TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond

Lisanne Siniväli 194103IAIB

RAJA LEIDMINE FS TEAM TALLINNA ISEJUHTIVALE VORMELILE

Bakalaureusetöö

Juhendaja: Gert Kanter
Filosoofiadoktor
(informaatika)

Tallinn 2022

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.

Autor: Lisanne Siniväli

30.05.2022

Annotatsioon

Selleks, et tulevased insenerid saaksid teoorias õpitut ka praktikas kasutada, loodi tudengivormeli sari Formula Student. Tallinna Tehnika Kõrgkool ja Tallinna Tehnika ülikool on koostöös loonud tiimi, Formula Student Team Tallinn, mille ülesanne on iga aasta luua uusi vormeleid ja arendada edasi isejuhtiva vormeli süsteemi. Selleks, et jõuda isejuhtiva vormeli tippkiirusteni on vaja süsteemile genereerida rada, mida masin saaks jälgida.

Töö ülesanne on raja planeerimine Formula Student Team Tallinna isejuhtivale autole. Raja planeerimine toimub Formula Student võistluste reeglite järgi. Planeerimine ei tohi toetuda koonuste värvidele, peab olema võimeline rada uuendama 10 korda sekundis ja raja pikkus peab olema koonuste maksimaalne tuvastamise kaugus, ehk 20 meetrit.

Töös leitakse, et parim algoritm selleks on RRT*. Algoritmi testimisel ja võrdlemisel eelmise aasta lahendusega jõuti järeldusele, et ilma koonuste värvideta on raja planeerimine palju usaldusväärsem ja stabiilsem. Selleks, et raja leidmise kindlust tõsta jäeti sisse koonusta värvidega arvestamine ainult siis kui värvis ollakse 100%.

Lõputöö on kirjutatud eesti keeles ning sisaldab teksti 28 leheküljel, 6 peatükki, 30 joonist, 1 tabelit.

Abstract

Path Planning for FS Team Tallinn Autonomous Formula Student Vehicle

Formula Student competition is made for students all over the world to develop and build formula cars. In recent years the driverless class was also added to the competition. Driverless cars require fast and reliable path planning to reach the vehicle limits and drive even faster than the cars with drivers.

The aim of this project is to find the best path planning method for Formula Student Team Tallinn driverless car. The rules of the track layout are given by the Formula Student competition. The aim is to not rely on the cones colors to plan the path. One wrong color assumption and all paths are wrong. Path planning must be able to publish 10 times a second and 20 meters ahead. The path must be smooth, as it makes it easier to follow for the MPC.

Surveying different path planning methods the RRT* algorithm was most suitable for the task. RRT* algorithm generates random nodes on search space and forms branches from them. After that the best branch is chosen and from that the center line is found. Splines are placed between waypoints, so that the path is smoother to follow.

The RRT* algorithm is evaluated in simulators and also on the driverless car. MPC is able to follow it smoothly. The path planning is compared to last year's solution, which relies on cones colors. The RRT* algorithm is more reliable and does not break if the cone color is misdetected. The cone colors are only used when the color is 100% certain, to improve the path reliability in edge cases.

The thesis is in Estonian and contains 28 pages of text, 6 chapters, 30 figures, 1 table.

Lühendite ja mõistete sõnastik

FS	Formula Student
IMU	Inertial Measurement Unit
PRM	Probabilistic RoadMap
ROS	Robot Operating System
RRT	Rapidly-exploring random tree
SLAM	Simultaneous localization and mapping

Sisukord

1 Sissejuhatus 10
2 Taust 11
2.1 Motivatsioon 11
2.2 Formula Student 11
2.3 Isejuhtiv vormel 12
3 Analüüs 14
3.1 Path planning 14
3.2 Näidisevõtmisel põhinevad algoritmid 16
4 Rapidly-exploring Random Tree* (RRT*) 19
4.1 RRT* algoritm 19
4.2 Tööriistad raja planeerimiseks 21
4.3 RRT* tudengivormeli süsteemis 21
4.4 Splainid 25
4.5 Optimeerimine 26
5 Tulemused 28
5.1 RRT* esinenud vead vormelisüsteemis 28
5.2 Võrdlemine eelmise aasta lahendusega 35
5.3 Testimine 38
6 Kokkuvõte 39
Lisa 1 – Lihtlitsents lõputöö reprodutseerimiseks ja lõputöö üldsusele kättesaadavaks tegemiseks 42
Lisa 2 – Rqt graaf vormeli süsteemist 43
Lisa 3 – Sõnumite kujud 45
Lisa 4 – Formula Student East võistluse salvestus 47

Jooniste loetelu

Joonis 1. Isejuhtiv vormel FS East võistlustel 2021 Ungaris 12
Joonis 2. Sensorite ja arvutikasti asetus isejuhtival vormelil. 13
Joonis 3. Isejuhtiva vormeli komponentide skeem. 14
Joonis 4. Rajasõidu tingimused. 15
Joonis 5. PRM algoritmi loodud ühendused [14]. 18
Joonis 6. RRT ja RRT* algoritmi erinevus [15]. 19
Joonis 7. RRT* põhi algoritm. 20
Joonis 8. Uue tipu genereerimise algoritm. 20
Joonis 9. RRT* algoritm tudengivormeli süsteemis 23
Joonis 10. Delaunay triangulatsioon (lillalt) olemasolevate koonustega (must). 24
Joonis 11. Raja planeerimise algoritm. 25
Joonis 12. Catmull-Rom splines. 26
Joonis 13. Ideaaltrajektoori ja keskjoone vahe [24]. 27
Joonis 14. RRT* ehitamas ennast ümber koonuste ja mööda rada. 29
Joonis 15. Valesti valitud parim haru. 30
Joonis 16. Õigesti valitud parim haru. 30
Joonis 17. Hõre RRT* puu. 31
Joonis 18. Tihe RRT* puu. 32
Joonis 19. Kaaristamisel sisendiks tulevad ebaühtlased koonused. 33
Joonis 20. SLAM-ilt tulevad hõredad punktid ja sellest põhjustatud keskjoone katkemine tänu sellele katki. 34
Joonis 21. Vigadeta kogu ringi keskjoon. 34
Joonis 22. Gantt diagramm hooaja tööst 35
Joonis 23. Eelmise aasta keskjoone leidmise lahendus valesti 36
Joonis 24. RRT* algoritmi ei mõjuta valet värvi koonused. 37
Joonis 25. Rqt graaf esimene pool 43
Joonis 26. Rqt graaf teine pool, raja planeerimise osa näidatud rohelisega 44
Joonis 27. */odom* sõnum 45
Joonis 28. */map* sõnum 45

Joonis 29. /center sõnum 46
Joonis 30. FS East võistluse salvestus 47

Tabelite loetelu

Tabel 1. Näidisevõtmisel põhinevad lokaalsete algoritmide võrdlus 17

1 Sissejuhatus

Selle töö eesmärgiks on sõiduraja leidmine Formula Student Team Tallinna isejuhtivale vormelile, millega minnakse võistlema Formula Student võistlusele. FS Team Tallinn on tudengiorganisatsioon, mille eesmärk on ehitada ja arendada vormeleid. Võistlused toimuvad üle terve Euroopa ja nende eesmärgiks on anda noortele inseneridele võimalus koolis õpitut praktikas rakendada ja seda profesionaalidele kaitsta. Tiimis on kaks meeskonda: elektri- ja isejuhtiva vormeli meeskond. Isejuhtiv vormel peab hakkama saama tundmatul rajal ja läbima selle võimalikult kiiresti. Raja kohta on teada, et sinised koonused on vasakul, kollased paremal ja raja laius on minimaalselt 3 meetrit. Rada on maksimaalselt 700 meetrit pikk. Raja tuvastamiseks kasutatakse sensoreid milleks on lidar, kaamerad ja IMU. Raja planeerimine saab sisendiks koonuste asukohad, koonuse värvi tõenäosuse ja auto asukoha.

Raja leidmise ülesanne on leida takistustevaba tee mida vormel saaks järgida. Tähtis on, et rada ei läheks kordagi valesti ja oleks MPC-l võimalikult lihtne järgida. Algoritm peab olema efektiivne, et seda saaks uuendada 10 korda sekundis. Lisaks ei tohi raja leidmine sõltuda koonuste värvidest. Võib juhtuda, et koonuse värvi tuvastatkse valesti ja selle vea tõttu ei tohi vormel rajalt välja sõita.

Töös tutvustatakse lähemalt tudengivormelit ja isejuhtiva vormeli süsteemi. Selleks, et leida parim algoritm võrreldakse erinevaid meetodeid kuidas vormelile rada leida ja valitakse välja süsteemile ja eesmärgile sobivaim. Järgnevalt näidatakse kuidas algoritmi implementeeritakse ja mida selleks kasutatakse. Kolmandas peatükis näidatakse millised vead tulid algoritmil välja tulid, kui seda testiti vormeli peal ja kuidas need parandati. Lõpuks võrreldakse eelmiste ja selle aasta raja leidmise algoritmide vahel.

2 Taust

2.1 Motivatsioon

Peamine motivatsioon on kasutada selle uurimistöö tulemusi Formula Student Team Tallinna isejuhtival vormelil ja võistelda Formula Student võistlustel Euroopas. Team Tallinn arendab isejuhtivat autot neljandat hooaega ja tänavu tahetakse jõuda kiiruselt järgi tiimi teisele vormelile - juhiga vormelile. Selleks, et soovitud tulemus saada, peab raja plaanimine toimuma sagedusel 10 hertsi ehk 10 korda sekundis, koonuste värvist sõltumatu ja vigu vältiv.

2.2 Formula Student

Formula Student Team Tallinn [1] on tudengiorganisatsioon, mis tegutseb juba 15 hooaega. 2006. aastal alustati sisepõlemismootoriga vormeli ehitamist ja 2008. aastal mindi juba esimese vormeliga võistlema. Sellest hetkest on ehitatud iga aasta uus vormel ja 2012. aastal jõuti sisepõlemismootoriga maailma tippu. Uute eesmärkide püüdlemiseks mindi üle elektrimootoriga vormelitele ja maailma tippu naasti aastal 2017. Aastal 2021 saavutati elektrivormeliga maailma edetabelis 5. koht. 2019. aastal hakati paralleelselt elektrivormelile arendama ka isejuhtivat vormelit. Isejuhtiv vormel on 2018. aasta elektrivormel, millel iga aasta arendatakse edasi selle peal olevat süsteemi.

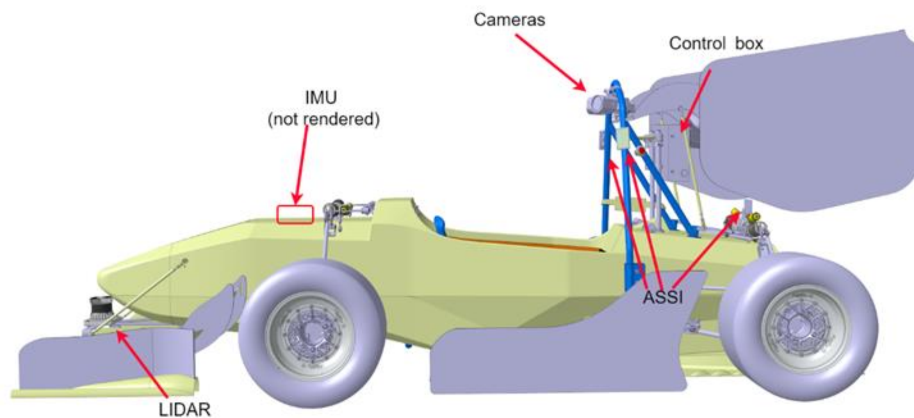
Formula Student [2] võistlussari on inseneridele mõeldud võistlussari, mis toimub üle terve maailma. Võistlustega antakse inseneridele platvorm, kus saadakse oma teoreetilisi teadmisi proovile panna ka päris elus. Võistlused on jaotatud staatilisteks ja dünaamilisteks aladeks. Staatilisel aladel peavad insenerid kaitsma oma lahenduse hinnastamist, disaini jne oma ala spetsialistidele. Enne sõitma asumist peavad vormelid läbi erinevad tehnilised kontrollid, kus kohtunikud kontrollivad, et kõik auto juures oleks reeglitepärane ja ohutu. Dünaamilistel aladel rakendatakse kogu teooria, disain ja arvutused praktikas. Vormelid osalevad kiirendusel, kaheksasõidus, aja- ja pikamaasõidus. Tippkiirused vormelitel jäävad 100-120 km/h juurde, et vältida ohtlikke

olukordi. Punkte antakse kõikide alade eest ja võidab tiim, kellel on lõpuks kõige rohkem punkte. Elektri- ja isejuhtiv vormel (Joonis 1) võistlevad küll erinevates kategooriates, aga võistlusalad on neil üsna sarnased.

Joonis 1. Isejuhtiv vormel FS East võistlustel 2021 Ungaris

2.3 Isejuhtiv vormel

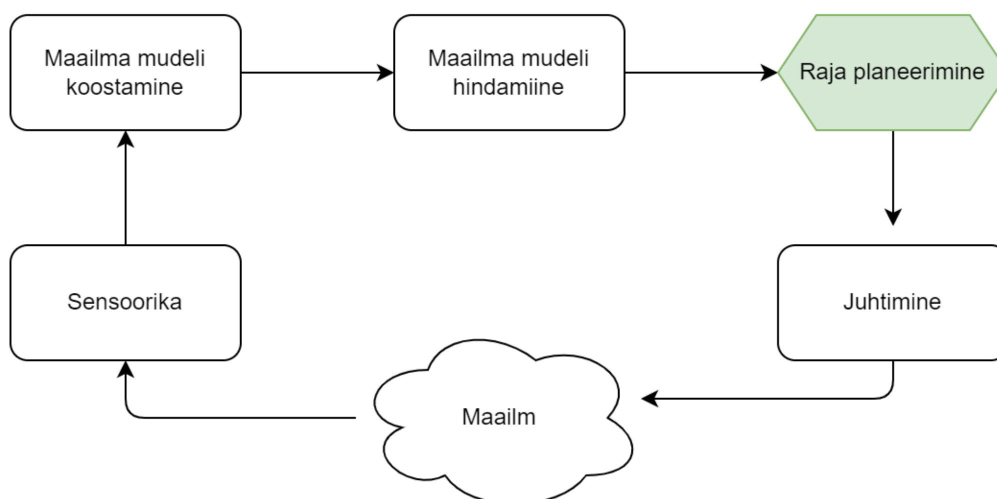
Isejuhtiva vormeli peale on lisatud sensorid lidar, kaamerad ja IMU (Joonis 2). Lisaks on vormeli tagatiiva alla, peatoe külge kinnitatud arvutikast. Võrreldes 2018. aasta vormeliga on ümber tehtud roolisüsteem ning pidurisüsteem. Roolilati külge on kinnitatud mootor, mis roolikontrolleri juhtimisel võimaldab autol ennast ise keerata. Pidurisüsteemile on juurde lisatud pneumosüsteem, mis on vajalik autonoomseks hädapidurdamiseks.



Joonis 2. Sensorite ja arvutikasti asetus isejuhtival vormelil.

Vormel asub sõitma tundmatus maailmas, mille info võtavad esmalt vastu sensorid. Lidari ja kaamerate algoritmide ülesanne on saadud andmete pealt tuvastada koonused, nende täpne asukoht maailmas ja värv. Sealt edasi hakkab tööle SLAM, mille ülesandeks on koonuste ja IMU info põhjal kokku panna maailma mudel.

Raja planeerimine asub isejuhtiva vormeli komponentide süsteemis maailma mudeli hindamise ja juhtimise vahepeal (Joonis 3). Planeerijale tulevad sisendina auto asukoht ja koonused, mille vahele arvutatakse rada. Sealt edasi lähevad raja punktid juhtimisalgoritmile, mis annab auto roolile ja pedaalidele juhtimiskäsud.



Joonis 3. Isejuhtiva vormeli komponentide skeem.

3 Analüüs

3.1 Path planning

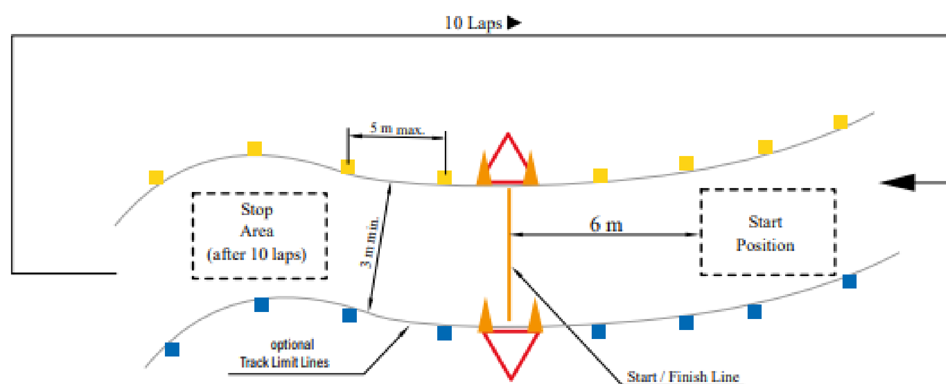
Selleks, et leida parim viis raja leidmiseks, pidi anaüüsima erinevaid algoritme. Path planning ehk raja planeerimine on NP-keerukusega [3] probleem, mille ülesanne on leida kuidas süsteem jõuaks hetkesest asukohast sihtpunktini. Probleemi keerukus tõuseb koos vabadusastmetega [4]. Vabadusastmed näitavad mitme sõltumatu muutujaga saab süsteemi kirjeldada. Mida rohkem on muutujaid, seda keerulisem on punktmassi liikumist süsteemis kirjeldada. Rada valitakse erinevate kriteeriumite alusel, nagu näiteks pikkus, turvalisus või ka optimaalsus. Tihti arvestatakse raja leidmisel erinevate kitsendustega korraga, näiteks kui on vaja leida kõige lühemat teed rajal mille suudaks mudel soovitud kiirusel läbida. Robootikas on raja plaanimine vajalik, et robotit saaks juhtida. Tee antakse ette koordinaatpunktidena. Mida tihedamini on punkte, seda täpsemalt saab robot etteantud teed jälgida.

Autonoomse sõidukile raja plaanimine on süsteemi üks tähtsamaid, aga samas ka keerukamaid osasid [5]. Rada peab olema ohutu, ökonoomne ja arvestama sõiduki

võimetega, nagu näiteks pöörderaadius ja olema ka ühtlane, et sõitmine oleks võimalikult kiire ja ohutu. Sõidukile raja planeerimine peab toimuma ka kõrge sagedusega, sest sõiduk liigub suure kiirusega ja keskkond ümber muutub pidevalt.

Raja leidmine saab olla nii lokaalne kui ka globaalne. Globaalsel raja planeerimisel teatakse ette suures osas olemasolevat keskkonda milles robot hakkab sõitma. Rada arvutatakse välja enne kui robot hakkab oma ülesannet täitma [6]. Plaanimine võib võtta rohkem aega ja olla tänu sellele täpsem, aga see ei arvesta kui keskkonnas peaks midagi muutuma. Lokaalne raja plaanimine toimub tavaliselt keskkonnas, mis on süsteemile tundmatu ja muutuv. Kuna Formula Student võistlustel on meile ette antud rada tundmatu, siis on esimese ringi raja leidmine samuti lokaalne. Algoritm peab olema suuteline uue trajektoori välja arvutama 10 korda sekundis, sest sama kiiresti uueneb kaardistamine.

Formula Student võistlustel on kinnine 500 - 700 meetrine rada [7], millel on sinised koonused vasakul ja kollased koonused paremal (Joonis 4). Raja minimaalne laius on 3 meetrit. Koonused on asetatud rajele maksimaalselt 5 meetrise vahega. Vormel suudab koonused ära tunda umbes 20 meetrit ette, aga värv ei pruugi olla õige. Süsteem peab leidma viisi kuidas ilma koonuseid puutumata läbida rada võimalikult kiiresti.



Joonis 4. Rajasõidu tingimused.

Path planning on vormeli takistusvaba tee leidmine alguspunktist lõpp punkti. Trajektoori leidmisel arvestatakse juurde vormeli matemaatiline mudel, et oleks olemas ka kiirendused ja sellest tulenevad jõud [6]. Raja optimeerimisel leitakse algsest teest ideaaltrajektoor. Planeerimine toimub lokaalselt, sest vormel näeb ette maksimaalselt paar kurvi ja rohkem raja kohta ei teata.

Esimesel ringil on vormelil olevale süsteemile maailm tema ümber täiesti tundmatu. Esmatähtis on leida tee, mis oleks takistusvaba. Kaardistamise käigus saadakse kätte koonuste asukohad, mille vahel toimub plaanimine. Plaanitud rada saab lugeda ka keskjooneks. Keskjoon on raja keskpunkte läbiv joon. Rada on minimaalselt 3 meetrit lai ja optimeerimis- või dünaamika vea korral riskitakse 3 sekundilise trahvi ajaga, mis määratakse iga kord kui sõidetakse koonusele otsa. Kuna ei ole täpselt teada kuhu suunas peab vormel koonuste vahel liikuma ja ei saa lootma jääda koonuste värvide õigele tuvastamisele, siis ei saa anda ka sihtpunkti kuhu trajektoor minema peaks. Raja planeerimiseks tuleb välja töötada algoritm, mis ei otsi otseselt kõige lühemat teed, vaid genereeritakse erinevaid teid vaadeldavas plaanimisaknas ja sealt võetakse tee, mis sobib kõige rohkem keskjoone tingimustega.

Varasemates uurimustes tulevad välja kolm põhilist meetodit kuidas rada leida: geomeetriline (ingl *geometry based*), graafikupõhine (ingl *graph based*) ja näidisevõtmisel põhinev (ingl *sampling based*). Geomeetriline raja leidmine, nagu näiteks splainide kasutamine kurvides on lihtne, loogiline ja kergest implementeeritav lahendus lihtsasse tingimustesse. Kuigi geomeetrilised lahendused on mugavad, siis jäävad nad tihti keerulistes kohtades hätta ja miinused ei kaalu üle plusse. [8]

Graafipõhine meetod otsib parimat teed alguse ja sihtpunkti vahel. Näiteks Dijkstra algoritm [9] otsib teed kahe punkti vahel, vaadates läbi kõik tippudest võimalikud teed teistesse tippudesse. Liikumine tippudes toimub nii kaua kuni jõutakse sihtpunktini. Lõpuks leitakse kõige lühem tippude vaheline tee. Teine populaarne algorithm on A* [10] mille tööpõhimõte põhineb eesõigusega järjekorral, ehk algoritmis valitakse enne tipud, mis on lähemal sihtpunktile. Graafipõhine meetod on väga kiire ja hea lahendus lühima tee leidmiseks. Kuna pole võimalust algoritmile sihtpunkti ette anda ja tähtsam on uuritava ala läbivaatus, siis samuti ei sobi ka graafi põhised meetodid.

3.2 Näidisevõtmisel põhinevad algoritmid

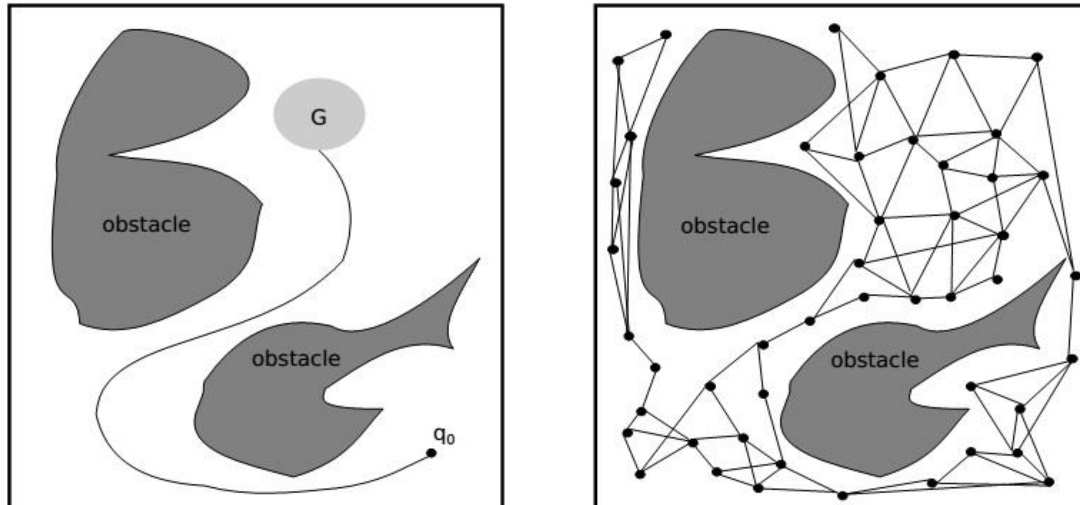
Selleks, et leida parim viis vaadeldava välja uurimiseks, võrreldi näidisevõtmisel põhinevaid lokaalseid algoritme PRM [11], RRT [12] ja RRT* [13] (Tabel 1). Need algoritmid ei vaja kindlat ühte sihtmärki kuhu suunas liikuda, vaid otsivad juhuslikult läbi

vaadeldava piirkonna. Tänu sellele saavad algoritmid töötada ka reaalajas kus teekond kuhu robot peab suunduma ei ole veel kindel ja suudavad hakkama saada pideva keskkonna uuenemisega.

	RRT	RRT*	PRM
Hea	On efektiivne ka maailma uuendamisel	Puu moodustub sirgelt ja sobivalt keskjoone leidmiseks	Ühe uuritud maailma juurest saab valida erinevaid radu efektiivselt
Halb	Otsitavatest radadest on raske vormelile sirget rada teha		Pole efektiivne raja uuendamisel

Tabel 1. Näidisevõtmisel põhinevad lokaalsete algoritmide võrdlus

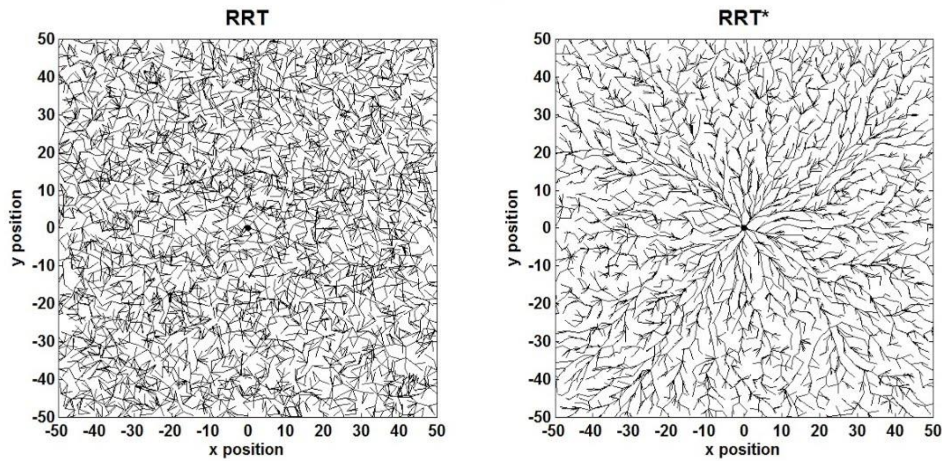
PRM-i tööpõhimõte on luua juhuslikke tippe ja nende vahele ühendusi *n* korda. Algoritm loob uue tipu ja kontrollib, ega see ei puutu kokku ühegi takistusega. Olles vabas ruumis vaadatakse üle tipu lähimad naabrid ja kas saab nendega takistus vabalt ühendusi teha. Selliselt tehakse ühendused paljude lähedal olevate tippudega. Olles *n* korda tegevust korranud on PRM loonud ühendused üle terve vaadeldava piirkonna. Selle põhjal on hea kasutada näiteks A* algoritmi ja leida olemasolevatest ühendustest kõige kiirem tee sihtpunktini. PRM loob ühendused üle terve vaadeldava piirkonna (Joonis 5) ja võimaldab leida erinevaid teid algpunktist sihtpunkti. Iga uue iteratsiooniga luuakse erinevate tippude vahel palju ühendusi ja see muudab PRM-i iga olukorra uuendamise järel ebaefektiivseks.



Joonis 5. PRM algoritmi loodud ühendused [14].

RRT tööpõhimõte on sarnaselt PRM-ile juhuslike tippude loomine vaadeldavas ruumis. Lisades uut tippu otsitakse talle teine lähim tipp ja ühendatakse need. Alati kontrollitakse ka, et tipp ei puutuks kokku takistusega. Erinevalt PRM-ile luuakse ühendus ainult ühe lähima tipuga. Tänu sellele on RRT efektiivsus tunduvalt parem ja sobiv rohkem vormeli raja leidmise probleemiga.

RRT* ainuke erinevus võrreldes RRT-ga on tipu lisamisel suunatakse see kuhugi eesmärgi poole (Joonis 6). Ehk ei otsita vaadeldavat ruumi juhuslikult läbi, vaid iga uus tipp on suunatud sihtpunkti suunas ja tänu sellele moodustavad rajad, mida saab hästi kasutada ka vormeli sõidutrajektoori leidmiseks. Joonisel 6 on näha, et samade tippude arvuga(*Nodes number:4999*) on RRT ja RRT* algoritmi tulemused täiesti erinevad.



Joonis 6. RRT ja RRT* algoritmi erinevus [15].

RRT* valiti sobivaks, sest sellel algoritmil oli kõige parem efektiivsus ja ka puu enda struktuur sobib selleks, et sealt edasi leida keskjoont. Puu iga tipp liigub koonuste suunas, mis on vormeli ees ja tänu sellele on kõik harud sõitmiseks kõlblikud. Edasi on vaja leida ainult haru, mis oleks kõige lähemal keskjoonele.

4 Rapidly-exploring Random Tree* (RRT*)

4.1 RRT* algoritm

RRT* põhialgoritmi saab näha Joonis 7 juures. RRT*-i esimene tipp asetatakse puu alguspunkti ehk nullpunkti. Uus tipp genereeritakse juhuslikult iga iteratsiooniga eesmärgi positsiooni juhuslikku lähedusse. *X(random)* ümber leitakse lähim tipp *X(close)* meetodiga *ClosestNeighbour*. Peale seda hakatakse kasvatama lähimat tippu *X(close)* uusima tipu suunas *X(random)* meetodiga *TreeConstrained* (Joonis 8). Seal leitakse, mis

suunda jääb *X(random)* *X(close)* ja sellest sõltuvalt lisatakse uus tipp *X(new)*, millele antakse kindle pikkus *X(close)* ja nurk sõltuvalt sellest mis suunas *X(random)* asus.

```
Algoritm 1: RRT* põhi algoritm

start = carPosition();

tree.insert(start);

for (i=0; i<n; i++){

                X(random) = RandomNode();
                X(close) = ClosestNeighbour(X(random));
                X(new) = TreeConstrained(X(Random), X(close));
                if (!X(new) in tree and IsCollisionFree(X(new))){
                                tree.insert(X(tree));
                }

}

return tree;
```

Joonis 7. RRT* põhi algoritm.

```
Algoritm 2: RRT* TreeConstrained algoritm (X(close), X(random))

expandAngle = constant;

 theta = atan2(X(random).y - X(close).y, X(random).x - X(close).x);

 angle = theta - X(close).yaw;

 angleChange = fmod(angle + PI, 2 * PI)) - PI;

   if (angleChange > degree_30) {
      angleChange = expandAngle;
   } else if (angleChange >= -degree_30) {
      angleChange = 0;
   } else {
      angleChange = -expandAngle;
   }

X(new) = X(close);
X(new).yaw += angleChange;
X(new).x += expandDistance * cos(X(new).yaw);
X(new).y += expandDistance * sin(X(new).yaw);
X(new).cost += angleChange;
X(new).parent = X(close);


return X(new);
```

Joonis 8. Uue tipu genereerimise algoritm.

Peale *X(new)* genereerimist, vaadatakse üle ega sellist tippu juba ei eksisteeri ja kontrollitakse kokkupõrkeid *IsCollisionFree* meetodiga. Olles sobiv, lisatakse uus tipp puu juurde koos oma arvutatud kuluga. Protsessi korratakse etteantud n korda.

4.2 Tööriistad raja planeerimiseks

Python ja C++ on kõige tihedamini kasutatavad programmeerimiskeeled robootikas [16]. Keelt Python õpitakse sagedasti kõige esimesena, sest selle struktuurist on võrdlemisi lihtne aru saada. Tänu sellele eelistatakse seda ka robootikas, sest koodi kirjutamine toimub kiiresti. Vähem peab keskenduma sellele kuidas keel ise töötab ja rohkem rakenduse eesmärgile endale. Pythonil jääb puudu aga jõudlusest, mis on autonoomsetel sõidukitel esmatähtis. C++ on jõudluse poolest kõige parem programmeerimiskeel [17]. Kuna trajektoori kood peab jooksma 10 Hz-ga ja kogu ülejäänud süsteem on üles ehitatud C++ keelele, siis valiti C++ ka raja leidmise keeleks.

ROS on robootika arendamiseks välja töötatud avatud lähtekoodiga tarkvara. Tudengivormelis kasutatakse seda Linux Ubuntu peal. ROS võimaldab robootikat virtuaalselt arendada. ROS-i kasutati algselt ülikoolides, aga aina rohkem kasutatakse seda ka ärimaailmas. [18]. Südamik(*roscore*) vahendab näiteks sensori sõnumeid kuulutajana (*publisher*) ja seda võtab vastu kood tellijana (*publisher*). Sellele toetudes saab luua erinevaid rubriike (*topic*), tänu millele juhitakse kogu süsteemi. Tudengivormelil sai ROS-i kasutada juba koodi testimise faasis, kus kasutati AMZi loodud simulaatorit [19]. Simulaatoris on olemas vormeli URDF (ingl *Unified Robot Description Format*), milles on ära kirjeldatud vormeli füüsiliste osade omavahelised seosed. Lisaks on loodud Gazebo keskkonnas vormeli rada. Seal sai testida algoritmide toimimist juba enne päris vormeliga sõitma minemist. Hetkel toimib kogu tudengivormeli isejuhtiv süsteem ROS-i peal.

4.3 RRT* tudengivormeli süsteemis

Inspiratsiooni valitud lahendusele leiti *Formula Student Team of TUHH* "e-gnition Hamburg" magistritöö [20] tulemusest. Tiimis olev liige kirjutas raja leidmise koodi ja sinna juurde ka MPC, et rada läbida. Kogu kood on kirjutatud Pythonis ja testimine toimus simulaatoris. Käesoleva töö autor sai sellelt tiimilt inspiratsiooni, hakati kirjutama

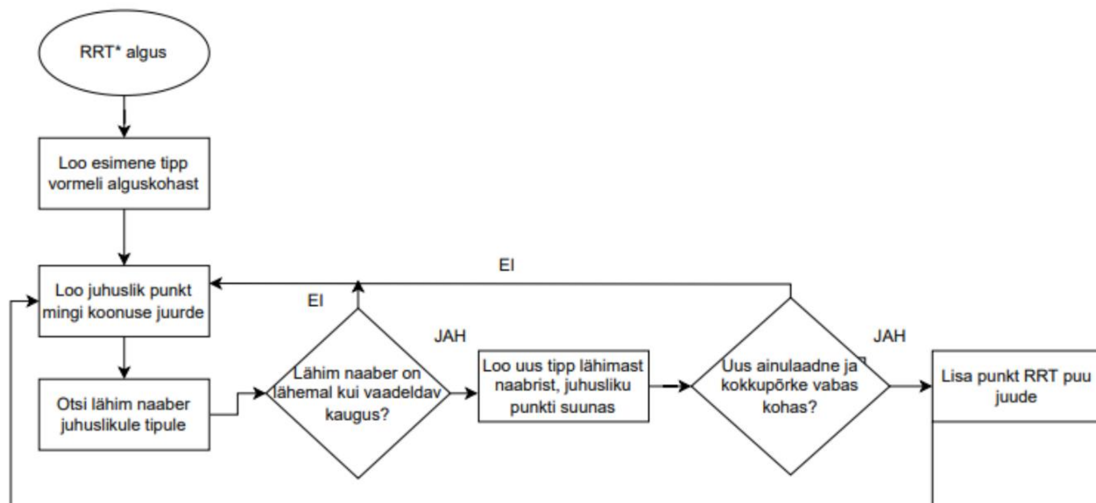
koodi, mis sobiks kokku FS Team Tallinna süsteemiga ja saaks testida vormeliga päris rajal.

Koodi sisendiks on kaardistamisest tulevad koonused koos koordinaatide ning koonuse värvi tõenäosusega (Lisa 3, Joonis 2) ja sellele lisaks odomeetria (Lisa 3, Joonis 1), kust saadakse auto asukoht ja sõidusuund. Selleks on loodud eraldi sõnum *MapOdom*, mis sisaldab mõlemat, et igal uuel kaardistamisel oleks juures ka auto selle hetke asukoht. Peale sisendi saamist võetakse auto asukohast 20 meetrit otse olevad koonused, sest see on hetkel maksimaalne kaugus millel kaardistamise ringil on võimalik koonuseid tuvastada..

Koonuste võtmise juures arvestatakse ka auto suunaga, et jätta välja taga olevad koonused. Selleks kasutatakse 2D rotatsioonimaatriksit (Valem 1), et vaadata kas koonused jäävad auto sõidusuuna suhtes sobivalt. Välja valitud koonuste vahel hakatakse planeerima RRT puud.

$$Rotation\ Matrix = \begin{bmatrix} cos\theta & -sin\theta \\ sin\theta & cos\theta \end{bmatrix} \tag{1}$$

RRT* esimeseks tipuks võetakse auto asukoht. Puu loomisel genereeritakse juhuslikke tippe koonuste lähedusse, et puu ehitaks ennast mööda rada või selle juurde. Selleks võetakse juhuslik koonus ja luuakse punkt selle lähedusse. Peale punkti genereerimist otsitakse välja lähim tipp sellele punktile. Kontrollimiseks vaadatakse üle ega leitud tipp ei ole 20 meetri kaugusel, sest siis ei saaks temast edasi enam puud kasvatada. Kui tipp on piisavalt lähedal, siis luuakse puule uus tipp, mis liigub varem genereeritud punkti suunas. RRT* puu sujuvaks liikumiseks on uus tipp vanast alati meeter edasi. Kraadide suhtes saab liikuda uus tipp kas otse või konstantse arvu võrra paremale või vasakule määratud punkti suunas Iga tipu juurde arvestatakse ka hind, mis näitab puu pikkust selle tipuni. Peale uue tipu loomist kontrollitakse ega ei ole kokkupõrkeid koonustega ja nende puudumisel lisatakse ta RRT* puu juurde. Plokkskeemil (Joonis 9) on näidatud kõik sammud.



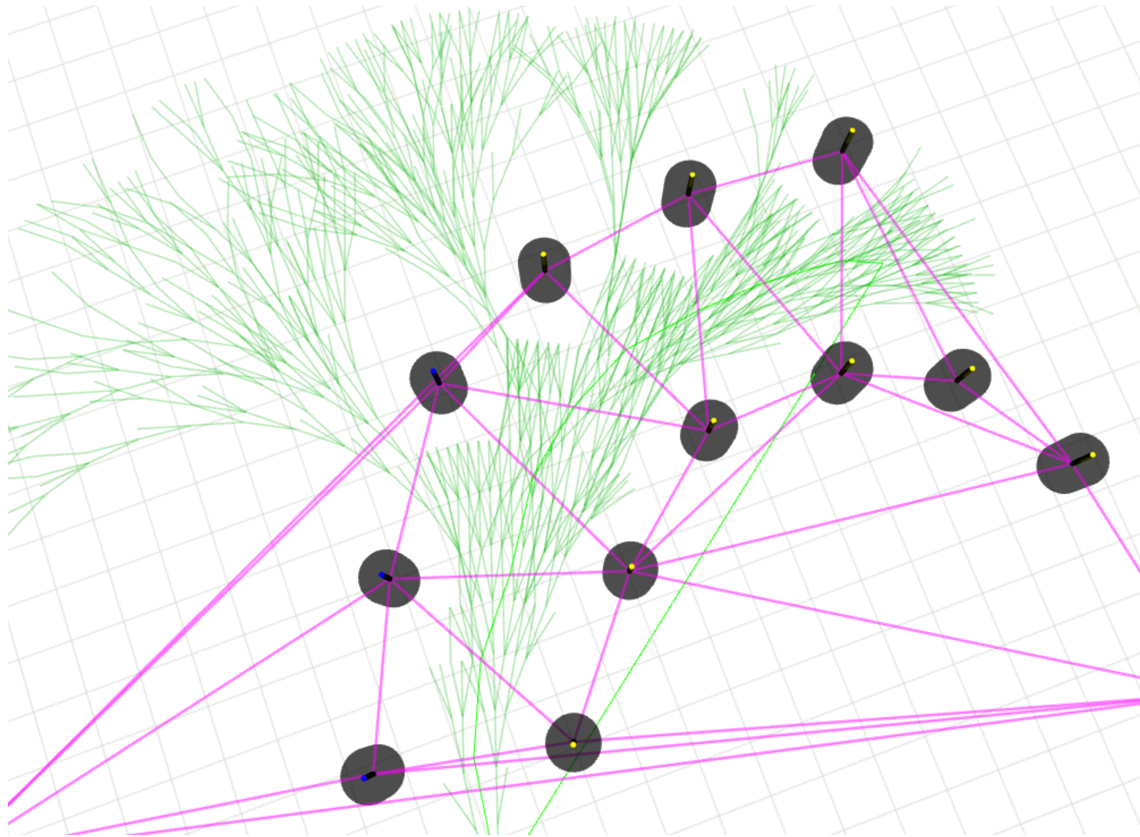
Joonis 9. RRT* algoritm tudengivormeli süsteemis

Peale puu loomist valitakse välja parim haru. Käiakse läbi iga haru kõik tipud ja antakse punkte tipule siis, kui temast on koonuseid nii paremal kui ka vasakul pool. Lisaks vaadatakse haru kõige kaugema tipu pikkust, punkt antakse iga tipu eest harus. Vältides juhuslike anomaaliate teket, on olemas ka filtreerimine, mis võrdleb uut parimat haru eelmisega ja kui uus on liiga erinev, siis jäetakse alles vana haru. Kui uus ja vana haru pole kolm tsüklit järjest sarnased olnud valitakse uueks haruks parim haru. Selle koodi osa lõpuks on salvestatud parim haru, mis teoreetiliselt jookseb raja keskelt läbi.

Eelnevalt välja valitud koonuste vahel tehakse ka Delaunay triangulatsioon. Selle keerukus *n* punktide hulgas ja *d* dimensioonide kõige rohkem $O(n^{\frac{d}{2}})$. [21] Olemasolevate punktide vahel tehakse kolmnurgad nii, et nende sees ei oleks ühtegi punkti ja tipud ühtivad kolmnurkade otstega. Selleks tehakse punktidest läbivad ringjooned ja võetakse arvesse ainult need, millel on alla 4 punkti. Sobivate ringide punktid ühendatakse kolmnurkadeks. Peale triangulatsiooni sooritamist on kõik olemasolevad punktid ühendatud ja nende vahel tehtud kolmnurkade küljed.

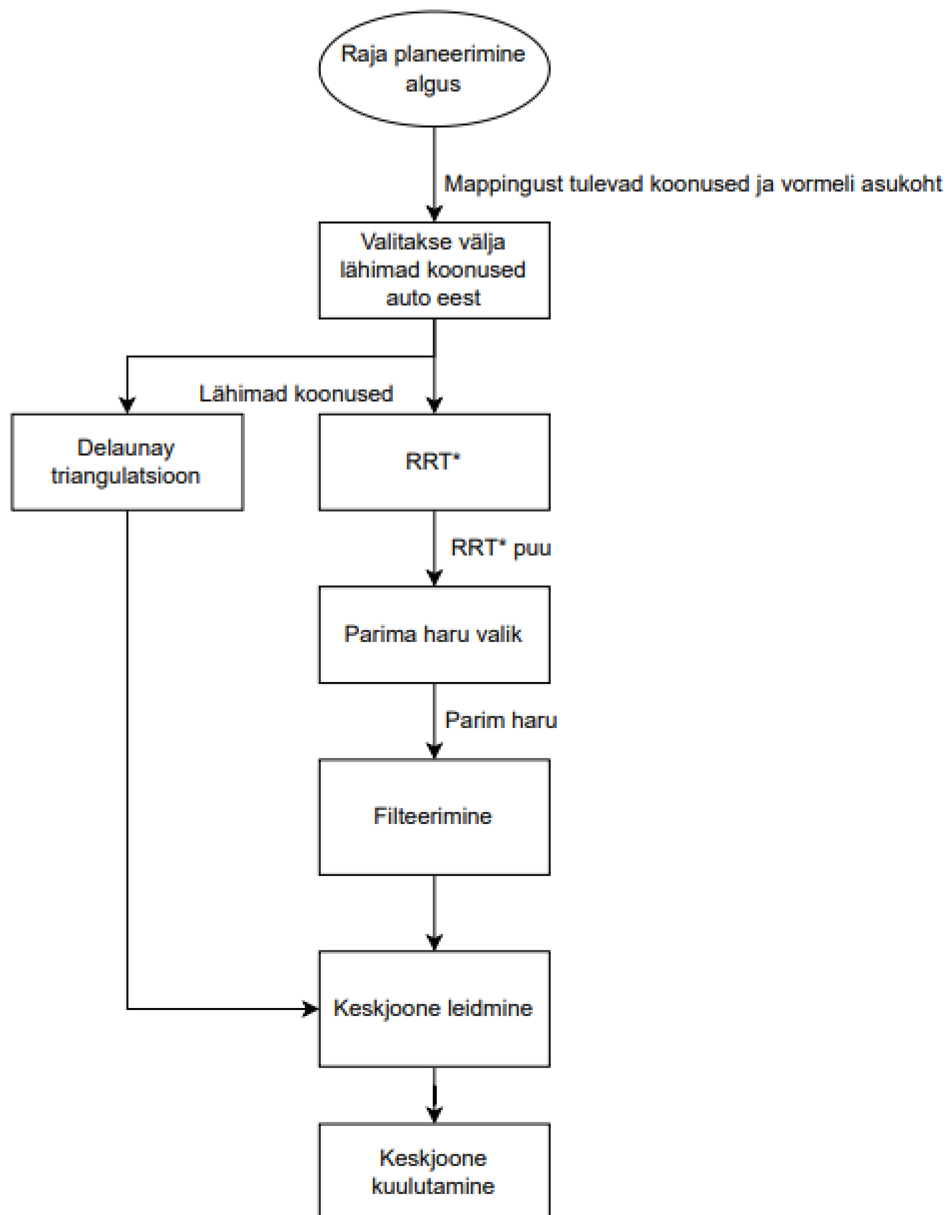
Lisades kokku parim haru ja Delaunay triangulatsioon (Joonis 10), vaadatakse, kus ühtivad RRT* haru punktid ja triangulatsiooni küljed. Kolmnurga küljel võetakse keskpunkt ja sealt saadakse keskjoon. Keskjoon on ROS-i geomeetriline sõnum

PolygonStamped, mis koosneb keskjoone x, y koordinaatidest, mida kuulutatakse välja MPC-le *"/center"* (Lisa 3, Joonis 3) rubriigina. (Joonis 11).



Joonis 10. Delaunay triangulatsioon (lillalt) olemasolevate koonustega (must).

MPC-d kasutatakse selleks, et jälgida planeeritavat rada võimalikult täpselt [22]. Selleks, et vormel suudaks sõita rajal, peab talle andma vajamineva roolinurga ja kiiruse. MPC suudab hakkama saada piirangutega ja mittelineaarsete süsteemidega, ehk see teeb ta ideaalseks kasutuseks vormeli kontrollimisel. Lahendama hakatakse optimeerimise probleemi. Eesmärk on minimeerida auto mööda minekut planeeritud rajast suurima võimaliku kiiruse juures. MPCl kasutatakse dünaamilist mudelid.

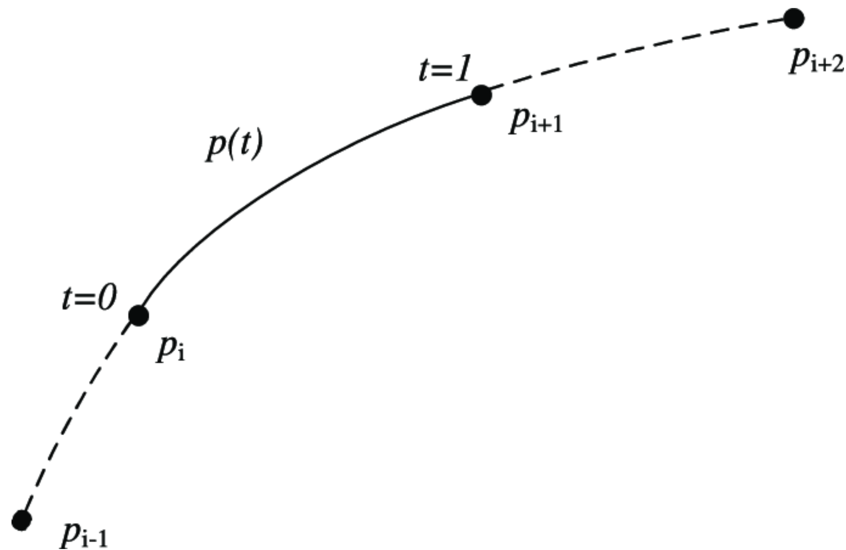


Joonis 11. Raja planeerimise algoritm.

4.4 Splainid

Keskjoon koosneb punktidest (x, y koordinaatidega). Selleks, et MPC-l oleks rada lihtsam jälgida on nende vahele on tõmmatud sirge joon, mis koosneb omaette paljudest punktidest. Sellise lahendusega on aga MPC väga ebaühtlane, sest üleminek ühelt suunalt teisele toimub järsku. Sõites kuni 120 km/h võib selline järsk pööre olla põhjus välja sõitmiseks. Selleks, et üleminek oleks sujuv, tuleb keskjoone punktide vahele teha splainid. Täpsemalt kasutati selleks Catmull-Rom splaine [23] (Joonis 12). Splaini

arvutatakse kasutades nelja järjestikust punkti. Kahe keskmise punkti vahele luuakse splain ja äärmised kaks määravad splaini kalde ja suuna. Kohta splainil märgitakse tähega t, mis asub 0 ja 1 vahepeal. Splaini tihedus määrab kui täpselt saab MPC seda järgida.

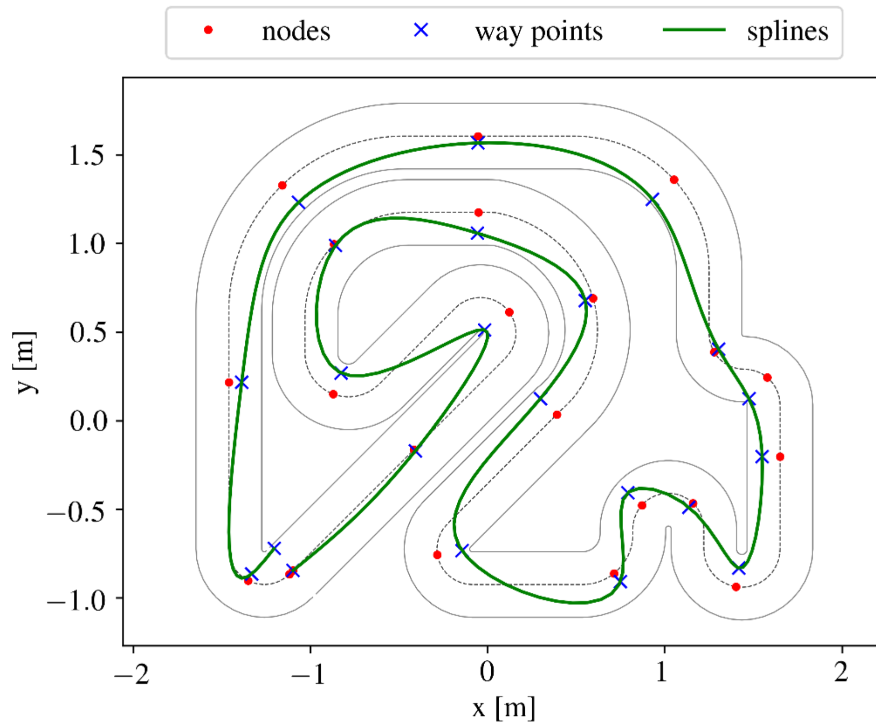


Joonis 12. Catmull-Rom splines.

Splaini moodustamine punktide vahele toimub hetkel ainult peale kaardistamise ringi, sest seal hakkab MPC sõitma vormeli võimete piiril ja sujuv rada on vajalik. Kaardistamise ringil on MPC kiirus kuskil 10 m/s ehk 36 km/h ja seal ei sega raja ebaühtlused auto kiirust ja stabiilsust.

4.5 Optimeerimine

Olemasolevat keskjoont saaks ka optimeerida, sest on olemas terve raja piirid. Selle tulemusena saaksime raja ideaaltrajektoori. Ideaaltrajektoor on kõige kiirem tee ette antud raja läbimiseks. Lihtsustatult saab ideaaltrajektoori võtta ka kui kõige sirgemat teed raja läbimiseks. See tähendab, et vormel hoiab alles võimalikult palju kiirust kurvi läbides on näha vahet keskjoonel (Joonisel 14 halliga tähistatud) ja ideaaltrajektooril (Joonisel 13 rohelisega tähistatud). Kuna tudengivormeli võistluste rada on kitsas võrreldes autoga, siis oleks mõtet keskjoont optimeerida ainult siis kui lokaliseerimine on väga täpne. Muidu võib auto endaga kaasa võtta mõne koonuse ja üldine skoor on halvem kui ilma optimeerimiseta. Sellele põhinedes otsustati Team Tallinna isejuhtival vormelil keskjoone optimeerimine vahele jätta.



Joonis 13. Ideaaltrajektoori ja keskjoone vahe [24].

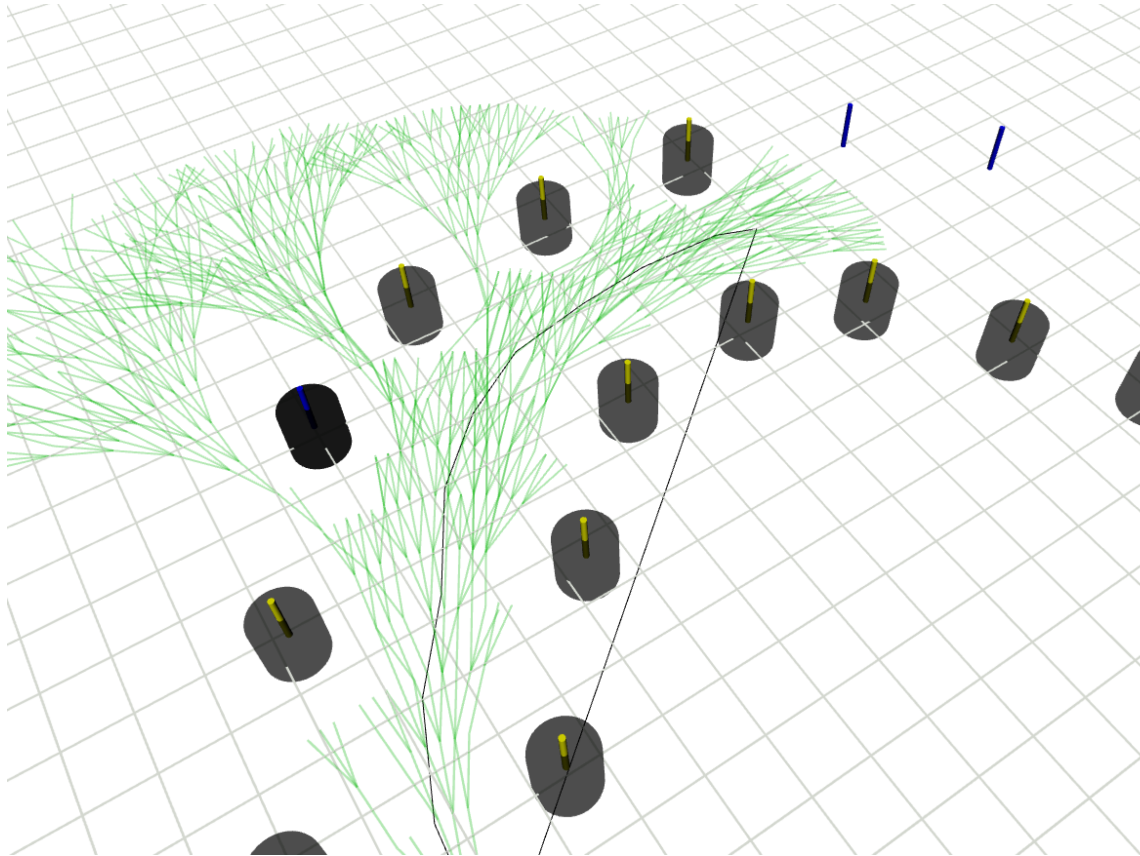
Tulevikus kui MPC ja lokaliseerimine on saanud piisavalt täpseks on kindlasti mõistlik raja optimeerimine. Vormel võib võita õige trajektooriga sama ringi pealt paar sekundit. Optimeerimise võimalusi on erinevaid. Lihtsamad muudavad raja läbimist võimalikult lühikeseks. Keerukamad algoritmid arvestavad aga ka vormeli enda füüsikalisi omadusi ja kasutatakse ära optimeerimis ülesannet. Ideaaltrajektoori leidmine on võimalik ka lisada MPC juurde. Arvestades raja piire ja auto mudelit tuleb MPC-l leida kõige kiirem viis kuidas kurvi läbida. Tuleb arvestada, et selline lahendus toimub lokaalselt ja igas ajahetkes peab toimuma palju arvutusi, mis võib süsteemi teha liiga aeglaseks.

5 Tulemused

5.1 RRT* esinenud vead vormelisüsteemis

Peale koodi kirjutamist saadi aru, et kuigi kood töötab simulaatoris, siis eelnevatel testidel salvestatud info peal see ei toimi ja pidi palju muudatusi sisse viima.

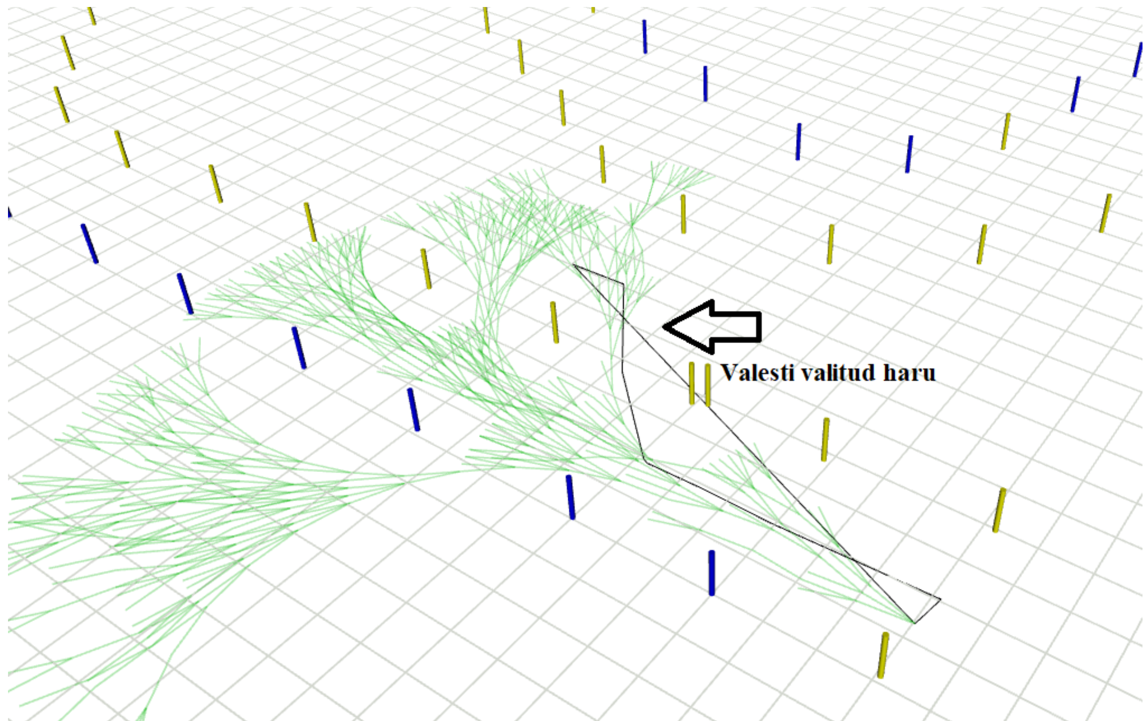
Esimesed probleemid tulid välja RRT* enda loomisel. Luues RRT* puud pannakse koonuste lähedal juhuslikkesse kohtadesse punkte, mis ühendatakse lähedal olevatega. Koonuste juurde sellepärast, et puu hakkaks liikuma rajaga samas suunas. Punkti ei lisata juhul, kui see asub koonuse peal või samade koordinaatidega punkt on juba olemas. Selleks, et luua uusi juhuslikke punkte, kirjutati funktsioon, mis võtab arvesse eelnevalt märgitud meetri sees olevate koonuste koordinaadid ja lisab sinna juurde juhusliku suuruse. Selleks kasutati C ++ *std::uniform_int_distribution* klassi, mis genereerib uusi punkte etteantud vahemikus diskreetse tõenäosus funktsiooni alusel. Võistlustel võib rada olla 9 meetrise diameetriga ja juhuslikkuse alusel võib esimene punkt sattuda kohta, kus selle edasi minnes on ainult keelatud ala. Olukorra vältimiseks lisati kontroll, et kui valmis RRT* puu suurus pole piisav, tehakse täiesti uus puu, mille esimene tipp on teistsugune ja tänu sellele ka tulenev RRT* puu. Koonuste ümber olev piirkond kuhu tippe ei tohi genereerida, tehti katsetamise käigus nii suur, et RRT* saaks küll sealt läbi minna, aga tihedam puu läks alati mööda rada. Joonisel 14 on RRT* algoritm (roheliselt) ehitanud puu koonuste juurde (mustad silindrid ümber kollaste ja siniste pulkade) ja on leitud raja keskjoon (must joon). Raja keskjoont visualiseerides tundub nagu oleks kaks joont, aga tegelikult see joon mis läheb otse vormeli juurest raja keskjoone tippu on visualiseerimise eripära.



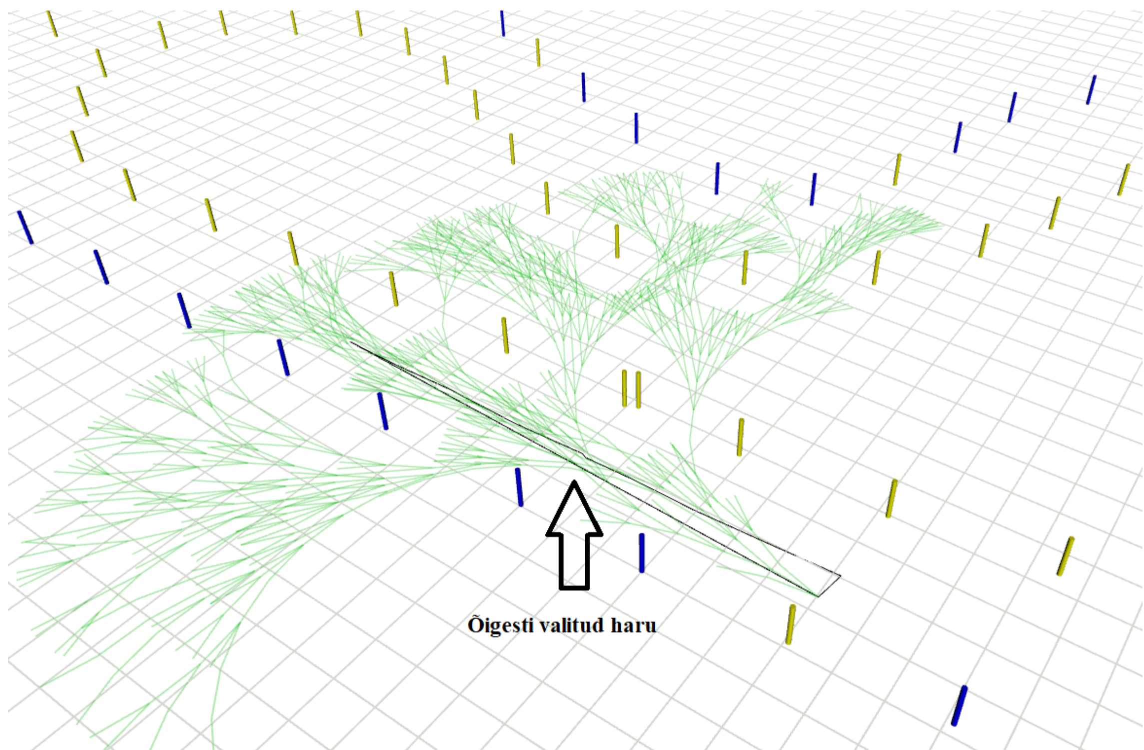
Joonis 14. RRT* ehitamas ennast ümber koonuste ja mööda rada.

RRT* toimis kindlalt kui raja tegemise kauguseks oli 10 meetrit. Nõutud sõidukiirus on esimesel ringil minimaalselt 10 m/s ja selleks on MPC-l vaja ette näha poole rohkem. Sensoorika võimaldab hetkel ette vaadata kindlusega 20 meetrit ja see võeti ka raja planeerimise eesmärgiks. Oli vaja saavutada minimaalselt 20 meetrit ette nägemist. Vaadates ette 20 meetrit on suur võimalus, et vaadeldavate koonuste hulka kuuluvad ka koonused, mis asuvad teises raja osas ja tänu sellele hakkab ka RRT* puu end sinnapoole ehitama . Tekkis olukord, kus ette nähti vähem koonuseid ja kiideti haru, mis liikus teise raja osa poole (Joonis 15). Vea parandamiseks kirjutati uus parima haru valimise funktsioon. Algselt taheti kasutada auto sõidusuunda ja kiita haru, mille viimane punkt asub yaw suunale kõige lähemal. See lahendus sobis hästi sirgetele, aga mitte kurvidesse. Probleemid tekkisid järskudes kurvides. Hakati kiitma harusid, mis liikusid kurvist välja ja sellest tulenevalt ei saanud seda lahendust kasutada. Teine lahendus oli rohkem punkte anda harule, kus on RRT* erinevate puude tipud rohkem koos (Joonis 16). Harule punkte andes vaadati tema tipu ümber 3 meetri raadiuses (minimaalne raja laius) teiste harude tippe ja anti iga tipu kohta punkt. Kui RRT* ehitab ennast mööda rada, siis on erinevad

harud kokku surutud, sest koonused ei luba lahku neil minna. Ehitades ennast aga rajalt välja on RRT*l palju vabadust ja sellest tulenevalt on puu väga laiali.

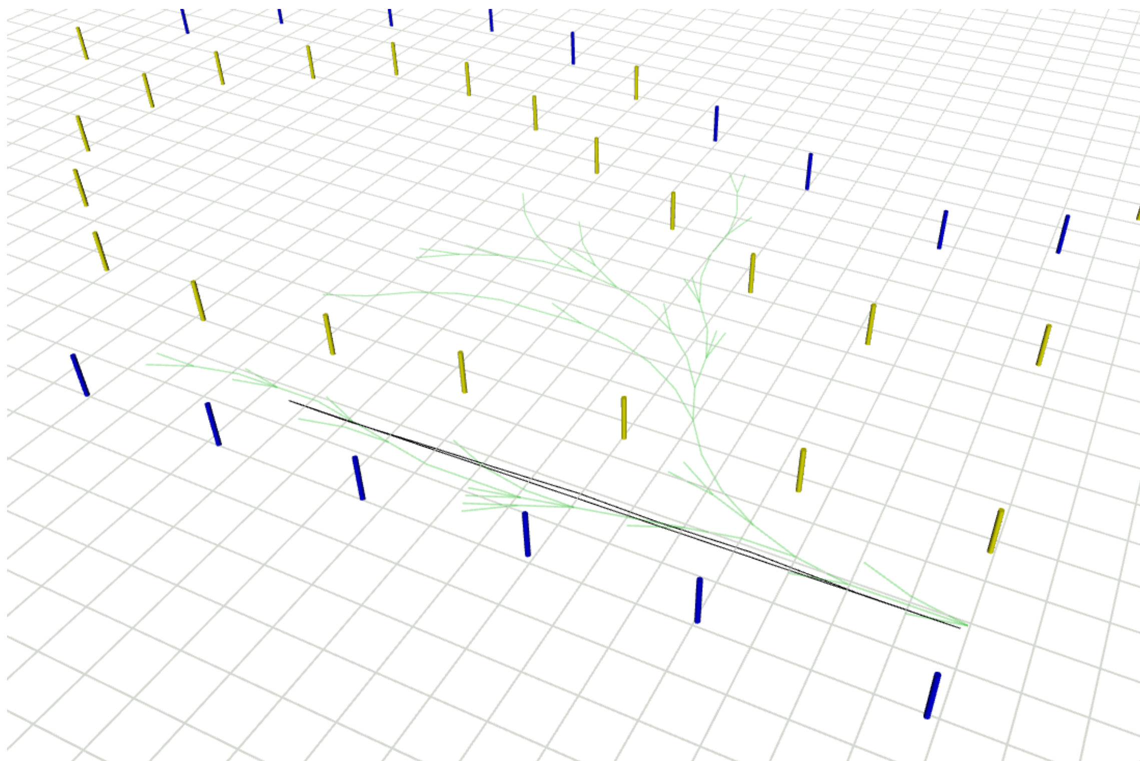


Joonis 15. Valesti valitud parim haru.

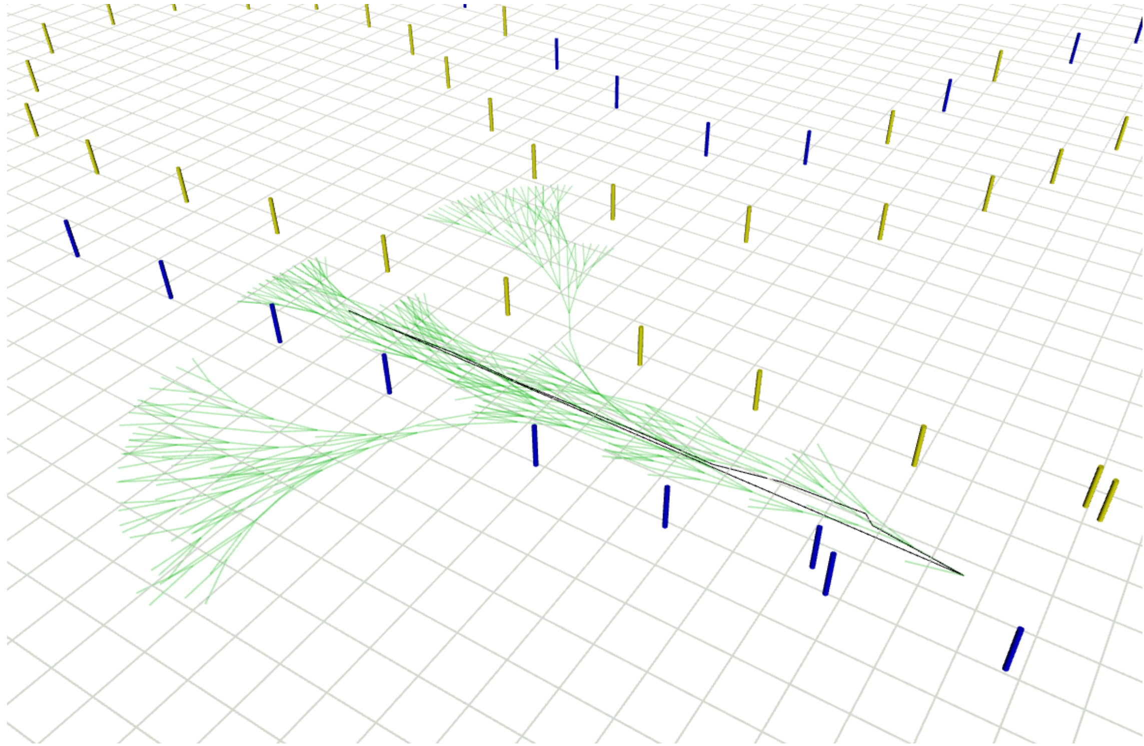


Joonis 16. Õigesti valitud parim haru.

Algselt oli RRT* puus olevate punktide arv minimaalne (200 punkti) (Joonis 17) Selleks, et parima puu valimine oleks veel kindlam, suurendati puus olevate punktide arvu 10 000 punkti peale (Joonis 18), sest siis on suurem võimalus, et palju erinevaid harusid liigub rajal õiges suunas. 10 000 valikul osutus otsustavaks koodi efektiivus. Efektiivsusel tekkisid probleemid 20 000 punkti juures, Hz langes 10 pealt 8 peale . Samas oli 20 000 punkti juba nii palju, et raske on erinevatel harudel vahet teha. Kiites haru, millel on palju teisi harusid ümber saadi soovitud tulemus. Seda võimaldas teha C++ keeles kirjutatud kood, sest jõudlus ja töösagedus ei langenud alla nõutud piiri peale punktide lisamist.



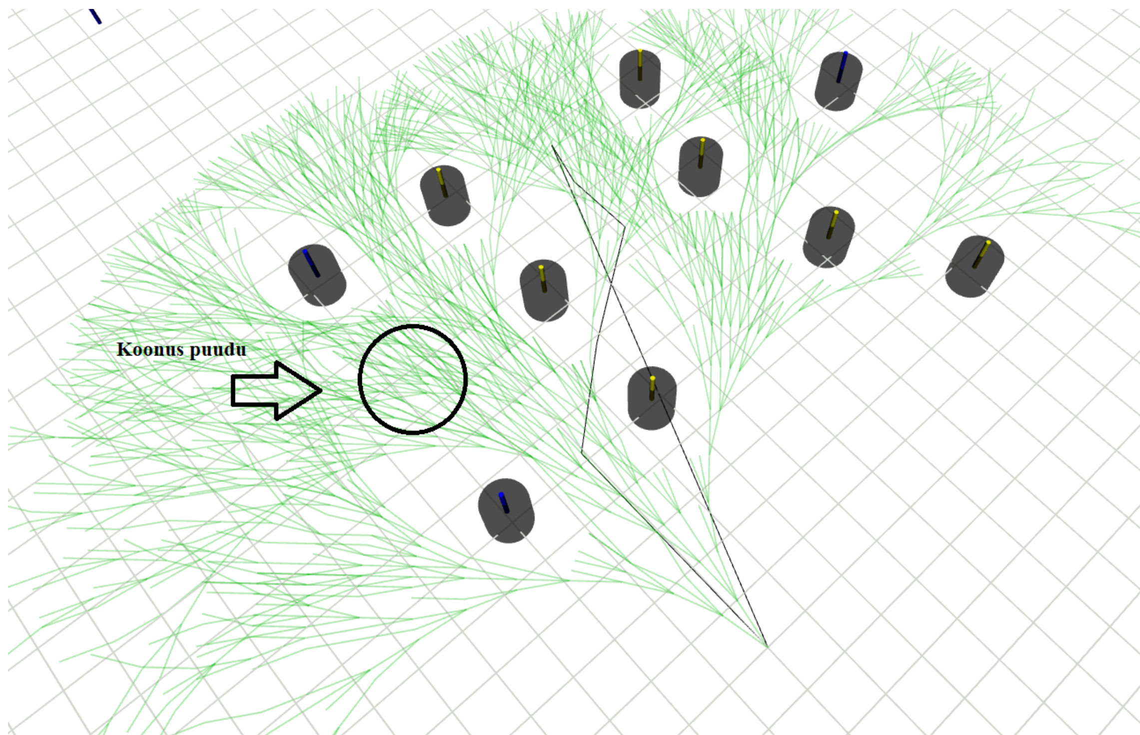
Joonis 17. Hõre RRT* puu.



Joonis 18. Tihe RRT* puu.

Kaardistamisel ringil varieerub koonuste arv. Simulaatorid testimisel oli 20 meetrit koonuseid ette vaadata hea, sest need olid alati olemas. Kaardistamisel võib juhtuda, et ette on näha ainult paar koonust ja kõrval raja osalt on näha palju, tänu sellele satub parimaks haruks see, mis liigub rajalt välja ja leiab muu raja osa koonuseid. Tavaliselt juhtub selline olukord siis auto pole veel liikunud ja paljud koonused jäävad üksteise taha ja neid ei saa lidar tuvastada (Joonis 19). Kuigi viga tuleb harva, siis on see piisav, et MPC läheks valesti. Probleemi lahendamiseks otsustati kasutusele võtta koonuste värvid. Eelnevatel aastatel trajektoor sõltus koonuste värvides, aga see aasta saadakse ilma nendeta hakkama. Selleks, et ka erijuhtudel ja ebaühtlaste koonuste arvudega hakkama saada, võeti arvesse koonuste värvid, mis on täiesti kindlad. Kaardistamine annab välja koos koonuse asukoha ja värviga ka koonuse värvi tõenäosuse. See näitab, mitu mõõtmist selle koonuse kohta on olnud sama värvi. Ainult siis kui on koonuse värvi tõenäosus on 100%, arvestab trajektoor seda parima haru leidmisel. See tähendab, et lisaks sellele, et kiidetakse haru, millel on koonuseid nii vasakul kui ka paremal pool, antakse punkte kui vasakul olevad koonused on sinised ja paremal olevad koonused on kollased. Värvi punkte antakse ainult haru osadele, mis asuvad autost 5 meetri kaugusel, sest siis on värvi tuvastamise protsent 98%. Selle lahendusega ei ole keskjoone leidmine värvides sõltuv,

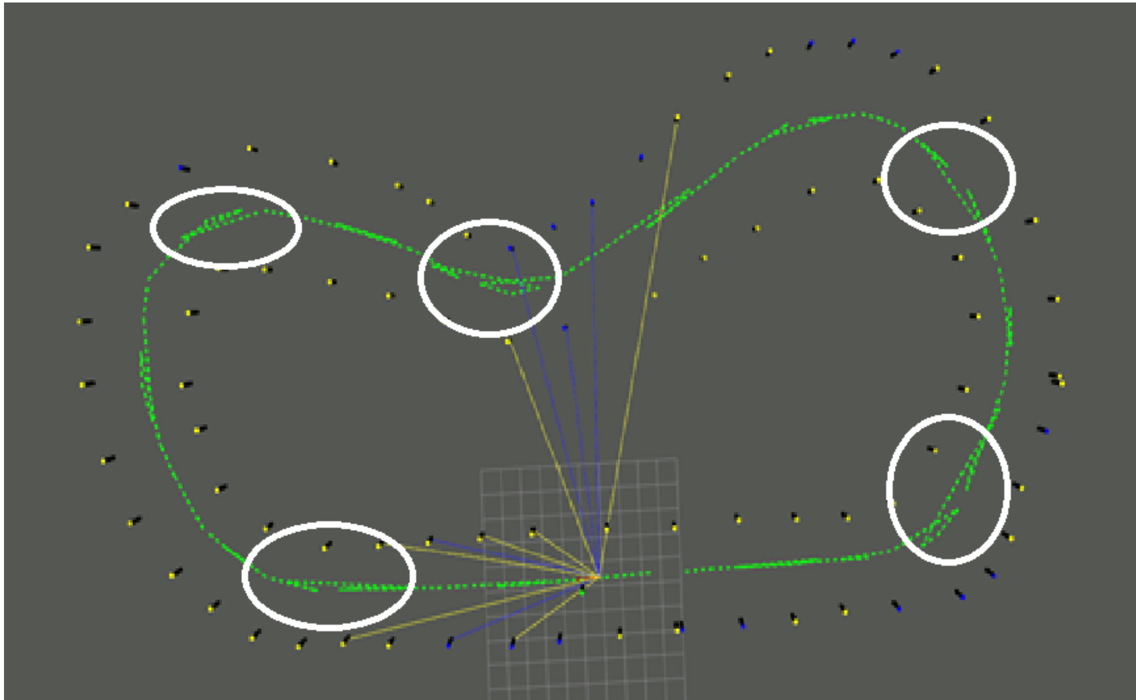
aga samal ajal kui võimalik saab kasutada ära lidari ja kaamera funktsiooni koonuste värvi leida.



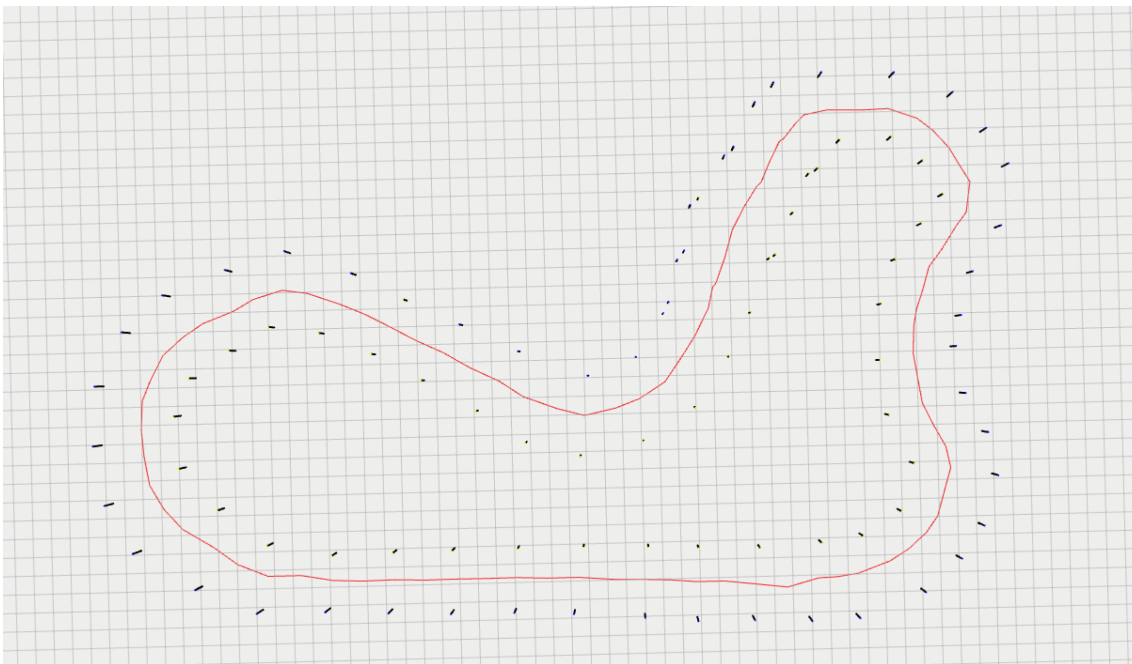
Joonis 19. Kaaristamisel sisendiks tulevad ebaühtlased koonused.

Peale esimese ringi ja kaardistamise lõpetamist peab hakkama MPC sõitma vormeli täiskiirusel, ehk ette vaatamiskaugus kasvab ja on tunduvalt suurem kui kaugus mida RRT* võimaldab vaadata. Selleks genereeritakse rada selle põhjal kus vormel eelmine ring sõitis ja võetakse sealt Delaunay triangulatsiooni abil rajalt punktid. Tulemus on taaskord raja keskjoon, aga seekord kogu rajale ja see on muutumatu koos kaardiga.

Vead lahenduses tulid välja siis kui SLAM-i algoritmi optimeeriti. SLAM-is hoitakse trajektoori sõnumina meeles vormeli sõit punktidena. Selleks, et liiga palju punkte meeles ei hoitaks ja koormust minimeerida, hõrendati meeles hoitavate punktide arvu poole võrra. Kuna raja leidmise algoritm tahab panna raja punktid järjestikult kõikide Delanay triangulatsiooni külgede peale, siis SLAM-i hõredad punktid panevad koodi vahepeal minema tagasi vahelejäetud punkte tagasi võtma (Joonis 20). Selle lahenduseks pidi muutma raja tegemise algoritmi. Kogu raja trajektoori tegemisel võeti ära käsk, et triangulatsiooni punktid peavad järjestikku olema. Hõredate punktide kompenseerimiseks pandi punktide vahele splainid (Joonis 21).



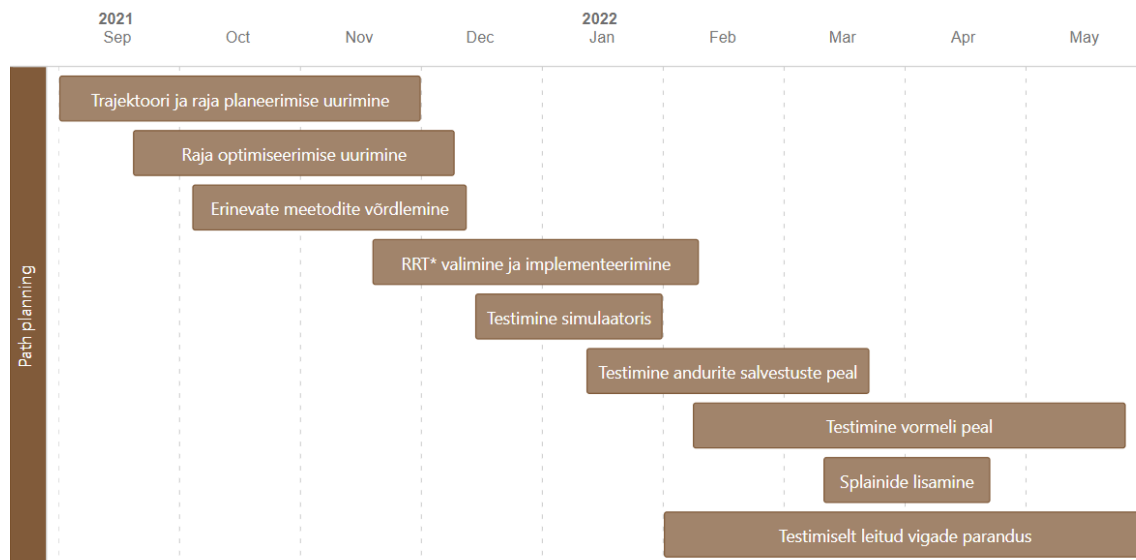
Joonis 20. SLAM-ilt tulevad hõredad punktid ja sellest põhjustatult keskjoon katkine.



Joonis 21. Vigadeta kogu ringi keskjoon.

Süsteemi planeerimine sai alguse septembris kui hakkas tudengivormelis uus hooaeg. Esimesed kuud uuriti erinevaid uurimistöid ja võrreldi meetodeid. Uurimise tulemusena

leiti, et raja optimeerimine ei ole see hooaeg veel vajalik ja keskenduma peaks keskjoone leidmisele, mis ei oleks sõltuv koonuste värvidest. Võrreldi erinevaid meetodeid mis otsivaid koonuste vahelt minevaid teid ja parimaks valiti RRT*. Algoritmi implementeerimisega alustati detsembris ja pidevalt testiti seda simulaatoris. Esimest korda sai uut lahendust auto peal testitud veebruaris. Järgnevad kuud tegeleti testimisel leitud vigade parandamisega. Märtsis sai algoritmile juurde lisatud splainid, et MPC ei peaks nii palju optimeerima ja toimiks kiiremini.



Joonis 22. Gantt diagramm hooaja tööst

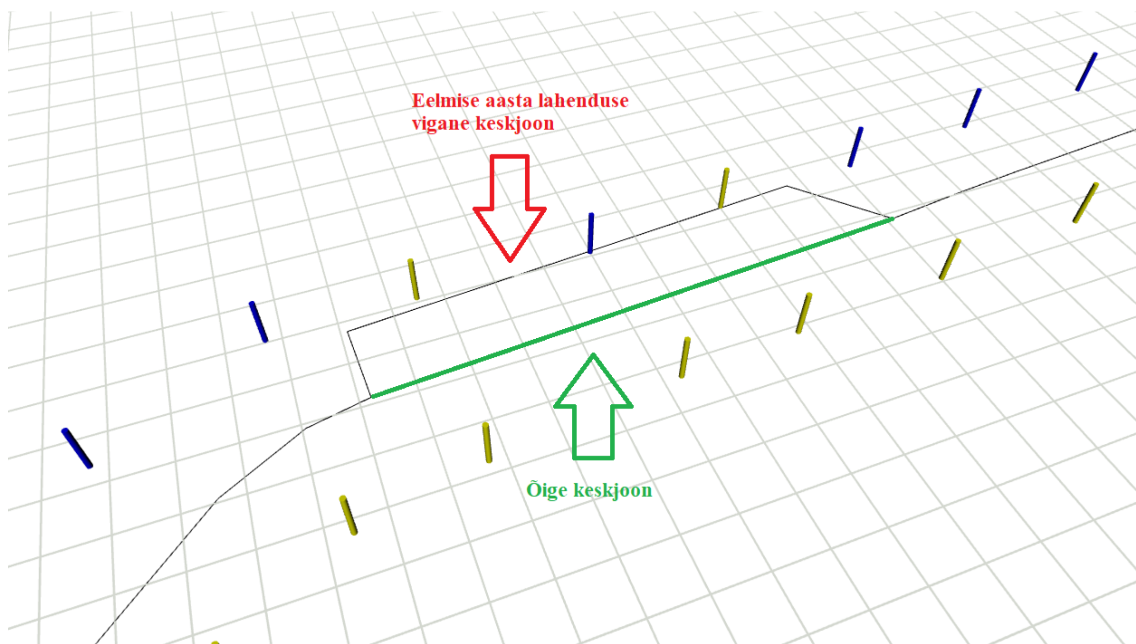
5.2 Võrdlemine eelmise aasta lahendusega

Esimese aasta raja leidmine toimus samuti SLAM-i ja odomeetria sisendite põhjal. Lisaks võeti arvesse auto kiirus. Loodi juhuslikult palju erinevaid radasid etteantud punktide vahele. Iga raja kohta vaadati kui kiire oleks vormel füüsikaliste parameetrite põhjal sellel rajal ja jäeti meelde kõige kiirem tulemus. Algoritmi viga tuli sisse selles, et taheti ette saada täpseid koonuste paare, ehk oldi sõltuvad värvidest. Lisaks, kuna algoritm oli vormeli peal ainult esimesel aastal, ei osata öelda kuidas oleks saadad hakkama auto suurtel kiirustel.

Lokaliseerimise ja dünaamika vigade tõttu otsustati teisel aastal üle minna keskjoonele. Raja leidmise algoritm oli koonuste sorteerimine ja siis nende vahelise keskpunkti võtmine. Sorteerimine toimus auto asukoha järgi. Kokku pandi kõige lähem kollane koonus lähima sinise koonusega ja siis jätkati seda kõikide nähtaval olevate koonustega. Vahepeal võib juhtuda, et vaadeldavate koonuste hulka võivad tulla ka teise raja osa

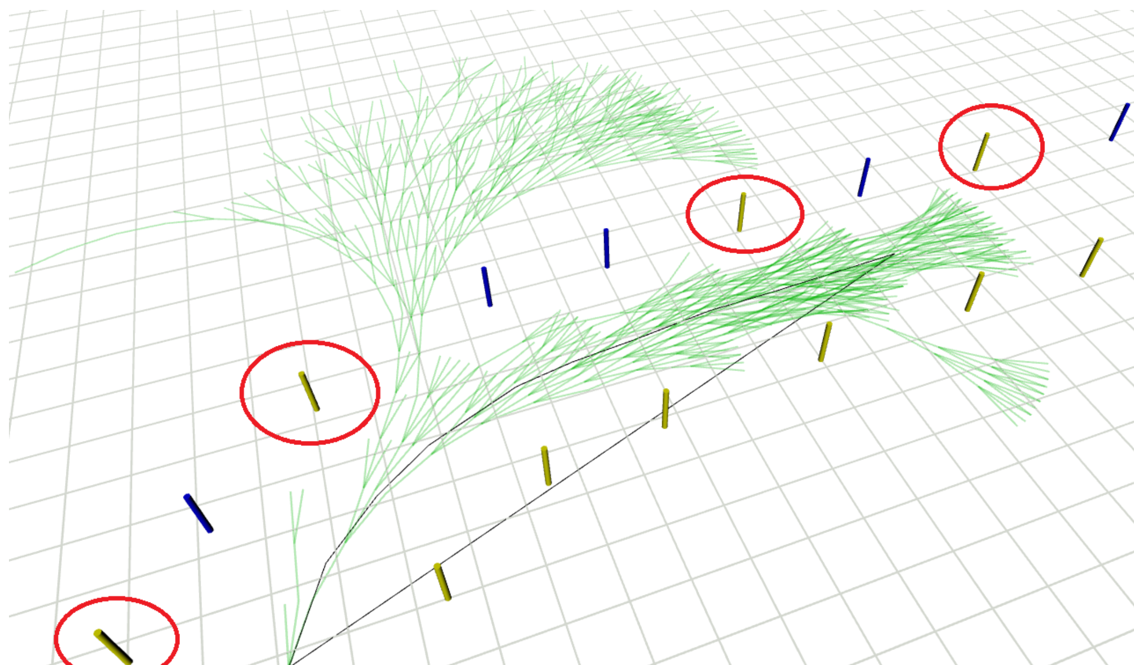
koonused ja algoritmil võis vahepeal valesti minna. Võistlustel pandi raja juurde aga punased koonused, mida reeglites kirjas ei olnud ja kuna kaamerad tuvastasid neid pidevalt erinevat värvi, siis raja tegemine jookseb valesti. Peale seda otsustati, et raja leidmine ei saa olla sõltuv koonuste värvidest.

RRT* algoritmi ja eelmise aasta raja leidmist jooksutati andurite salvestuste info (ingl *rosbag*) peal. Salvestuse peal on koonuste värvi mõõtmised läinud mõnes kohas valesti ja saab näha kuidas erinevad algoritmid seda lahendavad. Eelmise aasta lahendus teeb keskjoone punktid ka valede koonuste vahele, sest peale värvide midagi ei vaadata ja tänu sellele ei ole võimalik viga ära hoida (Joonis 23). Lootma ei saa jääda aga sellel, et kaamerad ei tee vigu värvi tuvastamisel, sest valguse ja varjudega võib ette tulla olukordi kus kaamera seaded on valed ja varem treenitud närvivõrk ei tööta.



Joonis 23. Eelmise aasta keskjoone leidmise lahendus valesti.

Selle aasta RRT* algoritmiga on näha, et isegi kui koonuste värvid on täiesti sassis ei mõjuta see raja planeerimise juures midagi (Joonis 24 valesti ennustatud koonuste värvid punase ringiga). Koodis küll kiidetakse tippe mis on kahe erineva koonuse vahel, aga ainult siis kui ollakse selles koonuse värvis täiesti kindlad.



Joonis 24. RRT* algoritmi ei mõjuta valet värvi koonused.

Eelmise aasta lahendus ja RRT* pandi võrdluse alla eelmisel hooajal toimunud FS Easti võistluse salvestusel. Eelmise aasta algoritm planeeris keskjoont õigesti kolm sekundit. Peale seda tuvastati valesti koonuse värv ja keskjoon arvutati rajalt välja ja katkestamist polnud võimalik vältida. Vormel ei jõudnud isegi stardi jooneni. RRT* puhul on näha, et keskjoon arvutatakse õigesti stardi joonest edasi ja koonuste valesti tuvastamine ei ole segav faktor. Sellega tõestati ära, et võistlustel jõutakse kaugemale RRT*-ga ja õigustati selle kasutamist süsteemis. (Lisa 4)

Võrreldes eelnevate aastatega saavutati raja leidmise stabiilsus ja värvidest sõltumatus nagu ka taheti. RRT* algoritmil on pole probleeme kiirusega ega tulemustega. Samas on näha, et on ka veel arenemisruumi. RRT* ei võta arvesse auto võimekusi. RRT* sõltub väga palju vormeli asukohast rajal. Raja leidmiseks eeldatakse, et MPC saab hakkama keskjoonel sõitmisega ja ei sõida rajalt välja. Rajal välja või koonusele otsa sõites ei suudeta luua RRT* puud ja rada mille järgi sõidetakse kaob täielikult.

Maksimaalne roolinurk on ette ära antud ja väga järskudes kurvides ei pööra vormel keskjoonel jälgides seda välja. Sõltuvalt olukorrast peaks algoritm ohverdama keskjoone ja rada optimeerima, et hakkama saada ka rajal raskemates olukordades. Kuigi hetkel pole

optimeeritud rada vaja, siis tulevikus on ideaaltrajektooril sõitmine kindel eesmärk, et võistlustel võita.

5.3 Testimine

Selleks, et koodi uuendamist ja testiks valmisolekut kontrollida, kirjutati raja leidmisele automaattestid. Selleks kasutati Python *unit teste* [25], sest Pyhton on autorile tuttav programmeerimiskeel, testid said kiirelt valmis ja saadi koheselt kasutusele võtta. Python *unit tests* on sisseehitatud moodul, mis kasutab XML väljundit, et ROS-i testide tulemusi näidata. Üks võimalus ROS-i *unit test*-idega testida on luua test kui sõlm (ingl *node*), mis jälgib ROS sõnumite väljundeid.

Raja planeerimise koodi testimiseks on kõige parem vaadata kas sõlmed saadavad infot välja. Kuna info on pidevalt muutuv, siis ei saa väljundit võrrelda. Valiti välja kõige tähtsamad sõnumid: RRT* jooned, koonused mille vahel planeerimist teha ja raja planeerimisest tulev keskjoon, mis läheb edasi MPC-le. Kirjutati sõlm, mis jälgib kõiki kolme sõnumit. Testimiseks peab panema tööle koodi ja testimise sõlme. Tulemusest näeb koheselt ära millised sõlmed annavad infot välja ja millised mitte. Testide kirjutamine vähendas oluliselt vormeliga testimisel vea leidmise aega, mis on mahuka süsteemiga väga oluline.

6 Kokkuvõte

Töö eesmärk oli leida FS Team Tallinna isejuhtivale vormelile rada efektiivselt ja koonuste värvidele toetumata. Eesmärgid saavutati edukalt. Sobivaks algoritmiks valiti RRT* ja implementeeriti C++ keeles. Kaardistamise ringil suudab algoritm keskjoone leida 10 korda sekundi jooksul. Koonuse värvid võetakse arvesse, kui nende värvides ollakse 100% kindlad, aga kui värve ei leita, siis raja planeerimine katki ei lähe. Selleks, et MPC-l oleks rada efektiivsem järgida, tehakse raja punktide vahele splainid. Uurimise tulemusena otsustati raja optimeerimist mitte kasutada, sest vormeli lokalisatsioon ja MPC pole veel nii täpsed, et ideaaltrajektoori suurel kiirusel järgida. Koonuse maha sõitmisega kaotatakse punktides rohkem kui paari tuhandikuga rajal aeglasemalt sõites. Algoritmi võrreldi eelmise aasta lahendusega FS Easti võistluste salvestuse peal. Võrdluse tulemusena nähti, et kui vana algoritm sõitis kohe rajal välja, siis RRT* oleks rajal püsinud nii kaua kuni salvestus kestis. Töös leitud raja planeerimise lahendusega minnakse võistlema Formula Student võistlustele sel suvel.

Kasutatud kirjandus

[1] „www.formulastudent.ee,“ [Võrgumaterjal]. Available: https://www.formulastudent.ee/galerii/.

[2] „www.formulastudent.de,“ [Võrgumaterjal]. Available: https://www.formulastudent.de/about/concept/.

[3] K. Karur, „A Survey of Path Planning Algorithms for Mobile Robots,“ pp. 449-450, 2021.

[4] A. Ganti, „www.investopedia.com,“ 2021. [Võrgumaterjal]. Available: https://www.investopedia.com/terms/d/degrees-of-freedom.asp.

[5] B. Paden, „A Survey of Motion Planning and Control Techniques for Self-Driving Urban Vehicles,“ 2016. [Võrgumaterjal]. Available: https://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=7490340. [Kasutatud Aprill 2022].

[6] G. Klancar, Wheeled Mobile Robotics, 2017.

[7] „www.formulastudent.de,“ 2022. [Võrgumaterjal]. Available: https://www.formulastudent.de/fileadmin/user_upload/all/2022/rules/FS-Rules_2022_v1.0.pdf. [Kasutatud Aprill 2022].

[8] N. Ganganath, „A 2–Dimensional ACO-based Path Planner for Off-line Robot Path Planning,“ 2013. [Võrgumaterjal]. Available: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=6685700. [Kasutatud Aprill 2022].

[9] J.-d. Zhang, "Vehicle routing in urban areas based on the Oil Consumption Weight -Dijkstra algorithm," 2016.

[10] P. E. HART, „A Formal Basis for the Heuristic Determination of Minimum Cost Paths,“ 1968. [Võrgumaterjal]. Available: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=4082128. [Kasutatud Aprill 2022].

[11] L. Kavraki, „Probabilistic Roadmaps for Path Planning in High-Dimensional Configuration Spaces,“ 4 August 1996. [Võrgumaterjal]. Available: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=508439. [Kasutatud Aprill 2022].

[12] J. Canny, „New lower bound techniques for robot motion planning problems,“ [Võrgumaterjal]. Available: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=4568255. [Kasutatud Aprill 2022].

[13] F. Peralta, „A comparison of local path planning techniques of autonomous surface vehicles for monitoring applications: The Ypacarai lake case-study,“ 2020.

[14] C. Murray, „Robot Motion Planning on a Chip,“ 2016.

[15] Y. Dong, „www.youtube.com,“ 19 oktoober 2015. [Võrgumaterjal]. Available: https://www.youtube.com/watch?v=JeEk_CWcRFI.

[16] „roboticsbackend.com,“ [Võrgumaterjal]. Available: https://roboticsbackend.com/when-to-use-python-vs-c-in-robotics/.

[17] J. Buntinx, „cryptomode.com,“ 19 oktoober 2020. [Võrgumaterjal]. Available: https://cryptomode.com/c-is-the-most-energy-efficient-and-fastest-programming-language-study-finds/. [Kasutatud aprill 2022].

[18] R. Tellez, „www.theconstructsim.com," 2019. [Võrgumaterjal]. Available: https://www.theconstructsim.com/what-is-ros/. [Kasutatud 2022].

[19] AMZ, „github.com," 2019. [Võrgumaterjal]. Available: https://github.com/AMZ-Driverless/fssim. [Kasutatud detsember 2021].

[20] M. Yastremsky, „github.com," 13 veebruar 2019. [Võrgumaterjal]. Available: https://github.com/MaxMagazin/ma_rrt_path_plan. [Kasutatud 2021].

[21] N. Amenta, „Complexity of Delaunay Triangulation for Points on Lower-dimensional Polyhedra," HAL Open Science, 2007.

[22] T. M. Vu, „Model Predictive Control for Autonomous Driving Vehicles," MDPI, 2021.

[23] A. W, „Cubic Spline Trajectory Planning and Vibration Suppression of SemiconductorWafer Transfer Robot Arm," 2014.

[24] M. M. Achin Jain, „Computing the racing line using Bayesian optimization," 2020.

[25] "wiki.ros.org," [Online]. Available: http://wiki.ros.org/unittest.

Lisa 1 – Lihtlitsents lõputöö reprodutseerimiseks ja lõputöö üldsusele kättesaadavaks tegemiseks[1]

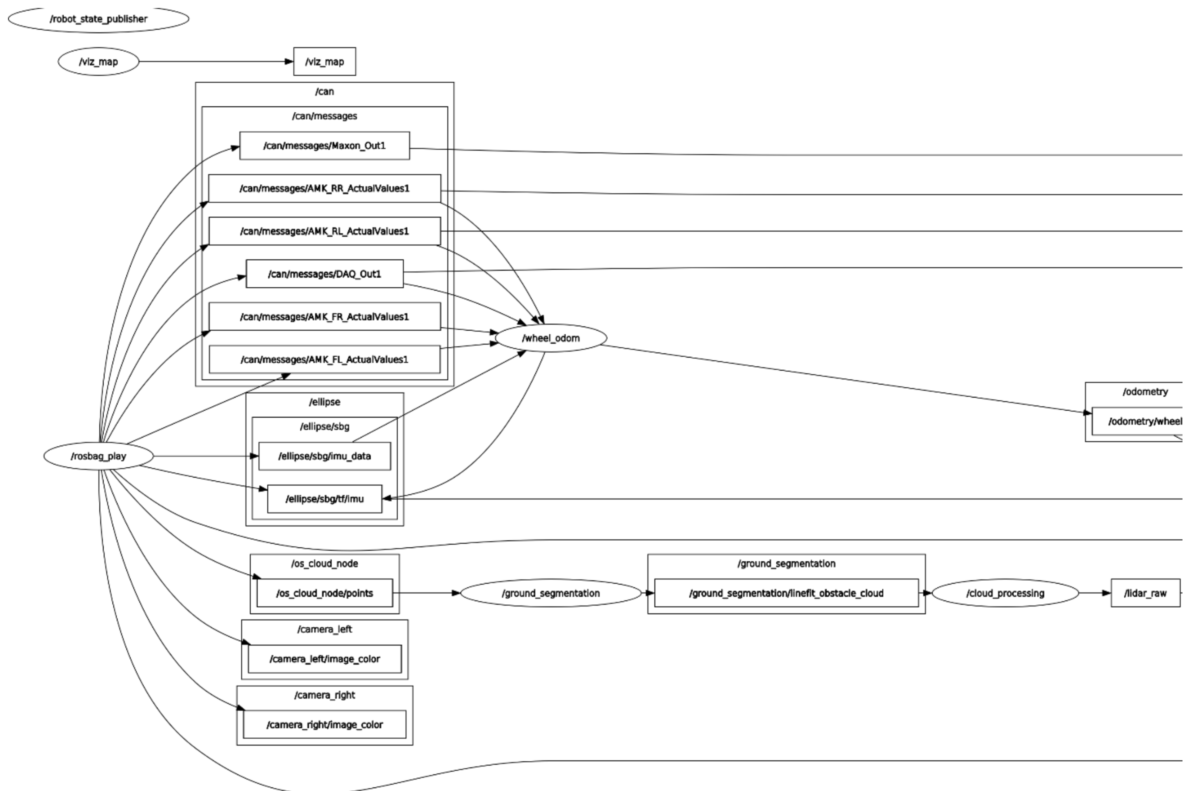
Mina, Lisanne Siniväli

1. Annan Tallinna Tehnikaülikoolile tasuta loa (lihtlitsentsi) enda loodud teose "Raja leidmine FS Team Tallinn isejuhtivale vormelile", mille juhendaja on Gert Kanter.
   1.1. reprodutseerimiseks lõputöö säilitamise ja elektroonse avaldamise eesmärgil, sh Tallinna Tehnikaülikooli raamatukogu digikogusse lisamise eesmärgil kuni autoriõiguse kehtivuse tähtaja lõppemiseni;
   1.2. üldsusele kättesaadavaks tegemiseks Tallinna Tehnikaülikooli veebikeskkonna kaudu, sealhulgas Tallinna Tehnikaülikooli raamatukogu digikogu kaudu kuni autoriõiguse kehtivuse tähtaja lõppemiseni.
2. Olen teadlik, et käesoleva lihtlitsentsi punktis 1 nimetatud õigused jäävad alles ka autorile.
3. Kinnitan, et lihtlitsentsi andmisega ei rikuta teiste isikute intellektuaalomandi ega isikuandmete kaitse seadusest ning muudest õigusaktidest tulenevaid õigusi.

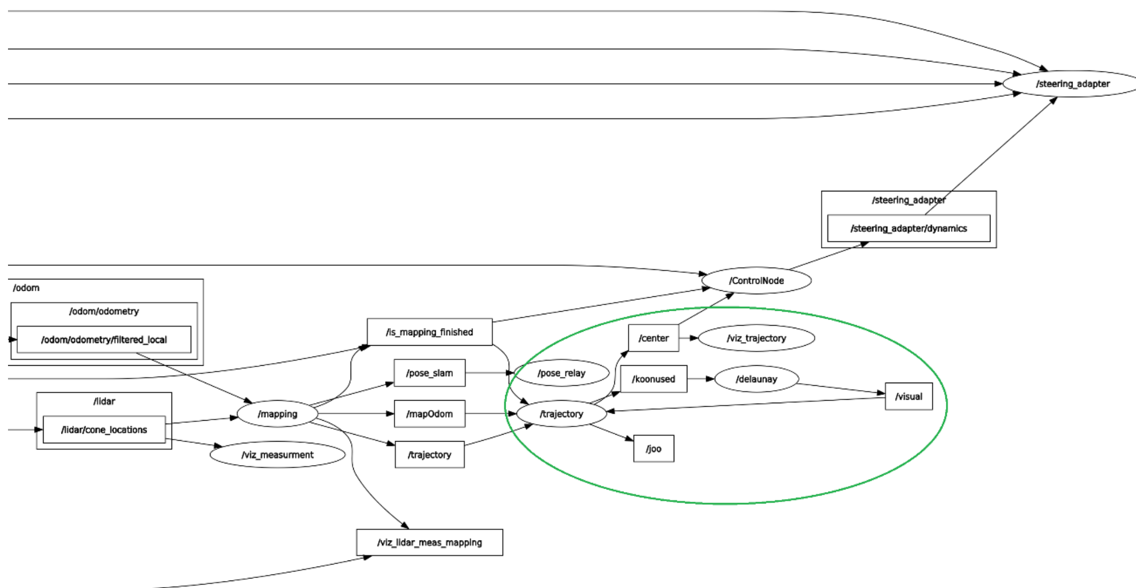
30.05.2022

1 Lihtlitsents ei kehti juurdepääsupiirangu kehtivuse ajal vastavalt üliõpilase taotlusele lõputööle juurdepääsupiirangu kehtestamiseks, mis on allkirjastatud teaduskonna dekaani poolt, välja arvatud ülikooli õigus lõputööd reprodutseerida üksnes säilitamise eesmärgil. Kui lõputöö on loonud kaks või enam isikut oma ühise loomingulise tegevusega ning lõputöö kaas- või ühisautor(id) ei ole andnud lõputööd kaitsvale üliõpilasele kindlaksmääratud tähtajaks nõusolekut lõputöö reprodutseerimiseks ja avalikustamiseks vastavalt lihtlitsentsi punktidele 1.1. ja 1.2, siis lihtlitsents nimetatud tähtaja jooksul ei kehti.

Lisa 2 – Rqt graaf vormeli süsteemist



Joonis 25. Rqt graaf esimene pool



Joonis 26. Rqt graaf teine pool, raja planeerimise osa näidatud rohelisega

Lisa 3 – Sõnumite kujud


```
header:
  seq: 34
  stamp:
    secs: 1630674550
    nsecs: 652917504
  frame_id: "/map"
child_frame_id: ''
pose:
  pose:
    position:
      x: -1.8816052675247192
      y: -0.03939858078956604
      z: 0.0
    orientation:
      x: 0.0
      y: 0.0
      z: 0.004784945765713908
      w: 0.9999885520814822
  covariance: [0.006333691533654928, 0.0008345707901753485, -6.75524061080
05, -0.00039113464299589396, 0.4946528971195221, 0.0, 0.0, 0.0, 0.0, 0.0,
twist:
  twist:
    linear:
      x: 0.0
      y: 0.0
      z: 0.0
    angular:
      x: 0.0
      y: 0.0
      z: 0.0
  covariance: [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
```


Joonis 27. */odom* sõnum

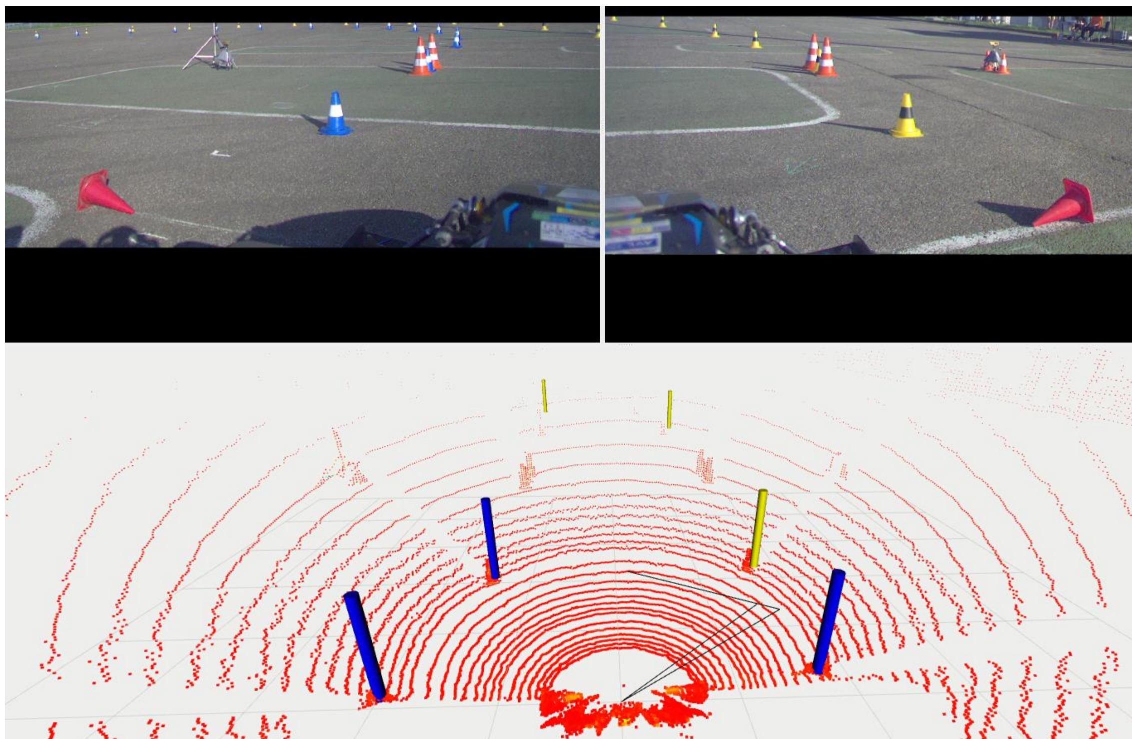
```
Header header

dv_main/Point[] yellowCones
dv_main/Point[] blueCones
dv_main/Point[] unknownCones
dv_main/Point[] smallOrangeCones
dv_main/Point[] bigOrangeCones
```

Joonis 28. */map* sõnum

```
header:
  seq: 15
  stamp:
    secs: 1651336331
    nsecs: 701673651
  frame_id: "map"
polygon:
  points:
    -
      x: 14.069589614868164
      y: -0.2081647366285324
      z: 0.0
    -
      x: 14.269364356994629
      y: -0.217656672000885
      z: 0.0
    -
      x: 14.469139099121094
      y: -0.2271486073732376
      z: 0.0
    -
      x: 14.668913841247559
      y: -0.2366405427455902
      z: 0.0
    -
      x: 14.868688583374023
      y: -0.246132493019104
      z: 0.0
    -
```


Joonis 29. /center sõnum

Lisa 4 – Formula Student East võistluse salvestus



Joonis 30. FS East võistluse salvestus