TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond
Tarkvarateaduse instituut

Autonoomse sõiduki raja järgimine vabavaralise tarkvaraga Autoware

Magistritöö

Priit Trink
163100IAPM

Juhendaja
Juhan-Peep Ernits, PhD

Tallinn 2018

Autorideklaratsioon

Kinnitan, et olen koostanud antud lõputöö iseseisvalt ning seda ei ole kellegi teise poolt varem kaitsmisele esitatud. Kõik töö koostamisel kasutatud teiste autorite tööd, olulised seisukohad, kirjandusallikatest ja mujalt pärinevad andmed on töös viidatud.

Priit Trink

14. mai 2018. a.

.....
(Allkiri)

Annotatsioon

TTÜ sajanda aastapäeva puhul tehakse TTÜ ja AS Silberauto koostööga isesõitev auto. Sõiduki tarkvaraks on valitud vabavaraline tarkvara Autoware, mis on suuteline 3D kaardil autonoomselt sõidukit juhtima. Antud töö eesmärk on katsetada ühte autonoomse sõiduki alamprobleemidest - raja järgimine - ning seda võimalikult sõltumatult teistest alamprobleemidest.

Tulemuseks on lahendus raja järgimise katsetamine Autoware'iga, kasutades kolme põhikomponenti: 3D lidarit, sõidukit ning tarkvara Autoware. Ühenduse Autoware'i ja robotplatvormi vahel loome kahe sõidukiga. Katsetusi viiakse läbi ühel sõidukil, kuna ainult üks oli töö kirjutamise ajal katsetamiseks piisavalt valmis. Sellega seoses analüüsime ka raja järgimise sooritust erinevate parameetritega.

Lõputöö on kirjutatud eesti keeles ning sisaldab teksti 79 leheküljel, 6 peatükki, 15 joonist, 2 tabelit.

Abstract

In the honour of TTÜ's 100th anniversary, TTÜ and AS Silberauto are creating a self-driving car. An open source software called Autoware, which can drive a vehicle autonomously on a 3D map, is used. The purpose of this thesis is to test one of the autonomous vehicle subproblems - path following - with having minimal dependencies of the other subproblems.

As a result, we will implement a solution to test path following using three major components: a 3D lidar, a real vehicle and the software Autoware. A connection between a vehicle and Autoware is implemented on two vehicles. We use one of them for testing, because only one is ready for testing at the time this thesis is written. Regarding to that, we analyse the performance of the path following using different input parameters.

The thesis is in Estonian and contains 79 pages of text, 6 chapters, 15 figures, 2 tables.

Lühendite ja mõistete sõnastik

ROS Roboti Operatsioonisüsteem

USB Universaalne järjestiksiin

UDP Kasutajadatagrammi protokoll

CSV Komaga eraldatud väärtused

Sisukord

Joonised

Tabelid

Sissejuhatus

Tallinna Tehnikaülikool koostöös Silberautoga valmistab ülikooli 100. aastapäevaks isesõitvat autot. Autonoomsed sõidukid on järjest rohkem saanud ühiskonnas suuremat tähelepanu. Teema pakub huvi tarkvaratootjatele, kes tegelevad mistahes sõiduki juhtimisega, ning ka autotootjaid, kes realiseerivad automaatse sõitmise funktsioone ja abisid.

Kuna autonoomse sõiduki realiseerimine on tervikuna keeruline, jaotatakse see mitmeks väiksemaks probleemiks. Alamprobleem, mida antud töös käsitletakse, on raja järgimine. [3] Raja järgimeks on Autoware'is implementeeritud algoritm, mille nimi on Pure Pursuit. Algoritm eeldab, et meil on olemas masina asukoht ja rada, mööda mida on plaanis sõita. Kui eelmainitud sisendid on antud, arvutatakse välja auto esirataste asendid, et sõiduk liiguks mööda rada.

Käesoleva töö eesmärk on katsetada raja järgimist vabavaralise tarkvaraga Autoware ning seda teistest autonoomse sõiduki probleemidest võimalikult sõltumatult. Samuti on eesmärgiks luua ühendus raja järgimise algoritmi ning päriselulise sõiduki vahel. Hindame ka raja järgimise sooritusvõimet graafikute abil.

Magistritöö eripäraks on asjaolu, et luuakse lahendus raja järgimise testimiseks sõltudes ainult sõiduki lokaliseerimisest. Samuti luuakse tarkvara, et ühendada Autoware meile olemasolevate platvormidega. Kuna Autoware on kesise dokumentatsiooniga vabavara, siis antud töö aitab süvendada arusaamu, milleks Autoware võimeline on.

TTÜ 100. aastapäev on 2018. aasta sügisel. Selle tõttu on eesmärk saada lahendus valmis sellele eelnevaks suveks.

Probleemiga tegeletakse TTÜ Iseauto projekti raames. Katsetusi tehakse päriselulisel platvormil (elektriline ATV Uku) ning simulatsioone ja arendust kodu- või laboriarvutites.

Probleemi on vaja lahendada, et luua ning testida autonoomset sõidukit. Samuti valideerime me hetkel olemasolevat vabavaralist tarkvara ning loome uut dokumentatsiooni,

et tarkvarast paremini aru saada.

Raja järgimise katsetamiseks kasutatakse avatud lähtekoodiga tarkvara Autoware. Autoware on autonoomse sõiduki tarkvara, mis on loodud linnas liiklemiseks. Tarkvara on suuteline tegema punktikaarte, võtma vastu otsuseid vastavalt andurite näitudele ning sealjuures juhtima sõidukit. Raja järgimine Pure Pursuit algoritmiga on väike osa antud tarkvarast.

Autoware on ülesehitatud ROS-ile (Robot Operating System), mis on disainitud olema võimalikult modulaarne, et erinevate tarkvaratükkide testimine ning välja vahetamine oleks võrdlemisi lihtne. Modulaarsuse tõttu on hüpoteesiks see, et raja järgimist saab sellega edukalt katsetada.

Raja järgimise katsetamiseks kasutatakse Autoware'i nii simulatsioonikeskkonnas Gazebo kui ka päriselulistel platvormil Uku (tulevikus ka reaalse autoga). Selleks on vaja luua vastavad liidesed, et sõiduk oleks juhitav raja järgimise algoritmi väljunditega.

Raja järgimise põhjalikumaks uurimiseks üritame mõõta kiirendussensoriga sõidu sujuvust, et hinnata mugavust reisijale.

Päriselulisi katseid viiakse läbi TTÜ linnakus samal marsruudil, mis on eesmärgiks läbida Iseauto projekti raames loodud sõidukil.

Tulemustest saame järeldada, kuivõrd kõlblik on Autoware'i raja järgimise implementatsioon erinevatel platvormidel ning kui hästi saab tarkvara hakkama robotiga, mis on alles arengujärgus.

1. Taust ja seotud töö

1.1 Kontekst

Autonoomse sõiduki loomine on terviklikult keerukas ja mahukas probleem. Selle tõttu tasub probleemi jupitada mitmeks väiksemaks tükiks:

- lokalisatsioon;
- teekonna planeerimine;
- liikumise planeerimine;
- raja läbimine;
- objektide tuvastamine;
- objektide jälgimine.

Rajaläbimine on probleem, mis on käesolevas töös täpsema käsitlemise all. Probleem eeldab esmalt lokalisatsiooni (sõiduki positsiooni määramise) lahendust. Kui on teada sõiduki asukoht, planeeritakse sellele kaardi peal teekond. Edasi planeeritakse sõiduki liikumist - selles etapis võetakse arvesse objektituvastusest tulevat informatsiooni. Vastavalt objektidele ja kaardile genereeritakse rada, mida tuleb sõidukil läbida. Avatud lähtekoodiga tarkvara Autoware pakub raja läbimise lahendamiseks Pure Pursuit algoritmi [5].

Autoware'i kohta on olemas artikkel [5], mis annab ülevaate, millised on autonoomse sõiduki alamprobleemid ja kuidas neid Autoware'is lahendatakse. On olemas ka töö [7] selle kohta, milliseid edasiarendusi on tehtud raja järgimises, et Pure Pursuiti sooritusvõime oleks parem. Tasuta dokumentatsioon tarkvara kasutamise kohta on puudulik: Autoware'i Githubi projekt pakub välja inglise keeles juhendi [8] - mis seletab, mida erinevad Autoware'i kasutajaliidese elemendid teevad - ning 5 harjutust [6] erinevate prob-

leemide lahendamiseks Autoware'is. Leidub ka artikkel [4], mis võrdleb erinevaid raja järgimise algoritme ning toob välja nende tugevused ja nõrkused.

1.2 Autoware

Kuna autonoomsed autod on saamas aina enam populaarsemaks, on selleks välja töötatud avatud tarkvaraline platvorm nimega Autoware. [5] Platvorm on sobilik nii autonoomse sõiduki loomise õppimiseks kui ka uute tehnoloogiate ja algoritmide katsetamiseks. Tarkvara põhineb ROS-il (Robot Operating System) ning katsetamiste hõlbustamiseks on loodud sellele ka kasutajaliides. Autoware'is on autonoomse sõiduki probleemid jaotatud just nii nagu on kirjeldatud Peatükis 1.1.

Lokaliseerimiseks kasutatakse 3D lidarit, mille andmed sobitatakse olemasoleva punktikaardiga kasutades NDT algoritmi. Lidarit kasutatakse ka objektide kauguse määramiseks, kui kaameraga on tuvastatud objekt (nt inimene või teine auto). [5]

Teekond luuakse missiooni ning liikumise planeerimise abil. Teekonna loomisel kasutatakse mitmeid sõiduridu, et oleks võimalik ümber põigata seisvatest objektidest, mis asuvad sõidureal. Autoware'i artikkel mainib, et selle jaoks on loodud ainult põhifunktsionaalsus ning keerulisemate olukordade lahendamiseks on vaja teha edasiarendust. [5]

Kui teekonna planeerimine on lahendatud, antakse teekond edasi raja järgimise algoritmile. Raja järgimiseks kasutatakse Pure Pursuit algoritmi ning sellele on tehtud ka mõned edasiarendused. Edasiarendustest tuleb rohkem juttu Peatükis 1.4.3.

Autoware'il on olemas tasuta ingliskeelne ja jaapanikeelne dokumentatsioon [8], mis kirjeldab tarkvara kasutajaliidest ning põhifunktsioone.

1.3 Robotplatvorm Uku

Uku on üks platvormidest, mis on välja pakutud Iseauto projekti raames autonoomse sõitmise katsetamiseks. Kuna antud töö kirjutamise hetkel oli see ainus platvorm, mis oli katsetamiseks valmis, tehakse raja järgimise proove just sellel masinal. Iseauto projekti hilisemas faasis hakatakse katseid tegema ka elektriautol Mitsubishi i-Mievil.

Magistritöö [1] räägib, et Uku platvorm oligi algselt mõeldud erinevate algoritmide

Joonis 1.1: [1] Robotplatvorm Uku.

ja liikumisviiside uurimiseks. Platvormi, millest esimesed versioonid olid ehitatud tudengite poolt õppetöö raames, hakati arendama aastal 2007. Aastal 2010 hakati robotit arendama uutel alustel, mille tulemusel saadi Uku piisavasse valmidusastmesse, et seda saada katsetada reaalsetes oludes. Roboti juhtimine töötas Windowsi operatsioonisüsteemil, mis tuli Iseauto projekti raames ehitada ümber, et juhtimismoodulid oleksid võimelised võtma käske vastu ROS-ilt, mis paraku töötab ainult Ubuntu operatsioonisüsteemil. Joonisel 1.1 on näha, milline Uku välja näeb.

1.4 Pure Pursuit

Pure Pursuit on algoritm, mida Autoware kasutab raja järgimise probleemi lahendamiseks. Antud alampeatükk kirjeldab täpsemalt, kuidas algoritm töötab, millised on parameetrid ning kuidas on Autoware'is algoritmi edasi arendatud.

1.4.1 Ajalugu

1992. aasta teadustöö Pure Pursuit algoritmi loomisest räägib, et algoritmi ei loodud algselt mitte raja järgimiseks, vaid roboti rajale tagasi juhtimiseks. Esimesi katsetusi algoritmiga tehti 1980. aastate alguses 6 rattalise roboti Terragatoriga, mis manööverdab oma külgmisi rattaid libistades (erinevalt käesolevas töös kasutatavatest sõidukitest, mis on neljarattalised ja keeravad esirattaid). Hiljem, kui hakati tegelema roboti juhtimisega

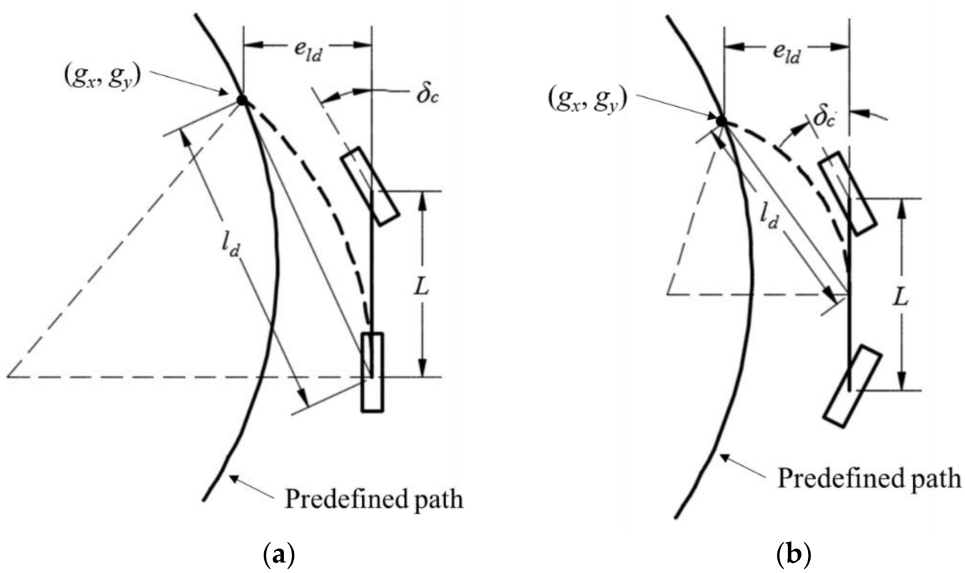
mööda kindlat trajektoori, katsetati mitmeid erinevaid algoritme. Katsetuste tulemused näitasid, et Pure Pursuit on kõige robustsem ja usaldusväärsem testitud variantidest. Kuna Pure Pursuiti kood oli juba muudel eesmärkidel valmis tehtud, siis otsustati, et antud lahendust mugandatakse raja järgimiseks. Töö autor mainib, nii tema kui ka projekti kaasliikmed, olid Pure Pursuiti võimetest positiivselt üllatunud - algoritm toimis isegi ebaloogiliste ja vigaste parameetritega. [3]

1.4.2 Teooria

Pure Pursuit on algoritm, mille eesmärk on juhtida sõiduk mingisugusest algpunktist kindlaks määratud sihtpunkti. Selleks, et seda saavutada, arvutatakse välja kurv, mille järgi seatakse rataste pöördenurk. Kuna sõiduki asukoht on pidevalt muutuv, tuleb rataste pöördenurka muuta nii tihti kui võimalik. Selle tõttu arvutatakse eelmainitud kurvi iga kord, kui saadakse infot sõiduki positsiooni kohta. [3] Pure Pursuit algoritmi ühte iteratsiooni kirjeldatakse nõnda:

1. määratakse kindlaks sõiduki asukoht;
2. leitakse lähim raja punkt;
3. leitakse sihtpunkt, kuhu sõidetakse;
4. sihtpunkt viiakse vastavusse sõiduki koordinaatidega;
5. arvutatakse kurv, mille järgi seatakse paika rataste pöördenurk;
6. uuendatakse auto asukohta.

Siin kohal tasub mainida, et sõiduki asukoht ja rada on algoritmi sisendid, mida võetakse arvesse kurvi arvutuste tegemisel. Algoritmi ülesanne on leida vastavalt nendele sisenditele hetkeline sobiv sihtpunkt ning sellele vastav rataste pöördenurk.



Joonis 1.2: [2] Pure Pursuit algoritm. **(a)** Ackermanni juhtimine (esirataste pööramine) **(b)** esi- ja tagarataste pööramine. L - telgede vahe, l_d - vaateulatus, δ_e - pöördenurk, (g_x, g_y) - sihtpunkt.

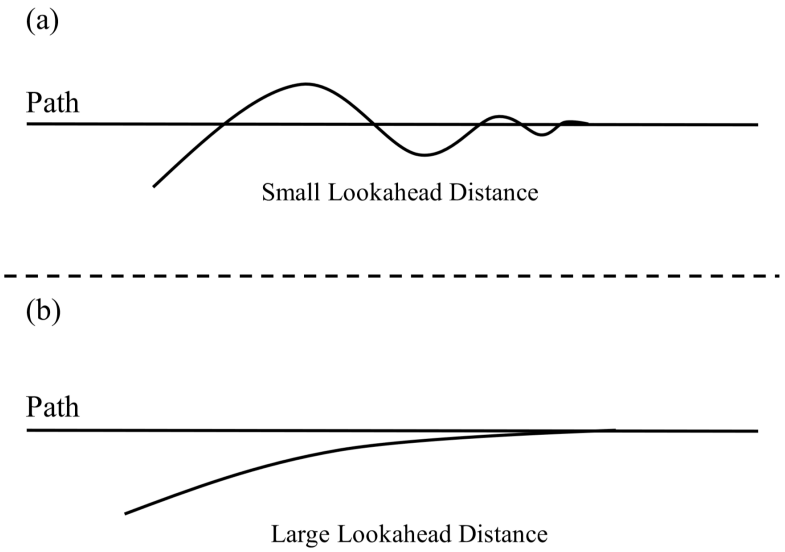
Kuidas leitakse sihtpunkt? Selleks kasutab algoritm parameetrit, mida nimetame vaateulatuseks. Kui on teada rada, mida tuleb järgida, ning sõiduki asukoht, valitakse raja peal välja punkt, mis asub sõidukist 1 vaateulatuse kaugusel. Kui selline punkt leitakse, siis punktist saab algoritmi hetkeline sihtpunkt.

Kui sihtpunkt on olemas, tuleb leida rataste pöördenurk. Pöördenurka saab Pure Pursuitiga arvutada nii Ackermanni juhtimist (esirataste pööramist) kui ka esi- ja tagarataste pööramist kasutatavatel sõidukitel. Ackermanni juhtimise korral tuleb viimase arvutamiseks leida selline ringjoon, mis läbib auto tagatelje keskpunti ning eelnevalt kindlaks määratud sihtpunkti. Ringjoone keskpunkt peab jääma sõiduki tagateljega samale joonele (vt Joonis 1.2). Kui ringjoon on leitud, tuleb antud ringi keskpunktist tõmmata sirgjoon sõiduki esiratta juurde ning pöördenurgaks saab selline nurk, mis oleks viimase sirgjoonega risti. Sellise arvutuse tegemiseks on meil vaja teada sõiduki telgede vahet L.

Vaateulatuse parameeter

Pure Pursuit algoritmi töö suutlikus sõltub suuresti parameetrite valikust. Kõige mõjukam parameeter antud algoritmi puhul on vaateulatus. Kui viimane valida liiga väike, võib sõiduk hakata sõitma sirgel rajal slaalomit. Kui valida liiga suur, siis hakkab sõiduk kurve lõikama (vt Joonis 1.3). Kuna mõlemad juhud on soovimatud, siis tuleb see

parameeter valida nii, et mainitud vigu oleks minimaalselt. [3]



Joonis 1.3: [3] Vaateulatuse parameetri mõju sõiduki trajektoorile. **(a)** väike vaateulatus **(b)** suur vaateulatus.

Kui leida mõni sobiv vaateulatus, ei pruugi see töötada erinevate kiiruste korral. Suurema kiirusega sõidame tavaliselt sirgetel radadel - vaateulatus võiks sellisel juhul olla suur, kuna slaalom sellisel kiirusel oleks ohtlik nii reisijatele kui ka kaasliiklejatele. Lisaks, sirgel rajal ei ole meil kurve, mida lõigata. Samas väiksema kiirusega me sõidame kohtades, mis on kurvilised ja nõuavad juhi poolt suuremat täpsust. Sellisel juhul tuleks omakorda minimeerida ka kurvide lõikamist - seega vaateulatus peaks olema võimalikult väike. Seega erinevatel kiirustel oleks vaja kasutada erinevaid vaateulatusi. [4] Levinud viis probleemi lahendamiseks on panna vaateulatus suhestuma sõiduki kiirusega. [9] Seda lahendust implementeerib ka töös kasutatav tarkvara Autoware.

1.4.3 Senised tulemused

Edasiarendused

Autoware'i loojad on algoritmi edasi arendanud. Nad on Pure Pursuitile juurde implementeerinud vaateulatuse parameetri sõltuvuse kiirusest - sellega lahendatakse ära probleem, et sama vaateulatus ei sobi igale kurvile ja kiirusele. Lahendusega tekib algoritmile juurde mõned parameetrid: kiiruse kordaja (kiirus korrutatud selle parameetriga võrdub hetkeline vaateulatus) ning minimaalne lubatud vaateulatus. Viimasega välditakse seda,

et järgmine sihtpunkt ei tekiks sõidukile liiga lähedale. [7]

Teisena on Autoware algoritmi edasiarenduseks pakkunud välja lineaarse interpolatsiooni kasutamise. Põhjuseks probleem, et raja punktid on liiga hõredad ning nende üks haaval välja valimine sihtpunktideks võib põhjustada kurvides pööramise jõnksutamist. Lineaarne interpolatsioon võimaldab arvutada välja sihtpunkti ka hõredate rajapunktide vahele, mille tõttu sihtpunkti edasi liikumine toimub sujuvamalt. Nii välditakse pööramise jõnksutamist. [7]

Senine kasutus

[10] On teada, et Pure Pursuit kasutati autonoomsete sõidukite võistlusel DARPA Grand Challenge 2005. aastal kahel sõidukil. [11] Samuti kasutasid algoritmi kolm autot aastal 2007 DARPA Urban Challenge võistlusel. Tasub mainida, et viimane toimus linnatingimustes. Nagu ennem mainitud, kasutab Pure Pursuiti ka avatud lähtekoodiga autonoomse sõiduki tarkvara Autoware [5].

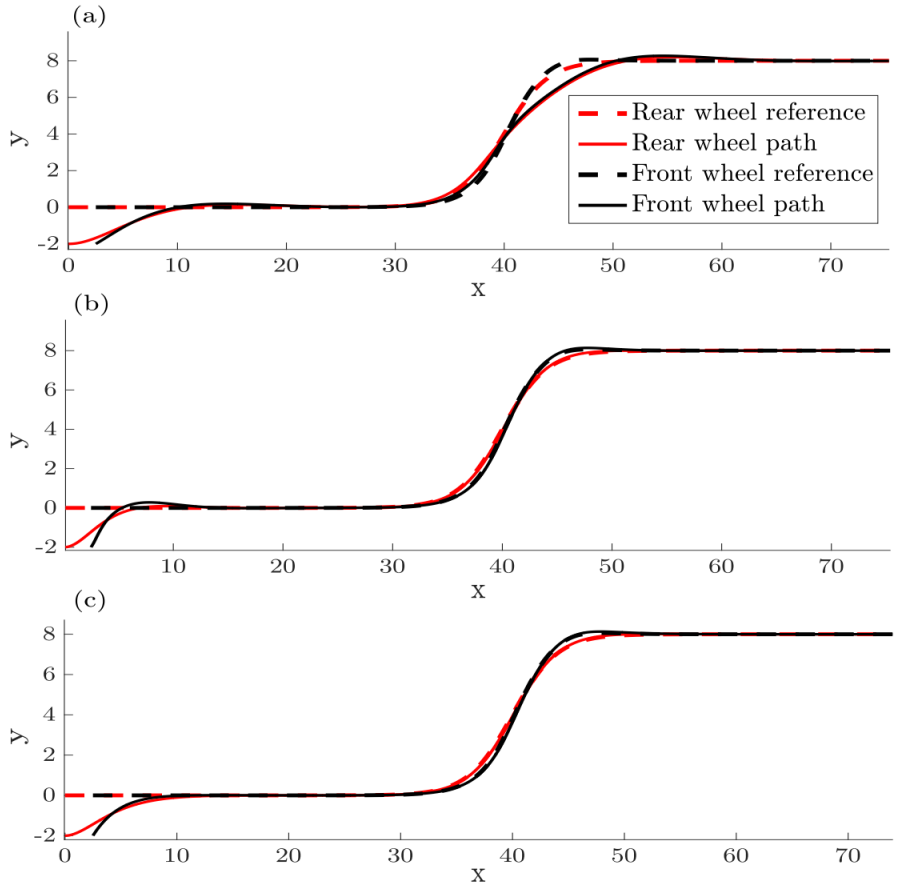
1.4.4 Alternatiivsed algoritmid

Ühe alternatiivina Pure Pursuit algoritmile pakub [4] välja tagaratta tagasisidel põhineva algoritmi. Algoritm kasutab tagumise ratta väljundit, et stabiliseerida tagumise ratta läbitavat rada.

Teise alternatiivina pakub [4] välja esiratta tagasisidel põhineva algoritmi. Algoritm kasutab samu muutujaid, mis tagarattal põhinev algoritm - ainult arvutusi tehakse esiratta põhjal, mitte tagaratta. Seda algoritmi kasutas auto, mis 2005. aastal võitis autonoomsete sõidukite võistluse DARPA Grand Challenge [12].

[4] Kuigi Pure Pursuit ei pruugi olla kõige täpsem juht võrreldes teistega (vt Joonis 1.4), on selle jõudlus rahuldav. Küll aga Pure Pursuitil on eelis teiste ees oma robustsuse tõttu. Kuna teised algoritmid kasutavad spetsiifilisemat tagasisidet, võib nende implementeerimine osutuda keeruliseks.

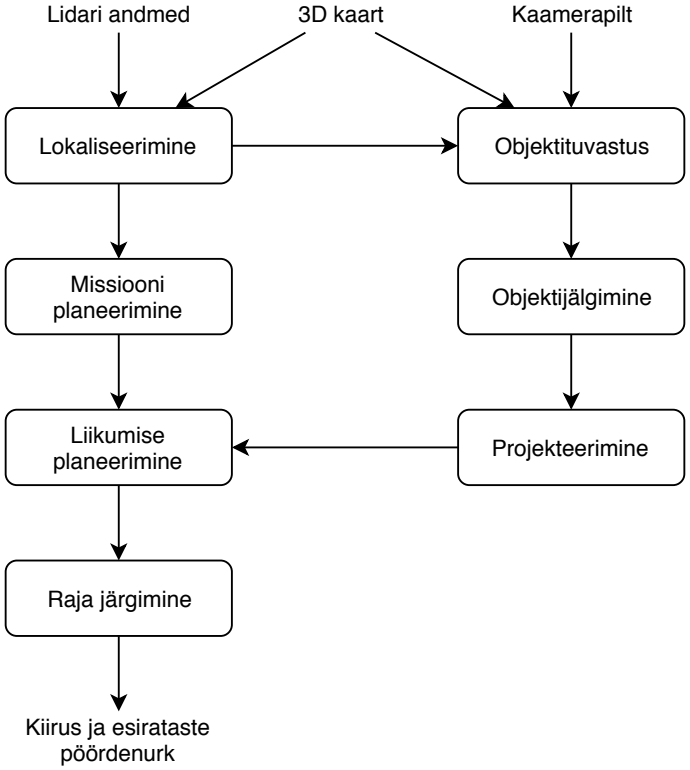
[5], [7] Autoware'i loojad ei maini, miks otsustati kasutada just Pure Pursuiti. Kuna projekti nõue on kasutada Autoware'i tarkvara, siis otsustame siiski uurida just seda algoritmi. Nii valideerime vabavaraliselt jagatavat tarkvara.



Joonis 1.4: [4] Algoritmide võrdlus reavahetusel. **(a)** Pure Pursuit **(b)** Tagaratta tagasisidel põhinev algoritm **(c)** Esiratta tagasisidel põhinev algoritm.

2. Autoware'i kasutamine

[5] Vabavaraline tarkvara Autoware on suuteline lahendama kõik vajalikud tarkvaralised alamprobleemid, et sõiduk saaks autonoomselt sõita. Meie eesmärk on uurida, kuidas Autoware lahendab raja järgimise probleemi. Raja järgimine on üks alamprobleemidest, mis on probleemide ahelas sõltuv mitmest teisest probleemist. Joonisel 2.1 on näha, et raja järgimise probleem nõuab eelnevalt kõikide teiste alamprobleemide lahendamist. Lisaks tasub mainida, et missiooni ja liikumise planeerimine nõuavad vektorkaardi olemasolu. Kuigi sõidukeskkonnast punktikaarti saab Autoware'iga luua, siis vektorkaarti antud tarkvaraga realiseerida ei saa.



Joonis 2.1: [5] Autoware'i lahendatavad autonoomse sõiduki alamprobleemid.

Meie autonoomse sõiduki projekt on jaotatud nõnda, et erinevad inimesed tegelevad

erinevate alamprobleemidega. Kuna alamprobleemide uurimine toimub paralleelselt, ei saa me eeldada, et probleemid, millest rajajärgimine sõltub, on lahendatud. Selle tõttu on tarvis luua lahendus, et raja järgimist saaks katsetada võimalikult väheste sõltuvustega. Kuidas seda saavutada, tuleb juttu järgnevas alampeatükis.

2.1 Raja järgimise sõltuvuste vähendamine

Tarvis on teada, millised on minimaalsed sõltuvused, millega raja järgimine töötab. Selleks vaatame täpsemalt, kuidas uuritav alamprobleem on Autoware'is lahendatud.

Raja järgimise väljund on sõiduki kiirus ning pöördenurk. Et neid kahte väärtust arvutada, kasutab Autoware algoritmi nimega Pure Pursuit. Algoritmi sisenditeks on:

1. sõiduki asukoht koos orientatsiooniga;
2. läbimist vajab rada;
3. sõiduki kiirus (valikuline).

Me ei saa vabaneda nendest alamprobleemidest, mis on vajalikud andmaks meile mainitud vajalikud sisendid Pure Pursuit algoritmi jaoks.

2.1.1 Raja järgimise sisendite hankimine

Sõiduki positsioon ja kiirus

Sõiduki asukoha määramisega tegeleb lokaliseerimise alamprobleem. Selleks kasutatakse Autoware'is lidarit. Sõidukit lokaliseerides on võimalik ka hinnata selle kiirust, seega kiiruse sisend algoritmile on meil tänu lokaliseerimisele olemas.

Rada

Raja etteandmisega tegeleb nii missiooni planeerimine, kui ka liikumise planeerimine. Nende kahe alamprobleemi koostööna tekitatakse lõplik rada, mis antakse ette raja

järgimise algoritmile. Nende kahe alamprobleemi sõltuvuse kasutamine lisab meile keerukad nõudmised: missiooni planeerimine nõuab vektorkaarti ning liikumise planeerimine objektituvastust kaamerate ja lidari andmeid sünkroniseerides.

Kuna liikumise planeerimise väljund on sama andmetüübiga, mis missiooni planeerimisest tulev rada, siis teoreetiliselt on meil võimalik objektituvastust ignoreerida. Seega liikumise planeerimise sõltuvuse me saame likvideerida.

Järgmisena vaatame, mis kuidas tekitatakse rada missiooni planeerimisel. Autoware'i kasutusjuhend [6] seletab, et rada on võimalik tekitada kahte moodi:

1. vektorkaardil alguspunkti ja lõpp-punkti vajutades;
2. Rosbag lindistust maha mängides.

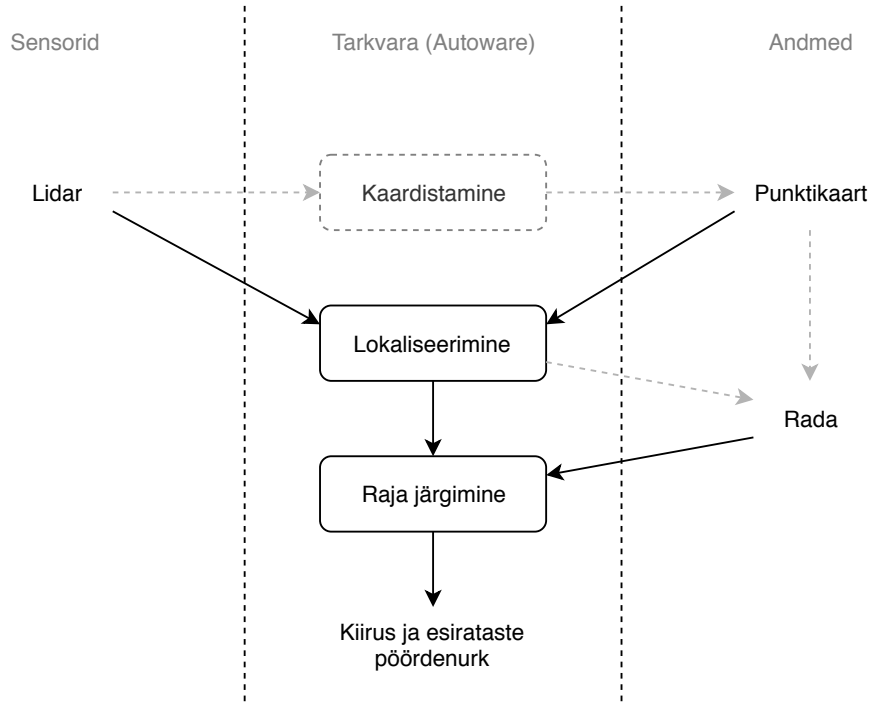
See on meile oluline info, kuna vektorkaardi loomist Autoware ei toeta. Seega ainus võimalus vektorkaardist sõltumata rada tekitada, on kasutada Rosbag lindistusi.

Rosbag võimaldab meil läbida sõidukiga mingisugune teekond ning lindistada lidari andmeid. Hiljem on meil võimalus lokaliseerida lindistatud lidari andmetega ning läbitud teekond salvestatakse CSV faili. See eeldab, et sõidetud on keskkonnas, mis on eelnevalt kaardistatud 3D punktipilvena. Seega meie puhul raja tekitamine on jällegi sõltuv lokaliseerimisest.

2.1.2 Lõplik sõltuvuste ahel

Vaatame uuesti Joonist 2.1. Teeme selle joonise ümber, jättes alles ainult need sõltuvused, mida on vaja, et raja järgimise algoritmile anda vajalikud sisendid. Tulemuseks on Joonis 2.2.

Järelikult ainukene autonoomse sõiduki alamprobleem, millest sõltume, on lokaliseerimine. Lokaliseerimine omakorda nõuab punktikaardi ja lidari olemasolu. Kui punktikaart on olemas, saame Rosbagiga salvestada sellele läbitava teekonna.



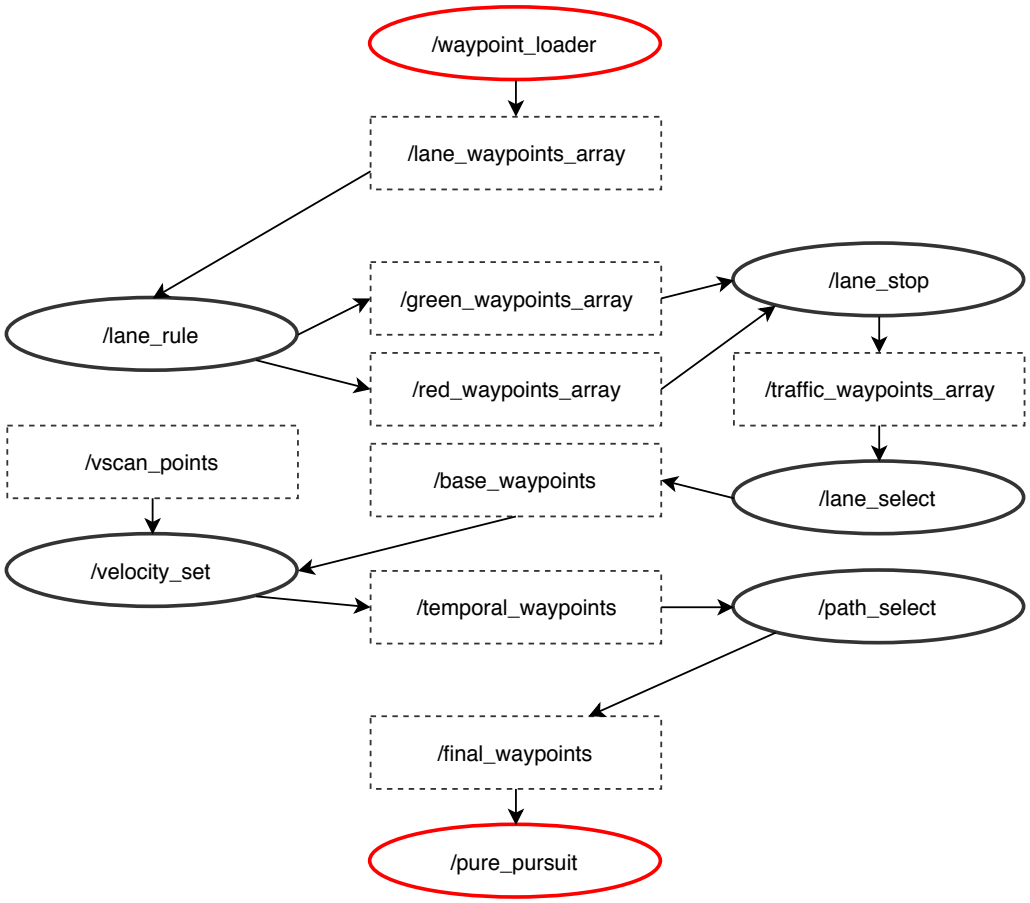
Joonis 2.2: Minimaalne sõltuvuste ahel raja järgimise katsetamiseks.

2.2 Sõltuvuste vähendamise realiseerimine ROS-i tasemel

Kuna punktikaardi loomine, lokaliseerimine, teekonna lindistamine ja teekonna sisselaadimine on Autoware'is juba realiseeritud, siis sellest detailsemalt juttu ei tule. Lahendus, mida tuleb ise realiseerida, on sisse antud teekonna üle andmine otse raja järgimise algoritmile. Autoware'is on olemas sõlm, mis laeb sisse teekonna CSV failist. Küll aga ei ole teada, et oleks sõlme, mis annaks selle otse edasi raja järgimise algoritmile. Teekond, mis laetakse failist sisse, on mõeldud läbi käima läbi liikumise planeerimise sõlmede, mis võtavad arvesse nii ümbritsevaid objekte kui ka vektorkaarti, et luua andmete põhjal lõplik teekond (vt Joonis 2.3).

2.2.1 Raja otsetee ROS-i sõlme loomine

Sõlme otsustame teha programmeerimiskeeles Python, kuna selle käitamine ja silumine on lihtsam, kui C++'il. Liiga palju seda valikut analüüsida ei ole vaja, kuna lõplikkusse autonoomse sõiduki lahendusse see sõlm ei lähe.



Joonis 2.3: [6] Lõpliku teekonna loomine mööda liikumisplaneerimise sõlmede.

Sõlmede vahelised sõnumid

Esimese asjana tuleb meil kindlaks teha, et ühendada omavahel raja sisselaadimise sõlm ja raja järgimise algoritm, missuguseid ROS-i teemasid ja sõnumitüüpe need kasutavad. Raja sisse laadimise sõlm kasutab teemat nimega lane_array_waypoints ning raja järgimise sõlm final_waypoints. Seega tuleb meil teha sõlm, mis kuulab lane_array teemat ning saadab välja final_waypoints teemat. Lisaks on meie sõlmel vaja teada sõiduki asukohta - selle saame kätte lokaliseerimise sõlmest, mis kasutab teemat nimega ndt_pose. Miks seda vaja läheb, tuleb täpsemalt juttu raja manipuleerimise alampeatükis.

Järgmisena vaatame, millist sõnumitüüpi vajaminevad teemad kasutavad. Raja sisse laadimise sõlm saadab välja autoware_msgs/LaneArray tüüpi sõnumeid ja raja järgimise sõlm kuulab autoware_msgs/lane tüüpi. Seega tuleb meil oma sõlmes teha nende kahe sõnumitüübi vahel ümbertõlgendus. Kui vaatame autoware_msgs/LaneArray sõnumitüübi lähtekoodi [13], leiame, et see koosneb autoware_msgs/lane tüüpi massiivist, ehk siis massiiv tüübist, mida kasutab raja järgimise algoritm. Kui uurime, kuidas sõlm käitub, kui

laeme talle sisse raja CSV-faili, siis näeme, et rada tekib massiivi esimeseks elemendiks. Järelikult piisab meie juhul, kui me edastame raja järgimise algoritmile esimese massiivi elemendi.

Kui uurime ROS-i siseselt, kui tihti raja sisselaadimise sõlm sõnumeid saadab, siis saame teada, et see sõnum saadetakse üks kord. Kui uurime lähtekoodist [9], kuidas raja-järgimise algoritm raja sisendit kasutab, siis saame teada, et see tahab saada sõnumit iga arvutuse korral. Seega me ei saa saata raja järgimise sõlmele teemat iga kord, kui ilmub lane_array sõnum. Meil on vaja saata teekond raja järgmisele tihedamini. Raja järgimise algoritmi koodi järgi [9] toimub uute väärtuste arvutamine sagedusega 30 Hz ning iga iteratsiooni lõpus eeldatakse, et rada enam ei ole. Seega on meil mõistlik saata uus rada sama sagedusega.

Raja manipuleerimine

Kui uurime, kuidas raja järgimise sõlm [9] rada kasutab, siis saame teada, et sõiduki kiirust võetakse alati esimese rajapunkti järgi. Seega tuleb meil oma sõlmes pidevalt rada muuta nii, et sõiduk asuks alati esimesele rajapunktile võimalikult lähedal. Sellega seoses tekib meil kaks probleemi:

1. sõlme käima pannes meil on olemas kõik rajapunktid ja me ei tea, milline rajapunkt peaks olema esimene;
2. rajapunkte tuleb pidevalt eemaldada, et sõiduk püsiks alati esimeses rajapunktis.

Esimese probleemi lahenduseks on meil vaja sõlme käima pannes leida sõidukile kõige lähem rajapunkt. Kui kõige lähem rajapunkt on olemas, seame selle esimeseks rajapunktiks ning kõik, mis sellele eelnevad, eemaldatakse. Selles probleemis tuleb ka välja, et meie sõlmel on vaja teada sõiduki asukohta.

Teise probleemi lahendamiseks on meil vaja tuvastada, kas järgnev rajapunkt on sõidukile lähemal, kui see, kus me hetkel oleme. Kui on, siis kustutame esimese rajapunkti ära ja teisest rajapunktist saab uus esimene. Selle abil me kindlustame, et rajapunktide eemaldamine käib järjekorras ning teekonna kattuvuse korral ei lähe meil rada kaduma.

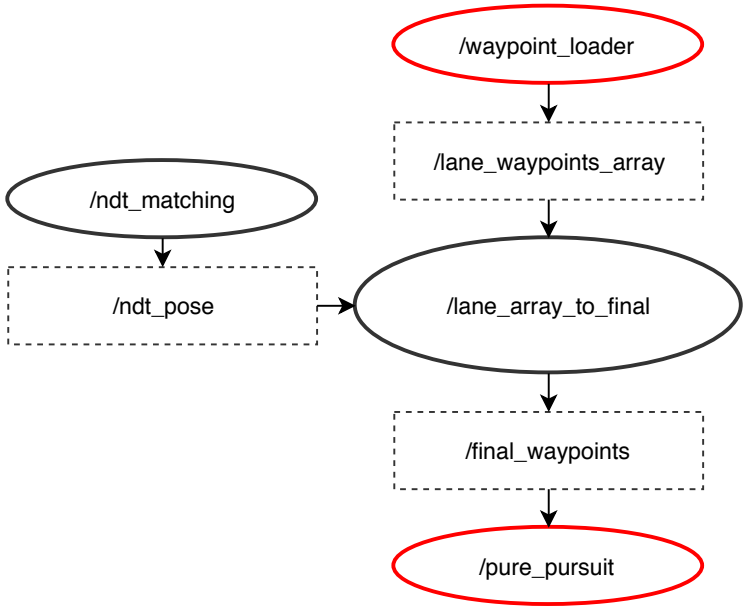
Kuna mainitud probleemid on funktsionaalselt väga erinevad, ei saa me neid kasutada korraga. Teise probleemi lahendamiseks me ei saa kasutada esimese lahendust, kuna kui esineb rajapunktide kordusi (näiteks ringiratast sõitmine), siis võidakse hakata osasid

rajapunkte vahele jätma. Teiseks, meil ei ole mõtet läbi käia kõiki rajapunkte igal iteratsioonil, et teada saada, kas me oleme järgmisele rajapunktile lähemal. Pikemal teekonnal, millel on rohkem rajapunkte, võib arvutamine võtta liiga palju aega.

Et need kaks lahendust üksteist ei segaks, on meil tarvis teha muutuja, mis hoiab mälus seda, missugune probleem meil hetkel on. Sõlme käima pannes on meil alati esimene probleem ning olles selle lahendanud, on meil käsil teine. Selle jaoks teeme muutuja is_at_first_waypoint, mis eesti keeles on "kas on esimese rajapunkti juures", ja kasutame seda boolean tüübina. Kui sõlm pannakse käima, on muutuja tüübi väärtus väär ja kui esimene probleem on lahendatud, siis seatakse muutuja väärtuseks tõene.

Kokkuvõte

Meil on sõlm, mis kuulab kahte teemat: lane_array_waypoints ja ndt_pose. Meie sõlm saadab ühte sõnumit nimega final_waypoints sagedusega 30 Hz. Sõlm muudab dünaamiliselt teekonda, et sõiduk asuks alati esimeses rajapunktis, kuna raja järgimise algoritm võtab oma kiiruse alati esimese teekonna punkti järgi.



Joonis 2.4: Meie loodud sõlm, lane_array_to_final, ahelas.

2.3 Lahenduse valideerimine simulaatoris

Raja järgimise sõlmele teekonna etteandmise lahendust valideerime kasutades simuleerimisprogrammi Gazebo. Koos Autoware'iga tuleb kaasa projekt Catvehicle [14], millele saab anda sisenditeks nii auto kiiruse kui ka esirataste pöördenurga. Lisaks on Catvehicle'il olemas simuleeritud lidar, mida saab katsetada maailma kaardistamiseks, raja tekitamiseks ning lokaliseerimiseks.

2.3.1 Andmete tekitamine

Esimese asjana tuleb simulaatori maailmast tekitda 3D punktikaart, mille abil me hakkame hiljem lokaliseerima. Kaardi loomiseks kasutame Rosbagi [15], millesse lindistame kõik sõidust tekkinud lidari andmed, mis tulevad meie simulatsiooni maailmat läbi sõites. Kui andmed on olemas, tuleb lindistus maha mängida koos Autoware'i sõlmega ndt_mapping. Programmi RViz abil näeme, kuidas tekib 3D punktikaart.

Seejärel sõidame autoga veel maailma läbi, et salvestada teekond, mida sõiduk hakkab hiljem autonoomselt järgima. Selleks kasutame Autoware'i sõlmesid ndt_matching ning waypoint_saver. Esimene tegeleb sõiduki positsiooni hindamisega, sobitades hetkelisi lidari andmeid olemasoleva punktikaardiga. Teine salvestab maha lokaliseerimisest tulevad positsiooni andmed. Raja salvestamise sõlm on konfigureeritav selliselt, et see salvestaks rajapunkti iga mingisuguse kauguse tagant. Meie kasutame vaikimisi kaugust 1 meeter. Kui teekond on läbitud, tuleb waypoint_saver sõlm kinni panna ning selle tulemusel salvestatakse läbitud teekond punktide kujul CSV-faili. Iga teekonna punkt koosneb neljast parameetrist: x-koordinaat, y-koordinaat, z-koordinaat, suund radiaanides ning hetkeline kiirus rajapunktis.

Sellega on meil eeltöö tehtud. Olemas on kõik vajalikud andmed, et simulaatoris katsetada autonoomset sõiduki liikumist.

2.3.2 Autonoomne sõitmine simulaatoris

Sõlm simulatsiooni sõidukile käskude andmiseks

Enne, kui me saame raja järgimise algoritmist anda käske edasi simulatsioonis olevale sõidukile, on meil vaja teha üks täiendav sõlm. Sõlm peab tõlgendama järgimisalgoritmist tuleva teema ctrl_cmd catvehicle/twist_cmd-ks. Ctrl_cmd nimeline teema sisaldab sõnumeid tüübist autoware_msgs/CommandControlStamped [16], mis tähendab, et sõnum hoiab endas päist, sõiduki kiirust meetrites sekundis ning soovitud pöördenurka radiaanides. Simulatsiooni sõiduki teema võtab vastu aga sõnumeid tüübist geometry_msgs/Twist [17], mis sisaldab kahte kolmemõõtmelist vektorit - üks lineaarne, teine nurgeline. Lineaarse vektori x muutuja annab autole sisendkiiruse meetrites sekundis. Nurgelise vektori z muutja annab sõidukile vastava pöördenurga radiaanides.

Nagu eelnevast analüüsist on näha, on sõiduki kiiruse ja pöördenurga ühikud saatvas ja vastuvõtvas teemas samad. Seega ainuke protseduur, mida uus sõlm peab tegema, on muutma sõnumi tüüpi.

Sõidu käivitamine

Sõiduk peab olema meie simulatsioonimaailma punktikaardil lokaliseeritud. Selleks tuleb käima panna Gazebo simulaator koos Catvehicle'i ja maailmaga, millest meil on tehtud 3D punktikaart. Peale seda tuleb meil Autoware'is käivitada sõlmed, mis võimaldavad meil simuleeritud lidariga lokaliseerida. Esiteks, on meil vaja selleks sõlme, mis laeb ROS-i sisse meie punktikaardi, ning teiseks, sõlm, mis mis hindab punktikaardil meie asukohta.

Edasi on meil vaja teekonda, mida läbida. Selleks kasutame me enne mainitud teekonda CSV kujul, mille me salvestasime sõidukiga maailmast läbi sõites. Raja sisse laadimise ROS-i teeb meie jaoks Autoware, kuid selleks, et raja saaks kätte ka teekonna järgimise algoritm, peame nüüd käivitama ka meie loodud otsetee sõlme. Kuna see ei ole Autoware'i sõlm, ei saa me kasutada selleks Autoware'i kasutajaliidest. Et sõlme käivitada, kasutame eraldi käsurida.

Edasi jääb meil üle panna tööle raja järgimise algoritm Pure Pursuit. Selleks valime Autoware'i kasutajaliidesest vastavad Pure Pursuiti parameetrid ning käivitame selle. Lisaks tuleb Autoware'ist eraldi panna veel tööle sõlm, mis tõlgendab järgimisalgoritmi

käsud simulatsiooni sõiduki jaoks.

Kui need etapid on läbitud, näeme Gazebo aknas sõiduki autonoomset liikumist. Rada, mida läbitakse, on näha RVizi aknas koos punktikaardi, lidari skaneeringu ja raja järgimise algoritmi visualiseerimisega.

Täiendused ja tulemused

Simulatsioonis töötab raja järgimine ootuspäraselt. Umbes 10 testimisega on võimalik saada raja järgimise algoritmi parameetrid rahuldavasse seisu, et sõiduk ei lõikaks liiga palju kurve ning et ei oleks mööda rada slaalomi sõitmist või kurvi korral rajalt välja sõitmist.

Nagu mainitud, on meil soov teha rohkem katsetusi erinevate parameetriga. Autoware'i kasutajaliidest kasutades võib see olla aega nõudev, kuna iga sõlme käima panemine nõuab kasutaja sisendit. Et kiirendada katsetamise käivitamise on meil võimalus luua ROS-i käivitusfailid. Käivitusfailid võimaldavad meil initsialiseerida mitmeid ROS-i sõlmesid ühe käsurea lausega. See tuleb kasuks, kui tahta muuta erinevaid parameetreid ning nende tulemusi võrrelda, nähes vähem vaeva kogu protsessi käivitamisega.

3. Uku roboti ühendamine Autoware'iga

Selleks, et Uku saaks suhelda Autoware'iga, on meil tarvis teha ROS-i sõlm, mis võtab vastu vajalikud ROS-i teemad ja sõnumid ning saadab käsud robotile. Uurime, mis tehnoloogiaid saame kasutada, et Ukut juhtida ning koostame nõuded, mis teeksid katsetamised platvormiga võimalikult lihtsaks ja turvaliseks.

3.1 Nõuded

Selleks, et sõlme luues oleks meil olemas tugipunkt, koostame Autoware'i ühildussõlmele nõuded:

1. robot peab olema puldiga juhitav (puldi mudel: Logitech Wireless Gamepad F710);
2. puldiga peab saama erinevaid sõidurežiime muuta;
3. Autoware'i käsud peavad edasi kanduma robotile;
4. turvalisuse huvides peab autonoomsel sõitmisel sõiduki liikumiskiirus olema kontrollitav;
5. turvalisuse huvides sõidurežiimi aktiveerimine ei tohi juhtuda kogemata nupule vajutades.

Esimene nõue on vajalik, kuna Uku peab olema käsitsi juhitav. Kuigi meie eesmärk on panna robot autonoomselt sõitma, on meil vaja sõiduk sõita laborist testimise alale ning vastupidi. Puldivalik on selline, kuna Logitech F710 pult on meile katsetamiseks antud. Arendamiseks saab kasutada ka Xbox 360 pulti, kuna nende mõlema puldi ROS-i liides ning füüsilised nupud ja juhtkangid on samad, kuid erineva paigutusega.

Nõue number 2 on vajalik, et teha ümberlülitusi võimalikult lihtsaks ja kiireks. Selleks, et lülitada ümber autonoomne sõit manuaalseks sõiduks, võib teha ka eraldi ROS-i sõlmed, kuid see nõuab testimise ajal nende ümberkäivitamist. Kuna ümberkäivitus võtab aega, on seda mõistlikum teha puldi pealt nupu vajutusega.

Kolmas nõue on selleks, et Autoware saaks ühenduse robotiga. See nõue tingib meile autonoomse sõidu.

Neljas nõue on eelkõige turvalisuse tagamiseks. Kui autonoomne sõidurežiim on sisse lülitatud, võib sõiduk hakata koheselt esirattaid pöörama, kuid mitte edasi liikuma. Sõiduki edasiliikumiseks peab katse läbi viija hoidma all puldil vastavaid nuppe. Selle lahendusega väldime soovimatuid avariisid, mis võivad tekkida autonoomsel sõitmisel.

Nõue number 5 on vajalik selleks, et manuaalse ja autonoomse sõidu režiim ei lülituks sisse katse läbiviija tahtmata. Selle tagamiseks peab enne sõidurežiimi sisse lülitamist hoidma puldil all vastavaid turvanuppe. Selleks, et panna sõiduk parkimisrežiimi, turvanuppe all ei pea hoidma - seda selleks, et häda korral saaks sõidukit peatada võimalikult ruttu.

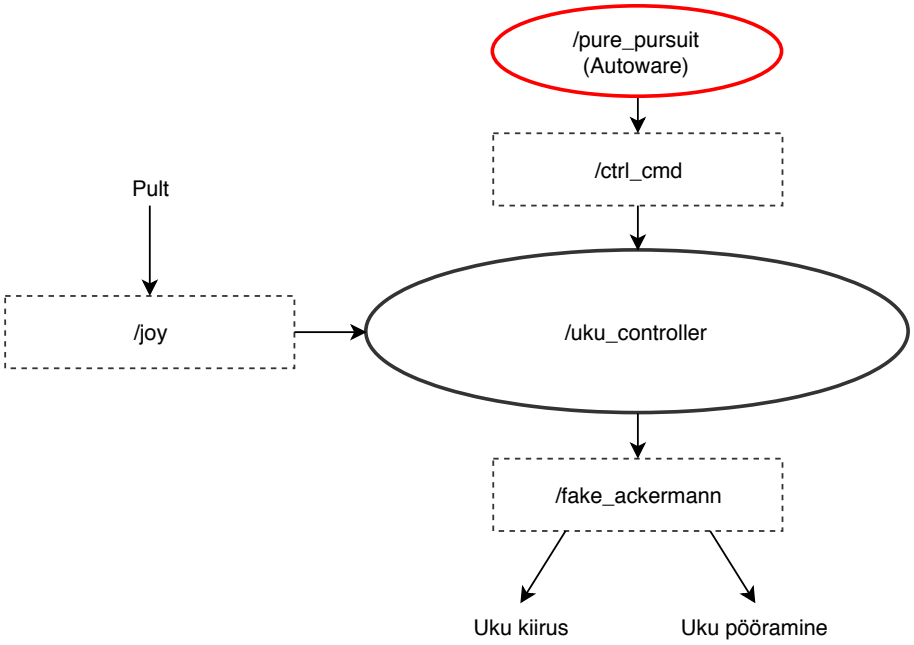
3.2 Lahendus

Iseauto projekti raames ehitati Uku ümber nii, et ROS oleks suuteline andma juhtimiskäske edasi robotile. Selleks on välja arendatud süsteem, kus Uku mootoreid kontrollivad kaks Arduino mikrokontrollerit, mis on programmeeritud olema osa ROS-ist. Üks Arduino juhib pööramismootorit, teine sõiduki kiirust. Selleks, et tekiks ühendus ROS-iga, tuleb mõlemad mikrokontrollerid ühendada juhtarvutisse USB-kaabliga ning käivitada arvutis vastavad ROS-i sõlmed, mis loovad ühenduse ROS-i tuumikuga. Seejärel tekib ROS-i teema nimega fake_ackermann_cmd, mis on tüübiga geometry_msgs/Vector3Stamped. [18] Mainitud sõnum hoiab endas päist ning väärtuseid x, y ja z. Kuidas need väärtused mõjutavad Uku platvormi, on näha Tabelis 3.1.

Atribuut	Funktsioon	Ühik
x	Sõiduki kiirus	m/s
y	-	-
z	Esirataste pöördenurk	rad

Tabel 3.1: geometry_msgs/Vector3Stamped atribuudid ja nende funktsionaalsus Uku platvormil.

Peatükist 2.3.2 on teada, et raja järgimise algoritm annab meile ctrl_cmd teema sõnumitüübiga autoware_msgs/ControlCommandStamped. Juhtpuldist tulevad sõnumid läbi sõlme, mis on ROS-is vaikimisi olemas, ning see saadab välja teemat nimega joy. Antud teema sees liiguvad sõnumid tüübiga sensor_msgs/Joy [19]. Selleks, et Uku reageeriks nendele teemadele ja sõnumitele nõuete kohaselt, tuleb meil mainitud teemasid kastudes tekitada väljund teemasse fake_ackermann_cmd tüübiga geometry_msgs/Vector3Stamped (vt Joonis 3.1).



Joonis 3.1: Uku kontrolleri sõlm ahelas.

3.2.1 Puldi sisend

Puldi sõlm saadab väljundi nii pea, kui toimub muutus kasutaja sisendis. Kui puldil ei vajutada ühtegi nuppu või ei muudeta ühegi kangi positsiooni, siis sõnumite saatmist teema sees ei toimu. Tänu sellele ei ole meil mõistlik saata robotile kiiruse ja pöördenurga käskusid koheselt, kui sõnum puldist tuleb. Vastasel juhul, kui puldist tuleb näiteks sõnum, et kiirus olgu 0 m/s, siis see sõnum saadetakse robotile ainult ühe korra. Kui see sõnum juhuslikult kohale ei jõua, siis robot seisma ei jää. See risk on suurem, kui muidu, kuna suhtlus roboti sõlmedega käib läbi USB-kaabli.

Selle probleemi lahendamiseks hoiame puldi väljundeid sõlme sees muutujatena, mida uuendatakse vastavalt siis, kui puldist tuleb uus sõnum. Andmed, mis on muutujates, saadame robotile edasi 30Hz sagedusega. St, kui nüüd tuleb puldist käsk, et kiirus olgu 0

m/s, siis seda ei saadeta ainult ühe korra, vaid 30Hz sagedusega nii kaua, kuni see väärtus jälle ära muudetakse.

Olekurežiimid

Loome kolm olekurežiimi, millest kaks on sõidurežiimid:

1. parkimine;
2. autonoomne (sõidurežiim);
3. pult (sõidurežiim).

Selleks, et režiime saaks puldilt muuta, määrame igale režiimile oma nupu, mida vajutades see aktiveerub. Nõude nr 5 järgi peab sõidurežiimi aktiveerimine olema turvatud. Selleks määrame ka turvanupu/nupud, mida tuleb all hoida samal ajal, kui turvet vajavat funktsiooni kasutada. Antud projektis realiseerime selle nõnda, et sõidurežiimi aktiveerides tuleb all hoida kahte puldi tagumist nuppu ja seejärel valida vastav sõidurežiim. Selleks, et panna sõiduk parkimisrežiimi, turvanuppe vaja ei ole.

Sõlme sees hoiame režiimi jaoks vastavat muutujat, mille järgi valime, millisest allikast me parasjagu andmeid saadame. Parkimisrežiimi puhul saadame robotile sisenditeks 0 m/s kiiruse ning 0 rad pöördenurga sagedusega 30 Hz.

Puldiga juhtimine

Selleks, et Ukut juhtida, kasutame puldil olevaid juhtkange - üks kiiruse, teine pöördenurga muutmiseks. Juhtkangid annavad välja väärtuseid -1.0 kuni 1.0. St kui kiiruse kangi väljundväärtus on 1.0, siis roboti liikumiskiiruseks on maksimaalne lubatud kiirus. Kui väljundväärtus on -0,5, siis robot liigub tagasisuunas poole maksimaalse kiirusega. Sama kehtib ka pöördenurga korral.

Maksimaalne kiirus ning pöördenurk on konfigureeritavad ROS-i tasemel. Sõlme käima pannes antakse kaasa vastavad parameetrid, mis need defineerivad.

Nagu enne mainitud, saadame me robotile käske 30 Hz sagedusega. Selleks on meil sõlme sees tsükkel, mis töötab 30 Hz sagedusega. Igal tsüklil saadetakse robotile viimane aktiivne kiirus ja rataste pöördenurk, mis puldilt kätte saadi.

3.2.2 Autonoomne sõit ehk Autoware'i sisend

Autonoomsel sõidul on meil vaja edasi kanda Autoware'ist tulevead kiiruse ja esirataste pöördenurga väärtused. Kuna siin on reageerimiskiirus olulisem, kui puldiga juhtimisel, siis saadame käsud edasi nii pea, kui need meie sõlme jõuavad - erinevalt puldiga juhtimisest, kus meil on tsükkel, mis väärtuseid saadab.

Nõude nr 4 järgi peab roboti kiirus olema kontrollitav manuaalselt. Seega on meil tarvis defineerida, kuidas me seda teeme. Kuna meil on olemas pult, millega saab saata väärtuseid, on meil mõistlik kontrollimiseks kasutada just seda. Puldi peal on olemas veel juhtkange, mis annavad meile väärtuseid vahemikus -1.0 kuni 1.0. Meil on teada, et raja järgimise algoritm annab meile kindla kiiruse, millega sõita. Seega on meil võimalus korrutada puldi väärtus raja järgimise algoritmist tuleva väärtusega. Sellisel juhul, kui hoida juhtkangi põhjas, sõidab robot selle kiirusega, mis raja järgimise algoritm sellele annab. Kui kangi ei liigutata, siis robot edasi ei liigu.

Nagu ennem mainitud, on meil olemas kaks turvanuppu. Selleks, et autonoomse sõidu ohutus oleks suurem, otsustame siin kasutada ühte neist. See tähendab, et selleks et autonoomse sõidu kiirust kontrollida, peab ühte turvanuppu all hoidma. Sellega ennetame tahtmatut juhtkangi liigutamist.

3.3 Käivitamine

3.3.1 Puldidga juhtimise käivitamine

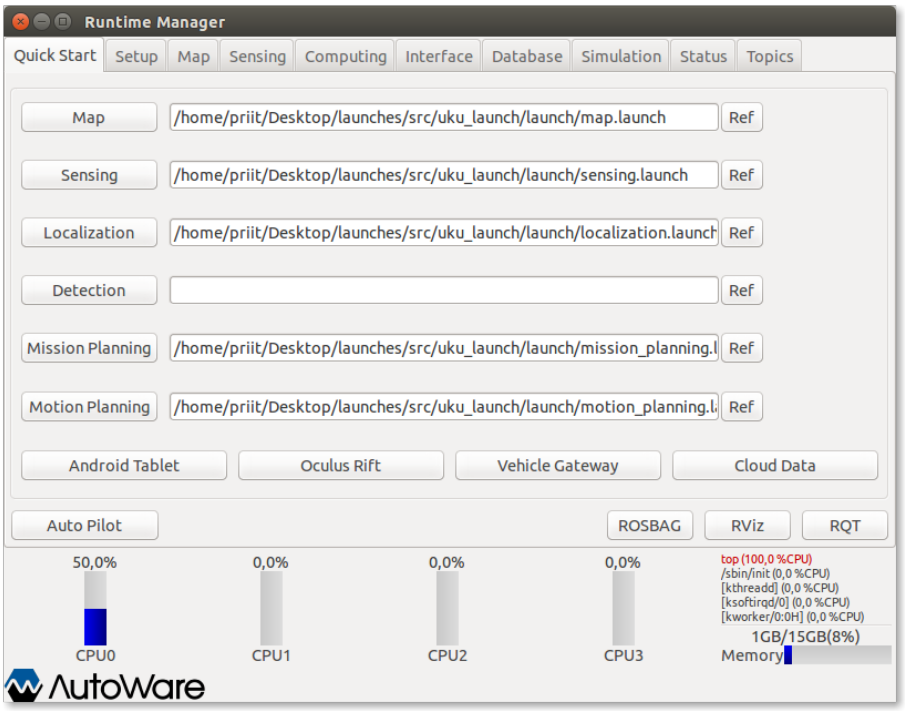
Puldiga Uku juhtimise jaoks on meil tarvis panna käima 4 sõlme:

1. puldi kuulamise sõlm "joy"(vaikimisi on see ROS-is olemas);
2. rataste pööramise mikrokontrolleri sõlm;
3. Uku liikumiskiiruse mikrokontrolleri sõlm;
4. meie loodud sõlm, mis ühendab puldi ning Autoware'i Uku mikrokontrolleritega.

Kui need sõlmed on käivitud, saame me Ukut juhtida puldi abil. Uku on puldiga juhitav ka siis, kui Autoware'i autonoomse sõidu sõlmed pole sisse lülitatud.

Kuna 4 sõlme käivitamine iga kord, kui tahetakse sõita, on tülikas, teeme selle lihtsamaks. Selleks, lisame kõikide sõlmede käivitused ühte ROS-i käivitusfaili. Selle abil saame kõik vajalikud sõlmed panna käima käsurealt ühe lausega.

Kui käivitusfail on olemas, saame seda lihtsasti kasutada ka Autoware'i kasutajaliideses. Autoware'is on olemas nupp, nimega "Vehicle Gateway", mis on mõeldud sideme loomiseks tarkvara ja roboti vahel (vt Joonis 3.2). Selleks, et side tekiks Uku ja tarkvara vahel, on meil vaja välja vahetada antud nupu funktsionaalsus. Et seda saavutada, otsime välja vastava ROS-i käivitusfaili, mis pannakse käima vajutades antud nuppu. Seejärel kustutame või kommenteerime välja seal olemas olevad käivitused ning lisame sinna oma käivitusfaili. Seejärel saame Uku, puldi ja Autoware'i vahelise liidese käivitada lihtsa nupu vajutusega Autoware'i kasutajaliidesest.



Joonis 3.2: Autoware'i kasutajaliides.

3.3.2 Autonoomse sõidu käivitamine

Selleks, et panna robot autonoomselt sõitma mööda rada, on meil vaja käima vähemalt 12 erinevat sõlme. Käsurealt neid eraldi käima pannes läheb katsetamine segaseks. Erinevalt simulaatoris katsetamisest, ei taha me kõiki sõlmesid ühte käivitusfaili panna, kuna katsetades samal testimisel erinevaid radu, me tahame osasid sõlmesid hoida töös terve testimise aja - nt lokaliseerimise sõlmed. Iga kord, kui lokaliseerimise sõlmed pan-

nakse tööle, on vaja käsitsi defineerida algpositsioon punktikaardi suhtes. Minnes testima ning hoides lokaliseerimist algusest peale sees, on vaja seda ainult ühe korra terve testimise jooksul - eeldades, et lokaliseerimine vahepeal ära ei kao. Selleks pakub Autoware välja sõlmede jaotamist erinevatesse kategooriatesse (vt Joonis 3.2):

1. kaart;
2. tajumine (sensorite kontekstis);
3. lokaliseerimine;
4. objektituvastus;
5. missiooni planeerimine;
6. liikumise planeerimine.

Jaotame oma vajalikud sõlmed eelmainitud kategooriatesse ära. Ainus kategooria, mida meil raja läbimiseks kasutada vaja ei ole, on objektituvastus. Kui jaotamine eraldi käivitusfailidesse on tehtud, tuleb nende asukohad laadida paremalt poolt kategoorianuppu kasutajaliidesesse (vt Joonis 3.2). Seejärel vastavat kategoorianuppu vajutades pannakse käima kõik sõlmed, mis antud kategooria käivitusfailis defineeritud on.

Kategooriate käivitamise üksikasjad

Eeldades, et puldiga juhtimine on robotil käivitatud (vt Peatükk 3.3.1), ning autonoomse sõidu kategooriate käivitusfailid on Autoware'i kasutajaliidesesse sisse laetud (vt Peatükk 3.3.2), saame hakata käivitama autonoomset roboti juhtimist.

Esimese asjana tuleb käima panna kategooriad kaart ja tajumine. Kaardi käivitamine laeb sisse punktikaardi ning selle paigutuse RVizi maailma suhtes. Hiljem saab sinna lisada ka vektorkaardi sisselaadimise. Tajumise käivitus paneb käima suhtluse lidariga. Tajumise kategooria on mõistlik kaardist eraldi hoida, kuna hiljem lindistusi maha mängides ei ole tajumise kategooriat vaja.

Seejärel lülitame sisse lokaliseerimise. Lokaliseerimise käivitamine eeldab kindlasti kaardi ja lidari andmete olemasolu. Peale lokaliseerimise käivitamist tuleb määrata robotile algne positsioon punktikaardi suhtes. Seda saab teha läbi programmi RViz, kus silma

järgi saab hinnata hetkelist asukohta ning orientatsiooni. Kui sõiduki asukoht on korrektselt määratud, näeme RVizis lidari andmete ja punktikaardi kokkulangevust.

Läbitavat teekonda saame sisse laadida vajutades missiooni planeerimise nupule. See eeldab, et meil on olemas teekonna CSV fail ning õiges kohas, kust see sisse laetakse (see on konfigureeritav käivitusfailides). Seejärel on meil RVizis näha teekond, mida plaanitakse sõita. Antud kategooriat on soovituslik käivitada peale kaardi sisse laadimist, kuna siis on RVizist aru saada, kuhu rada tekkis.

Kui eelnevad sammud on tehtud, võime käima panna raja järgimise algoritmi - selleks vajutame nuppu "Motion planning". Enne käivitamist on vajalik, et rada on sisselaetud ning sõiduki asukoht on lokaliseerimise abil korrektselt määratud. Kui käivitamine on korrektselt tehtud, näeme RVizis algoritmi töö projektsiooni raja peal.

Edasi jääb meil üle lülitada sõlm puldist üle autonoomse sõitmise režiimile ja hoida nuppe all, mis lubavad robotil sõita. Kui meil on plaanis teha mitu katsetust, siis testide vahel tasub taaskäivitada ainult missiooni ja liikumise planeerimist. Kaart, tajumine ja lokaliseerimine võivad terveks testimise ajaks tööle jääda. Kui juhtub, et lokaliseerimine on ära kadunud, tuleb algpositsioon RVizis uuesti määrata.

4. Mitsubishi i-Mievi ühendamine Autoware'iga

Mitsubishi i-Mievi ühendamiseks Autoware'iga on välja töötatud sõlm, mis suhtleb auto juhtkontrolleriga üle Etherneti kaabli kasutades UDP protokolli. Sõlm võtab ROS-i tasemel vastu käsud raja järgimise algoritmilt ning edastab need üle Etherneti auto kontrollerile. See on erinev Ukust, millel kasutati Arduino arendusplaate, mis olid arvutiga ühenduses USB kaablitega ning mõlemad kontrollerid kuulasid ROS-i sõnumeid.

Suhtlemiseks on valitud UDP protokoll, kuna autot juhtides on oluline, et kohale jõuaks kõige uuem käsk. TCP protokoll tekitaks võrgus liiga palju sõnumite saatmist ning selle tõttu võib saadetud sõnum olla aegunud või kohale jõuda vales järjekorras.

UDP sõnumeid vahetatakse bait-kujul sagedusega 100 Hz. Hetkel on välja töötatud ainult arvuti poolne sõnumite saatmine - kontrollerist infot veel vastu ei võeta.

Kontrollerisse mineva sõnumi sisuks on sõiduki kiirus, esirataste pöördenurk ning sõidurežiim (vt Tabel 4.1). Sõidurežiimideks on parkimine, autonoomne ja puldiga juhtimine. Sõnumi suurus on 12 baiti ning kõik andmed esitatakse jadas üksteise järgi. Igale andmele eelneb vastava andme ühebaidine ID. Kuna kontroller loeb baite vastupidises järjekorras arvutiga, siis selle tõttu tuleb mitmebaidistel andmetel (sõidukiirus ning esirataste pöördenurk) baitide järjekord muuta vastupidiseks. Näide sõnumist, kus kiiruseks on seatud 1.23 m/s, rataste pöördenurgaks 0.2571 rad ning sõiduk on autonoomses režiimis: `0x10 0x3f9d70a4 0x20 0x3e83a29c 0x30 0x02`.

ID	Funktsioon	Andmetüüp (keeles C)	Ühik
0x10	Sõiduki kiirus	float	m/s
0x20	Esirataste pöörde nurk	float	rad
0x30	Sõidurežiim	uint8	režiimi id

Tabel 4.1: Mitsubishi i-Mievi kontrollerile saadetavad andmed.

Erinevalt Ukust, realiseeritakse puldiga juhtimine riistvara tasemel. Sellest hoolima-

ta on meil siiski kasulik kasutada pulti ROS-i tasemel, et muuta katsetustel sõidurežiimi ning kontrollida autonoomsel sõidul sõiduki kiirust. Puldi sisendi realiseerime eraldi sõlmena, et sõidukiga suhtlemise sõlm realiseeriks ainult neid funktsioone, mis lõplikus autos olema peab.

Kuna Mitsubishi i-Mievi platvorm pole selle töö kirjutamise ajal autonoomseteks katsetusteks valmis, ei ole tehtud ühtegi praktilist katsetust autoga sõitmisel. Katsetatud on ainult ROS-i sõnumite saatmist kontrollerisse UDP protokolliga.

Mievi platvormi käivitamine läbi Autoware'i kasutajaliidese näeb välja täpselt samasugune nagu Uku puhul. Käivitusfailides tuleb ära vahetada algoritmide parameetrid ning sõlm, mis suhtleb autoga.

5. Tulemused

Antud peatükis keskendume raja järgimise jõudluse hindamisele. Selleks, et hinnata, kui hästi algoritm rada järgib, uurime, kuidas erineb sõiduki trajektoor ettenähtud rajast ning uurime ka sõidu sujuvust kasutades kiirendussensorit.

Katsetusi tehakse ainult robotplatvormil Uku, kuna töö kirjutamise ajal oli antud platvorm ainsana piisavalt valmis. Iseauto projekti hilisemas faasis katsetused jätkuvad sõiduauto Mitsubishi i-Mievi peal.

Tulemusi vaadates tasub silmas pidada, et Uku roboti arendamine käis paralleelselt raja järgimise katsetustega. See tähendab, et roboti jõudlus muutus erinevatel katsetel märgatavalt. Seega saame tulemustest järeldada, kuidas Autoware'is implementeeritud raja järgimine toimib ebastabiilsel platvormil.

5.1 Raja järgimise täpsus

Selleks, et hinnata raja järgimise täpsust, lindistame kogu sõidu vältel maha sõiduki asukoha. Sõiduki asukoha hindamine käib punktikaardi ning lidari abil, kus NDT algoritm sobitab hetkelisi lidari andmeid olemasoleva punktikaardiga. Lindistatud teekonnad tähistavad lidari asukohta. Lidar asub sõiduki peal ülevalt alla vaadates esirataste keskel.

Teeme läbi 3 katsetust, kus proovime erinevaid raja järgimise algoritmi, Pure Pursuiti, parameetreid. Seejärel hindame saadud tulemusi graafikute põhjal. Kaks asja, mille puhul Pure Pursuiti parameetreid tasub hinnata, on see, kui palju lõigatakse kurve ning kui palju esineb sirgel teel slaalomit. Seega vaatleme eraldi, kuidas raja järgimise algoritm erinevate parameetritega töötab nii sirgel kui ka kurvis.

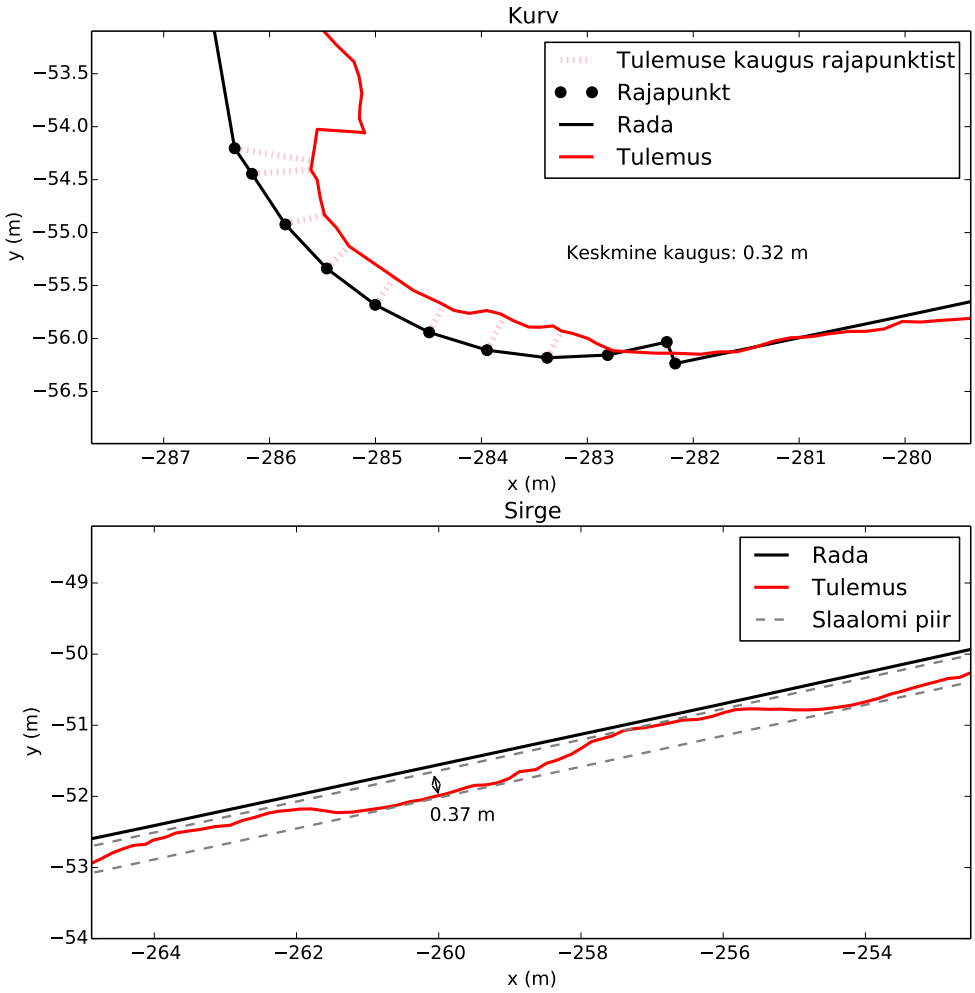
Pure Pursuit algoritmil, mis on implementeeritud Autoware'is, saame muuta kahte parameetrit - minimaalne vaateulatus ning kiiruse kordaja. Viimane parameeter korruta-

takse hetkelise kiirusega, mille tulemusel saadakse väärtus, millest saab uus vaateulatus. Kui viimane väärtus on väiksem, kui minimaalne vaateulatus, kasutatakse väikseimat lubatud vaateulatust.

Et hinnata kurvi lõikamist, leiame iga kurvi rajapunktist kõige lähema sõiduki läbitud trajektoori punkti ning arvutame punktide vahelise kauguse. Seejärel arvutame välja kauguste keskmise ning meil tekib keskmine sõiduki kaugus rajast antud kurvis. Põhjuseks, miks leida kaugus igast rajapunktist, mitte sõiduki läbitud teekonnapunktist, on see, et rajapunktid on võrdsemate kaugustega hajutatud ja igal katsetusel samad. Seetõttu erinevate katsetuste tulemused on paremini võrreldavad.

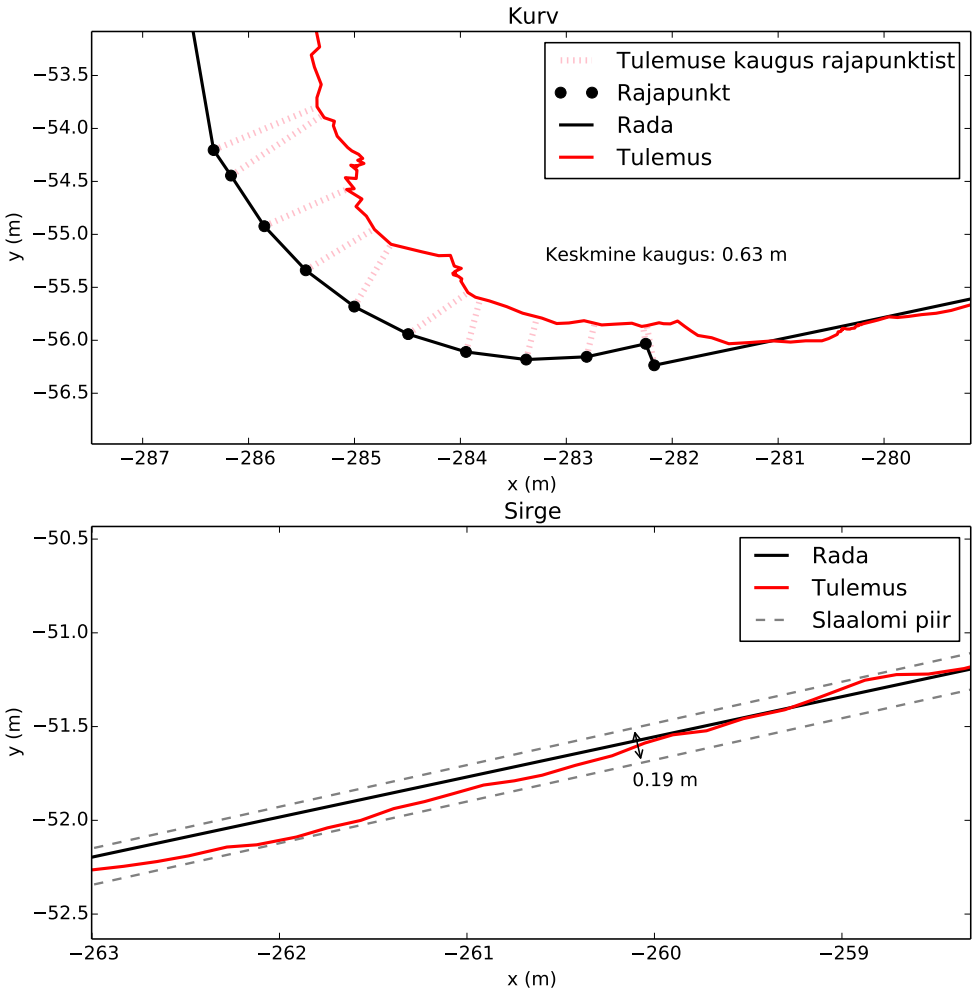
Selleks, et hinnata sõiduki slaalomit, määrame kaks paralleelset joont, mille vahel sõiduk slaalomit sõidab. Seejärel arvutame välja paralleelsete joonte kauguse ning saame tipust tippu slaalomi amplituudi.

Joonised ja analüüs



Joonis 5.1: Raja järgimise tulemus. Pure Pursuiti parameetrid: minimaalne vaatelulatus: 4,5 m, kiiruse kordaja: 1,3.

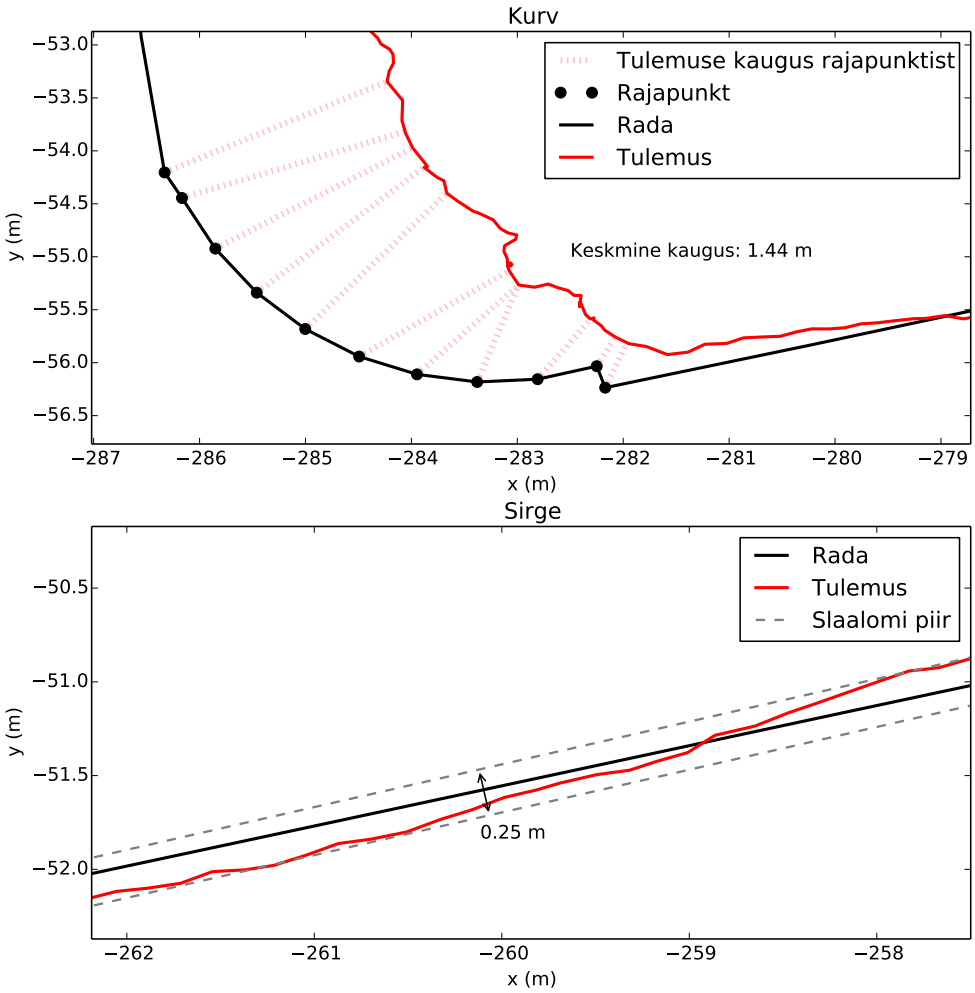
Joonisel 5.1 on näha tulemust, kus minimaalne vaateulatus on 4,5 m ning kiiruse kordaja 1,3. Sirge raja graafikult on näha, et robot sõidab ettenähtud raja kõrval. See on põhjendatud sellega, et esirataste pööramise null-punkt on ebakorrektselt kalibreeritud. See tähendab, et kui pööramisel antakse käsk, et rataste pöördenurk olgu 0 radiaani, siis roboti esirattad ei ole otse, kuigi peaks. Jooniselt on näha, et antud parameetritega on jõudlus kurvides rahuldav - kurvi lõigatakse keskmiselt 32 cm, mis on väiksem, kui sõiduki laius. Antud tulemusse tuleb suhtuda skeptiliselt, kuna pööramise null-punkt oli antud katsetamisel korrektselt kalibreerimata.



Joonis 5.2: Raja järgimise tulemus. Pure Pursuiti parameetrid: minimaalne vaatelulatus: 4,0 m, kiiruse kordaja: 3,3.

Joonisel 5.2 katsetasime väiksemat minimaalset vaateulatust, kuid oluliselt suuremat kiiruse kordajat võrreldes Joonisel 5.1 oleval katsel. Jooniselt on näha, et kurvi lõigatakse rohkem (63 cm), kuid sirgel on slaalomi tipust tippu amplituud väiksem (19 cm). Antud tulemus sirgel on ootuspärane, kuna suurendasime oluliselt kiiruse kordajat, mille tulemusel suurema kiirusega sõites valitakse suurem vaateulatus. Mis ei ole ootuspärane, on see, et kurvide lõikamist esineb 31 cm rohkem. Järelikult kiiruse kordaja on nii suur, et minimaalset vaateulatust selles kurvis ei kasutatud. Sooritus kurvis on siiski üpris rahuldav, kuna keskmine kaugus rajast on väiksem, kui sõiduki laius.

Joonisel 5.3 on näha katsetust, kus suurendasime mõlemat parameetrit veelgi, et näha, kas õnnestub vähendada slaalomi amplituudi. Jooniselt on näha, et amplituud väik-



Joonis 5.3: Raja järgimise tulemus. Pure Pursuiti parameetrid: minimaalne vaatelulatus: 6,5 m, kiiruse kordaja: 3,5.

semaks ei läinud, küll aga kurvi lõigati kaks korda rohkem, kui Joonisel 5.2. Seega on võimalik, et oleme ületanud piiri, millest vaateulatust rohkem suuremaks ei ole mõtet seada. Mida tasub selles katses veel arvestada, on see, et katse toimus tühjemate akudega ning esirataste pööramine töötas silma järgi aeglasemalt.

Raja järgimise täpsususe uurimise kokkuvõte

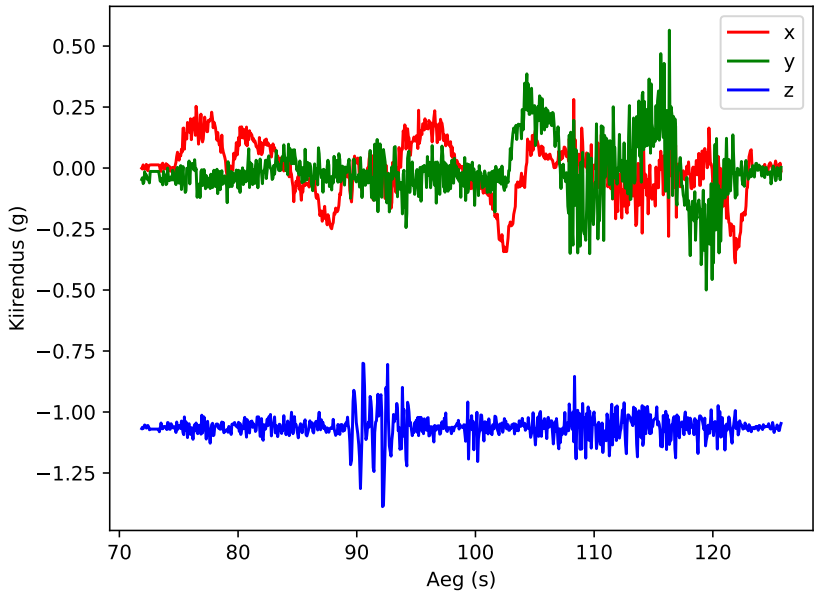
Vaadates kõiki tulemusi, saame järeldada, et vaateulatuse suurendamine tõesti vähendab slaalomi amplituudi, kuid slaalomist lahti me täielikult ei saaanud. See võib tulla parameetrite halvast valikust või roboti ebastabiilsest käitumisest (katsetuse algul pöö-

ravad rattad kiiremi, kui hiljem). Avastasime ka, et kiiruse kordajat ei tasu liiga suureks panna, kuna see võib tingida minimaalse vaateulatuse parameetri mitte kasutamist. Kiiruse kordaja tuleks valida nii, et kurvides võetakse kasutausele ikkagi minimaalne vaateulatus, et kurve lõigataks minimaalselt. Seega minimaalne vaateulatus tuleks paika seada nõnda, et kurvid oleksid läbitud rahuldava lõikamisega ning kiiruse kordaja nõnda, et sirgetel esineks minimaalse amplituudiga slaalomit.

5.2 Sõidu sujuvus

Sõidu sujuvuse uurimiseks kasutame kiirendussensorit CHR-UM6 [20]. Sensoriga saame mõõta kiirenduse andmeid kolmes suunas. Sensori maksimaalne mõõdetav kiirendus on 2 g. Et teada saada, kas see on piisav, katsetame sensorit esialgu sõiduautoga tänaval sõites.

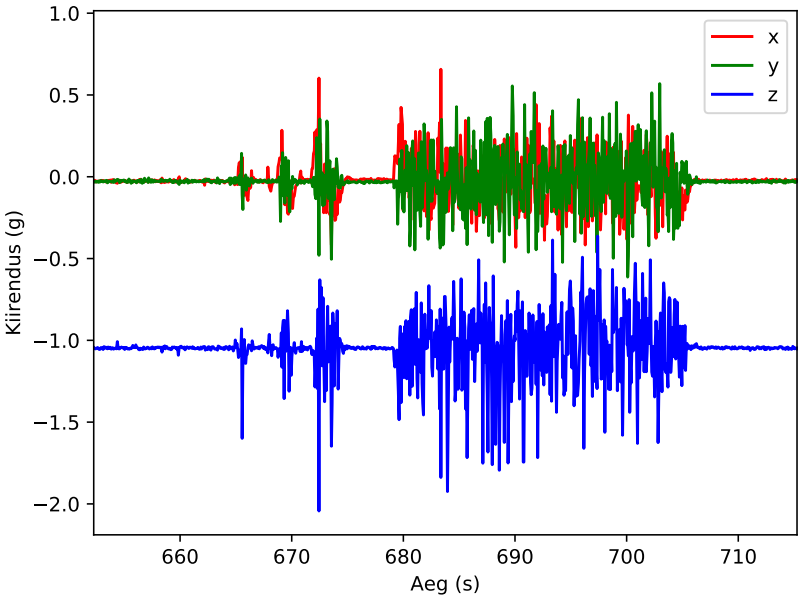
Et öelda, kas Uku sõit on sujuv, võrdleme selle kiirendusi sõiduauto omadega. Uurimise läbi viimiseks paigutame sensori katsetatava sõiduki peale nõnda, et x-, y- ja z-telg oleks õigetes suundades (x oleks suunatud ette, y kõrvale ning z üles). Samuti hoolitseme selle eest, et sensor oleks korralikult paigal ning ei liiguks. Marsruudina kasutame mõlemal sõidukil sama teekonda.



Joonis 5.4: Kiirenduste graafik sõiduautoga.

Andmed näitavad, et sõiduautoga on aru saada kurvidest, pidurdustest ning kiirusetõketest (vt Joonis 5.4). Kurvides näeme y väärtuse kõikumist ümber 0 väärtuse - paremale keerates läheb y väärtus positiivsele poolele, ning vasakpööretel negatiivsele. Sõidu kiiruse muutumisel teeme vahet x-teljel - kiiruse suurendamisel x väärtus kasvab ning pidurdamisel langeb. Z-teljel on näha gravitatsiooni mõju sensorile - selle tõttu asuvad z väärtused teistest all pool 1 g ümbruses. Z-telg näitab meile ka kiirusetõkkest üle sõitmist väärtuste muutumise kujul - sellele vastavalt muutub ka x-telg, kust on näha, et enne tõket võetakse hoogu maha ning peale tõket suurendatakse jälle kiirust. Andmetest saame järeldada, et sensor on piisavalt võimekas, et mõõta sõiduki kiirendusi tänaval liiklemise tingimustes.

Uku andmed erinevad sõiduauto omadest oluliselt - graafikult on näha, et nii pea, kui Uku liikuma hakkab, esineb kõikidel telgedel oluliselt rohkem müra, kui sõiduauto puhul (vt Joonis 5.5). Kahjuks selliste andmete pealt on meil keeruline midagi hinnata. Müra tekib tõenäoliselt sellest, et Ukul on maastikurehvid (nagu on näha Joonisel 1.1) ning jäik vedrustus.



Joonis 5.5: Kiirenduste graafik Ukuga.

Sujuvuse uurimise kokkuvõte

Kahjuks ülaltoodud andmetest ei saa me välja lugeda raja järgimise algoritmi sõidu sujuvust Uku platvormil. Tulevikus, kui testimised tulevad Mitsubishi i-Mieviga, mis sarnaneb omaduste poolest oluliselt rohkem võrreldava sõiduautoga, võib antud meetod anda tulemuse.

Kokkuvõte

Töötasime välja lahenduse, millega saame kasutada vabavaralist autonoomse sõiduki tarkvara Autoware'i, et katsetada ainult raja järgimise probleemi võimalikult väheste vahenditega. Katsetamiseks läheb vaja kolme põhikomponenti: 3D lidarit, sõidukit ning arvutit, milles jookseb Autoware. Et seda saavutada, pidime analüüsima raja järgimiseks kasutatava algoritmi Pure Pursuit sisendeid ning need Autoware'i kaudu kätte saama. Selleks on tarvis lahendada autonoomse sõiduki alamprobleem lokaliseerimine ning kirjutada juurde ROS-i sõlm, mis annab algoritmile tema tööle vastavalt läbitava raja. Lahendust valideerisime simulatsiooni keskkonnas. Seal õnnestus meil läbi mängida kõik etapid, mis meil reaalses maailmas läbi on vaja teha, et saada sõiduk autonoomselt rada järgima.

Kui simulatsioonis saime raja järgimise tööle, liikusime reaalse roboti platvormile Uku. Et Ukut juhtida, analüüsisime, kuidas me saame anda robotile liikumiskäske raja järgimise algoritmist. Seejärel realiseerisime ROS-i sõlme, mis ühendab Autoware'i robotiga ning samas oleks kontrollitav juhtmevaba puldiga Logitech Wireless Gamepad F710. Implementeerisime puldi liidese, et robotiga katsetamine oleks võimalikult mugav ning ohutu, arvestades sellega, et järgmiseks platvormiks on meil reaalne auto. Realiseeritud said ka ROS-i käivitusfailid, et katsetamine reaalse robotiga läbi Autoware'i kasutajaliidese oleks võimalikult lihtne.

Lisaks Uku platvormile, lõime ka ühenduse meie järgmise platvormi (Mitsubishi i-Mievi) jaoks. Uue platvormi ühendamiseks lõime sõlme, mis võtab ROS-ist käsud ning saadab need üle Ethernet kaabli auto juhtkontrollerisse kasutades UDP protokolli. Realiseeritud sai sõnumite saatmine bait-kujul ning edukalt testiti ka kontrolleri poolel sõnumite vastu võtmist. Kahjuks sõiduki juhtimist ei ole veel katsetatud, kuna platvorm ei olnud töö kirjutamise ajal veel piisavalt valmis.

Raja järgimise valideerimiseks tegime kolm katsetust erinevate raja järgimise algoritmi Pure Pursuit parameetritega. Üritasime likvideerida slaalomi esinemist kiiruse kordaja suurendamisega, kuid kõige suurema proovitud kiiruse kordajaga see ei andnud positiiv-

semat tulemust. Sellel on kaks potentsiaalset põhjust - parameetrid ei ole veel optimaalsed või robot ei ole piisavalt stabiilse sooritusvõimega. Üritasime hinnata ka autonoomse sõidu sujuvust kiirendussensoriga CHR-UM6. Kahjuks ei andnud see meile tulemust, kuna müra andmetes oli liiga suur. Eeldatavasti selle pärast, et Uku peal oli liiga palju vibratsiooni, mis võisid tulla maastikurehvidest ning liiga jäigast vedrustusest. Kuna sõiduauto peal olid andmed paremad - eristasime keeramisi, pidurdamisi ning kiirendamisi - siis meetod võib teistel platvormidel, mis sarnanevad rohkem sõiduautoga, anda tulemust.

Viited

[1] Heiko Pikner. Universaalne andmevahetuse infrastruktuur mobiilsele mehitamata robotile. Master's thesis, Tallinn University of Technology, 2013.

[2] Yunxiang Ye, Zhaodong Wang, Dylan Jones, Long He, Matthew E. Taylor, Geoffrey A. Hollinger, and Qin Zhang. Bin-dog: A robotic platform for bin management in orchards. *Robotics*, 6(2), 2017. ISSN 2218-6581. doi: 10.3390/robotics6020012. URL http://www.mdpi.com/2218-6581/6/2/12.

[3] R. Craig Coulter. Implementation of the pure pursuit path tracking algorithm. Technical Report CMU-RI-TR-92-01, Carnegie Mellon University, Pittsburgh, PA, January 1992.

[4] B. Paden, M. Čáp, S.Ž. Yong, D. Yershov, and E. Frazzoli. A survey of motion planning and control techniques for self-driving urban vehicles, March 2016. ISSN 2379-8858.

[5] S. Kato, E. Takeuchi, Y. Ishiguro, Y. Ninomiya, K. Takeda, and T. Hamada. An open approach to autonomous vehicles. *IEEE Micro*, 35(6):60–68, Nov 2015. ISSN 0272-1732. doi: 10.1109/MM.2015.133.

[6] Tier IV Academy, Buildin Autonomous Driving System, Autoware Hands-on Exercises, 2017. URL https://github.com/CPFL/Autoware-Manuals/blob/master/en/Autoware_TierIV_Academy_v1.1.pdf. Accessed 2018-04-22.

[7] H.Õhta, N. Akai, E. Takeuchi, S. Kato, and M. Edahiro. Pure pursuit revisited: Field testing of autonomous vehicles in urban areas. In *2016 IEEE 4th International Conference on Cyber-Physical Systems, Networks, and Applications (CPSNA)*, pages 7–12, Oct 2016. doi: 10.1109/CPSNA.2016.10.

[8] Autoware manuals, 2018. URL https://github.com/CPFL/Autoware-Manuals. Accessed 2018-05-07.

[9] Hiroki Ohta, Yamato Ando, and Shohei Fujii. Pure pursuit core, 2018. URL https://github.com/CPFL/Autoware/blob/master/ros/src/computing/planning/motion/packages/waypoint_follower/nodes/pure_pursuit/pure_pursuit_core.cpp. Accessed 2018-03-26.

[10] Martin Buehler, Karl Iagnemma, and Sanjiv Singh. *The 2005 DARPA Grand Challenge: The Great Robot Race*. Springer Publishing Company, Incorporated, 1st edition, 2007. ISBN 3540734287, 9783540734284.

[11] Martin Buehler, Karl Iagnemma, and Sanjiv Singh. *The DARPA Urban Challenge: Autonomous Vehicles in City Traffic*. Springer Publishing Company, Incorporated, 1st edition, 2009. ISBN 3642039901, 9783642039904.

[12] Sebastian Thrun, Mike Montemerlo, Hendrik Dahlkamp, David Stavens, Andrei Aron, James Diebel, Philip Fong, John Gale, Morgan Halpenny, Gabriel Hoffmann, Kenny Lau, Celia Oakley, Mark Palatucci, Vaughan Pratt, Pascal Stang, Sven Strohband, Cedric Dupont, Lars-Erik Jendrossek, Christian Koelen, Charles Markey, Carlo Rummel, Joe van Niekerk, Eric Jensen, Philippe Alessandrini, Gary Bradski, Bob Davies, Scott Ettinger, Adrian Kaehler, Ara Nefian, and Pamela Mahoney. Stanley: The robot that won the darpa grand challenge: Research articles. *J. Robot. Syst.*, 23(9):661–692, September 2006. ISSN 0741-2223. doi: 10.1002/rob.v23:9. URL http://dx.doi.org/10.1002/rob.v23:9.

[13] Hiroki Ohta, Yamato Ando, and Shohei Fujii. Lanearray message source code, 2017. URL https://github.com/CPFL/Autoware/blob/master/ros/src/msgs/autoware_msgs/msg/LaneArray.msg. Accessed 2018-04-23.

[14] Jonathan Sprinkle, Rahul Bhadani, Sam Taylor, Kennon McKeever, Alex Warren, Swati Munjal, Ashley Kang, Matt Bunting, and Sean Whitsitt. Autoware, catvehicle, 2018. URL https://github.com/CPFL/Autoware/tree/master/ros/src/system/gazebo/catvehicle. Accessed 2018-04-23.

[15] Tim Field, Jeremy Leibs, and James Bowman. rosbag, 2017. URL http://wiki.ros.org/rosbag. Accessed 2018-04-23.

[16] Hiroki Ohta, Yamato Ando, and Shohei Fujii. Controlcommandstamped message source code, 2017. URL https://github.com/CPFL/Autoware/blob/master/ros/src/msgs/autoware_msgs/msg/ControlCommandStamped.msg. Accessed 2018-04-23.

[17] Twist message, 2018. URL http://docs.ros.org/api/geometry_msgs/html/msg/Twist.html. Accessed 2018-04-23.

[18] Vector3stamped message, 2018. URL http://docs.ros.org/api/geometry_msgs/html/msg/Vector3Stamped.html. Accessed 2018-04-24.

[19] joy, 2017. URL http://wiki.ros.org/joy. Accessed 2018-04-23.

[20] Um6 orientation sensor, 2018. URL http://www.chrobotics.com/shop/orientation-sensor-um6. Accessed 2018-05-07.

Lisad

Lisa 1 - Pure Pursuit algoritmile rada saatev sõlm

```
1  #!/usr/bin/env python
2  #
3  # This node translates /lane_waypoints_array to /final_waypoints.
4  #
5  # Use this if you have recorded waypoints and you want to skip using
       the vector map. If
6  # you do not use this node, /final_waypoints will not be published
       without a vector map.
7  #
8  # See Autoware_TierIV_Academy_v1.1.pdf document (slide 53, "Path
       Planning - Structure") -
9  # https://github.com/CPFL/Autoware-Manuals/blob/master/en/
       Autoware_TierIV_Academy_v1.1.pdf
10 #
11
12 from math import sqrt
13 from math import pow
14
15 import rospy
16
17 from autoware_msgs.msg import LaneArray
18 from autoware_msgs.msg import lane
19 from geometry_msgs.msg import PoseStamped
20 from geometry_msgs.msg import Pose
21 from std_msgs.msg import String
22
23 class Point:
24     def __init__(self, pose):
25         self.x = pose.position.x
26         self.y = pose.position.y
```

```
27          self.z = pose.position.z
28
29      def distance(self, other):
30          dist_x = other.x - self.x
31          dist_y = other.y - self.y
32          dist_z = other.z - self.z
33          return sqrt(pow(dist_x, 2) + pow(dist_y, 2) + pow(dist_z, 2))
34
35  class Node:
36      def __init__(self):
37          rospy.init_node("lane_array_to_final", anonymous=True)
38          self.all_waypoints = lane()
39          self.pose = Pose()
40          self.is_pose_set = False
41          self.is_at_first_point = False
42          self.pub = rospy.Publisher("final_waypoints", lane, queue_size
    =10)
43          self.waypoints_sub = rospy.Subscriber("lane_waypoints_array",
    LaneArray, self.waypoints_callback)
44          self.pose_sub = rospy.Subscriber("current_pose", PoseStamped,
    self.pose_callback)
45          self.send_data()
46
47      def waypoints_callback(self, data):
48          rospy.loginfo("Got new waypoints.")
49          self.is_at_first_point = False
50          self.all_waypoints = data.lanes[0]
51
52      def pose_callback(self, data):
53          self.pose = data.pose
54          self.is_pose_set = True
55
56      def send_data(self):
57          rate = rospy.Rate(20)
58          while not rospy.is_shutdown():
59              try:
60                  if self.is_pose_set:
61                      self.filter_waypoints_ahead_of_pose()
62                      if len(self.all_waypoints.waypoints) <= 2:
63                          self.all_waypoints.waypoints[0].twist.twist.
    linear.x = 0.0
64                      self.pub.publish(self.all_waypoints)
65                  rate.sleep()
66              except Exception as e:
67                  rospy.loginfo("Failed to publish: %s" % e.args)
```

```
68
69      def filter_waypoints_ahead_of_pose(self):
70          index = self.get_closest_waypoint_index()
71          remaining_waypoints = self.all_waypoints.waypoints[index:]
72          self.all_waypoints.waypoints = remaining_waypoints
73          self.is_at_first_point = True
74
75      def get_closest_waypoint_index(self):
76          min_distance = float("inf")
77          min_index = 0
78          last_distance = 0
79          distance_risen_count = 0
80          vehicle_point = Point(self.pose)
81
82          for index, waypoint in enumerate(self.all_waypoints.waypoints):
83              waypoint_point = Point(waypoint.pose.pose)
84              distance = vehicle_point.distance(waypoint_point)
85
86              if distance < min_distance:
87                  min_distance = distance
88                  min_index = index
89
90              if self.is_at_first_point:
91                  if distance > last_distance:
92                      distance_risen_count = distance_risen_count + 1
93                  last_distance = distance
94                  if distance_risen_count >= 5:
95                      break
96
97          return min_index
98
99
100 def main():
101     Node()
102     rospy.spin()
103
104 if __name__ == "__main__":
105     try:
106         main()
107     except rospy.ROSInterruptException:
108         pass
```

Koodi näide 1.1: Sõlme lane_array_to_final kood.

Lisa 2 - Uku ühendussõlm

```
1 #include <ros/ros.h>
2 #include <sensor_msgs/Joy.h>
3 #include <geometry_msgs/Vector3Stamped.h>
4 #include <autoware_msgs/ControlCommandStamped.h>
5
6 #define MODE_STOP 0
7 #define MODE_AUTONOMOUS 1
8 #define MODE_JOY 2
9
10 #define MODE_NAME_STOP "STOP"
11 #define MODE_NAME_AUTONOMOUS "AUTONOMOUS"
12 #define MODE_NAME_JOY "JOY"
13 #define MODE_NAME_UNDEFINED "UNDEFINED"
14
15 #define PUBLISHING_LOOP_RATE_HZ 30
16
17 #define JOY_BUTTON_A 0
18 #define JOY_BUTTON_B 1
19 #define JOY_BUTTON_X 2
20 #define JOY_BUTTON_LEFT_BUMPER 4
21 #define JOY_BUTTON_RIGHT_BUMPER 5
22 #define JOY_STICK_LEFT_VERTICAL 1
23 #define JOY_STICK_RIGHT_HORIZONTAL 3
24 #define JOY_TRIGGER_RIGHT 5
25
26 uint mode = MODE_STOP;
27 double joy_velocity_multiplier = 0.0;
28
29 double max_joy_velocity = 1.0;
30 double max_joy_steering_angle = 1.0;
31
32 double joy_velocity = 0.0;
33 double joy_steering_angle = 0.0;
34
35 ros::Publisher uku_pub;
36
37 bool is_one_safety_button(const sensor_msgs::Joy joy)
38 {
39     return joy.buttons[JOY_BUTTON_LEFT_BUMPER] || joy.buttons[
    JOY_BUTTON_RIGHT_BUMPER];
40 }
```

```
41
42 bool are_safety_buttons(const sensor_msgs::Joy joy)
43 {
44     return joy.buttons[JOY_BUTTON_LEFT_BUMPER] && joy.buttons[
       JOY_BUTTON_RIGHT_BUMPER];
45 }
46
47 bool is_parking_button(const sensor_msgs::Joy joy)
48 {
49     return (bool)joy.buttons[JOY_BUTTON_B];
50 }
51
52 bool is_controller_button(const sensor_msgs::Joy joy)
53 {
54     return (bool)joy.buttons[JOY_BUTTON_X];
55 }
56
57 bool is_autonomous_button(const sensor_msgs::Joy joy)
58 {
59     return (bool)joy.buttons[JOY_BUTTON_A];
60 }
61
62 void publish_values(float velocity, float steering_angle)
63 {
64     geometry_msgs::Vector3Stamped vector3;
65     vector3.vector.x = velocity;
66     vector3.vector.y = 2.2;
67     vector3.vector.z = steering_angle;
68     uku_pub.publish(vector3);
69 }
70
71 void update_joy_values(const sensor_msgs::Joy joy)
72 {
73     joy_velocity = joy.axes[JOY_STICK_LEFT_VERTICAL] * max_joy_velocity
       ;
74     joy_steering_angle = joy.axes[JOY_STICK_RIGHT_HORIZONTAL] *
       max_joy_steering_angle;
75 }
76
77 void update_joy_velocity_multiplier(const sensor_msgs::Joy data)
78 {
79     const double joy_value = data.axes[JOY_TRIGGER_RIGHT];
80     if (is_one_safety_button(data))
81     {
82         joy_velocity_multiplier = fabs(-((joy_value - 1.0) / 2.0));
```

```
83      }
84      else
85      {
86          joy_velocity_multiplier = 0.0;
87      }
88  }
89
90  bool has_mode_changed(uint old_mode)
91  {
92      return mode != old_mode;
93  }
94
95  const char *get_mode_name()
96  {
97      switch (mode)
98      {
99      case MODE_STOP:
100         return MODE_NAME_STOP;
101     case MODE_JOY:
102         return MODE_NAME_JOY;
103     case MODE_AUTONOMOUS:
104         return MODE_NAME_AUTONOMOUS;
105     default:
106         return MODE_NAME_UNDEFINED;
107     }
108 }
109
110 void display_mode_info()
111 {
112     const char *mode_name = get_mode_name();
113     ROS_INFO("MODE: %s", mode_name);
114 }
115
116 void update_mode(const sensor_msgs::Joy data)
117 {
118     if (is_parking_button(data))
119     {
120         mode = MODE_STOP;
121     }
122     else if (are_safety_buttons(data))
123     {
124         if (is_controller_button(data))
125         {
126             mode = MODE_JOY;
127         }
```

```
128         else if (is_autonomous_button(data))
129         {
130             mode = MODE_AUTONOMOUS;
131         }
132     }
133 }
134
135 void joy_callback(const sensor_msgs::Joy data)
136 {
137     uint old_mode = mode;
138
139     update_mode(data);
140     update_joy_velocity_multiplier(data);
141     update_joy_values(data);
142
143     if (has_mode_changed(old_mode))
144     {
145         display_mode_info();
146         publish_values(0.0f, 0.0f);
147     }
148 }
149
150 void control_command_callback(const autoware_msgs::
      ControlCommandStamped data)
151 {
152     if (mode != MODE_AUTONOMOUS)
153         return;
154
155     const float velocity = (float)data.cmd.linear_velocity *
      joy_velocity_multiplier;
156     const float steering_angle = (float)data.cmd.steering_angle;
157     publish_values(velocity, steering_angle);
158 }
159
160 void start_publishing_loop()
161 {
162     ros::Rate loop_rate(PUBLISHING_LOOP_RATE_HZ);
163     while (ros::ok())
164     {
165         if (mode == MODE_STOP)
166         {
167             publish_values(0.0f, 0.0f);
168         }
169         else if (mode == MODE_JOY)
170         {
```

```
171             publish_values(joy_velocity, joy_steering_angle);
172         }
173         ros::spinOnce();
174         loop_rate.sleep();
175     }
176 }
177
178 int main(int argc, char **argv)
179 {
180     ros::init(argc, argv, "uku_controller");
181
182     ros::NodeHandle nh;
183     ros::NodeHandle private_nh("~");
184
185     private_nh.param("max_joy_velocity", max_joy_velocity, 1.0);
186     private_nh.param("max_joy_steering_angle", max_joy_steering_angle,
    1.0);
187
188     ros::Subscriber ctrl_cmd_sub = nh.subscribe("ctrl_cmd", 1000,
    control_command_callback);
189     ros::Subscriber joy_sub = nh.subscribe("joy", 1000, joy_callback);
190
191     uku_pub = nh.advertise<geometry_msgs::Vector3Stamped>("
    fake_ackermann_cmd", 1000);
192
193     start_publishing_loop();
194
195     return 0;
196 }
```

Koodi näide 2.1: Uku ühendussõlme kood.

```
<launch>
    <node name="joy_node" pkg="joy" type="joy_node">
        <param name="dev" type="string" value="/dev/input/js1" />
    </node>

    <node name="uku_controller" pkg="vehicle_controller" type="
    uku_controller" output="screen">
        <param name="max_joy_velocity" value="1.1" />
        <param name="max_joy_steering_angle" value="0.7" />
    </node>

    <node pkg="rosserial_python" type="serial_node.py" name="
    steering_arduino" output="screen">
        <param name="port" value="/dev/ttyACM1" />
        <param name="baud" value="57600" />
    </node>

    <node pkg="rosserial_python" type="serial_node.py" name="
    back_motors_arduino" output="screen">
        <param name="port" value="/dev/ttyACM0" />
        <param name="baud" value="57600" />
    </node>
</launch>
```

Koodi näide 2.2: Uku ühendussõlme käivitusfail.

Lisa 3 - Uku käivitusfailid

```
<launch>

    <include file="$(env ISEAUTO_RESOURCES)/tf/tf.launch"/>

    <node
        name="points_map_loader"
        pkg="map_file"
        type="points_map_loader"
        args="noupdate $(env ISEAUTO_RESOURCES)/pcd_maps/
    campus_map_zero_starting_point_binary.pcd"
        />

    <node
        pkg="map_file"
        type="vector_map_loader"
        name="vector_map_loader"
        args="$(env ISEAUTO_RESOURCES)/vectormap/campus/area.csv $(env
    ISEAUTO_RESOURCES)/vectormap/campus/crosswalk.csv $(env
    ISEAUTO_RESOURCES)/vectormap/campus/curb.csv $(env
    ISEAUTO_RESOURCES)/vectormap/campus/dtlane.csv $(env
    ISEAUTO_RESOURCES)/vectormap/campus/lane.csv $(env
    ISEAUTO_RESOURCES)/vectormap/campus/line.csv $(env
    ISEAUTO_RESOURCES)/vectormap/campus/node.csv $(env
    ISEAUTO_RESOURCES)/vectormap/campus/point.csv $(env
    ISEAUTO_RESOURCES)/vectormap/campus/roadedge.csv $(env
    ISEAUTO_RESOURCES)/vectormap/campus/whiteline.csv"
        />

</launch>
```

Koodi näide 3.1: Uku kaardi käivitusfail.

```
<launch>

    <!--           -->
    <!-- VELODYNE -->
    <!--           -->

    <include file="$(find velodyne_pointcloud)/launch/velodyne_vlp16.launch">
        <arg name="calibration" value="$(env ISEAUTO_RESOURCES)/velodyne/Velo_Iseauto.yaml" />
    </include>

</launch>
```

Koodi näide 3.2: Uku tunnetamise käivitusfail.

```
<launch>

    <!--              -->
    <!-- LOCALIZATION -->
    <!--              -->

    <node pkg="tf" type="static_transform_publisher" name="
    link1_broadcaster" args="1.18 0.0 0.98 0 0 0 1 base_link velodyne
    100" />

    <!-- Run voxel_grid_filter node, which filters the velodyne's
    scanned points. -->
    <include file="$(find points_downsampler)/launch/points_downsample.
    launch"/>

    <!-- Initialize params for ndt_matching node. -->
    <param name="localizer" type="string" value="velodyne" />

    <!-- Velodynes position relative to base_link. -->
    <param name="tf_x" type="double" value="1.18" /> <!-- Try with all
    zeros -->
    <param name="tf_y" type="double" value="0.0" />
    <param name="tf_z" type="double" value="0.98" />
    <param name="tf_roll" type="double" value="0.0" />
    <param name="tf_pitch" type="double" value="0.0" />
    <param name="tf_yaw" type="double" value="0.0" />

    <!-- Run ndt_matching node, which localizes in the point cloud. -->
    <include file="$(find ndt_localizer)/launch/ndt_matching.launch" />

    <!-- Publish configuration for ndt_matching. -->
    <node
        name="rostopic_ndt_config"
        pkg="rostopic"
        type="rostopic"
        args="pub /config/ndt autoware_msgs/ConfigNdt -- &amp;quot;
            {
                header: auto,
                init_pos_gnss: 0,
                x: 0.0,
                y: 0.0,
                z: 0,
                roll: 0,
                pitch: 0,
```

```
40                 yaw: 0,
41                 use_predict_pose: 1,
42                 error_threshold: 1,
43                 resolution: 1,
44                 step_size: 0.1,
45                 trans_epsilon: 0.01,
46                 max_iterations: 30
47             }
48             &amp;quot;"
49         />
50 
51 </launch>
```

Koodi näide 3.3: Uku lokaliseerimise käivitusfail.

```
<launch>

    <!--                  -->
    <!-- MISSION PLANNING -->
    <!--                  -->

    <!-- Load the node, that measures pose and velocity from the point cloud. -->
    <include file="$(find autoware_connector)/launch/vel_pose_connect.launch">
        <arg name="topic_pose_stamped" value="/ndt_pose" />
        <arg name="topic_twist_stamped" value="/estimate_twist" />
    </include>

    <!-- Load waypoints. -->
    <include file="$(find waypoint_maker)/launch/waypoint_loader.launch">
        <arg
            name="multi_lane_csv"
            value="$(env ISEAUTO_RESOURCES)/waypoints/tty/new_waypoints.csv"
            />
    </include>

</launch>
```

Koodi näide 3.4: Uku missiooni planeerimise käivitusfail.

```
<launch>

    <!--                  -->
    <!-- MOTION PLANNING -->
    <!--                  -->

    <!-- Load pure_pursuit node, which produces movement messages
    according to waypoints. -->
    <include file="$(find waypoint_follower_iseauto)/launch/
    pure_pursuit.launch" />

    <!-- Publish configuration for pure_pursuit node. -->
    <param name="vehicle_info/wheel_base" type="double" value="1.3" />
    <node
        name="rostopic_pure_pursuit_config"
        pkg="rostopic"
        type="rostopic"
        args="pub /config/waypoint_follower autoware_msgs/
    ConfigWaypointFollower -- &quot;
            {
                header: auto,
                param_flag: 0,
                velocity: 5.0,
                lookahead_distance: 6.5,
                lookahead_ratio: 3.8,
                minimum_lookahead_distance: 6.5,
                displacement_threshold: 0.0,
                relative_angle_threshold: 0.0
            }
            &quot;"
        />

    <!-- Load a node, that makes loaded waypoints into final_waypoints.
     -->
    <node
        name="lane_array_to_final"
        pkg="waypoint_maker_iseauto"
        type="lane_array_to_final.py"
        />

</launch>
```

Koodi näide 3.5: Uku liikumise planeerimise käivitusfail.

Lisa 4 - Mitsubishi i-Mievi ühendussõlm

```
1  #include "ros/ros.h"
2  #include "autoware_msgs/ControlCommandStamped.h"
3  #include "iseauto_msgs/VehicleControllerModeStamped.h"
4  #include "network_address.h"
5  #include "udp_client_server.h"
6  #include "sending_data.h"
7
8  #define FLOAT_SIZE 4
9
10 SendingData sending_data;
11 NetworkAddress controller_address;
12
13 void control_command_callback(const autoware_msgs::
       ControlCommandStamped data)
14 {
15     const float velocity = (float)data.cmd.linear_velocity;
16     const float steering_angle = (float)data.cmd.steering_angle;
17     sending_data.set_velocity(velocity);
18     sending_data.set_steering_angle(steering_angle);
19 }
20
21 void mode_callback(const iseauto_msgs::VehicleControllerModeStamped
       data)
22 {
23     uint8_t mode = data.mode.mode;
24     if (mode > 0 && mode < 4)
25     {
26         sending_data.set_safety_brake(mode);
27     }
28 }
29
30 void start_udp_client()
31 {
32     const char *ip = controller_address.ip.c_str();
33     const unsigned int port = (unsigned int)controller_address.port;
34     udp_client_server::udp_client client = udp_client_server::
       udp_client(ip, port);
35
36     ros::Rate loop_rate(100);
37     while (ros::ok())
38     {
```

```
39          unsigned char *msg = sending_data.get_udp_bytes();
40          client.send(msg, SendingData::UDP_MSG_LENGTH);
41          free(msg);
42
43          ros::spinOnce();
44          loop_rate.sleep();
45      }
46  }
47
48  int main(int argc, char **argv)
49  {
50      if (sizeof(float) != FLOAT_SIZE)
51      {
52          ROS_ERROR("Float size is not %d.", FLOAT_SIZE);
53          return 1;
54      }
55
56      ros::init(argc, argv, "vehicle_controller");
57      ros::NodeHandle nh;
58      ros::NodeHandle private_nh("~");
59
60      const std::string default_controller_ip = "127.0.0.1";
61      const int default_controller_port = 4444;
62
63      private_nh.param<std::string>("controller_ip", controller_address.
        ip, default_controller_ip);
64      private_nh.param("controller_port", controller_address.port,
        default_controller_port);
65
66      const std::string default_ctrl_cmd_topic = "ctrl_cmd";
67      std::string ctrl_cmd_topic;
68      private_nh.param<std::string>("control_command_topic",
        ctrl_cmd_topic, default_ctrl_cmd_topic);
69      ros::Subscriber ctrl_cmd_sub = nh.subscribe(ctrl_cmd_topic, 1000,
        control_command_callback);
70
71      ros::Subscriber mode_sub = nh.subscribe("vehicle_controller/mode",
        1000, mode_callback);
72
73      start_udp_client();
74      return 0;
75  }
```

Koodi näide 4.1: Mievi ühendussõlme juur.

```
#ifndef SENDING_DATA
#define SENDING_DATA

class SendingData
{
    static const uint8_t VELOCITY_ID = 0x10;
    static const uint8_t STEERING_ANGLE_ID = 0x20;
    static const uint8_t SAFETY_BRAKE_ID = 0x30;
    static const uint8_t SAFETY_BRAKE_STOP = 1;
    static const uint8_t SAFETY_BRAKE_AUTONOMOUS = 2;
    static const uint8_t SAFETY_BRAKE_REMOTE = 3;

    float velocity;
    float steering_angle;
    uint8_t safety_brake;

    static float reverse_float_bytes(float input);

  public:
    static const size_t UDP_MSG_LENGTH = 12;

    SendingData() : velocity(0.0f), steering_angle(0.0f), safety_brake(
    SAFETY_BRAKE_STOP) {}

    unsigned char *get_udp_bytes();

    void set_velocity(const float velocity)
    {
        this->velocity = velocity;
    }

    void set_steering_angle(const float steering_angle)
    {
        this->steering_angle = steering_angle;
    }

    void set_safety_brake(const uint8_t safety_brake)
    {
        this->safety_brake = safety_brake;
    }
};

#endif
```

Koodi näide 4.2: Mievi ühendussõlme saadetavate andmete klassi päis.

```cpp
#include <cstring>
#include <stdlib.h>
#include <unistd.h>
#include <stdint.h>
#include "sending_data.h"

float SendingData::reverse_float_bytes(const float input)
{
    float reversed = 0.0f;

    uint8_t *from = (uint8_t *) &input;
    uint8_t *to = (uint8_t *) &reversed;

    *(to + 0) = *(from + 3);
    *(to + 1) = *(from + 2);
    *(to + 2) = *(from + 1);
    *(to + 3) = *(from + 0);

    return reversed;
}

unsigned char *SendingData::get_udp_bytes()
{
    unsigned char *msg = (unsigned char *)malloc(UDP_MSG_LENGTH);
    if (msg == NULL)
        exit(1);

    unsigned char *cursor = msg;

    memcpy(cursor, &VELOCITY_ID, sizeof(uint8_t));
    cursor += sizeof(uint8_t);

    const float velocity_bytes_reversed = SendingData::
    reverse_float_bytes(this->velocity);
    memcpy(cursor, &velocity_bytes_reversed, sizeof(float));
    cursor += sizeof(float);

    memcpy(cursor, &STEERING_ANGLE_ID, sizeof(uint8_t));
    cursor += sizeof(uint8_t);

    const float steering_angle_bytes_reversed = SendingData::
    reverse_float_bytes(this->steering_angle);
    memcpy(cursor, &steering_angle_bytes_reversed, sizeof(float));
    cursor += sizeof(float);
```

```
43
44      memcpy(cursor, &SAFETY_BRAKE_ID, sizeof(uint8_t));
45      cursor += sizeof(uint8_t);
46
47      memcpy(cursor, &this->safety_brake, sizeof(uint8_t));
48
49      return msg;
50  }
```

Koodi näide 4.3: Mievi ühendussõlme saadetavate andmete klassi kood.

```
1  #ifndef NETWORK_ADDRESS
2  #define NETWORK_ADDRESS
3
4  struct NetworkAddress
5  {
6      std::string ip;
7      int port;
8  };
9
10 #endif
```

Koodi näide 4.4: Mievi ühendussõlme päis võrguaadressi hoidmiseks.

```
1  #include <ros/ros.h>
2
3  #include <sensor_msgs/Joy.h>
4  #include <autoware_msgs/ControlCommandStamped.h>
5  #include <iseauto_msgs/VehicleControllerMode.h>
6  #include <iseauto_msgs/VehicleControllerModeStamped.h>
7
8  #include <stdlib.h>
9
10 #define JOY_BUTTON_A 0
11 #define JOY_BUTTON_B 1
12 #define JOY_BUTTON_X 2
13 #define JOY_BUTTON_LEFT_BUMPER 4
14 #define JOY_BUTTON_RIGHT_BUMPER 5
15 #define JOY_TRIGGER_RIGHT 5
16
17 ros::Publisher ctrl_cmd_pub;
18 ros::Publisher mode_pub;
19
20 double joy_multiplier = 0.0;
21
22 bool is_one_safety_button(const sensor_msgs::Joy joy)
23 {
24     return joy.buttons[JOY_BUTTON_LEFT_BUMPER] || joy.buttons[
       JOY_BUTTON_RIGHT_BUMPER];
25 }
26
27 bool are_safety_buttons(const sensor_msgs::Joy joy)
28 {
29     return joy.buttons[JOY_BUTTON_LEFT_BUMPER] && joy.buttons[
       JOY_BUTTON_RIGHT_BUMPER];
30 }
31
32 bool is_parking_button(const sensor_msgs::Joy joy)
33 {
34     return (bool)joy.buttons[JOY_BUTTON_B];
35 }
36
37 bool is_controller_button(const sensor_msgs::Joy joy)
38 {
39     return (bool)joy.buttons[JOY_BUTTON_X];
40 }
41
42 bool is_autonomous_button(const sensor_msgs::Joy joy)
```

```
43 {
44     return (bool)joy.buttons[JOY_BUTTON_A];
45 }
46
47 void publish_mode(uint8_t mode)
48 {
49     iseauto_msgs::VehicleControllerModeStamped vcm;
50     vcm.mode.mode = mode;
51     mode_pub.publish(vcm);
52 }
53
54 void set_joy_multiplier(const sensor_msgs::Joy data)
55 {
56     const double joy_value = data.axes[JOY_TRIGGER_RIGHT];
57     if (is_one_safety_button(data))
58     {
59         joy_multiplier = fabs(-((joy_value - 1.0) / 2.0));
60     }
61     else
62     {
63         joy_multiplier = 0.0;
64     }
65 }
66
67 void set_mode(const sensor_msgs::Joy data)
68 {
69     if (is_parking_button(data))
70     {
71         publish_mode(iseauto_msgs::VehicleControllerMode::PARK);
72     }
73     else if (are_safety_buttons(data))
74     {
75         if (is_controller_button(data))
76         {
77             publish_mode(iseauto_msgs::VehicleControllerMode::
    CONTROLLER);
78         }
79         else if (is_autonomous_button(data))
80         {
81             publish_mode(iseauto_msgs::VehicleControllerMode::
    AUTONOMOUS);
82         }
83     }
84 }
85
```

```
86 void joy_callback(const sensor_msgs::Joy data)
87 {
88     set_joy_multiplier(data);
89     set_mode(data);
90 }
91
92 void ctrl_cmd_callback(autoware_msgs::ControlCommandStamped data)
93 {
94     data.cmd.linear_velocity = data.cmd.linear_velocity *
      joy_multiplier;
95     ctrl_cmd_pub.publish(data);
96 }
97
98 int main(int argc, char **argv)
99 {
100     ros::init(argc, argv, "vehicle_controller_joy");
101     ros::NodeHandle nh;
102
103     ros::Subscriber joy_sub = nh.subscribe<sensor_msgs::Joy>("joy", 10,
       joy_callback);
104     ros::Subscriber ctrl_sub = nh.subscribe<autoware_msgs::
      ControlCommandStamped>("ctrl_cmd", 10, ctrl_cmd_callback);
105
106     ctrl_cmd_pub = nh.advertise<autoware_msgs::ControlCommandStamped>("
      ctrl_cmd_from_joy", 10);
107     mode_pub = nh.advertise<iseauto_msgs::VehicleControllerModeStamped
      >("vehicle_controller/mode", 10);
108
109     ros::spin();
110
111     return 0;
112 }
```

Koodi näide 4.5: Mievi ühendussõlme jaoks puldiga juhtimise sõlme kood.

```
<launch>
    <node name="vehicle_controller" pkg="vehicle_controller" type="
    vehicle_controller">
        <param name="control_command_topic" value="ctrl_cmd" />
        <param name="controller_ip" value="127.0.0.1" />
        <param name="controller_port" value="4444" />
    </node>
</launch>
```

Koodi näide 4.6: Mievi ühendussõlme käivitusfail.

```
<launch>
    <node name="joy_node" pkg="joy" type="joy_node" />

    <node name="vehicle_controller_joy" pkg="vehicle_controller" type="
    vehicle_controller_joy" />

    <node name="vehicle_controller" pkg="vehicle_controller" type="
    vehicle_controller">
        <param name="control_command_topic" value="ctrl_cmd_from_joy" /
    >
        <param name="controller_ip" value="127.0.0.1" />
        <param name="controller_port" value="4444" />
    </node>
</launch>
```

Koodi näide 4.7: Mievi ühendussõlme käivitus koos puldiga.

```
<launch>
    <node name="rqt_gui" pkg="rqt_gui" type="rqt_gui" />

    <node name="rqt_to_ctrl_cmd" pkg="rqt_steering_test" type="
    rqt_to_ctrl_cmd.py" />

    <include file="$(find vehicle_controller)/launch/vehicle_controller
    .launch" />
</launch>
```

Koodi näide 4.8: Mievi ühendussõlme käivitus koos testimisprogrammiga RQT.

```
<launch>
    <node name="rqt_gui" pkg="rqt_gui" type="rqt_gui" />

    <node name="rqt_to_ctrl_cmd" pkg="rqt_steering_test" type="
    rqt_to_ctrl_cmd.py" />

    <include file="$(find vehicle_controller)/launch/
    vehicle_controller_with_joy.launch" />
</launch>
```

Koodi näide 4.9: Mievi ühendussõlme käivitus koos puldi ja testimisprogrammiga RQT.

```
int8 PARK=1
int8 AUTONOMOUS=2
int8 CONTROLLER=3
int8 mode
```

Koodi näide 4.10: Sõidurežiimi ROS-i sõnumi tüüp.

```
Header header
VehicleControllerMode mode
```

Koodi näide 4.11: Sõidurežiimi ROS-i sõnumi tüüp koos ajatempliga.