Széchenyi István Egyetem
Gépészmérnöki, Informatikai és Villamosmérnöki Kar
Informatika Tanszék

DIPLOMAMUNKA

Tőzsér Zoltán

Mérnökinformatikus MSc szak

2023

DIPLOMAMUNKA

Önvezető jármű LIDAR pontfelhő szegmentálása, vezethető útfelület meghatározása

Tőzsér Zoltán

Mérnökinformatikus MSc szak

2023

ADATLAP DIPLOMAMUNKA TÉMA ENGEDÉLYEZÉSÉHEZ

Hallgató adatai

Név: Tőzsér Zoltán
Neptun-kód: KMA8OR
Cím: 3400 Mezőkövesd, Petőfi Sándor utca 42.
Telefon: +36302776568
e-mail: zoltantozser1982@gmail.com
Szak: Mérnök informatikus
Képzési szint: MSc
Tagozat: levelező

A szakdolgozat adatai

Kezdő tanév és félév: 2020/21 Tavas
Várható leadás: 2021/22 Ősz
Cím: Önvezető jármű LIDAR pontfelhő szegmentálása, vezethető útfelület meghatározása
Nyelv: magyar
Típus: nyilvános
Rövid leírás, részfeladatok:

A vezethető útfelület észlelése mind az vezetéstámogatás, mind az önvezető járművek fontos alfeladata. A rendszer megköveteli az úthatár azonosítását és a környező akadályok, például járművek, gyalogosok, védőkorlátok és épületek felderítését. A fejlesztendő algoritmus által észlelt eredmény felhasználható ütközések elkerülésére és az útvonal tervezésére. A feladat része, hogy ROS sensor_msgs/PointCloud2 típusban érkező pontfelhőből visualization_msgs/MarkerArray típusú üzenetben jelenítse meg az út szélét. A marker az út szélét jelző félpoligonokból álljon, lehetőleg egyszerűsített módon, kevés pontszámból álljon. A fejlesztendő algoritmussal szemben támasztott további követelmény a 20Hz-es feldolgozási idő.

Első félév:

- A választott téma körülhatárolása, általános tájékozódás
- Szakirodalmi anyagok gyűjtése, rendszerezése és feldolgozása
- Diplomamunka dokumentumának készítése

Második félév:

- Tervezés
- Kódolás
- Tesztelés,
- Diplomamunka végső megfogalmazása, elkészítése

Belső konzulens adatai

Név: Horváth Ernő
Tanszék: Informatika Tanszék
Beosztás: Egyetemi tanársegéd

Külső konzulens adatai

Név: Unger Miklós
Munkahely: Járműipari Kutatóközpont
Beosztás: Autonóm járműirányítási mérnök
Cím: 9026 Győr, Egyetem tér 1.
Telefonszám: +36305845244
e-mail: unger.miklos@ga.sze.hu

Győr, 2021. január 12.

Belső konzulens | Külső konzulens | Tanszékvezető

NYILATKOZAT

Alulírott, Tőzsér Zoltán (KMA8OR), Mérnökinformatikus MSc szakos hallgató kijelentem, hogy az ***Önvezető jármű LIDAR pontfelhő szegmentálása, vezethető útfelület meghatározása*** című diplomamunka feladat kidolgozása a saját munkám, abban csak a megjelölt forrásokat, és a megjelölt mértékben használtam fel, az idézés szabályainak megfelelően, a hivatkozások pontos megjelölésével.

Eredményeim saját munkán, számításokon, kutatáson, valós méréseken alapulnak, és a legjobb tudásom szerint hitelesek.

Győr, 2023. 04. 19.

hallgató

TARTALOMJEGYZÉK

KIVONAT

A vezethető útfelület észlelése, az úthatár azonosítása alapvető fontosságú az önvezető járművek navigálásához. Fontos az is, hogy a program a környező akadályokat (ilyenek például a járművek, a gyalogosok, az épületek) felderítse, ezáltal az eredmény felhasználható ütközések elkerülésére és az útvonalak tervezésére is.

A diplomamunkámban egy olyan projektfeladatot mutatok be, ami képes a LIDAR szenzorból érkező nyers adatokból (pontfelhőből), különböző szűrési eljárások segítségével meghatározni a nem út pontokat, az út pontokat, valamint a vezethető útfelület határvonalát. Az út szélét meghatározó pontok redukálva vannak vonallánc egyszerűsítő Lang algoritmussal, ezáltal egy egyszerűbb modell keletkezik a félpoligonokból. Mindezt úgy tesszük meg, hogy az eredeti vonallánc fontos vizuális jellemzőit megtartjuk.

A program részét képezi még egy dynamic reconfigure paraméterező felület is, melynek segítségével futás közben lehet frissíteni a paramétereket. A fejlesztett program különböző típusú LIDAR szernzorral is képes eredményesen működni. A tesztelések eredményei azt bizonyítják, hogy a fejlesztett algoritmus képes tartani a követelményben meghatározott 20 Hz-es feldolgozási időt.

ABSTRACT

The detection of the drivable road surface and the identification of the road boundary are of fundamental importance for the navigation of self-driving vehicles. It is also important that the program detects surrounding obstacles (such as vehicles, pedestrians, buildings), so that the result can be used to avoid collisions and plan routes.

In my thesis, I present a project task that is able to determine the non-road points, the road points, and the boundary line of the drivable road surface from the raw data (point cloud) coming from the LIDAR sensor, using different filtering procedures. The points defining the edge of the road are reduced with the polyline simplifying Lang algorithm, thereby creating a simpler model from the half-polygons. We do all this while maintaining the important visual characteristics of the original polyline.

The program also includes a dynamic reconfigure parameter setting interface, which can be used to update the parameters while running. The developed program can also work effectively with different types of LIDAR sensors. The results of the tests prove that the developed algorithm can keep the 20 Hz processing time specified in the requirement.

1 BEVEZETÉS

Az autonóm járművek és az ehhez szorosan kapcsolódó technológiák, rendszerek a közelmúltban az egyik legnépszerűbb kutatási témává váltak. Társadalmi szempontból rengeteg cikk és tanulmány mutatta be e technológiák potenciális fejlődését. Ipari szempontból sok vállalat és kutatóközpont törekedett e rendszerek tökéletesítésére. Számos olyan projekt került bemutatásra, amelyek igyekeztek megoldani azokat a kihívásokat, amelyekkel e technológiák még ma is szembesülnek. Ezen kívül nyitott a technológiai vita arról, hogy a rendelkezésre álló módszerek közül melyik a biztonságosabb, melyik észlelési rendszer megbízhatóbb. Az Egészségügyi Világszervezet (WHO) által közzétett globális állapotjelentés szerint az éves közúti közlekedésben bekövetkezett halálesetek száma 2018-ban elérte az 1,35 milliót, ezzel a világ nyolcadik leggyakoribb oka a természetellenes halálnak. Igaz, hogy az Európai Unióban a jelentett éves közúti halálesetek száma csökken, évente még mindig meghaladja a 40.000 halálesetet, amelynek 90%-át emberi tévedés okozta. A forgalom javítása és az emberi hibák elkerülése érdekében a globális befektetők jelentős összegeket szántak az autonóm járművek fejlesztésének támogatására.

Az autonóm jármű kifejezés megegyezik a vezető nélküli autóval, önvezető autóval, vagy a robotautóval. A jármű önállóságának azt a képességét tekintik, amely képes a forgalomban vezető nélkül és általában emberi erőforrások nélkül közlekedni. Műszakilag ez azt jelenti, hogy az emberi járművezetőt mesterséges alrendszerekre cserélik le, amelyeknek képesnek kell lenniük a meghatározott feladatok hasonló módon történő elvégzésére. Ilyen körülmények között a mesterséges rendszernek megfelelő ismeretekkel kell rendelkeznie, képesnek kell lennie arra, hogy helyesen érveljen és a korábbiakkal összhangban viselkedjen. Ez természetesen többet jelent, mint a hagyományos automatizálás és az adaptív vezérlés. Az idő múlásával hardverek és a szoftverek fejlődésének köszönhetően az automatizálás tömegesen behatolt a közlekedési rendszerekbe, mind az infrastruktúrán, mind a járműveken, de az ember továbbra is a járművezetés kulcseleme maradt egyenlőre.

Az autonóm járműveknek számos előnyük van, többek között a jobb biztonság és az alacsonyabb forgalmi torlódások, amelyek alacsonyabb üzemanyag -és energiafogyasztást eredményeznek. A hatékonyabb és biztonságosabb érzékelők csökkentik a balesetek számát, azonban ezen kifinomult eszközök telepítésével és javításával kapcsolatos költségek jelentősen megnőnek. Ezen előnyök és hátrányok mellett vannak olyan kérdések természetesen, amelyeket szükséges megoldani az autonóm járművek esetében. Iyen például

az, hogy ki fogja viseli a jogi felelősségeket, mi lesz a teendő, ha feltörik az autó vezérlőrendszerét. A vezeték nélküli kommunikáció fejlesztésével az autonóm járművek lehetővé teszik az együttműködést több jármű között. A járműhálózatok technológiai fejlődése és automatizálása jobb közúti biztonsághoz és alacsonyabb torlódásokhoz vezet a jelenlegi városi területeken, ahol jelenleg a hagyományos közlekedési rendszer egyre inkább rendezetlen és hatástalan. Összességében azt állíthatjuk, hogy ha megoldják vagy minimalizálják a hátrányokat, akkor az autonóm autók jelentős technológiai fejlődést jelentenek a következő években, mivel megkönnyítik az emberek életét és növelik a közúti biztonságot.

Az érzékelők és a kommunikációs technológia jelentős fejlődésével, valamint az akadályok felismerésére szolgáló technikák és algoritmusok megbízható alkalmazásával az automatizált vezetés olyan sarkalatos technológiává válik, amely forradalmasíthatja a közlekedés és a mobilitás jövőjét. Az érzékelők alapvető fontosságúak a jármű környezetének érzékelésében egy automatizált vezetési rendszerben, több integrált érzékelő használata és teljesítménye közvetlenül meghatározhatja az automatizált vezetésű járművek biztonságát és megvalósíthatóságát. A szenzor kalibrálása minden autonóm rendszer alapköve, tehát helyesen kell végrehajtani, mielőtt az érzékelő fúziós és akadály detektálási folyamatai megvalósulnak. Ezen kívül várhatóan az autonóm járművek hozzájárulnak a szén-dioxid-kibocsátás csökkentéséhez. Az önvezető járművek biztosítják a hagyományos járművek szállítási képességeit, nagyrészt képesek érzékelni a környezetet és önállóan navigálni minimális emberi beavatkozással vagy anélkül. A legtöbb autonóm vezetési rendszer sok közös kihívással és korlátozással rendelkezik a valós helyzetekben, például a biztonságos vezetés és a navigálás rossz időjárási körülmények között, valamint a biztonságos interakció a gyalogosokkal és más járművekkel. A rossz időjárási viszonyok, például a tükröződés, a hó, az eső és a köd jelentősen befolyásolhatják az érzékelők teljesítményét az észleléshez és a navigációhoz [1] [2] [3] [4] [5].

2 SAE INTERNATIONAL J3016 SZABVÁNYA

2014-ben a SAE International, korábban Society of Automotive Engineers bevezette a fogyasztók számára a J3016 szabványt, amely meghatározza a vezetési automatizálás hat különböző szintjét. Ugyanezt a szabványt más szervezetek is elfogadják, ezek például az International Organisation of Motor Vehicle Manufacturers (OICA) és a German Federal Highway Reserach Institute (BASt). Az ipar is gyakran hivatkozik rá a magasan automatizált járművek biztonságos tervezésében, fejlesztésében, tesztelésében és üzembe helyezésében.
Az első három szintet a járművezető által támogatott kategóriába lehet besorolni.
A 0. szinten (No automation) az összes feladatot a járművezető hajta végre és csak automatizált fedélzeti figyelmeztetéseket biztosít a jármű.
A 1. szinten (Driver assistance) az egyes feladatok irányítását meg lehet osztani a vezető és az automatizált rendszer között. Ilyen eset például az adaptív sebességtartó automatika (ACC), melyben az automatizált rendszer irányítja az autó sebességét, a vezető pedig a kormányzást. Egy másik jó példa erre a parkolássegítő, ahol a sebességet a vezető vezérli, a kormányzást pedig az automatizált rendszer. Ezen a szinten a vezető irányítja a gázpedált és a féket a környezet ellenőrzése közben.
A 2. szinten (Partial automation) az automatizált rendszer teljes mértékben átveszi a jármű dinamikájának irányítását, tehát a gyorsítást, a fékezést és a kormányzást irányítja. Abban az esetben, ha az automatizált rendszer nem reagál megfelelően, akkor a vezetőnek figyelnie kell a vezetést és egyben készen is kell állnia a megfelelő időben történő beavatkozásra. Itt a sofőr felelős a biztonságos kritikus műveletekért.
A további szintek (3-as, 4-es és 5-ös) a vezetési környezet figyelemmel kísérésére alkalmas rendszereket tartalmaznak.
A 3. szinten (Conditional automation) a jármű teljes környezeti megfigyelést végez, tehát feltételes vezetési automatizálást biztosít. Olyan funkciókat biztosít, amelyek alkalmassá teszik a járművet olyan helyzetek kezelésére, amikor a vezető eltereli a figyelmét az aktuális vezetési feladatoktól. Ezek a funkciók hasznosak olyan kritikus helyzetekben, amelyek azonnali reagálást igényelnek, ilyen például a vészfékezés. A sofőr ezen a szinten már nem felelős a biztonságtechnikai kérdésekért.
A 4. szinten (High automation) az önvezető autó funkcióinak bővítését és fejlesztését biztosítják. Itt a sofőr csak akkor rendlkezik az irányítással, ha az automatizált helyzet nem

válik biztonságossá. A kormányzást, a fékezést, a gyorsulást és az ellenőrzéseket az automatizált rendszer végzi.

Az 5. szinten (Full automation) a rendszer teljes mértékben ellenőrzi a járművet, nincs szükség emberi beavatkozásra. Ezen a szinten a sofőr már utasként van a járműben. Erre nagyon jó példa lehet a robot taxi.

A vezetési automatizálás hat különböző szintjének az áttekintését az 1. ábra mutatja be.

0. szint	**1. szint**	**2. szint**	**3. szint**	**4. szint**	**5. szint**
No automation	Driver assistance	Partial automation	Conditional automation	High automation	Full automation
Nincs automatizálás	Vezetői segítség	Részleges automatizálás	Feltételes automatizálás	Magas automatizáltság	Teljes automatizálás
A vezető elvégzi az összes feladatot.	A vezetési asszisztens funkciói benne vannak.	Automatizált funkciók tartoznak ide.	A sofőr átveszi az irányítást egy adott értesítés során.	A jármű bizonyos körülmények között minden vezetési funkciót ellát.	Teljes körű vezérlés minden körülmények között.
Az emberi járművezetők figyelik a vezetési környezetet.			Az automatizált rendszer figyeli a vezetési környezetet.		

1. ábra – Vezetési automatizálás hat szintje [3] [5]

Jelenleg olyan autógyártók, mint az Audi, vagy a Tesla átvették a SAE 2. szintű automatizálási szabványokat a funkciók fejlesztése során, nevezetesen a Tesla Autopilot és az Audi A80’s Traffic Jam Pilot. Az Alphabet Waymo viszont 2016 óta vizsgál egy olyan üzleti modellt, amely a SAE 4-es szintű önvezető taxis szolgáltatásokon alapul, amely tarifákat generálhat egy korlátozott területen az USA-ban, Arizonában [2] [3] [5].

3 HARDVER ARCHITEKTÚRA

Ebben a részben szeretném részletesen bemutatni a járműre általában felszerelt hardverelemeket, amelyek magas szintű érzékelést és vezérlést végeznek. Ezeknek a hardver kialakításoknak a legfontosabb célja a magas szintű autonóm járműtechnológiák tesztelésének és fejlesztésének az elősegítése. Magába foglalja a magas szintű szabályozást, észlelést és a környezettel való interakciót. A jármű különféle érzékelőkkel, számítógépekkel és kommunikációs eszközökkel van felszerelve, amelyek lehetővé teszik az önálló vezetést. A zord irdőjárási viszonyok például a tükröződés, a hó, a köd, az eső jelentősen befolyásolják az észlelést és a navigációt működtető érzékelők teljesítményét. A közúti autonóm járművek esetében ezeknek a kihívásoknak a bonyolultsága növekszik más járművek váratlan körülményei és viselkedése miatt. Ezért az autonóm járművek előrejelző modulja kritikus az összes jövőbeli helyzetmozgás azonosításához, az ütközési veszélyek csökkentése érdekében.

Az önvezető autók rendszerei kissé eltérhetnek egymástól, mindegyik összetett, amely sok alkomponensből áll. Az autonóm jármű rendszerének architektúrája magába foglalja a hardveres és szoftveres összetevőket. A hardver és a szoftver technikai szempontból a két elsődleges réteg, mindegyik réteg különféle alkomponenseket tartalmaz, amelyek a teljes rendszer különböző aspektusait képviselik. Néhány alkomponens gerincként szolgál a rétegben a hardver és a szoftver réteg közötti kommunikációhoz. Funkcionális szempontból az önvezető autó rendszerek négy elsődleges funkcionális blokkból állnak: észlelés, tervezés és döntés, mozgás és járművezérlés, valamint rendszerfelügyelet. Ezeket a funkcionális blokkokat a feldolgozási szakaszok és az adatgyűjtéstől a jármű vezérléséig tartó információáramlás alapján határozzuk meg [1] [2] [4].

3.1 Beavatkozók

Ahhoz, hogy egy önvezető autót a vezetője teljes mértékben képes legyen irányítani, bizonyos mechanikai és elektromos átalakításokat kell elvégezni a járműn. Az elvégzett módosítások lehetővé teszik a kormánykerék, a gázpedál, a fékpedál és a jármű egyéb kezelőszerveinek emberi beavatkozását, ezáltal lehetőségünk van a biztonságos manőverezésre és a jármű különböző teszteléseire [1].

3.2 Érzékelők (szenzorok)

Az érzékelők olyan eszközök, amelyek a detektált eseményeket, vagy a környezetben bekövetkezett változásokat kvantitatív mérésként térképezik fel további feldolgozás céljából. Az érzékelőket általában két osztályba sorolják a működési elv alapján: proprioceptív és exteroceptív szenzorok. A proprioceptív érzékelők (más néven belső állapot szenzorok) rögzítik a dinamikus állapotot és mérik a dinamikus rendszer belső értékeit (erő, szögsebesség, kerékterhelés, akkumulátor feszültsége). A proprioceptív szenzorokra példák az inerciális mérőegység (IMU), az enkóderek, a tehetetlenségi érzékelők (giroszkópok, magnetométerek) és a pozicionáló érzékelők (GNSS) vevői. Ezzel szemben az exteroceptív érzékelők (más néven külső állapot szenzorok) olyan információkat érzékelnek és szereznek, mint például a távolságmérését vagy a fény intenzitását a rendszer környezetéből. Ilyen exteroceptív szenzorok a kamerák, a rádióérzékelés és távolságmérés (radar), a fényérzékelés és a távolságmérés (LIDAR), valamint az ultrahangos érzékelők. Ezen kívül az érzékelők lehetnek passzív érzékelők, vagy aktív érzékelők. A passzív érzékelők a környezetből kibocsátott energiát veszik fel, hogy elő tudjanak állítani egy kimenetet (ilyen például a kamera). Ezzel szemben az aktív szenzorok energiát bocsátanak ki a környezetbe és mérik az adott energiára gyakorolt környezeti reakciót a kimenetek előállításához a LIDAR és radar szenzorokkal.

Az intelligens szenzor meghatározása az elmúlt évtizedekben sokat fejlődött, az Internet of Things (IOT) megjelenésével együtt, az egymással összefüggő, internethez kapcsolt eszközök rendszerével, amivel emberi beavatkozás nélkül képesek adatokat gyűjteni és továbbítani a vezeték nélküli hálózatokon. Az IOT összefüggésben az intelligens érzékelő olyan eszköz, amely külön számítógép nélkül képes kondicionálni a bemeneti jeleket, feldolgozni és értelmezni az adatokat, döntéseket hozni. Az autonóm járművek kontexusában a környezeti érzékelésre alkalmas tartományérzékelők, mint például a kamerák, a LIDAR-ok és a radarok akkor tekinthetők okosnak, ha az érzékelők például célkövetést, eseményleírást és egyéb információt képesek nyújtani. Ezzel szemben a nem intelligens szenzor olyan eszköz, amely csak az érzékelő nyers adatait vagy hullámformáit kondicionálja és továbbítja az adatokat távoli feldolgozás céljából. Külső számítási erőforrásokra van szükség az adatok feldolgozásához és értelmezéséhez, hogy további információkat nyújtson a környezetének. Egy érzékelő csak akkor tekinthető okosnak, ha a számítógépes erőforrások a fizikai érzékelő tervezésének szerves részét képezik.

Az önvezető autó érzékelőinek beállítása talán a tervezés egyik legkritikusabb eleme. Fontos, hogy megbizonyosodjunk arról, hogy a járművet körülvevő egyéb elemeket ezek a szenzorok megfelelően érzékelik, ezáltal betartva a vezetési szabályokat és a teljes biztonságot. A különböző érzékelő konfigurációk viselkedését már nagyon sokan elemezték szimulált környezetben, tehát vannak tapasztalatok arról, hogy mely a legmegfelelőbb kialakítás, amely elegendő információt szolgáltat a környezetről. A szenzorok kiválasztása mellett nagyon fontos meghatározni az egyes érzékelők fizikai elhelyezkedését a járművön. Az ehhez kapcsolódó kutatások az érzékelők helyzetét sokszor módosítják, esetleg új szenzorokat adnak hozzá, hogy minél jobb eredményeket nyújtsanak ezek az eszközök. Sok esetben egy fém állványzatot szoktak a jármű tetejére szerelni, ahová az érzékelőket csavarok segítségével egyszerűen rögzítik. Az állvány elülső rúdja közepére a kamerát helyezik el, ezzel egy 189 fokos látást, tehát egy jól belátható területet biztosítanak. A fő LIDAR, ami 128 csatornás, szintén középre van felszerelve egy kis szerkezetre, amely extra magasságot bitosít neki a tetőtől és az állványzattól. A két darab kiegészítő 64 csatornás LIDAR-t az állványzat bal és jobb szélén helyezik el. Ezeknek a LIDAR-oknak alacsonyabb a magasságuk és a dőlésszögűk is nagyobb, amely lehetővé teszi a fő LIDAR vakterületeinek feltérképezését. A GNSS (globális navigációs műholdas rendszer) és az IMU (inerciális mérőegység) egységek az állványzat hátulján helyezkednek el.

2. ábra – Az Atlas járműve, az autonóm járműtechnológiák kutatási platformja [1]

Az egyes szenzoroktól kapott információk egyesítéséhez és összeolvasztásához nagy pontossággal kell ismernünk a közöttük lévő relatív helyzetet és orientációt. Különböző algoritmusok segítségével lehetséges ezen pontok automatikus kalibrálása a LIDAR és a kamera szenzorok számára nagy pontossággal.
Az autonóm jármű elsősorban több kamerát, radarérzékelőt, LIDAR érzékelőt és ultrahangos érzékelőt alkalmaz a környezetének érzékeléséhez. Ezen kívül más érzékelőket, köztük a GNSS-t, az IMU-t és a jármű kilométer érzékelőit használják az autó relatív és abszolút helyzetének meghatározására. Az önvezető autó relatív lokalizációja a járművek koordinátáinak a környező tereptárgyakhoz viszonyított hivatkozására utal, míg az abszolút lokalizáció a jármű helyzetére utal egy globális referenciakerethez viszonyítva. Az autonóm járművekben az érzékelők kritikus jelentőségűek a jármű észleléséhez és az elhelyezkedéséhez, az útvonaltervezéshez és a döntéshozatalhoz, amelyek a jármű mozgásának vezérléséhez elengedhetetlen kelléke [1] [2] [3].

3.2.1 Ultrahangos érzékelők

Ezek a szenzorok ultrahangos hullámokat használnak és 20-40 kHz-es tartományban működnek. A hullámokat egy membrán generálja, amelyet az objektumtól való távolság mérésére használnak. A kibocsátott hullám és a visszahangozott jel repülési idejének (ToF) kiszámításával mérjük a távolságot. Az ultrahangos érzékelők hatósugara nagyon korlátozott, általában 3 méternél kisebb.
Az autonóm járművekben ezeket az érzékelőket használják kis távolságok mérésére kis sebességnél. Ezek a szenzorok irányítottak és nagyon szűk sugárérzékelési tartományt biztosítanak, ezért több érzékelőre van szükség a teljes terület lefedésére. Ezen felül több szenzor befolyásolja egymást és szélsőséges távolsági hibákat okozhat. Előnyként jegyezhető meg, hogy ezek a szenzorok bármilyen anyaggal kölcsönhatásban kielégítően működnek, rossz időjárási körülmények között és akár poros környezetben is [3].

3.2.2 Radar

A radar milliméteres hullám spektrumban működik, jellemzően katonai és polgári alkalmazásokban használják őket, például repülőtereken vagy meteorológiai rendszerekben. A radarok nagyobb átjárhatósággal, szélesebb sávszélességgel és az elektromágneses hullámok Doppler-effektus variációjának felhasználásával bármilyen irányban pontosan

meg tudják mérni a rövid hatótávolságú célpontok relatív helyzetét és relatív sebességét. A radarok ezen képessége alkalmassá teszi őket olyan autonóm jármű alkalmazásokra, mint az akadályok, a gyalogosok és a járművek felismerése.

Az modern járművekben különböző frekvenciasávokat alkalmaznak, mint például a 24, 60, 77, 79 GHz és ezek 5 és 200 méter közötti tartományt képesek mérni. A 79 GHz-es radarérzékelőkhöz képest a 24 GHz-es radarérzékelőknek a tartomány, a sebesség és a szög felbontása korlátozottabb, ami problémákhoz vezet a többszörös veszélyek azonosításában és azokra történő reagálásban. Az előrejelzések szerint a jövőben fokozatosan megszüntetik a 24 GHz-es radarérzékelőket. Az autonóm jármű és egy objektum közötti távolság kiszámítása a kibocsátott jel és a vett visszhang közötti ToF (repülési idő) mérésével történik. A járművekben a radar mikro antennák tömbjét alkalmazza, amelyek lebenykészletet generálnak és ezzel javítják a tartomány felbontását, valamint több célpont azonosítását.

Három fő kategóriája van az autóipari radarrendszereknek: a közepes hatótávolságú radar (MRR), a nagy hatótávolságú radar (LRR) és a rövid hatótávolságú radar (SRR). Az autonóm járművek radarérzékelői láthatatlanul integrálva vannak több helyre. SRR-t használnak a burkolat és az ütközés közelségének figyelmeztetéséhez, MRR-t az oldalsó/hátsó ütközéselkerülő rendszerhez és a vakfolt észleléséhez, valamint LRR-t az adaptív sebességtartó automatika és a korai észlelés alkalmazásához. Létfontosságú a radarok felszerelési helyzetének és tájolási pontosságának biztosítása, mivel minden szögeltolódás végzetes következményekkel járhat a jármű működésében. A radarérzékelő általában nem alkalmas tárgyfelismerő alkalmazásokra, mivel a kamerákhoz képest durva felbontásúak. Ezért az önvezető autó kutatói gyakran egyesítik a radarinformációt más szenzoros adatokkal, például a kamerával és a LIDAR-ral, hogy kompenzálják a radarérzékelők korlátjait.

Általánosságban elmondható, hogy a radarérzékelők az autonóm rendszerek egyik legismertebb érzékelője, általában megbízható és pontos észlelést biztosítsanak az akadályokról éjjel-nappal, mivel képesek működni a világítástól és a kedvezőtlen (ködös, havas vagy esős) időjárási viszonyoktól függetlenül. A radarérzékelők hátrányai közé tartozik a csökkent látótér, kisebb pontosság, az érzékelt környezet körüli fémtárgyak, például útjelző táblák vagy védőkorlátok téves felismerése, valamint a statikus, álló tárgyak megkülönböztetésének kihívásai [2] [3] [8].

3.2.3 LIDAR (lézer alapú távérzékelés)

A LIDAR technológiák fejlődése az elmúlt évtizedekben folyamatosan jelentős ütemben fejlődött és jelenleg az Advanced Driver Assistance System (ADAS) és az autonóm járművek egyik legfontosabb technológiája. A lézer alapú távérzékelő (LIDAR) az 1960-as években jött létre, amelyet széles körben alkalmaztak a repülés és az űrkutatás terepének feltérképezésében. A LIDAR egyszerű működési elven alapszik, amely az események közötti idő nagyságrendű, a fény által végrehajtott számlálásán alapul egy impulzusnyaláb visszaszórt energiájával. Ezen időmérések alapján a levegőben lévő fénysebességet használják a távolságok kiszámításához vagy a térképezéshez. A LIDAR használatával történő képalkotáshoz használt mérési elv a repülési idő (ToF), ahol a mélységet az események késleltetésének számlálásával mérjük a forrásból kibocsátott fényben. Így a LIDAR egy aktív, érintés nélküli tartomány keresési technika, amelyben egy optikai jelet vetítenek egy objektumra, amelyet célnak hívunk. A visszaverődött, vagy visszaszórt jelet detektáljuk és feldolgozzuk a távolság meghatározásához. Amint a LIDAR átvizsgálja a környezetét, a jelenet 3D-s ábrázolását generálja pontfelhő formájában.

A LIDAR érzékelők három elsődleges változata, amelyek széles körben alkalmazhatók, az 1D, 2D és 3D LIDAR. A lézer alapú távérzékelők pontok sorozataként adják ki az adatokat 1D, 2D és 3D terekben, más néven pontfelhők formában. Az 1D vagy egydimenziós érzékelők csak a környezetben lévő tárgyak távolság információit (X koordinátáit) mérik. A 2D vagy kétdimenziós érzékelők további információkat nyújtanak a megcélzott objektumok szögéről (Y koordinátáiról). A 3D vagy háromdimenziós érzékelők lézersugarat sugároznak a függőleges tengelyeken, hogy mérjék a tárgyak magasságát (Z koordinátát) a környezet körül. A 3D LIDAR érzékelők esetében a PCD (Point Cloud Data) tartalmazza a jelenetben, vagy a környezetben lévő akadályok X, Y és Z koordinátáit, intenzitásának adatait.

A LIDAR szenzorok tovább kategorizálhatók mechanikus LIDAR, vagy szilárdtest LIDAR kategóriába. A mechanikus LIDAR a legnépszerűbb nagy hatótávolságú környezeti letapogatási megoldás az autonóm járműkutatás és fejlesztés területén. Kiváló minőségű optikát és villanymotorral hajtott rotációs lencséket használ a lézersugarak irányításához és a kívánt látómező rögzítéséhez az önvezető autó körül. A forgó lencsék 360 fokos vízszintes látómezőt képesek elérni, amely lefedi a jármű környezetét. Ezzel szemben a szilárdtest LIDAR-ok kiküszöbölik a forgó lencsék használatát és így elkerülik a mechanikai meghibásodást. A szilárdtest LIDAR-ok sokféle mikrostrukturált hullámvezető segítségével irányítják a lézersugarakat a környezet érzékelésére. Ezek az LIDAR-ok az elmúlt években

egyre nagyobb érdeklődésre tettek szert, mint a forgó LIDAR-ok alternatívái, azok robusztussága, megbízhatósága és általában alacsonyabb költségei miatt mechanikus társaikkal szemben. Ugyanakkor kisebb és korlátozott vízszintes látómezővel rendelkeznek, általában 120 fokos, vagy kevesebb, mint a hagyományos mechanikus LIDAR-ok.
A lézer visszatérés olyan diszkrét megfigyelés, amelyet akkor rögzítenek, amikor a lézer impulzust elfogják és visszaverik a célok. A LIDAR-ok több visszatérést is gyűjthetnek ugyanabból a lézerimpulzusból, a modern érzékelők pedig akár öt visszatérést is rögzíthetnek minden egyes lézerimpulzusból. A LIDAR szenzorok hullámhossza 905 nm (nanométer) és 1550 nm. A 905 nm-es spektrum retinakárosodást okozhat az emberi szemben, ezért inkább a modern 1550 nm-es spektrumot alkalmazzák az autonóm járművekben. A LIDAR maximális hatótávolsága általában 200 méter. Az autonóm járművekhez általában 32, 64, vagy 128 csatornás LIDAR szenzorokat használnak nagy felbontású lézerképek (vagy pontfelhő adatok) előállításához. A LIDAR szenzoroknak kettős szerepe is van az önvezető autókban: egyrészt a LIDAR érzékelők távolságbecslésének pontosságát nagyon jól lehet használni lokalizációs célokra, másrészt az akadályok észlelésére és lokalizálására is tökéletes.
Általánosságban elmondható, hogy jelenleg a 3D-s forgó LIDAR-okat gyakrabban alkalmazzák az önvezető járművekben a szélesebb látómezője, távolabbi érzékelési tartománya és mélységérzékelése miatt. Megbízható és pontos észlelést biztosít nappal és éjszaka is. A megszerzett adatok pontfelhő formátumban sűrű 3D térbeli ábrázolást biztosítanak az autonóm jármű környezetéről. A LIDAR szenzorok nem szolgáltatnak színes információkat a környezetről, összehasonlítva a kamerarendszerekkel. Bár a LIDAR mérési pontossággal és 3D érzékeléssel felülmúlja a milliméteres hullámú radart, teljesítménye súlyos időjárási körülmények között, például hóban, esőben és ködben nem megfelelő. Ezen kívül a működési tartományának észlelési képessége az objektum tükröződésétől függ. Ezek az okai annak, hogy a 3D-s pontfelhőt gyakran egyesítik különböző szenzorok adataival, szenzorfúziós algoritmusok segítségével [1] [2] [3] [6] [8].

3.2.4 Kamera

A kamerák az egyik leginkább elfogadott technológia a környezet érzékelésére. A kamera azon az elven működik, hogy egy fényérzékeny felületen (képsíkon) az érzékelő elé szerelt lencsén keresztül érzékeli a fényeket, hogy tiszta képeket készítsen a környezetről. A kamerák viszonylag olcsók és megfelelő szoftverrel rendelkeznek, így a mozgó és a statikus

akadályokat egyaránt képesek észlelni a látómezőjükön belül és nagy felbontású képeket nyújtanak a környezetről. Ezek a képességek lehetővé teszik a jármű szenzor rendszerének, hogy közúti közlekedési járművek esetében azonosítsa a közúti jelzőtáblákat, a jelzőlámpákat, a közúti sávjelzéseket és az akadályokat.

Az autonóm jármű kamerarendszere monokuláris, binokuláris kamerákat, vagy ezek kombinációját alkalmazhatja. Ahogy a neve is mutatja, a monokuláris kamerarendszer egyetlen kamerát használ a képsorok létrehozásához, amely három sávra oszlik: piros, zöld és kék (RGB). Ugyanolyan hullámhosszt használnak, mint az emberi szem, azaz 400-780 nm. A hagyományos monokuláris kamerák alapvetően korlátozottabbak, mint a sztereó kamerák, mivel hiányoznak a natív mélységi információk, bár egyes alkalmazásokban, vagy a fejlettebb monokuláris kamerákban, amelyek dual-pixeles autofókusz hardvert használnak, a mélységi információkat összetett algoritmusok segítségével lehet kiszámítani. Ennek eredményeként gyakran két kamerát telepítenek egymás mellé, hogy binokuláris kamera rendszert hozzanak létre az autonóm járművekben. A sztereó kamera, más néven binokuláris kamera, utánozza az állatok mélységének érzékelését, ahol az egyes szemekben képződő kissé eltérő képek közötti különbségeket a mélység érzetének biztosítására használják. A sztereó kamerák két képérzékelőt tartalmaznak, alapvonallal választva el őket. Az alapvonal kifejezés a két képérzékelő közötti távolságra utal és a kamera modelljétől függően eltér. Egy ilyen szolgáltatás lehetővé teszi, hogy a kamera (RGBD) 3D képet kapjon a jármű körüli helyzetről, ahol a D a mélységet jelöli.

A kamera maximális hatótávolsága 250 méter körüli az objektív minőségétől függően. A kamera technológia mindenütt elérhető, amely nagy felbontású videókat és képeket nyújt, beleértve az érzékelt környezet színét és textúráját is. A képadatokat gyakran egyesítik más szenzoradatokkal, például radar és LIDAR adatokkal, hogy megbízható és pontos környezeti érzékelést hozzanak létre. Az autó tetejére általában három monokuláris kamerát szoktak felszerelni, 90 fokos optikai vízszintes látómezővel. Ez a kamera lefedi a jármű elülső részének nagy részét, biztosítja a színes képeket, amelyeket később felhasználnak az akadályok besorolásához (ilyen például a gyalogosok, kerékpárosok, vagy más autók), valamint arra is szolgál, hogy átlátó elemzésre is képes az úttestről, ide értve a közúti sávtopológiáját is.

A kamera fő előnye, hogy pontosan képes összegyűjteni és rögzíteni a környezet textúráját, színeloszlását és kontúrját. A megfigyelési szög azonban korlátozott a kamera lencséjének szűk látószöge miatt. A kamerák által rögzített képek minőségét (felbontását) jelentősen befolyásolhatja a kedvezőtlen időjárási viszonyok ilyen például az intenzív napsütés,

viharok, havazások, ködös időjárások és a rossz fényviszonyok. A kamerák egyéb hátrányai között szerepelhet a nagy számítási teljesítmény követelménye a képadatok elemzésekor [2] [3] [8].

3.2.5 GNSS és IMU

A GNSS (globális navigációs műholdas rendszer) a Föld felszínén keringő műholdak egy részét használja a lokalizáláshoz. A rendszer tartalmazza az önvezető jármű helyzetének, sebességének és pontos idejének adatait. A műhold által kibocsátott jel és a vevő közötti repülési idő (ToF) kiszámításával működik. A jármű pozíció általában a GPS koordinátából származik. A GPS által kinyert koordináták nem mindig pontosak és általában hibát okoznak a helyzetekben 3 méter átlagértékkel és 1 méter szórással. A teljesítmény városi környezetben tovább romlik, a helyzethiba akár 20 méterre is növekedhet és néhány szélsőséges esetben a GPS helyzet hibája elérheti a 100 méter körülit.
Odometria néven ismert technikával lehet mérni a jármű helyzetét úgy, hogy a forgásérzékelőket a jármű kerekeire rögzítik. Annak érdekében, hogy az autonóm jármű képes legyen megcsúszás vagy oldalirányú mozgások detektálására, az inerciális mérőegységet (IMU) használják. Az IMU minden egységgel együtt kijavítja a hibákat és növeli a mérőrendszer mintavételi sebességét. A GNSS kombinálható IMU technikával, ami megerősíti és javítja az önvezető autó helyzetbecslését [1] [3].

3.3 Feldolgozó egységek

Minden érzékelő nagy mennyiségű adatot állít elő, amelyet fel kell dolgozni. A feldolgozó egységek az önvezető autó drága részei is lehetnek. Emiatt meg kell határozni az összes algoritmus valós idejű futtatásához szükséges minimális CPU és GPU teljesítmény mennyiségét. Általában különálló számítógépek között osztják el a számításokat, itt most egy 3 számítógépes esetet mutatok be. Ezzel a megoldással több folyamatot párhuzamosan is végre lehet hajtani, így eloszlható és kiegyenlíthető a számítási terhelések. Minden egyes feldolgozó egységnek különböző képességei vannak, amelyeket kifejezetten a végrehajtandó feladatokhoz választották ki.
Az első a vezérlő számítógép, ami CAN (Controller Area Network) buszon keresztül kezeli a kommunikációt az autó rendszerekkel. Ezen kívül felelős a vezérlő és az útvonaltervezési algoritmusok végrehajtásáért, amelyek lehetővé teszik a jármű önálló mozgását. Ez a

számítógép gyors CPU-val rendelkezik, amely képes elvégezni a szükséges műveleteket valós időben. A második számítógép a lokalizációért és annak leképezéséért felel. Ez felel a három LIDAR által biztosított hatalmas mennyiségű adat feldolgozásáért, valamint a LIDAR odometria és térkép készítéséért. Ezen kívül elvégzi az összes többi lokalizációs feladatot, beleértve a GNSS és IMU érzékelőket. Mivel a legtöbb LIDAR feldolgozó algoritmust csak a CPU hajtja végre, ezért ebben a számítógépben is gyors processzornak és legalább 16 GB RAM memóriának kell lennie, ami elegendő az összes adat feldolgozásához az érzékelő sebességével. A harmadik számítógép az akadályok osztályozásának és a helyzetek megértésére használt összes mély tanulási algoritmus végrehajtásáért felel. Ennek a számítógépnek számítási képességei magasak, általában csúcskategóriás grafikus kártyával felszereletek, amely lehetővé teszi a jármű számára összes szükséges algoritmus végrehajtását. Ez a számítógép az autó belsejében lévő képernyőhöz is csatlakozik, amely interfészként működik a felhasználóval [1] [2].

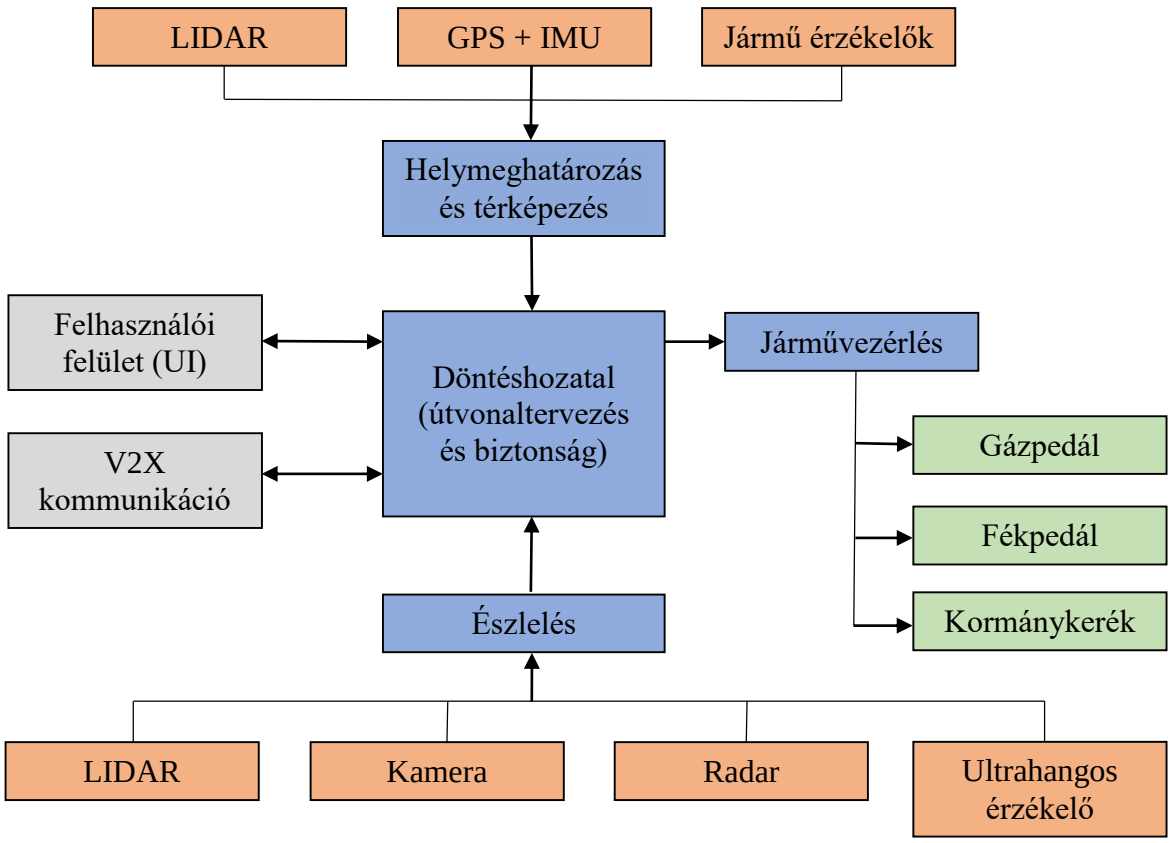
3.4 Kommunikáció

Az érzékelők által szolgáltatott összes nyers adatot és a számítógépek által generált feldolgozott adatokat meg kell osztani egymással. Egy szenzor adataira egynél több számítógépnek is szüksége lehet, az algoritmus által generált feldolgozott adatokra pedig másik számítógépre lehet szükség. Ezen információk megosztása érdekében csillag topológiát alkalmaznak, amelyben az összes eszköz egy gigabites kapcsolóhoz csatlakozik. A topológia legfőbb előnye, hogy könnyen bővíthető, ha szeretnénk több érzékelőt vagy számítógépet adni a hálózathoz.

A járműhálózaton belüli rendszerek közötti kommunikáció mellett internetkapcsolatra is szükség lehet a GPS differenciális korrekciók megszerzéséhez, digitális térképek letöltéséhez, vagy a V2X (Vehicle to Everything) hálózatokhoz való csatlakozáshoz. Ebből a célból általában egy 4G útválasztó csatlakozik a rendszerhez, amely internet hozzáférést biztosít a hálózat minden számítógépéhez. A kommunikációs modul több olyan képességet is kínál, amelyek hasznosak az intelligens városokban folyó kutatások folytatásához, valamint az autonóm járművek és más szervezetek közötti együttműködéshez [1].

4 SZOFTVERARCHITEKTÚRA

Az autonóm járművek szoftverarchitektúrája sokszor egyedi kialakítású. Az egyedi tervezésű szoftverarchitektúrák azért előnyösek, mert így extra szintű rugalmasságot biztosítanak a különféle kutatásokhoz. Ez különösen hasznos lehet a nagyon specifikus kísérletek tervezésénél, mivel lehetővé teszi a kísérlethez szükséges szoftvermodulok testreszabhatóbb integrálását. A javasolt szoftverarchitektúra a Robot Operációs Rendszeren (ROS) alapul. A ROS egy keretrendszer a robotikai alkalmazások fejlesztéséhez és integrálásához, amit széles körben használnak az autonóm vezetési szoftverekben is. Ez a rendszer biztosítja a folyamatok közötti kommunikáció köztes szoftvert és lehetővé teszi a különböző program modulok elosztott módon történő telepítését. Ezen kívül a ROS sok eszközt kínál a kutatók számára a feladatok kidolgozásában és a hibakeresésében. A 3. ábra mutatja be azokat a fő alapelemeket, amelyek a jármű javasolt szoftverarchitektúráját alkotják.



3. ábra – Az önvezető autó szoftverarchitektúrája [1] [4]

Minden szoftvermodul felelős egy adott feladat végrehajtásáért. A modulok közötti kapcsolatok előre meghatározott formátumokat követnek a ROS interfészek alapján. Ez általában lehetővé teszi az ugyanazon feladatot megoldó, különböző komponensek felcserélhetőségét, amennyiben követik a megadott bemeneti és kimeneti interfészeket. Így az architektúrában jelen lévő modularitás ideális a különböző megoldások összehasonlítására és tesztelésére, így tehát az architektúra a kutatás szempontjából egyedülállóvá válik.
Ezzel az elrendezéssel az adatok, az érzékelőtől a több feldolgozó modulig áramlanak, végül generálják a vezetési parancsokat a kormányzáshoz, a gyorsításhoz és a fékezéshez. A jármű képes kommunikálni a felhasználókkal egy grafikus felhasználói felületen keresztül, amelyen megadható a cél. Ezen kívül más entitásokkal való kommunikáció egy V2X modulon keresztül is megvalósítható [1] [2].

4.1 Információk gyűjtése

Az adatgyűjtő modulok kiolvassák a szenzorok nyers adatait. A nyers adatok beérkezése után ezeknek a moduloknak kettő feladata van. Az egyik, hogy az szenzorokból származó bemenő nyers adatok előfeldolgozási lépést igényelhetnek. Ez az előfeldolgozási lépés állhat egy szűrő algoritmusból az érzékelő zajának megtisztítására, vagy más típusú algoritmusból, hogy összetettebb információt generáljon a kapott egyszerű adatokból. A másik feladata, hogy át kell alakítaniuk a rendelkezésre álló érzékelő adatokat ROS msg formátumba, annak érdekében, hogy megoszthassák azokat a rendszer többi moduljával. Az érzékelőktől kapott információkat standard formátumokká konvertálják és így az architektúra többi alkotóelemei képesek elvégezni feladataikat, függetlenül a használt érzékelőktől [1].

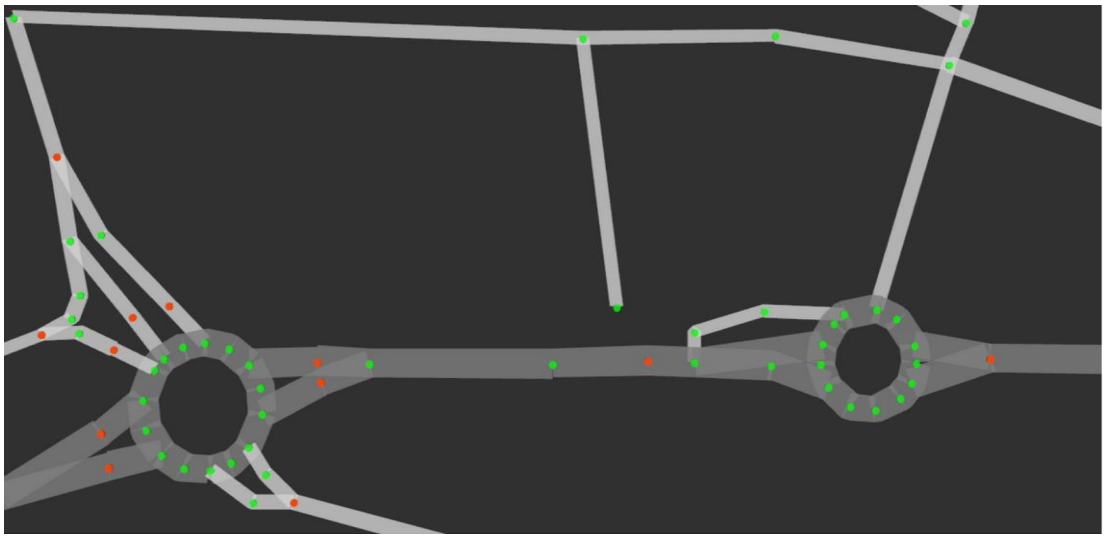
4.2 Helymeghatározás

A helymeghatározás kritikus feladat egy autonóm jármű számára. A biztonságos navigálás érdekében különböző szenzorokat és algoritmusokat használ ennek a feladatnak a végrehajtásához. Kettő típusra osztható a helymeghatározás: lokális és globális. A lokális helymeghatározó feladat, amely közismert néven a jármű odometriája, kiszámítja a jármű mozgását a kiindulási helyzetéhez viszonyítva, a transzláció és a forgás mérésével. Ezek a módszerek általában helyileg nagyon pontosak, de hosszú távú lokalizációra alkalmatlanok.

Az autónom járműveken általában két különböző odometriai módszert alkalmaznak. Az egyik a kerék kilométer mérése, a másik a LIDAR odometria. Az első módszer a jármű által biztosított sebességet és a kormányzási szöget használja. Ezen mérések integrálásával a kerék odometria algoritmus képes biztosítani a jármű transzlációját és forgását. A kerék odometria módszer mellett a javasolt megközelítés magában foglal egy exterocetív odometria módszert is, amely LIDAR 3D pontfelhőkön alapul. A rendszerbe integrált LIDAR odometria módszer a LOAM (Lidar Odometry and Mapping), amely valós időben nagyon megbízható odometriát képes előállítani, miközben a helyzet 3D térképét is elkészíti. Bár a javasolt odometriai módszerek gyors frissítési sebességgel és nagy helyi pontossággal bírnak, egy autonóm jármű számára nem elegendőek. A helyi módszerekkel biztosított lokalizáció arra a pontra vonatkozik, amikor a jármű elindult, tehát nem adnak meg lokalizációs koordinátákat globális közös keretre vagy digitális térképre hivatkozva. Erre a referenciára van szükség a valós világban való navigáláshoz. Tehát két különböző globális lokalizációs módszert is integrálnak, azzal a céllal, hogy a jármű helyzete globális referenciát adjon. Az egyik ilyen a GNSS szenzor, ami jelenleg a leggyakoribb globális lokalizációs forrás, mind az autonóm, mind az ember által vezérelt autókban. Ezeket az információkat az IMU által szolgáltatott adatokkal is kombinálják, hogy megbecsüljék a jármű tájolását is. Ezeknek az érzékelőknek az a nagy előnyük, hogy a fogadott adatok nem igényelnek jelentős feldolgozást, csak a szélességi és hosszúsági koordináták UTM-re történő átalakítását. A kapott helyzet pontosságát azonban gyakran csökkentik a magas épületek, vagy az alagutak. Éppen ezért - bár a GNSS alapú lokalizáció globális helymeghatározást biztosít - önmagában nem elegendő az autonóm jármű számára szükséges robosztus és pontos helymeghatározás biztosítására. A másik módszer az adaptív Monte Carlo lokalizációs algoritmus, amely képes kiszámítani a jármű globális helyzetét egy referencia térképen belül. Ez a részecskeszűrőn alapuló módszer három különböző bemenetet igényel: a terület globális térképét, az aktuális LIDAR mérést és a jármű lokális odometriáját. A lokalizációs folyamat a részecske súlyának frissítéséből áll, annak költsége alapján, hogy a LIDAR mérés illeszkedik az egyes részecskék által adott helyzetben lévő térképhez. Lokális odometriát használnak a részecskék szaporítására az előrejelzési lépésben. A végső becslést a részecskekészlet által képviselt poszterből számítják ki [1] [2] [3].

4.3 Térképezés

Az önvezető jármű a térképeket az útvonalak előállításához és a környéken való eligazodáshoz használja. A leképezési technikák globális és lokális térképezésbe sorolhatók. Az útvonal megtervezéséhez és létrehozásához globális térképre van szükség a vezetési területről. Ezt a térképet a globális lokalizáció referenciájaként is felhasználják. Egy terület globális térképét általában offline módon készítik el, a korábbi szekvenciákból származó több mérés integrálásával a leképezendő terület köré. Néhány nagy hatékonyságú SLAM (Simultaneous Localization and Mapping) módszer, például a LOAM lehetőséget nyújt pontos térképek valós időben történő elkészítésére. Ezen kívül a globális térkép beszerezhető digitális térképszolgáltatóktól is, ilyen például az OpenStreetMaps.



4. ábra – OpenStreetMap adatokból épített digitális térkép [1]

A globális térképek fő hátránya, hogy statikusak. Globális tervezéshez használják őket, ettől függetlenül a dinamikus akadályokat nem ismerik fel, például más gépkocsikat, vagy gyalogosokat és ezért használhatatlan az autonóm vezetéshez. Ennek az információhiánynak a kompenzálása érdekében a rendszerbe helyi térképek kerülnek be. A helyi térképek online módon készülnek és csak a jármű legközelebbi környezetét fedik le. Tehát a LIDAR szenzor információi például valós időben kerülhetnek feldolgozásra. Ezzel a módszerrel egy foglaltsági rács térképet (OGM) lehet generálni, amelyet később a tervezési és navigációs modulok használnak a meglévő akadályok kezelésére [1] [3].

4.4 Észlelés

A járműbe integrált környezeti érzékelési modulok felelősek a nyers érzékelőadatok elemzéséért és a helyzetről szóló értelmes információk kinyeréséért. Ezek a modulok két csoportba sorolhatók, az egyik az akadályok felderítése és osztályozása, a másik pedig a helyzet megértése. Először is egy autonóm járműnek képesnek kell lennie a többi közúti jármű, kerékpár, gyalogos stb azonosítására. Ezt a nehéz feladatot különböző technikákkal lehet végrehajtani. Az utóbbi évek tendenciája azonban az volt, hogy mély tanulási módszereket alkalmaznak a jármű körüli különböző akadályok felderítésére és osztályozására. Az általam vizsgált autonóm jármű kialakításban két különböző módszert is integráltak. Az egyik a kamera által készített képek elemzését végzi el, ez az algoritmus a tárgyak képen való elhelyezkedését és azok orientációját mutatja. A másik módszer a LIDAR szenzor az elérhető 3D információit használja fel a közúti járművek észlelésére és osztályozására is. Az akadályok mellett a városban való navigáláshoz egy autonóm járműnek képesnek kell lennie a helyszín különböző elemeinek a felismerésére. Emiatt számos helyzet megértési módszer is integrálva van a rendszerbe. A fő algoritmus a kamera képeinek teljes szemantikus szegmentálását hajtja végre, információkat szolgáltatva a környezet különböző elemeiről [1] [2] [7].

5. ábra – Szemantikus szegentáció végrehajtása egy kamera felvételen [7]

4.5 Biztonság

Az autonóm járművek egyik legkritikusabb kérdése a biztonság. A járműnek képesnek kell lennie bármilyen rendellenesség, vagy probléma észlelésére, majd át kell adnia a vezetés kezelőszerveit az emberi vezetőnek. Ennek megfelelően a javasolt architektúra magában foglalja a rendszer folyamatos introspektív elemzését, hogy értékelje azokat a biztonsági feltételeket, amelyek meghatározzák, hogy a jármű mikor képes önálló üzemmódban haladni. Ezeket a feltételeket különböző tényezők alapján értékelik. Az öntudat mutatója ellenőrzi az érzékelők bemeneteinek állapotát és a rendszerben futó egyes szoftverfolyamatok működését. A másik ilyen fontos tényező a helymeghatározási pontosság. Amikor a jármű nem képes az autonóm navigációhoz megfelelő pontossággal lokalizálni a világot, akkor a vezérlőparancsok váratlan és nem kívánt viselkedést eredményezhetnek. Tehát ez a tényező az önlokációs rendszer állapotát vizsgálja és szükség esetén az ember átveszi a jármű vezetését. Előfordulhat olyan eset is, amikor a jármű nem képes elérési utat létrehozni a rendeltetési helyre, vagy amikor a létrehozott út nem követhető (például ösvény). Ez a tényező az útvonaltervezésből és a navigációs modulokból származik. Ilyen esetben is megkövetelhető az autó irányításának átvétele az ember által [1] [3].

4.6 Útvonal tervezése

Az útvonaltervezés feladata egy járható útvonal (pálya) létrehozása, amelyet a jármű követhet a cél eléréséhez. A pályalétrehozás összetettségének kezelése érdekében az útvonaltervezés általában globális tervezésre és lokális tervezésre oszlik. A globális tervezés figyelembe veszi az összes rögzített akadályt, például az utak jellemzőit (sávok száma, szélesség, összeköttetések a különböző utak között), hogy megtalálják a legrövidebb utat a jármű jelenlegi helyzete és a cél rendeltetési helye között. A globális útvonal tartalmaz információkat az egyes utak, vagy sávok maximális sebességéről is, amelyeken a járműnek menni kell. Az 6. ábra egy globális útvonalat mutat (piros vonal) az aktuális helyzet (kék pont) és a cél (zöld pont) között, megjelenítve az összes OSM (OpenStreetMaps) csomópontot (piros pontok).

6. ábra – Globális útvonaltervezés OpenStreetMaps-ben [1]

A lokális tervezés megpróbálja követni a globális tervet azzal a különbséggel, hogy figyelembe veszi azokat a helyi akadályokat, amelyek nem szerepelnek a globális térképen, de különböző algoritmusokkal észlelhetők. Ezen kívül a modul által generált pályának meg kell felelnie bizonyos korlátozásoknak, amelyek a jármű által követhetővé teszik az utat. Mindezeket az információkat egy frenet keretbe transzformálják az optimális pálya meghatározása érdekében, majd ezt a pályát visszaalakítják valós koordinátákká, hogy a vezérlő modul képes legyen követni azt. Így a jármű képes megelőzni a lassabb járműveket, ha lehetséges, vagy ennek megfelelően gyorsulhat vagy fékezhet, célként mindig a referencia globális utat és a célsebességet fenntartva [1] [2].

4.7 Navigáció és vezérlés

Az úti környezetben történő navigáció nagyban függ a lokalizáció és az útvonaltervezés moduljaitól. Miután a jármű helyesen lett lokalizálva, követnie kell a létrehozott lokális tervet. Az erre a célra alkalmazott útkövető Stanley algoritmus, amely egy oldalirányú vezérlő, mely minden egyes lépésnél kimenti a kormányzási szöget, valamint a járművet az út követésére készteti. A kormányparancsokat a lokális tervező által biztosított célsebességgel együtt elküldik az alacsony szintű vezérlőmodulnak, amely ennek megfelelően mozgatja a kormánykereket, valamint a gázpedált és a fékpedált [1] [2].

5 SZENZOROK FÚZIÓJA

A szenzorfúzió folyamat ma már minden önvezető autó rendszerben szükséges az egyes szenzortípusok hiányosságainak kiküszöbölésére, javítva ezzel a hatékonyságát és megbízhatóságát. A szenzorok pontos kalibrálása elengedhetetlen a további feldolgozási lépésekhez, például az érzékelők összevonásához és az akadályok észleléséhez, lokalizálásához és feltérképezéséhez, valamint az irányításhoz szükséges algoritmusok megvalósításához. A szenzorfúzió az autonóm autó alkalmazások egyik alapvető feladata, amely egyesíti a több érzékelőtől kapott információkat, hogy csökkentse a bizonytalanságokat az érzékelők egyedi használatához képest. A fúziós algoritmusokat elsősorban az architektúra észlelési blokkjában használják, amely magában foglalja az objektumdetektálási folyamatokat.

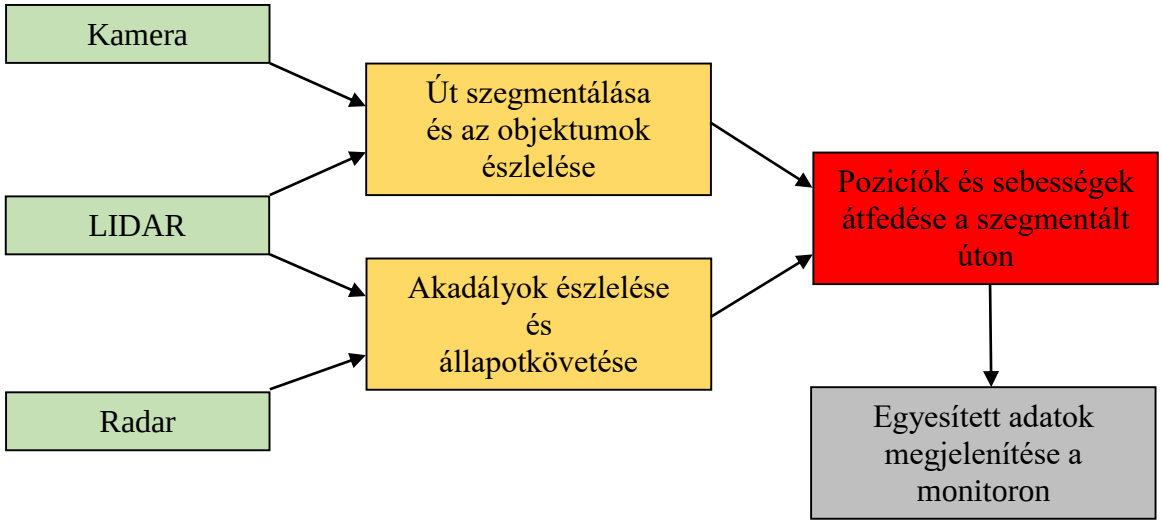
Ezen kívül a szenzorfúzió segít olyan konzisztens modell kidolgozásában, amely a különböző környezeti körülmények között pontosan érzékeli a környezetet. Például a kamera és a radar fúziója nagy felbontású képeket és a jelenet észlelt akadályainak relatív sebességét szolgáltathatja. Jelenleg az akadályok detektálására három elsődleges érzékelő kombináció van elterjedve a szakirodalomban: a kamera - LIDAR (CL), a kamera - radar (CR) és a kamera – LIDAR - radar (CLR) érzékelő kombinációk. Egy felmérés szerint a CR érzékelő kombináció a leginkább alkalmazható a több érzékelős fúziós rendszerekben a környezet érzékelésére. A CR érzékelő kombináció nagyfelbontású képeket kínál, miközben további információkkal szolgál a környező akadályok távolságáról és sebességéről. A Tesla CR érzékelők kombinációját és más érzékelőket, például ultrahangos érzékelőket használt a jármű környezetének érzékelésére. A CLR szenzor kombinációja nagyobb tartományban képes felbontást biztosítani és a LIDAR pontfelhőkön és a mélység térképen keresztül pontosan megérti a környezetet. Ez javítja a teljes autonóm rendszer biztonsági redundanciáját is.

Három elsődleges megközelítés létezik a különböző érzékelési módok szenzoros adatainak egyesítésére: magas szintű fúzió (HLF), alacsony szintű fúzió (LLF) és középszintű fúzió (MLF). A magas szintű fúzió megközelítésben minden érzékelő objektumdetektálást vagy nyomkövető algoritmust hajt végre önállóan, majd ezután fúziót hajt végre. Ezt a megközelítést a gyakran alacsonyabb relatív komplexitás miatt alkalmazzák. A magas szintű fúzió azonban nem ad megfelelő információt, mivel az alacsonyabb megbízhatósági értékű osztályozásokat elvetik, ha például több átfedő akadály áll fenn. Ezzel szemben az alacsony

szinű fúzió megközelítéssel az egyes szenzorok adatait egyesítik az absztrakció legalacsonyabb szintjén (nyers adatok). Itt minden információ megmarad, ami javíthatja az akadályok észlelésének pontosságát. A gyakorlatban az alacsony szintű fúzió megközelítés számos kihívással jár, nem utolsósorban a megvalósítás során. Az érzékelők pontos külső kalibrálását igényli, hogy pontosan összeolvashassák a környezetről alkotott érzékelésüket. Az érzékelőknek emellett egyensúlyba kell hozniuk az egomozgást (egy rendszer 3D-s mozgása a környezetben), és időbeli kalibrációval kell rendelkezniük. A középszintű fúzió, vagy más néven funkciószintű fúzió egyesíti a megfelelő szenzoradatokból kinyert többcélú szolgáltatásokat (nyers mérések), például képek információi, vagy a radar és az LIDAR helymeghatározási jellemzői. Ezt követően felismerik és osztályozzák az egyesített multiszenzoros jellemzőket.

A szenzorfúziós technikákat és algoritmusokat az elmúlt évek során alaposan tanulmányozták és mostanra jól megalapozottak az irodalomban. Ezeket a technikákat és algoritmusokat klasszikus szenzorfúziós algoritmusokba és mély tanulási szenzorfúziós algoritmusokba sorolták. A klasszikus szenzorfúziós algoritmusok, mint például a tudásalapú módszerek, a statisztikai módszerek, a valószínűségi módszerek az adatok tökéletlenségéből fakadó bizonytalanság elméleteit használják, ideértve a pontatlanságot és a bizonytalanságot az érzékelő adatok fúziójára. A mély tanulási szenzorfúziós algoritmusok különféle, többrétegű hálózatok létrehozását foglalják magukban. Ezek lehetővé teszik számukra a nyers adatok feldolgozását és a funkciók kivonását a kihívást jelentő és intelligens feladatok elvégzéséhez, ilyen például az objektum észlelése városi környezetben. A következőkben szeretnék bemutatni egy többszenzoros fúziós algoritmus megoldást. Ez a fúziós algoritmus két részből tevődik össze, amelyek párhuzamosan futnak egymás mellett, melyet a 7. ábra jól szemléltet. Mindegyik részben a szenzorok egy meghatározott konfigurációját és egy fúziós módszert alkalmaznak, amely a legjobban megfelel a szóban forgó fúziós feladatnak. Az első rész az objektumok osztályozásának, lokalizálásának és szemantikus útszegmentálásának nagy felbontású feladataival foglalkozik a kamera és a LIDAR érzékelők segítségével. A kamera nyers képét és a LIDAR mélységi csatornáját egyesítik, mielőtt elküldenék őket egy mély neurális hálózatra (DNN), amely az objektumok osztályozásáért és az útszakaszolási feladatokért felel. Az olyan mély hálózatok, mint a konvolúciós neurális hálózatok (CNN) és a teljesen konvolúciós hálózatok (FCN) jobb teljesítményt és pontosságot mutattak a számítógépes látás feladatoknál, mint a hagyományos módszerek, azaz a funkciógörbék és a klasszikus gépi tanulás (ML) algoritmus. A kamera és a LIDAR kombinációja az FCN architektúrával a legjobb

kombinációt biztosítja számunkra az osztályozás és a szegmentálás elvégzéséhez. Az automatizált vezetéshez elengedhetetlen az út vezethető terének és akadályosztályainak ismerete az útvonal tervezéséhez és a döntéshozatalhoz.



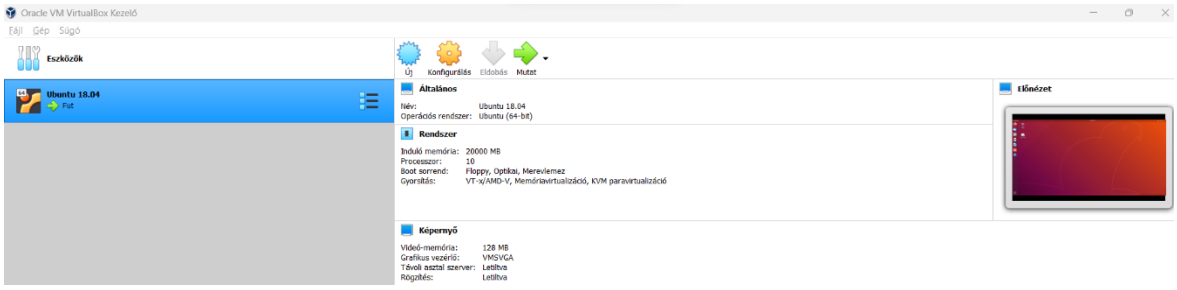
7. ábra – Többszenzorból álló fúziós algoritmus megoldás [8]

A második rész tárgyak felismerésével és állapotuk követésével foglalkozik LIDAR és radar szenzorok segítségével. A LIDAR pontfelhő-adatok (PCD), a radarjelek feldolgozása és egyesítése az objektum szintjén történik. A LIDAR és a radar adatfeldolgozás az útvonalon lévő akadályok csoportjait eredményezi a megfigyelt területen. A feldolgozott érzékelő adatok objektum szintű fúzióját késői fúziónak nevezzük. Az így kapott késői fuzionált LIDAR és radar adatokat egy állapotbecslési módszerhez továbbítják, hogy az egyes érzékelők zajos mért állapotait a legjobban kombinálják. Az állapotbecsléshez választott módszer a Kálmán-szűrő, ami lineáris mozgásmodellek feltételezésén működik. Mivel az akadályok, például az autók mozgása nem lineáris, ezért egy módosított változatát, nevezetesen a kiterjesztett Kálmán-szűrőt használjuk az állapotbecsléshez. Az úton lévő akadályok állapotának ismerete és nyomon követése segít megjósolni és figyelembe venni viselkedésüket az útvonaltervezésnél és döntéshozatalnál. Végül pedig átfedjük az egyes fúziós kimeneteket és megjelenítjük őket az autó belső monitorán [2] [3] [8].

6 LIDAR FILTER PROJEKT BEMUTATÁSA

Az alábbi projektfeladat megvalósítása azt a célt szolgálja, hogy bemutassam a LIDAR szenzorral kapcsolatos pontszűrési lehetőségeket, elméleteket, matematikai összefüggéseket, algoritmusokat, a Robot Operating System használatát, markerek létrehozását. A projektem elkészítését megelőzően a 2022-ben megjelent Urban Road Filter tudományos cikket és a hozzátartozó forráskódot tanulmányoztam hosszas ideig. Sok esetben ebből merítettem ötleteket a projektfeladatom elkészítéséhez. A projektfeladatom bemutatása a forráskód fontosabb szakaszainak részletezéséből és az ehhez kapcsolódó szakirodalmakban talált elméletekből tevődik össze.

Először is szeretnék néhány mondatot írni a környezetről, amiben a projektfeladatot elkészítettem. Hardveres erőforrásként egy Asus FX507ZR típusú laptopot használtam, amiben egy Intel Core i-12700H 2.30 GHz típusú processzor, 32 GB RAM memória, Nvidia GeeForce RTX 3070 típusú videókártya és 1TB SSD tárhely volt. Szoftverek tekintetében a laptopon alapesetben egy Microsoft Windows 11-es operációs rendszer volt telepítve. Erre a rendszerre feltelepítettem az Oracle VirtualBox 6.1. verziójú alkalmazást, mellyel képesek vagyunk virtuális gépeket telepíteni, futtatni, felügyelni. A programnak a segítségével létrehoztam egy virtuális gépet, melyre Ubuntu 18.04 verziójú operációs rendszert telepítettem. A konfigurációs részben beállítottam, hogy a processzor 14 magjából használjon 10 magot és a memória tekintetében pedig a 32 GB-ból használjon 20 GB-ot a virtuális gép. A 8. ábra szemlélteti az Ubuntu 18.04 virtuális gép információit. Ezzel meghatároztam az Ubuntu operációs rendszer teljesítményét és kialakítottam egy stabil munkakörnyezetet.



8. ábra – Ubuntu 18.04 virtuális gép információi

Ezek után az Ubuntu rendszerre telepítettem a Robot Operating System nyílt forráskódú middleware szoftvercsomagját, abból is a Melodic Morenia nevű disztribúciót. Erről a

szoftverről a következő részben bővebben fogok írni. A forráskód írásához a szintén nyílt forráskódú Visual Studio Code 1.75.1. verziójú kódszerkesztőt választottam, ami támogatja a hibakeresőket, valamint beépített GIT verziókezelő támogatással rendelkezik, továbbá képes az intelligens kódkiegészítésre az IntelliSense segítségével. A szoftverek tekintetében még egy programot telepítettem az Ubuntu rendszerre, ez pedig a Terminator 1.91. verziója. Ez egy terminálemulátor alkalmazás, mellyel kényelmesen egy ablakban több terminált nyithatunk meg egymás mellett. A projektfeladat folyamatos tesztelésekor, futtatásakor nagy hasznomra vált ez a szoftver. Miután telepítésre kerültek a szükséges szoftverek, elkezdhettem a program fejlesztését.

6.1 Robot Operating System (ROS)

A robotika fejlődése egyre nagyobb hatással van az emberek mindennapi életére. A feltörekvő tervezések mellett, ahol nagy teljesítményű számítógépekre van szükség, egyre kifinomultabb szoftvereket fejlesztenek, amelyek nélkül nem léteznének autonóm gépek.
A Robot Operációs Rendszer (ROS) a robotok működtetésére szolgáló alapvető szoftverkészletet kínálja, amely bővíthető meglévő csomagok létrehozásával vagy használatával, lehetővé téve a különböző hardverplatformokon újra felhasználható robotszoftverek írását. Stabil disztribúcióként több ezer csomaggal, beágyazó algoritmusokkal, érzékelő-illesztőprogramokkal ez az elsődleges szoftver a robotika számára. Architektúra szempontból a ROS egy köztes (middleware) szoftverréteg, amely meghatározott robot platformokon, meglévő operációs rendszerek és a felhasználók által létrehozott alkalmazások között helyezkedik el. Számos helyen alkalmazzák a szoftvert: többek között mobil robotokhoz, manipulátorokhoz, autonóm járművekhez, humanoid robotokhoz, pilóta nélküli légi járművekhez.
2007-ben hozták létre Kaliforniában a Stanford Egyetem és egy robotikai cég a Willow Garage együttműködésének köszönhetően. A platformot létrehozása óta folyamatosan fejleszti egy robotikával foglalkozó programozók nemzetközi csoportja. A ROS környezet magas szintű nyelveken fejlesztett programokat támogat, például Python és C++ nyelv. Számos grafikus eszközzel is rendelkezik, amelyek lehetővé teszik az egyes járműalkatrészek és szoftvercsomagok működésének vezérlését. Emellett olyan eszközökkel is ellátták, amelyek lehetővé teszik a jármű működésének megjelenítését az érzékelő rendszerek adatai alapján, valamint a jármű működésének szimulálását. Az Rviz

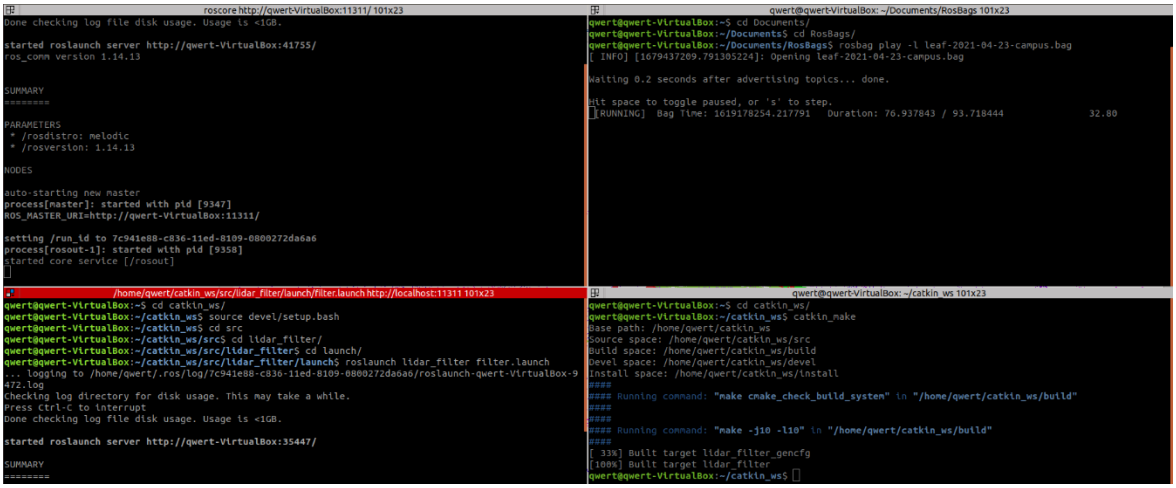
csomag például egy koreai egyetemen kifejlesztett ROS üzenetek háromdimenziós megjelenítésére használt csomag. Lehetővé teszi az érzékelőktől származó adatok megjelenítését és ábrázolja a robot környezetét. Ezen kívül lehetővé teszi az adatok megjelenítését egy kiválasztott koordinátarendszer szemszögéből. A grafikus eszköz felülete lehetővé tette az önjáró autót körülvevő környezet digitális térképének elkészítését szenzoradatok alapján.

A ROS alkalmazások közzétételi – előfizetési (publisher – subscriber) architektúrán keresztül kommunikálnak, ahol a cél az, hogy a nem szakértők gyorsan olyan szoftvert készítsenek, amely olyan kifinomult funkciókat tartalmaz, mint például az útvonaltervezés, vagy az objektumfelismerés. A ROS alkalmazások csomópontok (node) hálózataiként vannak modellezve, ezek olyan folyamatok, amelyek bizonyos feladatokat látnak el, mint például a navigációs algoritmusok futtatása, a képek feldolgozása, az érzékelőadatok közzététele stb. Ezek a csomópontok témaköröknek (topic) nevezett csatornákon keresztül kommunikálnak. A közzététel – előfizetés architektúrát követve a csomópontok ezeken a csatornákon keresztül teszik közzé az üzeneteket (message) és a csomópontok előfizetnek azokra a csatornákra, amelyek rendelkeznek a szükséges adatokkal [9] [10] [11].

A projektfeladatom készítésének első lépése egy úgynevezett catkin_ws munkakörnyezet (workspace) létrehozása, majd annak inicializálása volt. Az mkdir paranccsal létrehoztam egy catkin_ws nevű könyvtárat, majd azon belül egy src könyvtárat is. A catkin_ws mappába belépve a catkin_make parancs hatására létrejönnek a szükséges könyvtárak (devel, build, logs), fájlok. Ez az első build, ami lefut a projekten. A forráskód, vagy a futtatáshoz szükséges egyéb fájlok módosításakor minden alkalommal újra kell buildelni a projektet, hogy a változások érvényesüljenek. A devel mappában létrejött egy setup.bash, amire hivatkoznunk kell, mielőtt a roslaunch parancsot használni szeretnénk. A catkin_ws mappába lépve a source devel/setup.bash parancs kiadásával ezt aktiválhatjuk, így el lehet majd indítani a későbbiekben létrehozott launch fájlt. Ezek után az src mappán belül egy úgynevezett ROS csomagot (package) hoztam létre a catkin_create_pkg lidar_filter roscpp paranccsal (a lidar_filter a létrehozott csomagnak a neve, a roscpp pedig egy függőséget jelez), ami tartalmazza majd a forráskódot, a launch fájlt, a különböző konfig fájlokat és egyéb fájlokat.

A projektfeladat készítése közben folyamatosan tesztelnem kellett a már elkészült programrészeket a ROS rendszerben. Ehhez szükséges volt néhány ROS parancs elsajátítására, amikről néhány mondatot írnék.

A roscore olyan csomópontok és programok gyűjteménye, amelyek egy ROS alapú rendszer előfeltételei. Ahhoz, hogy a ROS csomópontok kommunikálni tudjanak, mindenképp futnia kell egy roscore-nak. Ezt a roscore paranccsal lehet elindítani. A rosbag egy eszközkészlet, amit a ROS témákból (topic) történő rögzítéshez és lejátszáshoz lehet használni. A projektem folyamatos teszteléséhez én is egy ilyen rosbag-et használtam, ami a Győri Egyetem kampuszán lett rögzítve 2021-ben. Ebben a rosbag-ben néhány LIDAR alapú pontfelhő adatai és kamerafelvétel található meg. A rosbag play –l leaf-2021-04-23-campus.bag paranccsal tudtam elindítani a lejátszást. A parancsban lévő –l paraméterrel lehetőség van a felvételt folyamatosan újra játszani, úgymond loop-olni, így megkönnyítve az elemzéseket, teszteléseket. A roslaunch egy olyan eszköz, ami képes több ROS csomópont egyszerű elindítására helyileg, vagy távolról SSH-n keresztül, valamint lehetőség van a paraméterek beállítására is. A roslaunch egy, vagy több XML konfigurációs fájlt képes fogadni, ezeknek a fájloknak a kiterjesztése .launch. A konfigurációs fájlok meghatározzák a beállítandó paramétereket és az indítandó csomópontokat, valamint azokat a gépeket, amelyeken futni kell. A projektfeladatom launch könyvtárában található meg az általam létrehozott filter.launch nevű indító fájl. A program elindítását a roslaunch lidar_filter filter.launch paranccsal tehetjük meg. A parancsban a lidar_filter a csomag nevét, míg a filter.launch a fájl nevét határozza meg. A rostopic parancssori eszközzel ROS témákkal kapcsolatos információkat lehet megjeleníteni, beleértve a kiadókat, az előfizetőket, a közzétételi arányt és a ROS üzeneteket is. A projektfeladat során folyamatosan ellenőriztem a rostopic hz /topic_name paranccsal a már publikált topic-ok közzétételi arányát, mivel 20 Hz-es feldolgozási idő volt meghatározva követelményként.
A forráskód fejlesztésekor és tesztelésekor a terminálemulátor alkalmazás ablakát 4 részre osztottam fel, így átlátható és könnyen kezelhető volt minden futó folyamat. Az első részben a roscore-t, a második részben a rosbag-et, a harmadik részben a roslaunch-ot indítottam el. A negyedik részt pedig a folyamatos buildelésre használtam. A Terminator program 4 ablakos futását a 9. ábra szemlélteti.



9. ábra – A terminálemulátor 4 részre osztott ablaka

6.2 Pontfelhő (point cloud)

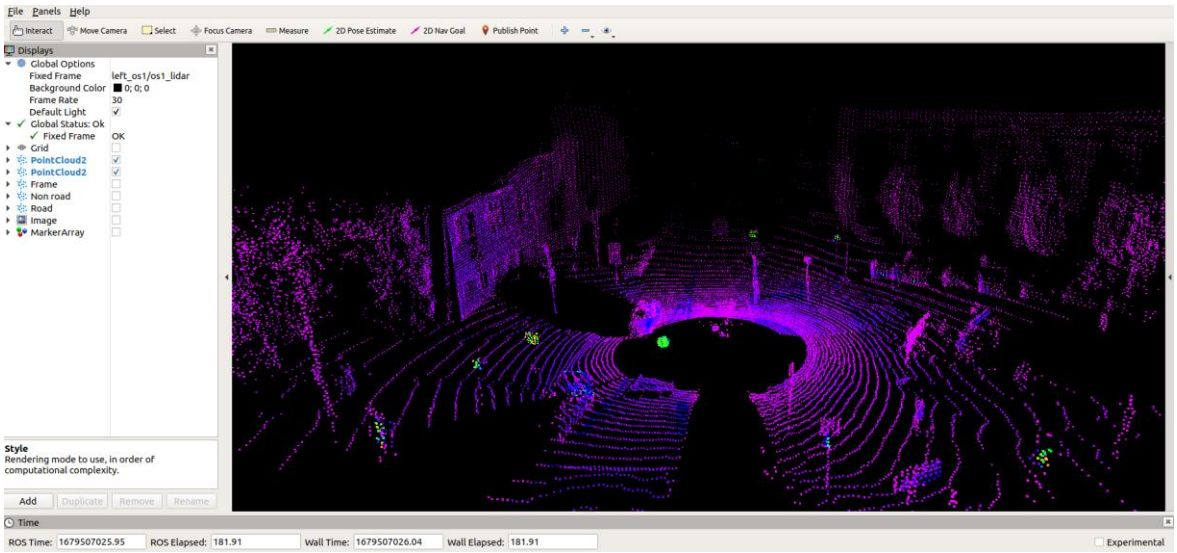
A 3D-s terek elemzése abból az igényből fakad, hogy megértsük a minket körülvevő környezetet és egyre pontosabb virtuális reprezentációkat készítsünk a térről. A 3D-s városi környezetmodell a földfelszín és a hozzá kapcsolódó objektumok, például épületek, fák, növényzet és néhány, a város területéhez tartozó ember alkotta elem digitális ábrázolása. A 3D-s városi környezeti modellekre különféle kifejezéseket használnak, például kiberváros, virtuális város, vagy digitális város. Mindegyik alapvetően egy számítógépes utcamodell, amely épületek és egyéb objektumok grafikus ábrázolását tartalmazza 3D-s térben.

A pontfelhő egy széles körben használt 3D-s adatforma, amely mélységérzékelőkkel, például fényérzékeléssel és távolságmérővel (LIDAR), vagy RGB-D kamerákkal állítható elő. Az objektumokról és környezetekről részletes információkat biztosító pontfelhőt széles körben használják különféle alkalmazásokban, mint például a digitális megőrzés, a földmérés, az építészet, a 3D-s játékok, a robotika, a virtuális valóság, várostervezés, bűnmegelőzés. A digitális megőrzési területen épületek és történelmi városok vizuálisan esztétikus és részletes 3D-s modelljei készülnek lézerszkenneléssel és digitális fotogrammetriával. A robotika területén pontfelhőket használnak a céltárgy azonosságának, elhelyezkedésének, valamint a robot mozgásának és manipulációjának akadályainak felismerésére.

Ami a keletkezett pontfelhők pontsűrűségét illeti, a lézereszköz mechanizmusa és az objektum visszaverő képessége befolyásolja azt. Egy tipikus LIDAR modell, mint például a Velodyne márkájú HDL-64E típusú, akár 2,2 millió pont/másodperc pontfelhőt is képes

generálni 120 méteres hatótávolságig. Hatékony és gyors módszerekre van szükség ahhoz, hogy kiszűrjük a jelentős adatokat ezekből az adatfolyamokból és nagy számítási teljesítményre van szükség ennek a nagy mennyiségű adatnak az utófeldolgozásához. A pontfelhő 3D strukturálatlan vektorokkal rendelkező pontokból áll. Minden pont kifejezhető egy vektorral, amely jelzi a 3D koordinátáját és néhány további jellemző csatornát, például a visszaverődés intenzitását, színét és normálértékeit. A pontfelhőnek három alapvető tulajdonsága van, beleértve a rendezetlenséget, a pontok közötti interakciót és a transzformációk invarianciáját [12] [13] [14].

A Győri Egyetem kampuszán rögzített rosbag csomagban 4 darab pontfelhő adatai találhatók meg. Ezeket a pontfelhőket az önjáró autóra szerelt távolságérzékelő szenzorok (LIDAR) állították elő. Ahhoz, hogy ezeket a pontfelhőket meg tudjuk nézni, szükség van egy ROS csomagra, aminek a neve Rviz. A rosrun rviz rviz paranccsal tudjuk ezt a vizualizációs eszközt elindítani. Ezzel a csomaggal képesek vagyunk a szenzor adatokat (vagyis a pontfelhőkben található pontokat) megjeleníteni egy háromdimenziós térben. Fontos az is, hogy a rögzített rosbag fájl lejátszását elindítsuk, hiszen abban vannak a szenzorok által készített adatok. A 10. ábra szemlélteti az Rviz grafikus felületét és a megjelenített pontokat a térben, jelen esetben itt kettő darab pontfelhő van ábrázolva együttesen. Ezeket az adatokat nevezhetjük a LIDAR nyers adatainak, hiszen itt még semmilyen szűrést nem végeztünk ezeken az adatokon, csupán csak megjelenítettük.

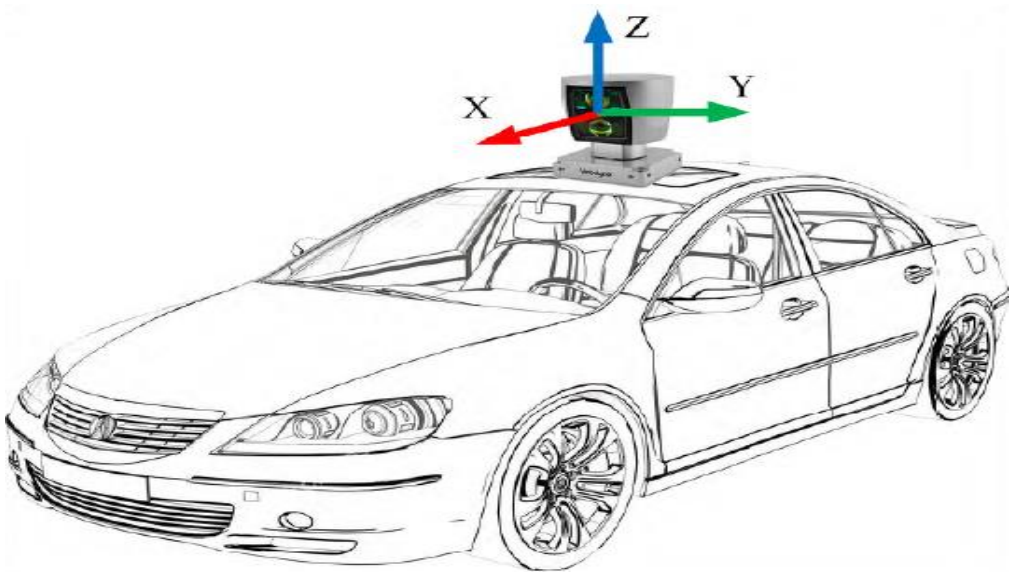


10. ábra – LIDAR adatok megjelenítése az Rviz vizualizációs eszközzel

Az Rviz grafikus felületen lehetőség van a kamera nézetet egyénileg beállítani (mozgatással, forgatással), így bármilyen irányból, pontból lehetséges elemezni pontfelhő halmazokat.

Akár több pontfelhőt is hozzáadhatunk az Add gomb segítségével az Rviz-ben, amiket utána konfigurálhatunk (topic kiválasztása, a megjelenítés beállítása, stb), majd el is menthetjük egy konfig fájlba a beállított értékeket. A launch fájlba megadható, hogy a lementett rviz konfigurációval induljon majd el a program.

A háromdimenziós térben egy pont helyzetét X, Y, Z koordináták segítségével lehet meghatározni. Az origót ebben az esetben a LIDAR helyzete adja meg. A pontfelhő egyes pontjainak értékeit (X, Y, Z) az origótól mért távolsága határozza meg. Az önjáró autón található LIDAR is egy háromdimenziós térben közlekedik, tehát X koordináta a kocsi haladási irányát (előremenet, hátramenetel), az Y koordináta a bal és jobb oldali irányt, és a Z koordináta pedig a magasságot határozza meg. A LIDAR koordináta rendszerét a 11. ábra szemlélteti.



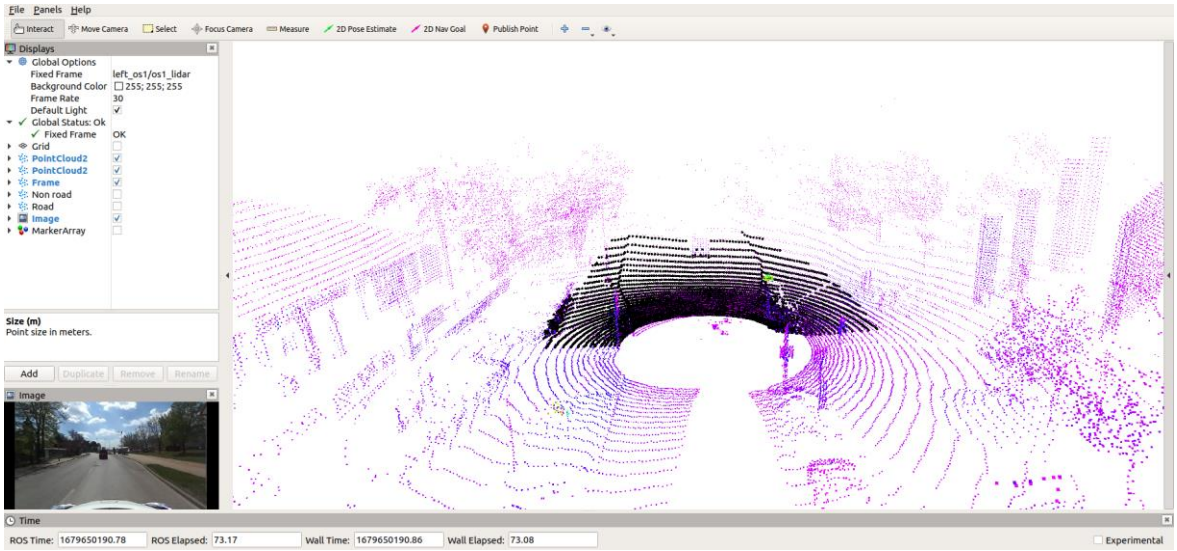
11. ábra – LIDAR koordináta rendszere [14]

6.3 Vizsgált terület meghatározása

A különböző szűrési eljárásokat az önjáró autóra felszerelt LIDAR szenzor adatain végezzük el, amiket jelen esetben a rosbag lejátszása szimulálja számunkra. Első lépésként feliratkoztam a left_os1/os1_cloud_node/points nevű topic-ra. Az önjáró autónak ez a bal oldalán elől elhelyezkedő Ouster márkájú LIDAR szenzorja, ami 64 csatornás. A feliratkozás után megvizsgáltam, hogy a LIDAR üzenetekben megközelítőleg hány darab pont van regisztrálva adott időben. Készítettem egy pontfelhő objektumot, ami az összes pont tárolásához szükséges, majd egy for ciklussal végig iteráltam a szenzor üzeneten, a pontokat hozzáadtam az előzőleg létrehozott pontfelhő objektumhoz és a végén egy

változóban eltároltam a pontok darabszámát. Adott időben több mint 32 ezer pontot regisztrált a LIDAR szenzorja. Amennyiben az összes ponton végezném el a szűrési vizsgálatokat, az valószínűleg rontana sokat a feldolgozási időn, illetve az önjáró autótól túlságosan távol lévő pontokat sincs értelme vizsgálni. A háromdimenziós térben ezért egy vizsgálandó területet kell meghatározni az autó körül. Ennek a vizsgált területnek a határait az origóhoz (LIDAR-hoz) viszonyítva adjuk meg. Hat darab változót használtam ehhez fel, ami a minimális és maximális értékeket tárolja X, Y, Z koordináták tekintetében. Egy if feltétellel meghatároztam, hogy csak azokat a pontokat adja hozzá a pontfelhő objektumhoz, ami a minimális és maximális értékeken belül helyezkednek el. A vizsgált terület meghatározásával a továbbiakban így kevesebb ponttal (kb. 4500-5500 pont) kell a különböző szűréseket elvégezni.

Miután sikerült redukálni a pontok számát, ezt a ponthalmazt hozzá rendeltem a filtered_frame nevű topic-hoz, létrehoztam egy pub_frame nevű publisher-t, aminek a segítségével közzétettem őket. A rostopic list nevű paranccsal ellenőriztem, hogy az előzőleg létrehozott topic valóban publikálva lett-e, így a listában szerepelt a filtered_frame nevű topic. Ezek után az Rviz eszközzel ellenőriztem vizuálisan is a már kész működő topic-ot. Az Add gomb segítségével létrehoztam egy PointCloud2-es nevű vizualizációs eszközt, amit Frame-nek neveztem el, majd ehhez hozzárendeltem az előzőleg közzétett filtered_nevű topic-ot. A közzétett pontok megjelenítését pontszerű stílusra, méretét 4-es pixel-re, a színét pedig feketére állítottam be.



12. ábra – Vizsgált területen elhelyezkedő pontok megjelenítése

6.4 Pontok tárolása és egyéb számítások

A pontok koordináta értékeinek (X, Y, Z) tárolását és az egyéb számításokhoz szükséges tárolásokat dinamikus tömbbel oldottam meg. A 6.3. részben, a vizsgált terület meghatározásakor egy változóban eltároltam a pontok darabszámát minden egyes vizsgálatkor. Ezt az értéket több helyen is felhasználtam a forráskód írásakor. Először egy kétdimenziós dinamikus tömböt hoztam létre, ami pontosan akkora területet foglalt le, amennyi a pontok darabszáma volt adott vizsgálatkor. A 13. ábra szemlélteti a kétdimenziós tömb szerkezetét.

	X	**Y**	**Z**	**D**	**α**
1. pont					
2. pont					
.					
.					
n. pont					

13. ábra – Kétdimenziós dinamikus tömb szerkezete

Az X, Y, Z oszlopok a pontok koordináta értékeit, a D oszlop az adott pont és az origótól vett távolságát, az alfa oszlop pedig a pont szögfelbontását jelentik. Az egyes sorok pedig az adott pontot határozzák meg. A vizsgált terület összes pontján végig iteráltam és közben feltöltöttem az értékekkel a kétdimenziós tömb X, Y, Z, D, α oszlopait. Az adott pont és az origótól vett távolságát (D) az alábbi matematikai képlettel számítottam ki:

$$D = \sqrt{(X_2 - X_1)^2 + (Y_2 - Y_1)^2 + (Z_2 - Z_1)^2}$$

Az adott pont szögfelbontását koszinusz és szinusz szögfüggvények segítségével számítottam ki. Abban az esetben, amikor a Z értéke kisebb nullánál, akkor koszinusz függvényt kell alkalmazni. Ha viszont a Z értéke nagyobb vagy egyenlő nullánál, akkor szinusz függvényt kell alkalmazni. Ilyenkor hozzá kell adni még 90 fokot az értékhez. Az ezzel kapcsolatos matematikai összefüggések az alábbiak:

$$ha\ Z < 0 \rightarrow \alpha = cos^{-1}\left(\frac{|Z|}{D}\right)$$

$$ha\ Z \geq 0 \rightarrow \alpha = sin^{-1}\left(\frac{Z}{D}\right) + 90°$$

A LIDAR szenzoroknak van egy meghatározott vertikális szögfelbontásuk, attól függően, hogy hány csatornás (16, 32, 64 stb.) az érzékelő. Az általam vizsgált LIDAR típus (Ouster

64 csatornás) vertikális szögfelbontását 0.35 fok és 2.8 fok közötti intervallumban lehet szabályozni. A szenzorok ezen tulajdonsága határozza meg, hogy egy adott távolságban hány darab körív leképzését tudja elkészíteni. A vertikális szögfelbontás tulajdonság segítségével a vizsgált pontokat körökre lehet csoportosítani. Egy tömbben eltároltam a különböző szögfelbontásokat, darabszáma megegyezik a LIDAR csatornaszámával, azaz a körívekkel. A tömbben tárolt értékeket növekvő sorrendbe rendeztem. Ezek után létrehoztam egy háromdimenziós dinamikus tömböt, ami a vizsgált pontokat körívekre csoportosította. A 14. ábra szemlélteti a háromdimenziós tömb szerkezetét.

1. körív							
1. pont	**X**	**Y**	**Z**	**D**	**α**	Y’	1,2
2. pont							
.							
.							
n. pont							

2. körív							
1. pont	**X**	**Y**	**Z**	**D**	**α**	Y’	1,2
2. pont							
.							
.							
n. pont							

3. körív							
1. pont	**X**	**Y**	**Z**	**D**	**α**	Y’	1,2
2. pont							
.							
.							
n. pont							

n. körív							
1. pont	**X**	**Y**	**Z**	**D**	**α**	Y’	1,2
2. pont							
.							
.							
n. pont							

14. ábra – Háromdimenziós dinamikus tömb szerkezete

Az egyes táblák az adott körívet, a bennük lévő sorok pedig az adott pontot határozzák meg. Itt is X, Y, Z oszlopok a pontok koordináta értékeit, a D oszlop az adott pont és az origótól vett távolságát, az alfa oszlop a pont forgásszögét jelentik. Az Y’ oszlop az X = 0 érték mellett az új Y koordinátákat (amit majd a nem út pontok szűrésénél fogunk használni), az 1,2 elnevezésű utolsó oszlop pedig az csoportszámokat határozzák meg. A csoportszámoknál az 1-es szám az út pontokat, a 2-es szám pedig a nem út pontokat fogja jelenteni.

A vizsgált terület összes pontján végig iteráltam és közben feltöltöttem az értékekkel a háromdimenziós tömb X, Y, Z, D, α oszlopait. Az X, Y, Z értékeket átmásoltam a kétdimenziós tömbből, mivel ezek az adatok nem változtak. Az adott pont és az origótól vett távolságát (D) ugyanazzal a matematikai képlettel számítottam ki, mint kétdimenziós tömbnél, annyi különbséggel, hogy itt csak az X és az Y értékkel számolunk távolságot, Z-t nullának tekintjük. Az adott pont helyzetét egy körben a forgásszöggel határoztam meg,

amit az alábbi matematikai képlettel számítottam ki, attól függően, hogy a kör melyik negyedében található a pont:

$$\alpha = sin^{-1}\left(\frac{|X|}{D}\right)$$

ha az első negyedben található: [0°; 90°], akkor α,
ha a második negyedben található:]90°; 180°], akkor 180° - α,
ha a harmadik negyedben található:]180°; 270°], akkor 180° + α,
ha a negyedik negyedben található:]270°; 360°[, akkor 360° - α

Létrehoztam egy max_distance nevű egydimenziós tömböt, amiben eltároltam az adott köríven legnagyobb távolságra lévő pont értékét az origótól. A háromdimenziós tömb feltöltésekor egy if feltétellel vizsgáltam és hozzáadtam az értéket a tömbhöz minden egyes körívnél. Ezeket az értékeket az út pontok szűrésénél fogom majd felhasználni. A dinamikus tömbök használatakor fontos teendő még, hogy amikor már nem használjuk őket, tehát nincs szükség a bennük tárolt adatokra, akkor ezeket a területeket fel kell szabadítani, különben memóriaszivárgás keletkezik és hibával leáll a program futása. A programunkban például mikor a kétdimenziós tömbből az összes adatot átemeljük a háromdimenziós tömbbe és már további számításokat nem végzünk velük, akkor a delete kulcsszóval felszabadítjuk a kétdimenziós dinamikus tömb területét. A projektfeladat során többféle adattárolókat (változók, tömbök) használtam fel, amik közül felsoroltam a fontosabbakat az alábbi táblázatban (15. ábra) jellemzőjükkel együtt [17].

Azonosító	**Funkció**	**Típus**
arr_2d	Pontokat és egyéb számításokat tároló tömb.	Háromdimenziós dinamikus valós tömb
arr_3d	Pontokat és egyéb számításokat tároló tömb.	Kétdimenziós dinamikus valós tömb
channels	Az alkalmazott LIDAR csatornaszáma.	Egész
piece	A vizsgált területen meghatározott pontok darabszáma.	Egész
part_result	A szögfüggvényeknél a részeredmények tárolásához szükséges változó.	Valós
angle	Egy tömb, amiben eltároljuk a különböző szögfelbontásokat. Ez megegyezik a LIDAR csatornaszámával.	Egydimenziós valós tömb
new_circle	Új körvonal feltétel vizsgálatkor használt változó.	Egész
internal	Az alkalmazott LIDAR vertikális szögfelbontásának intervalluma.	Valós

index_array	Az adott köríveket tartalmazó csoportok, megfelelő sorindexeinek beállításához szükséges tömb.	Egydimenziós egész tömb
max_distance	Egy tömb, ami tárolja az adott körív en a legnagyobb távolságra lévő pont értékét az origótól.	Egydimenziós valós tömb
curb_points	A becsült pontok száma a járdaszegélyen.	Egész
curb_height	A becsült minimális járdaszegély magasság.	Valós
angle_filter1	Három pont által bezárt szög, X = 0 érték mellett.	Valós
angle_filter2	Két vektor által bezárt szög, Z = 0 érték mellett.	Valós
beam_zone	A vizsgált sugárzóna mérete.	Valós
marker_array_points	Marker pontokat tárolja.	Kétdimenziós valós tömb
epsilon	Merőleges távolságméréshez szükséges küszöbérték.	Valós
max_distance_road	Adott fokban a legtávolabbi út pont (zöld pont) távolságát tárolja a változó.	Valós
simp_marker_array_points	Az egyszerűsített marker pontokat tárolja.	Kétdimenziós valós tömb

15. ábra – Fontosabb adattárolók és azok jellemzőik

6.5 Nem út pontok szűrése

A leggyakoribb úthatárolók városi környezetben a járdaszegélyek. Ezek hatékony észlelése alapvető és kulcsfontosságú az önvezető autók navigációjához. A legtöbb városi vezetési forgatókönyvben az út határát a szegélyek helyzete határozza meg mindkét oldalon. A járdaszegélyek általában az út mindkét oldalán vannak és az út mentén folyamatosak. A kereszteződésekben azonban szegmentálva vannak, ami bonyolulttá teszi a járdafelderítést. A járdaszegély magassága általában 10–15 cm, a magasság pedig élesen változik. Emellett az útfelület pontjai simák és összefüggőek, a járdaszegély pedig általában az út szélein jelenik meg. Mivel a járdaszegélyek lényeges jellemzői a vezethető útfelületnek és a korlátozott területek megkülönböztetésének, jelentősek az önvezetés biztonsága szempontjából. Geometriai alakzatként legjobban vonalszakaszok segítségével írhatók le [15] [16].

Ahhoz, hogy minél jobb eredményt kapjunk a magaspontok megtalálásához nem elég a pontok vertikális irányú vizsgálata. A horizontális irányú elemzéssel a pontok egymáshoz viszonyított helyzetét lehet vizsgálni. A nem út pontok keresésére kétféle szűrési módszert alkalmaztam. Az első eljárásban a pontok X értékét nullának tekintettem, tehát nem vettem

figyelme az X koordinátákat. Ezen kívül a háromdimenziós tömb 6. oszlopát egy for ciklussal feltöltöttem új Y értékekkel úgy, hogy a szomszédos pontok távolsága 1cm-es legyen. Egy for ciklussal végig iteráltam az összes körön, azon belül pedig egy újabb for ciklussal az adott körvonal pontjait vizsgáltam három pont által közbezárt szög alapján. A LIDAR forgása és a körív szakadások miatt kiszámoltam minden esetben a két szélső pont közötti távolságot (d), majd egy if feltételhez kötöttem, hogy ennek a távolságnak kisebbnek kell lenni, mint 5 méter. A két szélső pont közötti távolságot az alábbi matematikai képlettel számítottam ki:

$$d = \sqrt{(p3_x - p1_x)^2 + \left(p3_y - p1_y\right)^2}$$

Ezek után meghatároztam a három pont által bezárt háromszög oldalainak hosszát (x1, x2, x3) és kiszámítottam a három pont és a két vektor által bezárt szöget (α) az alábbi képletekkel:

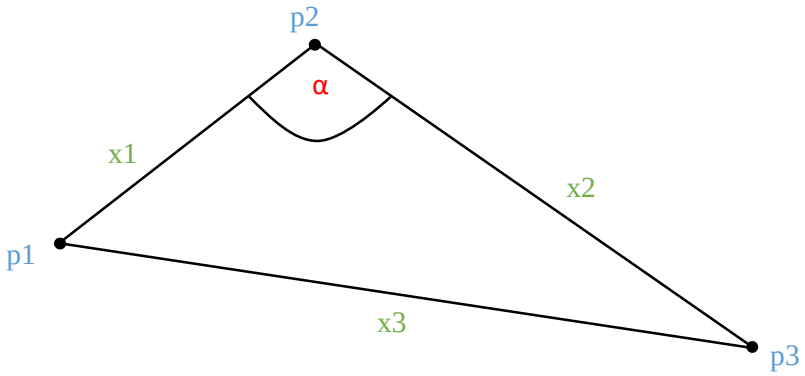
$$X1 = \sqrt{\left(p2_{y'} - p1_{y'}\right)^2 + (p2_z - p1_z)^2}$$

$$X2 = \sqrt{\left(p3_{y'} - p2_{y'}\right)^2 + (p3_z - p2_z)^2}$$

$$X3 = \sqrt{\left(p3_{y'} - p1_{y'}\right)^2 + (p3_z - p1_z)^2}$$

$$\alpha = cos^{-1}\left(\frac{x3^2 - x2^2 - x1^2}{(-2) * x2 * x1}\right)$$

Készítettem egy egyszerű ábrát, ami szemlélteti a három pont által bezárt háromszög oldalainak hosszát és a két vektor által bezárt szöget (16. ábra).



16. ábra – Háromszög oldalainak hossza és a két vektor által bezárt szög [17]

Miután kiszámításra került a két vektor által bezárt szög (α), egy if feltétellel vizsgáltam a pontokat. Három feltételnek kell teljesülnie ahhoz, hogy a közbenső pontnak a tömb 7.

oszlopába 2-es szám kerüljön, azaz az adott pont magaspontnak számítson. Az első feltétel az, hogy a két vektor által bezárt szög (α) kisebb vagy egyenlő legyen, mint az angle_filter1 nevű változóban tárolt szög értéke (alapesetben 150 fokra van beállítva). A második feltétel az, hogy a curb_height nevű változóban tárolt becsült minimális járdaszegély magassága (alapesetben 5cm-re van beállítva) kisebb vagy egyenlő legyen, mint az adott két pont Z értékének a különbsége (abszolút értékben). A harmadik feltétel pedig az, hogy a két szélső pont Z értékének a különbsége (abszolút értékben) nagyobb vagy egyenlő legyen, mint 0.05. A második szűrési módszerben a pontok Z értékét nullának tekintettem, tehát nem vettem figyelme a Z koordinátákat. Egy újabb for ciklussal ismét az adott körvonal pontjait vizsgáltam, ebben az esetben a két vektor által bezárt szög alapján. Emellett figyelembe vettem a vektor pontjainak magasság változását is. Először kiszámoltam a két vektor szélső pont közötti távolságot (d), majd hasonlóan az előző módszerhez, itt is egy if feltételhez kötöttem, hogy ennek a távolságnak kisebbnek kell lenni, mint 5 méter. Két for ciklust felhasználva, meghatároztam mind a két vektor (’a’ vektor és ’b’ vektor) pontjai közül a legnagyobb Z értékűt (azaz a legmagasabbat), majd ezt összehasonlítottam a vektorok kiindulási pontjának Z értékével. A vektorok meghatározását az alábbi matematikai képletekkel számítottam ki:

$$v_a = \frac{1}{n} * \left[\sum_{k=1}^{n} (x_{i-k} - x_i), \sum_{k=1}^{n} (y_{i-k} - y_i) \right]$$

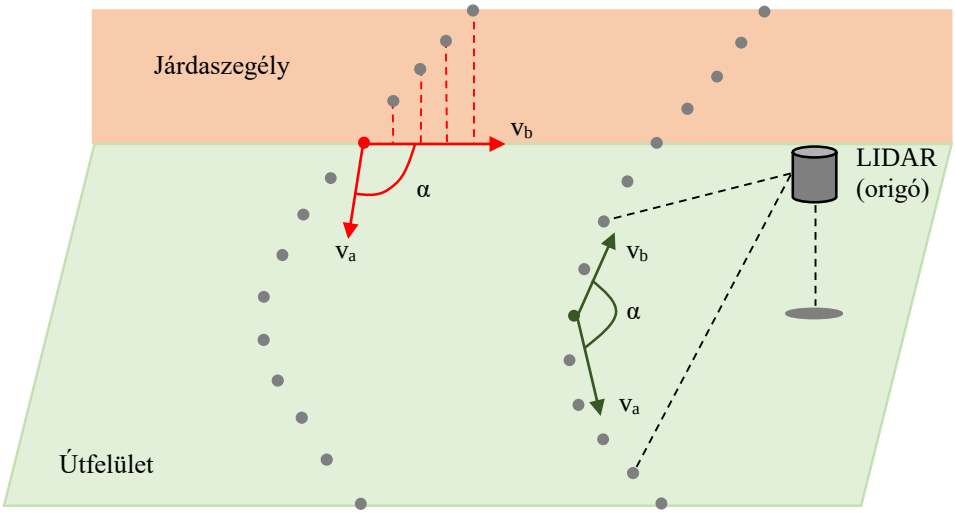
$$v_b = \frac{1}{n} * \left[\sum_{k=1}^{n} (x_{i+k} - x_i), \sum_{k=1}^{n} (y_{i+k} - y_i) \right]$$

A vektorok meghatározásánál az x és az y az egyes pontok koordinátáit, az n pedig a curb_points változó értékét (azaz a becsült pontok számát a járdaszegélyen) jelentik. Ezek után kiszámítottam a két vektor által bezárt szöget (α) az alábbi matematikai képlettel:

$$\alpha = cos^{-1}\left(\frac{v_a * v_b}{|v_a| * |v_b|}\right)$$

Miután kiszámításra került a két vektor által bezárt szög (α), ismét egy if feltétellel vizsgáltam a pontokat. Három feltételnek kell teljesülnie itt is ahhoz, hogy a két vektor közös pontjának a tömb 7. oszlopába 2-es szám kerüljön, azaz magaspontnak számítson. Az első feltétel az, hogy a két vektor által bezárt szög (α) kisebb vagy egyenlő legyen, mint az angle_filter2 nevű változóban tárolt szög értéke (alapesetben 140 fokra van beállítva). A második feltétel az, hogy a curb_height nevű változóban tárolt becsült minimális

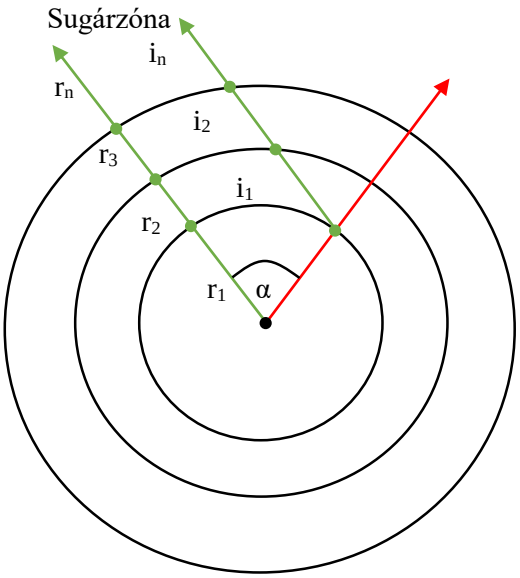
járdaszegély magassága (alapesetben 5cm-re van beállítva) kisebb vagy egyenlő legyen, mint a vektor legmagasabb Z értéke (max1 ,max2) és az adott pont Z értékének (abszolút értékben) a különbsége. A harmadik feltétel pedig az, hogy a két vektor legmagasabb Z értékének a különbsége (abszolút értékben) nagyobb vagy egyenlő legyen, mint 0.05. Készítettem itt is egy egyszerű ábrát, ami szemlélteti a második szűrési módszert, amely a két vektor által bezárt szöget és a magasság változásokat vizsgálja (17. ábra) [17].



17. ábra – Két vektor által bezárt szög és magasság változások vizsgálata [16]

6.6 Út pontok szűrése

Az út pontok meghatározásához sugárzóna elemzést alkalmaztam. Ennek az a lényege, hogy minden egyes körnél egyforma körívhosszt vizsgál az algoritmus. Tehát egy intervallumot határoz meg minden esetben. Ennek a körívhosznak az elejét az adott pont szöge, a végét pedig az adott pont szöge és a beam_zone nevű változóban tárolt érték (alapesetben ez 30 fokra van beállítva) összege fogja megadni. Az első körre természetesen az előző meghatározás nem vonatkozik, hiszen ott a körívhossz szögét magát a pont szöge jelenti. A 18. ábrán látható, hogy a második körívtől kezdve a körívhossz egyre nagyobb lesz egy adott fokban.



18. ábra – Sugárzóna vizsgálat

Az előző fejezetben meghatározásra kerültek a nem út pontok minden egyes körön, ezeket a magaspontokat használtam fel az itteni vizsgálat során is. Minden vizsgált pontnál előzőleg kiszámításra került a forgásszög, amit a háromdimenziós tömb 5. oszlopában tároltam el. A tömb elemeit gyorsrendezéssel körönként a szögek szerint növekvő sorrendbe rendeztem először. Ehhez gyorsrendező segédfüggvényeket alkalmaztam. Három függvényből áll ez a gyorsrendező: az első függvény egy úgynevezett elemfelcserélő (swap), a második függvény a partition nevű, ami a rendezésekért felel, a harmadik függvény pedig a rekurzív hívásokat végzi. A háromdimenziós tömb feltöltésekor el lett tárolva a max_distance nevű tömbben az origótól legnagyobb távolságra lévő pont értéke is adott köríven. Tehát minden egyes köríven a LIDAR-hoz képest a legtávolabb lévő pont értéke van eltárolva a tömbben, ami minden körön a legnagyobb sugárértéket fogja jelenteni. A körív méretét az alábbi matematikai képlettel számítottam ki:

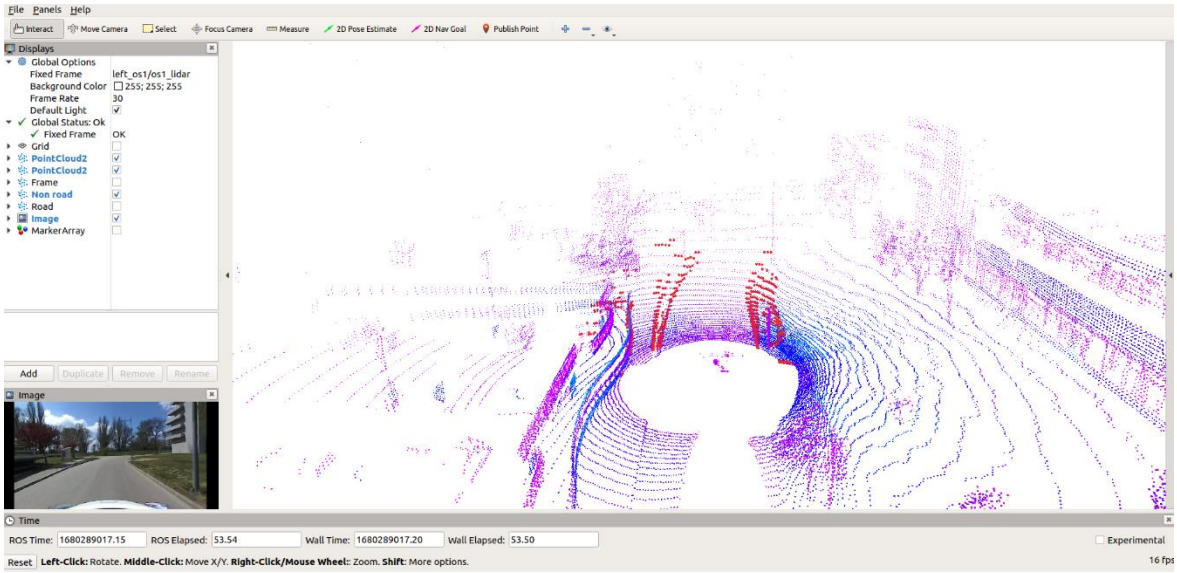
$$i = \left(\frac{r * \pi}{180}\right) * \alpha$$

Ezek után egy for ciklussal 0 foktól egészen 360 fok – beam_zone-ig (vizsgált sugárzóna mérete) elemezzük a köröket. Megvizsgáljuk először az első kör adott szakaszát. Ha találunk ebben a szakaszban olyan pontot, ami magaspontnak tekinthető (tömb 7. oszlopában 2-es szám szerepel), akkor nem vizsgáljuk tovább az adott szakaszt és egy break utasítással kilépünk a ciklusból. Amennyiben nem találtunk az első kör adott szakaszán magaspontot, folytatjuk a vizsgálatot a következő körrel. Ezen felül az első kör adott szakaszában található

pontokat út pontnak tekintjük (azaz az adott pontnak a tömb 7. oszlopába 1-es szám kerül). A további körök vizsgálatánál új szöget kell meghatározni, hogy a távolabbi körvonalakon is ugyanakkora körív hosszt vizsgáljunk. Az aktuális köríven a szög nagyságát a current_degree változóban tároljuk el minden esetben. Egy if feltétel ellenőrzi azt, hogy ha a sugár egy magasponton elakad, akkor a további köröket már ne is vizsgálja az algoritmus, hanem break utasítással lépjen ki a ciklusból. A további köröknél is ellenőrizzük, hogy az adott szakaszban található-e magaspont és ha nincs benne, akkor ezeket is elfogadottnak tekintjük, azaz az adott szakasz pontjai út pontnak tekinthető [17].

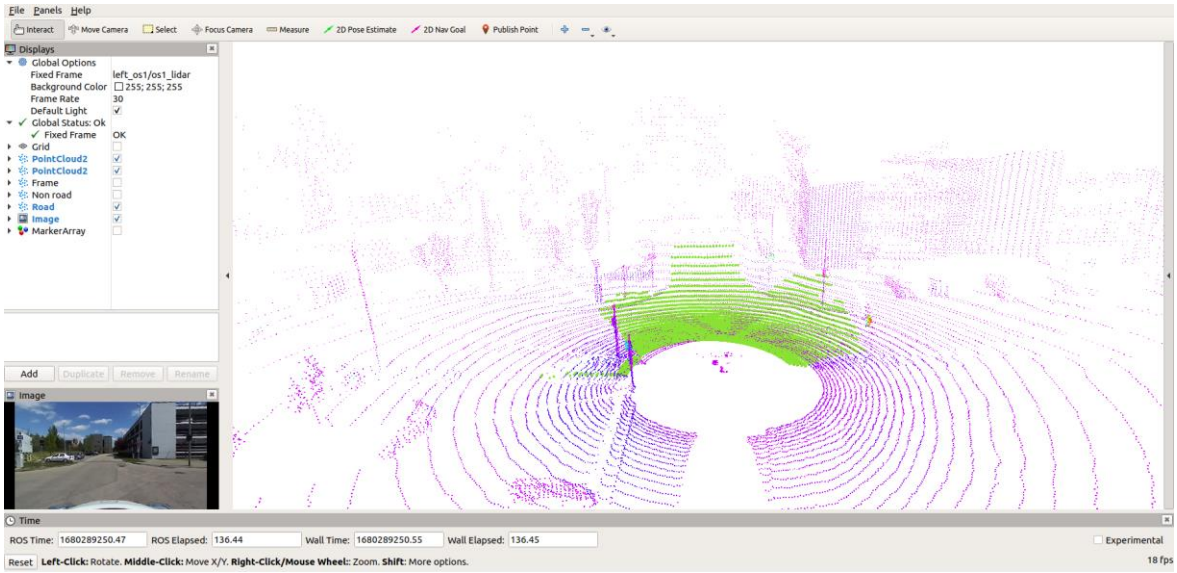
6.7 Topic-ok feltöltése és közzététele

Az előző fejezetekben a vizsgált területen elhelyezkedő pontokból meghatároztam a nem út pontokat és az út pontokat. Mind a két pont típusnál a háromdimenziós tömb 7. oszlopába egy érték került be, amivel azonosítani lehet, hogy egy pont melyik kategóriába tartozik (út pont, vagy nem út pont). Ha az értéke 1-es, akkor az út pontot, amennyiben az értéke 2-es, akkor pedig nem út pontot jelent. A topic-ok feltöltéséhez létrehoztam kettő darab pontfelhő objektumot filtered_non_road és filtered_road néven. Kettő darab for ciklust alkalmaztam a feltöltéshez, az egyikkel végig iteráltam az összes körön, a másikkal pedig a köríven található összes ponton. If feltételekkel vizsgáltam minden egyes pontnak a tömb 7. oszlopának értékét. Amelyik pontnál ez az érték 1-es volt, azt hozzáadtam a filtered_road pontfelhő objektumhoz, ahol pedig 2-es érték volt, azt hozzáadtam a filtered_non_road pontfelhő objektumhoz. A közzétételhez létrehoztam kettő darab publisher-t pub_non_road és pub_road néven, amelyek segítségével közzétettem a két topic-ot. Ezek után az Rviz eszközzel ellenőriztem vizuálisan is a már kész működő topic-okat. Az Add gomb segítségével (ahogy azt tettem a vizsgált terület meghatározásánál is) létrehoztam kettő darab PointCloud2-es nevű vizualizációs eszközt, amelyeket elneveztem Non road-nak és Road-nak. Ezek után hozzárendeltem az előzőleg közétett topic-okat hozzájuk.
Készítettem néhány képet, ahol látszódnak a szűrt pontok eredményei. A 19. ábrán látható a vizsgálandó területen elhelyezkedő nem út pontok ábrázolása a térben. Piros színnel vannak jelölve a nem út pontok.



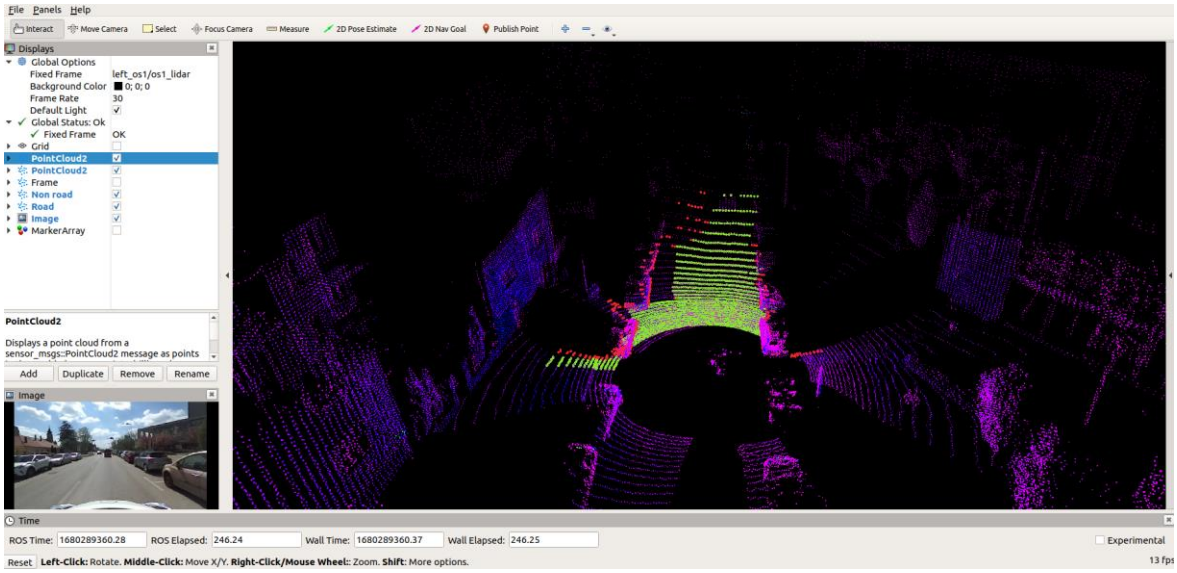
19. ábra – Vizsgált területen elhelyezkedő nem út pontok megjelenítése

A 20. ábrán látható a vizsgálandó területen elhelyezkedő út pontok ábrázolása a térben. Zöld színnel vannak jelölve az út pontok.



20. ábra – Vizsgált területen elhelyezkedő út pontok megjelenítése

A 21. ábrán látható a vizsgálandó területen elhelyezkedő nem út pontok és út pontok ábrázolása a térben.



21. ábra – Vizsgált területen elhelyezkedő nem út és út pontok megjelenítése

6.8 Marker pontok keresése

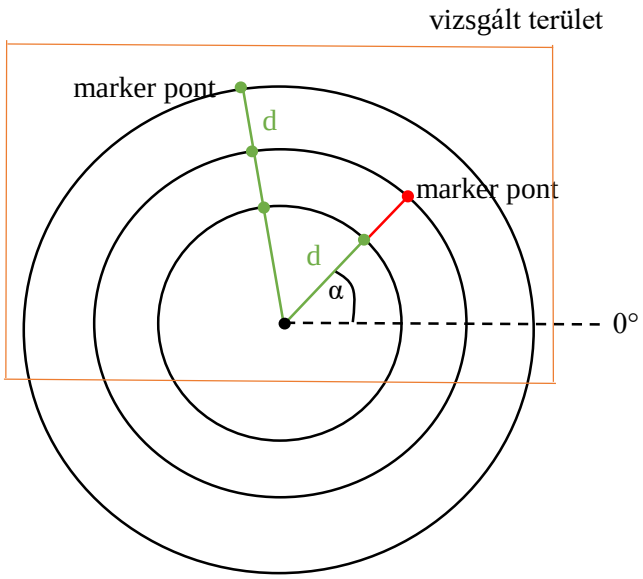
A marker pontok keresésekor azokat a pontokat kell meghatározni, ami adott fokban a legtávolabbi út pont az origóhoz, azaz a LIDAR-hoz képest. A marker pontok tárolásához egy kétdimenziós tömböt hoztam létre. A tömb első három oszlopa tartalmazza az adott pont X, Y, Z koordinátáját. A negyedik oszlop 0-ás vagy 1–es értéket vehet fel. Amennyiben a vizsgált fokban találunk nem út pontot, akkor 1-es értéket kap, ha pedig nem találunk, akkor 0-ás értéket kap. Három for ciklust használtam a marker pontok kereséséhez. Az első for ciklussal megvizsgáltam a pontokat fokonként, 0 foktól egészen 360 foking. Ezen belül két for ciklust implementáltam, az egyik a körvonalak iterációját, a másik pedig az adott körvonal összes pontjai iterációját végezte. Létrehoztam egy red_points nevű egész változót és egy max_distance_road nevű valós változót, amiben az adott fokban a legtávolabbi út pontnak (zöld pont) a távolságát tároltam. A változók minden egyes fok vizsgálatakor nulla kezdő értéket kaptak.

Amennyiben az adott fokban találunk nem út pontot (piros pont), akkor egy break utasítással kilépünk a körvonal elemzéséből, mert utána már biztosan nem lesz út pont (zöld pont). Ezen felül a red_points nevű változóba 1-es értéket tárolunk el. Az algoritmus azt is figyeli egy if feltétellel, hogy ha már talált adott fokban egy nem út pontot, akkor a további köríveket már nem fogja vizsgálni feleslegesen. Abban az esetben, ha adott fokban találunk út pontot (azaz zöld pontot), akkor egy távolságmérést végzünk, azaz megvizsgáljuk az adott

pontnak az origótól vett távolságát. Az alábbi matematikai képletet alkalmaztam a vizsgálathoz:

$$d = \sqrt{(0 - i_x)^2 + \left(0 - i_y\right)^2}$$

Ahol a két nulla érték az origó x és y koordinátája, i_x és i_y pedig az adott pont x és y koordinátája. Ha az adott pontnak az origótól vett távolsága nagyobb, mint az eddig a max_distance_road változóban tárolt érték, akkor új értékként ez kerül eltárolásra. Kettő darab segédváltozóban eltároltam az adott pont azonosítóit is, amikkel pontosan meghatároztam, hogy melyik körvonal és annak hányadik pontjáról van szó. Ezeket a változókat a tömb feltöltésekor használtam fel. Amikor megtaláltam a legtávolabb lévő út pontot adott fokban, akkor a tömbhöz hozzáadtam a pont koordinátáit és a red_ponints változó értékét. Egy c nevű változóval folyamatosan számoltattam a marker pontok darabszámát, amit majd a markerek beállításánál fogok felhasználni. Készítettem egy egyszerű ábrát, ami szemlélteti a marker pontok keresését (22. ábra) [17].



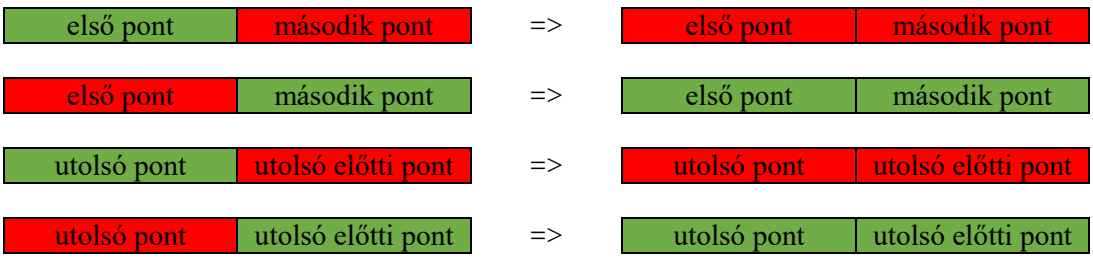
22. ábra – Marker pontok keresése

6.9 Marker összeállítása

Az előző fejezetben megkerestük a marker pontokat és egy kétdimenziós tömbbe ezen pontok X, Y, Z koordinátáit eltároltuk, valamint a tömb 4. oszlopába minden ponthoz hozzárendeltünk 0-ás, vagy 1-es értéket. 1-es értéket akkor kapott, ha az adott fokban (0

foktól 360 fokig vizsgáltuk) találtunk nem út pontot (magaspontot), 0-ás értéket pedig akkor, ha nem találtunk magaspontot. Ezeket a pontokat használtam fel a marker összeállításához. A kétdimenziós tömbben a marker pontok úgy lettek eltárolva egymás után, hogy a pontokat vonallal összekötve egy félpoligont kapunk. Ezzel a félpoligonnal határoztam meg a járható útfelület határvonalát.

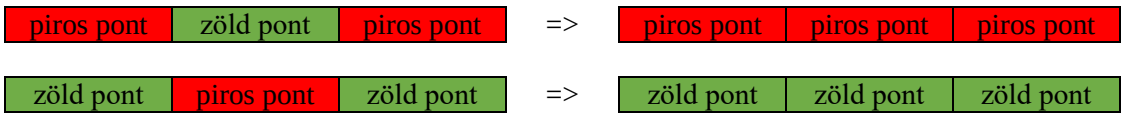
A marker összeállításához három darab objektumot hoztam létre: egy visualization_msgs::MarkerArray (neve: marker_arr) típusút, egy visualization::Marker (neve: line_segment) típusút és egy geometry_msgs::Point (neve: point) típusút. A marker objektumban tároltam az adott zöld, vagy piros szakaszt (line segment). A point objektumban az adott pont értékét tároltam, amivel feltöltöttem az adott line segment-et. A marker array objektumban a zöld és piros line segment—eket. Kezdésként a marker első és az utolsó pontjainak beállításával foglalkoztam. Ennek célja a zöld-piros-zöld és a piros-zöld-piros esetek kizárása volt az első és utolsó pontok tekintetében. 4 darab if feltétellel vizsgáltam ezeket az eseteket. Amikor az első pont zöld és a második pont piros, abban az esetben az első piros line_segment-be kerül, tehát az első pont is piros színű lesz. Amikor az utolsó pont zöld és az utolsó előtti pont piros, abban az esetben az utolsó piros line_segment-be kerül, tehát az utolsó pont is piros színű lesz. Ez volt a zöld-piros-zöld eset bemutatása, a piros-zöld-piros eset ennek a fordítottja. A 23. ábra szemlélteti az első és az utolsó marker pontok beállítását.



23. ábra – Első és utolsó marker pontok beállítása

Ezek után a marker további pontjait is meg kellett vizsgálni, ahol szintén egy zöld pontot közrefog kettő piros pont, vagy ennek a fordítottja, ahol egy piros pontot közrefog kettő zöld pont. A két eset beállításához kettő darab for ciklust használtam. Mind a kettő ciklusnál figyelembe vettem, hogy előzőleg már az első kettő pontot és az utolsó kettő pontot beállítottam, tehát ezeket nem kellett vizsgálni. Ahol egy zöld pontot közrefog kettő piros pont, ott a zöld pontot is pirosra állítottam be, így a piros line segment-hez kerül. Ahol pedig

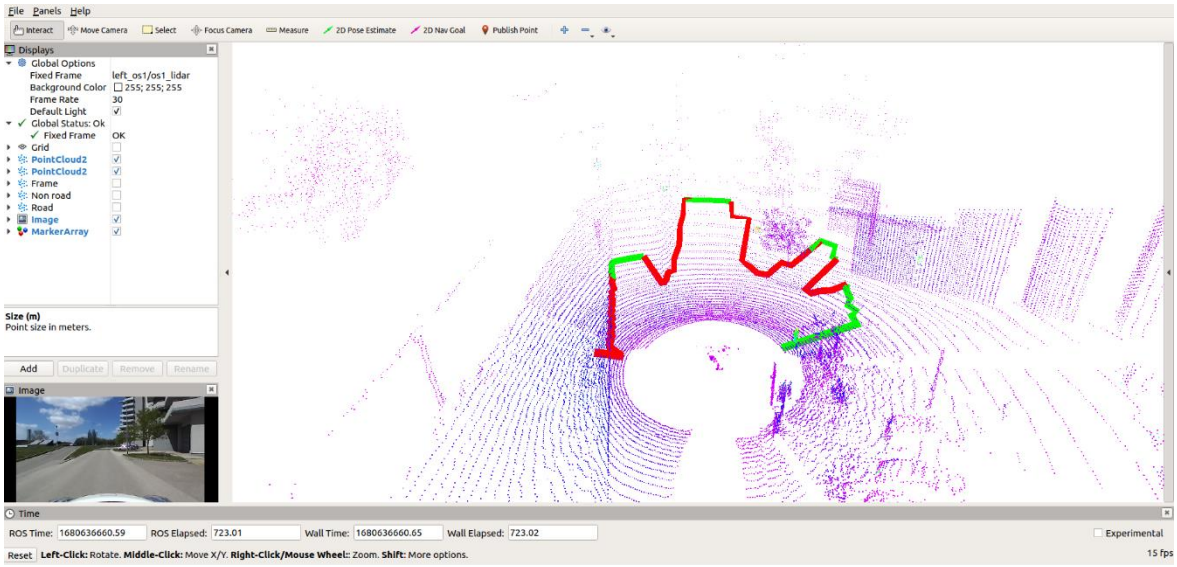
egy piros pontot közrefog kettő zöld pont, ott a piros pontot is zöldre állítottam be, így a zöld line segment-hez kerül. A 24. ábra szemlélteti a további marker pontok beállítását.



24. ábra – További marker pontok beállítása

A beállítások után egy for ciklussal végig mentem az összes marker ponton (azaz a tömb elemein) és különböző feltételekkel szabályoztam a csoportváltozásokat. Az első pontot hozzáadtam az adott line_segment-hez, mivel itt semmilyen feltételt nem kellett megadni. Amennyiben a következő pont is ugyanabba a csoportba tartozik, mint az előző, akkor ezt a pontot is hozzá lehet adni az adott line_segment-hez. Ezt egy else if feltétellel vizsgáltam. Ugyancsak ennél a feltételnél vizsgáltam az utolsó pontot is, itt készült el az utolsó line_segment. Különböző beállításokat kellett elvégezni az adott line_segment-en (pozíció, irány, vonal színe), majd utána a line_segment-et hozzá lehetett adni a marker array-hez. A pontokat a line_segment-ből kitöröltem, hogy feleslegesen ne tároljuk. 2 darab else if feltétellel vizsgáltam a csoportváltozásokat. Az egyik eset az, amikor pirosról zöldre fog váltani. Ebben az esetben még a kettő pontot piros marker fogja összekötni, így hozzá lehet adni a pontot az adott line_segment-hez. A másik eset pedig az, amikor zöldről pirosra fog váltani. Ebben az esetben először beállítjuk a zöld line_segment-et, majd hozzáadjuk az utolsó pontot a piroshoz. is. Zöld és piros pont között mindig piros line_segment van. Természetesen ezeknél a csoportváltozásoknál is figyelni kell a szín beállítására és hozzá kell adni a line_segment-et a marker array-hez. Ezek után beállítottam a line_segment élettaramának idejét és marker array-t publikáltam.

Létrehoztam egy pub_marker_array nevű publisher-t, aminek a segítségével közzétettem marker array-t. A rostopic list nevű paranccsal ellenőriztem, hogy az előzőleg létrehozott topic valóban publikálva lett-e, így a listában szerepelt a marker_array nevű topic. Ezek után az Rviz eszközzel ellenőriztem vizuálisan is a már kész működő topic-ot. Az Add gomb segítségével létrehoztam egy MarkerArray-es nevű vizualizációs eszközt, amit MarkerArray-nek neveztem el, majd ehhez hozzárendeltem az előzőleg közzétett marker_array topic-ot. A 25. ábrán látható az eredmény, azaz a vizsgálandó területen elhelyezkedő marker array ábrázolása a térben [17].



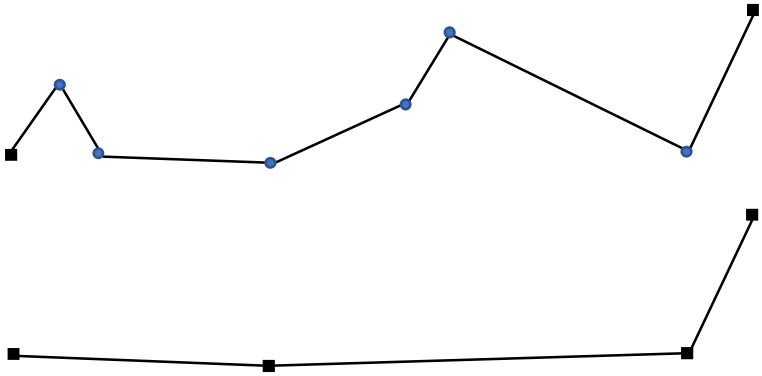
25. ábra – Vizsgált területen elhelyezkedő marker array megjelenítése

6.10 Vonallánc egyszerűsítő algoritmusok vizsgálata

Az előző fejezetben sikerült a marker pontokból összeállítanom egy marker array-t, amivel meghatároztam a járható útfelület határvonalát. A diplomamunka követelményben az is meg volt határozva, hogy ezek a félpoligonok lehetőleg egyszerűsítve, kevés marker pontból álljanak. Első lépésként megvizsgáltam a marker pontok darabszámát. A vizsgálat során kiderült, hogy általában ~145 és ~175 közötti marker pont volt a kétdimenziós tömbben. Ezek után megismerkedtem a vonallánc egyszerűsítésének a folyamatával, használatával. A vonallánc egyszerűsítésnek az a célja, hogy pontokat és éleket távolítunk el az eredeti vonalláncból, így egy egyszerűbb modellt állítunk elő belőle. Mindezt úgy tesszük meg, hogy az eredeti vonallánc fontos vizuális jellemzőit megtartjuk. A legtöbb vonalegyszerűsítési algoritmus megköveteli a felhasználótól, hogy adjon meg egy tűrésértéket (tűréshatárt), amelyet annak meghatározására használnak fel, hogy milyen mértékben kell az egyszerűsítést alkalmazni. A vonallánc egyszerűsítő algoritmusok közül én hármat vizsgáltam meg a projektfeladatom során: az N-edik pont módszert, a merőleges távolságmérés módszert és a Lang módszert.

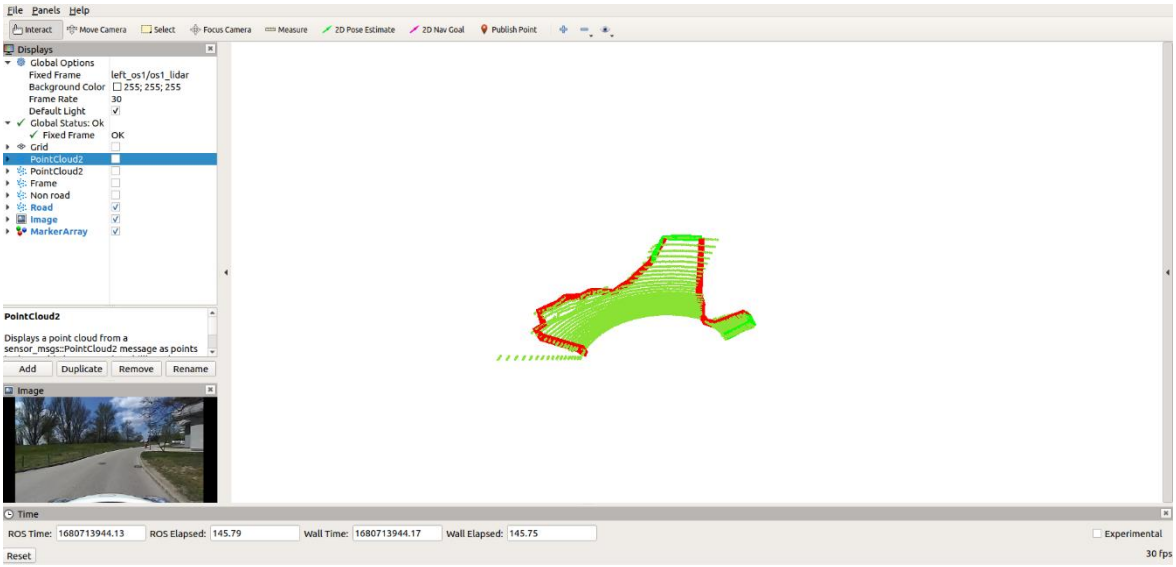
Az N-edik pont módszer a független pontalgoritmusokba tartozik. Ezek az algoritmusok meglehetősen egyszerűek és nem veszik figyelembe a szomszédos koordináló pontok matematikai kapcsolatát. Az N-edik pont algoritmus alapvetően csak az első, az utolsó és minden N-edik pontot tartja meg az eredeti vonalláncon, ahol N egy természetes szám. Az összes többi pont eltávolításra kerül. Ez az algoritmus rendkívül gyors és hatékony, azonban

a pontosság szempontjából nem túl jó, itt gondolok a vonal geometriai jellemzőinek (azaz a görbületi információkra) megőrzésére. Az N-edik algoritmus vonallánc egyszerűsítését a 26. ábra szemlélteti, ahol n = 3, azaz az első, utolsó és minden harmadik pontot tartja meg.



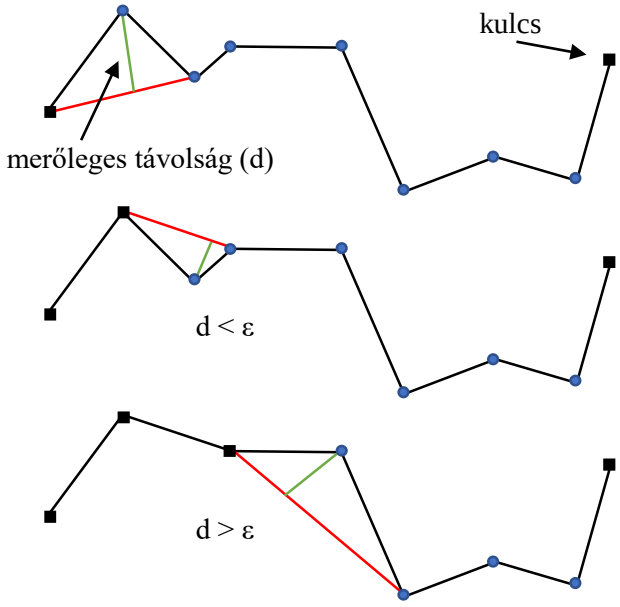
26. ábra – N-edik algoritmus vonallánc egyszerűsítése [19]

Miután megértettem az algoritmus működését, megírtam a kódot a marker pontok egyszerűsítésére és megvizsgáltam a pontosságát. Azt tapasztaltam, hogy ha n = 4, azaz minden negyedik pontot tartottam meg az eredeti vonalláncból, akkor a járható útfelület meghatározásában a határvonalnál viszonylag nagy torzulás jelentkezett. A határvonaltól kívülre esnek a zöld pontok, amik járható útfelületek lennének. Az algoritmus tehát nem veszi figyelembe a nagyobb kiugrásokat, így pontatlan lesz a határvonal meghatározása. Készítettem egy képet (27. ábra), ami szemlélteti az egyszerűsítéskor megjelenő torzítást.



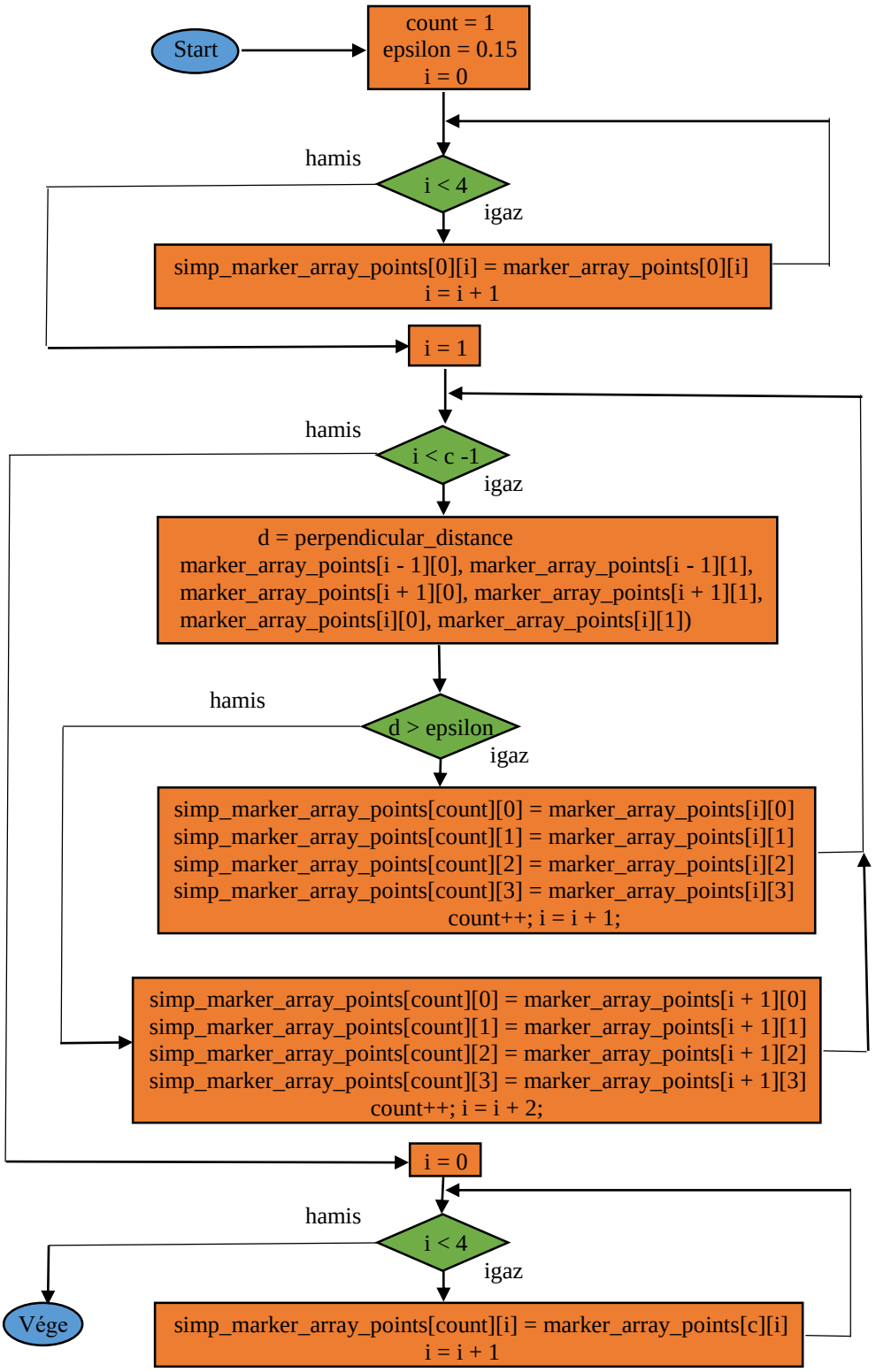
27. ábra – N-edik algoritmusnál megjelenő pontatlanság

A merőleges távolságmérés módszere a lokális feldolgozási algoritmusokba tartozik. A merőleges távolság esetén a vizsgált, vagy más néven kritikus pont előtti és utáni csomópontokat köti össze egy képzeletbeli egyenenessel, majd a kritikus pont egyenestől való távolságát vizsgálja meg az algoritmus. Ezek a távolságok nem lehetnek kisebbek az egyéni tűrésértéktől, amit a felhasználó határoz meg (epsilon). A tűrésértéken belüli pontok megszűnnek, míg a tűrésértéket meghaladó pontok megmaradnak. Tehát kezdetben az első három pontot vizsgáljuk, kiszámítjuk a második csúcs merőleges távolságát. Amikor ez a távolság a tűrésértéket meghaladja, akkor a második csúcsot megtartjuk, úgymond kulcsnak tekintjük. Az algoritmus ezután egy csúccsal tovább lép a vonalláncon és megkezdi a következő három csúcsból álló ponthalmaz feldolgozását. Amikor számított távolság a tűrésérték alá esik, akkor a közbülső csúcs megszűnik. Az algoritmus a vonallánc két csúcsának felfelé mozgatásával folytatódik. Ezzel a módszerrel az eredeti vonallánc legfeljebb 50%-kal csökkenthető. Amennyiben magasabb csúcscsökkentési arányt szeretnénk elérni, akkor több lépésre van szükség. A merőleges távolság algoritmus vonallánc egyszerűsítését a 28. ábra szemlélteti.



28. ábra – Merőleges távolság algoritmus vonallánc egyszerűsítése [19]

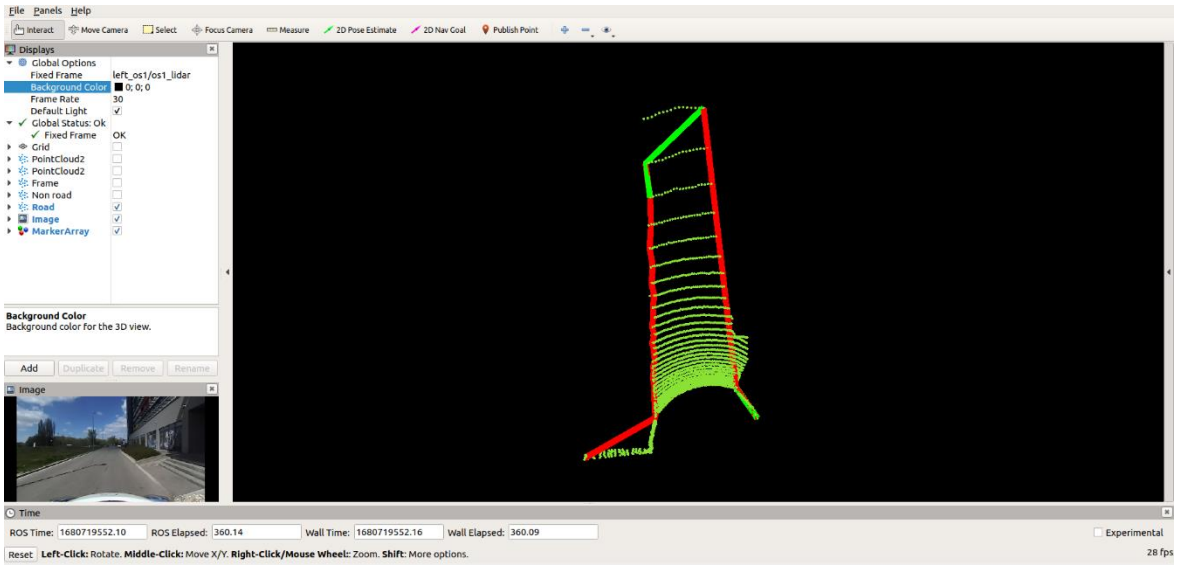
A vonallánc egyszerűsítő algoritmusok közül elkészítettem a merőleges távolság módszerének folyamatábráját is. Ennek az algoritmusnak a folyamatát szemlélteti a 29. ábra.



29. ábra – Merőleges távolság algoritmus folyamatábrája

Ezek után a merőleges távolság algoritmust is leprogramoztam és megvizsgáltam ennél is a marker pontok egyszerűsítését. Amikor a felére csökkentettem az eredeti vonallánc pontjait

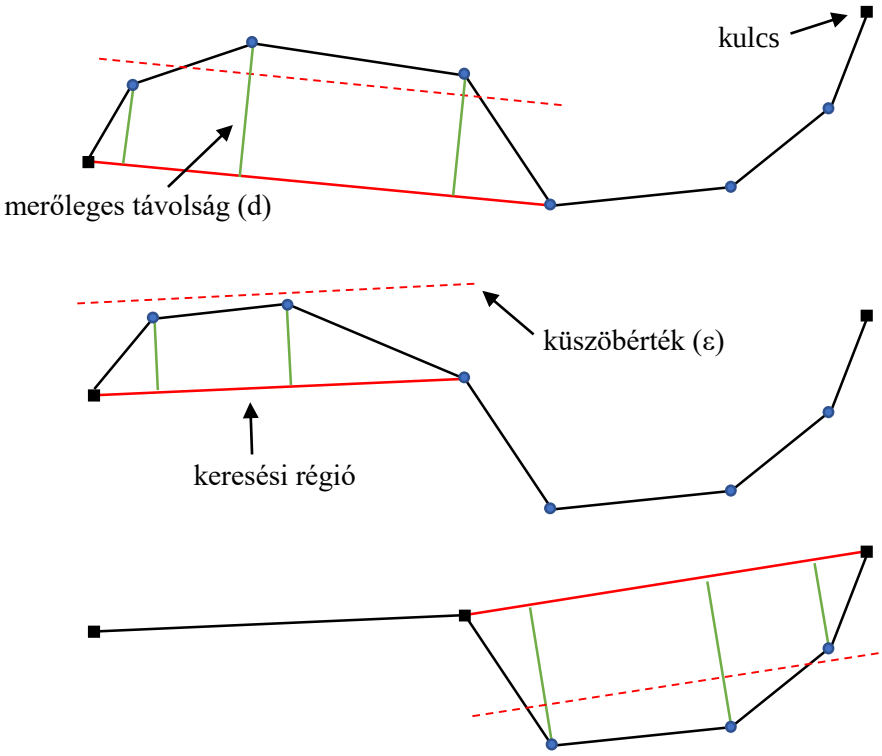
(a tűrésértéket 0.15-re állítottam be), akkor az algoritmus pontossága elfogadható volt (elvétve tapasztaltam határvonaltól kívül eső út pontokat). Ezek után a tűrésértéket (epsilon) 0.2-re állítottam be, ezzel az eredeti vonallánc pontjait megközelítőleg a harmadára csökkentettem. Hasonlóan az N-edik pont algoritmushoz sajnos itt is megjelent a határvonal pontatlansága, ezt szemlélteti a 30. ábra.



30. ábra – Merőleges távolság algoritmusnál megjelenő pontatlanság

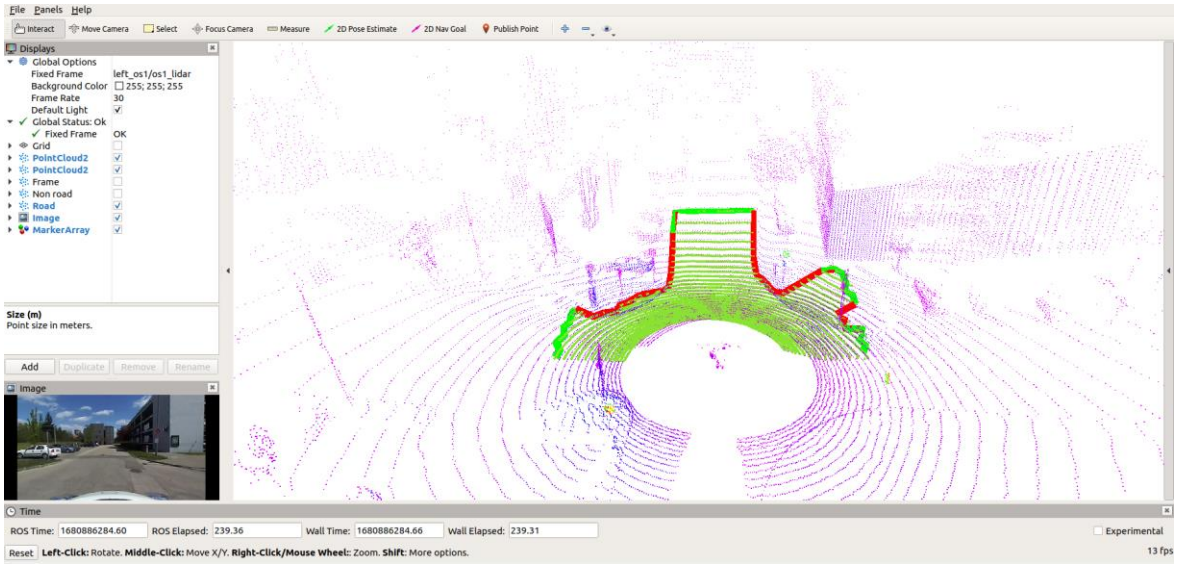
A Lang módszer a korlátozottan kiterjesztett lokális feldolgozási algoritmusokba tartozik, amit Lang fejlesztett ki 1969-ben. Ennél a vonallánc egyszerűsítő algoritmusnál a felhasználó egy rögzített méretű keresési tartományt határoz meg, aminek az első és utolsó pontja egy szegmenst alkot. Ez a szegmens az egyes közbenső pontok merőleges távolságának kiszámítására szolgál. Minden keresési régió meghatározott számú, egymást követő eredeti pontot tartalmazó régióként inicializálódik. A szegmens és a közbülső pontok közötti merőleges távolságok kiszámításra kerülnek. Amennyiben a számított merőleges távolság (d) nagyobb, mint a felhasználó által meghatározott tűrésérték (epsilon), akkor a keresési tartomány az utolsó pont kizárásával összezsugorodik (csökken) és a távolságokat újra kiszámítja. Ez a folyamat mindaddig folytatódik, amíg a közbülső pontoktól számított összes merőleges távolság a felhasználó által meghatározott tűrésérték alá nem kerül, vagy amíg nincs több közbenső pont. Az összes közbenső pont áthelyezése után egy új keresési régió kerül meghatározásra a legutóbbi keresési régió utolsó pontjának megadásával. Ez a folyamat megismétlődik és az eredeti vonal fölé tolódik, amíg el nem éri a vonallánc utolsó pontját. Ennél a vonallánc egyszerűsítő algoritmusnál tehát a felhasználónak kettő fontos

tulajdonságot kell meghatározni: a keresési tartományt és a tűréshatárt (epsilon). A Lang algoritmus vonallánc egyszerűsítését a 31. ábra szemlélteti, ahol a keresési tartomány a 4-es előretekintési érték felhasználásával mutatja be. Ez azt jelenti, hogy az eredményül kapott egyszerűsítés mindig az eredeti pontok legalább negyedét, vagy ha úgy tetszik 25%-át fogja tartalmazni.



31. ábra – Lang algoritmus vonallánc egyszerűsítése [19]

Miután megértettem az algoritmus működését, megírtam ezt a kódot is, ami a keresési tartomány 4-es előretekintési értéket alkalmazza a marker pontok egyszerűsítésére és megvizsgáltam ennek is a pontosságát. Először a harmadára csökkentettem az eredeti vonallánc pontjait úgy, hogy a tűrésértéket 0.30-ra állítottam be. Ezek után a tűrésértéket (epsilon) 0.45-re állítottam be, amivel az eredeti vonallánc pontjait megközelítőleg a negyedére csökkentettem. Azt tapasztaltam, hogy a Lang algoritmussal sokkal pontosabb határvonal meghatározást lehet elérni úgy, hogy közben az eredeti vonalláncot is akár a negyedére csökkentettem. A végleges forráskódban is ezt a vonallánc egyszerűsítő algoritmust alkalmaztam. Készítettem egy képet (32. ábra), mely a Lang vonallánc egyszerűsítő algoritmus pontosságát mutatja be, ahol a marker pontok darabszáma megközelítőleg negyedére van csökkentve [18] [19] [20].



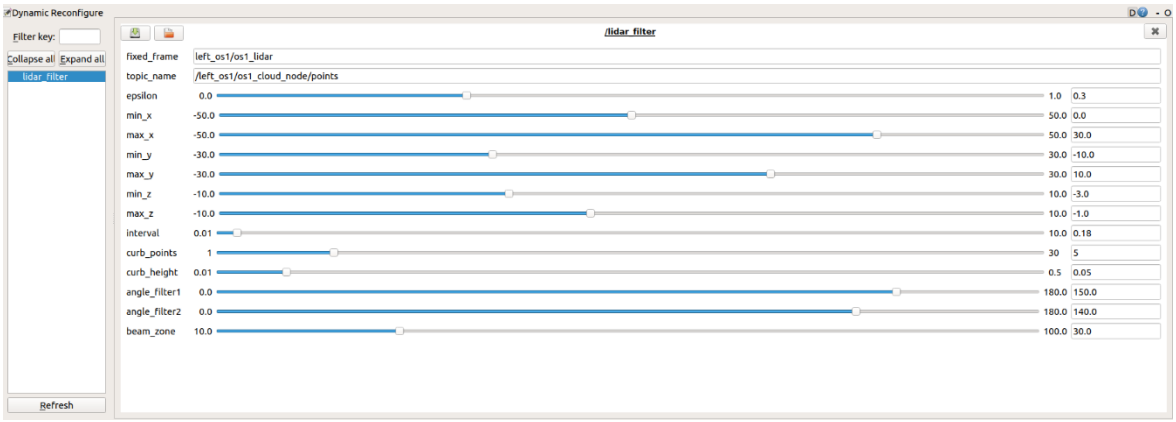
32. ábra – Lang algoritmus pontossága

6.11 Dynamic reconfigure

A paraméterek globálisan elérhető értékek például egész számok, valós számok, karakterláncok, vagy logikai értékek. A ROS bármely csomópontja (node) hozzáférhet és módosíthatja ezeket az értékeket futás közben. Olyan adatokhoz használunk paramétereket, amely várhatóan nem fognak sokat változni az idő múlásával. Ezeket a paramétereket egy központi paraméter szerver tárolja. A paraméter szerver olyan, mint egy szótár, amely tartalmazza a paraméter nevét és a paraméter megfelelő értékét. A paraméter szerver a ROS master része, így automatikusan elindul a roscore, vagy a roslaunch indításakor. A dynamic reconfigure csomag a paraméterek értékeinek dinamikus frissítésére biztosít lehetőséget. Ez a csomag lehetővé teszi számunkra, hogy futás közben frissítsük a paraméter szerveren lévő paramétereket. Ezeket a változásokat kiküldhetjük egy aktív csomópontnak (node), amelynek például a paraméter legfrissebb értékére van szüksége. Ez egy nagyszerű dolog, mert ez azt jelenti, hogy nem kell újraindítani a csomópontot ahhoz, hogy felismerje egy paraméter értékének változását [17].

Az elkészített projektfeladatomban én is alkalmaztam dinamikusan változtatható paramétereket. Azért szükséges ilyen paramétereket használni, mert a különböző LIDAR szenzorok, különböző tulajdonságokkal rendelkeznek (például eltérő vertikális, vagy horizontális szögfelbontás). Tehát ezeket a paramétereket a LIDAR jellemzőihez kell igazítani, ha azt szeretnénk, hogy a program megfelelően dolgozza fel az adatokat. Létrehoztam első lépésként egy dynamic_reconf.cfg nevű fájlt a cfg könyvtárban. Ebben a

fájlban az add függvény segítségével hozzáadtam a paramétereket a paraméterlistához és meghatároztam ezek alapértékeit és határértékeit. Ez egy python programozási nyelven írt forráskód. Ezek után a CMakeLists.txt fájlban hozzáadtam a catkin működéséhez szükséges komponensekhez a dynamic_reconfigure-t, majd a paraméterek generálásához megadtam az előzőleg létrehozott dynamic_reconf.cfg nevű fájl elérhetőségét és hozzáadtam a gencgf-t , mint függőség. A lidar_filter.cpp forráskódban készítettem egy params_callback nevű függvényt, ami a cfg könyvtárban található dynamic_reconf.cfg nevű fájlból veszi a paraméterek értékeit és a függvényben ezekkel az értékekkel inicializálja a programban használt változókat. A main függvényben létrehoztam a dynamic reconfigure szervert, ami az előzőleg létrehozott params_callback nevű függvényt hívja meg. Amikor elindítjuk a roslaunch paranccsal a programot, akkor megjelenik egy ablakban a paraméterlista is, ahol lehetőségünk van futás közben is módosítani a paraméterek értékét. A dynamic reconfigure grafikus felületét szemlélteti a 33. ábra.



33. ábra – A dynamic reconfigure grafikus felülete

Az alábbi táblázatban (34. ábra) felsoroltam a dynamic reconfigure-ban található paramétereket, a hozzájuk tartozó leírásokat, a beállított alapértékeket és a határértékeket.

Azonosító	Paraméter leírása	Alapérték	Határértékek
epsilon	Merőleges távolságméréskor a küszöbérték.	0.3 m	[0.0; 1.0]
min_x	Vizsgált terület X (min) koordinátája méterben.	0.0 m	[-50.0; 50.0]
max_x	Vizsgált terület X (max) koordinátája méterben.	30.0 m	[-50.0; 50.0]
min_y	Vizsgált terület Y (min) koordinátája méterben.	-10.0 m	[-30.0; 30.0]
max_y	Vizsgált terület Y (max) koordinátája méterben.	10.0 m	[-30.0; 30.0]

min_z	Vizsgált terület Z (min) koordinátája méterben.	-3.0 m	[-10.0; 10.0]
max_z	Vizsgált terület Z (max) koordinátája méterben.	-1.0 m	[-10.0; 10.0]
interval	LIDAR vertikális szögfelbontásának intervalluma fokban.	0.18°	[0.01; 0.18]
curb_points	Járdaszegélyen a pontok becsült száma.	5 db	[1; 30]
curb_height	Járdaszegély becsült minimális magassága méterben.	0.05 m	[0.01; 0.5]
angle_filter1	Három pont által bezárt szög vizsgálata X = 0 mellett. A szög küszöbértéke fokban.	150°	[0;180]
angle_filter2	Két vektor által bezárt szög vizsgálata Z = 0 mellett. A szög küszöbértéke fokban.	140°	[0;180]
beam_zone	A vizsgált sugárzóna fokban.	30°	[10; 100]

34. ábra – Paraméterek jellemzői

6.12 Összehasonlító elemzések

Ebben a fejezetben egy összehasonlító elemzést készítettem, ami az általam készített projektfeladatot és az Urban Road Filter tudományos cikkhez tartozó forráskódot hasonlítja össze. A programok teszteléséhez hardveres erőforrásként egy Asus FX507ZR típusú laptopot használtam, amiben egy Intel Core i-12700H 2.30 GHz típusú processzor, 32 GB RAM memória, Nvidia GeeForce RTX 3070 típusú videókártya és 1TB SSD tárhely volt. A kialakított virtuális gép a processzor 14 magjából 10 magot használt és a memória tekintetében pedig a 32 GB-ból 20 GB-ot használt. Ezeket a beállításokat a VirtualBox alkalmazás konfigurációs részében állítottam be.

Az összehasonlító elemzéshez a Győri Egyetem kampuszán 2021-ben rögzített rosbag-et használtam fel (leaf-2021-04-23-campus.bag), melyben megtalálhatóak a LIDAR alapú pontfelhő adathalmazok. A teljesítményértékek meghatározásához az Ubuntu Linux rendszerben található System Monitor (rendszerfigyelő) alkalmazást használtam. A System Monitor program képes a futó folyamatokat (process) különválasztva megjeleníteni, így könnyen látható a grafikus felületén, hogy az egyes programok milyen erőforrásokat használnak fel.

A programok összehasonlításakor egyforma Rviz konfigurációt alkalmaztam: az Ouster LIDAR teljes ponthalmazát (/left_os1/os1_cloud_node/points), az út pontokat (filtered_road) és az út határvonalát (marker_array) jelenítettem meg. Ugyanígy a paraméterek tekintetében is egyforma konfigurációt alkalmaztam: itt az alapértékeket

állítottam be (ami megtalálható a 34. ábrán) mindkét vizsgálatkor. A teszteléskor kapott teljesítményértékek eredményeit az alábbi összehasonlító táblázat (35. ábra) tartalmazza.

Leírás	Lidar filter projekt feladat	Urban Road Filter projekt
Program CPU igénye százalékban	~5.64 – 7.22 %	~4.44 - 6.20 %
Program memória igénye megabájtban	~12.1 - 31.1 MB	~22.5 – 24.0 MB
Rviz megjelenítő CPU igénye százalékban	~10.96 – 12.02 %	~10.34 - 11.52 %
Rviz megjelenítő memória igénye megabájtban	~110.1 - 121.7 MB	~107.2 – 118.8 MB
Rosbag CPU igénye százalékban	~0.96 - 1.30 %	~0.96 - 1.30 %
Rosbag memória igénye megabájtban	~34.6 MB	~34.6 MB
Nem út pontok feldolgozási ideje Hertzben	~19 – 20 Hz	~19 – 20 Hz
Nem út pontok feldolgozási ideje másodpercben	~0.012 – 0.127 sec	~0.020 – 0.115 sec
Út pontok feldolgozási ideje Hertzben	~19 – 20 Hz	~19 – 20 Hz
Út pontok feldolgozási ideje másodpercben	~0.026 – 0.105 sec	~0.016 – 0.109 sec
Marker array feldolgozási ideje Hertzben	~19 – 20 Hz	~20 Hz
Marker array feldolgozási ideje másodpercben	~0.018 – 0.102 sec	~0.017 – 0.098 sec

35. ábra – Teszteléskor kapott teljesítményértékek eredményei

Az összehasonlító táblázatban szereplő teljesítményértékeknél közelítő intervallumot határoztam meg, mivel gyakran változott a feltérképezendő terület. Az elemzések azt mutatják, hogy az általam írt program a CPU és a memória tekintetében is több erőforrást igényel, mint az Urban Road Filter. Az értékek itt egy-két százalékban térnek el. Az Rviz megjelenítéséhez és a rosbag futtatásához mind a kettő projektnél hasonló erőforrásigények jelentkeztek. A topic-ok feldolgozási idejében sem vehető észre nagyobb különbség, amennyiben összehasonlítjuk a kettő projektet: ~19-20 Hz az átlagos futási idejük, másodpercben meghatározva ~0.012 – 0.127 sec.

6.13 Továbbfejlesztési lehetőségek

A projektfeladat készítése közben nagyon sok forráskódot tanulmányoztam az interneten, amik további ötleteket adtak a továbbfejlesztési lehetőségek tekintetében. Vizsgáltam olyan

projektet, ahol a pontok tárolása struktúrával volt megoldva, de olyat is láttam, ahol ezt vektorokkal oldották meg. A különböző szűrési algoritmusok elkészítését és az ehhez kapcsolódó számításokat egyszerűbben lehet velük leprogramozni, mint a tömbökkel, tehát érdemes lenne ezeket a lehetőségeket megvizsgálni. Ezen kívül az adatok tárolásához és a számításokhoz a lehető legkevesebb adattárolót (változók, tömbök) kellene felhasználni, amely további javulást eredményezne a futási időn.

A nem út pontok szűrését további algoritmusokkal finomítani kellene ahhoz, hogy minél pontosabb eredményt kapjunk. Ez az első és egyben a legfontosabb szűrési szakasz, mivel erre épül a többi is. A projektfeladat elkészítésekor három vonallánc egyszerűsítő algoritmussal ismerkedtem meg (N-edik, merőleges távolság, Lang). Létezik még sok más vonallánc egyszerűsítő algoritmus, mint például a Reumann-Witkam, az Opheim, a Ramer-Douglas-Peucker. Érdemes lenne ezeket is leprogramozni és tesztelni a marker pontokon. A projektfeladatomban egy 64 csatornás Ouster LIDAR adatit használtam fel, azonban manapság léteznek már 128 csatornás szenzorok is, ezért további optimalizálásokat kellene elvégezni a forráskódon a megfelelő működés elérése érdekében.

A vezethető útfelület pontos meghatározásához manapság egyre többen alkalmaznak konvolúciós neurális hálókat is. Ehhez további ismeretek megszerzése szükséges, mint például TensorFlow, PyTorch.

7 ÖSSZEGZÉS

A diplomamunkám elkészítése során megértettem azt, hogy mit is jelent pontosan az autonóm jármű kifejezés. Megismerkedtem a Sae International J3016 szabvánnyal, amely a vezetési automatizálás hat különböző szintjét határozza meg. Elméleti ismeretekkel bővítettem tudásomat, melyek az autonóm járműveknél alkalmazott hardver- és szoftverarchitektúrákkal kapcsolatosak. Sok magáncég és egyetem is foglalkozik autonóm járművek kutatásával és az tapasztalatok azt mutatják, hogy nem elég egyetlen szenzort alkalmazni az autonóm járműveken. Ahhoz, hogy pontos eredményeket kapjunk többféle érzékelőt (LIDAR, kamera, radar) kell használni együttesen. A szenzorfúzió folyamat egyesíti a több érzékelőtől kapott információkat, ezzel csökkenti az egyes szenzortípusok hiányosságait és javítja a hatékonyságot, megbízhatóságot.

A lidar filter projektfeladatom elkészítése során megismerkedtem a robotok működtetésére szolgáló Robot Operating System szoftverkészlettel is. A LIDAR szenzortól érkező ponthalmaz tárolásához és az egyéb számításokhoz dinamikus tömböket használtam fel. Különböző szűrési eljárásokat alkalmazva, meghatároztam a vizsgált területen a nem út pontokat és az út pontokat (LIDAR adatok szegmentálása). A projektfeladatnak ezek voltak a legfontosabb részei, ezért itt a lehető legpontosabb eredményeket kell elérni leprogramozott algoritmussal.

Ezek után egy kétdimenziós tömbbe azokat a pontokat gyűjtöttem össze a vizsgált területből, amik a vezethető út szélét határozták meg, azaz a marker pontokat. Vonallánc egyszerűsítő algoritmusokat vizsgálva, majd alkalmazva sikerült a marker pontokat közel a negyedére redukálni, így a félpoligonok egyszerűsítve, kevés pontokból állnak. A paraméterek futás közbeni módosításához dynamic reconfigure csomagot használtam fel, ami lehetővé teszi a különböző típusú (16 – 32 – 64 csatornás) szenzorokból érkező adatok vizsgálatát. A projektfeladat végén pedig egy összehasonlító elemzést készítettem, ami az általam készített programot és az Urban Road Filter tudományos cikkhez tartozó forráskódot hasonlítja össze. A program fejlesztésekor arra törekedtem, hogy a feladat-kiíró lapon meghatározott követelményeknek eleget tudjak tenni.

IRODALOMJEGYZÉK

1. **A research platform for autonomous vehicles technologies research in the insurance sector**
   By: de Miguel, M.Á.; Moreno, F.M.; Marín-Plaza, P.; Al-Kaff, A.; Palos, M.; Martín, D.; Encinar-Martín, R.; García, F.. Applied Sciences (Switzerland), August 2020, 10(16) Language: English. MDPI AG DOI: 10.3390/app10165655, Database: Scopus®, pp. 1 – 12

2. **Sensor and Sensor Fusion Technology in Autonomous Vehicles: A Review**
   By: De Jong Yeong; Gustavo Velasco-Hernandez; John Barry; Joseph Walsh. In: Sensors, Vol 21, Iss 2140, p 2140 (2021); MDPI AG, 2021. Language: English, Database: Directory of Open Access Journals, pp. 1 – 29

3. **A Survey of Autonomous Vehicles: Enabling Communication Technologies and Challenges**
   By: M. Nadeem Ahangar; Qasim Z. Ahmed; Fahd A. Khan; Maryam Hafeez. In: Sensors, Vol 21, Iss 706, p 706 (2021); MDPI AG, 2021. Language: English, Database: Directory of Open Access Journals, pp. 1 – 12

4. **Architecture Design and Implementation of an Autonomous Vehicle**
   By: Zong, W.; Zhang, C.; Wang, Z.; Zhu, J.; Chen, Q.. In: IEEE Access Access, IEEE. 6:21956-21970 2018; USA: IEEE Language: English, Database: IEEE Xplore Digital Library, pp. 21956 – 21960

5. **Autonomous vehicles: from paradigms to technology**
   By: Silviu Ionita. IOP Conference Series: Materials Science & Engineering, Oct2017, Vol. 252 Issue 1, p1-1, 1p. Publisher: IOP Publishing, Database: Complementary Index, pp. 1 – 6

6. **An Overview of Lidar Imaging Systems for Autonomous Vehicles**
   By: Royo, Santiago; Ballesta-Garcia, Maria. Applied Sciences-Basel; OCT 2019; 9; 19, Database: Science Citation Index, pp. 1 – 9

7. **Semantic segmentation - Udaity's self-driving car engineer nanodegree**
   https://medium.com/intro-to-artificial-intelligence/semantic-segmentation-udaitys-self-driving-car-engineer-nanodegree-c01eb6eaf9d

8. **Real-time hybrid multi-sensor fusion framework for perception in autonomous vehicles**
   By: Jahromi, B.S.; Tulabandhula, T.; Cetin, S.. Sensors (Switzerland), 2 October 2019, 19(20) Language: English. MDPI AG DOI: 10.3390/s19204357, Database: Scopus®, pp. 1 – 8

9. **Autonomous Robot Project Based on the Robot Operating System Platform**
   By: Szymon Cherubin, Wojciech Kaczmarek, Natalia Daniel (Poland), 30 December 2022, DOI 10.5604/01.3001.0016.1462, pp. 1 – 17

10. **The Robot Operating System: Package Reuse and Community Dynamics**
By: Pablo Estefo, Jocelyn Simmonds, Romain Robbes, Johan Fabry, Journal of Systems and Software; 2 October 2018, DOI: 10.1016/j.jss.2019.02.024, pp. 1 – 10

11. **A Robot Operating System Framework for Secure UAV Communications**
By: Hyojun Lee, Jiyoung Yoon, Min-Seong Jang and Kyung-Joon Park, Sensors 2021, 15 February 2021, doi.org/10.3390/s21041369, pp. 1 – 8

12. **Deep Learning on Point Clouds and Its Application: A Survey**
By: Weiping Liu, Jia Sun, Wanyi Li, Ting Hu and Peng Wang, In: Sensors 2019, Vol 19, 4188, Published: 26 September 2019, Database: www.mdpi.com/journal/sensors, pp. 1 – 22

13. **Comprehensive Automated 3D Urban Environment Modelling Using Terrestrial Laser Scanning Point Cloud**
By: Pouria Babahajiani, Lixin Fan, Joni-Kristian Kämäräinen and Moncef Gabbouj, 19 December 2016, DOI: 10.1109/CVPRW.2016.87, Database: IEEE Xplore, pp. 1 – 7

14. **A Point Cloud-Based Robust Road Curb Detection and Tracking Method**
By: Guojun Wang, Jian Wu, Rui He and Shun Yang, 14 February 2019, DOI: 10.1109/access.2019.2898689, Database: IEEE Access, pp. 24611 – 24626

15. **Curb detection in urban traffic scenarios using LiDARs point cloud and semantically segmented color images**
By: Selma Evelyn Catalina Deac, Ion Giosan, and Sergiu Nedevschi, October 2019, DOI: 10.1109/ITSC.2019.8917020, Database: ResearchGate, pp. 1 – 9

16. **Road-Segmentation based Curb Detection Method for Self-driving via a 3D-LiDAR Sensor**
By: Yihuan Zhang, Jun Wang Xiaonian Wang and John M. Dolan, 14 February 2018, DOI: 10.1109/TITS.2018.2789462, Database: IEEE Access, pp. 1-11

17. **Real-Time LIDAR-Based Urban Road and Sidewalk Detection for Autonomous Vehicles**
By: Ernő Horváth, Claudiu Pozna, Miklós Unger, 28 December 2021, Sensors 2022, 22(1), 194, DOI:10.3390/s22010194, Database: ResearchGate, pp. 1-17

18. **Vonalak automatizált generalizálása az elméletben és a gyakorlatban – Vonalegyszerűsítő és -simító eljárások**
By: Ungvári Zsuzsanna, Somogyi Árpád, Dr. Lovas Tamás, Database: Geodézia és Kartográfia, 2017/2 (69. évfolyam), pp. 1-9

19. **Polyline simplification**
By: Elmar de Koning, 25 June 2011, Code Project
https://www.codeproject.com/Articles/114797/Polyline-Simplification#headingIntro

20. **Efficient Implementation of Polyline Simplification for Large Datasets and Usability Evaluation**
By: Şadan Ekdemir, September 2011, Uppsala Universitet, Sweden, pp. 1-19

ÁBRAJEGYZÉK

MELLÉKLETEK

1. Lidar filter forráskódja

```
// A program futtatásához a szükséges include állományok. Ezek javarészt
matematikai függvények gyűjteménye,
// ROS-hoz, pointcloud-ok, marker-ek, dynamic reconfigure használatához
szükséges állományok.
#include <iostream>
#include <cmath>
#include <math.h>
#include <algorithm>
#include <ros/ros.h>
#include <pcl_ros/point_cloud.h>
#include <visualization_msgs/Marker.h>
#include <visualization_msgs/MarkerArray.h>
#include <dynamic_reconfigure/server.h>
#include <lidar_filter/dynamic_reconfConfig.h>

// GLOBÁLIS VÁLTOZÓK

// A topic név (amire felíratkozunk) és a fixed frame tárolásához szükséges
string változók.
std::string topic_name;
std::string fixed_frame;

// Az alkalmazott LIDAR csatornaszáma.
int channels = 64;

// A vizsgált területhez szükséges változók (X, Y, Z koordináta szerint a
minimális és maximális közötti rész).
float min_x;
float max_x;
float min_y;
float max_y;
float min_z;
float max_z;

// A LIDAR vertikális szögfelbontásának intervalluma.
float interval;

// A becsült pontok száma a járdaszegélyen.
int curb_points;

// A becsült minimális járdaszegély magasság.
float curb_height;

// Három pont által bezárt szög, X = 0 érték mellett.
float angle_filter1;

// Két vektor által bezárt szög, Z = 0 érték mellett.
float angle_filter2;

// A vizsgált sugárzóna mérete.
float beam_zone;
```

```
// Merőleges távolságméréshez szükséges küszöbérték (Lang algoritmusnál a
megfelelő érték: 0.3).
float epsilon;

// PARAMÉTEREK BEÁLLÍTÁSÁHOZ SZÜKSÉGES FÜGGVÉNY

// Az értékeket a cfg könyvtárban található dynamic_reconf.cfg nevű fájlból
veszi.
void params_callback(lidar_filter::dynamic_reconfConfig &config, uint32_t
level)
{
    fixed_frame = config.fixed_frame;
    topic_name = config.topic_name;
    epsilon = config.epsilon;
    min_x = config.min_x;
    max_x = config.max_x;
    min_y = config.min_y;
    max_y = config.max_y;
    min_z = config.min_z;
    max_z = config.max_z;
    interval = config.interval;
    curb_points = config.curb_points;
    curb_height = config.curb_height;
    angle_filter1 = config.angle_filter1;
    angle_filter2 = config.angle_filter2;
    beam_zone = config.beam_zone;
}


// GYORSRENDEZŐ SEGÉDFÜGGVÉNYEK

// Segédfüggvény, ami a két elemet felcseréli (swap).
void swap(float *a, float *b)
{
    float temp = *a;
    *a = *b;
    *b = temp;
}

// Ez a függvény az utolsó elemet pivotnak veszi, a pivot elemet a megfelelő
helyre helyezi a rendezett tömbben.
// Az összes kisebbet (tehát kisebb, mint a pivot) a pivottól balra, a
nagyobb elemeket pedig jobbra helyezi.
int partition(float ***arr_3d, int arc, int piece, int low, int high)
{
    float pivot = arr_3d[arc][low][4];
    int i = low - 1;
    int j = high + 1;

    while (true)
    {
        // Keresse meg a bal szélső elemet, amely nagyobb, mint a pivot.
        do
        {
            i++;
        } while (arr_3d[arc][i][4] < pivot);

        // Keresse meg a jobb szélső elemet, amely kisebb, mint a pivot.
```

```
        do
        {
            j--;
        } while (arr_3d[arc][j][4] > pivot);

        // Ha a két mutató találkozik.
        if (i >= j)
            return j;

        for (int sw = 0; sw < 7; sw++)
            {
                swap(&arr_3d[arc][i][sw], &arr_3d[arc][j][sw]);
            }
    }
}

// A gyorsrendező fő függvénye.
// Az arr_3d a rendezendő tömb, low a kezdő index, high a befejező index.
void quicksort(float ***arr_3d, int arc, int piece, int low, int high)
{
    if (low < high)
    {
        // Pi a particionálási index.
        int pi = partition(arr_3d, arc, piece, low, high);

        // Az elemek külön rendezése a partíció előtt és a partíció után.
        quicksort(arr_3d, arc, piece, low, pi);
        quicksort(arr_3d, arc, piece, pi + 1, high);
    }
}

// MERŐLEGES TÁVOLSÁGMÉRŐ FÜGGVÉNY

float perpendicular_distance(float ax, float ay, float bx, float by, float
point_x, float point_y)
{
    float dx = bx - ax;
    float dy = by - ay;

    // Normalizálás
    float mag = sqrt(pow(dx, 2) + pow(dy, 2));

    if (mag > 0.0)
    {
        dx /= mag;
        dy /= mag;
    }

    float pv_x = point_x - ax;
    float pv_y = point_y - by;

    float pv_dot = dx * pv_x + dy * pv_y;

    // Léptékvonal irányvektor
    float ds_x = pv_dot * dx;
    float ds_y = pv_dot * dy;

    float a_x = pv_x - ds_x;
    float a_y = pv_y - ds_y;
```

```
    // Visszatérünk a merőleges távolság eredményével.
    return sqrt(pow(a_x, 2) + pow(a_y, 2));
}

// PUBLISHER-EK LÉTREHOZÁSA

// A vizsgált terület pontjai (Frame)
ros::Publisher pub_frame;

// Nem út pontok
ros::Publisher pub_non_road;

// Út pontok
ros::Publisher pub_road;

// Marker array
ros::Publisher pub_marker_array;

// A SZŰRÉST VÉGZŐ FÜGGVÉNY

void filter(const pcl::PointCloud<pcl::PointXYZ> &msg)
{
    // Segédváltozók a ciklusokhoz a függvényben.
    int i, j, k, l;

    // Egy pont tárolásához szükséges.
    pcl::PointXYZ pt;

    // A vizsgált terület pontjainak tárolásához szükséges pontfelhő
létrehozása.
    pcl::PointCloud<pcl::PointXYZ> filtered_frame;

    // Nem út pontok tárolásához szükséges pontfelhő létrehozása.
    pcl::PointCloud<pcl::PointXYZ> filtered_non_road;

    // Út pontok tárolásához szükséges pontfelhő létrehozása.
    pcl::PointCloud<pcl::PointXYZ> filtered_road;

    // VIZSGÁLT PONTOK KERESÉSE ÉS HOZZÁADÁSA A FILTERED_FRAME-HEZ

    // A for ciklussal végigmegyünk a bejövő (msg) pontfelhőn (az összes
ponton).
    // Amelyik pont megfelel az if feltételnek (azaz a frame méreten belül
elhelyezkedő pontok),
    // azokat hozzáadjuk a filtered_frame topic-hoz (push.back()).
    for (i = 0; i <= msg.size(); i++)
    {
        if (msg.points[i].x >= min_x && msg.points[i].x <= max_x &&
            msg.points[i].y >= min_y && msg.points[i].y <= max_y &&
            msg.points[i].z >= min_z && msg.points[i].z <= max_z &&
            msg.points[i].x + msg.points[i].y + msg.points[i].z != 0)
            {
                pt.x = msg.points[i].x;
                pt.y = msg.points[i].y;
                pt.z = msg.points[i].z;
                filtered_frame.push_back(pt);
            }
    }
```

```
    // Meghatározzuk a pontok darabszámát és egy változóban tároljuk (ezt
később több helyen is felhasználjuk majd).
    int piece = filtered_frame.points.size();

    // Feltételként megadjuk, hogy legalább 40 pont legyen a filtered_frame-
ben, azaz a vizsgált területen.
    // Kevés pontot vizsgálni nincs értelme.
    if (piece >= 40)
    {
        // DINAMIKUS 2D TÖMB LÉTREHOZÁSA A PONTOK ÉRTÉKEIHEZ ÉS EGYÉB
SZÁMÍTÁSOKHOZ

        // 0. oszlop tárolja a pontok X értékét, 1. oszlop tárolja a pontok Y
értékét,
        // 2. oszlop tárolja a pontok Z értékét, 3. oszlop tárolja a pont és
az origótól vett távolságot (D),
        // 4. oszlop tárolja a pontok szögfelbontását (Alfa).
        float **arr_2d = new float*[piece]();

        for (i = 0; i < piece; i++)
        {
            arr_2d[i] = new float[5];
        }

        // DINAMIKUS 2D TÖMB FELTÖLTÉSE ADATOKKAL

        // A szögfüggvényeknél a részeredmények tárolásához szükséges
változó.
        float part_result;

        // Egy tömb, amiben eltároljuk a különböző szögfelbontásokat. Ez
megegyezik a LIDAR csatornaszámával.
        // A tömböt feltöltjük nulla értékekkel.
        float angle[channels] = {0};

        // A szögfelbontásokat tároló tömb feltöltéséhez szükséges változó.
        int index = 0;

        // Az adott szög új körvonalhoz tartozik vagy sem segédváltozója?
        int new_circle;

        // Végigmegyünk az összes ponton és feltöltjük az értékekkel a tömb
oszlopait (X, Y, Z).
        // Kiszámítjuk az origótól vett távolságát (D) és a szögfelbontását
(Alfa).
        for (i = 0; i < piece; i++)
        {
            arr_2d[i][0] = filtered_frame.points[i].x;
            arr_2d[i][1] = filtered_frame.points[i].y;
            arr_2d[i][2] = filtered_frame.points[i].z;

            arr_2d[i][3] = sqrt(pow(arr_2d[i][0], 2) + pow(arr_2d[i][1], 2) +
pow(arr_2d[i][2], 2));

            part_result = abs(arr_2d[i][2]) / arr_2d[i][3];

            // Kerekítési problémák miatt szükségesek az alábbi sorok.
            if (part_result < -1)
```

```
                part_result = -1;

            else if (part_result > 1)
                part_result = 1;

            // Ha a Z értéke kisebb nullánál, akkor koszinusz függvényt kell
alkalmazni.
            if (arr_2d[i][2] < 0)
                arr_2d[i][4] = acos(part_result) * 180 / M_PI;

            // Ha a Z értéke nagyobb vagy egyenlő nullánál, akkor pedig
szinusz függvényt kell alkalmazni.
            else if (arr_2d[i][2] >= 0)
                arr_2d[i][4] = (asin(part_result) * 180 / M_PI) + 90;

            // Az alapvetés az, hogy az adott szög új körvonalhoz tartozik.
            new_circle = 1;

            for (j = 0; j < channels; j++)
            {
                if (angle[j] == 0)
                break;

                // Ha már korábban volt ilyen érték (egy meghatározott
intervallumon belül), akkor ez nem egy új körív.
                // A new_circle-be így nulla kerül és break-kel kilépünk a
folyamatból.
                if (abs(angle[j] - arr_2d[i][4]) <= interval)
                {
                    new_circle = 0;
                    break;
                }
            }

            // Amennyiben nem szerepel még a tömbben ilyen érték, akkor az
egy új körívet jelent.
            if (new_circle == 1)
            {
                // Ha több körív keletkezne, mint 64 valamilyen okból
kifolyólag, akkor hiba keletkezne.
                // Az alábbi feltétel ezt kezeli le, illetve eltároljuk a
különböző szögfelbontásokat.
                if(index < channels)
                {
                    angle[index] = arr_2d[i][4];
                    index++;
                }
            }
        }

        // A sort függvénnyel növekvő sorrendbe rendezzük a
szögfelbontásokat.
        std::sort(angle, angle + index);

        // DINAMIKUS 3D TÖMB LÉTREHOZÁSA A PONTOK ÉRTÉKEIHEZ ÉS EGYÉB
SZÁMÍTÁSOKHOZ

        // 0. oszlop tárolja a pontok X értékét, 1. oszlop tárolja a pontok Y
értékét,
```

```
        // 2. oszlop tárolja a pontok Z értékét, 3. oszlop tárolja a pont és
az origótól vett távolságot, de itt Z = 0 értékkel (D),
        // 4. oszlop tárolja a pontok helyzetét egy körben (forgásszög
számítása) (Alfa),
        // 5. oszlop tárolja X = 0 érték mellett az új Y koordinátákat (új
Y),
        // 6. oszlop tárolja a csoportszámokat, ami lehet 1 (út pontot
jelent), vagy 2 (nem út pontot jelent).
        float ***arr_3d = new float**[channels]();

        for (i = 0; i < channels; i++)
        {
            arr_3d[i] = new float*[piece];

            for (j = 0; j < piece; j++)
            {
                arr_3d[i][j] = new float[7];
            }
        }

        // DINAMIKUS 3D TÖMB FELTÖLTÉSE ADATOKKAL

        // Az adott köríveket tartalmazó csoportok (channels), megfelelő
sorindexeinek beállításához szükséges tömb.
        // Nulla értékkel fel kell tölteni.
        int index_array[channels] = {0};

        // Egy tömb, ami tárolja az adott köríven a legnagyobb távolságra
lévő pont értékét az origótól.
        // Tehát minden köríven egy pont értékét tárolja a tömbben. Nulla
értékkel fel kell tölteni.
        float max_distance[channels] = {0};

        // Hibás LIDAR csatornaszám esetén szükséges változó.
        int results;

        // Végigmegyünk az összes ponton a for ciklussal.
        for (i = 0; i < piece; i++)
        {
            results = 0;

            // Kiválasszuk a megfelelő körívet.
            for (j = 0; j < index; j++)
            {
                if (abs(angle[j] - arr_2d[i][4]) <= interval)
                {
                    results = 1;
                    break;
                }
            }

            if (results == 1)
            {
                // A 2D tömbből hozzáadjuk az X, Y, Z koordináta értékeit a
3D tömbhöz (a megfelelő körívhez).
                arr_3d[j][index_array[j]][0] = arr_2d[i][0];
                arr_3d[j][index_array[j]][1] = arr_2d[i][1];
                arr_3d[j][index_array[j]][2] = arr_2d[i][2];
```

```
                // A 2D tömbből kiszámítjuk az origótól vett távolságát, de
itt csak az X és Y értéket adjuk hozzá.
                arr_3d[j][index_array[j]][3] = sqrt(pow(arr_2d[i][0], 2) +
pow(arr_2d[i][1], 2));

                // Minden pontnak van egy szöge 360 fokban. Az adott pont
helyzete egy körben.
                part_result = (abs(arr_3d[j][index_array[j]][0])) /
arr_3d[j][index_array[j]][3];

                // Kerekítési problémák miatt szükségesek az alábbi sorok.
                if (part_result < -1)
                    part_result = -1;

                else if (part_result > 1)
                    part_result > 1;

                // A pont a kör I. negyedében található
                if (arr_3d[j][index_array[j]][0] >= 0 &&
arr_3d[j][index_array[j]][1] <= 0)
                    arr_3d[j][index_array[j]][4] = asin(part_result) * 180 /
M_PI;

                // A pont a kör II. negyedében található
                else if (arr_3d[j][index_array[j]][0] >= 0 &&
arr_3d[j][index_array[j]][1] > 0)
                    arr_3d[j][index_array[j]][4] = 180 - (asin(part_result) *
180 / M_PI);

                // A pont a kör III. negyedében található
                else if (arr_3d[j][index_array[j]][0] < 0 && 0 &&
arr_3d[j][index_array[j]][1] >= 0)
                    arr_3d[j][index_array[j]][4] = 180 + (asin(part_result) *
180 / M_PI);

                // A pont a kör IV. negyedében található
                else
                    arr_3d[j][index_array[j]][4] = 360 - (asin(part_result) *
180 / M_PI);

                // Vizsgálja, hogy az adott köríven az origótól a legnagyobb
távolságra van-e a pont.
                // Ha igen, akkor annak értékét elmenti.
                if (arr_3d[j][index_array[j]][3] > max_distance[j])
                    max_distance[j] = arr_3d[j][index_array[j]][3];

                index_array[j]++;
            }
        }

        // 2D DINAMIKUS TÖMB FELSZABADÍTÁSA

        for (i = 0; i < piece; i++)
        {
            delete [] arr_2d[i];
        }
        delete [] arr_2d;
```

```
        // NEM ÚT PONTOK SZŰRÉSE

        // A három vizsgált pontból a második és a harmadik pont
segédváltozója.
        int point_2, point_3;

        // A két szélső pont közötti távolság. A LIDAR forgása és a körív
szakadások miatt.
        float d;

        // A három pont által bezárt háromszög oldalainak hossza.
        float x1, x2, x3;

        // A három pont és a két vektor által bezárt szög.
        float alpha;

        // A két vektor változói.
        float va1, va2, vb1, vb2;

        // A magasságot is kell vizsgálni nem csak a szöget.
        float max1, max2;

        // Végig iterálunk az összes körön.
        for ( i = 0; i < index; i++)
        {
            // FILTER 1

            // Feltöltjük új Y értékekkel a 6. oszlopot.
            for (j = 1; j < index_array[i]; j++)
            {
                arr_3d[i][j][5] = arr_3d[i][j-1][5] + 0.0100;
            }

            // A adott kör pontjainak vizsgálata, X = 0 érték mellett.
            for (j = curb_points; j <= (index_array[i] - 1) - curb_points;
j++)
            {
                point_2 = j + curb_points / 2;
                point_3 = j + curb_points;

                // Kiszámoljuk a két szélső pont közötti távolságot.
                d = sqrt(
                    pow(arr_3d[i][point_3][0] - arr_3d[i][j][0], 2) +
                    pow(arr_3d[i][point_3][1] - arr_3d[i][j][1], 2));

                // Feltételhez kötjük, hogy a két szélső pont közötti
távolság kisebbnek kell lenni 5 méternél.
                if (d < 5.0000)
                {
                    // Meghatározzuk a három pont által bezárt háromszög
oldalainak hosszát (x1, x2, x3).
                    x1 = sqrt(
                        pow(arr_3d[i][point_2][5] - arr_3d[i][j][5], 2) +
                        pow(arr_3d[i][point_2][2] - arr_3d[i][j][2], 2));

                    x2 = sqrt(
                        pow(arr_3d[i][point_3][5] - arr_3d[i][point_2][5], 2)
+
```

```
                        pow(arr_3d[i][point_3][2] - arr_3d[i][point_2][2],
2));

                    x3 = sqrt(
                        pow(arr_3d[i][point_3][5] - arr_3d[i][j][5], 2) +
                        pow(arr_3d[i][point_3][2] - arr_3d[i][j][2], 2));

                    part_result = (pow(x3, 2) - pow(x1, 2) - pow(x2, 2)) / (-
2 * x1 * x2);

                    // Kerekítési problémák miatt szükségesek az alábbi
sorok.
                    if (part_result < -1)
                        part_result = -1;
                    else if (part_result > 1)
                        part_result = 1;

                    // Kiszámítjuk a három pont és a két vektor által bezárt
szöget.
                    alpha = acos(part_result) * 180 / M_PI;

                    // Ha a feltétel teljesül, akkor a 7. oszlopba 2-es szám
kerül, azaz nem út pont.
                    if (alpha <= angle_filter1 &&
                    (abs(arr_3d[i][j][2] - arr_3d[i][point_2][2]) >=
curb_height ||
                    abs(arr_3d[i][point_3][2] - arr_3d[i][point_2][2]) >=
curb_height) &&
                    abs(arr_3d[i][j][2] - arr_3d[i][point_3][2]) >= 0.05)
                    {
                        arr_3d[i][point_2][6] = 2;
                    }
                }
            }

            // FILTER 2

            // A adott kör pontjainak vizsgálata. Z = 0 érték mellett.
            for (j = curb_points; j <= (index_array[i] - 1) - curb_points;
j++)
            {
                // Kiszámoljuk a két vektor szélső pont közötti távolságot.
                d = sqrt(
                    pow(arr_3d[i][j + curb_points][0] - arr_3d[i][j -
curb_points][0], 2) +
                    pow(arr_3d[i][j + curb_points][1] - arr_3d[i][j -
curb_points][1], 2));

                // Feltételhez kötjük, hogy a két vektor szélső pont közötti
távolság kisebbnek kell lenni 5 méternél.
                if (d < 5.0000)
                {
                    // Kezdeti értékek beállítása. A max változókba bekerül Z
abszolút értéke.
                    max1 = abs(arr_3d[i][j][2]);
                    max2 = abs(arr_3d[i][j][2]);
                    va1 = 0, va2 = 0;
                    vb1 = 0, vb2 = 0;
```

```
                    // Az 'a' vektor és a legnagyobb magasság meghatározása.
                    for (k = j - 1;k >= j - curb_points; k--)
                    {
                        va1 = va1 + arr_3d[i][k][0] - arr_3d[i][j][0];
                        va2 = va2 + arr_3d[i][k][1] - arr_3d[i][j][1];

                        if (abs(arr_3d[i][k][2]) > max1)
                            max1 = abs(arr_3d[i][k][2]);
                    }

                    // A 'b' vektor és a legnagyobb magasság meghatározása.
                    for (k = j + 1;k <= j + curb_points; k++)
                    {
                        vb1 = vb1 + arr_3d[i][k][0] - arr_3d[i][j][0];
                        vb2 = vb2 + arr_3d[i][k][1] - arr_3d[i][j][1];

                        if (abs(arr_3d[i][k][2]) > max2)
                            max2 = abs(arr_3d[i][k][2]);
                    }

                    va1 = (1 / (float)curb_points) * va1;
                    va2 = (1 / (float)curb_points) * va2;
                    vb1 = (1 / (float)curb_points) * vb1;
                    vb1 = (1 / (float)curb_points) * vb1;

                    part_result = (va1 * vb1 + vb1 * vb2) / (sqrt(pow(va1, 2)
+ pow(va2, 2)) * sqrt(pow(vb1, 2) + pow(vb2, 2)));

                    // Kerekítési problémák miatt szükségesek az alábbi
sorok.
                    if (part_result < - 1)
                        part_result = -1;

                    if (part_result > 1)
                        part_result = 1;

                    // Kiszámítjuk a két vektor által bezárt szöget.
                    alpha = acos(part_result) * 180 / M_PI;

                    // Ha a feltétel teljesül, akkor a 7. oszlopba 2-es szám
kerül, azaz nem út pont.
                    // Ha a szög kisebb vagy egyenlő mint a beállított érték
(alapesetben ez 140 fok) és
                    // a beállított küszöbértéktől (alapesetben ez 5cm)
nagyobb vagy egyenlő Z értéke és
                    // a max1, max2 különbsége (abszolút értékben) nagyobb
vagy egyenlő mint 0.05
                    if (alpha <= angle_filter2 &&
                        (max1 - abs(arr_3d[i][j][2]) >= curb_height ||
                         max2 - abs(arr_3d[i][j][2]) >= curb_height) &&
                         abs(max1 - max2) >= 0.05)
                         {
                            arr_3d[i][j][6] = 2;
                         }
                }
            }
        }

        // ÚT PONTOK SZŰRÉSE
```

```
        // A tömb elemeinek gyorsrendezése körönként a szögeknek megfelelő
növekvő sorrendben.
        for (i = 0; i < index; i++)
        {
            quicksort(arr_3d, i, piece, 0, index_array[i] - 1);
        }

        // A körív mérete a megadott foknál. Minden körön ugyanakkor körív
méretet kell vizsgálni.
        float arc_distance;

        // Amennyiben az adott köríven, az adott szakaszon található
magaspont,
        // akkor 1-es értéket vesz fel a változó, egyébként nullát.
        int not_road;

        // Az aktuális köríven a szög nagysága.
        float current_degree;

        // A körív méret meghatározása. ((sugár * PI) / 180) * vizsgált
sugárzóna mérettel).
        arc_distance = ((max_distance[0] * M_PI) / 180) * beam_zone;

        // 0 foktól 360 fok - beam_zone-ig vizsgáljuk a ciklussal.
        for (i = 0; i <= 360 - beam_zone; i++)
        {
            // Azt veszük alapul, hogy az adott szakaszon nincs magaspont.
            not_road = 0;

            // Az első kör adott szakaszának vizsgálata.
            for (j = 0; arr_3d[0][j][4] <= i + beam_zone && j <
index_array[0]; j++)
            {
                if (arr_3d[0][j][4] >= i)
                {
                    // Nem vizsgáljuk tovább az adott szakaszt (break), mivel
találtunk benne magaspontot.
                    if (arr_3d[0][j][6] == 2)
                    {
                        not_road = 1;
                        break;
                    }
                }
            }

            // Amennyiben nem találtunk az első kör adott szakaszán
magaspontot, akkor továbblépünk a következő körre.
            if (not_road == 0)
            {
                // Az első kör szakaszát elfogadjuk, az adott pont tömb 7.
oszlopába 1-es szám kerül, azaz út pont.
                for (j = 0; arr_3d[0][j][4] <= i + beam_zone && j <
index_array[0]; j++)
                {
                    if (arr_3d[0][j][4] >= i)
                    {
                        arr_3d[0][j][6] = 1;
                    }
```

```
                }

                // További körök vizsgálata.
                for (k = 1; k < index; k++)
                {
                    // Új szöget kell meghatározni, hogy a távolabbi
körvonalakon is ugyanakkora körív hosszt vizsgáljunk.
                    if (i == 360 - beam_zone)
                    {
                        current_degree = 360;
                    }
                    else
                    {
                        current_degree = i + arc_distance / ((max_distance[k]
* M_PI) / 180);
                    }

                    // Az új kör pontjait vizsgáljuk.
                    for (l = 0; arr_3d[k][l][4] <= current_degree && l <
index_array[k]; l++)
                    {
                        if (arr_3d[k][l][4] >= i)
                        {
                            // Nem vizsgáljuk tovább az adott szakaszt
(break), mivel találtunk benne magaspontot.
                            if (arr_3d[k][l][6] == 2)
                            {
                                not_road = 1;
                                break;
                            }
                        }
                    }

                    // Ha a sugár elakadt egy magasponton, akkor a többi kört
nem vizsgáljuk (break).
                    if (not_road == 1)
                        break;

                    // Az adott kör szakaszát elfogadjuk, az adott pont tömb
7. oszlopába 1-es szám kerül, azaz út pont.
                    for (l = 0; arr_3d[k][l][4] <= current_degree && l <
index_array[k]; l++)
                    {
                        if (arr_3d[k][l][4] >= i)
                        {
                            arr_3d[k][l][6] = 1;
                        }
                    }
                }
            }
        }

        // A TOPIC-OK FELTÖLTÉSE

        // Végig iterálunk az összes körön.
        for (i = 0; i < index; i++)
        {
            // Végig iterálunk a köríven található összes ponton.
            for (j = 0; j < index_array[i]; j++)
```

```
            {
                // Nem út pontok hozzáadása a filtered_non_road-hoz,
                // amennyiben a tömb 7. oszlopában 2-es szám van.
                if (arr_3d[i][j][6] == 2)
                {
                    pt.x = arr_3d[i][j][0];
                    pt.y = arr_3d[i][j][1];
                    pt.z = arr_3d[i][j][2];
                    filtered_non_road.push_back(pt);
                }

                // Út pontok hozzáadása a filtered_road-hoz,
                // amennyiben a tömb 7. oszlopában 1-es szám van.
                else if (arr_3d[i][j][6] == 1)
                {
                    pt.x = arr_3d[i][j][0];
                    pt.y = arr_3d[i][j][1];
                    pt.z = arr_3d[i][j][2];
                    filtered_road.push_back(pt);
                }
            }
        }


        // LEGTÁVOLABBI ÚT PONT KERESÉSE ADOTT FOKBAN (MARKER PONTOK
KERESÉSE)

        // Két dimenziós tömb létrehozása. A tömb első három oszlopa
tartalmazza az adott pont X, Y, Z koordinátáját.
        // A negyedik oszlop 0-ás, vagy 1-es értéket vesz fel. Ha az adott
fokban találunk nem út pontot, akkor
        // 1-es értéket kap, ha nem találunk, akkor pedig 0-ás értéket kap.
        float marker_array_points[piece][4];

        // Adott fokban a legtávolabbi út pont (zöld pont) távolságát tárolja
a változó.
        float max_distance_road;

        // A markerek feltöltéséhez szükséges segédváltozó.
        int c = 0;

        // Az adott pont azonosításához szükséges változók. Az id_1 jelenti a
körvonal azonosítását (melyik).
        // Az id_2 jelenti a korvonalon elhelyezkedő pont azonosítását
(hanyadik a körvonalon).
        int id_1, id_2;

        // A vizsgált sávban van-e magaspont, vagy olyan pont, amit nem
jelölt a program se útnak, se magaspontnak.
        int red_points;

        // A for ciklussal megvizsgáljuk a pontokat fokonként (0 foktól
egészen 360 fokig).
        for (i = 0; i <= 360; i++)
        {
            id_1 = -1;
            id_2 = -1;
            max_distance_road = 0;
            red_points = 0;
```

```
            // A for ciklussal végigmegyünk az összes körvonalon.
            for (j = 0; j < index; j++)
            {
                // A for ciklussal végigmegyünk az adott körvonal összes
pontján.
                for (k = 0; k < index_array[j]; k++)
                {
                    // Ha találunk az adott fokban nem út pontot (piros
pont), akkor break utasítással kilépünk,
                    // nem vizsgáljuk tovább, mert nem lesz utána már út
pont.
                    // A red_point változóba 1-es érték kerül.
                    if (arr_3d[j][k][6] != 1 && arr_3d[j][k][4] >= i &&
arr_3d[j][k][4] < i + 1)
                    {
                        red_points = 1;
                        break;
                    }

                    // Ha adott fokban találunk út pontot (zöld pont), akkor
megvizsgáljuk az adott pontnak az origótól vett távolságát.
                    if (arr_3d[j][k][6] == 1 && arr_3d[j][k][4] >= i &&
arr_3d[j][k][4] < i + 1)
                    {
                        d = sqrt(pow(0 - arr_3d[j][k][0], 2) + pow(0 -
arr_3d[j][k][1], 2));

                        // Ha az origótól vett távolsága nagyobb az adott
pontnak, mint az eddig a max_distance_road változóban tárolt érték,
                        // akkor új értékként ez kerül eltárolsára. Emellett
eltároljuk az adott pont azonosítóit is (körvonal, hanyadik pont)
                        if (d > max_distance_road)
                        {
                            max_distance_road = d;
                            id_1 = j;
                            id_2 = k;
                        }
                    }
                }

                // Ezzel a break utasítással nem vizsgáljuk a további
körvonalakat, hanem a következő fok vizsgálatával folytatjuk.
                if (red_points == 1)
                    break;
            }

            // Amennyiben megtaláltuk a legtávolabb lévő út pontot adott
fokban, akkor a tömbhöz hozzáadjuk a pontot.
            // Hozzáadjuk az adott pont X, Y, Z koordinátáját és a red_points
értékét. A c változóval számoljuk a pontok darabszámát.
            if (id_1 != -1 && id_2 != -1)
            {
                marker_array_points[c][0] = arr_3d[id_1][id_2][0];
                marker_array_points[c][1] = arr_3d[id_1][id_2][1];
                marker_array_points[c][2] = arr_3d[id_1][id_2][2];
                marker_array_points[c][3] = red_points;
                c++;
            }
```

```
        }

        // MARKER PONTHALMAZ EGYSZERŰSÍTÉSE LANG ALGORITMUSSAL

        // Egyszerűsített ponthalmaz tárolásához szükséges kétdimenziós tömb.
        float simp_marker_array_points[c][4];

        // Számolja az egyszerűsített pontok számát.
        int count = 0;

        // Vizsgálatok számát figyeli.
        int counter = 0;

        // A for ciklussal hozzáadjuk a tömbhöz az első pont értékeit (X, Y,
Z, 0-ás vagy 1-es)
        for (i = 0; i < 4; i++)
        {
            simp_marker_array_points[0][i] = marker_array_points[0][i];
        }
        // A for ciklussal vizsgáljuk a pontokat.
        for (i = 4; i <= c - 1; i = i + 4)
        {
            // Első vizsgálat (i).
            counter = 0;

            // Kiszámoljuk a három pont merőleges távolságát.
            for (j = 1; j <= 3; j++)
            {
                float d = perpendicular_distance(marker_array_points[i -
4][0], marker_array_points[i - 4][1],
                                                 marker_array_points[i][0],
marker_array_points[i][1],
                                                 marker_array_points[i -
j][0], marker_array_points[i - j][1]);

                // Ha adott pontnál a merőleges távolság nagyobb, mint az
epsilon,
                // akkor nem vizsgálódik tovább, break utasítással kilép.
                if (d > epsilon)
                {
                    counter++;
                    break;
                }

                // Ha kiszámoltuk mind a három pont merőleges távolságát és
                // epsilon mind a három esetben nagyobb volt, akkor az adott
pontot (i) hozzáadjuk az egyszerűsített tömbhöz.
                if (j == 3 && d < epsilon)
                {
                    simp_marker_array_points[count][0] =
marker_array_points[i][0];
                    simp_marker_array_points[count][1] =
marker_array_points[i][1];
                    simp_marker_array_points[count][2] =
marker_array_points[i][2];
                    simp_marker_array_points[count][3] =
marker_array_points[i][3];
                    count++;
                }
```

```
            }
            // Ha már az első merőleges távolság vizsgálatkor d > epsilon,
akkor újabb kettő vizsgálat történik (i - 1).
            if (counter == 1)
            {
                // Második vizsgálat.
                counter = 2;

                // Kiszámoljuk a kettő pont merőleges távolságát.
                for (j = 2; j <= 3; j++)
                {
                    float d = perpendicular_distance(marker_array_points[i -
4][0], marker_array_points[i - 4][1],
                                                     marker_array_points[i -
1][0], marker_array_points[i - 1][1],
                                                     marker_array_points[i -
j][0], marker_array_points[i - j][1]);

                    // Ha adott pontnál a merőleges távolság nagyobb, mint az
epsilon,
                    // akkor nem vizsgálódik tovább, break utasítással kilép.
                    if (d > epsilon)
                    {
                        counter++;
                        break;
                    }

                    // Ha kiszámoltuk mind a kettő pont merőleges távolságát
és
                    // epsilon mind a kettő esetben nagyobb volt, akkor az
adott pontot (i - 1) hozzáadjuk az egyszerűsített tömbhöz.
                    if (j == 3 && d < epsilon)
                    {
                        simp_marker_array_points[count][0] =
marker_array_points[i - 1][0];
                        simp_marker_array_points[count][1] =
marker_array_points[i - 1][1];
                        simp_marker_array_points[count][2] =
marker_array_points[i - 1][2];
                        simp_marker_array_points[count][3] =
marker_array_points[i - 1][3];
                        count++;
                        i = i - 1;
                    }
                }
            }

            // Ha már a második merőleges távolság vizsgálatkor d > epsilon,
akkor újabb vizsgálat történik (i - 2)
            if (counter == 3)
            {
                // Kiszámoljuk a pont merőleges távolságát.
                float d = perpendicular_distance(marker_array_points[i -
4][0], marker_array_points[i - 4][1],
                                                 marker_array_points[i -
2][0], marker_array_points[i - 2][1],
                                                 marker_array_points[i -
3][0], marker_array_points[i - 3][1]);
```

```
                // Ha adott pontnál a merőleges távolság nagyobb, mint az
epsilon,
                // akkor az adott pontot (i - 3) hozzáadjuk az egyszerűsített
tömbhöz.
                if (d > epsilon)
                {
                    simp_marker_array_points[count][0] =
marker_array_points[i - 3][0];
                    simp_marker_array_points[count][1] =
marker_array_points[i - 3][1];
                    simp_marker_array_points[count][2] =
marker_array_points[i - 3][2];
                    simp_marker_array_points[count][3] =
marker_array_points[i - 3][3];
                    count++;
                    i = i - 3;
                }


                // Ha adott pontnál a merőleges távolság kisebb, mint az
epsilon,
                // akkor az adott pontot (i - 2) hozzáadjuk az egyszerűsített
tömbhöz.
                if (d < epsilon)
                {
                    simp_marker_array_points[count][0] =
marker_array_points[i - 2][0];
                    simp_marker_array_points[count][1] =
marker_array_points[i - 2][1];
                    simp_marker_array_points[count][2] =
marker_array_points[i - 2][2];
                    simp_marker_array_points[count][3] =
marker_array_points[i - 2][3];
                    count++;
                    i = i - 2;
                }
            }
        }

        // A for ciklussal hozzáadjuk a tömbhöz az utolsó pont értékeit (X,
Y, Z, 0-ás vagy 1-es)
        for (i = 0; i < 4; i++)
        {
            simp_marker_array_points[count][i] = marker_array_points[c -
1][i];
        }

        // std::cout << count << std::endl;

        // MARKER ÖSSZEÁLLÍTÁSA

        // A program megvizsgálja, hogy van-e 3 darab marker pont, amit össze
lehet kötni.
        // Ha nincs, akkor nem hajtja végre az if feltétel alatt lévő
utasításokat.
        if (count > 2)
        {
            // Amennyiben a simp_marker_array_points tömbnek a 4. oszlopában
1-es érték szerepel, akkor az a piros line_segment-hez tartozik.
```

```
            // Ha 0-ás érték szerepel, akkor pedig a zöld line_segment-hez
tartozik.

            // Az első és az utolsó pontokat beállítjuk, hogy ne legyen zöld-
piros-zöld vagy piros-zöld-piros eset.

            // Amennyiben az első pont zöld és a második piros, akkor az első
is piros line_segment-be kerüljön.
            if (simp_marker_array_points[0][3] = 0 &&
simp_marker_array_points[1][3] == 1)
                simp_marker_array_points[0][3] = 1;

            // Amennyiben az utolsó pont zöld és az utolsó előtti piros,
akkor az utolsó is piros line_segment-be kerüljön.
            if (simp_marker_array_points[count - 1][3] == 0 &&
simp_marker_array_points[count - 2][3] == 1)
                simp_marker_array_points[count - 1][3] == 1;

            // Amennyiben az első pont piros és a második zöld, akkor az első
is zöld line_segment-be kerüljön.
            if (simp_marker_array_points[0][3] = 1 &&
simp_marker_array_points[1][3] == 0)
                simp_marker_array_points[0][3] = 0;

            // Amennyiben az utolsó pont piros és az utolsó előtti zöld,
akkor az utolsó is zöld line_segment-be kerüljön.
            if (simp_marker_array_points[count - 1][3] == 1 &&
simp_marker_array_points[count - 2][3] == 0)
                simp_marker_array_points[count - 1][3] = 0;

            // Egy for ciklussal végigmegyünk a pontokon. Amennyiben egy zöld
pontot közrefog két piros pont,
            // akkkor a zöld pont is piros line_segment-be kerül.
            // Itt fontos, hogy az első kettő és az utolsó kettő pontot nem
kell vizsgálni, hiszen ezeket már az előzőekben beállítottuk.
            for (i = 2; i <= count - 3; i++)
            {
                if (simp_marker_array_points[i][3] == 0 &&
simp_marker_array_points[i - 1][3] == 1 && simp_marker_array_points[i + 1][3]
== 1)
                    simp_marker_array_points[i][3] = 1;
            }

            // Egy for ciklussal végigmegyünk ismét a pontokon. Amennyiben
egy piros pontot közrefog két zöld pont,
            // akkkor a piros pont is zöld line_segment-be kerül.
            // Itt fontos, hogy az első kettő és az utolsó kettő pontot nem
kell vizsgálni, hiszen ezeket már az előzőekben beállítottuk.
            for (i = 2; i <= count - 3; i++)
            {
                if (simp_marker_array_points[i][3] == 1 &&
simp_marker_array_points[i - 1][3] == 0 && simp_marker_array_points[i + 1][3]
== 0)
                    simp_marker_array_points[i][3] = 0;
            }

            // Egy marker array objektum, amiben eltároljuk a zöld és piros
line_segment-eket.
            visualization_msgs::MarkerArray marker_arr;
```

```
            // Egy marker objektum, amiben az adott zöld, vagy piros szakaszt
(line strip) tároljuk.
            visualization_msgs::Marker line_segment;

            // Egy point objektum, ami az adott pont értékét tárolja. Ezzel
töltjük fel az adott line_segment-et.
            geometry_msgs::Point point;

            // Tárolja az adott line_segment ID-jét (azonosítóját).
            int line_segment_id = 0;

            // Beállítjuk a hozzáadáshoz szükséges parméterket a
line_segment-nek.
            line_segment.header.frame_id = fixed_frame;
            line_segment.header.stamp = ros::Time();
            line_segment.type = visualization_msgs::Marker::LINE_STRIP;
            line_segment.action = visualization_msgs::Marker::ADD;

            // Végigmegyünk az összes ponton, amik majd a markert fogják
alkotni.
            for (i = 0; i < count; i++)
            {
                // Hozzáadjuk az adott pontot a geometry_msgs::Point típusú
objektumhoz.
                point.x = simp_marker_array_points[i][0];
                point.y = simp_marker_array_points[i][1];
                point.z = simp_marker_array_points[i][2];

                // Az első pont hozzáadása az adott line_segment-hez. Itt
semmilyen feltételt nem kell megadni.
                if (i == 0)
                {
                    line_segment.points.push_back(point);
                }

                // Amennyiben a következő pont is ugyanabba a csoportba fog
tartozni, mint az előző,
                // akkor ezt is hozzá fogjuk adni az adott line_segment-hez.
                else if (simp_marker_array_points[i][3] ==
simp_marker_array_points[i - 1][3])
                {
                    line_segment.points.push_back(point);

                    // Az utolsó pont vizsgálata, itt készül el az utolsó
line_segment.
                    if (i == count - 1)
                    {
                        line_segment.id = line_segment_id;

                        // Line_segment beállítások
                        line_segment.pose.position.x = 0;
                        line_segment.pose.position.y = 0;
                        line_segment.pose.position.z = 0;

                        line_segment.pose.orientation.x = 0.0;
                        line_segment.pose.orientation.y = 0.0;
                        line_segment.pose.orientation.z = 0.0;
                        line_segment.pose.orientation.w = 1.0;
```

```
                        line_segment.scale.x = 0.5;
                        line_segment.scale.y = 0.5;
                        line_segment.scale.z = 0.5;

                        // Itt beállítjuk a line_segment színét.
                        // Ha a 4. oszlopban az érték 0, akkor zöld színű a
line_segment.
                        if (simp_marker_array_points[i][3] == 0)
                        {
                            line_segment.color.a = 1.0;
                            line_segment.color.r = 0.0;
                            line_segment.color.g = 1.0;
                            line_segment.color.b = 0.0;
                        }

                        // Egyébként piros színű a line_segment.
                        else
                        {
                            line_segment.color.a = 1.0;
                            line_segment.color.r = 1.0;
                            line_segment.color.g = 0.0;
                            line_segment.color.b = 0.0;
                        }

                        // Hozzáadjuk a line_segment-et a marker array-hez.
                        marker_arr.markers.push_back(line_segment);

                        // Kitöröljük a pontokat az utolsó line_segment-ből,
feleslegesen ne tároljuk.
                        line_segment.points.clear();
                    }
                }

                // Csoportváltozás történik, pirosról zöldre fog váltani.
                // Ebben az esetben még a két pontot piros marker fogja
összekötni, így hozzáadjuk a pontot az adott line_segment-hez.
                else if (simp_marker_array_points[i][3] !=
simp_marker_array_points[i - 1][3] && simp_marker_array_points[i][3] == 0)
                {
                    line_segment.points.push_back(point);

                    // A következő pontok már új line_segment-hez fognak
tartozni, elkészül az egyik piros.
                    line_segment.id = line_segment_id;
                    line_segment_id++;

                    // Line_segment beállítások.
                    line_segment.pose.position.x = 0;
                    line_segment.pose.position.y = 0;
                    line_segment.pose.position.z = 0;

                    line_segment.pose.orientation.x = 0.0;
                    line_segment.pose.orientation.y = 0.0;
                    line_segment.pose.orientation.z = 0.0;
                    line_segment.pose.orientation.w = 1.0;

                    line_segment.scale.x = 0.5;
                    line_segment.scale.y = 0.5;
```

```
                    line_segment.scale.z = 0.5;

                    line_segment.color.a = 1.0;
                    line_segment.color.r = 1.0;
                    line_segment.color.g = 0.0;
                    line_segment.color.b = 0.0;

                    // Hozzáadjuk a line_segment-et a marker array-hez.
                    marker_arr.markers.push_back(line_segment);

                    // Kitöröljük a pontokat a line_segment-ből, feleslegesen
ne tároljuk.
                    line_segment.points.clear();

                    // A következő zöld line_segment-nél erre a pontra
szükség van, ezért hozzáadjuk.
                    line_segment.points.push_back(point);
                }

                // Csoportváltozás történik, zöldról pirosra fog váltani.
                // Először beállítjuk a zöld line_segment-et, majd hozzáadjuk
az utolsó pontot a piroshoz is.
                // Zöld és piros pont között mindig piros line_segment van.
                else if (simp_marker_array_points[i][3] !=
simp_marker_array_points[i - 1][3] && simp_marker_array_points[i][3] == 1)
                {
                    line_segment.points.push_back(point);

                    // Zöld marker
                    line_segment.id = line_segment_id;
                    line_segment_id++;

                     // Line_segment beállítások
                    line_segment.pose.position.x = 0;
                    line_segment.pose.position.y = 0;
                    line_segment.pose.position.z = 0;

                    line_segment.pose.orientation.x = 0.0;
                    line_segment.pose.orientation.y = 0.0;
                    line_segment.pose.orientation.z = 0.0;
                    line_segment.pose.orientation.w = 1.0;

                    line_segment.scale.x = 0.5;
                    line_segment.scale.y = 0.5;
                    line_segment.scale.z = 0.5;

                    line_segment.color.a = 1.0;
                    line_segment.color.r = 0.0;
                    line_segment.color.g = 1.0;
                    line_segment.color.b = 0.0;

                    // Hozzáadjuk a line_segment-et a marker array-hez.
                    marker_arr.markers.push_back(line_segment);

                    // Kitöröljük a pontokat a line_segment-ből, feleslegesen
ne tároljuk.
                    line_segment.points.clear();
```

```
                    // A következő piros line_segment-hez szügség van az
előző pontra is.
                    point.x = simp_marker_array_points[i - 1][0];
                    point.y = simp_marker_array_points[i - 1][1];
                    point.z = simp_marker_array_points[i - 1][2];
                    line_segment.points.push_back(point);

                    // A következő piros line_segment-hez szükség van a
jelenlegi pontra.
                    point.x = simp_marker_array_points[i][0];
                    point.y = simp_marker_array_points[i][1];
                    point.z = simp_marker_array_points[i][2];
                    line_segment.points.push_back(point);
                }

                // A line_segment élettartamának idejét beállítjuk.
                line_segment.lifetime = ros::Duration(0.04);
            }

            // A marker array közzététele.
            pub_marker_array.publish(marker_arr);
        }

        // 3D DINAMIKUS TÖMB FELSZABADÍTÁSA

        for (i = 0; i < channels; i++)
        {
            for (j = 0; j < piece; j++)
            {
                delete [] arr_3d[i][j];
            }
            delete [] arr_3d[i];
        }

        delete[] arr_3d;
    }

    // A topic-ok header-je és a topic-ok publikálása.
    // Vizsgált terület pontjainak publikálása.
    filtered_frame.header = msg.header;
    pub_frame.publish(filtered_frame);

    // Nem út pont publikálása.
    filtered_non_road.header = msg.header;
    pub_non_road.publish(filtered_non_road);

    // Út pontok publikálása.
    filtered_road.header = msg.header;
    pub_road.publish(filtered_road);
}

int main(int argc, char **argv)
{
    // ROS inicializálása.
    ros::init(argc, argv, "lidarFilter");

    // Paraméter ablak beállításához szükség sorok (dynamic reconfigure).
    dynamic_reconfigure::Server<lidar_filter::dynamic_reconfConfig> server;
```

```
dynamic_reconfigure::Server<lidar_filter::dynamic_reconfConfig>::CallbackType
f;

    f = boost::bind(&params_callback, _1, _2);
    server.setCallback(f);

    // Node létrehozása.
    ros::NodeHandle node;

    // Felíratkozás egy adott topic-ra.
    ros::Subscriber sub = node.subscribe(topic_name, 1, filter);

    // A filterezett adatok közzététele.
    pub_frame = node.advertise<pcl::PCLPointCloud2>("filtered_frame", 1);
    pub_non_road = node.advertise<pcl::PCLPointCloud2>("filtered_non_road",
1);
    pub_road = node.advertise<pcl::PCLPointCloud2>("filtered_road", 1);
    pub_marker_array =
node.advertise<visualization_msgs::MarkerArray>("marker_array", 1);

    ros::spin();

    return 0;
}
```

2. Dynamic reconfigure forráskódja

```
#!/usr/bin/env python
PACKAGE = "lidar_filter"

from dynamic_reconfigure.parameter_generator_catkin import *
gen = ParameterGenerator()

gen.add("fixed_frame", str_t, 0, "Ha megvaltozik az erteke, akkor ujrainditas
szukseges.", "left_os1/os1_lidar")

gen.add("topic_name", str_t, 0, "Topic neve, amire fel akarunk iratkozni. Ha
megvaltozik az erteke, akkor ujrainditas szukseges.",
"/left_os1/os1_cloud_node/points")

gen.add("epsilon", double_t, 0, "Meroleges tavolsagmereskor a kuszobertek.",
0.3, 0.0, 1.0)

gen.add("min_x", double_t, 0, "Vizsgalt terulet X (min) koordinataja
meterben.", 0, -50, 50)

gen.add("max_x", double_t, 0, "Vizsgalt terulet X (max) koordinataja
meterben.", 30, -50, 50)

gen.add("min_y", double_t, 0, "Vizsgalt terulet Y (min) koordinataja
meterben.", -10, -30, 30)

gen.add("max_y", double_t, 0, "Vizsgalt terulet Y (max) koordinataja
meterben.", 10, -30, 30)

gen.add("min_z", double_t, 0, "Vizsgalt terulet Z (min) koordinataja
meterben.", -3, -10, 10)

gen.add("max_z", double_t, 0, "Vizsgalt terulet Z (max) koordinataja
meterben.", -1, -10, 10)

gen.add("interval", double_t, 0, "LIDAR vertikalis szogfelbontasanak
intervalluma.", 0.1800, 0.0100, 10)

gen.add("curb_points", int_t, 0, "Jardaszegelyen a pontok becsult szama.", 5,
1, 30)

gen.add("curb_height", double_t, 0, "Jardaszegely becsult minimalis magassaga
meterben.", 0.0500, 0.0100, 0.5000)

gen.add("angle_filter1", double_t, 0, "Harom pont altal bezart szog
vizsgalata X = 0 ertek mellett.", 150, 0, 180)

gen.add("angle_filter2", double_t, 0, "Ket vektor altal bezart szog
vizsgalata Z = 0 ertek mellett.", 140, 0, 180)

gen.add("beam_zone", double_t, 0, "A vizsgalt sugarzona fokban.", 30, 10,
100)

exit(gen.generate(PACKAGE, "lidar_filter", "dynamic_reconf"))
```