14] „Računalni vid“, Wikipedia: Slobodna enciklopedija, sa internetske stranice: https://hr.wikipedia.org/wiki/Ra%C4%8Dunalni_vid

[15] B. Lukovac, Sazrijevanje računalnog vida: Automatsko pronalaženje korespondencija, Fakultet elektrotehnike i računarstva, Zagreb, travanj 2008.

[16] S. Lujić, Pretraživanje slikovnih baza temeljeno na značajkama boje, Fakultet elektrotehnike i računarstva, Zagreb, lipanj 2008.

[17] „Boja i atributi boje“, sa internetske stranice Sveučilišta u Zagrebu, http://racunala.ttf.unizg.hr/files/Boja_i_atributi_boje.pdf, rujan 2015.g.

[18] „Jednadžba gibanja“, Wikipedia: Slobodna enciklopedija, sa internetske stranice: https://hr.wikipedia.org/wiki/Jednad%C5%BEba_gibanja

[19]

PRILOZI

A. Programski kod

B. CD-R disk

A. Programski kod

A.1. Main form.cs

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Net.Sockets;
using System.Windows.Forms;
using Microsoft.Kinect;
using System.Windows;
using System.IO;
using System.Globalization;
using System.Windows.Media;
using System.Windows.Media.Imaging;
using System.Windows.Media.Media3D;
using System.Drawing.Imaging;
using System.Runtime.InteropServices;
using Emgu.CV;
using Emgu.CV.Structure;
using Emgu.CV.UI;
using Emgu.CV.CvEnum;
using Emgu.CV.VideoSurveillance;
using System.Diagnostics;
using System.Windows.Forms.DataVisualization.Charting;
using MathNet.Numerics;
using MathNet.Numerics.LinearAlgebra.Double;
using MathNet.Numerics.LinearAlgebra.Factorization;

namespace RobotBallCatching
{
    public partial class MainForm : Form
    {
        #region vars
        KinectSensor kinectSensor = null;
        ColorImageFormat ColorFormat = ColorImageFormat.RgbResolution640x480Fps30;
        DepthImageFormat DepthFormat = DepthImageFormat.Resolution640x480Fps30;
        List<double> times = new List<double>();
        Vector3D pointVector = new Vector3D();
        Vector3D lastDetected = new Vector3D();
        PointF oldCont = new PointF();
        public double pX;
        public double pY;
        List<double> pXs = new List<double>();
        List<double> pYs = new List<double>();

        double time = 0;
        int b = 5;
        double dist = 0;

        int dMin = 950;
        int dMax = 750;
        int rMin = 20;
        int rMax = 30;
        public static int avd = 0;
        public int rad = 0;
```

```
        double[] radii = new double[] { };
        List<double> Radii = new List<double>();
        double[] depths = new double[] { };
        List<double> Depths = new List<double>();
        public static MathNet.Numerics.Tuple<double, double> p;

        public static bool cal = false;
        public static bool calibration = false;
        public static bool started = false;
        public static List<double[]> rp = new List<double[]>{new double[] { -0.71040,
0.06573, 0.93, 0.9615, -1.3938, 1.1038 },
            new double[] { -0.14943, 0.28783, 0.93, 0.9615, -1.3938, 1.1038 },
            new double[] { -0.50691, -0.27003, 0.93, 0.9615, -1.3938, 1.1038 }};
        public static List<double> robotPosition;
        List<string> robotPoints = new List<string>(new string[] { "movej(p[-0.71040,
0.06573, 0.93, 0.9615, -1.3938, 1.1038])\n",
            "movej(p[-0.14943, 0.28783, 0.93, 0.9615, -1.3938, 1.1038])\n",
            "movej(p[-0.50691, -0.27003, 0.93, 0.9615, -1.3938, 1.1038])\n" });
        public static List<Vector3D> kinectPoints = new List<Vector3D>();
        public static int i = 0;
        public static List<int> i_s = new List<int>();
        int robpoint = 0;
        public static Vector3D imagKinect_X;
        public static Vector3D imagKinect_Y;
        public static Vector3D imagKinect_Z;
        public static Vector3D imagRobot_X;
        public static Vector3D imagRobot_Y;
        public static Vector3D imagRobot_Z;
        public static MathNet.Numerics.LinearAlgebra.Matrix<double> kinectToBase;
        public static MathNet.Numerics.LinearAlgebra.Matrix<double> baseToRobot;
        bool pbf = false;
        #endregion vars

        public MainForm()
        {
            InitializeComponent();
            ConectToRobot();
            PopulateAvailableSensors();
        }

        private void ConectToRobot()
        {
            try
            {
                Client.client = new TcpClient("192.168.0.9", 30001);
                Client_RT.client_RT = new TcpClient("192.168.0.9", 30003);
            }
            catch (ArgumentNullException e)
            {
                Console.WriteLine("ArgumentNullException: {0}", e);
            }
            catch (SocketException e)
            {
                Console.WriteLine("SocketException: {0}", e);
            }
        }

        private void PopulateAvailableSensors()
        {
            foreach (KinectSensor sensor in KinectSensor.KinectSensors)
```

```
            {
                kinectSensor = sensor;
                SetupSensorVideoInput();
            }
        }

        private void SetupSensorVideoInput()
        {
            if (kinectSensor != null)
            {
                kinectSensor.ColorStream.Enable(ColorFormat);
                kinectSensor.DepthStream.Enable(DepthFormat);

                EventHandler<AllFramesReadyEventArgs> handler = new
EventHandler<AllFramesReadyEventArgs>(FilterImage);

                kinectSensor.AllFramesReady += handler;
                kinectSensor.Start();
            }
            else
                DeActivateSensor();
        }

        private void FilterImage(object sender, AllFramesReadyEventArgs e)
        {
            using (DepthImageFrame depthFrame = e.OpenDepthImageFrame())
            {
                using (ColorImageFrame colorFrame = e.OpenColorImageFrame())
                {
                    if (colorFrame != null && depthFrame != null)
                    {
                        DepthImagePixel[] depthPixels = new
DepthImagePixel[depthFrame.PixelDataLength];
                        depthFrame.CopyDepthImagePixelDataTo(depthPixels);

                        Bitmap bmpFrame = ColorImageFrameToBitmap(colorFrame);
                        Image<Bgr, Byte> colorImage = new Image<Bgr, Byte>(bmpFrame);

                        using (Image<Hsv, byte> hsv = colorImage.Convert<Hsv, byte>())
                        {
                            Image<Gray, byte>[] channels = hsv.Split();

                            try
                            {
                                CvInvoke.cvInRangeS(channels[0], new
Gray(20).MCvScalar, new Gray(32).MCvScalar, channels[0]);
                                CvInvoke.cvErode(channels[0], channels[0],
(IntPtr)null, 2);
                                channels[0]._SmoothGaussian(9);

                                if (calibration == true)
                                {
                                    #region calibration

                                    if (i_s.Count() == 0)
                                    {
                                        Client.Send(robotPoints[i]);
                                        i_s.Add(i);
                                    }
```

```
                                    else if (i_s.Count() != 0 && i != i_s[i_s.Count()
- 1])
                                    {
                                        Client.Send(robotPoints[i]);
                                        i_s.Add(i);
                                    }

                                    robotPosition = Client_RT.GetPosition();


                                    if (robotPosition != null &&
Math.Abs(robotPosition[0] - rp[i][0]) < 0.0002 && i <= 2)
                                    {
                                        calibration = false;
                                        PointF calCenter = new PointF();
                                        CircleF calCircle = new CircleF();

                                        using (MemStorage stor = new MemStorage())
                                        {
                                            for (var contours =
channels[0].FindContours(CHAIN_APPROX_METHOD.CV_CHAIN_APPROX_SIMPLE,
RETR_TYPE.CV_RETR_EXTERNAL); contours != null; contours = contours.HNext)
                                            {
                                                if (robpoint == 0)
                                                {
                                                    int calRad =
(contours.BoundingRectangle.Height + contours.BoundingRectangle.Width) / 4;
                                                    calCenter =
contours.GetMinAreaRect().center;
                                                    int calDepth =
depthPixels[(int)calCenter.X + (int)calCenter.Y * depthFrame.Width].Depth;

                                                    if (calRad <= 15 && calRad >= 8 &&
calDepth < 2000 && calDepth > 1380 && calDepth != 0)
                                                    {
                                                        calCircle.Center = calCenter;
                                                        calCircle.Radius = calRad;

                                                        Vector3D realCoord =
CurveFitting.ConvertToReal((int)calDepth, calCenter.Y, calCenter.X);

                                                        if
(System.Windows.Forms.MessageBox.Show("Ball found!\n" + "Coordinates:" +
realCoord.X.ToString() + "," + realCoord.Y.ToString() + "," + realCoord.Z.ToString() +
"\nPlease confirm the position:", "Confirm", MessageBoxButtons.YesNo,
MessageBoxIcon.Question) == System.Windows.Forms.DialogResult.Yes)
                                                        {

kinectPoints.Add(realCoord);
                                                            robpoint++;
                                                            i++;
                                                        }
                                                    }
                                                }
                                            }
                                        }

                                        if (kinectPoints.Count() == 3)
                                        {
```

```
System.Windows.Forms.MessageBox.Show("Calibrated");
                                        FindTransformation(kinectPoints);
                                        cal = true;
                                        calibration = false;
                                        kinectPoints.Clear();
                                        Client.Send("movej(p[-0.68811, -0.22103,
0.65948, 0.9615, -1.3938, 1.1038])\n");
                                        Client_RT.client_RT.Close();
                                    }
                                    else
                                    {
                                        cal = false;
                                        calibration = true;
                                        robpoint = 0;
                                    }
                                    robotPosition.Clear();
                                }
                                else
                                {
                                    calibration = true;
                                    robotPosition.Clear();
                                }

                                #endregion calibration
                            }

                            if (started == true)
                            {
                                if (pbf == false)
                                {
                                    for (int i = 0; i < 3; i++)
                                    {
                                        times.Add(time);
                                        Vector3D pointBefore = new Vector3D(320.0,
240.0, 500.0);

                                        CurveFitting.Fit(pointBefore,
times[times.Count() - 1]);
                                        time = time + ((1 / 30.0) * 1000);
                                        lastDetected = pointBefore;
                                        pbf = true;
                                    }
                                }

                                #region contours
                                PointF center = new PointF();
                                CircleF circle = new CircleF();

                                using (MemStorage stor = new MemStorage())
                                {
                                    for (var contours =
channels[0].FindContours(CHAIN_APPROX_METHOD.CV_CHAIN_APPROX_SIMPLE,
RETR_TYPE.CV_RETR_EXTERNAL); contours != null; contours = contours.HNext)
                                    {
                                        rad = (contours.BoundingRectangle.Height +
contours.BoundingRectangle.Width) / 4;
                                        center = contours.GetMinAreaRect().center;
                                        avd = depthPixels[(int)center.X +
(int)center.Y * depthFrame.Width].Depth;
```

```
                                        #region Filtering
                                        if (rad <= rMax && rad >= rMin)
                                        {
                                            dist = Math.Sqrt(Math.Pow((oldCont.X -
center.X), 2) + Math.Pow((oldCont.Y - center.Y), 2));
                                            if (Radii.Count() >= 2 && dist < 30)
                                            {
                                                if (avd != 0 && avd < dMin && avd
>= dMax)
                                                {
                                                    rMax = rad + 3;
                                                    if (rad >= 10)
                                                        rMin = rad - 5;
                                                    else
                                                        rMin = 10;

                                                    dMin =
(int)CurveFitting.zimpact;
                                                    dMax = avd;

                                                    circle.Center = center;
                                                    circle.Radius = rad;
                                                    oldCont = center;

                                                    pointVector =
CurveFitting.ConvertToReal(avd, (float)center.Y, (float)center.X);

                                                    Radii.Add(rad);
                                                    Depths.Add(avd);
                                                }
                                                else if (avd == 0 || avd > dMin ||
avd <= dMax)
                                                {
                                                    rMax = rad + 3;
                                                    if (rad >= 8)
                                                        rMin = rad - 5;
                                                    else
                                                        rMin = 8;

                                                    radii = Radii.ToArray();
                                                    depths = Depths.ToArray();

                                                    p =
MathNet.Numerics.Fit.Line(depths, radii);
                                                    avd = (int)((rad - p.Item1) /
p.Item2);

                                                    dMin =
(int)CurveFitting.zimpact;
                                                    dMax = avd;

                                                    circle.Center = center;
                                                    circle.Radius = rad;
                                                    oldCont = center;

                                                    pointVector =
CurveFitting.ConvertToReal(avd, (float)center.Y, (float)center.X);

                                                    Radii.Add(rad);
```

```
                                                Depths.Add(avd);
                                            }
                                        }
                                        if (Radii.Count() == 1 && dist < 30)
                                        {
                                            if (avd != 0 && avd < dMin && avd
>= dMax)
                                            {
                                                rMax = rad + 3;
                                                if (rad >= 8)
                                                    rMin = rad - 5;
                                                else
                                                    rMin = 8;

                                                dMin =
(int)CurveFitting.zimpact;
                                                dMax = avd;

                                                circle.Center = center;
                                                circle.Radius = rad;
                                                oldCont = center;

                                                pointVector =
CurveFitting.ConvertToReal(avd, (float)center.Y, (float)center.X);

                                                Radii.Add(rad);
                                                Depths.Add(avd);
                                            }
                                        }
                                        if (Radii.Count() == 0)
                                        {
                                            if (avd != 0 && avd < dMin && avd
>= dMax)
                                            {
                                                rMax = rad + 3;
                                                if (rad >= 8)
                                                    rMin = rad - 5;
                                                else
                                                    rMin = 8;

                                                dMin =
(int)CurveFitting.zimpact;
                                                dMax = avd;

                                                circle.Center = center;
                                                circle.Radius = rad;
                                                oldCont = center;

                                                pointVector =
CurveFitting.ConvertToReal(avd, (float)center.Y, (float)center.X);

                                                Radii.Add(rad);
                                                Depths.Add(avd);
                                            }
                                        }
                                    }
                                    #endregion Filtering
                                }
                            }
```

```
                                    colorImage.Draw(circle, new
Bgr(System.Drawing.Color.Blue), 2);
                                    PredictTrajectory(pointVector);
                                    pointVector = new Vector3D(0, 0, 0);

                                    #endregion contours
                                }
                                picVideoDisplay.Image = colorImage.ToBitmap();
                            }

                            finally
                            {
                                channels[0].Dispose();
                                channels[1].Dispose();
                                channels[2].Dispose();
                            }
                        }
                    }
                }
            }
        }

        private void FindTransformation(List<Vector3D> kinectPoints)
        {
            #region kinectToBase
            double dk1 = Math.Sqrt(Math.Pow((kinectPoints[2].X - kinectPoints[0].X),
2) + Math.Pow((kinectPoints[2].Y - kinectPoints[0].Y), 2) +
Math.Pow((kinectPoints[2].Z - kinectPoints[0].Z), 2));

            imagKinect_X.X = (kinectPoints[2].X - kinectPoints[0].X) / dk1;
            imagKinect_X.Y = (kinectPoints[2].Y - kinectPoints[0].Y) / dk1;
            imagKinect_X.Z = (kinectPoints[2].Z - kinectPoints[0].Z) / dk1;

            imagKinect_Y.X = (kinectPoints[1].X - kinectPoints[0].X);
            imagKinect_Y.Y = (kinectPoints[1].Y - kinectPoints[0].Y);
            imagKinect_Y.Z = (kinectPoints[1].Z - kinectPoints[0].Z);

            imagKinect_Z = Vector3D.CrossProduct(imagKinect_X, imagKinect_Y);
            double dk2 = Math.Sqrt(Math.Pow(imagKinect_Z.X, 2) +
Math.Pow(imagKinect_Z.Y, 2) + Math.Pow(imagKinect_Z.Z, 2));
            imagKinect_Z.X = imagKinect_Z.X / dk2;
            imagKinect_Z.Y = imagKinect_Z.Y / dk2;
            imagKinect_Z.Z = imagKinect_Z.Z / dk2;

            imagKinect_Y = Vector3D.CrossProduct(imagKinect_Z, imagKinect_X);

            double[,] kToB = new double[,] { { imagKinect_X.X, imagKinect_Y.X,
imagKinect_Z.X, kinectPoints[0].X }, { imagKinect_X.Y, imagKinect_Y.Y, imagKinect_Z.Y,
kinectPoints[0].Y }, { imagKinect_X.Z, imagKinect_Y.Z, imagKinect_Z.Z,
kinectPoints[0].Z }, { 0, 0, 0, 1 } };
            kinectToBase =
MathNet.Numerics.LinearAlgebra.Double.Matrix.Build.DenseOfArray(kToB);
            #endregion kinectToBase

            #region baseToRobot
            double dr1 = Math.Sqrt(Math.Pow((rp[2][0] - rp[0][0]) * 1000, 2) +
Math.Pow((rp[2][1] - rp[0][1]) * 1000, 2) + Math.Pow((rp[2][2] - rp[0][2]) * 1000,
2));

            imagRobot_X.X = ((rp[2][0] - rp[0][0]) * 1000) / dr1;
```

```
            imagRobot_X.Y = (rp[2][1] - rp[0][1]) * 1000 / dr1;
            imagRobot_X.Z = (rp[2][2] - rp[0][2]) * 1000 / dr1;

            imagRobot_Y.X = (rp[1][0] - rp[0][0]) * 1000;
            imagRobot_Y.Y = (rp[1][1] - rp[0][1]) * 1000;
            imagRobot_Y.Z = (rp[1][2] - rp[0][2]) * 1000;

            imagRobot_Z = Vector3D.CrossProduct(imagRobot_X, imagRobot_Y);
            double dr2 = Math.Sqrt(Math.Pow(imagRobot_Z.X, 2) +
Math.Pow(imagRobot_Z.Y, 2) + Math.Pow(imagRobot_Z.Z, 2));
            imagRobot_Z.X = imagRobot_Z.X / dr2;
            imagRobot_Z.Y = imagRobot_Z.Y / dr2;
            imagRobot_Z.Z = imagRobot_Z.Z / dr2;

            imagRobot_Y = Vector3D.CrossProduct(imagRobot_Z, imagRobot_X);

            double[,] bToR = new double[,] { { imagRobot_X.X, imagRobot_Y.X,
imagRobot_Z.X, rp[0][0] * 1000}, { imagRobot_X.Y, imagRobot_Y.Y, imagRobot_Z.Y,
rp[0][1] * 1000 }, { imagRobot_X.Z, imagRobot_Y.Z, imagRobot_Z.Z, rp[0][2] * 1000 }, {
0, 0, 0, 1 } };
            baseToRobot =
MathNet.Numerics.LinearAlgebra.Double.Matrix.Build.DenseOfArray(bToR);
            #endregion baseToRobot
        }

        private void PredictTrajectory(Vector3D point)
        {
            if (point.X != 0 && point.Y != 0 && point.Z != 0)
            {
                if (point.Z < CurveFitting.zimpact)
                {
                    times.Add(time);
                    chart1.Series["Series1"].Points.Add(new
DataPoint((int)times[times.Count() - 1], point.X));
                    chart2.Series["Series1"].Points.Add(new
DataPoint((int)times[times.Count() - 1], point.Y));
                    Console.Out.WriteLine("Detected point:" + point.ToString() +
"____Radius:" + Radii[Radii.Count() - 1]);
                    CurveFitting.Fit(point, times[times.Count() - 1]);
                    lastDetected = point;
                }
            }
            time = time + ((1 / 30.0) * 1000);

            if ((time - times[times.Count() - 1]) > 3000.0 && lastDetected.X != 320)
            {
                StopAlgorithm();
                b = 0;
            }
        }

        private void StopAlgorithm()
        {
            if (b != 0)
            {
                if (CurveFitting.coeffsX.Count() > 1)
                {
                    for (int i = 0; i <
CurveFitting.t_list[CurveFitting.t_list.Count() - 1].Count(); i++)
                    {
```

```
                        double t = CurveFitting.t_list[CurveFitting.t_list.Count() -
1][i];
                        //pX = CurveFitting.coeffsX[1][2] * Math.Pow(t, 2) +
CurveFitting.coeffsX[1][1] * t + CurveFitting.coeffsX[1][0];
                        pX = CurveFitting.coeffsX[1].Item2 * t +
CurveFitting.coeffsX[1].Item1;
                        pY = CurveFitting.coeffsY[1][2] * Math.Pow(t, 2) +
CurveFitting.coeffsY[1][1] * t + CurveFitting.coeffsY[1][0];
                        pXs.Add(pX);
                        pYs.Add(pY);
                    }
                    for (int i = 0; i <
CurveFitting.t_list[CurveFitting.t_list.Count() - 1].Count(); i++)
                    {
                        double t = CurveFitting.t_list[CurveFitting.t_list.Count() -
1][i];
                        chart1.Series["Series2"].Points.Add(new DataPoint((int)t,
pXs[i]));
                        chart2.Series["Series2"].Points.Add(new DataPoint((int)t,
pYs[i]));
                    }
                }
                kinectSensor.Stop();
                Client_RT.client_RT.Close();
            }
        }

        private static Bitmap ColorImageFrameToBitmap(ColorImageFrame colorFrame)
        {
            byte[] pixelBuffer = new byte[colorFrame.PixelDataLength];
            colorFrame.CopyPixelDataTo(pixelBuffer);

            Bitmap bitmapFrame = new Bitmap(colorFrame.Width, colorFrame.Height,
System.Drawing.Imaging.PixelFormat.Format32bppRgb);
            BitmapData bitmapData = bitmapFrame.LockBits(new Rectangle(0, 0,
colorFrame.Width, colorFrame.Height), ImageLockMode.WriteOnly,
bitmapFrame.PixelFormat);

            IntPtr intPointer = bitmapData.Scan0;
            Marshal.Copy(pixelBuffer, 0, intPointer, colorFrame.PixelDataLength);

            bitmapFrame.UnlockBits(bitmapData);

            return bitmapFrame;

        }

        private void DeActivateSensor()
        {
            if (kinectSensor != null)
            {
                kinectSensor.Stop();
                kinectSensor.AllFramesReady -= new
EventHandler<AllFramesReadyEventArgs>(FilterImage);
                kinectSensor.Dispose();
            }
        }

        protected override void OnClosing(CancelEventArgs e)
        {
```

```
            base.OnClosing(e);

            DeActivateSensor();
        }

        private static void OnClickCalibrate(object sender, EventArgs e)
        {
            calibration = true;
        }

        private static void OnStart(object sender, EventArgs e)
        {
            //if (cal == false)
            //    System.Windows.Forms.MessageBox.Show("You have to calibrate
first!");
            //if (cal == true)
                started = true;
        }
    }
}
```

A.2. CurveFitting.cs

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;
using Microsoft.Kinect;
using System.Windows;
using System.IO;
using System.Globalization;
using System.Windows.Media;
using System.Windows.Media.Imaging;
using System.Windows.Media.Media3D;
using System.Drawing.Imaging;
using System.Runtime.InteropServices;
using Emgu.CV;
using Emgu.CV.Structure;
using Emgu.CV.UI;
using Emgu.CV.CvEnum;
using Emgu.CV.VideoSurveillance;
using System.Diagnostics;
using System.Windows.Forms.DataVisualization.Charting;
using MathNet.Numerics;
using MathNet.Numerics.LinearAlgebra.Double;
using MathNet.Numerics.LinearAlgebra.Factorization;

namespace RobotBallCatching
{
    class CurveFitting
    {
        #region vars
        public static double zimpact = 1650.0;
        public static double impact_time;
        public static int k = 0;
        public static double[] pos_X = new double[] { };
```

```
        public static double[] pos_Y = new double[] { };
        public static List<double> x_val = new List<double>();
        public static List<double> y_val = new List<double>();
        public static List<double> z_val = new List<double>();
        public static List<double> t_val = new List<double>();
        public static MathNet.Numerics.Tuple<double, double> p;
        public static MathNet.Numerics.Tuple<double, double> pX;
        public static List<MathNet.Numerics.Tuple<double, double>> coeffsX = new
List<MathNet.Numerics.Tuple<double, double>>();
        public static List<double[]> coeffsY = new List<double[]>();
        public static List<double[]> t_list = new List<double[]>();
        public static MathNet.Numerics.LinearAlgebra.Vector<double> bCP;
        public static MathNet.Numerics.LinearAlgebra.Vector<double> rCP;
        #endregion vars

        internal static void Fit(Vector3D point,double time)
        {
            Vector3D predictedPoint = new Vector3D();

            x_val.Add((double)point.X);
            y_val.Add((double)point.Y);
            z_val.Add((double)point.Z);
            t_val.Add(time);

            if (k == 0)
            {
                if (point.X != 320.0 && point.Y != 240.0 && point.Z != 500.0)
                {
                    x_val.RemoveAll(item => item == 320);
                    y_val.RemoveAll(item => item == 240);
                    z_val.RemoveAll(item => item == 500);
                    t_val.Clear();
                    t_val.Add(time);
                    k = k + 1;
                }
            }
            double[] z = z_val.ToArray();
            double[] x = x_val.ToArray();
            double[] y = y_val.ToArray();
            double[] t = t_val.ToArray();

            if (x.Count() >= 3)
            {
                pos_X = MathNet.Numerics.Fit.Polynomial(t, x, 2);
                pX = MathNet.Numerics.Fit.Line(t, x);
                pos_Y = MathNet.Numerics.Fit.Polynomial(t, y, 2);

                coeffsX.Add(pX); //
                coeffsY.Add(pos_Y);
                t_list.Add(t);

                p = MathNet.Numerics.Fit.Line(t, z);
                impact_time = (zimpact - p.Item1) / p.Item2;

                if (Double.IsInfinity(impact_time) == false)
                    Console.Out.WriteLine("Impact time:" + (impact_time -
t[0]).ToString() + "ms");

                //predictedPoint.X = pos_X[2] * Math.Pow(impact_time, 2) + pos_X[1] *
impact_time + pos_X[0];
```

```
                predictedPoint.X = impact_time * pX.Item2 + pX.Item1;
                predictedPoint.Y = pos_Y[2] * Math.Pow(impact_time, 2) + pos_Y[1] *
impact_time + pos_Y[0];
                predictedPoint.Z = zimpact;

                if (Double.IsNaN(predictedPoint.X) == false)
                {
                    double[] kinectCatchPoint = new double[] { predictedPoint.X,
predictedPoint.Y, predictedPoint.Z, 1 };
                    MathNet.Numerics.LinearAlgebra.Vector<double> kCP =
MathNet.Numerics.LinearAlgebra.Vector<double>.Build.DenseOfArray(kinectCatchPoint);
                    MathNet.Numerics.LinearAlgebra.Matrix<double> kTbInverted =
MainForm.kinectToBase.Inverse();
                    bCP = kTbInverted.Multiply(kCP);

                    rCP = MainForm.baseToRobot.Multiply(bCP);
                    double[] robotCatchPoint = rCP.ToArray<double>();
                    string XValue = robotCatchPoint[0].ToString().Replace(",", ".");
                    string YValue = robotCatchPoint[1].ToString().Replace(",", ".");
                    string ZValue = (robotCatchPoint[2] -
50.0).ToString().Replace(",", ".");

                    Console.Out.WriteLine("Predicted point:" +
predictedPoint.ToString());
                    Console.Out.WriteLine("Robot catch point:" + XValue + ", " +
YValue + ", " + ZValue);
                    for (int i = 0; i < 2; i++)
                    {
                        robotCatchPoint[i] = robotCatchPoint[i] / 1000;
                    }
                    robotCatchPoint[2] = (robotCatchPoint[2] - 50.0) / 1000;
                    CheckRobotSpace(robotCatchPoint);
                }
            }
        }

        private static void CheckRobotSpace(double[] robotCatchPoint)
        {
            string XValue = robotCatchPoint[0].ToString().Replace(",", ".");
            string YValue = robotCatchPoint[1].ToString().Replace(",", ".");
            string ZValue = robotCatchPoint[2].ToString().Replace(",", ".");

            double dist = Math.Sqrt(Math.Pow(-0.68811 - robotCatchPoint[0], 2) +
Math.Pow(-0.22103 - robotCatchPoint[1], 2));
            if(dist < 0.25)
                Client.Send("movej(p[" + XValue + ", " + YValue + ", " + ZValue + ",
0.9615, -1.3938, 1.1038], t = 0.12)\n");
        }

        public static Vector3D ConvertToReal(int zv, float yv, float xv)
        {
            double fh= 0;
            double fv = 0;
            int h = 640;
            int v = 480;
            double yw;
            double xw;
            Vector3D realWorld = new Vector3D();
            fh = h / (2 * Math.Tan(62 * Math.PI / 360));
            fv = v / (2 * Math.Tan(48.6 * Math.PI /360));
```

```
            double alpha = Math.Atan((320 - xv) / fh);
            double beta = Math.Atan((yv - 240) / fv);
            double zwh = Math.Cos(alpha * Math.PI / 180) * zv;
            double zwv = Math.Cos(beta * Math.PI / 180) * zv;

            xw = Math.Sin(alpha) * zv;
            yw = Math.Sin(beta) * zv;

            realWorld.X = xw;
            realWorld.Y = yw;
            realWorld.Z = zwh;

            return realWorld;
        }
    }
}
```

A.3. Client.cs

```
using System;
using System.IO;
using System.Net;
using System.Text;
using System.Net.Sockets;
using System.Collections.Generic;

namespace RobotBallCatching
{
    class Client
    {
        public static TcpClient client;

        public static void Send(string mes)
        {
            // Translate the passed message into ASCII and store it as a Byte array.
            //mes = mes.Insert(mes.Length, ".0").Replace(",", ".").Replace(";", ",");
            //string message = "movej(p[" + mes + ",0.0,0.0,0.0])\n";
            string message = mes;
            //string message = "\n";
            Byte[] data = System.Text.Encoding.ASCII.GetBytes(message);

            // Get a client stream for reading and writing.
            //  Stream stream = client.GetStream();
            if (client != null)
            {
                NetworkStream stream = client.GetStream();

                // Send the message to the connected TcpPoslužitelj.
                stream.Write(data, 0, data.Length);

                Console.WriteLine("Sent: {0}", message);
            }
        }
    }
}
```

A.4. Client_RT.cs

```
using System;
using System.IO;
```

```
using System.Net;
using System.Text;
using System.Net.Sockets;
using System.Collections.Generic;

namespace RobotBallCatching
{
    class Client_RT
    {
        public static List<double> robotPos = new List<double>();
        public static TcpClient client_RT;
        public static List<double> GetPosition()
        {
            if (client_RT != null)
            {
                NetworkStream stream = client_RT.GetStream();
                Byte[] data = new Byte[8192];
                Int32 bytes = stream.Read(data, 0, data.Length);

                double val = 0;

                for (int j = 1; j <= 6; j++)
                {
                    string value = "";
                    for (int i = 444 + (j - 1) * 8; i < 444 + j * 8; i++)
                    {
                        string bstr = Convert.ToString(data[i], 2);
                        if (bstr.Length == 7)
                            bstr = "0" + bstr;
                        else if (bstr.Length == 6)
                            bstr = "00" + bstr;
                        else if (bstr.Length == 5)
                            bstr = "000" + bstr;
                        else if (bstr.Length == 4)
                            bstr = "0000" + bstr;
                        else if (bstr.Length == 3)
                            bstr = "00000" + bstr;
                        else if (bstr.Length == 2)
                            bstr = "000000" + bstr;
                        else if (bstr.Length == 1)
                            bstr = "0000000" + bstr;
                        value += bstr;
                    }
                    if (j < 4)
                        val = BitConverter.Int64BitsToDouble(Convert.ToInt64(value,
2));
                    else if (j >= 4)
                        val = BitConverter.Int64BitsToDouble(Convert.ToInt64(value,
2));
                    robotPos.Add(val);
                }
                //Close everything.
                //stream.Close();
                //client_RT.Close();
            }
            return robotPos;
        }
    }
}
```