



EÖTVÖS LORÁND TUDOMÁNYEGYETEM

INFORMATIKAI KAR

ALGORITMUSOK ÉS ALKALMAZÁSAIK TANSZÉK

SLAM algoritmus implementációja

Szerző:

Virágh Ákos

programtervező informatikus BSc

Belső témavezető:

Eichhardt Iván, PhD

Adjunktus

ELTE IK - ALAL

Külső témavezető:

Fazekas Máté

PhD hallgató

BME KJK - KJIT

Budapest, 2022

Köszönetnyilvánítás

Ezúton szeretném megköszönni mindenazonknak, akik nélkül ez a dolgozat nem jöhetett volna létre. Először is hatalmas köszönet mindenkit konzulensemnek, *Fazekas Máténak* és *Eichhardt Ivánnak*, akik rendelkezésemre bocsátották tudásukat, hasznos tanácsokkal és rengeteg segítséggel láttak el a kutatás során.

Szintén hálával tartozom az Eötvös Loránd Tudományegyetem Informatikai Karának, hogy az itt eltöltött félévek alatt elsajátíthattam a kutatáshoz elengedhetetlenül szükséges matematikai és informatikai alapokat.

Továbbá szeretném megköszönni a *BME Formula Racing Team*-nek, hogy különböző erőforrásokat biztosítottak számomra, illetve hogy a köreikben végezhettem a kutatásom zömét.

Végül, de nem utolsósorban szeretnék köszönetet mondani a családomnak és barátaimnak, hogy minden támogatást megadtak, amire a tanulmányaim során szükségem volt.

Kivonat

A járműipar egyre korszerűbb vezetőtámogató rendszereket fejleszt, ezzel folyamatos haladást mutatva a teljesen önvezető járművek felé. Az önvezető rendszerek egyik, ha nem a legfontosabb része a megbízható helyzetbecslés és a környezet felterképezése.

A feladat a mobil robotika egy közismert problémája, melynek elterjedt megoldása az úgynevezett „Egyidejű lokalizáció és helymeghatározás” (Simultaneous Localization and Mapping, SLAM) módszer. A módszer nehézsége, hogy a térképkészítéshez szükségünk van egy precíz helyzetbecslésre, viszont a robot helyzetének pontos meghatározásához elengedhetetlen egy térkép. A SLAM során a robot iteratívan becsli a mozgását, eközben leképezi a környezetérzékelő szenzorból származó objektumokat, amelyek az algoritmus bemenetéül szolgálnak. A bemenet alapján a robot térképet készít a környezetéről és ezzel párhuzamosan elhelyezi magát a folyamatosan készülő térképen.

Abstract

The automotive industry is developing more and more advanced driver assistance systems, thus showing continuous progress towards fully self-driving vehicles. One, if not the most important part of self-driving systems is reliable pose estimation and environmental mapping.

The task is a well-known problem in mobile robotics, a common solution to which is the so-called „Simultaneous Localization and Mapping” (SLAM) method. The difficulty with this method is that we need a precise position estimate to make the map, but a map is essential to determine the exact position of the robot. During SLAM, the robot estimates its motion iteratively, while mapping the objects from the environment sensor that serve as input to the algorithm. Based on the input, the robot creates a map of its surroundings and at the same time places itself on the map that is being created continuously.

Tartalomjegyzék

Köszönetnyilvánítás	ii
Kivonat	iii
Abstract	iv
1. Bevezetés	1
1.1. A SLAM probléma	1
1.2. Online SLAM	1
1.3. Full SLAM	2
2. Felhasználói dokumentáció	4
2.1. Szimulációs környezet	4
2.1.1. A szoftver célja	4
2.1.2. A szoftver használata	4
2.2. SLAM	11
2.2.1. A szoftver célja	11
2.2.2. A szoftver használata	11
3. Elméleti háttér	20
3.1. Definíciók és tételek	20
3.2. Gauss-Newton módszer	23
3.3. Mozgási modell	24
3.4. Hurokzárás	25
4. Fejlesztői dokumentáció	26
4.1. Szimulációs környezet	26
4.1.1. A szoftver felépítése	26
4.1.2. Továbbfejlesztési lehetőségek	29

TARTALOMJEGYZÉK

4.2. SLAM	30
4.2.1. A szoftver felépítése	30
4.2.2. Fordítási útmutató	42
4.2.3. Továbbfejlesztési lehetőségek	43
4.3. A fejlesztés főbb mérföldkövei	44
4.3.1. Programozástechnikai megfontolások	46
4.3.2. Kompromisszumok	46
5. Tesztelés	47
5.1. Fehérdobozos tesztelés	47
5.1.1. Numerikus eljárásokhoz szükséges osztályok tesztelése	47
5.1.2. DataEnumerator osztály tesztelése	50
5.1.3. Graph osztály tesztelése	51
5.2. Kiértékelés	52
5.2.1. Zaj nélkül	52
5.2.2. Zajjal	62
5.2.3. Összegzés	82
6. Alkalmazás	83
6.1. Önvezető járművek általában	83
6.2. BME Formula Racing Team	84
7. Összegzés	86
Irodalomjegyzék	87
Ábrajegyzék	89
Táblázatjegyzék	92

1. fejezet

Bevezetés

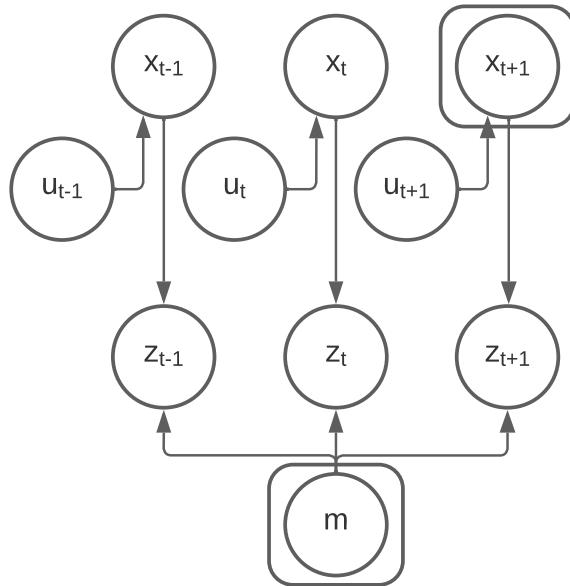
1.1. A SLAM probléma

Az „Egyidejű lokalizáció és helymeghatározás” (Simultaneous Localization and Mapping, SLAM) [1] probléma azt a kérdést taglalja, hogy lehetséges-e elhelyezni egy mobilis robotot egy ismeretlen környezeten belül egy ismeretlen helyre, hogy aztán a robot iteratívan felépítse egy konzisztens térképet a környezetéről, miközben egyidejűleg el is helyezze magát a készülő térképen. A SLAM probléma megoldására a robotikával foglalkozó közösség a robotika „szent gráljaként” tekintenek, ugyanis ez lehetőséget biztosítana egy robot valóban autonómmá tételere.

A SLAM probléma „megoldása” a robotikával foglalkozó közösség egyik figyelemre méltó sikere volt az elmúlt évtizedben. Elméleti problémaként fogalmazták és oldották meg számos különböző formában, de általában valószínűségszámítási problémaként szokták meghatározni. Különböző területeken kezdték el használni, beltéri, vagy akár kültéri robotknál, nem csupán a felszínen, de víz alatt és a levegőben is. Mindazonáltal jelentős problémák maradtak a gyakorlati megvalósításában, különösen a komplexebb, részletesebb környezetek feltérképezésénél.

1.2. Online SLAM

A SLAM ezen megközelítésében [2] a térkép mellett csupán az aktuális pozíció van becsülve, a legfrissebb szenzoradatokból.

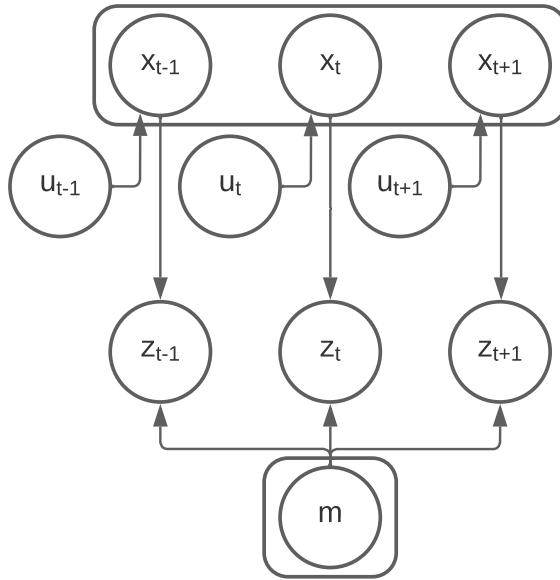


1.1. ábra. A t -dik időpillanatban véve x jelöli a járműpózt, u jelöli az odometriai mérést, z jelöli az érzékelésből adódó mérést és m jelöli a felépített térképet.

Minden számítás rekurzív módon történik, és számos algoritmust használnak a korábbi mérések frissítésére hurokzárás vagy más jelentős események miatt.

1.3. Full SLAM

A SLAM probléma ezen megközelítésében az a cél az „online SLAM”-mel ellenétben, hogy a mobilis robotunk teljes útvonalát és a környezetéről alkotott térképet megbecsüljük a rendelkezésünkre álló adatok alapján.



1.2. ábra. A t -dik időpillanatban véve x jelöli a járműpózt, u jelöli az odometriai mérést, z jelöli az érzékelésből adódó mérést és m jelöli a felépített térképet.

A „full SLAM” [3] probléma valós időben nehezen kezelhető, mivel az optimalizációs probléma komplexitása minden egyes újabb adattal növekedik.

Dolgozatom keretében egy ilyen „full SLAM” típusú, gráf alapú megvalósítást készítettem el.

2. fejezet

Felhasználói dokumentáció

Az alábbiakban bemutatom a szoftverek használatához szükséges legfontosabb tudnivalókat.

2.1. Szimulációs környezet

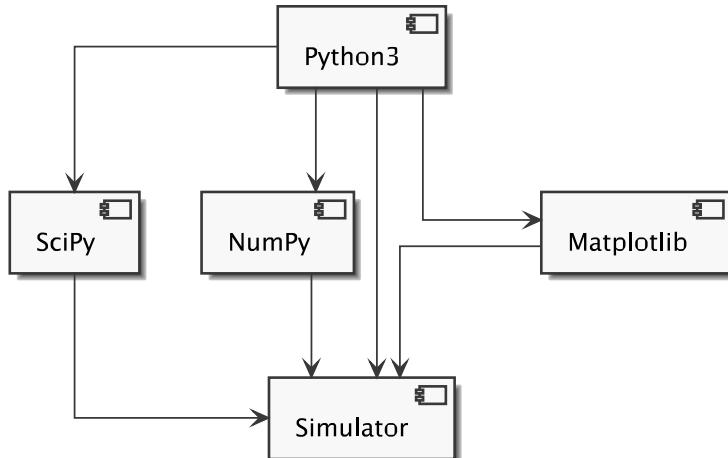
2.1.1. A szoftver célja

A szoftver egy olyan szimulációs környezetet biztosít, amelyből egy adott pályán haladva az aktuális pillanatban érzékelt objektumok távolságadatai lekérdezhetők lokális koordináta-rendszerben, és a jármű sebessége is rendelkezésre áll.

2.1.2. A szoftver használata

Rendszerkövetelmények

A program bármilyen operációs rendszeren használható, amely képes Python 3.8 [4]-at futtatni. A program futtatásához szükséges Python függőségek a következők: *Matplotlib* [5], *NumPy* [6], *SciPy* [7].



2.1. ábra. Szimulációs környezet függőségi diagramja

	Várható működés	Garantált működés
Operációs rendszer	Unix alapú operációs rendszerek	macOS 11.6.5
Processzor	1 GHz, vagy gyorsabb	Intel Core i7 - 2.6 Ghz
RAM	2 GB	32 GB
Szabad lemezterület	100 MB	100 MB
Videokártya	Integrált	Intel UHD Graphics 630
Képernyő	800 x 600	3072 x 1920

2.1. táblázat. Rendszerkövetelmények

Telepítési útmutató

Python3 - A fentebb említett operációs rendszerek közül mind Linux-on, mind macOS-en használhatjuk a preferált csomagkezelőnkét, a hivatalos weboldal¹ által biztosított telepítőket, vagy akár magunk is lefordíthatjuk a forráskódot.

Függőségek - A függőségek telepítéséhez a Python egy out-of-the-box megoldása a *pip*² csomagkezelő, mellyel ha még nem rendelkezünk, akkor vagy az operációs rendszerünk csomagkezelőjét használva, vagy a hivatalos weboldalon³ található leírás szerint tudjuk feltelepíteni.

Matplotlib

```
python3 -m pip install --upgrade matplotlib
```

¹<https://www.python.org/downloads/>

²<https://pip.pypa.io/>

³<https://pip.pypa.io/en/stable/installation/>

NumPy

```
python3 -m pip install --upgrade numpy
```

SciPy

```
python3 -m pip install --upgrade scipy
```

Szimulációs környezet - a szoftver a dolgozathoz mellékelt, vagy a git repo⁴-ból letölthető zip archívum kicsomagolása után és a függőségek telepítése után már használható.

Futtatás

A szoftver a kicsomagolt archívum *simulator* mappájában található, konzolban futtatható Python script, amely a következő parancssal indítható el:

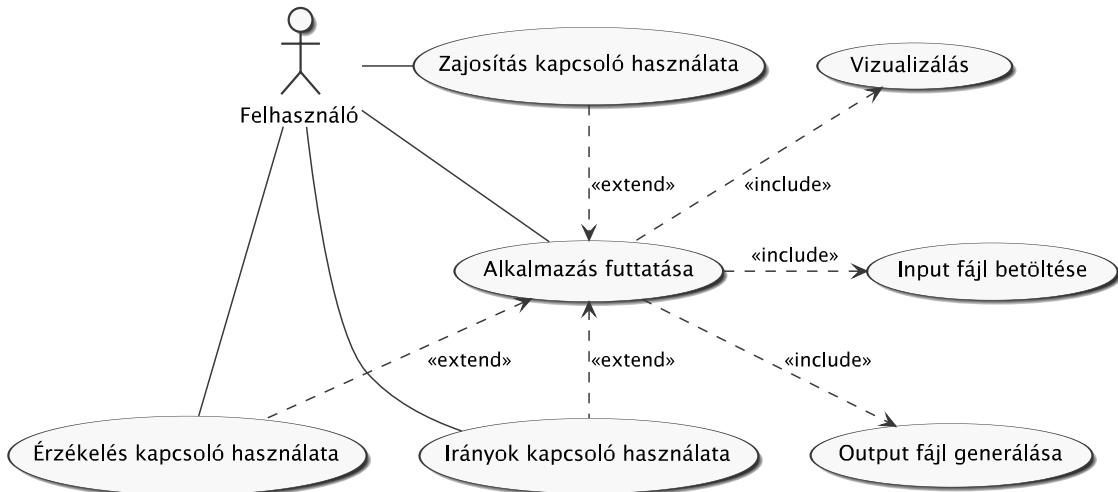
```
python3 simulator.py [-h] [-d] [-n] [-p] input_file
```

Kapcsoló	Hatás
-h, --help	A konzolra írja a program helyes használatának menetét
-d, --directions	Vizualizálja az autó irányát minden időpillanatban
-n, --noisy	Normális eloszlású zajt használ a kimeneten
-p, --perception	Vizualizálja, hogy az autó mely bójákat látja minden időpillanatban

2.2. táblázat. Kapcsolók és hatásuk

A Use case diagram a 2.2. ábrán látható.

⁴<https://github.com/laFette21/thesis-bsc>



2.2. ábra. Szimulációs környezet Use Case diagramja

Bemenet

A program bemenete egy csv formátumú szöveges fájl, amely tetszőleges számú sorból áll. minden sorban két lebegőpontos szám található, amelyek a fájl formátumából adódóan vesszővel vannak elválasztva és egy-egy x és y koordinátát reprezentálnak egy globális koordináta-rendszerben.

Bemenet formátuma:

$$x_0, y_0 \quad (2.1)$$

$$x_1, y_1$$

.

.

$$x_n, y_n$$

Kimenet

A program kimenete egy szöveges fájl, amely terjedelme a bemenettől függ. A kimenet pontos formátuma a program paramétere alapján változik, de még így is két különböző eset merülhet fel:

- Az adott időpillanatban tötenik érzékelés - ebben az esetben a kimenet ehhez az időbélyeghez tartozó része a következőképpen néz ki:

o <időbélyeg> <sebesség> <szögsebesség> (2.2)
p <időbélyeg> <távolság> <szög> <szín> <azonosító> <x> <y>
p <időbélyeg> <távolság> <szög> <szín> <azonosító> <x> <y>
·
·
·
p <időbélyeg> <távolság> <szög> <szín> <azonosító> <x> <y>

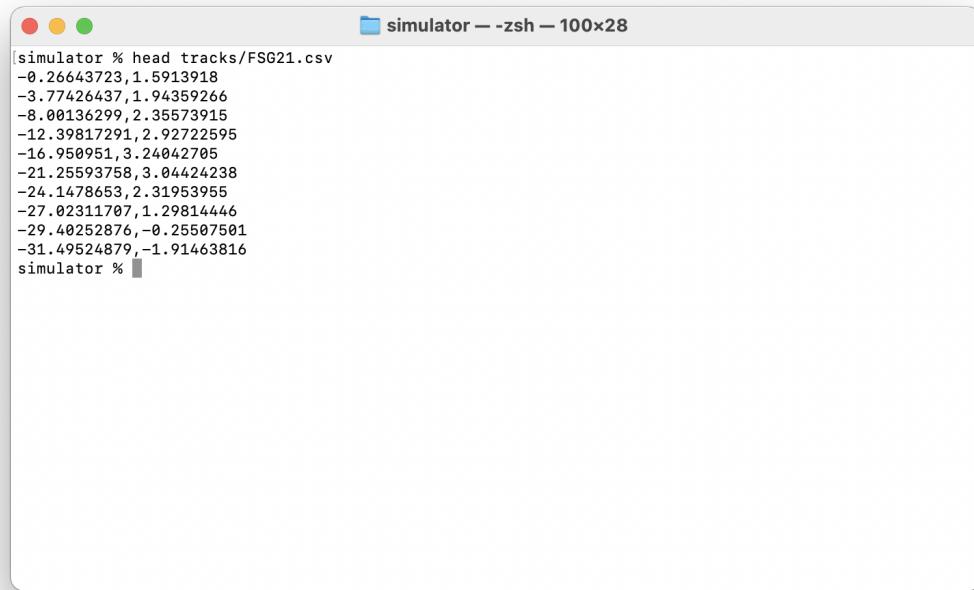
Az első sorban az odometriai mérés található, amely magába foglalja a szimulált járműünk aktuális *sebességét* és *szögsebességét*. A maradék, tetszőleges mennyiségű, de legalább egy sorban pedig a jármű érzékelési szenzora alapján kapott adatok találhatók. Ezek az adatok nem más, mint a *távolság* és *szög* egy adott *színű* bója irányába, ezen bója a programunk által generált *azonosítója* és az *x* és *y* pozíciója egy globális koordináta-rendszerben.

- Az adott időpillanatban nem töténik érzékelés - ebben az esetben a kimenet a következőképpen néz ki:

o <időbélyeg> <sebesség> <szögsebesség> (2.3)
o <időbélyeg> <sebesség> <szögsebesség>
·
·
·
o <időbélyeg> <sebesség> <szögsebesség>

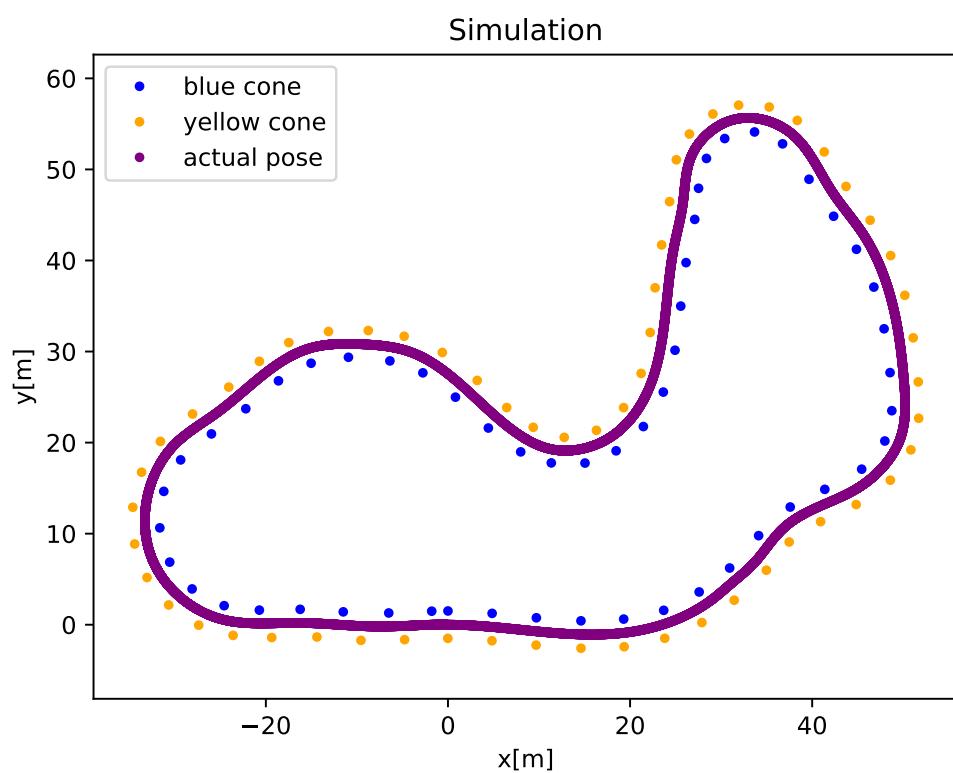
A kimenet ezen részén tetszőleges mennyiségű sorban a szimulált járműünk aktuális *sebességét* és *szögsebességét* láthatjuk.

Példák

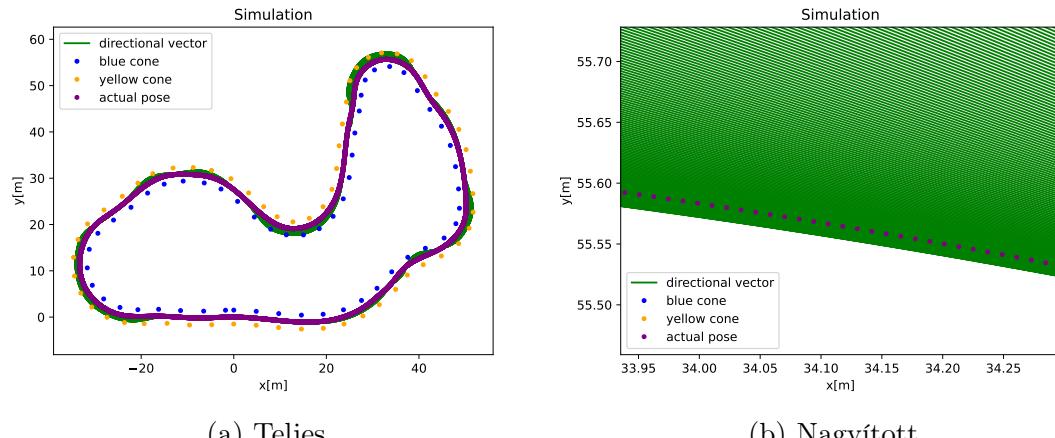


```
simulator % head tracks/FSG21.csv
-0.26643723,1.5913918
-3.77426437,1.94359266
-8.00136299,2.35573915
-12.39817291,2.92722595
-16.950951,3.24042705
-21.25593758,3.04424238
-24.1478653,3.31953955
-27.02311707,1.29814446
-29.40252876,-0.25507581
-31.49524879,-1.91463816
simulator %
```

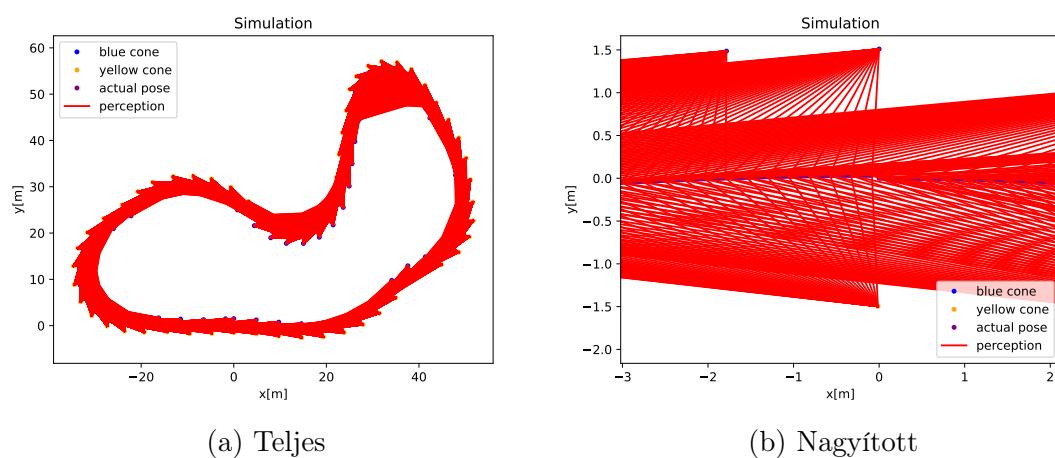
2.3. ábra. Bemeneti fájl részlete a szimulációs környezet FSG21 szimulációjához



2.4. ábra. Szimulációs környezet FSG21 szimulációja



2.5. ábra. Szimulációs környezet FSG21 szimuláció irányvektorai



2.6. ábra. Szimulációs környezet FSG21 szimuláció érzékelése

```

simulator % head -25 output_FSG21.txt
o 0.0 4.892791689046624 0.16081726319947692
p 0.0 5.0072211074047805 0.25160588876577195 1 413 4.849563089118597 1.2465957897804907
p 0.0 9.73882038326381 0.07823539274332628 1 827 9.709031010088323 0.7611434179036025
p 0.0 14.612456720545497 0.029353711024587066 1 1245 14.606161831390969 0.428868237379371
p 0.0 5.141890576300654 -0.34884297486702287 2 20382 4.832188467059364 -1.7575532189575216
p 0.0 9.932769613925936 -0.22731716058762827 2 20763 9.677244321210004 -2.238493813022296
p 0.0 14.835692469351988 -0.1750063566251396 2 21152 14.689083678051014 -2.5831076502709585
o 0.005 4.892928566241874 -0.12820726476010577
o 0.01 4.893066856333973 -0.12780128941710014
o 0.015 4.893206549396041 -0.1273952925520629
o 0.02 4.893347599557253 -0.1269892758081566
o 0.025 4.8934900150070115 -0.12658324086737993
o 0.03 4.893633767994083 -0.1261771893334174
o 0.035 4.893778839827228 -0.12577112282284292
o 0.04 4.893925211875494 -0.12536504299461448
o 0.045 4.89407286556605 -0.12495895136212208
o 0.05 4.8942217823871825 -0.12455284955417856
p 0.05 4.770840367280153 0.27085114359610124 1 413 4.849563089118597 1.2465957897804907
p 0.05 9.494973675317341 0.08665000281722843 1 827 9.709031010088323 0.7611434179036025
p 0.05 14.36791041593422 0.03622659174169563 1 1245 14.606161831390969 0.428868237379371
p 0.05 4.912384960525215 -0.3594010390946949 2 20382 4.832188467059364 -1.7575532189575216
p 0.05 9.694364711447731 -0.2266086518396767 2 20763 9.677244321210004 -2.238493813022296
p 0.05 14.594677720075376 -0.1715390587819355 2 21152 14.609083678051014 -2.5831076502709585
o 0.055 4.894371943886958 -0.1241467390836179
o 0.06 4.894523331672553 -0.123740621538654
simulator %

```

2.7. ábra. Kimeneti fájl részlete a szimulációs környezet FSG21 szimulációjához

2.2. SLAM

2.2.1. A szoftver célja

A szoftver egy gráf alapú SLAM algoritmust valósít meg, amely az aktuálisan beérkező sebesség- és érzékelési adatok alapján térképet készít, és becslést ad a jármű mozgására.

2.2.2. A szoftver használata

Rendszerkövetelmények

A szoftver a lentebb leírt operációs rendszereken vizualizáció nélkül gond nélküл használható. Vizualizáció használatához azonban szükségünk van a *Matplotlib* vizualizációs Python csomag feltelepítésére is.

	Várható működés	Garantált működés
Operációs rendszer	Unix alapú operációs rendszerek	macOS 11.6.5
Processzor	1 GHz, vagy gyorsabb	Intel Core i7 - 2.6 Ghz
RAM	2 GB	32 GB
Szabad lemezterület	500 MB	500 MB
Videokártya	-	Intel UHD Graphics 630
Képernyő	-	3072 x 1920

2.3. táblázat. A SLAM szoftver rendszerkövetelményei

Telepítési útmutató

Python3 - Linux-on, illetve macOS-en használhatjuk a preferált csomagkezelőket, a hivatalos weboldal által biztosított telepítőket, vagy akár magunk is lefordíthatjuk a forráskódot.

Megjegyzés. Python 3.8-as verzió minimális elvárás.

Matplotlib - telepítéshez használhatjuk a Python jól ismert csomagkezelőjét, a *pip*-et, mellyel ha még nem rendelkezünk, akkor vagy az operációs rendszerünk csomagkezelőjét használva, vagy a hivatalos weboldalon található leírás szerint tudjuk feltelepíteni.

A csomagkezelő biztosítása után pedig ezzel a parancssal tudjuk feltelepíteni a függőségünket:

```
python3 -m pip install --upgrade matplotlib
```

Megjegyzés. Unix alapú operációs rendszereken szükséges lehet a *gfortran* fordító feltelepítése is.

SLAM - a szoftver a dolgozathoz mellékelt, vagy a git repo-ból letölthető zip archívum kicsomagolása után és opcionálisan a függőségek telepítését követően már használható.

Futtatás

A szoftver a kicsomagolt archívum *slam_cpp/bin* útvonalon elérhető *linux* és *macos* mappákban található archívumok kicsomagolása után, egy jelenleg csak Intel

architektúrán tesztelt, konzolban futtatható bináris állományként lesz megtalálható, amely a következő parancssal indítható el:

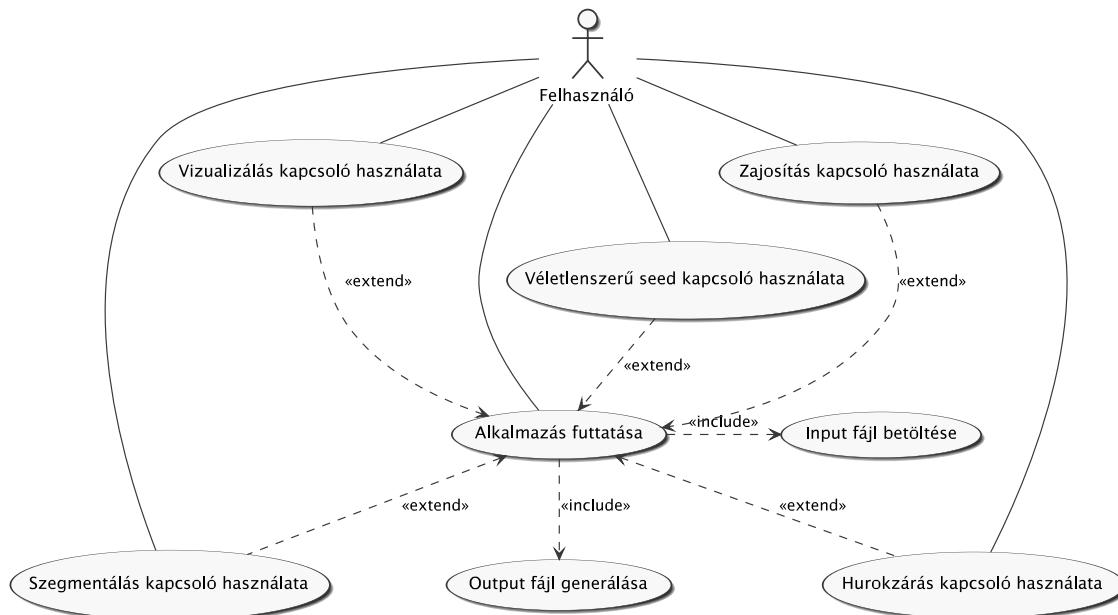
Unix alapú operációs rendszerek

```
./slam [-h] [-v] [-l] [-n] [-p] [-r] [-s] input_file output_file
```

Kapcsoló	Hatás
-h, --help	A konzolra írja a program helyes használatának menetét, majd kilép
-v, --version	A konzolra írja a program verzióját, majd kilép
-l, --loop_closure	Engedélyezi a hurokzárást az optimalizáláshoz
-n, --noise	Normális eloszlású zajt használ a bemeneten
-p, --plot	Vizualizálja a jármű által épített térképet és a lejegyzett pótokat a Matplotlib segítségével
-r, --random	Véletlenszerű seed-et használ a zajgeneráláshoz
-s, --segmentation	Engedélyezi a program számára a szegmentált adatfeldolgozást

2.4. táblázat. Kapcsolók és hatásuk

A Use case diagram a 2.8. ábrán látható.



2.8. ábra. A SLAM szoftver Use Case diagramja

Bemenet

A program bemenete a feljebb bemutatott szimulációs környezet kimenete, vagy egy szöveges fájl, mely formátumában két különböző eset merülhet fel:

- Az adott időpillanatban töténik érzékelés - ebben az esetben a bemenet ehhez az időbélyeghez tartozó része a következőképpen néz ki:

o <időbélyeg> <sebesség> <szögsebesség> (2.4)
p <időbélyeg> <távolság> <szög> <szín> <azonosító> <x> <y>
p <időbélyeg> <távolság> <szög> <szín> <azonosító> <x> <y>
.
. .
p <időbélyeg> <távolság> <szög> <szín> <azonosító> <x> <y>

Az első sorban az odometriai mérés található, amely magába foglalja a járműünk aktuális *sebességét* és *szögsebességét*. A maradék, tetszőleges mennyiségű, de legalább egy sorban pedig a jármű érzékelési szenzora alapján kapott adatok találhatók. Ezek az adatok nem más, mint a *távolság* és *szög* egy adott *színű* bőja irányába, ezen bőja *azonosítója* és az *x* és *y* pozíciója egy globális koordináta-rendszerben.

- Az adott időpillanatban nem töténik érzékelés - ebben az esetben a bemenet a következőképpen néz ki:

o <időbélyeg> <sebesség> <szögsebesség> (2.5)
o <időbélyeg> <sebesség> <szögsebesség>
. . .
o <időbélyeg> <sebesség> <szögsebesség>

A bemenet ezen részén tetszőleges mennyiségű sorban a járműünk aktuális *sebességét* és *szögsebességét* láthatjuk.

Kimenet

A program kimenete egy szöveges fájl, amely terjedelme a bemenettől függ. A kimenet pontos formátuma a program paraméterei alapján változik, de még így is két különböző eset merülhet fel:

- Szegmentálás kapcsoló nincs használva:

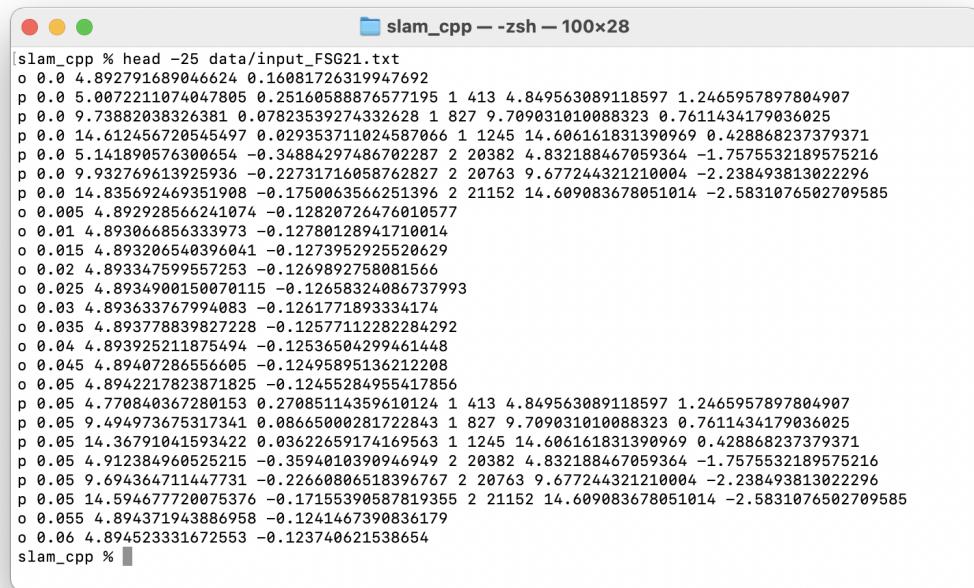
```
==== (2.6)
<azonosító> <x> <y> <orientáció>
<azonosító> <x> <y> <orientáció>
.
.
.
<azonosító> <x> <y> <orientáció>
==== 
<azonosító> <x> <y> <szín> <tényleges x> <tényleges y>
<azonosító> <x> <y> <szín> <tényleges x> <tényleges y>
.
.
.
<azonosító> <x> <y> <szín> <tényleges x> <tényleges y>
==== 
<azonosító> <x> <y> <orientáció>
<azonosító> <x> <y> <orientáció>
.
.
.
<azonosító> <x> <y> <orientáció>
==== 
<azonosító> <x> <y> <szín> <tényleges x> <tényleges y>
<azonosító> <x> <y> <szín> <tényleges x> <tényleges y>
.
.
.
<azonosító> <x> <y> <szín> <tényleges x> <tényleges y>
==== 
END
```

A kimenet "====" szöveget tartalmazó sorokkal különböző részekre van osztva. Az első ilyen részlet a gráfban található pózokat tartalmazza optimalizálás előtt, a második részlet a gráfban található bójaadatokat tartalmazza optimalizálás előtt, a harmadik részlet a gráfban található pózokat tartalmazza optimalizálás után, illetve a negyedik részlet a gráfban található bójaadatokat tartalmazza optimalizálás után. A kimenet "END" szöveget tartalmazó sora a program végét jelenti.

- Szegmentálás kapcsoló használva van:

A kimenet felépítése annyiban tér el az előző esettől, hogy ez a 4 részlet szegmenseneként ismétlődik.

Példák



```
[slam_cpp % head -25 data/input_FSG21.txt
o 0.0 4.892791689046624 0.16081726319947692
p 0.0 5.0072211074947805 0.25160588876577195 1 413 4.849563089118597 1.2465957897804907
p 0.0 9.73882038326381 0.07823539274332628 1 827 9.709031010088323 0.7611434179036025
p 0.0 14.612456720545497 0.029353711024587066 1 1245 14.606161831390969 0.428868237379371
p 0.0 5.141890576300654 -0.34884297486702287 2 20382 4.832188467059364 -1.7575532189575216
p 0.0 9.932769613925936 -0.22731716058762827 2 20763 9.677244321210004 -2.238493813022296
p 0.0 14.835692469351988 -0.1750063566251396 2 21152 14.609083678051014 -2.5831076502709585
o 0.005 4.892928566241874 -0.12820726476010577
o 0.01 4.893066856333973 -0.12780128941710014
o 0.015 4.893206549396041 -0.1273952925520629
o 0.02 4.893347599557253 -0.1269892758081566
o 0.025 4.8934900150070115 -0.12658324086737993
o 0.03 4.893633767994083 -0.1261771893334174
o 0.035 4.893778839827228 -0.12577112282284292
o 0.04 4.893925211875494 -0.12536504299461448
o 0.045 4.89407286556685 -0.12495895136212208
o 0.05 4.8942217823871825 -0.12455284955417856
p 0.05 4.770840367280153 0.27085114359610124 1 413 4.849563089118597 1.2465957897804907
p 0.05 9.494973675317341 0.08665000281722843 1 827 9.709031010088323 0.7611434179036025
p 0.05 14.36791041593422 0.03622659174169563 1 1245 14.606161831390969 0.428868237379371
p 0.05 4.912384960525215 -0.3594010390946949 2 20382 4.832188467059364 -1.7575532189575216
p 0.05 9.694364711447731 -0.22660896518396767 2 20763 9.677244321210004 -2.238493813022296
p 0.05 14.594677720075376 -0.17155390587819355 2 21152 14.609083678051014 -2.5831076502709585
o 0.055 4.894371943886958 -0.1241467390836179
o 0.06 4.894523331672553 -0.123740621538654
slam_cpp % ]
```

2.9. ábra. Bemeneti fájl részlete a SLAM FSG21 zaj nélküli futásához

```

slam_cpp -- zsh -- 100x43
|slam_cpp % ./build/src/slam data/input_FSG21.txt build/out.txt
]
Solver Summary (v 2.0.0-eigen-(3.4.0)-lapack-suitesparse-(5.10.1)-cxsparse-(3.2.0)-acceleratesparse-
eigensparse-no_openmp)

          Original           Reduced
Parameter blocks      12638        1434
Parameters            26597        4188
Residual blocks       11203        11203
Residuals             23726        23726

Minimizer             TRUST_REGION

Sparse linear algebra library   SUITE_SPARSE
Trust region strategy        LEVENBERG_MARQUARDT

          Given           Used
Linear solver    SPARSE_NORMAL_CHOLESKY
Threads          1
Linear solver ordering   AUTOMATIC        1434

Cost:
Initial          1.488273e+02
Final            7.690257e-02
Change           1.487504e+02

Minimizer iterations      8
Successful steps         8
Unsuccessful steps       0

Time (in seconds):
Preprocessor        0.004281

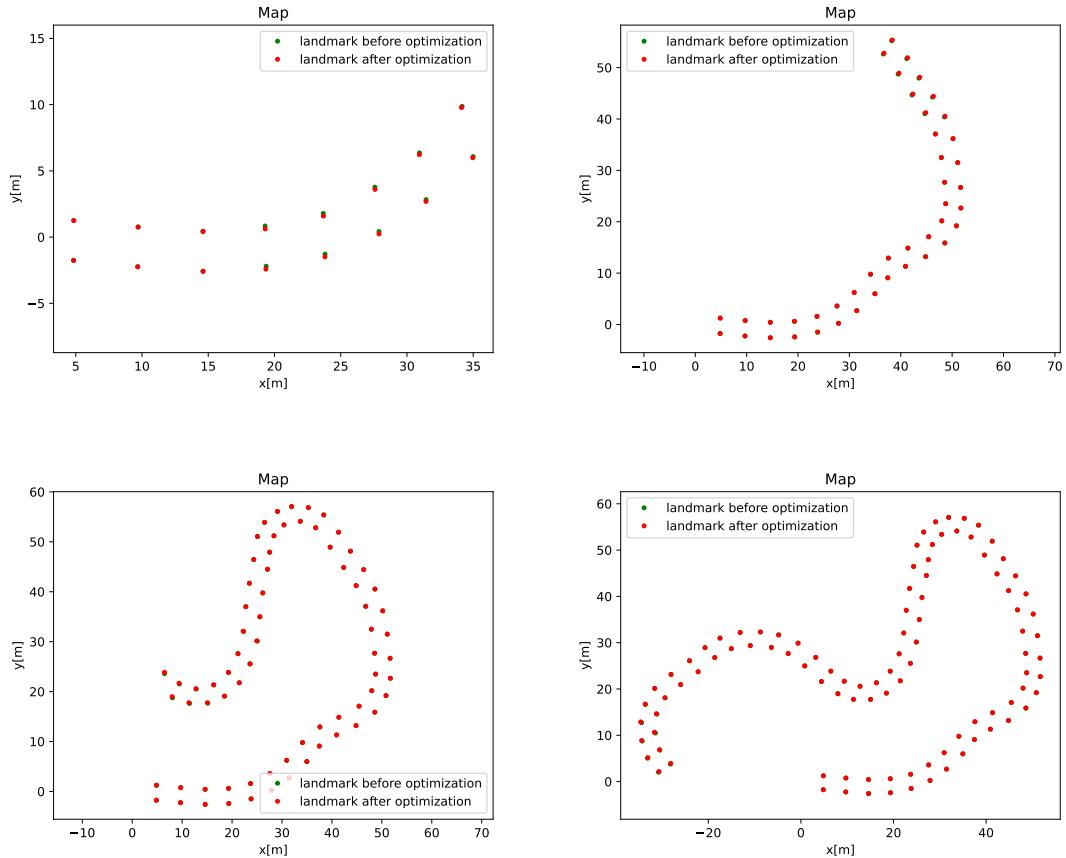
          0.005653 (8)
Residual only evaluation
          0.015009 (8)
          0.029895 (8)
Jacobian & residual evaluation
Linear solver        0.056305
Minimizer

Postprocessor        0.000187
Total              0.060773

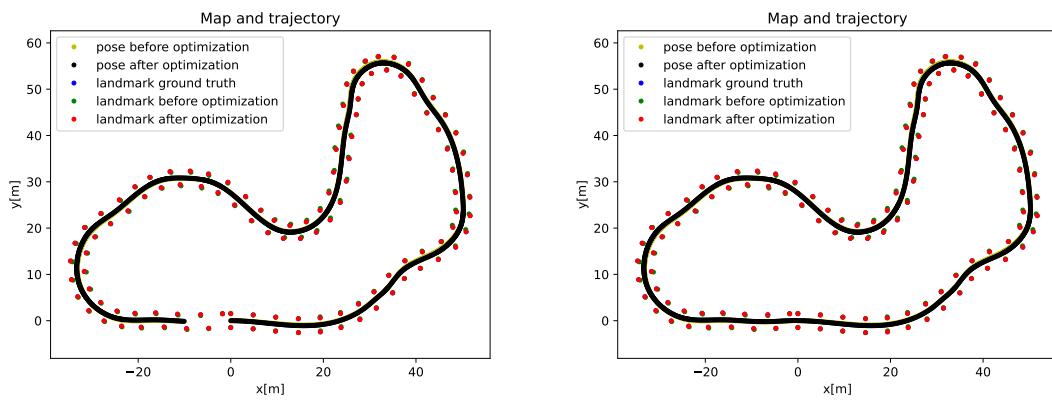
Termination:           CONVERGENCE (Function tolerance reached. |cost_change|/cost: 4.236
713e-07 <= 1.000000e-06)

```

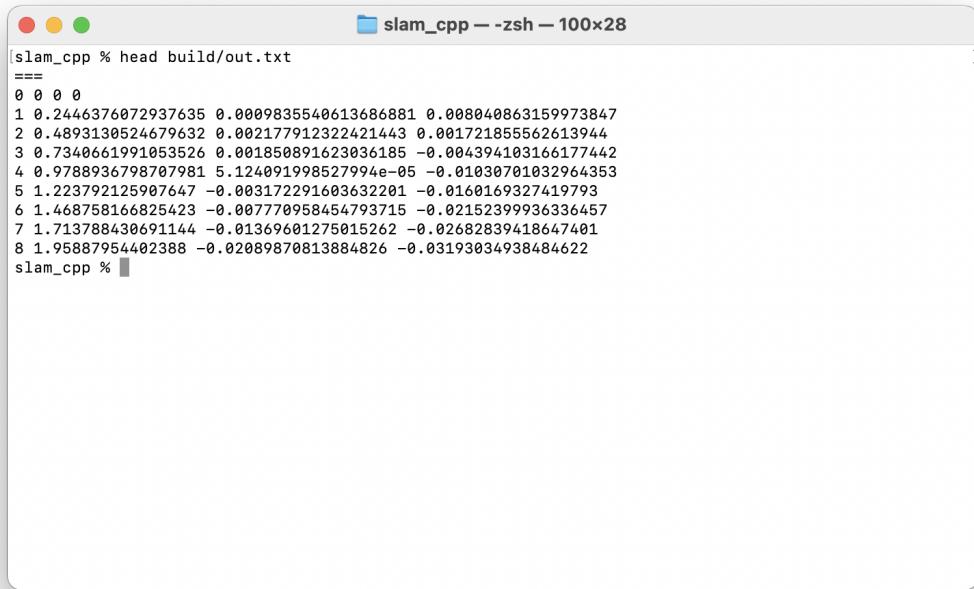
2.10. ábra. SLAM FSG21 zaj nélküli futása



2.11. ábra. SLAM FSG21 zaj nélküli futás szegmentált vizualizációja



2.12. ábra. SLAM FSG21 zaj nélküli futás vizualizációja



```
slam_cpp % head build/out.txt
===
0 0 0
1 0.2446376072937635 0.000983540613686881 0.008040863159973847
2 0.4893130524679632 0.002177912322421443 0.001721855562613944
3 0.7340661991053526 0.001850891623036185 -0.004394103166177442
4 0.9788936798707981 5.124091998527994e-05 -0.01030701032964353
5 1.223792128907647 -0.003172291603632201 -0.0160169327419793
6 1.468758166825423 -0.007770958454793715 -0.02152399936336457
7 1.713788430691144 -0.01369601275015262 -0.02682839418647401
8 1.95887954402388 -0.02089870813884826 -0.03193034938484622
slam_cpp %
```

2.13. ábra. Kimeneti fájl részlete a SLAM FSG21 zaj nélküli futásához

3. fejezet

Elméleti háttér

Ebben a fejezetben bemutatom a szoftverek megértéséhez szükséges fogalmakat, matematikai tételeket, algoritmusokat.

3.1. Definíciók és tételek

1. Definíció (Vektorok „hossza”). [8]

Az $x \in \mathbb{R}$ vektor hagyományos értelemben vett hosszát, avagy „kettes normáját” jelölje $\|\cdot\|_2$. A következőképpen számolható:

$$\|\cdot\|_2 := \sqrt{\langle x, x \rangle} = \sqrt{x^T x} = \left(\sum_{k=1}^n x_i^2 \right)^{\frac{1}{2}} \quad (3.1)$$

Megjegyzés. A szimulációs környezetben és a SLAM-ben is pontok közötti távolság kiszámítására van használva.

2. Definíció (Görbület). [9]

Legyen $\Gamma \subset \mathbb{R}^n$ egy egyszerű sima görbe. Tegyük fel, hogy $\Phi : [0, L] \rightarrow \Gamma$ ennek az ívhossz szerinti kétszer folytonosan deriválható paraméterezése (L a Γ ívhossza).

Az $s \in [0, L]$ -ben (azaz a $\Phi(s)$ pontban) a Γ görbe görbületén a

$$\kappa(s) := |\Phi''(s)| \quad (3.2)$$

számot értjük.

Megjegyzés. A szimulációs környezetben a sebességadatok és a bóják közötti távolság a görbület függvényében kerülnek kiszámításra.

1. Tétel (Görbület tetszőleges paraméterezéssel).

Legyen $\Gamma \subset \mathbb{R}^n$ egy egyszerű sima görbe, $\varphi : [\alpha, \beta] \rightarrow \Gamma$ egy tetszőleges, de C^2 -beli paramétereze. Ekkor a görbe $t_0 \in [\alpha, \beta]$ paraméterű $\varphi(t_0)$ pontjában a görbület:

$$\kappa(s) = |\Phi''(s)| = \frac{|\varphi'(t) \times \varphi''(t)|}{|\varphi'(t)|^3} \quad (3.3)$$

3. Definíció (Forgatási mátrix).

$$\mathbf{R}(\theta) = \begin{bmatrix} \cos \theta & -\sin \theta \\ \sin \theta & \cos \theta \end{bmatrix} \in SO(2) \quad (3.4)$$

4. Definíció (Normális eloszlás).

Az X valószínűségi változó normális eloszlást követ pontosan akkor, ha sűrűségfüggvénye

$$f(x) = \frac{1}{\sigma \sqrt{2\pi}} \cdot e^{-\frac{(x-m)^2}{2\sigma^2}}, \quad (3.5)$$

ahol a két paraméter m és $\sigma \in \mathbb{R}$, valamint $\sigma > 0$.

Megjegyzés. A szimulációs környezetben és a SLAM-ben is az adatok zajosítására van használva.

5. Definíció (B-Spline). [10]

Az $f : \mathbb{R} \rightarrow \mathbb{R}$ függvény tartója a következő valós számhalmaz

$$\text{supp}(f) := \overline{\{x \in \mathbb{R} : f(x) \neq 0\}}. \quad (3.6)$$

$\Omega_\infty := \{..., x_{-n}, ..., x_{-1}, x_0, x_1, ..., x_n, ...\}$ alappontrendszer

$S_l(\Omega_\infty)$: az Ω_∞ alappontrendszeren értelmezett l -edfokú spline-ok halmaza.

A $B_{l,k} \in S_l(\Omega_\infty)$ spline-okat *B-spline*-oknak nevezzük, ha

- $B_{l,k}(x) \geq 0 (\forall x \in \mathbb{R})$,
- $\text{supp}(B_{l,k})$ minimális,
- $\sum_{k \in \mathbb{Z}} B_{l,k}(x) \equiv 1 (\forall x \in \mathbb{R})$.

Megjegyzés. A szimulációs környezetben a pálya középvonalának és a bóják vonalának sűrűbbé tételeire van használva.

6. Definíció (Cholesky-felbontás). [11]

Az $A \in \mathbb{R}^{n \times n}$ mátrix Cholesky-felbontásának nevezük az $L \cdot L^T$ szorzatot, ha $A = LL^T$, ahol $L \in \mathbb{R}^{n \times n}$ alsó háromszögmátrix és $I_{i,i} > 0$ ($i = 1, 2, \dots, n$).

Megjegyzés. A SLAM-ben használt Ceres Solver-nél [12] tudjuk beállítani mint lineáris egyenletrendszer-megoldó algoritmust.

2. Tétel (Létezik Cholesky-felbontás).

Ha A szimmetrikus, pozitív definit mátrix, akkor egyértelműen létezik Cholesky-felbontása.

7. Definíció (A legkisebb négyzetek módszere). [13]

Legyenek $h_i \in \mathbb{R}^n \rightarrow \mathbb{R}$ ($i = 1, 2, \dots, m$) függvények, $y_i \in \mathbb{R}$ mérések, illetve egy $\mathbf{x} \in \mathbb{R}^n$ paramétervektor. Az i -ik mérésre definiálunk egy $f_i \in \mathbb{R}^n \rightarrow \mathbb{R}$ költségfüggvényt, mellyel a méréstől való eltérés a következőképpen fejezhető ki:

$$f_i(\mathbf{x}) = h_i(\mathbf{x}) - y_i. \quad (3.7)$$

Legyen adott a következő optimalizációs probléma:

$$\arg \min_{\mathbf{x}} \underbrace{\sum_{i=1}^m (f_i(\mathbf{x}))^2}_{J(\mathbf{x})}, \quad (3.8)$$

ahol $J(\mathbf{x}) \in \mathbb{R}^m \rightarrow \mathbb{R}_0^+$ az úgynévezett költségfüggvény – azaz a (3.7) alapján, a $h_i(\mathbf{x})$ predikciók az y_i mérésektől vett négyzetes eltérését minimalizáljuk.

A legkisebb négyzetek módszere alapján a megoldást a (3.8) egyenletben felírt optimalizációs problémára, \mathbf{x} -re, pedig a $\nabla J(\mathbf{x})$ kinullázásával kaphatjuk meg:

$$\nabla J(\mathbf{x}) = \mathbf{0}. \quad (3.9)$$

Megjegyzés (Reziduálisok). Az $f_i(\mathbf{x})$ függvényértékeket másnéven *reziduálisoknak* nevezzük.

Megjegyzés. A $f_i(\mathbf{x})$ költségfüggvény kifejezésére a dolgozatban alkalmazott Ceres Solver-ben [12] az általam definiált PoseErrorFunction (4.2.1) és LandmarkErrorFunction (4.2.1) generikus funktor típusok használhatóak.

3.2. Gauss-Newton módszer

A fent említett *legkisebb négyzetek módszerére* egy numerikus megoldást adó módszer a Gauss-Newton módszer [14], mely a „sima” Newton-módszerből származtattható, így itt is egy iteratív közelítő eljárásról van szó. minden egyes lépés után egy várhatóan jobb közelítést biztosít a megoldás felé, ám fontos megemlíteni, hogy erősen támaszkodik a paraméterek jó kezdeti értékére.

Az eredeti Newton-féle numerikus eljárásnál az alábbi összefüggés segítségével végzi az iterációt:

$$\mathbf{x}_{i+1} = \mathbf{x}_i - H_{J(\mathbf{x}_0)}^{-1} \nabla J(\mathbf{x}_i) \quad (3.10)$$

Tegyük fel, hogy $J(\mathbf{x})$ -et az alábbi alakra lehet hozni:

$$J(\mathbf{x}) = \sum_{i=1}^m (f_i(\mathbf{x}))^2 = f(\mathbf{x})^T f(\mathbf{x}), \quad (3.11)$$

ahol $f(\mathbf{x})^T = [f_1(\mathbf{x}), f_2(\mathbf{x}), \dots, f_m(\mathbf{x})]$. Úgy kell elképzelni, hogy a hibafüggvényünk legkisebb négyzetes alakban adott, és összesen m darab – magában skalár – elemre lehet bontani, melyek négyzetét adjuk össze. Ebben az esetben $J(\mathbf{x})$ gradiensét az alábbi alakban írhatjuk fel:

$$\nabla J(\mathbf{x}) = 2 \nabla f(\mathbf{x})^T f(\mathbf{x}), \quad (3.12)$$

ahol $\nabla f(\mathbf{x})$ most egy $m \times n$ -szeres mátrixot jelöl (amit Jacobi-mátrixnak szokás nevezni):

$$\nabla f(\mathbf{x}) = \begin{bmatrix} \frac{\partial f_1(\mathbf{x})}{\partial x^1} & \dots & \frac{\partial f_1(\mathbf{x})}{\partial x^n} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_m(\mathbf{x})}{\partial x^1} & \dots & \frac{\partial f_m(\mathbf{x})}{\partial x^n} \end{bmatrix} \quad (3.13)$$

Ezt úgy tudjuk belátni, hogy felírjuk a tetszőleges x^k szerinti parciális deriváltat: (A változók megkülönböztetésére most kivételesen felső indexet használunk, mert az alsó indexet már az iteráció számára felhasználtuk. Ezért $\mathbf{x} = [x^1, x^2, \dots, x^n]^T$ jelöli

most a változóvektorokat.)

$$\frac{\partial J(\mathbf{x})}{\partial x^k} = \frac{\sum \partial(f_i(\mathbf{x}))^2}{\partial x^k} = \sum 2f_i(\mathbf{x}) \frac{\partial f_i(\mathbf{x})}{\partial x^k} \quad (3.14)$$

Ezt pedig az $f(\mathbf{x})$ gradiens vektorának a k -dik eleme megszorozva $f_i(\mathbf{x})$ -ek összegével. A Hesse mátrix összeállításához szükség van a második parciális deriváltakra:

$$\frac{\partial J(\mathbf{x})}{\partial x^k x^l} = 2 \sum \left(\frac{\partial f_i(\mathbf{x})}{\partial x^k} \frac{\partial f_i(\mathbf{x})}{\partial x^l} + f_i(\mathbf{x}) \frac{\partial^2 f_i(\mathbf{x})}{\partial x^k \partial x^l} \right) \quad (3.15)$$

Ha a második tagot elhagyjuk, átírhatjuk az összefüggésünket:

$$\frac{\partial J(\mathbf{x})}{\partial x^k x^l} \approx 2 \sum \left(\frac{\partial f_i(\mathbf{x})}{\partial x^k} \frac{\partial f_i(\mathbf{x})}{\partial x^l} \right) \quad (3.16)$$

Ennek az elhanyagolásnak az az előnye, hogy nem kell a második parciális deriváltakat kiszámítani, hanem elég az első deriváltak szorzatával dolgozni. Mindez akkor igaz, ha az egyes $f_i(\mathbf{x})$ -ek értéke kicsi, azaz a hibafüggvényünk viszonylag kicsi értékeket tartalmaz.

A Hesse mátrix ebben az esetben az alábbi alakra egyszerűsödik le:

$$H_{J(\mathbf{x}_0)} \approx 2 \nabla f(\mathbf{x})^T \nabla f(\mathbf{x}) \quad (3.17)$$

Maga az optimalizálás pedig továbbra is egy iteráció, amelyet a visszahelyettesítések elvégzésével az alábbi alakra hozhatunk (a 2-es szorzó nélkül):

$$x_{i+1} = x_i - \left(\nabla f(\mathbf{x})^T \nabla f(\mathbf{x}) \right)^{-1} \nabla f(\mathbf{x})^T f(\mathbf{x}) \quad (3.18)$$

3.3. Mozgási modell

Az önvezető jármű mozgási modellje egy egyszerű kinematikus modell, ami megbecsüli az autó pozíóját a térképen. A bemenete a jármű v_t sebessége és ω_t szögsebessége.

$$\mathbf{u}_k = \begin{bmatrix} v_k \\ \omega_k \end{bmatrix} \quad (3.19)$$

A való életben ehhez a méréshez minimális zaj is társul a mozgási modell bizonytalansága miatt.

A jármű helyzete a következő egyenlettel számítható ki:

$$\begin{bmatrix} p_{x,k} \\ p_{y,k} \\ \psi_k \end{bmatrix} = \begin{bmatrix} p_{x,k-1} + v_{k-1} \cdot \cos(\psi_{k-1} + \frac{\omega_{k-1}}{2} + \beta_{k-1}) \\ p_{y,k-1} + v_{k-1} \cdot \sin(\psi_{k-1} + \frac{\omega_{k-1}}{2} + \beta_{k-1}) \\ \psi_{k-1} + \omega_{k-1} \end{bmatrix}, \quad (3.20)$$

ahol a k -dik időpillanatban véve $p_{x,k}$ és $p_{y,k}$ a jármű x és y koordinátája, ψ_k a jármű orientációja, v_k és ω_k a jármű sebessége és szögsebessége megszorozva a mintavételezési időalappal, β_k pedig az oldalcsúszás, ami a szimulált környezetünkben a 0 értéket veszi fel.

Megjegyzés. A szimulációs környezetben és a SLAM-ben is a járműpózok kiszámítására van használva.

3.4. Hurokzárás

Hurokzárásnak nevezük azt az esetet, amikor a robotunk visszatér egy olyan helyre, ahol korábban járt már az útvonala során. A hurokzárás folyamán egy olyan vizsgálat indul, amely végigveszi az eddig bezárt helyeket, legyen ez kamerakép, vagy más adatok alapján, és egyezést keres a jelenlegi pozíóján megfigyelt adatokkal. Az idő folyamán bekövetkezett sodródás kikiúszóbólésére használható egyszerű és teljesítményre nagy hatást gyakorló módszerről van szó.

Mivel az én implementációmban az érzékelés Lidar szenzort szimulálva valósul meg, így csupán távolság- és szögadatok állnak rendelkezésre, amelyek a szimulációs környezet miatt el vannak látva a megfigyelt bójához tartozó azonosítóval.

Látható, hogy minden adott a tökéletes hurokzáráshoz, ugyanis az optimalizációtól nincsenek jelen hibásan felismert érzékelési adatok, így egyszerűen a bóják egyedi azonosítója alapján megoldható a feladat.

4. fejezet

Fejlesztői dokumentáció

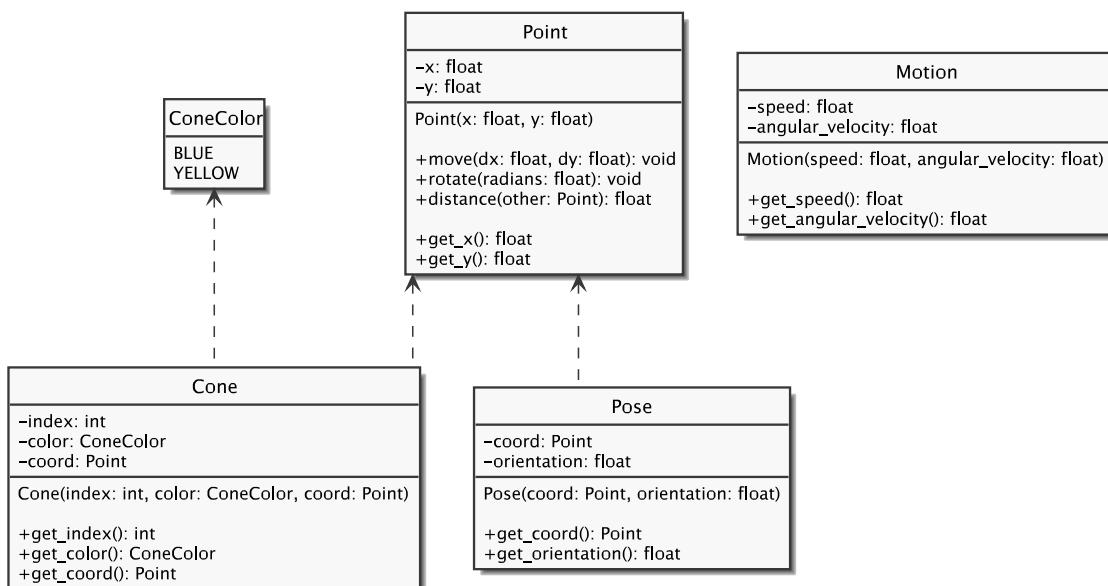
Az alábbiakban bemutatom a szoftverek fejlesztéséhez szükséges legfontosabb tudnivalókat.

4.1. Szimulációs környezet

4.1.1. A szoftver felépítése

Függőségek - A függőségek listáját, illetve telepítési útmutatóját feljebb, a felhasználói dokumentáció 2.1 fejezetében találjuk.

Statikus terv



4.1. ábra. Szimulációs környezetben használt típusok osztálydiagramja

Cone - egy bóját valósít meg.

Implementációja: *simulator/cone.py*.

Adattagok:

- index: int - a bójához tartozó egyedi azonosító
- color: ConeColor - a bójá színe (1 - kék, 2 - sárga)
- coord: Point - a bójá pozíciója a globális koordináta-rendszerben

Metódusok:

- Cone(index: int, color: ConeColor, coord: Point)
 - index: int - a bójához tartozó egyedi azonosító
 - color: ConeColor - a bójá színe (1 - kék, 2 - sárga)
 - coord: Point - a bójá pozíciója a globális koordináta-rendszerben
- get_index(): int
- get_color(): ConeColor
- get_coord(): Point

Motion - egy odometriai mérést valósít meg.

Implementációja: *simulator/motion.py*.

Adattagok:

- speed: float - sebesség(v)
- angular_velocity: float - szögsebesség(ω)

Metódusok:

- Motion(speed: float, angular_velocity: float)
 - speed: float - sebesség(v)
 - angular_velocity: float - szögsebesség(ω)
- get_speed(): float
- get_angular_velocity(): float

Point - egy síkbeli pontot valósít meg.

Implementációja: *simulator/point.py*.

Adattagok:

- x: float
- y: float

Metódusok:

- Point(x: float, y: float)
- move(dx: float, dy: float): void - a paraméreknek megfelelően elmozgatja az *x* és *y* koordinátákat.
- rotate(radians: float): void - forgatást végez a paraméterül kapott *radians* függvényében.
- distance(other: Point): float - kiszámolja a távolságot az objektum és a paraméterül kapott *other* objektum között.
- get_x(): float
- get_y(): float

Pose - egy járműpózt valósít meg.

Implementációja: *simulator/pose.py*.

Adattagok:

- coord: Point - a jármű pozíciója a globális koordináta-rendszerben
- orientation: float - a jármű orientációja

Metódusok:

- Pose(coord: Point, orientation: float)
 - coord: Point - a jármű pozíciója a globális koordináta-rendszerben
 - orientation: float - a jármű orientációja

- get_coord(): Point
- get_orientation(): float

Dinamikus terv



4.2. ábra. Szimulációs környezet folyamatábrája

4.1.2. Továbbfejlesztési lehetőségek

Grafikus kezelőfelület

A parancssorból meghívandó script jelenleg nem éppen a leginkább felhasználóbarátabb, ezért érdemes lehet egy grafikus kezelőfelületet megvalósítani az alkalmazás kezeléséhez.

Pályaszerkesztő

Az alkalmazás jelenleg bemenetként kapott pályaközépvonalal dolgozik, azonban érdemes lehet a felhasználónak biztosítani egy funkciót, amelyben kézzel tudja alkalmazáson belül megtervezni a pályát, amire aztán a szimulációt futtatná.

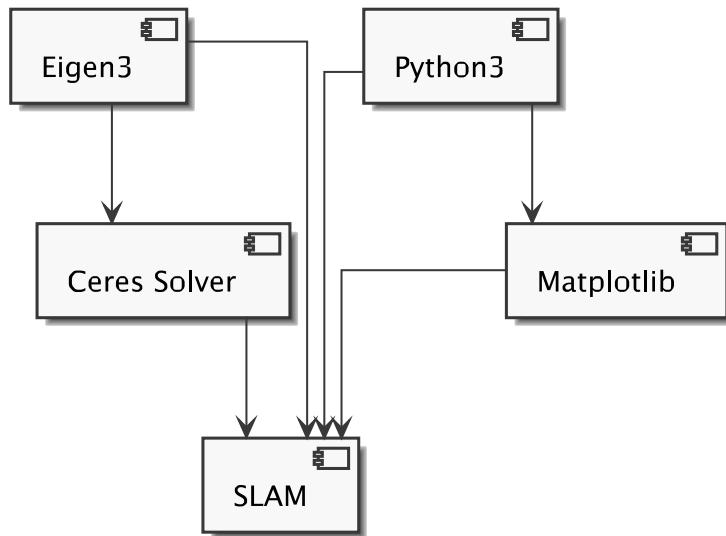
Vizualizáció

Az adatok vizualizálására használt Matplotlib több mint elég, ha az adatokat nem valós időben szeretnénk folyamatosan megjeleníteni. A valós idejű ábrázoláshoz azonban szükséges egy saját vizualizációs eszközt kifejleszteni.

4.2. SLAM

4.2.1. A szoftver felépítése

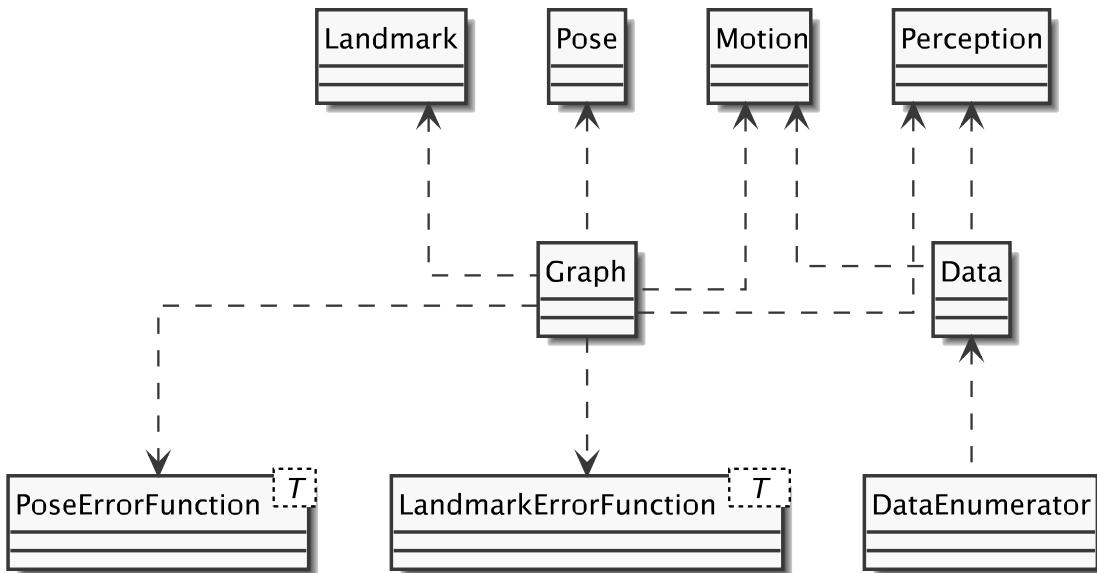
Függőségek



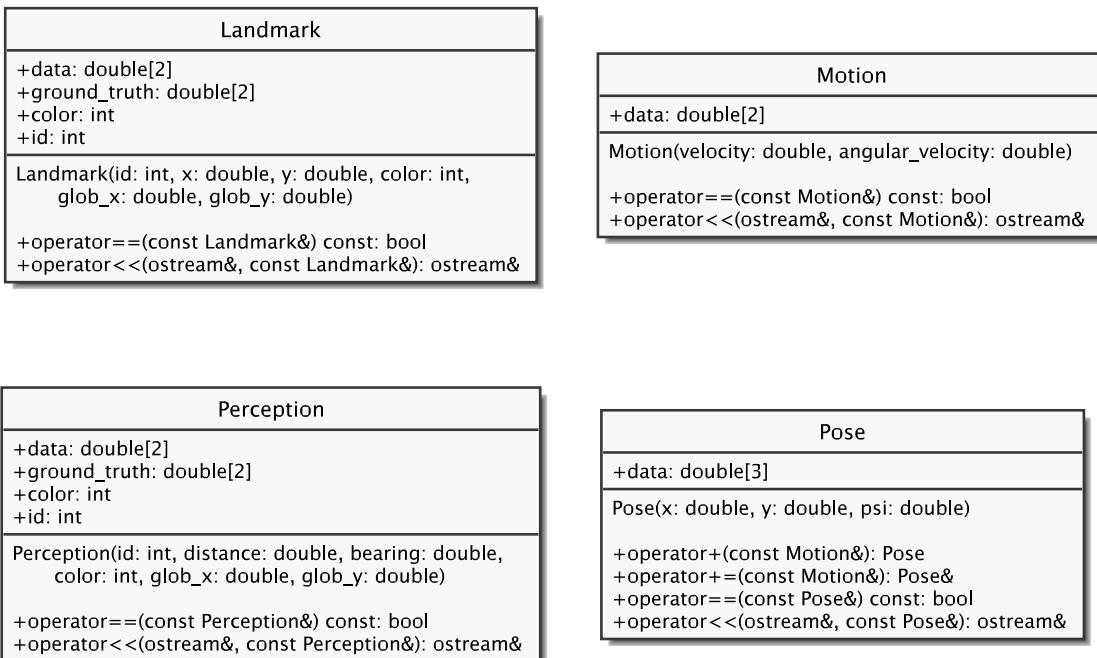
4.3. ábra. SLAM függőségi diagramja

- Eigen3 [15] - nyílt forráskódú, magas szintű C++ sablonkönyvtár lineáris algebra, mátrix- és vektorműveletek, geometriai transzformációk, numerikus megoldók és egyéb kapcsolódó algoritmusok számára.
- Ceres Solver [12] - nyílt forráskódú C++ könyvtár robusztus, bonyolult optimalizálási problémák modellezésére és hatékony megoldására, melyet a Google 2010 óta használ éles környezetben.
- Python3 - magas szintű, interpreteres, általános rendeltetésű programozási nyelv.
- Matplotlib [5] - egy széleskörű könyvtár statikus, animált és interaktív vizualizációk létrehozásához Python-ban.

Statikus terv



4.4. ábra. SLAM osztálydiagramja



4.5. ábra. Egyedi típusok osztálydiagramja

Landmark - egy tereptárgyat valósít meg.

Implementációja: *slam.cpp/src/misc/Types.h*.

Adattagok:

- `data: double[2]` - x és y koordináták

- ground_truth: double[2] - tényleges x és y koordináták
- color: int
- id: int

Metódusok:

- Landmark(id: int, x: double, y: double, color: int, glob_x: double, glob_y: double)
 - id - a bójához tartozó egyedi azonosító
 - x, y - a bójá pozíciója a globális koordináta-rendszerben
 - color - a bójá színe (1 - kék, 2 - sárga)
 - glob_x, glob_y - a bójá tényleges pozíciója a globális koordináta-rendszerben
- operator==(const Landmark&) const: bool
- operator«(ostream&, const Landmark&): ostream&

Motion - egy odometriai mérést valósít meg.

Implementációja: *slam_.cpp/src/misc/Types.h*.

Adattagok:

- data: double[2] - sebesség(v) és szögsebesség(ω)

Metódusok:

- Motion(velocity: double, angular_velocity: double)
 - velocity - sebesség(v)
 - angular_velocity - szögsebesség(ω)
- operator==(const Motion&) const: bool
- operator«(ostream&, const Motion&): ostream&

Perception - egy szimulált Lidar mérést valósít meg.

Implementációja: *slam_cpp/src/misc/Types.h*.

Adattagok:

- data: double[2] - távolság és szög
- ground_truth: double[2] - tényleges x és y koordináták
- color: int
- id: int

Metódusok:

- Perception(id: int, distance: double, bearing: double, color: int, glob_x: double, glob_y: double)
 - id - a bójához tartozó egyedi azonosító
 - distance - a bója érzékelő szenzorunktól mért távolsága
 - bearing - a bója a jármű x tengelyével bezárt szöge
 - color - a bója színe (1 - kék, 2 - sárga)
 - glob_x, glob_y - a bója tényleges pozíciója a globális koordinátarendszerben
- operator==(const Perception&) const: bool
- operator«(ostream&, const Perception&): ostream&

Pose - egy járműpózt valósít meg.

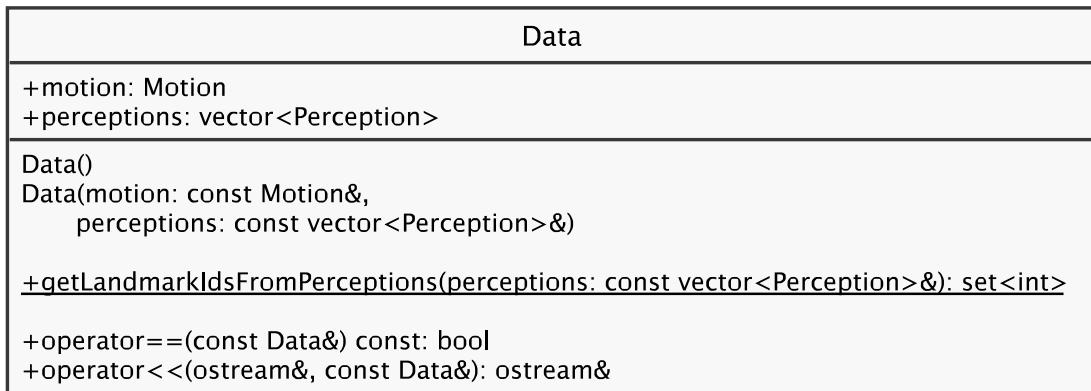
Implementációja: *slam_cpp/src/misc/Types.h*.

Adattagok:

- data: double[3] - x , y és ψ

Metódusok:

- Pose(x: double, y: double, psi: double)
 - x, y - a jármű pozíciója a globális koordináta-rendszerben
 - psi - a jármű orientációja
- operator+(const Motion&) const: Pose - lásd 3.3
- operator+=(const Motion&) const: Pose& - lásd 3.3
- operator==(const Pose&) const: bool
- operator«(ostream&, const Pose&): ostream&



4.6. ábra. Data osztálydiagramja

Data - egy jármű által küldött adatcsomagot valósít meg egy adott időpillanatban, ahol jelen van egy odometriai és nulla, vagy több szimulált Lidar mérés.

Implementációja: *slam_cpp/src/dataenumerator/DataEnumerator.h*.

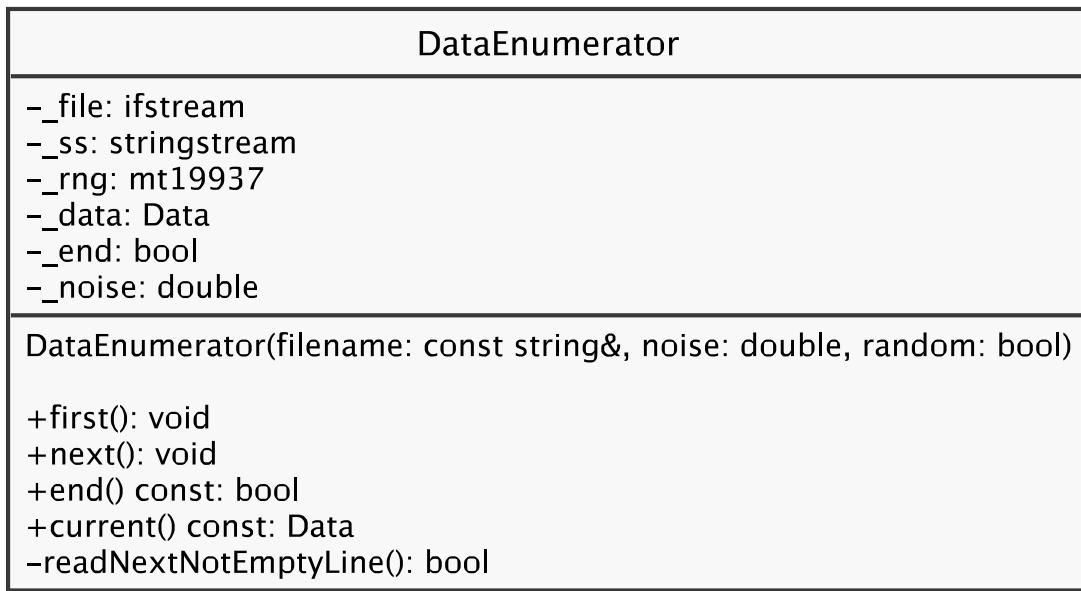
Adattagok:

- motion: Motion
- perceptions: vector<Perception>

Metódusok:

- Data()
- Data(motion: const Motion&, perceptions: const vector<Perception>&)

- motion - a sebességadatokat tartalmazó objektum
- percepitons - az érzékelési adatokat tároló objektumokat tartalmazó vektor
- getLandmarkIdsFromPerceptions(perceptions: const vector<Perception>&): static set<int> - kigyűjti a paraméterül kapott *perceptions* vector-ból az *id*-kat egy set-be, majd visszatér vele
- operator==(const Data&) const: bool
- operator«(ostream&, const Data&): ostream&



4.7. ábra. DataEnumerator osztálydiagramja

DataEnumerator - a jármű által küldött adatcsomagokat sorolja fel további fel-dolgozás céljából, illetve opcionálisan zajosítja is őket.

Implementációja: *slam_* *cpp/src/dataenumerator/DataEnumerator.[h, cpp]*.

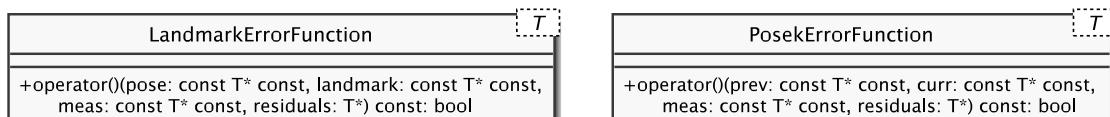
Adattagok:

- *_file*: ifstream
- *_ss*: stringstream
- *_rng*: mt19937

- `_data: Data`
- `_end: bool`
- `_noise: double`

Metódusok:

- `DataEnumerator(filename: const string&, noise: double, random: bool)`
 - `filename` - a bemeneti fájl elérési útja
 - `noise` - a normális eloszlású zaj szórása (σ)
 - `random` - random seed használata a zajgeneráláshoz
- `first(): void` - beolvassa az első sort a fájlból
- `next(): void` - feldolgozza a beolvasott adatokat, majd újraolvas a fájlból
- `end() const: bool`
- `current() const: Data`
- `readNextNotEmptyLine(): bool` - beolvassa a következő nem üres sort a fájlból



4.8. ábra. Hibafüggvények osztálydiagramja

LandmarkErrorFunction - az optimalizáláshoz használt egyik hibafüggvényt valósítja meg. Egy Landmark mérését veti össze annak globális pozíójával.

Implementációja: *slam_cpp/src/misc/Utils.h*.

Metódusok:

- `operator()(pose: const T* const, landmark: const T* const, meas: const T* const, residuals: T*) const: bool`
 - `pose` - x, y és ψ
 - `landmark` - x és y

- meas - távolság és szög
- residuals - az optimalizáláshoz használt reziduálisok

Matematikai háttér:

$$\mathbf{p}_i = \begin{bmatrix} p_{x,i} \\ p_{y,i} \\ \psi_i \end{bmatrix} = \begin{bmatrix} \mathbf{t}_i \\ \psi_i \end{bmatrix} \quad (4.1)$$

$$\mathbf{m}_j = \begin{bmatrix} m_{x,j} \\ m_{y,j} \end{bmatrix} \quad (4.2)$$

$$\mathbf{R}_i = \begin{bmatrix} \cos \psi_i & -\sin \psi_i \\ \sin \psi_i & \cos \psi_i \end{bmatrix} \quad (4.3)$$

A várható mérésünket a következőképpen számolhatjuk ki:

$$\widehat{\mathbf{z}}_{ij} (\mathbf{p}_i, \mathbf{m}_j) = \mathbf{R}_i^T (\mathbf{m}_j - \mathbf{t}_i), \quad (4.4)$$

ahol \mathbf{p}_i a jármű x és y koordinátáját, illetve ψ orientációját tartalmazó vektor az i -dik időpillanatban, \mathbf{t}_i a \mathbf{p}_i -hez tartozó x és y koordinátákat tartalmazó vektor, \mathbf{m}_j a j -dik bóna x és y koordinátája, \mathbf{R}_i^T pedig a jármű orientációjához tartozó forgási mátrix (3) transzponáltja az i -dik időpillanatban.

Ekkor a hibafüggvényünket a következőképpen írhatjuk fel:

$$\begin{aligned} e_{ij} (\mathbf{p}_i, \mathbf{m}_j) &= \widehat{\mathbf{z}}_{ij} - \mathbf{z}_{ij} \\ &= \mathbf{R}_i^T (\mathbf{m}_j - \mathbf{t}_i) - \mathbf{z}_{ij}, \end{aligned} \quad (4.5)$$

ahol $\widehat{\mathbf{z}}_{ij}$ a várható méréseket, \mathbf{z}_{ij} pedig a tényleges méréseket jelenti.

Megjegyzés. A fenti hibafüggvény a (3.18)-as egyenlethez hasonlóan használható fel az optimalizáció során.

PoseErrorFunction - az optimalizáláshoz használt egyik hibafüggvényt valósítja meg. Két Pose közötti eltérést veti össze (odometriai) méréssel.

Implementációja: *slam_.cpp/src/misc/Utils.h*.

Metódusok:

- operator()(prev: const T* const, curr: const T* const, meas: const T* const, residuals: T*) const: bool
 - prev - előző pose x, y és ψ adatai
 - curr - jelenlegi pose x, y és ψ adatai
 - meas - sebesség (v) és szögsebesség (ω)
 - residuals - az optimalizáláshoz használt reziduálisok

Matematikai háttér:

$$\mathbf{p}_i = \begin{bmatrix} p_{x,i} \\ p_{y,i} \\ \psi_i \end{bmatrix} \quad (4.6)$$

A várható mérésünket a következőképpen számolhatjuk ki:

$$\hat{\mathbf{d}}_{i,i+1} = \mathbf{p}_{i+1} - \mathbf{p}_i \quad (4.7)$$

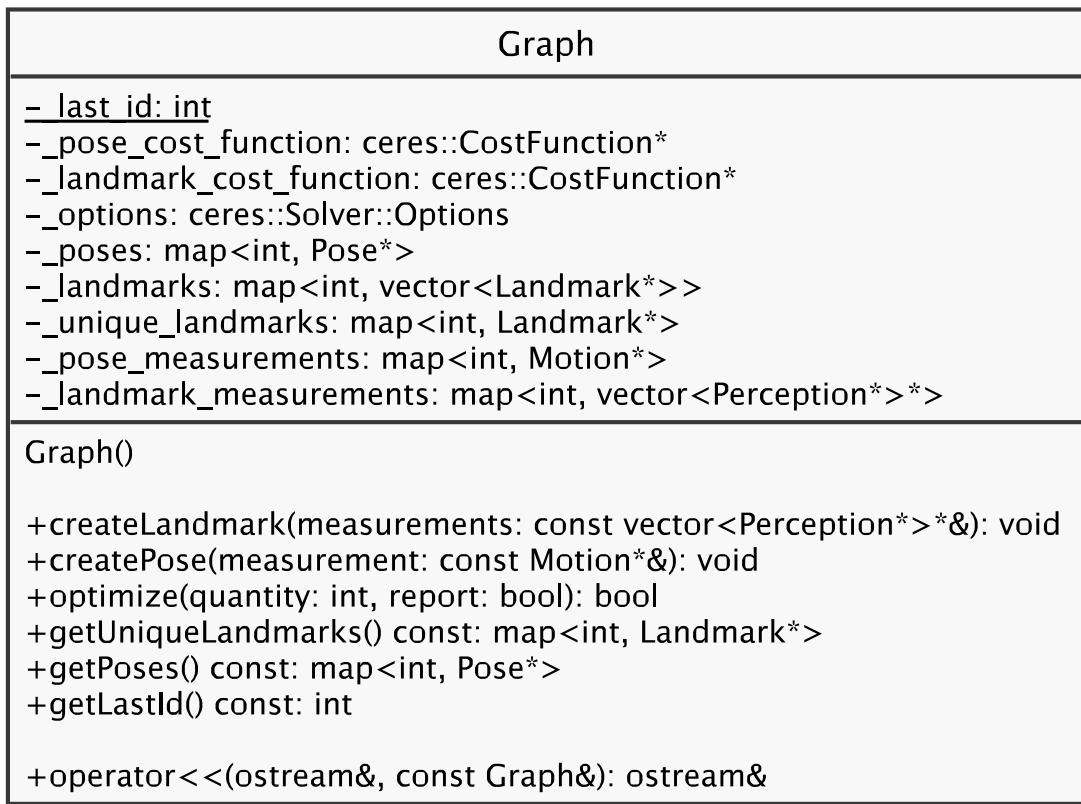
ahol \mathbf{p}_i és \mathbf{p}_j a jármű x és y koordinátáját, illetve ψ orientációját tartalmazó vektorok az i -dik és a j -dik időpillanatokban.

Ekkor a hibafüggvényünket a következőképpen írhatjuk fel:

$$\begin{aligned} e_{i,i+1}(\mathbf{p}_i, \mathbf{p}_{i+1}) &= \hat{\mathbf{d}}_{i,i+1} - \mathbf{d}_{i,i+1} \\ &= \mathbf{p}_{i+1} - \mathbf{p}_i - \mathbf{d}_{i,i+1} \end{aligned} \quad (4.8)$$

ahol $\hat{\mathbf{d}}_{ij}$ a várható méréseket, \mathbf{d}_{ij} pedig a tényleges méréseket jelenti.

Megjegyzés. A fenti hibafüggvény a (3.18)-as egyenlethez hasonlóan használható fel az optimalizáció során.



4.9. ábra. Graph osztálydiagramja

Graph - egy járműpózokból és tereptárgyakból alkotott gráf struktúrát valósít meg, illetve ennek a gráfnak az optimalizálásáért felel.

Implementációja: *slam.cpp/src/graph/Graph.h, cpp*.

Adattagok:

- `_last_id: static int`
- `_pose_cost_function: ceres::CostFunction*`
- `_landmark_cost_function: ceres::CostFunction*`
- `_options: ceres::Solver::Options`
- `_poses: map<int, Pose*>`
- `_landmarks: map<int, vector<Landmark*>>`
- `_unique_landmarks: map<int, Landmark*>`
- `_pose_measurements: map<int, Motion*>`

- `_landmark_measurements: map<int, vector<Perception*>*>`

Metódusok:

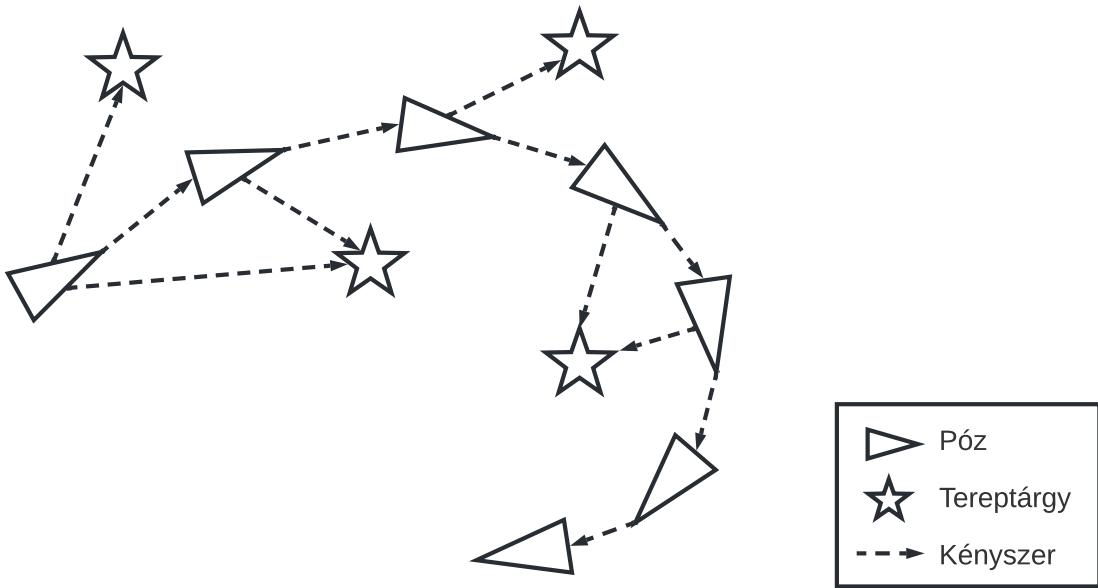
- `Graph()`
- `createLandmark(measurements: const vector<Perception*>*&): void` - új Landmark objektumokat hoz létre a paraméterül kapott *measurements* vector alapján, majd a gráfhoz adja őket egyesével, amennyiben az adott azonosítójú Landmark-hoz még nem érkezett mérés, különben pedig a már létező Landmark objektumot adja hozzá újra a gráfhoz.
- `createPose(measurement: const Motion*&): void` - új Pose objektumot hoz létre a paraméterül kapott *measurement* mérés alapján, majd a gráfhoz adja. A mérések a metódusban a PoseErrorFunction egyenlete alapján változnak.
- `optimize(quantity: int, report: bool): bool` - a Ceres AutoDiffCostFunctions segítségével felépíti az optimalizálási problémát. Ahogy a nevéből is látszik, az objektum automatikus differenciálást végez a probléma linearizálása érdekében.
- `getUniqueLandmarks() const: map<int, Landmark*>`
- `getPoses() const: map<int, Pose*>`
- `getLastId() const: int`
- `operator«(ostream&, const Graph&): ostream&` - a gráfban található Pose objektumokat írja ki a stream-re.

Dinamikus terv

Gráf struktúra

A SLAM probléma ábrázolására használt gráf a mi esetünkben egy irányított gráf, amely kétféle csúcsból és kétféle élből tevődik össze.

Minden járműpózt egy Pose csúcs, és minden tereptárgyat, amelyek a mi esetünkben bóják lesznek, egy Landmark csúcs képvisel. Az egymást követő Pose csúcsokat Pose-Pose élek kötik össze. A Landmark csúcsok minden egyes Pose csúcshoz kapcsolódnak Landmark-Pose élek használatával, ahonnan mérés történt feléjük.



4.10. ábra. SLAM probléma gráf alapú reprezentációja

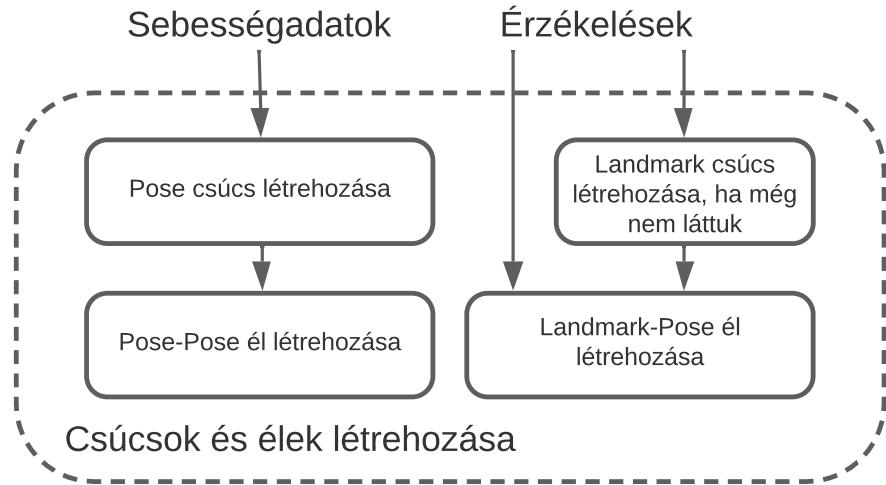
A gráf alapú SLAM folyamatot az én implementációmban az alábbiak szerint bonthatjuk fel:

Csúcsok és élek létrehozása

A szoftver ezen részének az a feladata, hogy összegyűjtse a beérkező sebesség- és érzékelési adatokat, melyek folyamatos érkezését a DataEnumerator nevezetű felsoroló biztosítja. A felsoroló létrejöttére nagy hatást gyakorolt Gregorics Tibor tanár úr osztály-sablon könyvtára [16], melyet az Objektumelvű Programozás óra keretein belül ismerhettem meg.

Sebességadat beérkezésekor elsősorban létrejön egy új Pose objektum, mint csúcs, ezt követően pedig egy map konténerbe felvezetjük az előző Pose és a jelenlegi Pose közötti mérést, mint kapcsolatot, ez lesz a két Pose közötti él.

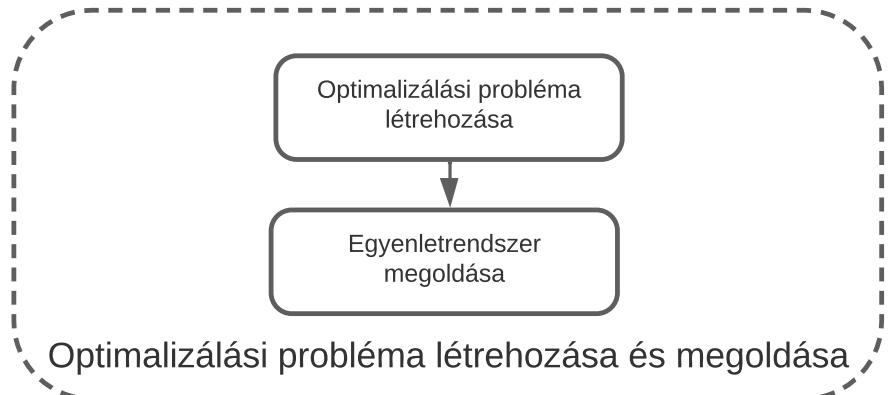
Érzékelés esetén nem minden esetben kell új Landmark objektumot létrehoznunk, ugyanis gyakran előfordul, hogy egy Landmark több érzékelési adatban is szerepel. Ilyenkor csak első alkalommal kell az objektumot létrehoznunk csúcsként és felvezetnünk a jelenlegi Pose és érzékelt Landmark közötti kapcsolatot, ezt követően már csak a Landmark-Pose élek létrehozásával kell foglalkoznunk.



4.11. ábra. Csúcsok és élek létrehozása

Optimalizálási probléma létrehozása és megoldása

A szoftver ezen részének a feladata, hogy az előzőleg összegyűjtött adatokból - a csúcsokból és az élekből - összeállítsa a megoldandó egyenletrendszeret. Miután az egyenletrendszer előállt elkezdődhet négyzetes hibák minimalizálása a hibafüggvények segítségével. Az első Pose csúcsot az optimalizálás előtt lefixáljuk, hogy minden minimalizálás ahhoz képest történjen.



4.12. ábra. Optimalizálási probléma létrehozása és megoldása

4.2.2. Fordítási útmutató

A program bármilyen operációs rendszeren fordítható, amelyen megtalálható valamelyen C++20 kompatibilis fordító, preferáltan Clang. A program fordításához

szükséges a fentebb említett függőségek telepítése is, mely folyamatban a Python3 egyszerűen beszerezhető egy csomagkezelővel Unix alapú operációs rendszerek alatt. A Matplotlib telepítéséhez ajánlatos a Python csomagkezelőjét, a *pip*-et használni. A Ceres Solver telepítéséhez a Microsoft *vcpkg*¹ csomagkezelőjét a legkézenfekvőbb használni, de magunk is lefordíthatjuk a forráskódot, viszont figyeljünk arra, hogy bármelyik megoldást választjuk, a Ceres Solver függőségei is legyenek feltelepítve.

A szoftver fordítása normál használathoz:

```
mkdir build  
cd build  
cmake ..  
make -j
```

(4.9)

A szoftver fordítása teszteléshez:

A teszteléshez szükséges binárisok normál esetben is lefordulnak, viszont csak a COVERAGE makró segítségével tudunk lefedettségi adatokat generálni a binárisokhoz.

```
mkdir build  
cd build  
cmake -DCOVERAGE=1 ..  
make -j
```

(4.10)

4.2.3. Továbbfejlesztési lehetőségek

Adatasszociáció

A dolgozatom megírása során arra törekedtem, hogy az adatasszociáció adott legyen a szimulációs környezet kimenetében, így a SLAM számára is, viszont az adatasszociáció problémájának megoldása elengedhetetlen a valóéletben történő egyidejű helymeghatározáshoz és térképezéshez.

Adatsúlyozás

Az optimalizációt végző algoritmus jelenleg minden adatot azonos fontossággal kezel, azonban érdemes lehet a felhasználó számára biztosítani lehetőséget arra, hogy

¹<https://vcpkg.io/en/getting-started.html>

a különböző típusú adatokat más súllyal lásson el, ezzel is növelve az optimalizáció hatékonyságát.

Grafikus kezelőfelület

A parancssorból meghívandó bináris jelenleg egy konzol alkalmazás, ezért érdemes lehet egy felhasználóbarátabb grafikus kezelőfelületet megvalósítani hozzá.

Lineáris egyenletrendszer megoldók

Jelenleg a felhasználó számára nincs lehetőség a LER megoldók változtatására, viszont érdemes lehet egy ilyen funkcionálitás kifejlesztése.

Többszálúsítás

Jelenleg a szoftver architektúrája nem támogatja a többszálú megvalósítást, azonban amennyiben nem egy másik SLAM algoritmussal szeretnénk együtt használni, érdemes lehet az adatfeldolgozást és az optimalizációt külön szálakon futtatnunk, ezzel egy valósidejű megoldást adva a felhasználó kezébe.

Véletlenszám generátor seed

A szoftver jelenleg, amennyiben nem használjuk a tényleges véletlenszám generáló kapcsolót, beégetett seed alapján dolgozik. Érdemes lehet engedélyezni a felhasználó számára a seed állítását, ezzel is több lehetőséget biztosítva a reprodukálható teszteléshez.

Vizualizáció

Az adatok vizualizálására használt Matplotlib több, mint elég, ha az adatokat nem valós időben szeretnénk folyamatosan megjeleníteni. A valós idejű ábrázoláshoz azonban szükséges egy saját vizualizációs eszközt kifejleszteni.

4.3. A fejlesztés főbb mérföldkövei

A fejlesztési időszakot a **szimulációs környezet** kialakításával kezdtem. Elsősorban meg kellett oldanom, hogy a bemeneti fájlokban szereplő pontok koordinátájának beolvasása után, tekintve, hogy ezek ritka pályapontok, interpoláció

segítségével egy sűrűbb középvonalat kapjak. Ezzel párhuzamosan a **vizualizáció**n kezdtem el dolgozni, hogy validálni tudjam az addig feldolgozott adatokat. A sűrű középvonal megléte után a *pálya két szélét jelző bóják legenerálása* volt a következő teendő, ami igazából az eredeti ritka pályapontok eltolását és két újabb interpolációt igényelt. Ezt követően szükségem volt egy algoritmusra, amely egy konstans sebesség és mintavételezési idő alapján a pálya középvonalából, továbbá a bóják sűrű vonalából fix távolság alapján pontokat választ ki. Mivel konstans sebességgel és fix távolsággal már működött a kiválasztó algoritmus, elkezdtem átalakítani, hogy a pálya görbületét az adott pontokban figyelembe véve, változó *sebesség* és *bójatávolság* adatokkal is működjön. Ekkor a szimulációs környezet nagyjából már körvonalazódni látszott, úgyhogy az egyszerű használatához parancssori argumentumkezelést valósítottam meg. Utolsó lépésként pedig az **érzékelés szimulálására** kellett egy algoritmust kifeljeszteni, illetve a kimeneti fájl formátumát kitalálni és az alapján kimenteni a szimulációból kapott adatokat a SLAM szoftver számára.

A szimulációs környezet elkészültét követően a **SLAM szoftver implementációja** következett. Első lépésként szükségem volt egy módszerre, amivel az adatokat egyszer olvasva tudom feldolgozni őket. Természetesen a tanulmányaim során is használt Gregorics Tibor tanár úr osztály-sablon könyvtárához nyúltam ihlet után és így jött létre a saját felsoroló osztályom. Ezt követően a **numerikus optimalizációhoz szükséges típusokat**, majd a **gráf struktúrát megvalósító osztályok** implementálása következett. Az optimalizáció megvalósításához egyre közeledvén elkezdtem az addig létrejött osztályok fehérdoboz tesztelését, illetve ezzel párhuzamosan az **optimalizációhoz szükséges hibafüggvények** szerepét ellátó funktorok megtervezését. A hibafüggvények elkészültével kezdetét vette az optimalizációért felelő metódus megírása. Miután ez is elkészült, még több **tesztelés** és **hibakeresés** következett. Mivel a SLAM szoftver nagyjából már elkészült, hasonlóan a szimulációs környezethez a szoftver egyszerű használatához itt is *parancssori argumentumkezelést* valósítottam meg. Először bár nem volt beépített vizualizáció az alkalmazásban, sikerült végül a szimulációs környezetben is használt **vizualizációt** működésre bírni. Utolsó lépésként a fentebb említett felsoroló átdolgozása következett, hogy képes legyen opcionálisan zajosítani a bemeneti sebességeket.

4.3.1. Programozástechnikai megfontolások

- C++20 - legfrissebb stabil kiadás használata.
- Smart pointer - a dinamikus memóriakezelést hivatott megkönnyíteni, ugyanis nincs szükség a mutatók által mutatott memória felszabadítására, mivel ez automatikusan megtörténik, amikor egy mutató hatásköre véget ér.
- CMake - szép és átlátható kódstruktúra, statikus könyvtárak létrehozása és egyszerű fehérdoboz tesztelés.

4.3.2. Kompromisszumok

- A kézenfekvőbb fejlesztői eszközök miatt először Python-ban kezdődött a SLAM szoftver fejlesztése, azonban a későbbi valós hardveren való felhasználása miatt elengedhetetlen volt a jó futásidő, így végül C++-ban valósult meg.
- A SLAM szoftver bementéül végül nem x , y (pozíció) és ψ (elfordulás) adatok változása került bevitelre, hanem a valós hardvert figyelembe véve sebesség és szögsebesség adatok.
- Időhiány miatt sajnos csak az odometriai adatok zajosítása valósult meg.

5. fejezet

Tesztelés

A szoftverek *Ubuntu 20.04*, *Fedora 32*, *Mint 20* és *macOS 11.6.5* operációs rendszereken lettek tesztelve a következő hardverspecifikációjú számítógépen:

Processzor	Intel Core i7 - 2.6 Ghz
RAM	32 GB
Videokártya	Intel UHD Graphics 630
Képernyő	3072 x 1920

5.1. Fehérdobozos tesztelés

A fehérdobozos tesztelés során használt automatikus tesztek a Catch¹ keretrendszerrel lettek megvalósítva. A szoftver fordításakor a *build/src* könyvtár alatt létrejön egy-egy külön alkönyvtár a fontosabb komponenseknek, melyeken belül megtalálható egy *test* mappa a komponenshez tartozó futtatható tesztállománnyal.

5.1.1. Numerikus eljárásokhoz szükséges osztályok tesztelése

Megjegyzés. A [types] címkével ellátott tesztesetekben az objektum létrehozása paraméterek nélkül alapértelemezzen minden adattagot 0-ra állít, kivéve az *id* adattagokat, melyek a –1 értéket veszik fel.

Test Landmark [types]

- Landmark objektum létrehozása paraméterek nélkül
- Landmark objektum létrehozása paraméterekkel

¹<https://github.com/catchorg/Catch2>

Test Motion [types]

- Motion objektum létrehozása paraméterek nélkül
- Motion objektum létrehozása paraméterekkel

Test Perception [types]

- Perception objektum létrehozása paraméterek nélkül
- Perception objektum létrehozása paraméterekkel

Test Pose [types]

- Pose objektum létrehozása paraméterek nélkül
- Pose objektum létrehozása paraméterekkel
- operator+ használata paraméterek nélkül létrehozott Motion objektummal
- operator+ használata paraméterekkel létrehozott Motion objektummal
- operator+= használata paraméterek nélkül létrehozott Motion objektummal
- operator+= használata paraméterekkel létrehozott Motion objektummal

Test RotationMatrix2D [utility]

- Egységmátrix theta = 0 esetén
- Mátrix theta = $\frac{\pi}{2}$ esetén

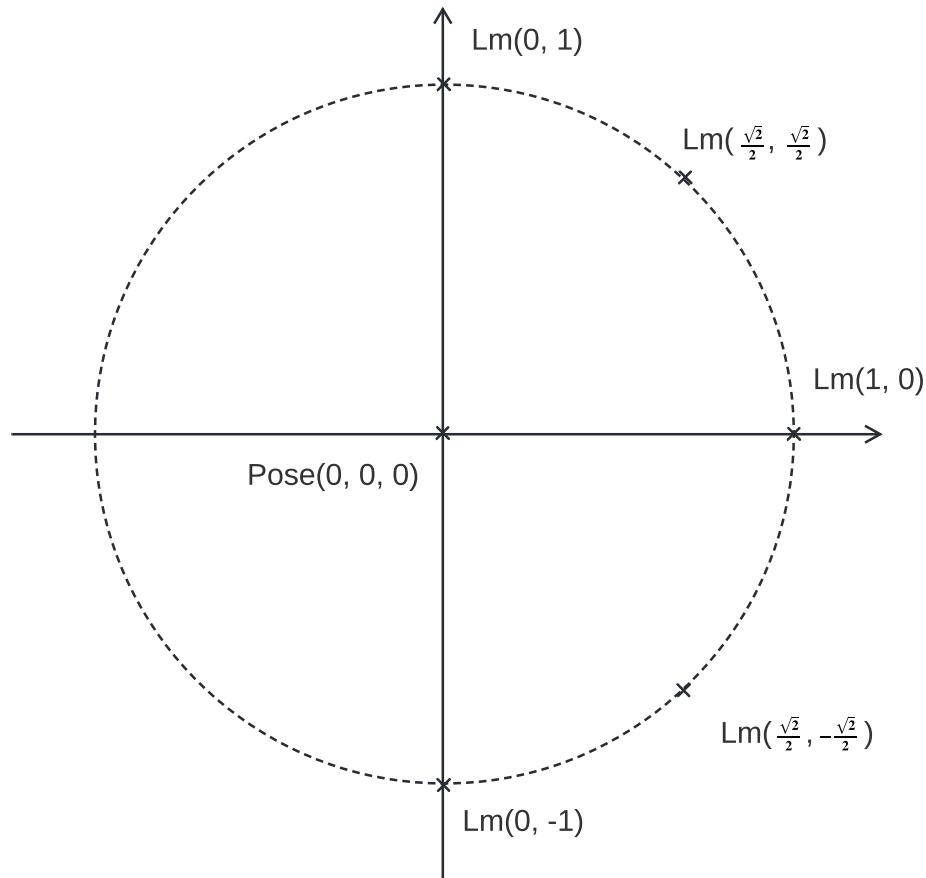
$$M = \begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix} \quad (5.1)$$

- Mátrix theta = $-\frac{\pi}{2}$ esetén

$$M = \begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix} \quad (5.2)$$

Test LandmarkErrorFunction [utility]

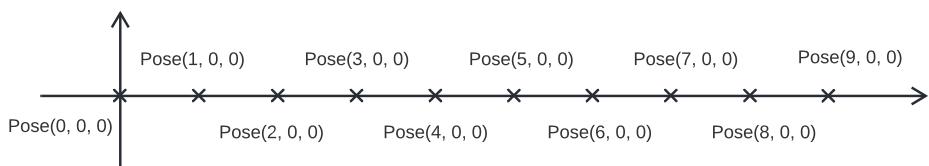
- Tereptárgyak érzékelése az egységkör körvonalán Pose(0, 0, 0) esetén



5.1. ábra. A járműünk a koordináta-rendszer origójában helyezkedik el 0° -os orientációval, az érzékelendő tereptárgyak pedig az egységkör körvonalán találhatók a feltüntetett pozíciókban.

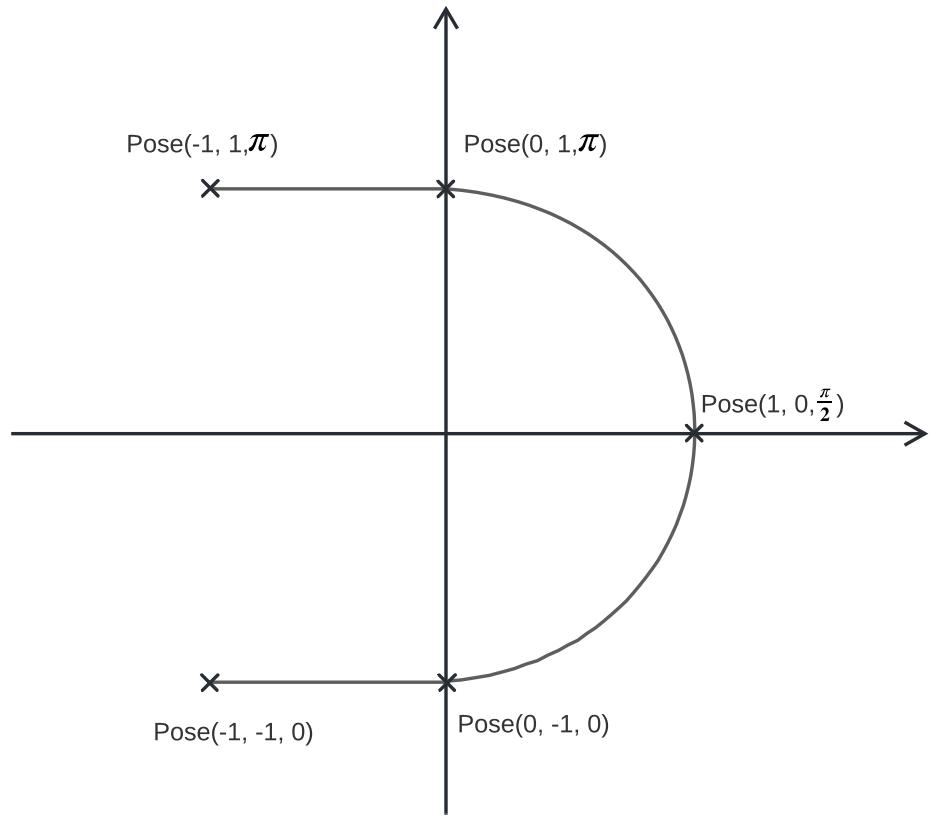
Test PoseErrorFunction [utility]

- Egyenes vonalú mozgás konstans 1 m/s sebességgel és 0 rad/s szögsebességgel



5.2. ábra. A járműünk kezdetben a koordináta-rendszer origójában helyezkedik el 0° -os orientációval. Konstans 1 m/s sebességgel és 0 rad/s szögsebességgel a tesztet végére a $(9, 0)$ pontba jut el, továbbra is 0° -os orientációval.

- Görbe vonalú mozgás konstans 1 m/s sebességgel és változó szögsebességgel



5.3. ábra. A járműünk kezdetben a $(-1, -1)$ pontban helyezkedik el 0° -os orientációval. A teszeset során az ábrán látható útvonalat járja be és a végére a $(-1, 1)$ pontba jut el 180° -os orientációval.

5.1.2. DataEnumerator osztály tesztelése

Megjegyzés. A Test Data teszesetben az objektum létrehozása paraméterek nélkül alapértelemezzen minden adattagot 0-ra állít.

Test Data [dataenumerator]

- Data objektum létrehozása paraméterek nélkül
- Data objektum létrehozása paraméterekkel
- Landmark azonosítók vizsgálata üres Perception vector esetén
- Landmark azonosítók vizsgálata nem üres Perception vector esetén

Test DataEnumerator [dataenumerator]

- DataEnumerator objektum létrehozása nem létező fájl esetén
- DataEnumerator objektum létrehozása létező üres fájl esetén
- Üres fájlból való olvasás
- Zaj nélkül
 - Data(1 Motion, 2 Perception) olvasása nem üres fájlból
 - 2 Data(1 Motion, 2 Perception) olvasása nem üres fájlból
 - Kizárolag Motion adatok olvasása nem üres fájlból
- Zajjal
 - Normál eloszlású zaj random seed-del
 - Normál eloszlású zaj pszeudorandom seed-del
 - Data(1 Motion, 2 Perception) olvasása nem üres fájlból
 - 2 Data(1 Motion, 2 Perception) olvasása nem üres fájlból
 - Kizárolag Motion adatok olvasása nem üres fájlból

5.1.3. Graph osztály tesztelése

Test Graph [graph]

- Graph objektum létrehozása
- Landmark objektum létrehozása a gráfba
- Pose objektum létrehozása a gráfba Motion(0, 0) esetén
- Pose objektum létrehozása a gráfba Motion(1, 1) esetén
- A gráf optimalizálása

5.2. Kiértékelés

A kiértékelés során 3 térképet fogunk vizsgálni:

- track
- FSG21 - a Formula Student Germany² 2021-ben megrendezett versenysorozat önvezető kategórijának *autocross*, illetve *track drive* versenyszámokban³ használt pályája
- kecso - a kecskeméti gokartpálya⁴ egy része

5.2.1. Zaj nélkül

Az alábbi SLAM futások a *--noise* kapcsoló használata nélkül kerültek kiértékelésre.

Hurokzárás nélkül

A SLAM ezen futása során nem használva a *--loop_closure* kapcsolót az alábbi statisztikákat kapjuk. A hurokzárásról és hatásáról a 3.4-ben olvashat.

	track	FSG21	kecso
Pose-ok száma	591	1321	1922
Landmark-ok száma	64	114	174
Csúcsok és élek létrehozása	~60 ms	~89 ms	~115 ms
Gráf felépítése	~10 ms	~22 ms	~27 ms
Gráf optimalizálása	~39 ms	~62 ms	~110 ms

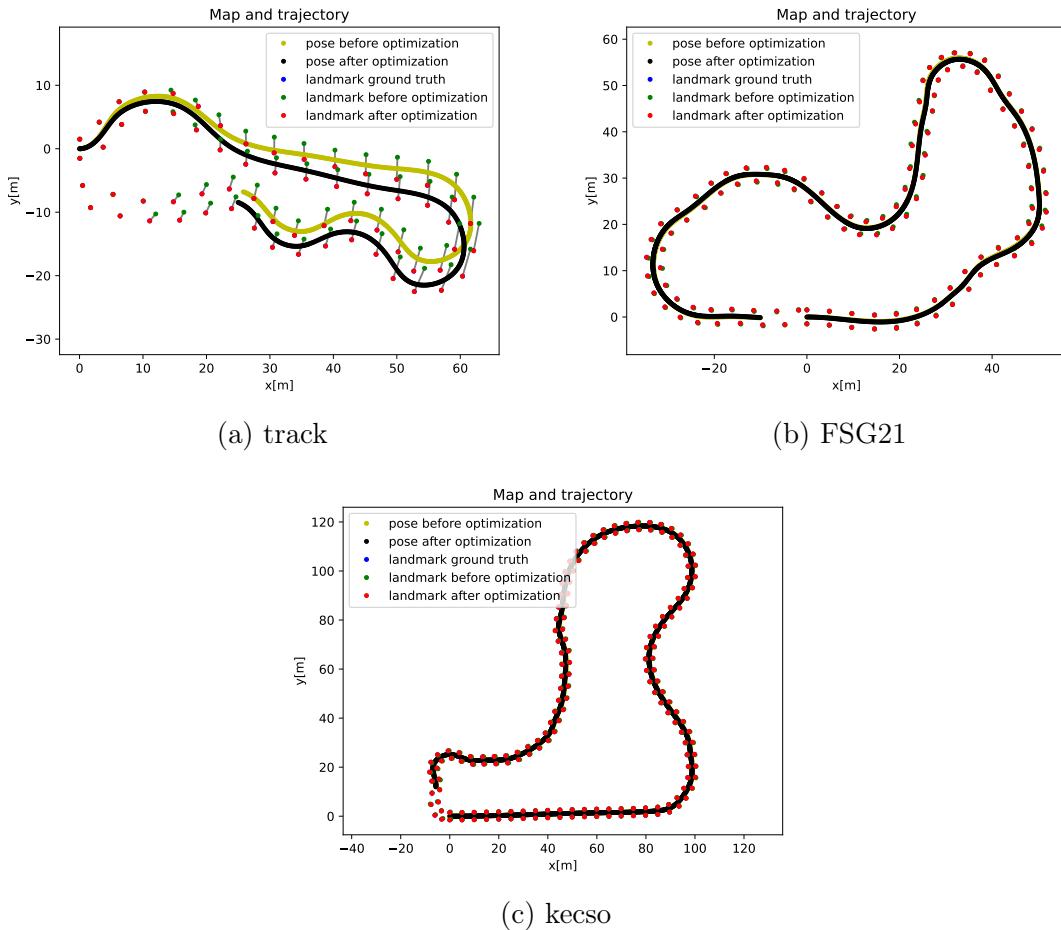
5.1. táblázat. Zajmentes esetek kiértékelése hurokzárás (3.4) nélkül

A vizualizáció során megfigyelhetjük, hogy a fekete pontsorozat, mely a jármű útvonalát jelzi, nem ér össze, ugyanis ha nem engedélyezzük a SLAM számára a hurokzárást, akkor az útvonal azon pontján megszakad az adatfelvétel (4.2.1), ahol az érzékelésben már korábban látott bójákkal találkozunk.

²<https://www.formulastudent.de/fsg/>

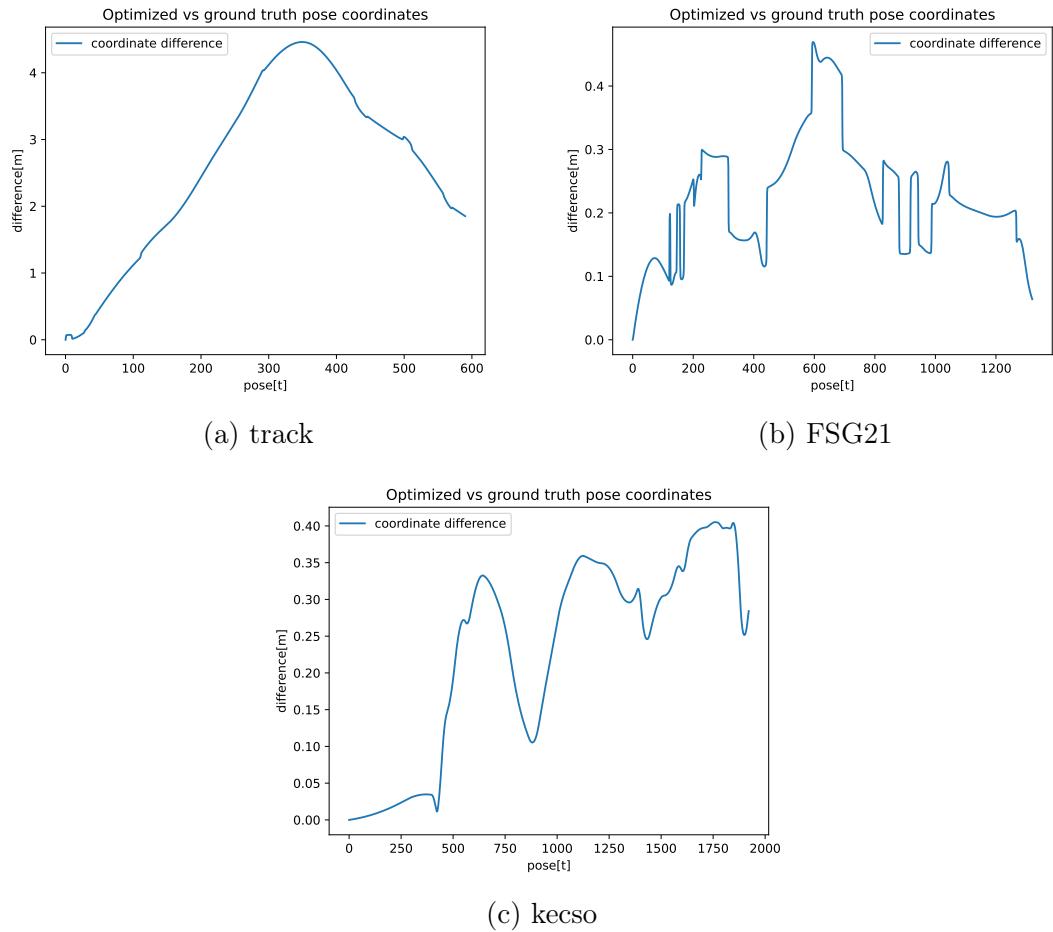
³<https://www.formulastudent.de/about/disciplines/>

⁴<https://www.birizdokart.hu/hu/a-palya>



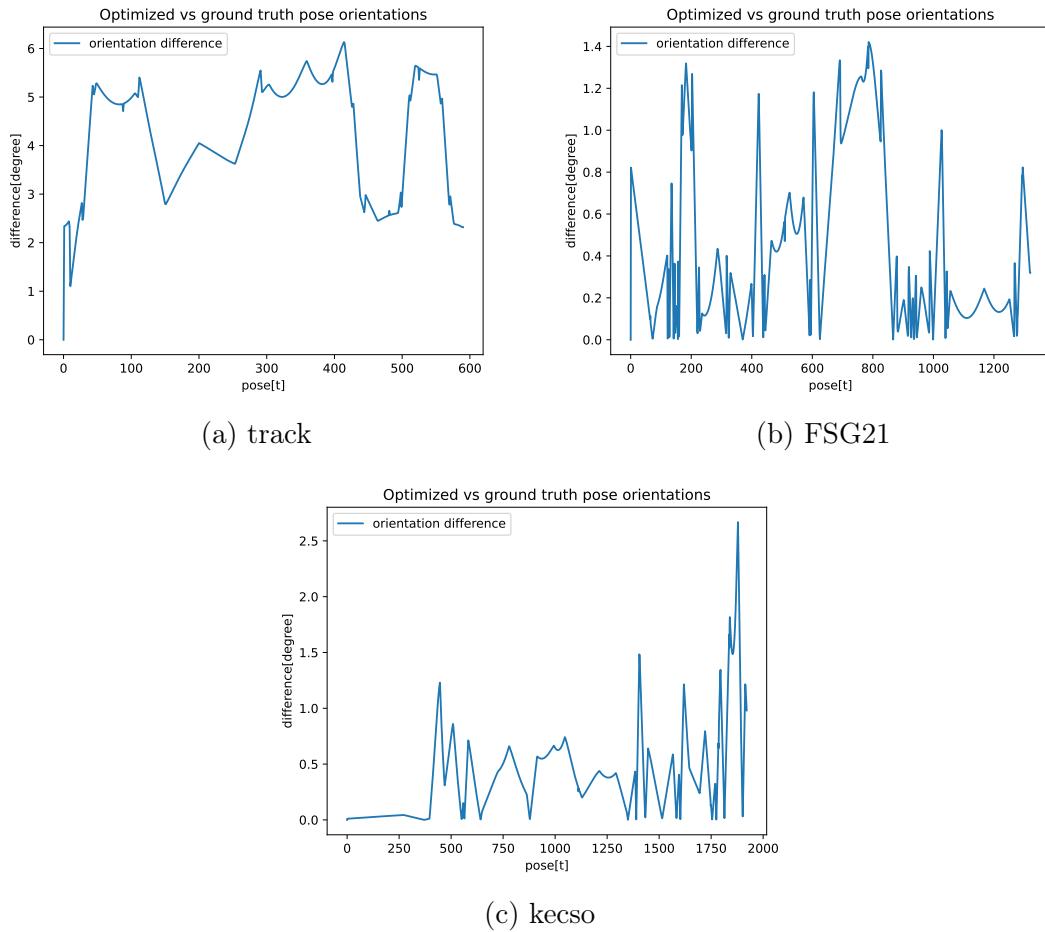
5.4. ábra. Zajmentes esetek vizualizációja hurokzárás nélkül

Az alábbi ábrákon megfigyelhetjük, hogy a különböző térképeken milyen pozícióbeli különbségek születtek az optimalizált, illetve a ground truth járműpózok között.



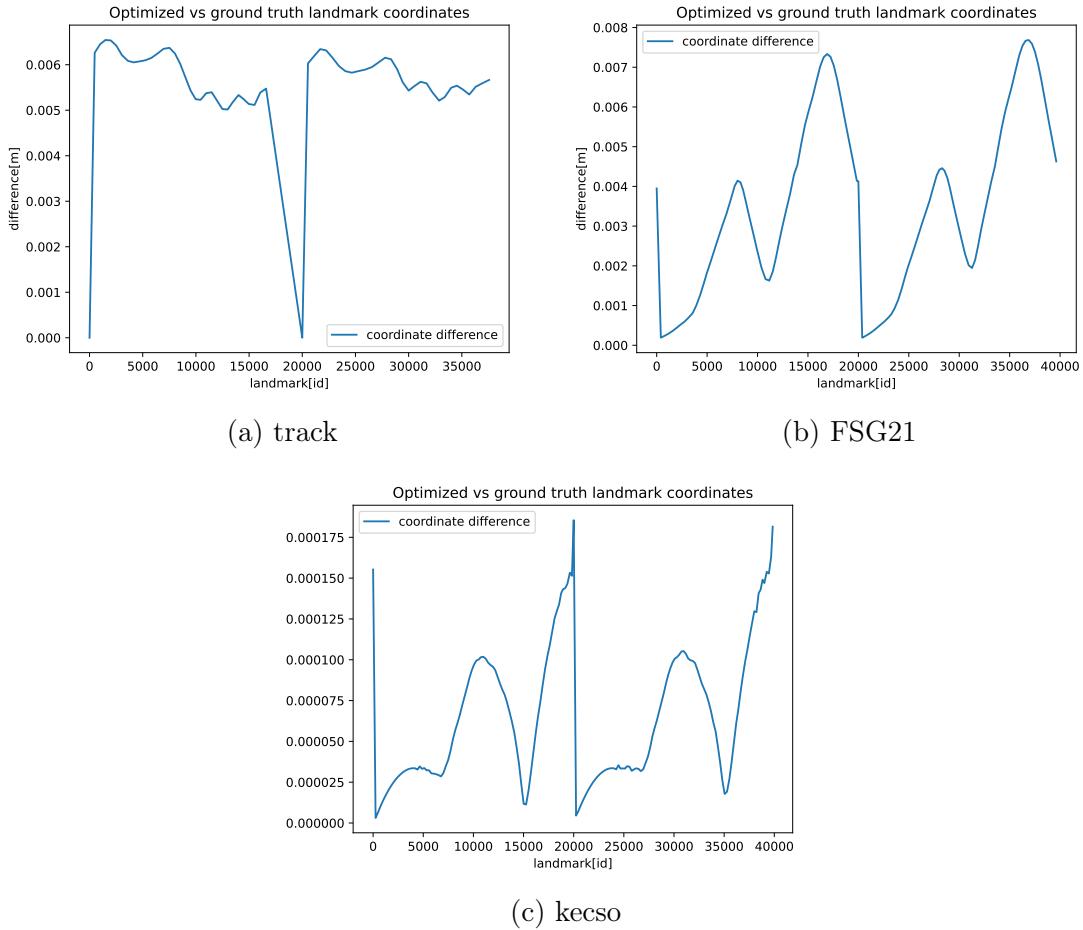
5.5. ábra. Optimalizált és ground truth pozíció koordináták különbsége - zajmentes eset lévén ez az egyik támponk a járműpóz koordináták kiértékelése során.

Az alábbi ábrákon megfigyelhetjük, hogy a különböző térképeken milyen orientációjú különbségek születtek az optimalizált, illetve a ground truth járműpózok között.



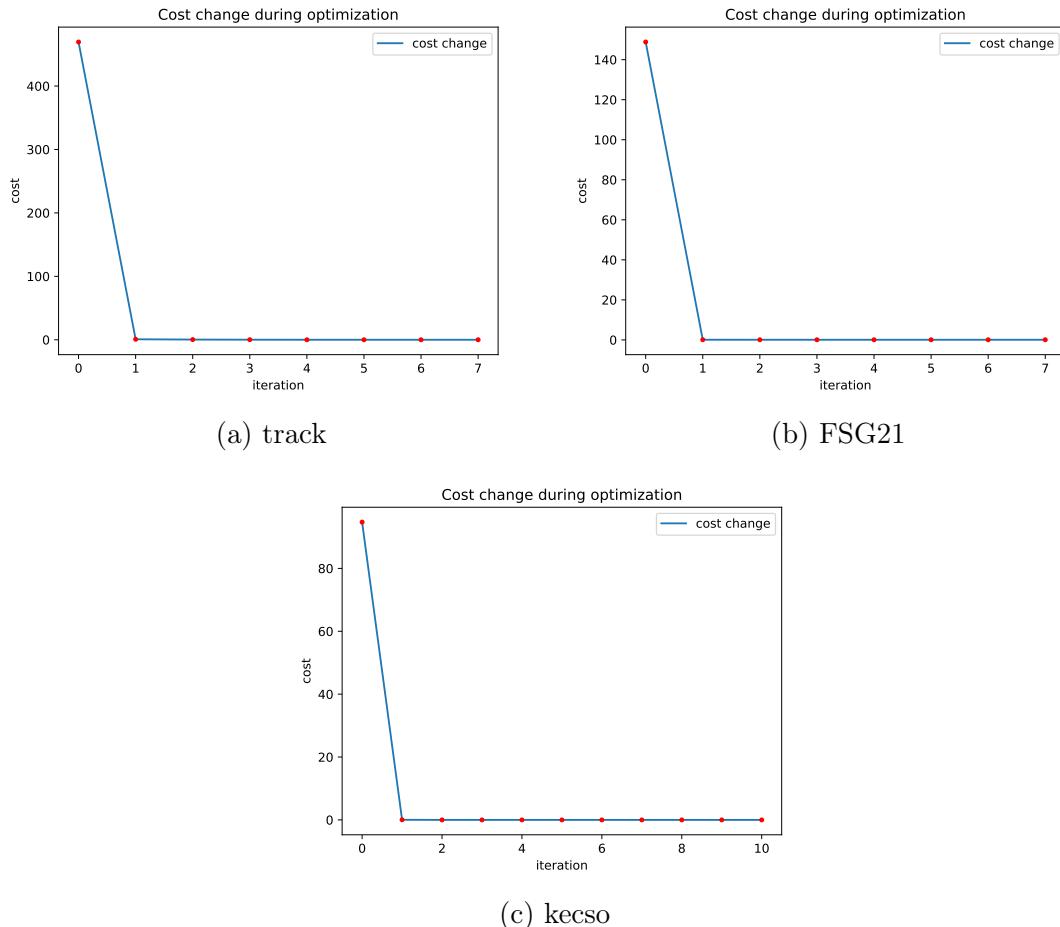
5.6. ábra. Optimalizált és ground truth pozíciók különbsége - zajmentes eset
lévén ez az egyik támpontunk a járműpóz orientációk kiértékelése során.

Az alábbi ábrákon megfigyelhetjük, hogy a különböző térképeken milyen pozícióból különbségek születtek az optimalizált, illetve a ground truth tereptárgyak között.



5.7. ábra. Optimalizált és ground truth tereptárgy koordináták különbsége

Az alábbi ábrákon megfigyelhetjük, hogy a SLAM algoritmus különböző térképeken hány iteráció alatt és milyen költséggel konvergált.



5.8. ábra. Hibafüggvények költségének csökkenése az optimalizáció során

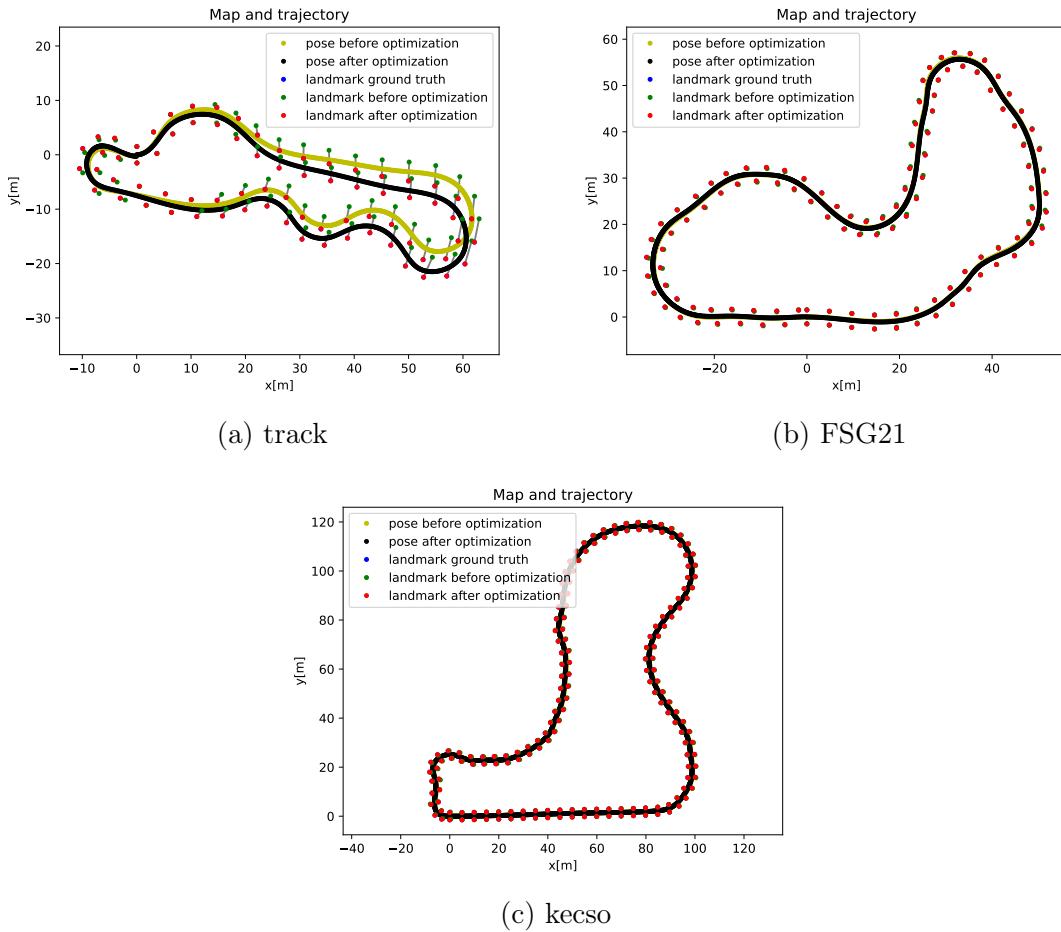
Hurokzárással

A SLAM ezen futása során használva a `--loop_closure` kapcsolót az alábbi statisztikákat kapjuk:

	track	FSG21	kecso
Pose-ok száma	843	1362	2001
Landmark-ok száma	74	114	174
Csúcsok és élek létrehozása	~60 ms	~89 ms	~118 ms
Gráf felépítése	~16 ms	~22 ms	~30 ms
Gráf optimalizálása	~53 ms	~60 ms	~106 ms

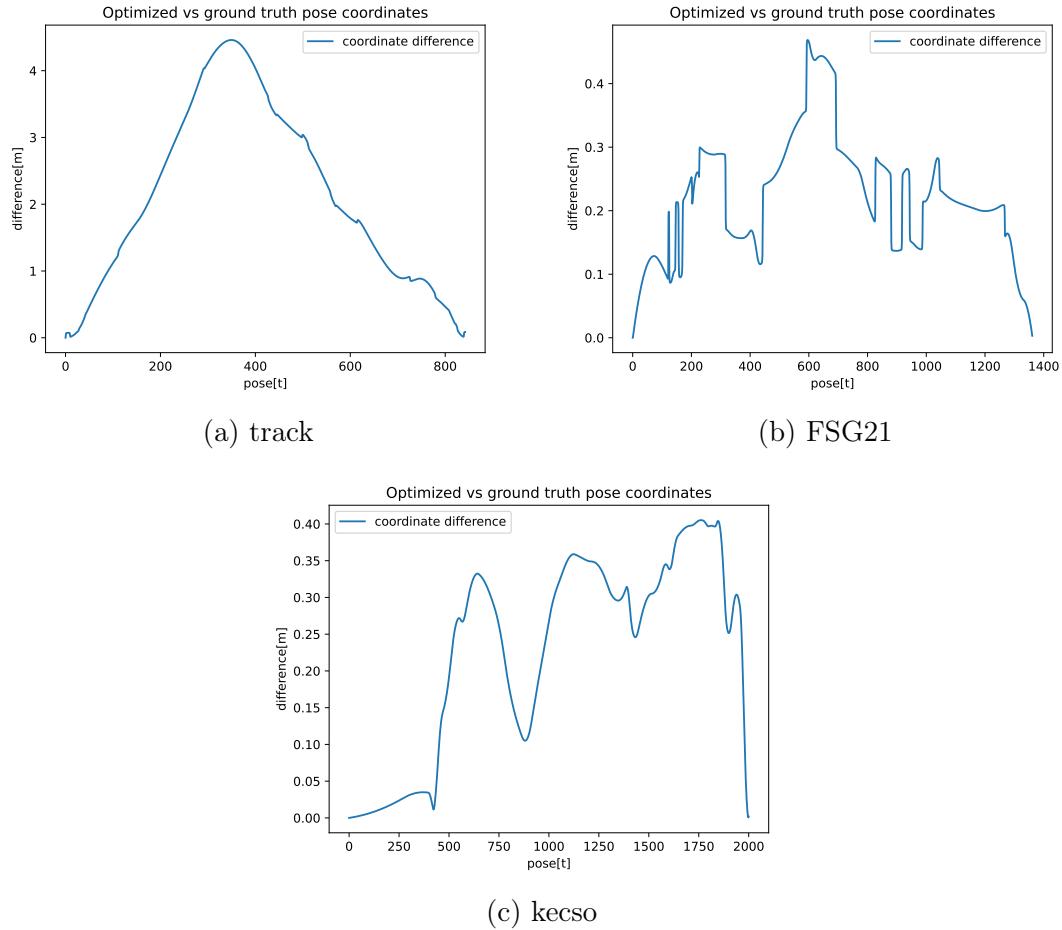
5.2. táblázat. Zajmentes esetek kiértékelése hurokzárással

A vizualizáció során megfigyelhetjük, hogy a fekete pontsorozat, mely a jármű útvonalát jelzi, összeér, ha engedélyezzük a SLAM számára a hurokzárást, mivel az adatfelvétel egészen addig folytatódik, amíg van feldolgozandó mérés.



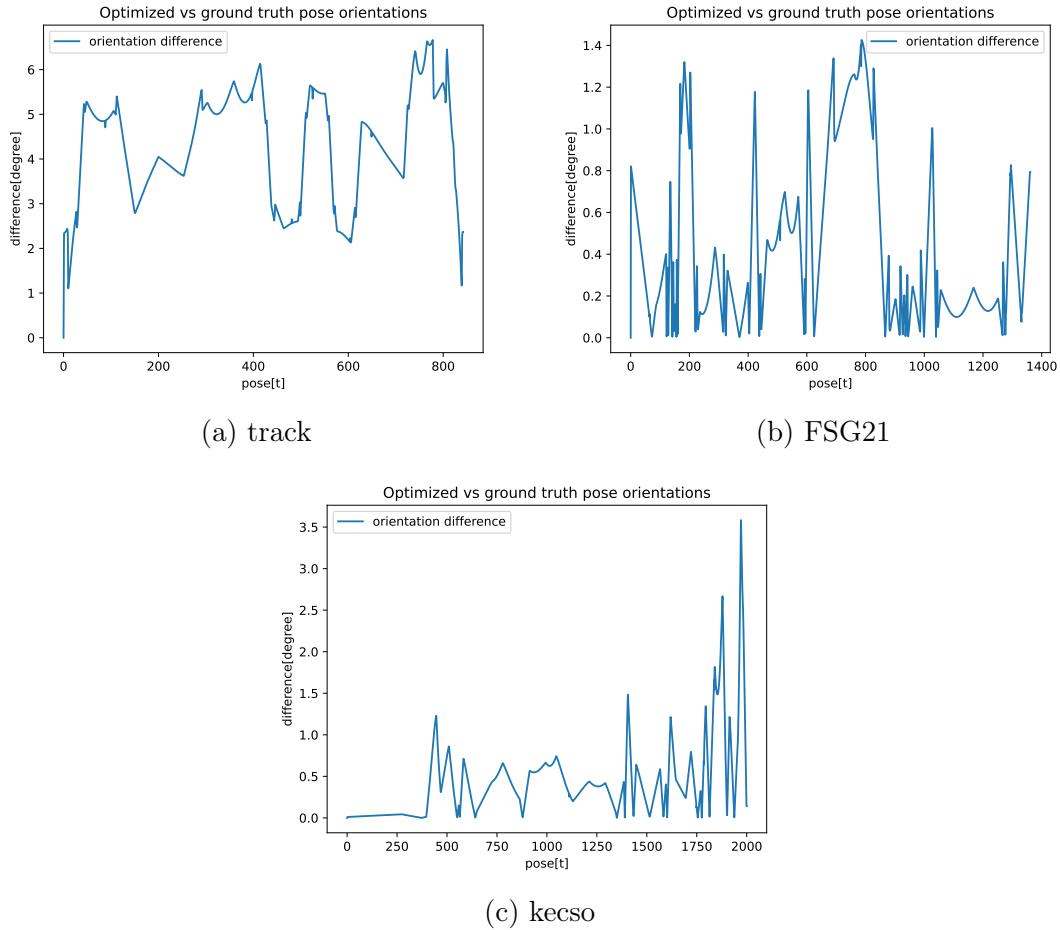
5.9. ábra. Zajmentes esetek vizualizációja hurokzárással

Az alábbi ábrákon megfigyelhetjük, hogy a különböző térképeken milyen pozícióbeli különbségek születtek az optimalizált, illetve a ground truth járműpózok között.



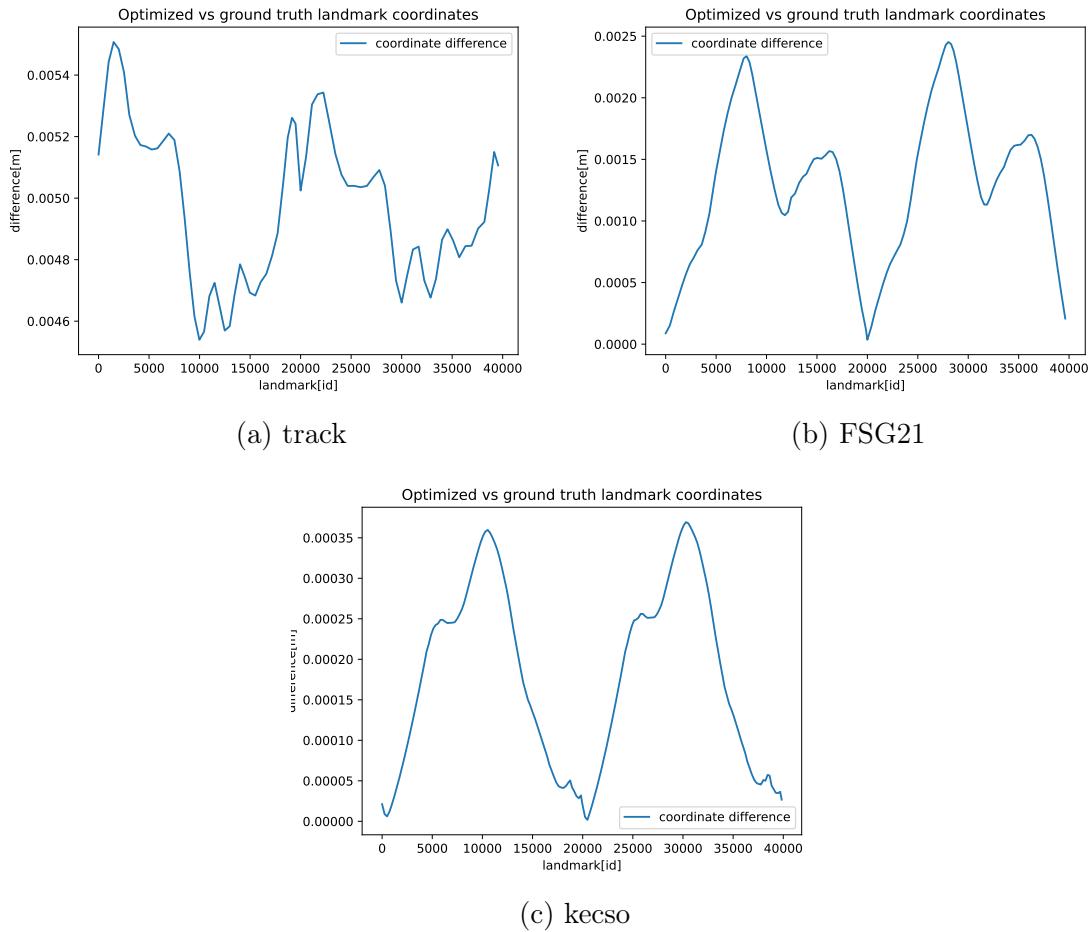
5.10. ábra. Optimalizált és ground truth pozíció koordináták különbsége - zajmentes eset lévén ez a másik támpontunk a járműpóz koordináták kiértékelése során.

Az alábbi ábrákon megfigyelhetjük, hogy a különböző térképeken milyen orientációjú különbségek születtek az optimalizált, illetve a ground truth járműpózok között.



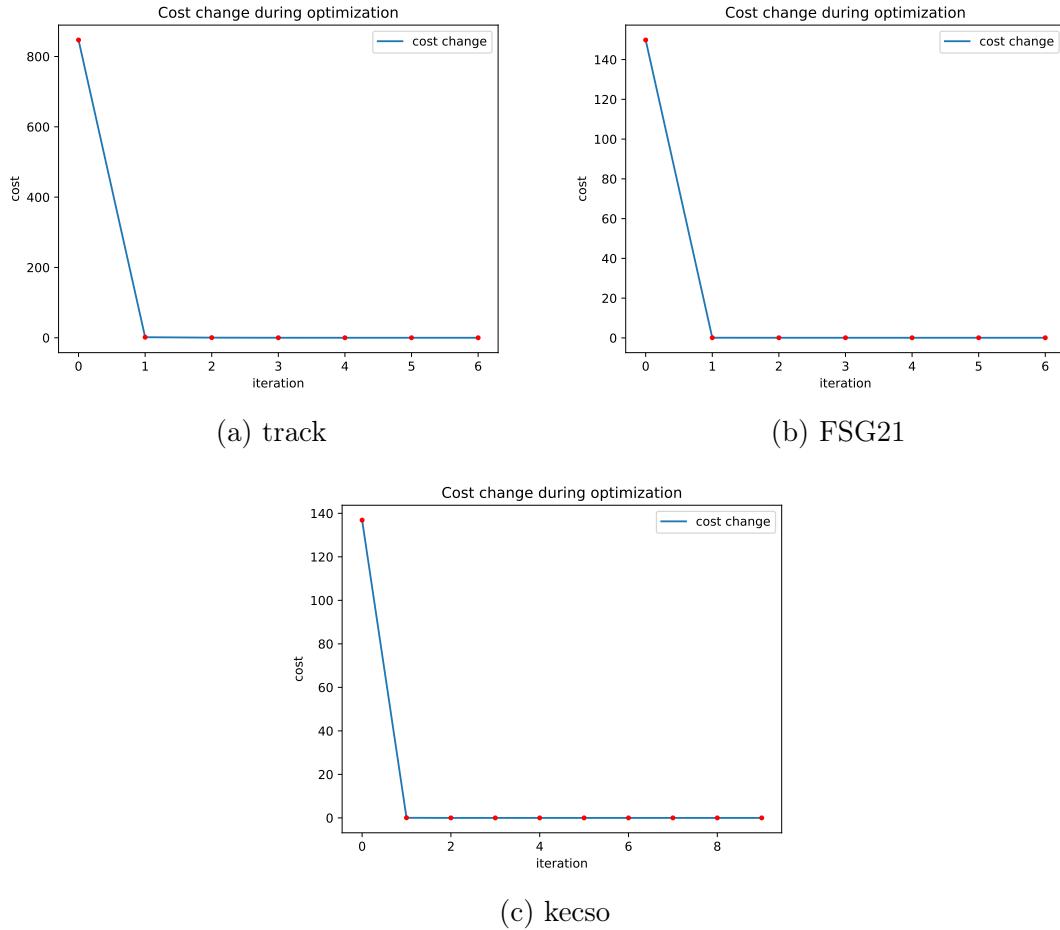
5.11. ábra. Optimalizált és ground truth poz orientációk különbsége - zajmentes eset lévén ez a másik támpontunk a járműpoz orientációk kiértékelése során.

Az alábbi ábrákon megfigyelhetjük, hogy a különböző térképeken milyen pozícióbeli különbségek születtek az optimalizált, illetve a ground truth tereptárgyak között.



5.12. ábra. Optimalizált és ground truth tereptárgy koordináták különbsége

Az alábbi ábrákon megfigyelhetjük, hogy a SLAM algoritmus különböző térképeken hány iteráció alatt és milyen költséggel konvergált.



5.13. ábra. Hibafüggvények költségének csökkenése az optimalizáció során

5.2.2. Zajjal

Az alábbi SLAM futások a `--noise` kapcsolót használva 0.5-ös, illetve 5-ös értékkel kerültek kiértékelésre.

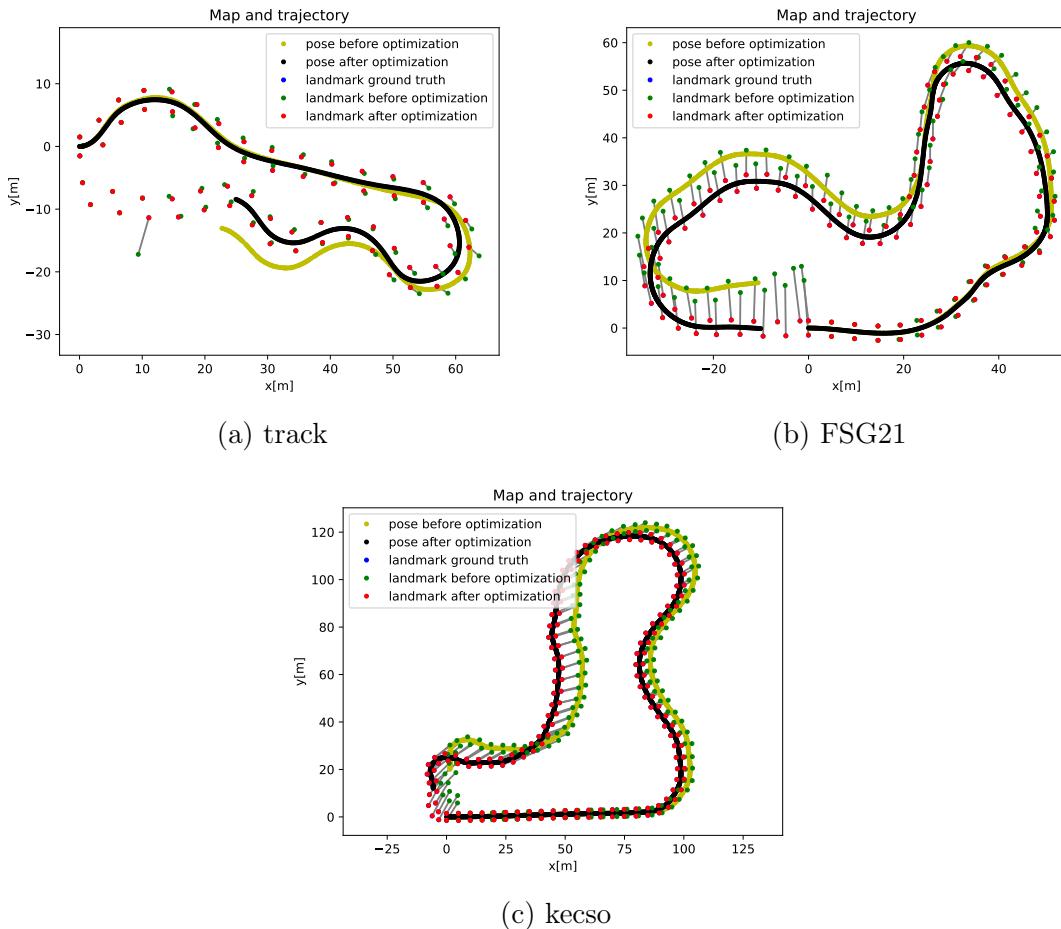
Hurokzárás nélkül

A SLAM ezen futása során nem használva a `--loop_closure` kapcsolót az alábbi statisztikákat kapjuk:

	track	FSG21	kecso
Pose-ok száma	591	1321	1922
Landmark-ok száma	64	114	174
Csúcsok és élek létrehozása	~60 ms	~89 ms	~118 ms
Gráf felépítése	~15 ms	~22 ms	~30 ms
Gráf optimalizálása	~31 ms	~88 ms	~142 ms

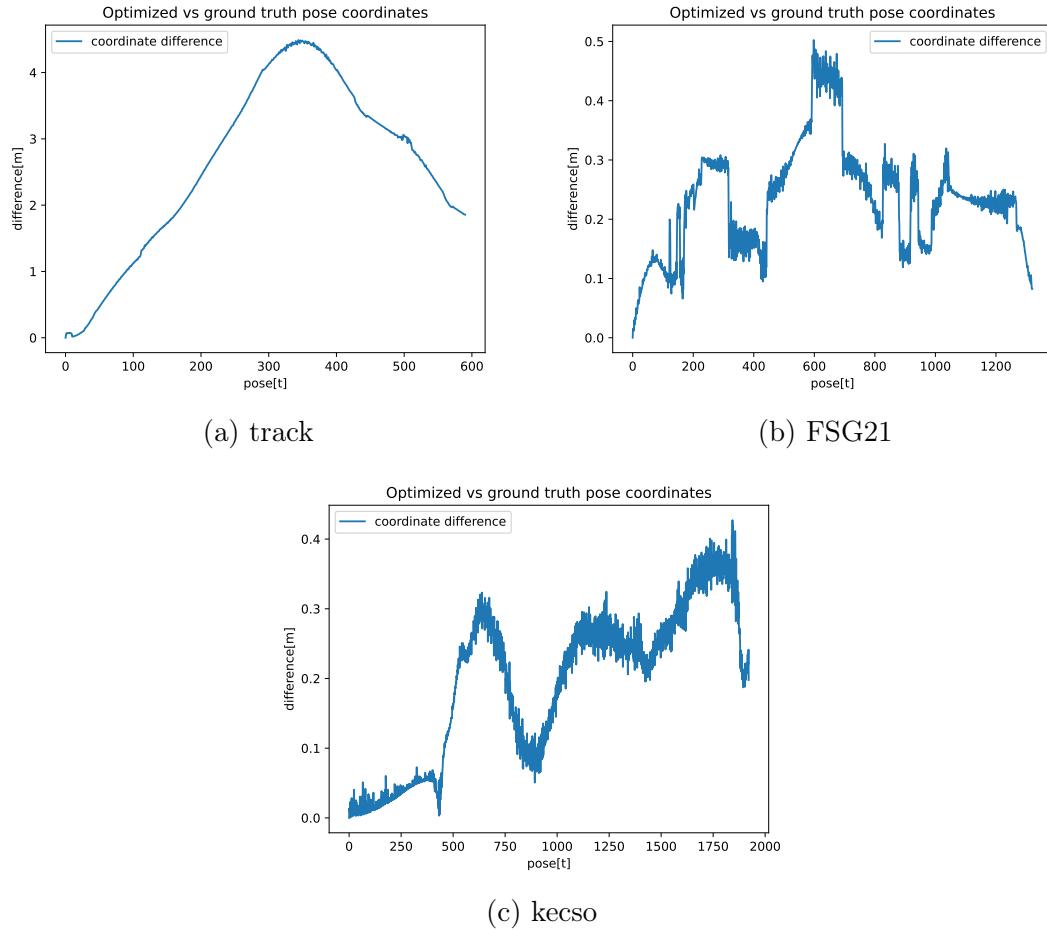
5.3. táblázat. Zajos esetek kiértékelése hurokzárás nélkül; $\sigma = 0.5$

A vizualizáció során megfigyelhetjük, hogy a fekete pontsorozat, mely a jármű útvonalát jelzi, nem ér össze, ugyanis ha nem engedélyezzük a SLAM számára a hurokzárást, akkor az útvonal azon pontján megszakad az adatfelvétel, ahol az érzékelésben már korábban látott bójákkal találkozunk.



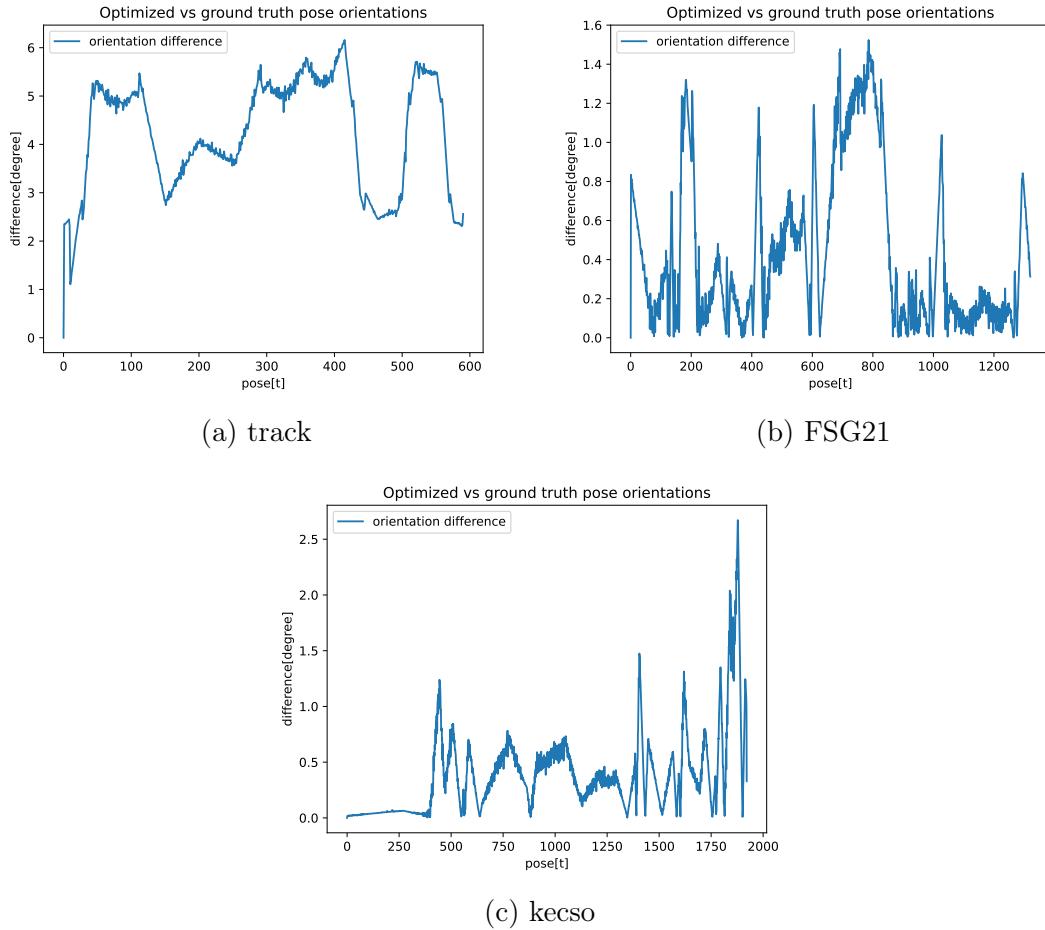
5.14. ábra. Zajos esetek vizualizációja hurokzárás nélkül; $\sigma = 0.5$

Az alábbi ábrákon megfigyelhetjük, hogy a különböző térképeken milyen pozícióbeli különbségek születtek az optimalizált, illetve a ground truth járműpózok között.



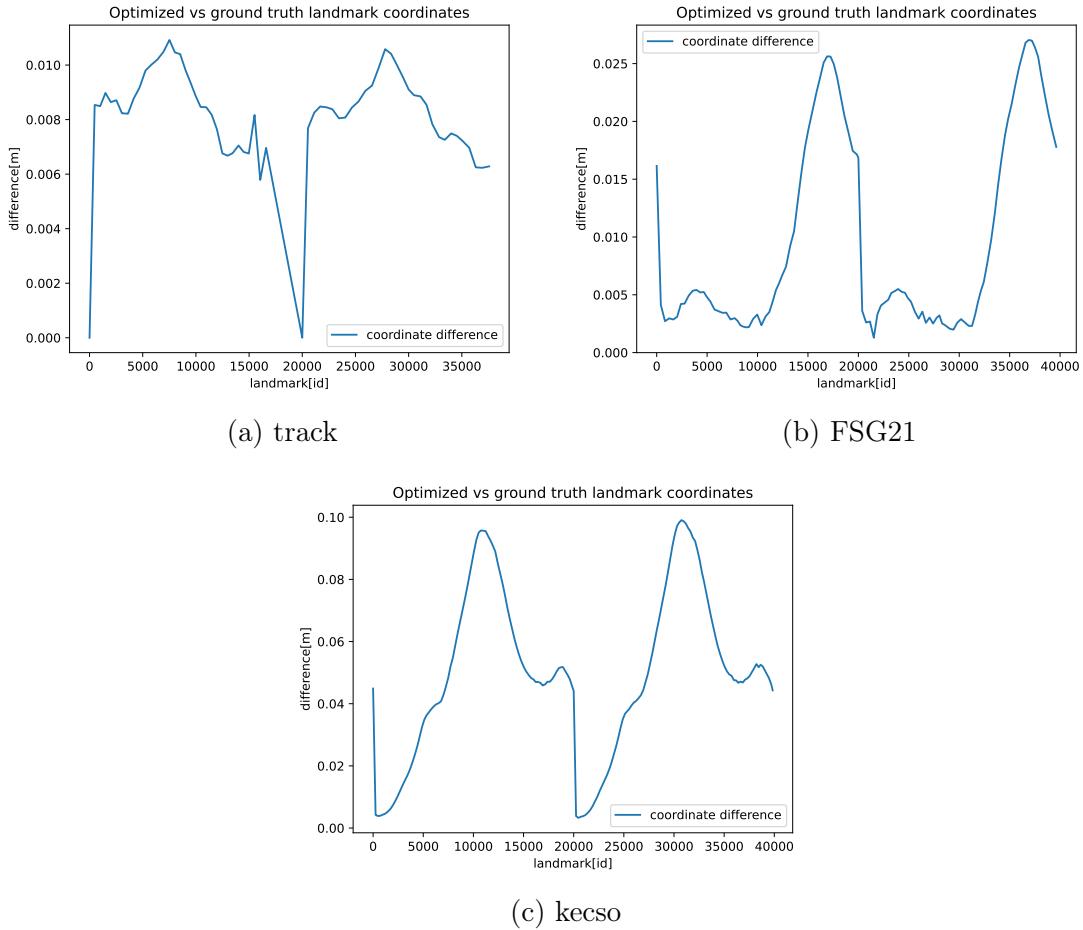
5.15. ábra. Optimalizált és ground truth pozíció koordináták különbsége - zajos eset lévén összehasonlítva az ehhez tartozó zajmentes esettel azt figyelhetjük meg, hogy a grafikonok zajjal terheltek.

Az alábbi ábrákon megfigyelhetjük, hogy a különböző térképeken milyen orientációból különbségek születtek az optimalizált, illetve a ground truth járműpózok között.



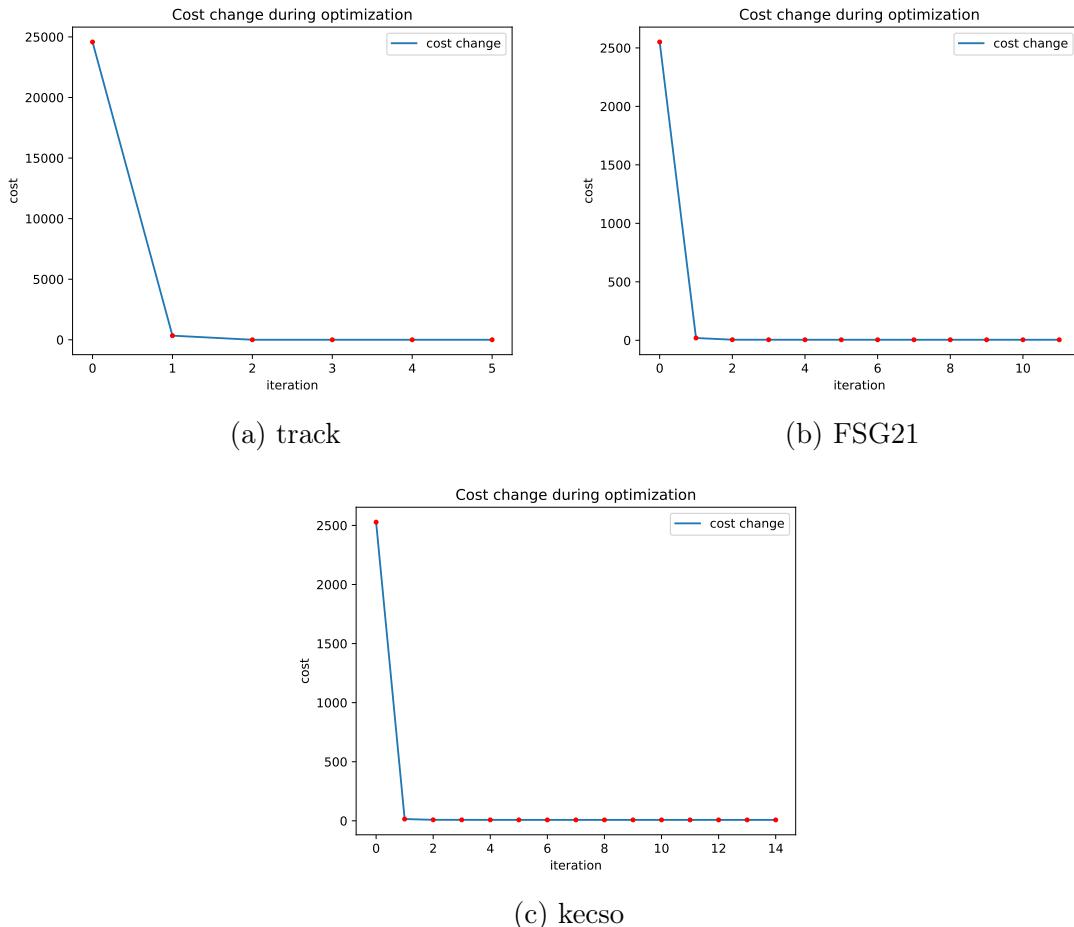
5.16. ábra. Optimalizált és ground truth póz orientációk különbsége - zajos eset lévén összehasonlítva az ehhez tartozó zajmentes esettel azt figyelhetjük meg, hogy a grafikonok zajjal terheltek.

Az alábbi ábrákon megfigyelhetjük, hogy a különböző térképeken milyen pozícióbeli különbségek születtek az optimalizált, illetve a ground truth tereptárgyak között.



5.17. ábra. Optimalizált és ground truth tereptárgy koordináták különbsége

Az alábbi ábrákon megfigyelhetjük, hogy a SLAM algoritmus különböző térképeken hány iteráció alatt és milyen költséggel konvergált.

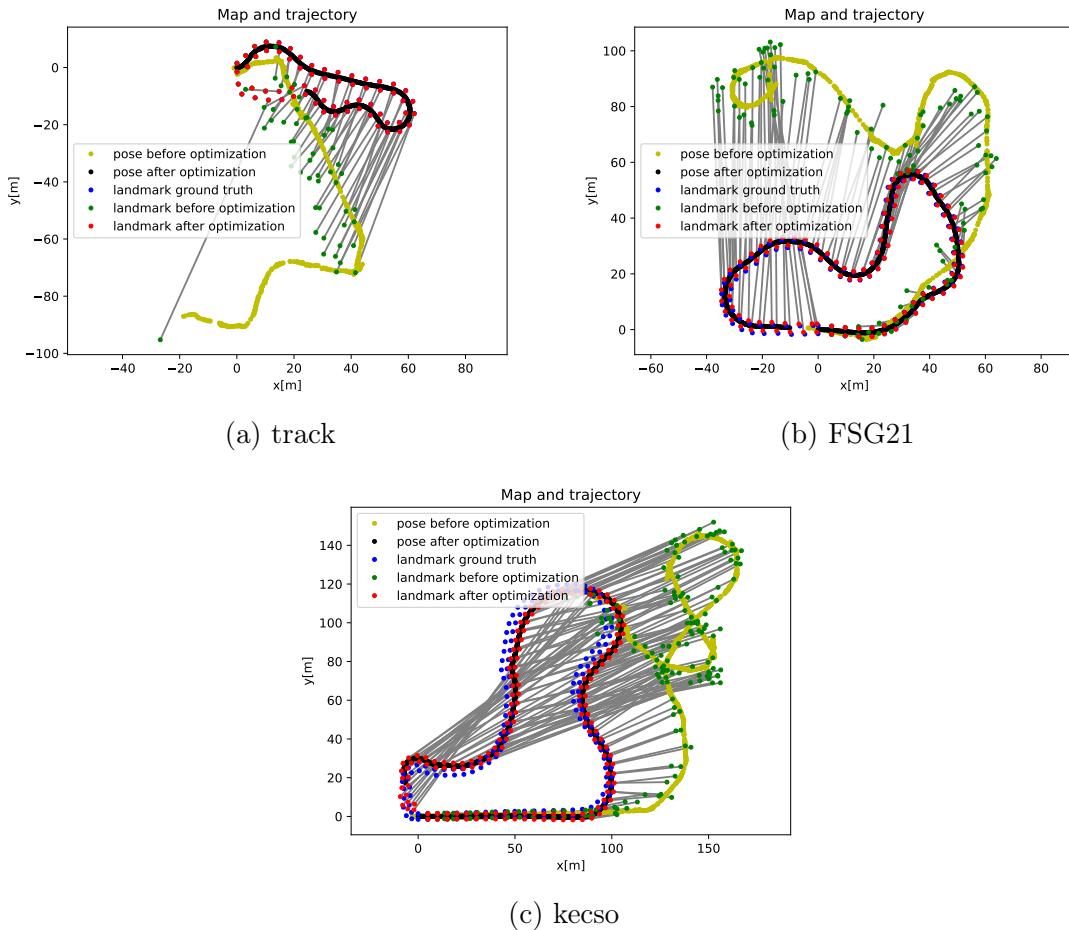


5.18. ábra. Hibafüggvények költségének csökkenése az optimalizáció során

	track	FSG21	kecso
Pose-ok száma	591	1321	1922
Landmark-ok száma	64	114	174
Csúcsok és élek létrehozása	~60 ms	~89 ms	~118 ms
Gráf felépítése	~15 ms	~30 ms	~30 ms
Gráf optimalizálása	~75 ms	~530 ms	~1270 ms

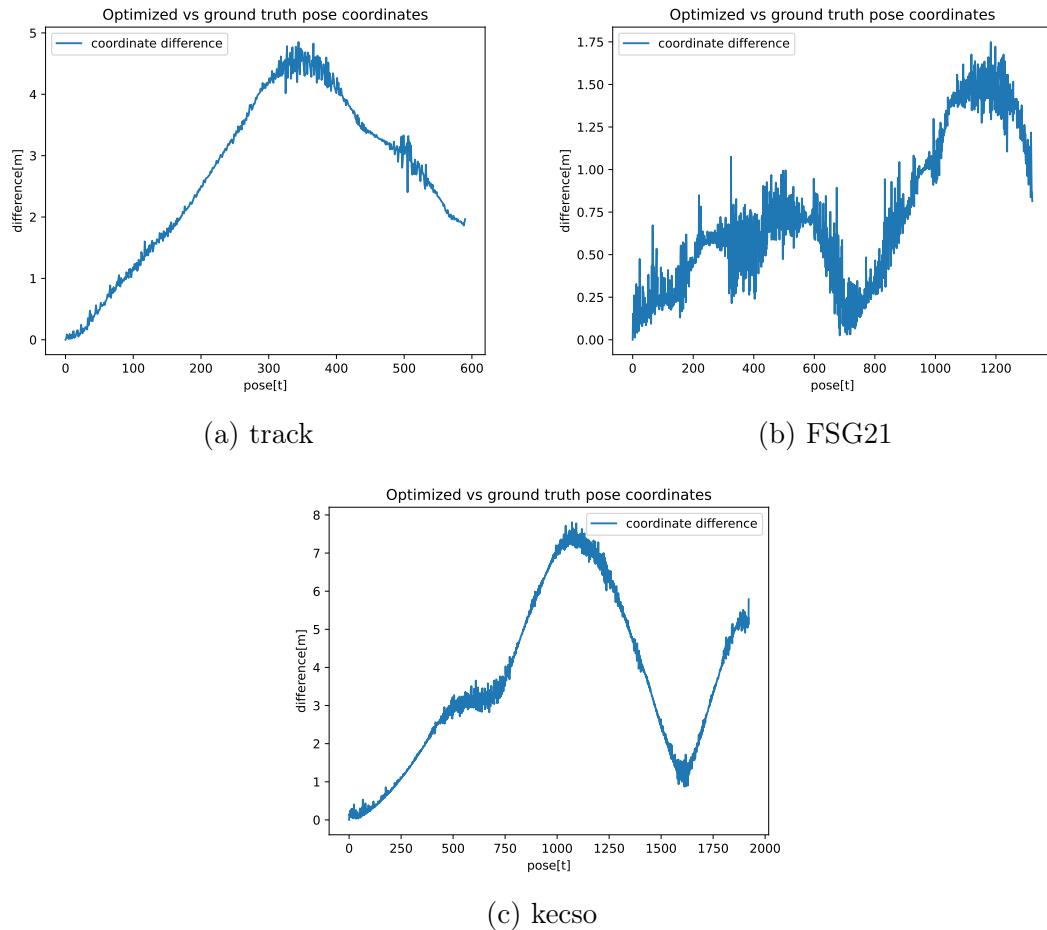
5.4. táblázat. Zajos esetek kiértékelése hurokzárás nélkül; $\sigma = 5$

A vizualizáció során megfigyelhetjük, hogy a fekete pontsorozat, mely a jármű útvonalát jelzi, nem ér össze, ugyanis ha nem engedélyezzük a SLAM számára a hurokzárást, akkor az útvonal azon pontján megszakad az adatfelvétel, ahol az érzékelésben már korábban látott bójákkal találkozunk.



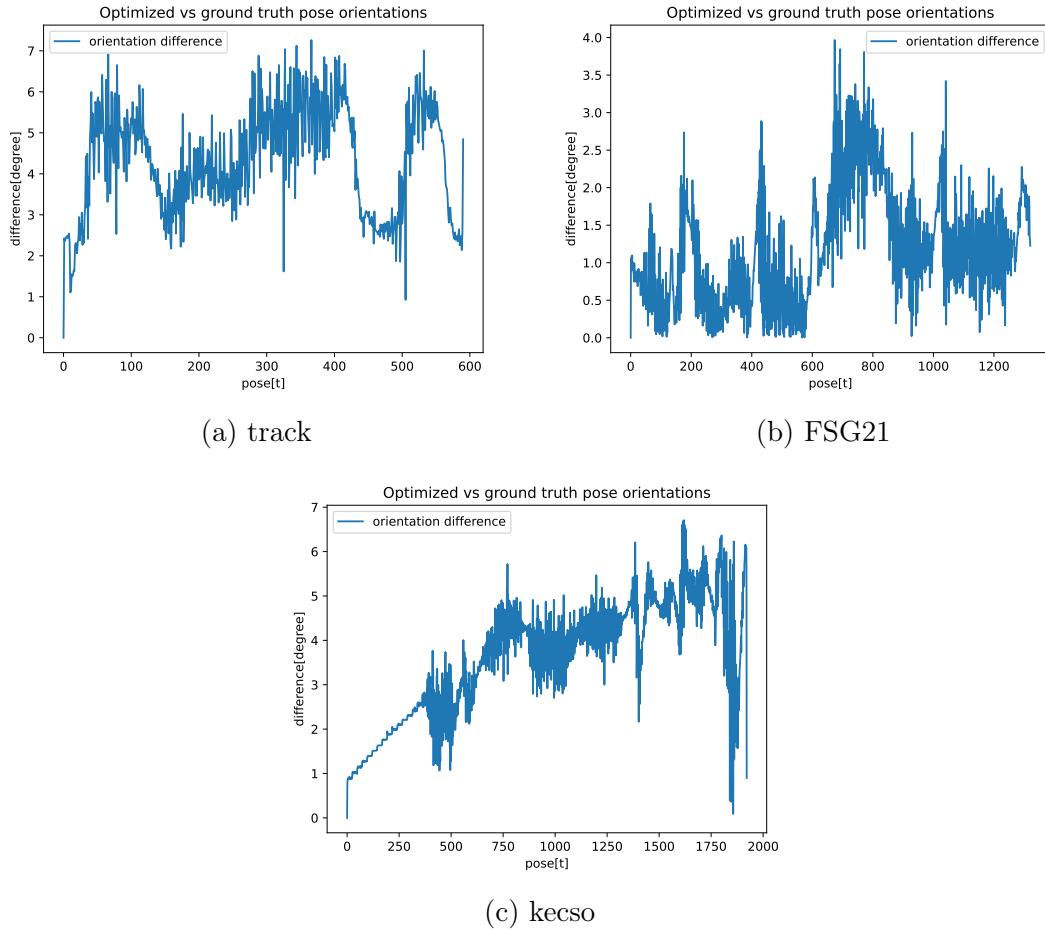
5.19. ábra. Zajos esetek vizualizációja hurokzárás nélkül; $\sigma = 5$

Az alábbi ábrákon megfigyelhetjük, hogy a különböző térképeken milyen pozícióbeli különbségek születtek az optimalizált, illetve a ground truth járműpózok között.



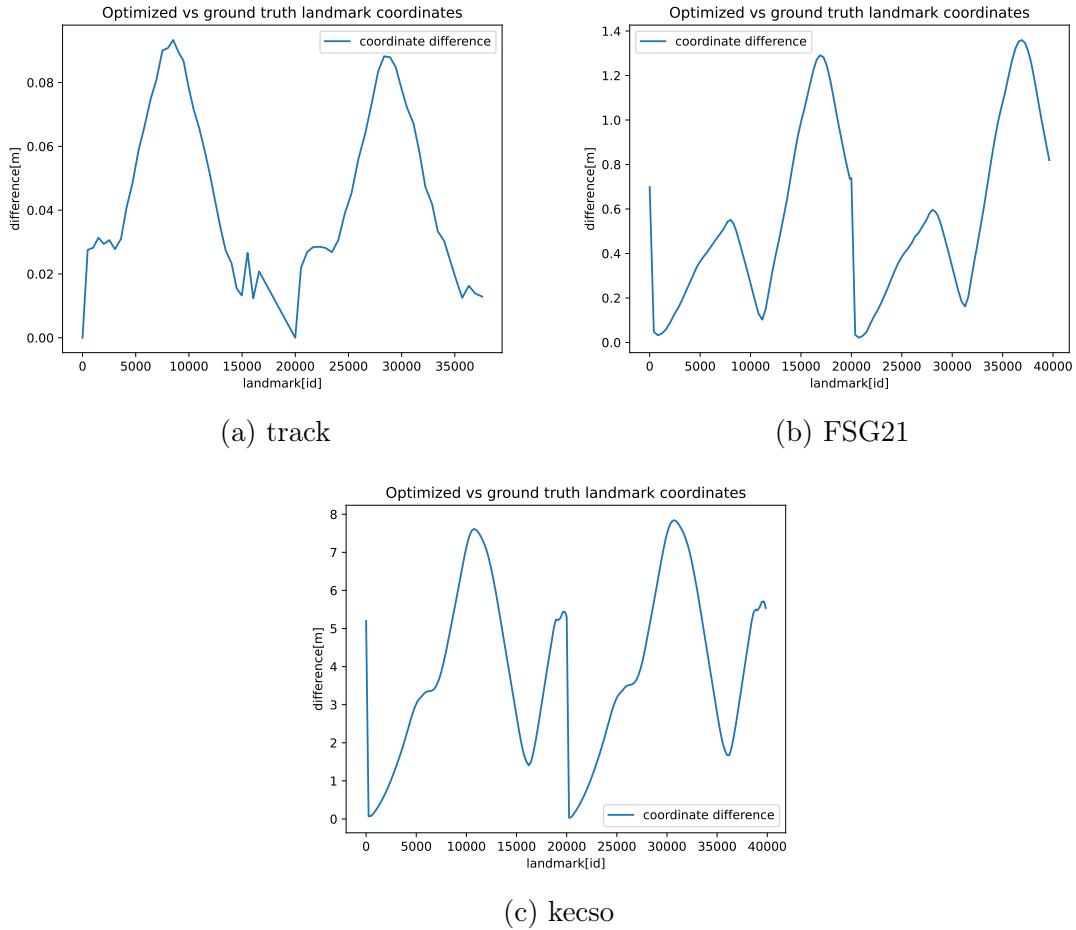
5.20. ábra. Optimalizált és ground truth póz koordináták különbsége - a zaj mértékének növelésével azt figyelhetjük meg, hogy a grafikonok méginkább zajjal terheltek az előző esethez képest.

Az alábbi ábrákon megfigyelhetjük, hogy a különböző térképeken milyen orientációból különbségek születtek az optimalizált, illetve a ground truth járműpózok között.



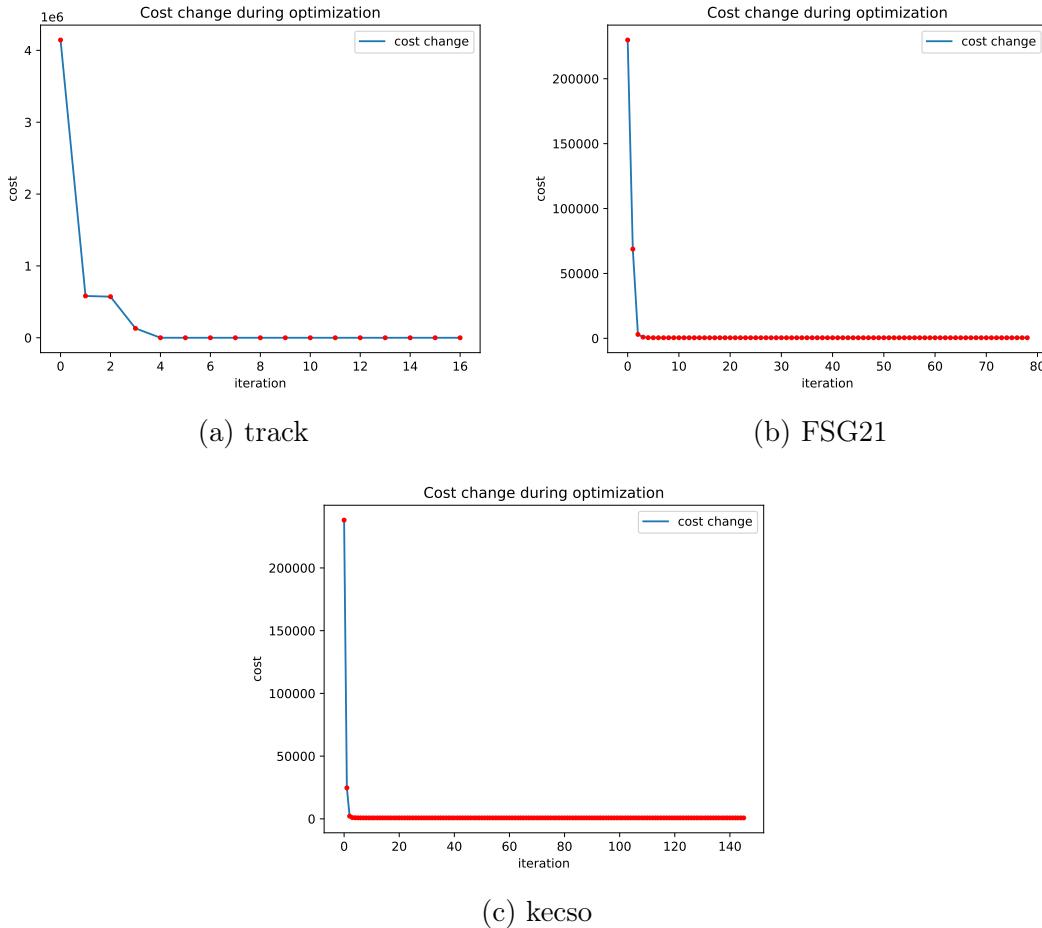
5.21. ábra. Optimalizált és ground truth pozíciók különbsége - a zaj mértékének növelésével azt figyelhetjük meg, hogy a grafikonok méginkább zajjal terheltek az előző esethez képest.

Az alábbi ábrákon megfigyelhetjük, hogy a különböző térképeken milyen pozícióbeli különbségek születtek az optimalizált, illetve a ground truth tereptárgyak között.



5.22. ábra. Optimalizált és ground truth tereptárgy koordináták különbsége

Az alábbi ábrákon megfigyelhetjük, hogy a SLAM algoritmus különböző térképeken hány iteráció alatt és milyen költséggel konvergált.



5.23. ábra. Hibafüggvények költségének csökkenése az optimalizáció során

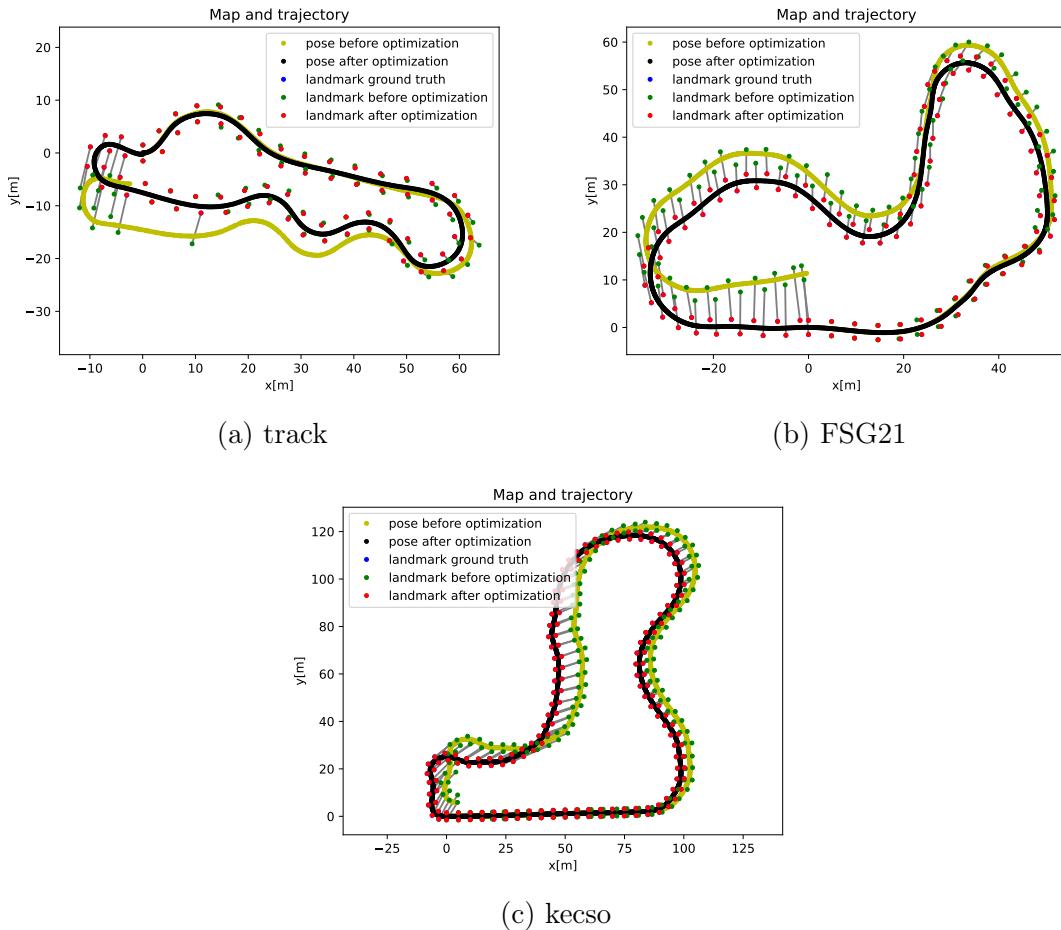
Hurokzárással

A SLAM ezen futása során használva a `--loop_closure` kapcsolót az alábbi statisztikákat kapjuk:

	track	FSG21	kecso
Pose-ok száma	843	1362	2001
Landmark-ok száma	74	114	174
Csúcsok és élek létrehozása	~60 ms	~89 ms	~118 ms
Gráf felépítése	~17 ms	~25 ms	~30 ms
Gráf optimalizálása	~45 ms	~66 ms	~124 ms

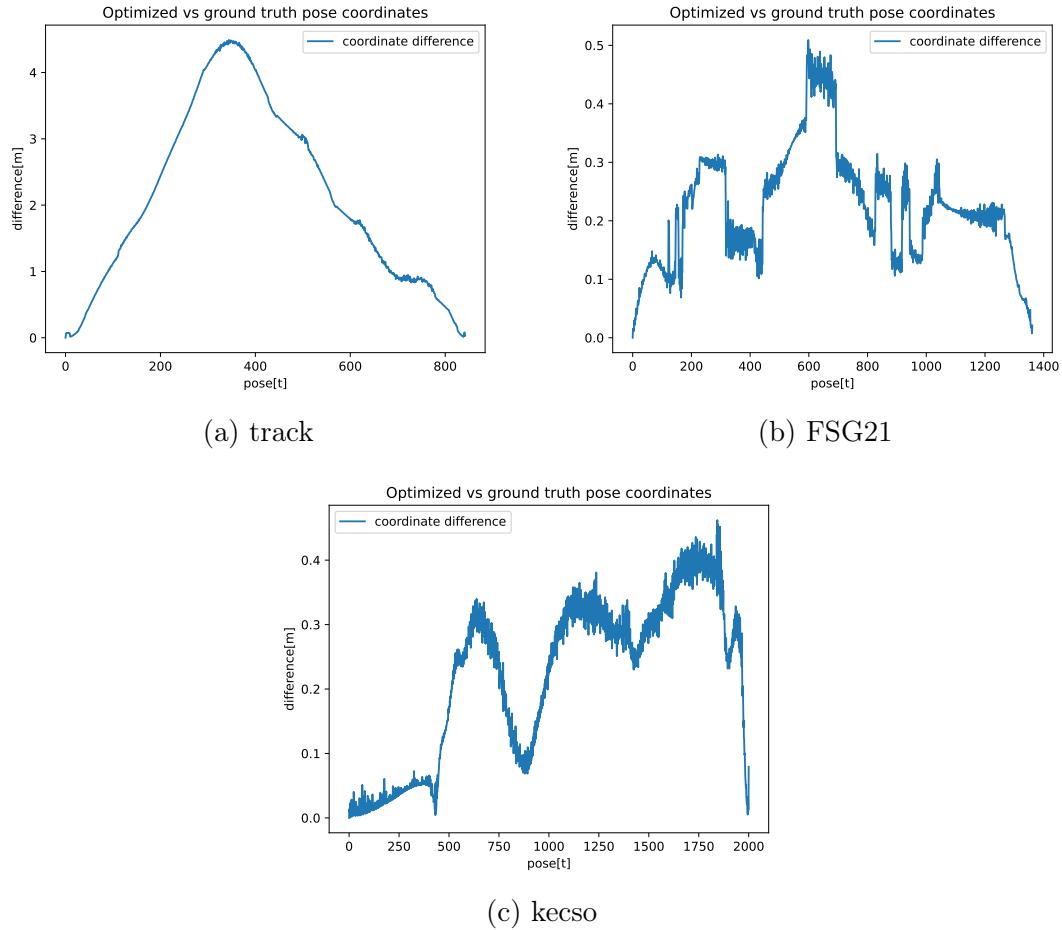
5.5. táblázat. Zajos esetek kiértékelése hurokzárással; $\sigma = 0.5$

A vizualizáció során megfigyelhetjük, hogy a fekete pontsorozat, mely a jármű útvonalát jelzi, összeér, ugyanis ha engedélyezzük a SLAM számára a hurokzárást, akkor az adatfelvétel egészen addig folytatódik, amíg van feldolgozandó mérés.



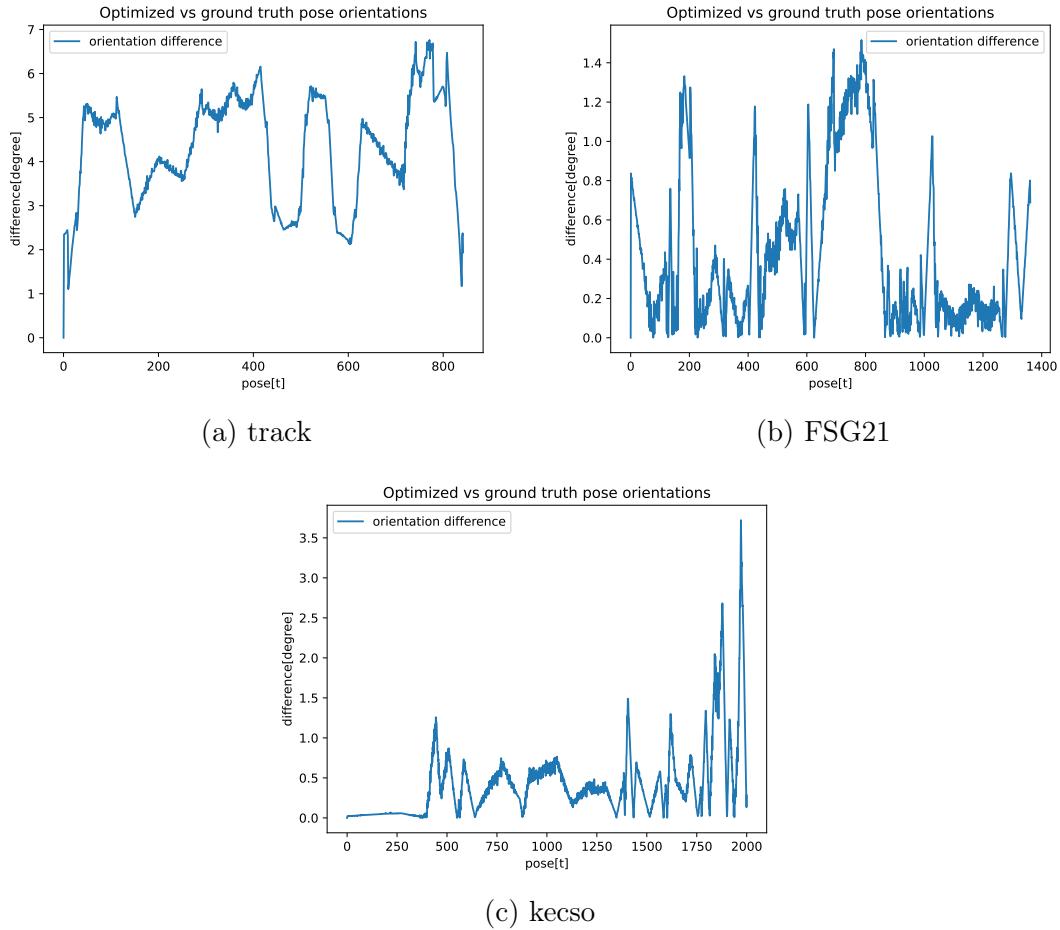
5.24. ábra. Zajos esetek vizualizációja hurokzárással; $\sigma = 0.5$

Az alábbi ábrákon megfigyelhetjük, hogy a különböző térképeken milyen pozícióbeli különbségek születtek az optimalizált, illetve a ground truth járműpózok között.



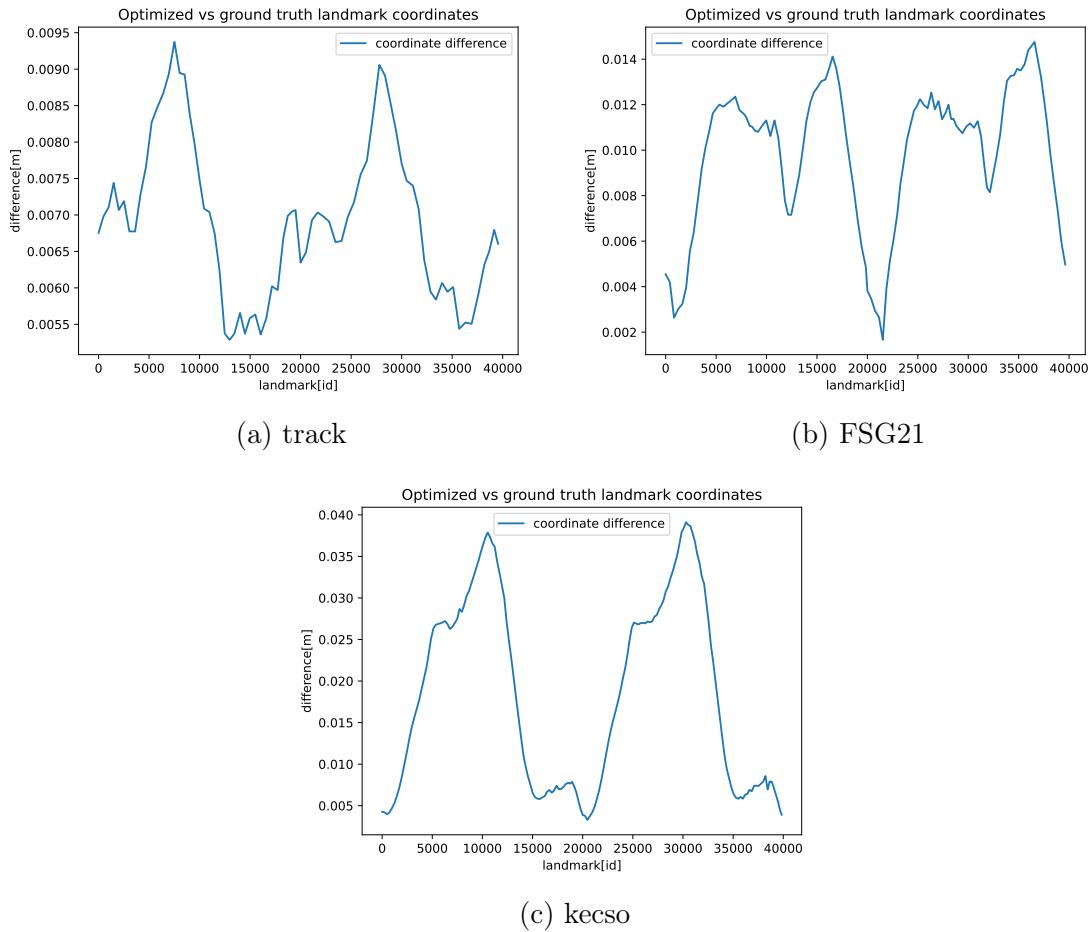
5.25. ábra. Optimalizált és ground truth pozíció koordináták különbsége - zajos eset lévén összehasonlítva az ehhez tartozó zajmentes esettel azt figyelhetjük meg, hogy a grafikonok zajjal terheltek.

Az alábbi ábrákon megfigyelhetjük, hogy a különböző térképeken milyen orientációból különbségek születtek az optimalizált, illetve a ground truth járműpózok között.



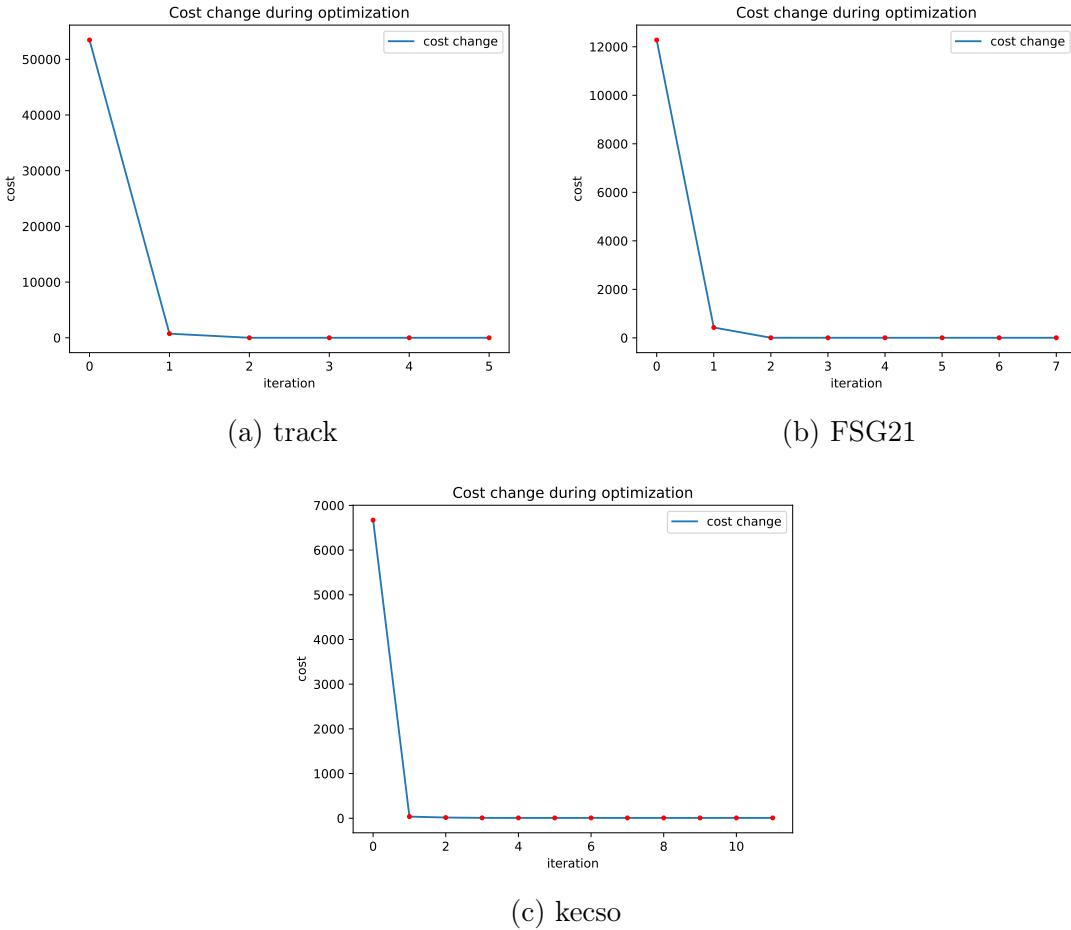
5.26. ábra. Optimalizált és ground truth póz orientációk különbsége - zajos eset lévén összehasonlítva az ehhez tartozó zajmentes esettel azt figyelhetjük meg, hogy a grafikonok zajjal terheltek.

Az alábbi ábrákon megfigyelhetjük, hogy a különböző térképeken milyen pozícióbeli különbségek születtek az optimalizált, illetve a ground truth tereptárgyak között.



5.27. ábra. Optimalizált és ground truth tereptárgy koordináták különbsége

Az alábbi ábrákon megfigyelhetjük, hogy a SLAM algoritmus különböző térképeken hány iteráció alatt és milyen költséggel konvergált.

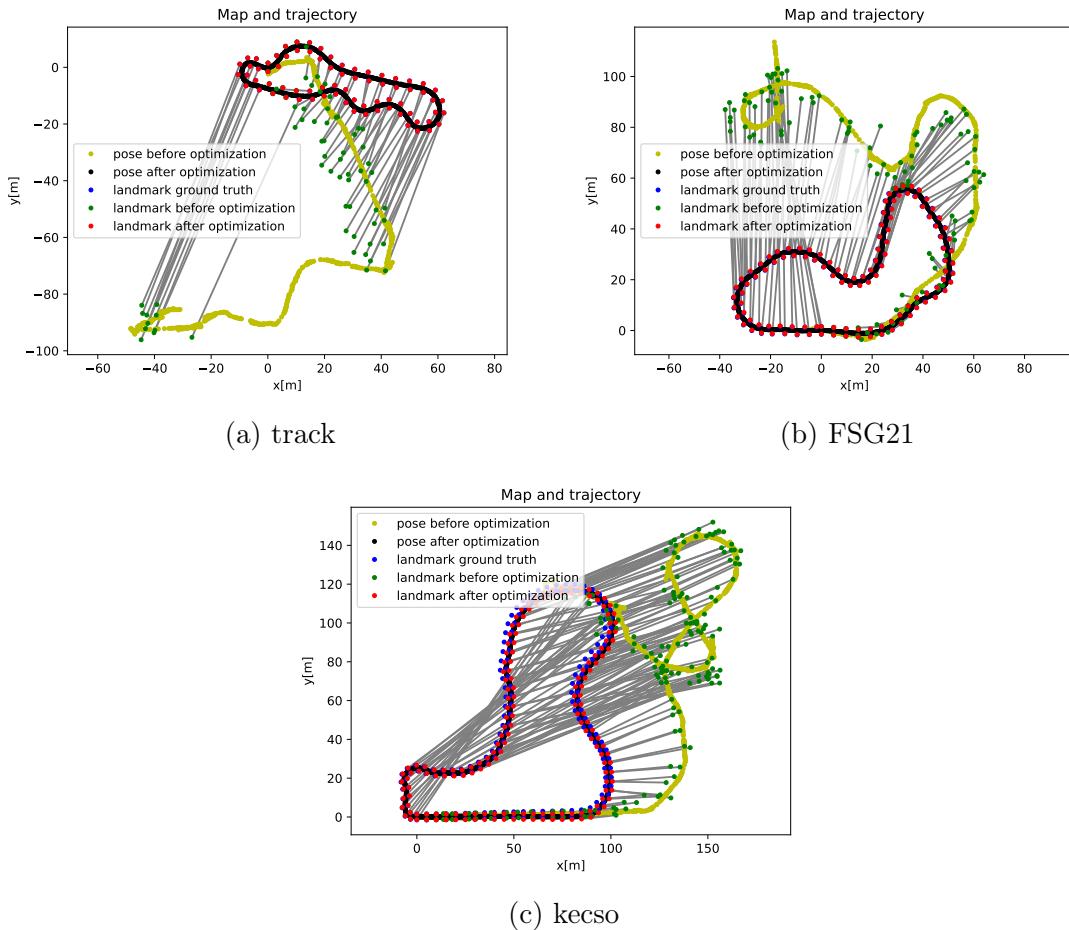


5.28. ábra. Hibafüggvények költségének csökkenése az optimalizáció során

	track	FSG21	kecso
Pose-ok száma	843	1362	2001
Landmark-ok száma	74	114	174
Csúcsok és élek létrehozása	~60 ms	~89 ms	~118 ms
Gráf felépítése	~15 ms	~22 ms	~30 ms
Gráf optimalizálása	~75 ms	~197 ms	~750 ms

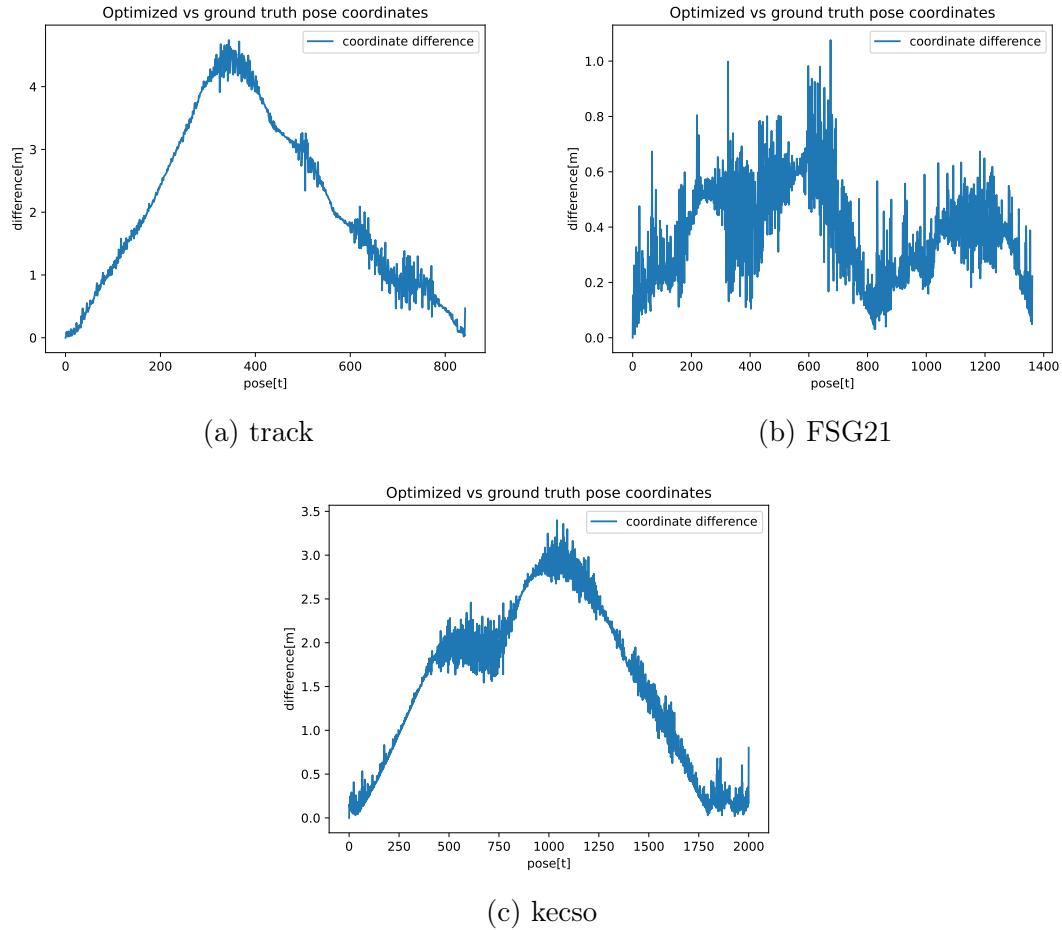
5.6. táblázat. Zajos esetek kiértékelése hurokzárással; $\sigma = 5$

A vizualizáció során megfigyelhetjük, hogy a fekete pontsorozat, mely a jármű útvonalát jelzi, összeér, ugyanis ha engedélyezzük a SLAM számára a hurokzárást, akkor az adatfelvétel egészen addig folytatódik, amíg van feldolgozandó mérés.



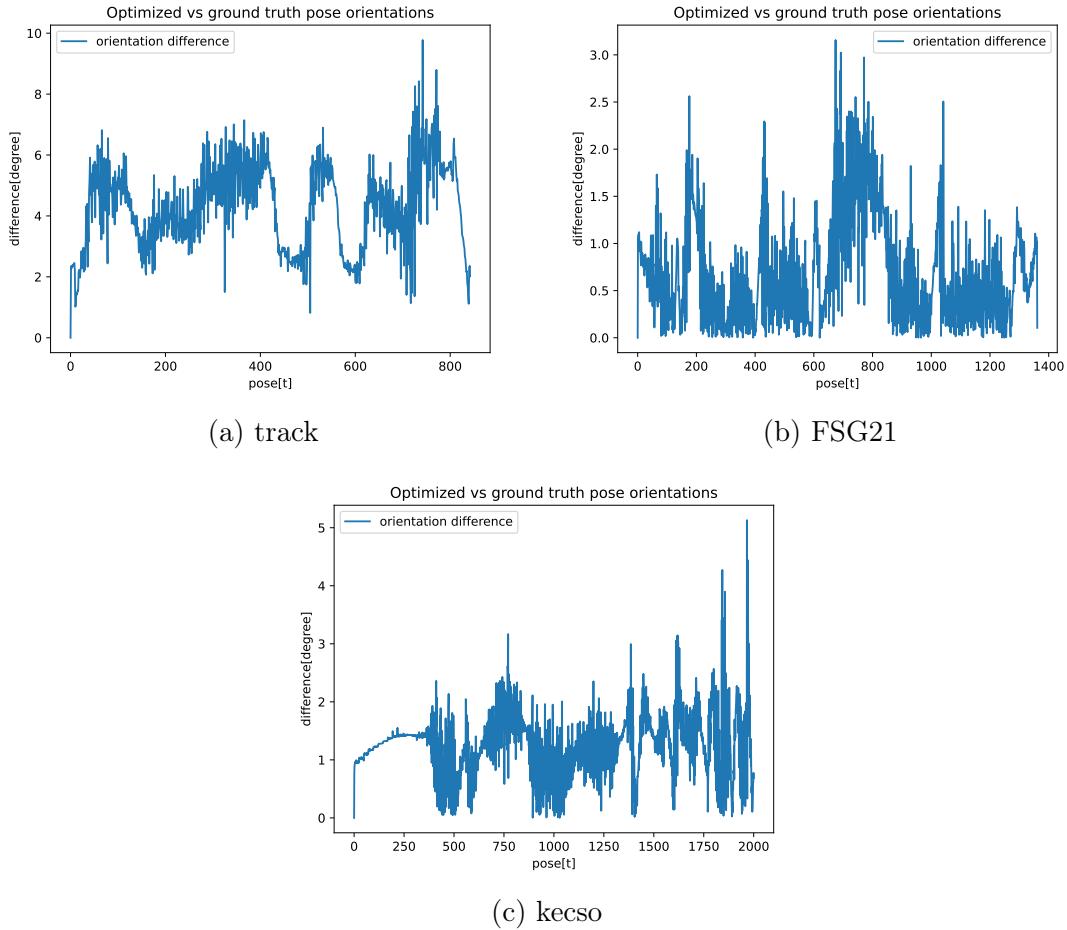
5.29. ábra. Zajos esetek vizualizációja hurokzárással; $\sigma = 5$

Az alábbi ábrákon megfigyelhetjük, hogy a különböző térképeken milyen pozícióbeli különbségek születtek az optimalizált, illetve a ground truth járműpózok között.



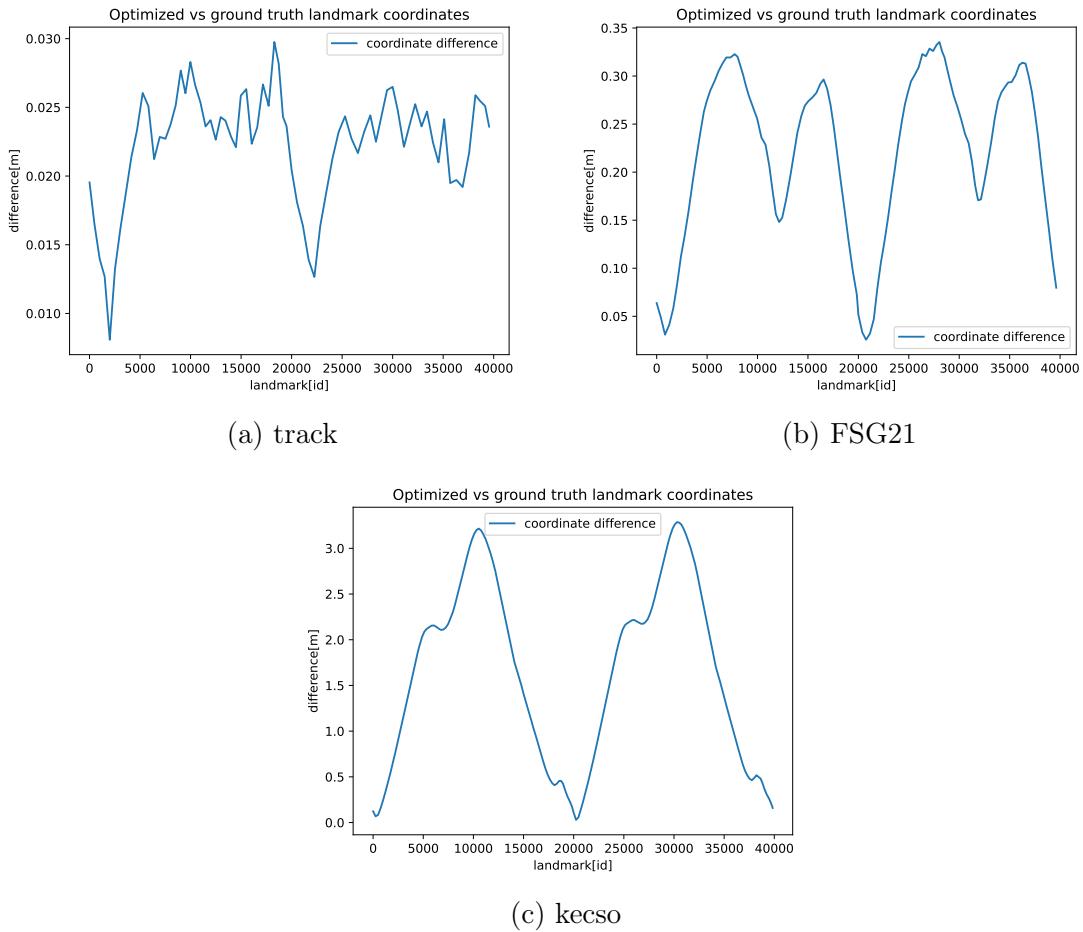
5.30. ábra. Optimalizált és ground truth póz koordináták különbsége - a zaj mértékének növelésével azt figyelhetjük meg, hogy a grafikonok méginkább zajjal terheltek az előző esethez képest.

Az alábbi ábrákon megfigyelhetjük, hogy a különböző térképeken milyen orientációból különbségek születtek az optimalizált, illetve a ground truth járműpózok között.



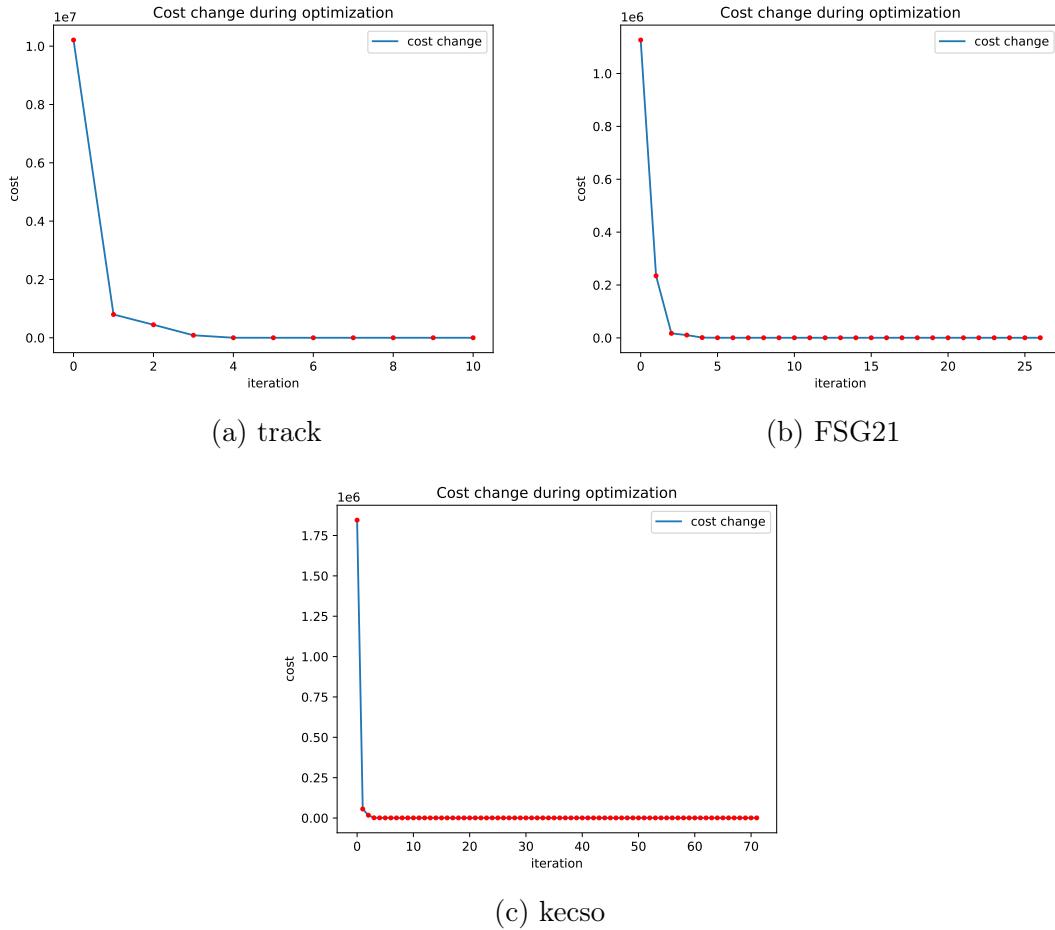
5.31. ábra. Optimalizált és ground truth pozíciók különbsége - a zaj mértékének növelésével azt figyelhetjük meg, hogy a grafikonok méginkább zajjal terheltek az előző esethez képest.

Az alábbi ábrákon megfigyelhetjük, hogy a különböző térképeken milyen pozícióbeli különbségek születtek az optimalizált, illetve a ground truth tereptárgyak között.



5.32. ábra. Optimalizált és ground truth tereptárgy koordináták különbsége

Az alábbi ábrákon megfigyelhetjük, hogy a SLAM algoritmus különböző térképeken hány iteráció alatt és milyen költséggel konvergált.



5.33. ábra. Hibafüggvények költségének csökkenése az optimalizáció során

5.2.3. Összegzés

A kiértékelésből azt szűrhetjük le, hogy míg az algoritmus futás ideje arányosan növekszik a gráf nagyságával, az iterációk száma nem feltétlen kell, hogy ezzel együtt növekedjen, hiszen az optimalizáció során nem a gráf nagysága határozza meg a lokális (globális) minimum megtalálásának nehézségét.

6. fejezet

Alkalmazás

6.1. Önvezető járművek általában

A SLAM használata gyakran megtalálható az autonóm navigációban, különösen a navigáció segítésére olyan területeken, amelyeken korábban még nem jártunk, vagy ahol a globális helymeghatározó rendszerek (GPS) meghibásodnak.

A GPS pontos helyzetet tud adni arról, hogy hol vagyunk a világban, de a GPS önmagában megbízhatatlan, különösen a magas épületekkel rendelkező városi területeken. A GPS sok tényezőtől függ, például a szabad égbolttól, vagy a látótávolságban lévő műholdak számától. Vannak helyzetek, amikor a GPS önmagában is megállna a helyét, mégis az az ésszerű döntés, ha a SLAM-et is bevetjük mellette, hiszen ilyenkor jóval pontosabb becsléseket kaphatunk a járműünk helyzetéről.

Azt hihetnénk, hogy ez a technológia annyira drága, hogy nagyon kevesek engedhetik meg maguknak, főleg akkor, ha elsőként például a Tesla¹ vállalat jut eszünkbe. A valóság ezzel szemben egészen megdöbbentő, ugyanis a legtöbb átlagos háztartásban is találhatunk példákat önvezető járművekre robotporszívók, vagy akár önjáró fűnyírók személyében.

Persze nem nehéz elrugaszkodni kicsit az átlagos dolguktól. A SLAM megtalálható bányák felderítésére szolgáló járművekben, de ha ilyenekről nem is, a NASA Perseverance² marsautójáról már biztosan hallottunk, melyet a Mars felszínének tanulmányozására használnak 2020 óta.

¹<https://www.tesla.com/>

²<https://www.nasa.gov/perseverance>

6.2. BME Formula Racing Team

A *BME Formula Racing Team*³ 2007-ben alakult, körülbelül 10-15 közlekedés-mérnök hallgatóból, amely azóta kinőtte magát, és már az egyetem szinte összes karáról csatlakoztak hallgatók a csapatba.

Fennállásuk során már 16 autót építettek, ebből 5 belsőégésű 11 pedig elektromos motorral hajtott. Három elektromos autót később önvezetővé alakítottak át. A csapat mérnökei és menedzserei különböző területeken dolgoznak ugyanazért a célért, a leendő mérnökök tudásának gyarapításáért és a minél jobb eredmények eléréséért, valamint hogy megteremtsék a magas szintű anyagi, szervezeti és tárgyi feltételeket.

Eddig több mint 300 hallgató tevékenykedett a tervezés, a beszerzés, a gyártás, a menedzsment, a logisztika és a gazdasági ügyek intézése területén. Jelenleg több mint 60 csapattaggal működnek, akik közt található gépész-, közlekedés-, villamosmérnök és közigazdaságтан hallgató is.

2011-ben mutatkozott be a Formula Student⁴ mezőnyében a BME FRT - és Magyarország - első elektromos hajtású Formula Student autója, a 2021-es évben pedig már a tizedik ilyen autójuk készült el.

Szerencsésnek mondhatom magam, hogy 2021 októberétől én is a csapat tagja lehetek, és első kézből tapasztalhatom meg, hogy milyen komplex feladatokkal kell szembesülnie egy autonóm rendszerekkel foglalkozó informatikusnak a csapat önvezető csapatjának tagjaként.

³<https://frt.bme.hu/>

⁴https://en.wikipedia.org/wiki/Formula_Student



6.1. ábra. Lola, a csapat 2022-es versenyautója. A képet Keresztfuri Levente készítette.

Mondhatjuk, hogy a dolgozatom termékgazdájának szerepét a csapat töltötte be, ugyanis ebben az évben a jelenleg is az autón működő FastSLAM [17] mellé készült el a gráf alapú implementációm, hogy az előbb említett algoritmussal fúzionálva pontosabb működést érhessünk el.

7. fejezet

Összegzés

A helymeghatározás és térképezés két elengedhetetlen kritériuma a robotika azon részének, ahol az eszközünk autonóm módon kényszerül feladatok megoldására.

A modern gráf alapú SLAM algoritmusok számítási költségüket tekintve nagyon hatékony megoldást kínálnak a probléma megoldására abban az esetben, ha a robotunk képes tökéletesen felismerni a környezetében jelenlévő tereptárgyakat.

A már korábban meglátogatott helyek felismerése borzalmasan nehéz feladat egy számítógép számára. Vegyük példának egy hosszú, egyenes és referenciaPontokban szegényes folyosót. Robotunk folyamatos bizonytalanságban mozogna, hiszen bár tudná, hogy történt elmozdulás, az érzékelésből nagyon kevés megerősítést kapna e felől. A hasonló helyszínekben történő navigáció tartogatta kihívásokon felül ráadásul minden valós esetben társul zaj a szenzorok pontatlanságából adódóan.

A kiértékelés (5.2) során kapott adatok megmutatatták, hogy a gyakorlatban egy nagyobb, globális optimalizáció futásideje akár egy másodpercet is igénybe vehet. Ez az önvezető járműünk valós idejű használatát nehezíti. Ezt többszálúsítással kiküszöbülhetjük úgy, hogy amíg az optimalizáció zajlik, nem áll meg az adatfel-dolgozás folyamata. A kiértékelés továbbá azt is megmutatta, hogy a hurokzárás a konvex optimalizáción alapuló SLAM algoritmusknál nagyon hatékony, közel teljes pontossággal tudjuk visszakapni a környezetről alkotott térképet (lásd 5.29. ábra).

A dolgozatom célja egy ilyen gráf alapú SLAM algoritmus, illetve ezen algoritmus tesztelését elősegítő szimulációs környezet megvalósítása volt, melyhez mélyebben bele kellett ásnom magam a numerikus optimalizálási problémák rejtelmeibe. Ezen felül a dolgozat lehetőséget adott arra, hogy megerősítsem készségeimet a verziókezelés, illetve a projektstrukturálás terén.

Irodalomjegyzék

- [1] H. Durrant-Whyte és T. Bailey. “Simultaneous localization and mapping: part I”. *IEEE Robotics Automation Magazine* 13.2 (2006), 99–110. old. DOI: 10.1109/MRA.2006.1638022.
- [2] Akshay Kumar. “An Introduction to Simultaneous Localization and Mapping (SLAM) for Robots”. *Control.com* (2020. máj.). URL: <https://control.com/technical-articles/an-introduction-to-simultaneous-localization-and-mapping-slam-for-robots/>.
- [3] Giorgio Grisetti és tsai. “A Tutorial on Graph-Based SLAM”. *IEEE Intelligent Transportation Systems Magazine* 2.4 (2010), 31–43. old. DOI: 10.1109/MITS.2010.939925.
- [4] Guido Van Rossum és Fred L. Drake. *Python 3 Reference Manual*. Scotts Valley, CA: CreateSpace, 2009. ISBN: 1441412697.
- [5] J. D. Hunter. “Matplotlib: A 2D graphics environment”. *Computing in Science & Engineering* 9.3 (2007), 90–95. old. DOI: 10.1109/MCSE.2007.55.
- [6] Charles R. Harris és tsai. “Array programming with NumPy”. *Nature* 585.7825 (2020. szept.), 357–362. old. DOI: 10.1038/s41586-020-2649-2. URL: <https://doi.org/10.1038/s41586-020-2649-2>.
- [7] Pauli Virtanen és tsai. “SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python”. *Nature Methods* 17 (2020), 261–272. old. DOI: 10.1038/s41592-019-0686-2.
- [8] Lócsi Levente. “Numerikus módszerek 1. - 6. előadás: Vektor- és mátrixnormák diasor”. (2013. okt.).
- [9] Bese Antal. “Analízis - Differenciálgeometria, vektoranalízis, mérték és integrálelmélet”. (2006. dec.).

- [10] Dr. Bozsik József. “Numerikus módszerek 2B. - 7. előadás: B-Spline interpoláció diasor”. (2021. okt.).
- [11] Lócsi Levente. “Numerikus módszerek 1. - 4. előadás: Cholesky- és QR-felbontás, Gram–Schmidt-módszer diasor”. (2013. szept.).
- [12] Sameer Agarwal, Keir Mierle és The Ceres Solver Team. *Ceres Solver*. 2.1. verzió. 2022. márc. URL: <https://github.com/ceres-solver/ceres-solver>.
- [13] Dr. Bozsik József. “Numerikus módszerek 2B. - 8–9. előadás: Szinguláris felbontás, általánosított inverz diasor”. (2021. nov.).
- [14] Dr. Hajder Levente. “Numerikus optimalizálás”. (2007. okt.). URL: http://cg.elte.hu/~latas/jegyzet/matek/numerikus_opt/numerikus_opt.pdf.
- [15] Gaël Guennebaud, Benoît Jacob és tsai. *Eigen v3*. <http://eigen.tuxfamily.org>. 2010.
- [16] Teréz Várkonyi, Tibor Gregorics és András Nagy. “Improvement of Abstract Reasoning in Teaching Computer Science at Higher Education Level”. 2020. jan., 239–248. old. ISBN: 978-3-030-36840-1. DOI: 10.1007/978-3-030-36841-8_23.
- [17] Michael Montemerlo és tsai. “FastSLAM: A factored solution to the simultaneous localization and mapping problem”. *Aaaai/iaai* 593598 (2002).

Ábrák jegyzéke

1.1.	Online SLAM	2
1.2.	Full SLAM	3
2.1.	Szimulációs környezet függőségi diagramja	5
2.2.	Szimulációs környezet Use Case diagramja	7
2.3.	Bemeneti fájl részlete a szimulációs környezet FSG21 szimulációjához	9
2.4.	Szimulációs környezet FSG21 szimulációja	9
2.5.	Szimulációs környezet FSG21 szimuláció irányvektorai	10
2.6.	Szimulációs környezet FSG21 szimuláció érzékelése	10
2.7.	Kimeneti fájl részlete a szimulációs környezet FSG21 szimulációjához	11
2.8.	A SLAM szoftver Use Case diagramja	13
2.9.	Bemeneti fájl részlete a SLAM FSG21 zaj nélküli futásához	16
2.10.	SLAM FSG21 zaj nélküli futása	17
2.11.	SLAM FSG21 zaj nélküli futás szegmentált vizualizációja	18
2.12.	SLAM FSG21 zaj nélküli futás vizualizációja	18
2.13.	Kimeneti fájl részlete a SLAM FSG21 zaj nélküli futásához	19
4.1.	Szimulációs környezetben használt típusok osztálydiagramja	26
4.2.	Szimulációs környezet folyamatábrája	29
4.3.	SLAM függőségi diagramja	30
4.4.	SLAM osztálydiagramja	31
4.5.	Egyedi típusok osztálydiagramja	31
4.6.	Data osztálydiagramja	34
4.7.	DataEnumerator osztálydiagramja	35
4.8.	Hibafüggvények osztálydiagramja	36
4.9.	Graph osztálydiagramja	39
4.10.	SLAM probléma gráf alapú reprezentációja	41
4.11.	Csúcsok és élek létrehozása	42

4.12. Optimalizálási probléma létrehozása és megoldása	42
5.1. Bóják az egységkör körvonalán	49
5.2. Egyenes vonalú egyenletes mozgás	49
5.3. Görbe vonalú mozgás	50
5.4. Zajmentes esetek vizualizációja hurokzárás nélkül	53
5.5. Optimalizált és ground truth pozíció koordináták különbsége	54
5.6. Optimalizált és ground truth pozíció orientációk különbsége	55
5.7. Optimalizált és ground truth tereptárgy koordináták különbsége	56
5.8. Hibafüggvények költségének csökkenése az optimalizáció során	57
5.9. Zajmentes esetek vizualizációja hurokzárással	58
5.10. Optimalizált és ground truth pozíció koordináták különbsége	59
5.11. Optimalizált és ground truth pozíció orientációk különbsége	60
5.12. Optimalizált és ground truth tereptárgy koordináták különbsége	61
5.13. Hibafüggvények költségének csökkenése az optimalizáció során	62
5.14. Zajos esetek vizualizációja hurokzárás nélkül; $\sigma = 0.5$	63
5.15. Optimalizált és ground truth pozíció koordináták különbsége	64
5.16. Optimalizált és ground truth pozíció orientációk különbsége	65
5.17. Optimalizált és ground truth tereptárgy koordináták különbsége	66
5.18. Hibafüggvények költségének csökkenése az optimalizáció során	67
5.19. Zajos esetek vizualizációja hurokzárás nélkül; $\sigma = 5$	68
5.20. Optimalizált és ground truth pozíció koordináták különbsége	69
5.21. Optimalizált és ground truth pozíció orientációk különbsége	70
5.22. Optimalizált és ground truth tereptárgy koordináták különbsége	71
5.23. Hibafüggvények költségének csökkenése az optimalizáció során	72
5.24. Zajos esetek vizualizációja hurokzárással; $\sigma = 0.5$	73
5.25. Optimalizált és ground truth pozíció koordináták különbsége	74
5.26. Optimalizált és ground truth pozíció orientációk különbsége	75
5.27. Optimalizált és ground truth tereptárgy koordináták különbsége	76
5.28. Hibafüggvények költségének csökkenése az optimalizáció során	77
5.29. Zajos esetek vizualizációja hurokzárással; $\sigma = 5$	78
5.30. Optimalizált és ground truth pozíció koordináták különbsége	79
5.31. Optimalizált és ground truth pozíció orientációk különbsége	80
5.32. Optimalizált és ground truth tereptárgy koordináták különbsége	81

ÁBRÁK JEGYZÉKE

Táblázatok jegyzéke

2.1.	Rendszerkövetelmények	5
2.2.	Kapcsolók és hatásuk	6
2.3.	A SLAM szoftver rendszerkövetelményei	12
2.4.	Kapcsolók és hatásuk	13
5.1.	Zajmentes esetek kiértékelése hurokzárás (3.4) nélkül	52
5.2.	Zajmentes esetek kiértékelése hurokzárással	57
5.3.	Zajos esetek kiértékelése hurokzárás nélkül; $\sigma = 0.5$	62
5.4.	Zajos esetek kiértékelése hurokzárás nélkül; $\sigma = 5$	67
5.5.	Zajos esetek kiértékelése hurokzárással; $\sigma = 0.5$	72
5.6.	Zajos esetek kiértékelése hurokzárással; $\sigma = 5$	77