



Budapesti Műszaki és Gazdaságtudományi Egyetem

Villamosmérnöki és Informatikai Kar

Automatizálási és Alkalmazott Informatikai Tanszék

Pályatervezési és pályakövető szabályozási algoritmusok fejlesztése differenciális robothoz

DIPLOMATERV

Készítette

Nagy Ákos

Konzulens

Kiss Domokos

2014. december 10.

Tartalomjegyzék

Kivonat	4
Abstract	5
Bevezető	6
1. Elméleti bevezetés	7
1.1. Problémafelvetés	7
1.2. Pályatervezés elmélete	7
1.2.1. Alapvető fogalmak	7
1.2.2. Pályatervezők osztályozása	9
2. Az RTR algoritmus	12
2.1. RRT algoritmus	12
2.2. RTR algoritmus	13
2.2.1. Mintavételezés	14
2.2.2. Csomópont kiválasztás	14
2.2.3. Kiterjesztés	15
2.2.4. Útvonal meghatározása, optimalizálása	17
2.3. Implementálás	18
2.3.1. Optimalizálások	21
2.4. Eredmények	22
3. Pálya időparaméterezése	25
3.1. Jelölések	26
3.2. Differenciális robotmodell	27
3.3. Korlátozások	27
3.4. Geometriai sebességprofil	29
3.4.1. Profil visszaterjesztés	30
3.5. Újramintavételezés	33
3.5.1. Mintavételezett pálya végpontja	36
3.6. Eredmények	39
4. Pályakövető szabályozás	42
4.1. Egy helyben fordulás	42

4.2.	Pályakövetés	44
4.2.1.	Sebességszabályozás	45
4.2.2.	Referenciapont-választás	45
4.2.3.	Orientációs szabályozás	46
4.2.4.	Túlhaladás problémája	47
4.2.5.	Orientációs szabályozás szegmens végpontjánál	48
4.3.	Eredmények	49
5.	Rendszer implementációja	51
5.1.	C*CS pályatervező	51
5.2.	V-REP robotszimulátor	52
5.2.1.	Keretrendszer	52
5.2.2.	Szimulációs eredmények	54
5.3.	Valós robot	57
5.3.1.	A robot irányítórendszer	58
5.3.2.	Valós eredmények	59
6.	Összegzés	65
Köszönnetnyilvánítás		66
Irodalomjegyzék		68
Függelék		69
F.1.	Megjegyzés a CD melléklettel kapcsolatban	69

HALLGATÓI NYILATKOZAT

Alulírott *Nagy Ákos*, szigorló hallgató kijelentem, hogy ezt a diplomatervet meg nem engedett segítség nélkül, saját magam készítettem, csak a megadott forrásokat (szakirodalom, eszközök stb.) használtam fel. minden olyan részt, melyet szó szerint, vagy azonos értelemben, de átfogalmazva más forrásból átvettettem, egyértelműen, a forrás megadásával megjelöltem.

Hozzájárulok, hogy a jelen munkám alapadatait (szerző(k), cím, angol és magyar nyelvű tartalmi kivonat, készítés éve, konzulens(ek) neve) a BME VIK nyilvánosan hozzáférhető elektronikus formában, a munka teljes szövegét pedig az egyetem belső hálózatán keresztül (vagy autentikált felhasználók számára) közzétegye. Kijelentem, hogy a benyújtott munka és annak elektronikus verziója megegyezik. Dékáni engedéllyel titkosított diplomatervek esetén a dolgozat szövege csak 3 év eltelte után válik hozzáférhetővé.

Budapest, 2014. december 10.

Nagy Ákos
hallgató

Kivonat

A mobil robotok az elmúlt évtizedben rengeteget fejlődtek. Annak ellenére, hogy az iparban már sok helyen alkalmaznak mobil robotokat, de még igen sok elméleti és gyakorlati problémát kell megoldani ahhoz, hogy a minden napokban mobil robotokkal találkozzunk.

Az egyik legalapvetőbb kérdés a robotikában az akadályok jelenlétében történő mozgás-tervezés és mozgásvéghajtás. A dolgozatomban ezt a téma körét járom körbe, kezdve a pályatervezés kérdésével, majd a pályatervezéshez kapcsolódó időparaméterezés problémájával, és legvégül pályakövető szabályozással. A pályatervező algoritmusnál felhasználjuk a szakirodalomban jól ismert *RRT* (*Rapidly Exploring Random Trees*) eljárást, és ismertetünk egy approximációs módszeren alapuló megközelítést is. A dolgozatban bemutatott pályakövető algoritmus a robot pályamenti sebességét és a szögsebességét függetlenül szabályozza.

Az ismertetésre kerülő algoritmusokat alapvetően differenciális robot esetében tárgyaljuk, de fontos szempont, hogy amennyiben lehetséges, általános eljárásokat dolgozzunk ki. Differenciális robot esetében szimulációs és valós környezetben végzett méréseket is bemutatunk. Az algoritmusokat *V-REP* robotszimulátor programban teszteljük, ahogyan a valós eredményeket is ebben a szoftverben jelenítjük meg.

Abstract

Nowadays the research of mobile robots is increasingly widespread. They are getting popular in industrial applications, but there are a lot of theoretical and practical issues to be resolved for personal usage.

One of the most fundamental aspects of mobile robotics is motion planning and control in environments populated with obstacles. In this thesis we discuss geometric path planning, velocity profile generation and motion control along the path. The presented global path planner is based on the well-known *RRT* (*Rapidly Exploring Random Trees*) method. We also discuss an approximation method for path planning, which is based on a global and a local planner algorithm. The motion control algorithm covered in this thesis controls the robot translational and angular velocity in two independent control loops.

The algorithms described above are developed primarily for differential robots, but can be extended for other robot types as well. We present simulation results for differential drive robot, and test results with a real, differential robot. The investigated methods are tested in *V-REP* robot simulation framework, and the real robot measurements are also carried out in this program.

Bevezető

A dolgozatom célja, hogy az elméletben kidolgozott pályakövetési és mozgásirányítási módszereket a gyakorlatban is kipróbáljam. Ehhez természetesen a pályatervezés, mozgáskövetés elméleti ismereteire, valamint a gyakorlati tapasztalatok alapján a kidolgozott algoritmusok módosítására is szükség van. A diplomatervem ezt a folyamatot követi végig.

A dolgozat elején röviden ismertetjük a pályatervezéshez és pályakövetéshez szükséges alapvető fogalmakat. Ezenkívül bemutatom a munkám során alkalmazott kinematikai robotmodellt, és az alapvető pályatervezési módszerek csoportosításait.

A második fejezetben rátérünk egy globális pályatervező, az RTR (Rotate-Translate-Rotate) algoritmus tárgyalására. Az RTR algoritmus megértéséhez szükséges az úgynevezett RRT eljárást bevezetni, amelyet pályatervezési feladatoknál igen gyakran alkalmazznak. Az RTR tervező elméleti tárgyalása után, ismertetem az implementáláshoz kapcsolódó gyakorlati problémákat. Az implementálás bemutatása után a tervezett, konkrét pályákat vizsgáljuk meg.

A pályatervezéshez kapcsolódik az időparaméterezés kérdése, amelyet a harmadik fejezetben tárgyalunk. Az időparaméterezés a megtervezett geometriai pályához sebességinformációt rendel, a robot korlátozásait figyelembe véve. Az időparaméterezés kimenete egy időben egyenletesen mintavételezett pálya, amelyet a pályakövető algoritmus használ fel. Itt is konkrét példán mutatjuk meg a megtervezett pályához tartozó sebesség- és gyorsulásprofilokat.

A pályatervezés után áttérünk a pályakövetés ismertetésére. A pályakövetés alapvetően két primitívvel dolgozik: egyhelyben fordulás és pályakövetés. Ezeket tárgyaljuk részletesen a negyedik fejezetben. A fejezet során ismertetjük a sebességszabályozást és az orientációszabályozást, valamint az ezekhez kapcsolódó problémákat. A fejezet végén az algoritmus által felhasznált paraméterek hatásait vizsgáljuk meg.

Végül, az ötödik fejezetben a pályatervező és mozgásirányító algoritmusokat rendszerbe foglaljuk. Ehhez kapcsolódóan röviden említést teszünk egy másik pályatervező eljárásról, a C*CS algoritmusról. Szintén ebben a fejezetben mutatjuk be az általam használt V-REP robotszimulátort, a szimulációs eredményeket és a szimulátorhoz kifejlesztett keretrendszerét. A dolgozat végén a valós robotot ismertetjük, különösképpen a robot irányítórendszerét. Az általam implementált algoritmusokat ebbe az irányítóprogramba kellett integrálni. A dolgozat zárásaként a valós roboton mért eredményeket vizsgáljuk meg.

1. fejezet

Elméleti bevezetés

1.1. Problémafelvetés

A helyváltoztatásra képes, úgynevezett mobil robotok esetében alapvető szituáció, hogy a robotnak a feladata végrehajtásához el kell jutnia egy célpontba. Ehhez önmagának kell az adott környezetben megterveznie a pályát és emberi beavatkozás nélkül kell sikeresen eljutnia a kívánt célpontba. A probléma nagyságrendileg nehezebb, amikor a robot környezetében akadályok is találhatóak.

Céлом azon megközelítések és módszerek áttekintése, amelyek megoldást nyújtanak az autonóm pályatervezés kérdésére. Néhány módszert részletesen is ismertetek, ezeket szimulátoron és valós roboton is implementáltam, illetve teszteltem. A pályatervezéshez szorosan kapcsolódó téma a mozgásirányítás, amivel szintén foglalkoznunk kell, hogy valós környezetben ténylegesen használható eljárásokat kapjunk.

1.2. Pályatervezés elmélete

Az elmúlt időszakban a pályatervezéssel igen sok kutatás foglalkozott [1]. Ahhoz, hogy ezeket az algoritmusokat ismertessük, be kell vezetnünk néhány alapvető fogalmat.

1.2.1. Alapvető fogalmak

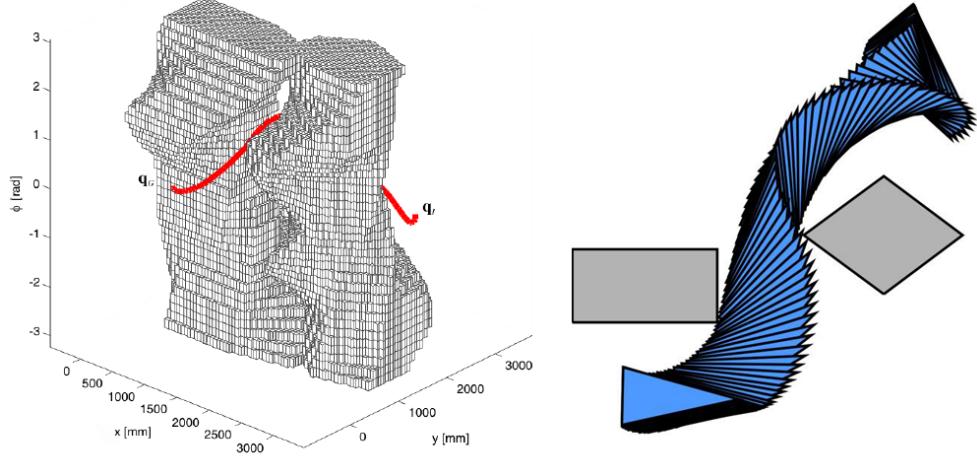
A pályatervezés során a robot pillanatnyi állapotát a *konfigurációjával* írhatjuk le. Síkban mozgó robotok esetében a konfiguráció a következőket tartalmazza [2]:

$$q = (x, y, \theta), \quad (1.1)$$

ahol q a robot konfigurációja, x, y határozza meg a robot pozícióját a síkon és θ határozza meg a robot orientációját.

Egy lehetséges környezetben a robot összes állapotát, a *konfigurációs tér* adja meg, amit C -vel jelölünk. A konfigurációs tér azon részhalmazát, amely esetében a robot a környezetben található akadályokkal nem ütközik, *szabad (konfigurációs) térnek* nevezzük (C_{free}). E halmaz komplementere azokat a konfigurációkat tartalmazza, amelyek esetén a robot

akadályokkal ütközne ($C_{obs} = C \setminus C_{free}$). A konfigurációs teret az 1.1. ábrán szemléltetjük.



1.1. ábra. Konfigurációs tér szemléltetése egy adott útvonal során (bal oldali ábra). A konfigurációs térben a piros vonal jelzi a robot útját a célpontja felé. A síkbeli robot mozgása felülnézetből a jobb oldalon látható [3].

A korlátozások ismerete alapvető fontosságú a pályatervezés és mozgásirányítás során. A környezetben elhelyezkedő akadályokat *globális korlátozásoknak* tekintjük, a robothoz kapcsolódó korlátozásokat pedig *lokális korlátozásoknak* [2]. A lokális korlátozásokat a robot konfigurációs változóinak differenciál-egyenletével írhatjuk le, ezért gyakran nevezik őket *differenciális korlátozásoknak* is. Differenciális korlátozások vonatkozhatnak sebesség (kinematikai) és gyorsulás mennyiségre is (dinamikai korlát).

Dolgozatunkban kinematikai korlátozásokkal fogunk foglalkozni, dinamikai korlátokkal nem.

Egy autó esetén mindenki számára egyértelmű, hogy csak bizonyos íveken tudunk mozogni, egy adott konfigurációból nem tudunk a konfigurációs tér bármely irányába elmozdulni, habár a szabad tér bármely konfigurációjába eljuthatunk. Autónál emiatt nem olyan egyszerű például a párhuзamosan parkolás. Azokat a robotokat, amelyek ehhez hasonló korlátozásokkal rendelkeznek, *anholonom rendszereknek* nevezzük. Anholonom korlátozásról akkor beszélünk, ha a korlátozás olyan differenciálegyenlettel írható le, amely nem integrálható.

Az általunk vizsgált robot típus, a *differenciális robot* is anholonom rendszer. Viszont léteznek olyan robotok, amelyek nem rendelkeznek anholonom korlátozásokkal (holonom rendszerek), ilyenek például az omnidirekcionális robotok. Egy omnidirekcionális robot képes bármilyen konfigurációból a tér bármely irányába elmozdulni.

Robotmodell

A differenciális robotot egy közös tengelyen lévő két kerék segítségével mozgatjuk, így a differenciális meghajtásra utal a differenciális robot elnevezés. A mozgás kinematikai leírását az 1.2. egyenletek adják meg [4].

$$v = \frac{v_r + v_l}{2} \quad (1.2)$$

$$\omega = \frac{v_r - v_l}{W},$$

ahol v a robot sebessége, v_r a robot jobb kerekének sebessége, v_l a robot bal kerekének sebessége, ω a robot szögsebessége és W a robot kerekei közti távolság. Innen a robot mozgás-egyenletei:

$$\begin{aligned}\dot{x} &= v \cos \theta \\ \dot{y} &= v \sin \theta \\ \dot{\theta} &= \omega,\end{aligned}\quad (1.3)$$

Ezt a robot típust gyakran alkalmazzák, mivel egyszerű felépítésű és anholonom mivolta ellenére könnyedén irányítható. Ahogyan az 1.2. egyenletekből is következik, a robot képes egyhelyben megfordulni, ami sok esetben előnyös, például párhuzamos parkolás esetén.

1.2.2. Pályatervezők osztályozása

Mielőtt belekezdenénk az általunk megvizsgált pályatervező algoritmusok részletesebb ismertetésébe, tekintsük át az irodalomban található, elterjedtebben használt módszereket.

Geometriai tervezés szerinti csoportosítás

A pályatervezők geometriai módszerei szerint alapvetően két csoportot különböztetünk meg: a *globális tervezők* és a *reaktív tervezők* csoportját [2].

A globális tervezők esetében a konfigurációs tér egészét figyelembe vesszük a tervezéskor, míg a reaktív tervezők csupán a robot környezetében lévő szűkebb tér ismeretére építenek. A globális tervezők előnye, hogy képesek akár optimális megoldást is találni, míg a reaktív tervezők egy lokális minimumhelyen ragadhatnak, nem garantálható, hogy a robot eljut a célponthoz. A globális tervezés hátránya azonban a lényegesen nagyobb futási idő, ezért gyakran változó vagy ismeretlen környezet esetén előnyösebb lehet a reaktív tervezők használata.

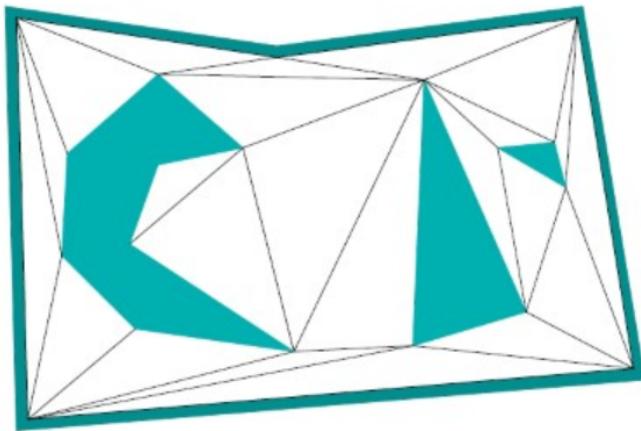
A reaktív tervezők esetében a robot alakját körrel szokták közelíteni, ezzel is egyszerűsítve a tervezés folyamatát. Ezzel szemben globális algoritmusok a robot pontos alakját figyelembe veszik, aminek nagy jelentősége van szűk folyosókat tartalmazó pálya esetén. Az általunk bemutatott algoritmusok esetén mi is figyelembe vesszük a robot pontos alakját.

A globális tervezők esetén megkülönböztetünk mintavételes és kombinatorikus módszereket [1]. A mintavételes módszerek a konfigurációs teret véletlenszerűen mintavételezik és ez alapján próbálnak utat keresni a célpontba, sok-sok iteráción keresztül. Ellenben a kombinatorikus módszerek direkt módon, a környezet pontos geometriai modellje alapján terveznek utat. Ennek az az előnye a mintavételes módszerrel szemben, hogy képesek

megállapítani, hogy létezik-e egyáltalán megoldás az adott környezetben. A mintavételest tervezők esetén viszont nem tudjuk ezt eldöntení.

A globális tervezők sok esetben topologikus gráfokat (speciális esetben fák) használnak a konfigurációs tér struktúrájának leírásához [5, 6]. Az általunk bemutatott globális tervező, az RTR algoritmus is ezt a megközelítést alkalmazza. A gráfok (ill. fák) csomópontjai a konfigurációs térből vett minták, élei pedig az ezeket összekötő ütközésmentes pályaszakaszok.

Kombinatorikus globális tervezők közé tartoznak például a celladekompozíció alapuló megoldások. Ezeknél az algoritmusoknál a szabad konfigurációs teret cellákra bontjuk fel, és e cellák alapján felépítünk egy a konfigurációs teret leíró gráfot. Az RTR algoritmusnál én is használok celladekompozíciót, a esetben a cellák háromszögek, és a háromszögek oldalfelező pontjai alkotják a gráf csúcsPontjait. A celladekompozíciót megvalósító algoritmushoz Bernhard Kornberger munkáját használtam fel [7].



1.2. ábra. Celladekompozíció háromszögek segítségével.

Irányított rendszer szerinti csoportosítás

A robotok, mint irányított rendszerek esetén megkülönböztetjük az anholonom és holonom rendszereket a pályatervezők csoportosítása esetén is. Anholonom rendszerek esetén önmagában a robot állapotváltoztatása sem triviális feladat. Azokat az eljárásokat, amelyek képesek egy anholonom rendszert egy kezdő konfigurációból egy cél konfigurációba eljuttatni az akadályok figyelembe vétele nélkül, *lokális tervezőknek* hívjuk [1].

Gyakran már a globális tervező figyelembe veszi a robot korlátozásait, és ennek megfelelő geometriai primitíveket használ, vagy esetleg egy lokális tervezőt használ minden állapotváltoztatásra.

Az általunk vizsgált RTR tervező egyenes mozgással és egy helyben fordulással dolgozik. Ezek a mozgásprimitívek ideálisak egy differenciális robot számára [1], így az RTR pályatervező által tervezett pályát módosítás nélkül végre tudja hajtani egy differenciális robot.

A globális tervező által megtervezett pályát közelíthetjük egy lokális tervezővel, ha az anholonom robotunk közvetlenül nem tudná lekövetni a globális tervező pályáját. Ezt az

eljárást, *approximációs módszernek* nevezik. Az 5.1. fejezetben felhasználunk egy lokális tervezőt, amely képes egy előzetes út alapján olyan utat generálni, amely akár autószerű robot számára is végrehajtható. Az előzetes pályát valamilyen globális tervező szolgáltatja, a mi esetünkben az RTR pályatervező. Habár egy differenciális robot közvetlenül már az RTR eljárás által generált pályát is le tudja követni, de az approximációs módszer által tervezett pályán lényegesen gyorsabban tud végighaladni.

2. fejezet

Az RTR algoritmus

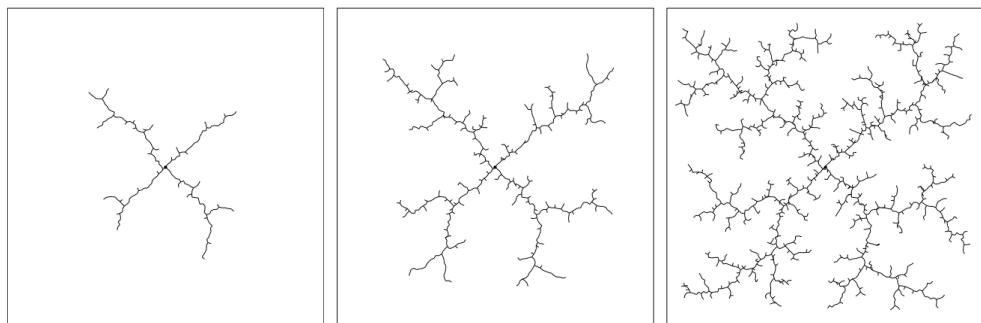
Az RTR (Rotate-Translate-Rotate) algoritmust Kiss Domokos dolgozta ki [8]. A feladatom az algoritmus implementálása volt C++ nyelven, majd az algoritmus tesztelése szimulációs környezetben és valós roboton.

Az RTR eljárás a szakirodalomból népszerű RRT algoritmuson alapszik, ezért ennek a bemutatásával kezdjük a fejezetet.

2.1. RRT algoritmus

Az RRT (Rapidly Exploring Random Trees) algoritmus lényege, hogy a kezdeti konfigurációból egy fát építünk a szabadon bejárható konfigurációs térben [6]. A fa csomópontjaiban konfigurációk találhatóak, és a fa terjesztését úgy irányítjuk, hogy a kívánt célkonfiguráció felé tartson. Ha a fa ténylegesen eléri a célkonfigurációt, akkor az utat a kezdeti konfigurációból a célkonfigurációba már könnyedén megkaphatjuk.

Létezik olyan változata az RRT algoritmusnak, ahol nemcsak a kezdeti konfigurációból építünk fát, hanem a célkonfigurációból, vagy akár több köztes pontból is.



2.1. ábra. Az RRT algoritmus három különböző iterációnál [6].

A fa építés úgy kezdődik, hogy véletlen konfigurációkat veszünk a környezetből (q_{rand}). Ezt hívják *mintavételezési szakasznak*. Ezután meghatározzuk, hogy a fában melyik konfiguráció van a legközelebb a mintavételezett konfigurációhoz (q_{near}). Ez a *csomópont kiválasztó szakasz*. Egy lehetséges megoldás, hogy a fa csomópontjaiból választunk konfigurációkat, vagy előfordulhat, hogy a fa csúcspontjai közötti élek egy köztes konfigurációját

választjuk.

A következő lépésben megpróbáljuk a q_{rand} és a q_{near} konfigurációkat interpolációval összekötni (*összekötő szakasz*). Itt több variációja is létezik az RRT algoritmusnak. Előfordul, hogy q_{near} konfigurációból csak egy bizonyos fix Δq értékkal közelítünk q_{rand} felé. A másik esetben addig terjesztjük a fát, amíg el nem érjük q_{rand} konfigurációt, vagy amíg nem ütközik a robot. Ebben az esetben a kapott konfiguráció a legmesszebb található ütközésmentes konfiguráció lesz q_{rand} irányában. Az újonnan kapott konfigurációt végül hozzáadjuk a fához.

Anholonom rendszerek esetén is használható az RRT eljárás. Ekkor az összekötésnél egyszerű interpolációt nem lehet alkalmazni, mert az azt feltételezné, hogy a robot minden irányba szabadon képes mozogni. Ehelyett az összekötést egy lokális tervező segítségével kell megoldani, vagy egyszerűbb esetben itt is használható az a módszer, hogy csak egy adott Δq értékkel közelítünk q_{rand} irányába. Ehhez megfelelő beavatkozó jelet (Δu) kell választanunk, amit Δt ideig alkalmazva elérhető Δq állapotváltozás. A Δu beavatkozójel például differenciális robot esetén, a két kerék sebessége.

Az előbb ismertetett fázisok alkotják a fa terjesztésének egy lépését. Természetesen, több fa esetén mindegyiknél végre kell hajtani a fázisokat. A terjesztést addig kell folytatni, amíg el nem érjük a kívánt konfigurációt, vagy több fa esetében, amíg a fák nem kapcsolhatók össze.

2.2. RTR algoritmus

Ha differenciális robotnál használunk valamilyen Δu beavatkozójelet az összekötő fázisban, akkor a fa csúcspontjai között görbek lesznek. Ez nehézséget okozhat, ha olyan *csomópont-választó eljárást* alkalmazunk, ami köztes konfigurációt ad vissza. Természetesen, alkalmazhatjuk azt az eljárást, hogy csak az élek végpontjait választjuk ki, köztük nem interpolálunk. Ehhez viszont kis távolságú élek szükségesek, ami növeli a fa csomópontjainak számát és ezzel összefüggésben a csomópont kiválasztások számát is.

Lehetőségünk van differenciális robotnál is lokális tervezőt alkalmazni két konfiguráció közti állapotváltozásra. A legegyszerűbb lokális tervező három lépésből áll:

- Egyhelyben fordulás a kívánt konfiguráció irányába (R).
- Mozgás egyenes pályán a célpozícióba (T).
- Egyhelyben fordulás a célkonfiguráció irányába (R).

Ennek a tervezőnek az az előnye, hogy a fa élei egyenes pályák lesznek, így egyszerűen tudjuk meghatározni a köztes konfigurációkat.

A jelenleg ismertetett módon alkalmazva az RRT algoritmust, szűk folyosók esetében rendkívül nehezen találna megoldást az eljárás, abban az esetben is, ha mind a kezdeti-, mind a célkonfigurációból növesztünk egy-egy fát. A problémát az okozza, hogy az *összekötés fázisa* gyakran nem ad eredményt, ezért a fák nem nőnek megfelelően. Ennek az az oka, hogy a lokális tervező használatakor, fal vagy egyéb akadályok közelében az első

egyhelyben fordulásnál már ütközne a robot. Mivel az összeköttetés fázisa addig tart, amíg nem érjük el q_{rand} -ot, vagy amíg nem ütközik a robot, így a fa további terjesztése nélkül választunk új q_{rand} értéket. A lokális tervező második lépése eredményezné a fa tényleges terjesztését.

Az RTR algoritmus, felhasználva az RRT eljárás előnyös tulajdonságait, igyekszik az előbbi problémára egy lehetséges megoldást bemutatni. Mind a kezdeti, mind a célkonfigurációból növeszt egy-egy fát, az összekötő fázisban a fent ismertetett lokális tervezőt alkalmazza. Az RRT eljárás minden fázisára módosításra kerül az RTR tervező esetében.

Az RTR algoritmusnál alkalmazott fa struktúrában, az RRT-hez hasonlóan a csomópontokban konfigurációk találhatóak, az élek pedig transzlációs mozgást (TCI - Translation Configuration Interval) vagy egyhelyben fordulást írnak le (Rotational Configuration Interval). Egy adott TCI vagy RCI leírható két konfigurációval és TCI esetén a köztük lévő távolsággal, míg RCI esetén a köztük lévő szögtávolsággal. Természetesen RCI esetén a két konfiguráció pozíciója megegyezik, TCI esetén pedig a két konfiguráció iránya egyezik meg. Fontos észrevennünk, hogy mindenkét éltípuszt egzakt módon írjuk le, mintavételezés nélkül.

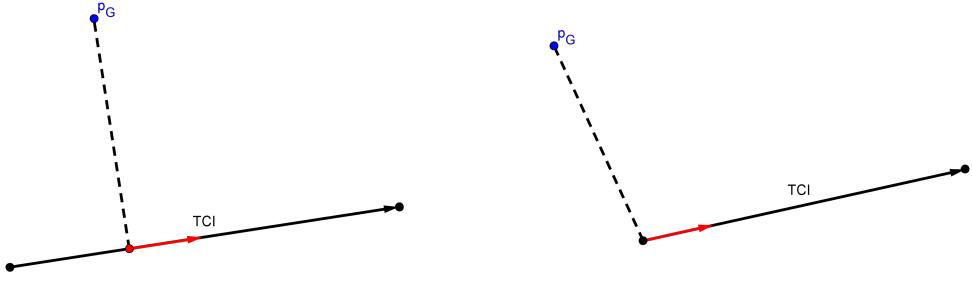
2.2.1. Mintavételezés

A mintavételezés fázisában különbséget jelent az eredeti RRT algoritmushoz képest, hogy a mi esetünkben a q_{rand} -nak megfelelő véletlen minta nem egy konfiguráció lesz, hanem egy pozíció a térben (p_G). Ezt a pozíciót tekintetük egy folytonos, egydimenziós konfigurációs listának, amelynek bármelyik eleme megfelelő célkonfiguráció lehet.

A mintavételezést kiegészítjük a pálya háromszög cellafelbontásából kapott mintákkal (1.2.2. fejezet). Természszerűleg ezek a minták az akadályuktól viszonylag távol találhatóak, és szűk folyosók esetén is segítenek terjeszteni a fákat.

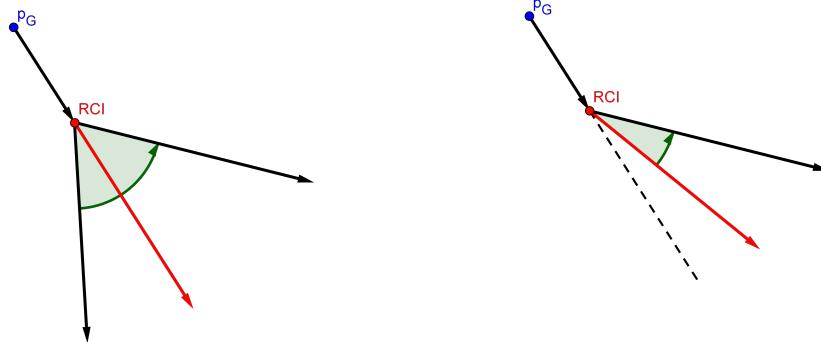
2.2.2. Csomópont kiválasztás

Mivel a mintavételezésnél pozíciót használunk és nem konfigurációt, ezért a *csomópont kiválasztás* egyszerűbb lesz. Az eljárás lényege, hogy az adott p_G esetén végigmegyünk a fa élein és meghatározzuk a legkisebb távolságot a p_G pont és az adott él között. TCI él esetében ez távolságot jelent, RCI esetében pedig szögtávolságot. Így minden egyes élnél kapunk egy konfigurációt, amely esetében a távolság a p_G -től minimális. A kapott konfigurációk közül azt választjuk, amelyiknél legkisebb a távolság p_G -hez képest. Abban az esetben, ha így több megoldást is kapunk, akkor azt a konfigurációt választjuk, amelynél a szögtávolság a legkisebb. Így egyértelműen meghatároztuk q_{near} -t.



2.2. ábra. Csomópont kiválasztás TCI esetén. A bal oldali ábra esetén köztes konfigurációt kapunk, míg a jobb oldali ábrán nem.

Egy adott TCI és p_G esetén a legkisebb távolsághoz tartozó konfigurációt a következőképpen határozzuk meg (2.2. ábra). Kiszámoljuk a p_G pont merőleges vetületét a TCI-t alkotó egyenesre, ezután meghatározzuk, hogy a vetület a TCI-n, mint szakaszon belül található-e. Ha a szakaszon belül található a vetület, akkor egy köztes konfiguráció van legközelebb p_G -hez. A köztes konfiguráció pozíciója a vetület, orientációja pedig a TCI orientációja. Ha a szakaszon kívül található a vetület, akkor a TCI közelebbi konfigurációja lesz a legkisebb távolságú konfiguráció.



2.3. ábra. Csomópont kiválasztás RCI esetén. A bal oldali ábra esetén köztes konfigurációt kapunk, míg a jobb oldali ábrán nem.

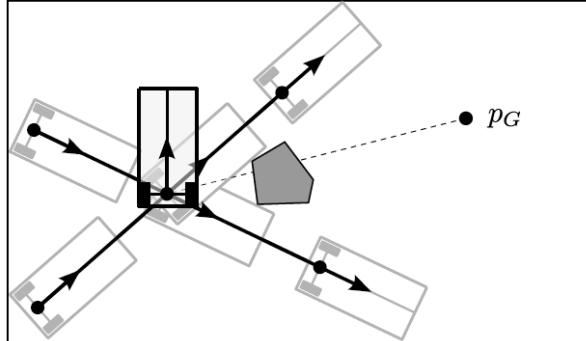
RCI esetén a legkisebb (szög)távolságú konfiguráció kiválasztása a 2.3. ábrán látható. Először kiszámoljuk a p_G pont és az RCI pozíójának irányát. Ha a kapott irány az RCI (szög)tartományába beleesik, akkor köztes konfigurációról van szó (bal oldali ábra) és ekkor a legközelebbi konfiguráció az RCI pozíciója és az előbb kiszámolt orientáció lesz. Abban az esetben, amikor a kapott irány nem esik bele az RCI tartományába, akkor az RCI irányban közelebbi, szélső konfigurációját választjuk (jobb oldali ábra).

2.2.3. Kiterjesztés

Az RTR algoritmus ezen fázisa különbözik leginkább az eredeti RRT algoritmustól. Küllönbség, hogy a transzlációs szakasz esetében nemcsak előre, hanem hátra is kiterjesztjük a fát. Ezenkívül a kiterjesztés nem p_G -ig történik, hanem mindenképpen addig, amíg nem

ütközik a robot. Ez minden két irányra érvényes. Fontos megjegyezni, hogy a kiterjesztés során az RTR lokális tervező első két elemét használjuk fel (RT), tehát a második forgatást nem hajtjuk végre.

Lényeges különbség az is, ahogyan a két algoritmus az ütközést kezeli. Az RTR tervező esetén ütközés esetén különbséget kell tenni, hogy transzlációs vagy forgatási fázisban történt-e ütközés. Amennyiben transzlációs fázisban történt ütközés, az adott iterációnak vége lesz, hiszen a fát már kiterjesztettük. Ugyanez történne az RRT eljárásnál is.



2.4. ábra. A kiterjesztés folyamata./8/

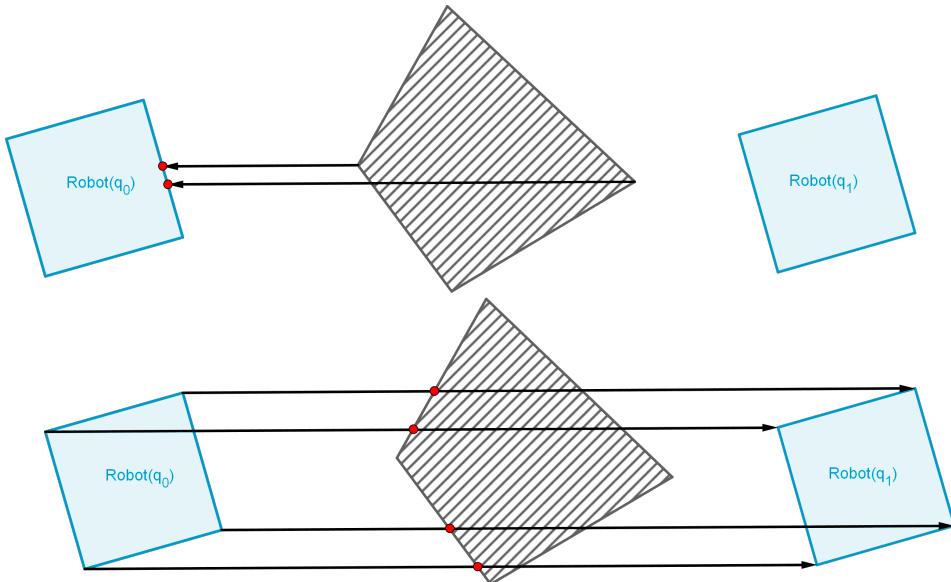
Amennyiben forgatás közben történik ütközés a helyzet bonyolultabb, mivel ilyenkor nem sikerült kiterjeszteni még a fát. Ekkor az ütközési orientációt előre és hátra terjesztjük ki a fát. Ezután a célorientációt megróbáljuk másik körüljárás szerint elérni. Függetlenül attól, hogy sikerült-e a második forgatással elérni a célkonfigurációt vagy ütközött a robot, ebben az állapotban megint kiterjesztjük a fát előre és hátra is. Amennyiben ütközés történt megint az ütközési orientációt alkalmazzuk a kiterjesztést, különben pedig a célkonfiguráció irányában terjesztjük ki a fát. A folyamat a 2.4. ábrán látható.

A fa terjesztésnél alapvetően két célunk van, egyszerűbb, hogy a célkonfiguráció felé haladjunk, másrészről pedig, hogy minél nagyobb szabad területet járunk be.

Ütközésdetektálás transzlációs kiterjesztés esetén

A robot transzlációs kiterjesztése előre- és hátrafelé is a következők szerint történik. minden egyes akadály esetén, beleértve a pályát határoló téglalapot is, végigmegyünk a robot alakját leíró polygon összes csúcsponján. Adott csúcspon esetén megvizsgáljuk, hogy a kiterjesztés irányában metszi-e a vizsgált akadály oldalát: ha igen, akkor eltároljuk az ütközésig megtehető távolságot. Ezután megvizsgáljuk, hogy az adott akadály csúcsponjai a mozgás során metszik-e a robot élét: ha igen, akkor szintén eltároljuk az ütközésig lévő távolságot. A minimális távolság a letárolt távolságok minimuma lesz.

A fent leírtaknál minden esetben egy egyenes és egy szakasz metszetét kell kiszámítanunk, majd megvizsgálni, hogy a metszéspont a mozgás irányában található-e. A szakasz a robot vagy az akadály egyik élé, az egyenes pedig az akadály vagy a robot egyik csúcsponja a transzláció irányában. A 2.5. ábra mutatja a kiterjesztést egy adott akadály és a robot között.



2.5. ábra. Transzlációs kiterjesztés esetében az ütközésvizsgálat.

Ütközésdetektálás forgatás esetén

A forgatás közbeni ütközésvizsgálat hasonlóképpen történik, mint transzlációnál. Itt is végigmegyünk az összes akadályon és mindegyiknél megvizsgáljuk a robot összes élét és csúcsPontját. Itt is két irányban ellenőrzünk, a robot éleit és az akadály csúcsPontjait, valamint az akadály éleit és a robot csúcsPontjait vizsgáljuk. A kapott szögelfordulások közül a minimális elfordulást használjuk fel később.

Különbséget jelent, hogy itt egy körív és egy szakasz közti metszéspontot kell számolnunk. A szakasz a robot vagy akadály egyik éle. A körivet pedig úgy kapjuk meg, hogy a robot vagy akadály csúcsPontját a robot pozíciója körül elforgatjuk az adott forgási szöggel.

2.2.4. Útvonal meghatározása, optimalizálása

Az RTR algoritmus sikeres futásához szükséges, hogy a kezdő és a célkonfigurációból terjesztett fák kapcsolódjanak egymáshoz. Ezért minden iteráció végén ellenőrizzük, hogy a legutóbb felvett TCI-k és a másik fa között létesíthető-e ütközésmentes kapcsolat egy RCI segítségével. Ha ez lehetséges, akkor a metszéspontból vissza kell mennünk a fák kezdőpontjáig, és így megkapjuk a keresett útvonalat a kezdő konfigurációból a célkonfigurációig. A kapott útvonalat egy TCI-ket és RCI-ket tartalmazó listaként képzelhetjük el.

Egy TCI és egy fa összekapcsolásának vizsgálata során a fát Breadth-First szélességi keresés segítségével járjuk be [9]. Az algoritmus egy FIFO szerkezetű tárolót használ, amely kezdetben a fa forrásainak gyerekeit tartalmazza. Ezután kivesszük a tárolóból az első elemet: ha az TCI, akkor megvizsgáljuk, hogy összevonható-e a vizsgált TCI-vel. Ha összevonható, akkor befejeztük a vizsgálatot, ha nem pedig a TCI gyerekeit berakjuk a tárolóba. Ha RCI volt a kivett elem, akkor egyszerűen a gyerekeit berakjuk a tárolóba. Az algoritmust addig futtatjuk, amíg a tároló nem ürül ki vagy nem találtunk összevonható TCI-ket. Abban az esetben ha üres lesz a tároló, a vizsgált TCI és a fa nem vonható össze.

Azt, hogy két TCI összevonható-e, egyszerűen eldönthetjük. Fogjuk a két TCI által leírt szakaszokat és megnézzük, hogy van-e metszéspontjuk. Ha nincsen, akkor biztosak lehetünk benne, hogy nem vonhatóak össze. Ha van metszéspontjuk, meg kell vizsgálnunk, hogy a metszéspontból találunk-e egy RCI-t, amely a két szakasz között ütközésmentes mozgást biztosít. Itt ugyanúgy járhatunk el, ahogy a kiterjesztés fázisában tettük a 2.2.3. részben. Ha nem sikerül ütközésmentes forgatást találnunk, ellentétes körüljárás szerint is megpróbálunk RCI-t keresni, ha így sem találunk, akkor a két TCI nem összevonható.

Abban az esetben, ha több útvonalat is találnánk a két fa között, akkor azt az útvonalat választjuk, amelyik a legkevesebb konfigurációból áll. Ha több ilyen út is van, akkor azok közül a legrövidebb távolságú útvonalat választjuk.

Miután megkaptuk az útvonalat, még további optimalizációt végezhetünk el rajta. Az első módszer lényege, hogy az egymás után következő több TCI-t, egy TCI-vel helyettesítjük. Ezt minden esetben megtehetjük, hiszen ha a TCI-k között nincsen RCI, akkor azoknak az iránya nem változik, egy egyenesen helyezkednek el.

A második lehetőség, hogy az összes TCI-t a kapott útvonalban kiterjesztjük, majd az így létrehozott kiterjesztett útvonal segítségével optimalizálunk. Felmerülhet, hogy miért van értelme kiterjeszteni a TCI-okat, hiszen ezeket úgy hoztuk létre, hogy nem p_G pozícióig történik a terjesztés, hanem amíg a robot nem ütközik. Azonban a *csomópont kiválasztásnál* köztes konfigurációkat is kiválasztunk, és így olyan TCI-k jönnek létre, amelyek nincsenek ütközésig terjesztve. Az optimalizálás lényege, hogy végigmegyünk a kiterjesztett útvonalon, és minden TCI esetében a kiterjesztett pálya végéről elindulva olyan TCI-kat keresünk, amelyeket össze lehet vonni. Két TCI-ról a fent ismertetett módon döntjük el, hogy összevonhatók-e. Ezzel a módszerrel a végleges útvonal hossza jelentősen csökkenthető.

2.3. Implementálás

Az algoritmus megvalósítását C++ nyelven végeztem el. Az implementáláshoz rendelkezésre állt az RTR tervező MATLAB környezetben megvalósított programkódja.

Az RTR tervező egyszerűsített pszeudokódja az 1. algoritmusnál látható. Az algoritmus felépítése hasonló az eredeti RRT eljáráshoz [6].

Az algoritmus minden iterációban továbbterjeszti a konfigurációs fákat. A terjesztés minden fa esetében ugyanúgy történik (*IterateTree* metódus). Látható, hogy az *IterateTree* metódus az RTR eljárás három fő lépését hajtja végre. A *GetGuidePoint* függvény a szabad konfigurációs térből véletlenszerűen kiválaszt egy pozíciót (mintavételezési szakasz), a *GetALC* függvény megkeresi az eddig bejárt fában a mintavételezett ponthoz tartozó legközelebbi konfigurációt (csomópont kiválasztás) és a *TurnAndExtend* függvény felelős a fa kiterjesztéséért.

A *MergeTreesGetPath* és a *OptimizePath* függvények a 2.2.4. részben leírtakat valósítják meg. A *MergeTreesGetPath* a két fa között próbál összekötést találni egy, míg a *OptimizePath* a már megkapott útvonalt optimalizálja, hogy minél rövidebb legyen.

Algoritmus 1 RTR tervező

```
procedure RTRPLANNER
    INITTREE(start)
    INITTREE(goal)
    while iteration ≤ maxIter do
        ITERATETREE(start)
        ITERATETREE(goal)
        if MERGETREESGETPATH() then
            OPTIMIZEPATH()
            break
        end if
    end while
end procedure

procedure ITERATETREE(tree)
    pG ← GetGuidePoint(tree)
    alc ← GetALC(pG, tree)
    if TURNANDEXTEND(pG, tree, alc, positive) then
        TURNANDEXTEND(pG, tree, alc, negative)
    end if
end procedure
```

A konfigurációs fa elemeinek leírására a *TreeElement* osztály szolgál, amely a következő tagokkal rendelkezik:

parentElement : A szülő azonosítója
childrenElements : A fa elemének gyerekeit tartalmazó vektor
CI : A fa elemét meghatározó TCI vagy RCI leírása (2.1)

A *TreeElement* osztályban található *CI* tagváltozót pedig a *ConfigInterval* osztály írja le:

type : A CI tipusa: RCI vagy TCI
q0 : A CI kiindulási konfigurációja
q1 : A CI cél konfigurációja
amount : Előjeles távolság a két konfiguráció között (2.2)

Ezenkívül természetesen külön osztályt alkotnak a konfigurációk, a különböző geometriai primitívek (kör, egyenes, háromszög, sokszög, pont). Ezeknek a pontos leírása a dolgozat mellékletében megtalálható a forráskódokban.

Mind a két folyamatosan növesztett fát *TreeElement*-ekből álló dinamikus tömbben tárolom, amire a C++ nyelv *Vector* tároló típusát használom. A *Vector* típus egy tömbben, folytonos memória területen tárolja az elemeket. A dinamikus memória foglalást és felsza-

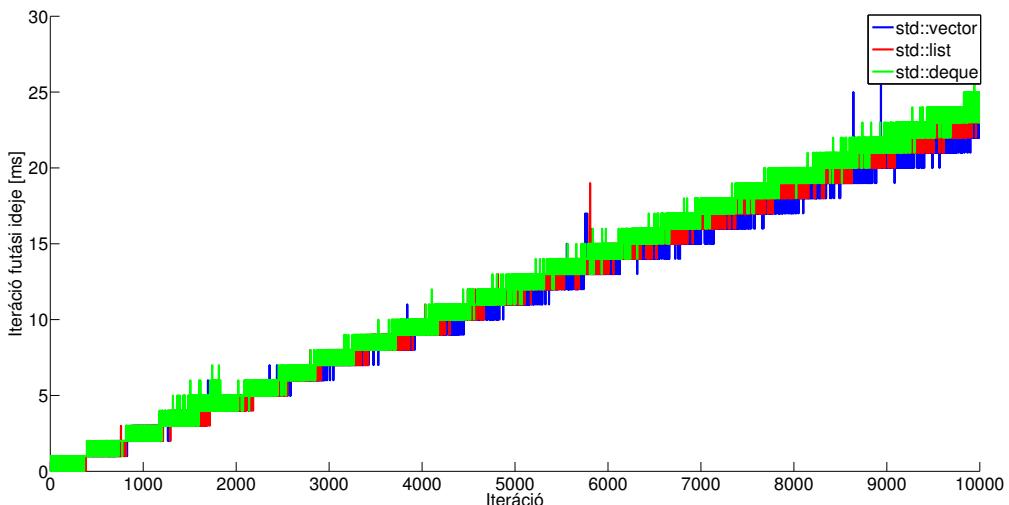
badítást a tároló automatikusan elvégzi.

Kipróbáltam más tárolókat is, mint például a *List* típust. A *List* egy kétszeresen láncolt lista. Adott elem törlése és beszúrása *List* esetén gyorsabb, mint *Vector* esetében. A mi esetünkben azonban nem szükséges a tömbbe közbülső elemeket beszúrni, az új elemek a tömb végére kerülnek.

A *Deque* típus hasonló a *Vector*-hoz, csak ennél a tárolónál a tömb elejére is hatékonyan lehet elemeket beszúrni és törölni, nemcsak a tömb végére. Viszont, a tárolásra nem folytonos memória területet használ.

Az előbb említett három tároló típust egy mérés segítségével is összehasonlítottam. A mérés során fixen 10000 iterációig futott az algoritmus és a *IterateTree* függvény futási idejét mértem minden két fa esetében. Amennyiben futás közben talált volna megfelelő útvonalat az algoritmus, akkor sem fejezte be a futást, 10000 iterációit mindenképpen végrehajtott. A *MergeTreesGetPath* függvényt azért nem mértem a teszt során, mivel a gyakorlatban csak addig fut az algoritmus, amíg nem lesz meg az első megfelelő út, és ez a tény jelentősen befolyásolja a *MergeTreesGetPath* függvény futási idejét.

A mérés elvégzéséhez a C++ nyelvben található, kifejezetten időmérésre kialakított *high_resolution_clock* osztályt használtam, amely a C++11 szabvány része.



2.6. ábra. A különböző C++ tároló típusok összehasonlítása. Az időmérés felbontása 100 ns.

A mérés ábráján (2.6. ábra) látható, hogy mekkora különbséget okoz a választott tároló típus. A mérés alapján úgy gondolom, jó választás volt a *Vector* használata, mert a többi tároló esetében nagyobb futási idő adódik, ha a különbség nem is túlságosan nagy. Általánosságban azt tanácsolják, hogy célszerű *Vector* típust használni tároláshoz, az esetek nagy részében így kapjuk a leghatékonyabb programkódot.

Az ábráról az is leolvasható, hogy a iterációk számának növekedésével (fa terjesztésével) egy iteráció futási ideje lineárisan nő. Ez azért van így, mert a *GetALC* függvénynek végig kell mennie a teljes fán, hogy megállapítsa, hogy melyik konfiguráció van a mintavételezett pozícióhoz legközelebb. Ez az egy függvény okozza azt, hogy a fa növekedésével lineárisan nő a futási idő. Ha azt vizsgáljuk, hogy adott számú iteráció futásához mennyi idő szükséges,

Függvény neve	Mintavétel	Mintavétel [%]	Futási idő [s]
RTRPlanner	5433	100.00	17.36
GetALC	2574	47.37	8.22
MergeTreesGetPath	2205	40.59	7.05
TurnAndExtend	652	12.00	2.08
OptimizePath	2	0.04	0.01

2.1. táblázat. Az algoritmus vizsgálata Profiler segítségével. A méréshez használt pálya a 2.9. ábrán látható.

akkor az négyzetesen fog nőni az iteráció szám növelésével.

Az algoritmus futási idejét a Microsoft Visual Studio beépített Profiler eszközével is megvizsgáltam. A Profiler a program futása során mintákat gyűjt (ezt jelöli a 2.1. táblázat *Mintavétel* oszlopa), és eltárolja minden mintához, hogy a program melyik része futott. Ennél a tesztnél már a teljes algoritmust használtam, nemcsak az *IterateTree* függvényt. Az algoritmus akkor fejezte be futását, ha talált utat a kezdő és cél konfiguráció között.

A Profiler eredményeit a 2.1. táblázat mutatja be. Az eredmények megfelelnek a váratkozásainknak, mivel az idő nagy részében a *GetALC* függvény fut. Ezenkívül a *MergeTreesGetPath* függvényhez tartozik jelentős processzor idő. Ez logikus, hiszen ennél a függvénynél is végig kell menni a teljes, két fán, hogy eldöntsük a két fa összevonhatóságát.

2.3.1. Optimalizálások

Megvizsgálva az algoritmust, a következő egyszerűbb optimalizálásokat végezhetjük el a programon. Mivel a *GetALC* függvény (csomópont kiválasztás) a legkritikusabb függvény, ezért próbáljuk meg elsősorban ezen optimalizálni.

A csomópont kiválasztás lényege, hogy megkeressük a legközelebbi konfigurációt a mintavételezett pozícióhoz (p_G). Legközelebbi konfigurációt távolságban legközelebbit értjük. Abban az esetben, ha több ilyen konfigurációt találunk, az elfordulásban közelebbi konfigurációt választjuk. Ennek a menetét a 2.2.2. fejezetben tárgyaltuk részletesen. A csomópont kiválasztást gyorsíthatjuk, ha az adott TCI/RCI és a p_G pont közötti vizsgálatot abbahagyjuk, ha a kapott távolság biztosan nagyobb lesz, mint az addigi minimális távolságú konfiguráció esetén. Ezzel TCI és RCI esetén nem szükséges kiszámolni a p_G és a konfiguráció közti szögtávolságot, ami jelentős javulást eredményez, mivel a szögtávolság meghatározásához atan ϑ műveletet kell használnunk.

Az algoritmusnál gyakran előfordul, hogy távolságokat kell összehasonlítanunk (*GetALC* esetén is). Ezeknél a részknél élhetünk azzal a triviális optimalizálással, hogy távolság-négyzetet számolunk, tehát megspórolunk egy négyzetgyököt.

Megvizsgálva a 2.2.3. fejezetekben tárgyalt ütközésvizsgálatot, a következő optimalizálást végezhetjük el. Amennyiben a robot egy akadálytól messze található, akkor értelmetlen az egyhelyben fordulásnál ütközésdetektálást végezni. Annak az eldöntése, hogy a robot messze található egy adott akadálytól, a következőképpen történik. Kiszámoljuk az RTR algoritmus elején, hogy mekkora a robot fordulási körének sugara és mekkora az akadályo-

Függvény neve	Mintavétel	Mintavétel [%]	Futási idő [s]
RTRPlanner	2669	100.00	8.55
MergeTreesGetPath	1786	66.92	5.72
GetALC	573	21.47	1.84
TurnAndExtend	307	11.50	0.98
OptimizePath	3	0.11	0.01

2.2. táblázat. Az algoritmus vizsgálata optimalizálás után Profiler segítségével

kat leíró sokszögek köré írható körének sugara. E körök segítségével meg tudjuk határozni egy akadályról, hogy messze található-e a robot aktuális pozíciójától. Ez a megoldás a *MergeTreesGetPath* függvény futási idejét is csökkenti.

Az ismertetett egyszerűbb optimalizálásokkal a 2.2. táblázatban látható futási eredményeket kapjuk a Profiler program segítségével. A táblázat értékei alapján elmondhatjuk, hogy sikerült az algoritmus futási idejét az adott pálya esetében felére csökkenteni. Ezenkívül szembetűnő különbség az optimalizálás előtti eredményhez képest, hogy a *GetALC* függvény futása 60%-ról 40%-ra csökkent. Tehát a csomópont kiválasztásnál alkalmazott optimalizálások, egyszerűségük ellenére, igen hatásosak voltak.

Az algoritmus futási idejét összehasonlíthatjuk az implementálás alapjául szolgáló MATLAB szkript futási idejével. Egy konkrét pálya esetében a Matlab szkriptben megvalósított algoritmus átlagosan 24 s alatt futott le, míg a C++ nyelvű megvalósítás esetén 21 ms alatt. Tehát körülbelül 3 nagyságrenddel lett gyorsabb a C++ változat.

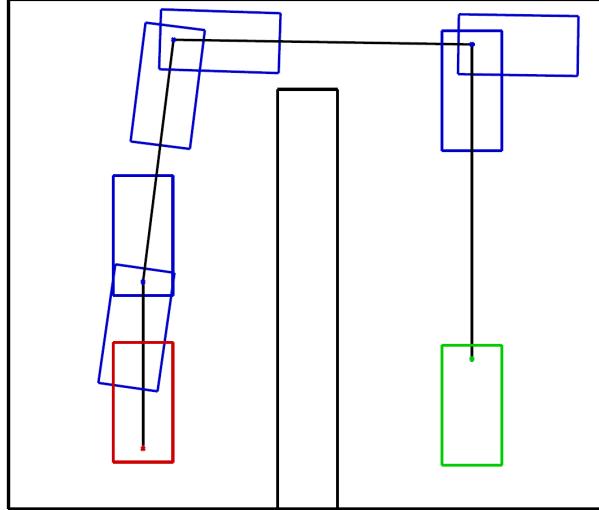
Természetesen a MATLAB szkriptben megírt változat elsődleges feladata az algoritmus bemutatása, nem a minél rövidebb futási idő elérése volt. Azonban bonyolultabb pályák esetében igazán hasznos a három nagyságrenddel rövidebb futási idő, például a Profiler által az előbbieken tesztelt pálya esetében MATLAB szkript esetén hozzávetőleg két és fél óra lenne a futási idő.

Véleményem szerint a két megvalósítás közötti különbség legfőképpen abból adódik, hogy MATLAB esetében interpretált módon hajtjuk végre a programkódot, míg a C++ nyelvnél a programkód fordításra kerül, ráadásul a fordító optimalizálás után készíti el a processzor által végrehajtandó gépi kódot.

2.4. Eredmények

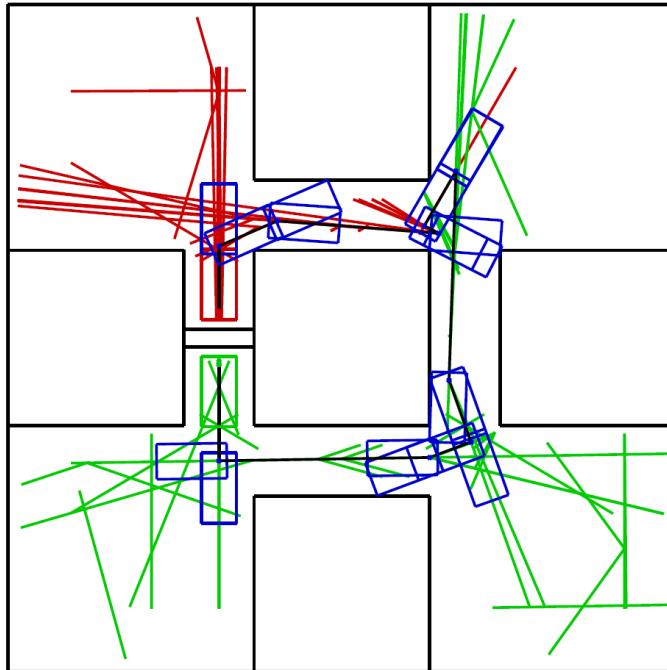
Most pedig bemutatjuk az RTR pályatervező által generált pályákat. A 2.7. ábrán látható az algoritmus által tervezett pálya, valamint a pálya sarokpontjaiban a robot helyzete. A kezdő- és végkonfigurációt külön kiemeltük zöld és piros színekkel.

Szimulációs eredmények alapján belátható, hogy az RTR tervező szűk folyosókat tartalmazó pályák esetében jobb eredményt ad, mintha az egyszerű RRT eljárást alkalmaznánk differenciális robotnál [8].



2.7. ábra. Az RTR algoritmus által megtervezett útvonal egy akadály esetén.

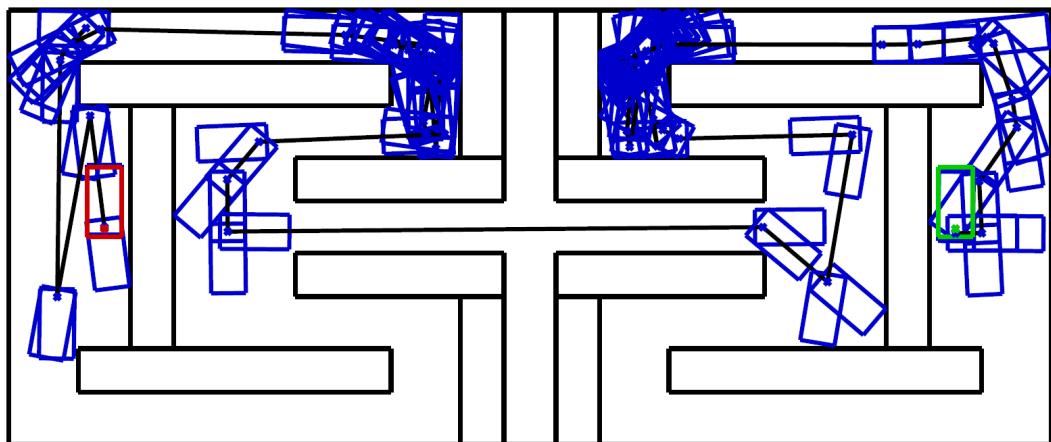
Bonyolultabb környezet esetén felerősödik a véletlen mintavételezés szerepe, az egymás utáni futtatások között nagyobb eltérés mutatkozhat. Az RTR tervező a 2.8. ábrán látható útvonalat 35 iteráció után találta meg, de előfordult, hogy ugyanennél a pályánál, egy hasonló úthoz 130 iterációra volt szüksége. Ebben az esetben a kezdő és végkonfigurációból terjesztett fákat is ábrázoltuk.



2.8. ábra. Az RTR algoritmus által megtervezett útvonal bonyolultabb környezet esetén.

Az RTR algoritmus futási eredményeit (a 2.1. és a 2.2. táblázat), a 2.9. ábrán látható pálya esetében vizsgáltuk. Látható, hogy a pálya két részén igen szűk kanyarban kell elfordulnia a robotnak. Ekkor a robot nagyon kis (1 milliméter alatti) mozgásokat hajt végre. Összehasonlításképpen a pálya 6 méter hosszú és 2.5 méter széles. Ezen pálya megtalálá-

sához 3000-5000 iterációra volt szükség, teljesen véletlen mintavételezés esetén, tehát nem használjuk fel a celladekompozíció által adott pontokat.



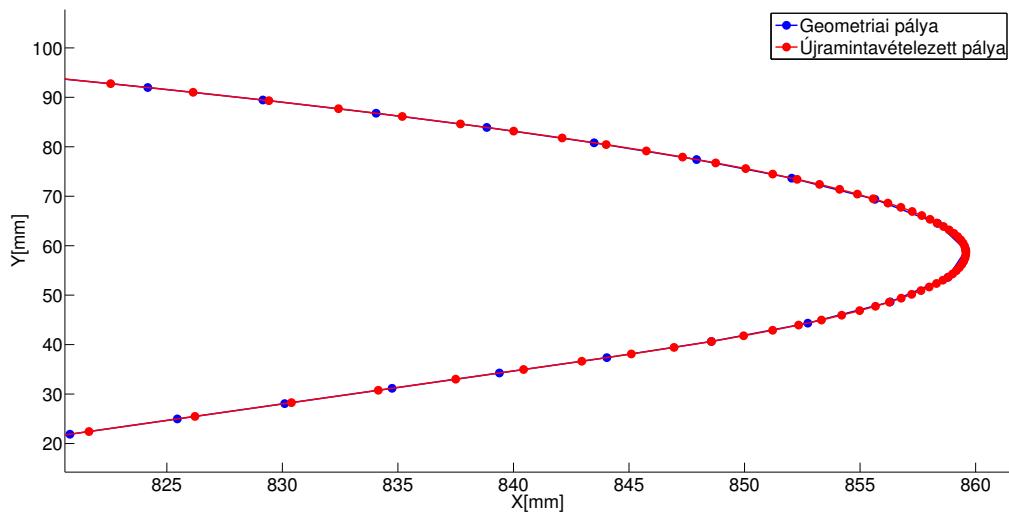
2.9. ábra. Az futás idők mérésekor használt környezet.

3. fejezet

Pálya időparaméterezése

A pályatervező által elkészített ütközésmentes pálya nem tartalmaz semmilyen idővel kapcsolatos információt. Ebben a fejezetben a pálya pontjaihoz sebesség értékeket rendelünk hozzá. Ezt a többletinformációt a pályakövető algoritmus használja fel, hogy mozgás során a robot kinematikai korlátai ne okozzanak problémát. Tehát az időparaméterezés elsősorban a robot korlátait használja fel, de arra is alkalmas, hogy meghatározzuk a pálya bejárásának idejét.

Az időparaméterezés két fő lépésből áll. Elsőként a kapott geometriai pályához sebesség értékeket rendelünk hozzá, majd ezután újramintavételezzük a pályát. Az újramintavételezés után a pálya időben egyenletes lesz, tehát az egymást követő pályapontok között azonos idő telik el. A mintavételezés idejét a pályakövető algoritmus mintavételi ideje határozza meg. A geometriai pályát általában távolságban egyenletesen mintavételezve kapjuk a pályatervező algoritmus kimeneteként, de ez nem kötelező feltétel az időparaméterezéshez.



3.1. ábra. A pálya időparaméterezése.

A szakirodalomban nem sok időparaméteréssel kapcsolatos munka található. Egy, az itt bemutatott hozzá hasonló megközelítést Christoph Sprunk munkájában találhatunk [10]. A legfontosabb elméleti eltérés, hogy Sprunk külön korlátozza a robot tangenciális és centri-

petális gyorsulását, míg mi a robot kerekeinek eredő gyorsulását korlátozzuk. Ez a megoldás a valóságot jobban közelíti, hiszen attól, hogy a gyorsulás két komponense a korlátok alatt marad, nem biztos, hogy az eredő gyorsulás sem haladja meg a korlátot.

Az időparaméterezés során nem használjuk ki az előző fejezetben bemutatott pályatervező által tervezett pálya speciális tulajdonságait, a célunk egy olyan algoritmus készítése, amely tetszőleges geometriai pályából képes sebesség információval ellátott, időben egyenletes mintavételű pályát készíteni. Emiatt nem építhetünk a pályatervező által használt geometriai elemekre és ezek speciális tulajdonságaira.

A geometriai elemek egyik legalapvetőbb tulajdonsága az lenne, hogy a görbületüket analitikusan ki tudnánk számolni (ami a konkrét elemek esetében ráadásul triviális). Általános esetben azonban nem tudjuk a pálya görbületét analitikusan meghatározni, így görbület becslést kell alkalmaznunk. A szakirodalomban sok cikket találhatunk görbület becslésről, főleg képfeldolgozással kapcsolatos témaiban, azonban dolgozatomnak nem ez a téma. Az algoritmus fejlesztésekor több becslőt is kipróbáltam, ezeket úgy teszteltem, hogy olyan pályát adtam meg nekik, amelynek a görbülete analitikusan is számolható, így össze tudtam hasonlítani az ideális megoldással a becslést. Ez alapján választottam egy eljárást [11]. Természetesen abban az esetben, ha a pályatervező rendelkezik már a pálya görbületével, az időparaméterező algoritmus azt fogja használni a becslés helyett.

3.1. Jelölések

Ebben a fejezetben a 3.1. táblázatban megadott jelöléseket fogjuk használni. Azokban az esetekben, ahol fontos megkülönböztetni a geometriai pályát és az (újra)mintavételezett pályát, ott a felső indexben található **g** betű a geometriai pályát jelöli, az **s** betű pedig a mintavételezett pályát. A pálya pontjait 1-től számozzuk.

$$\begin{aligned}
 \Delta t(k) &: \text{A } k \text{ és a } k + 1 \text{ pontok között eltelt idő} \\
 t(k) &: \text{A } k. \text{ pontban az addig eltelt idő} \\
 \Delta s(k) &: \text{A } k \text{ és a } k + 1 \text{ pontok között megtett távolság} \\
 s(k) &: \text{A } k. \text{ pontban az addig megtett távolság} \\
 v(k) &: \text{A } k. \text{ pontban a robot sebességének nagysága} \\
 \omega(k) &: \text{A } k. \text{ pontban a robot szögsebességének nagysága} \\
 a_t(k) &: \text{A } k. \text{ pontban a robot tangenciális gyorsulásának nagysága} \\
 r(k) &: \text{A } k. \text{ pontban a pálya görbületi sugara a robot középpontjához viszonyítva} \\
 c(k) &: \text{A } k. \text{ pontban a pálya görbületének nagysága, a görbületi sugár reciproka} \\
 N &: \text{A pálya pontjainak száma}
 \end{aligned} \tag{3.1}$$

Azokban az esetekben, amikor a robot kerekére vonatkozó mennyiségekről beszélünk, külön jelöljük, hogy bal (*l*) vagy jobb (*r*) kerékről van szó. Ezenkívül a kereknél megkülönböztetjük, hogy tangenciális (*a_t*), centripetális (*a_c*) vagy eredő (*a_e*) gyorsulásról beszé-

lünk.

Fontos megjegyezni, hogy a $\Delta s(k)$ távolságot úgy kell értelmezni, hogy a k . és $k+1$. pont között egy körív található, és az ezen mért távolság lesz $\Delta s(k)$. A körívet a $c(k)$ görbület határozza meg. Ha nem köríveket használnánk, hanem egyenessel kötnénk össze a pályapontokat, akkor a görbületnek szükségszerűen 0-nak kellene lennie.

3.2. Differenciális robotmodell

Mivel a fejezetben később még többször szükség lesz rá, az 1.2. egyenletet írjuk át úgy, hogy a kerekek sebességeit fejezzük ki akár a szögsebesség, akár a pálya adott görbülete alapján:

$$\begin{aligned} v_l(k) &= v(k) - \frac{W \cdot \omega(k)}{2} = v(k) \cdot p_l(k) \\ v_r(k) &= v(k) + \frac{W \cdot \omega(k)}{2} = v(k) \cdot p_r(k), \end{aligned} \quad (3.2)$$

ahol

$$\begin{aligned} p_l(k) &= 1 - \frac{W \cdot c(k)}{2} \\ p_r(k) &= 1 + \frac{W \cdot c(k)}{2}, \end{aligned} \quad (3.3)$$

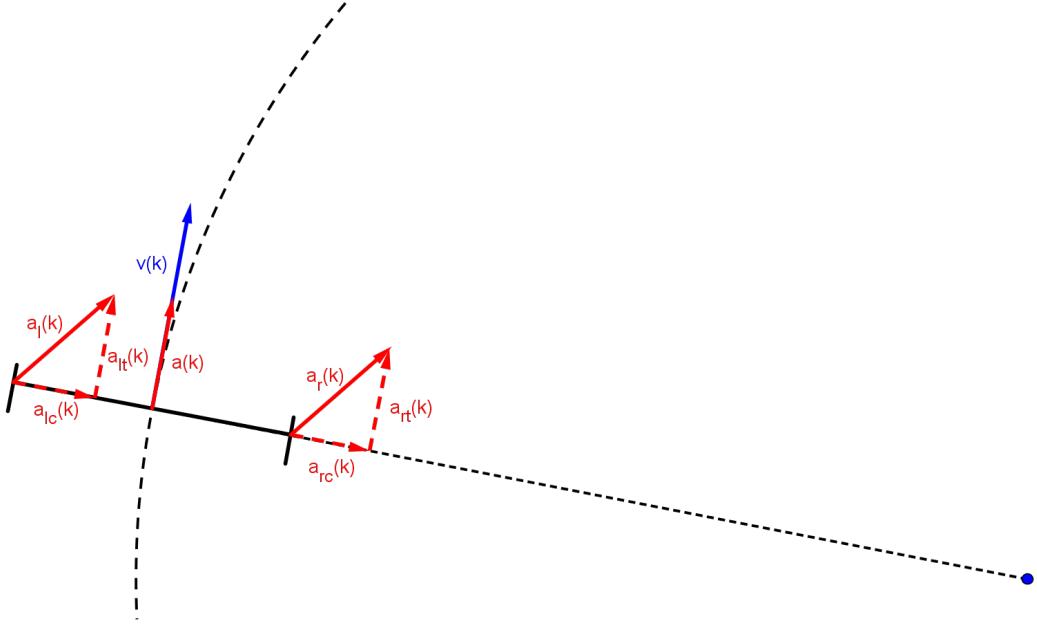
és felhasználtuk, hogy $v(k) \cdot c(k) = \omega(k)$.

3.3. Korlátozások

A robot mozgását általános esetben a 3.2. ábra mutatja be. Az időparaméterezés során figyelembe vesszük a robot pályamenti sebességét és szögsebességét, valamint a robot kerekeinek tangenciális és eredő gyorsulását. Adott robot esetében ezekre a mennyiségekre határozunk meg korlátozásokat:

$$\begin{aligned} v^{max} &: \text{A robot pályamenti sebesség korlátja} \\ \omega^{max} &: \text{A robot szögsebesség korlátja} \\ a_{lt}^{max} &: \text{A robot bal kerekének tangenciális gyorsulás korlátja} \\ a_{rt}^{max} &: \text{A robot jobb kerekének tangenciális gyorsulás korlátja} \\ a_l^{max} &: \text{A robot bal kerekének eredő gyorsulás korlátja} \\ a_r^{max} &: \text{A robot jobb kerekének eredő gyorsulás korlátja} \end{aligned} \quad (3.4)$$

Mivel a robot kerekeinek tangenciális gyorsulásából már adódik a robot tangenciális gyorsulása is, így a robot gyorsulását nem szükséges külön korlátozni.



3.2. ábra. Differenciális hajtású robot mozgása köríven.

$$a_t^{max} = \frac{a_{lt}^{max} + a_{rt}^{max}}{2} \quad (3.5)$$

Ugyanez a helyzet a kerekek sebességkorlátjával, ami meghatározható a robot sebesség és szögsebesség korlátaiból.

$$v_l^{max} = v^{max} - \frac{W \cdot \omega^{max}}{2} \quad (3.6)$$

$$v_r^{max} = v^{max} + \frac{W \cdot \omega^{max}}{2} \quad (3.7)$$

A kerekek maximális eredő gyorsulását a maximális tapadási súrlódási együttható, $\mu_{tap_{max}}$ határozza meg, amelynél a robot kerekei még nem csúsznak meg. A maximális gyorsulás és a tapadási együttható között a következő egyszerű összefüggés áll fent:

$$a_{max} = \mu_{tap_{max}} \cdot g, \quad (3.8)$$

ahol g a nehézségi gyorsulás

Írjuk fel a 3.2. ábra alapján a robot kerekeinek gyorsulását:

$$a(k) = \sqrt{a_c(k)^2 + a_t(k)^2} \leq g \cdot \mu_{tap_{max}}, \quad (3.9)$$

ahol $a_c(k)$ a kerék centripetális gyorsulása, $a_t(k)$ a kerék tangenciális gyorsulása

A 3.9. egyenletben azzal a feltevéssel élünk, hogy a robot kerekei és a talaj között a tapadási súrlódási együttható állandó és nem függ az erő irányától. Az általunk használt differenciális robotnál ez a közelítés megengedhető, mivel a gumikerekek homogénnek tekinthetők. Ha barázdákat tartalmaznak, akkor már nagyobb eltérést okozna ez a közelítés.

Fontos megjegyezni, hogy a kerékgyorsulás korlátokat lassulásnál is alkalmazzuk, tehát a kerék gyorsulásának abszolút értékét korlátozzák ezek a korlátozások. Így azt tesszük fel, hogy a kerekek viselkedése gyorsulás és lassulás esetében megegyezik. A robot sebességénél viszont nem engedünk előjelváltást: úgy vesszük, hogy a robot végig egy irányba halad. A tervező viszont megadhat olyan pályát, ahol tolatnia kell a robotnak, vagy egyhelyben megfordulnia, de ezt a pályatervező algoritmus kezeli.

3.4. Geometriai sebességprofil

Az időparaméterezés első lépéseként a geometriai pályapontokhoz rendelünk a korlátoknak megfelelő sebességeket, és a későbbiekben ezt a sebességprofilt használjuk fel a pálya újramintavételezéséhez.

A pályamenti sebességeket úgy határozzuk meg, hogy a robot pályamenti gyorsulása a lehető legnagyobb legyen. A 3.5. egyenlet alapján ezt megtehetjük úgy, hogy a robot kerekeinek tangenciális gyorsulását maximalizáljuk. Több hatás miatt azonban nem tudjuk a kerekek gyorsulását folyamatosan növelni.

Egyrészt a robot sebesség és szögsebesség korlátját nem sérthetjük meg. Ebből a két korlátból a pálya minden pontjára kiszámolhatunk egy maximális sebességet, függetlenül az előző pályapont sebességétől:

$$v^{max}(k) = \min \left(v^{max}, \frac{\omega^{max}}{c(k)} \right) \quad (3.10)$$

Valamint a kerekek centripetális gyorsulása nem haladhatja meg az előírt eredő gyorsuláskorlátot, különben a robot kereke megcsúszna. A pálya adott k . pontjában a kerekek centripetális gyorsulását a következőképpen számolhatjuk ki:

$$\begin{aligned} a_{lc}(k) &= (v(k) \cdot p_l(k))^2 \cdot c(k) \\ a_{rc}(k) &= (v(k) \cdot p_r(k))^2 \cdot c(k) \end{aligned} \quad (3.11)$$

Fontos megjegyezni, hogy mivel a robot gyorsulását határozzuk meg a k . pontban, így a $v(k)$ már rendelkezésünkre áll a $k - 1$. pontban számított gyorsulásból. Amennyiben a kiszámolt centripetális gyorsulások már önmagukban is meghaladják az előírt eredő gyorsuláskorlátot, úgy $v(k)$ értékét addig kell csökkenteni, hogy a centripetális gyorsulás az eredő gyorsuláskorlátot már ne haladja meg. Erről a folyamatról a későbbiekben még szót ejtünk.

Ezután a kerekek tangenciális gyorsulását a 3.12. egyenlet alapján határozhatjuk meg.

$$\begin{aligned} a_{lt}(k) &= \min \left(\sqrt{(a_l^{max})^2 - a_{lc}(k)^2}, a_{lt}^{max} \right) \\ a_{rt}(k) &= \min \left(\sqrt{(a_r^{max})^2 - a_{rc}(k)^2}, a_{rt}^{max} \right) \end{aligned} \quad (3.12)$$

Eddig a két kerék gyorsulását teljesen függetlenül tárgyaltuk, azonban minden két gyorsulást nem választhatjuk meg szabadon, mert a pálya görbülete meghatározza a köztük lévő arányt. Ezt a következőképpen láthatjuk be:

$$a_{lt}(k) = \beta(k) \cdot (r(k) - \frac{W}{2}) \quad (3.13)$$

$$a_{rt}(k) = \beta(k) \cdot (r(k) + \frac{W}{2}) \quad (3.14)$$

$$\frac{a_{lt}(k)}{a_{rt}(k)} = \frac{r(k) - \frac{W}{2}}{r(k) + \frac{W}{2}} = \frac{p_l(k)}{p_r(k)}, \quad (3.15)$$

ahol $\beta(k)$ a robot szögggyorsulása (a 3.2. egyenlet alapján következik, hogy a sebességek aránya is ugyanez lesz).

A 3.15. és a 3.12. egyenletek alapján 2-2 lehetséges kerék gyorsulást tudunk számolni. Ezek közül azt a gyorsulás párt fogjuk választani, amelyiknek egyik eleme sem sérti a 3.12. egyenletek által meghatározott korlátokat.

Miután kiszámoltuk, hogy az adott pályapontnál mekkora legyen a robot kerekeinek tangenciális gyorsulása, már könnyedén számolható a robot gyorsulása és sebessége:

$$a_t(k) = \frac{a_{lt}(k) + a_{rt}(k)}{2} \quad (3.16)$$

$$v(k+1) = \min \left(v^{max}(k+1), \sqrt{v(k)^2 + 2 \cdot a_t(k) \cdot \Delta s_c(k)} \right) \quad (3.17)$$

3.4.1. Profil visszaterjesztés

Két esetben előfordulhat, hogy az előző pályaponthoz meghatározott sebességértéket módosítani kell. Egyszerűen, ha a centripetalis gyorsulás önmagában meghaladja a megengedhető maximális gyorsulást, akkor az előző pályaponthoz tartozó sebességet minden képp csökkenjen kell. Másrészt a 3.17. egyenlet esetében előfordulhat, hogy a robot gyorsulás korlátját megséríti, és így módosítani kell az előző ponthoz tartozó sebességet. Ez például a pálya végpontjában fordulhat elő, ahol előírjuk, hogy a robot álljon meg, tehát $v^{max}(N) = 0$. Ha nem terjesztenénk vissza a profilt, akkor az utolsó pontnál lévő félezés meghaladhatja az előírt korlátot, hiszen az előző pontokban nem tudtuk, hogy később meg kell állni a robotnak.

Minden esetben ugyanazt az eljárást alkalmazhatjuk a visszaterjesztéshez. Azért beszélünk visszaterjesztésről, mivel addig kell visszafelé haladni a pályán, amíg minden korlátot betartunk.

Kezdetnek kiszámoljuk, hogy a megváltozott sebesség következtében hogyan alakulnak a kerekek tangenciális gyorsulásai. A 3.18. egyenletben felhasználjuk a 3.2. egyenlet összefüggését a robot és kerék sebesség kapcsolatára.

$$\begin{aligned} a_{lt}(k) &= \frac{v_l(k+1)^2 - v_l(k)^2}{2 \cdot \Delta s_l(k)} = \frac{v(k+1)^2 - v(k)^2}{2 \cdot \Delta s_l(k)} \cdot p_l(k)^2 \\ a_{rt}(k) &= \frac{v_r(k+1)^2 - v_r(k)^2}{2 \cdot \Delta s_l(k)} = \frac{v(k+1)^2 - v(k)^2}{2 \cdot \Delta s_r(k)} \cdot p_r(k)^2 \end{aligned} \quad (3.18)$$

Amennyiben a kapott tangenciális gyorsulások megsértik a tangenciális vagy eredő gyorsulásra vonatkozó korlátokat, kiszámoljuk, hogy mekkora robotsebesség esetében teljesülnének a korlátok. Ezt minden két kerék esetén megtesszük, és a szigorúbb sebesség korlátot fogjuk választani, mint robotsebesség. Ezt az eljárást mindaddig megtesszük visszafelé a pályán, amíg azt nem kapjuk, hogy egyik kerék sem sérti meg a korlátokat.

Most vizsgáljuk meg, hogy ha a kerékgyorsulás egy adott korlátot megsért, akkor hogyan kapjuk meg belőle azt a robotsebességet, amely esetében még nem sértjük meg a korlátot. Először tekintsük a tangenciális gyorsulásra vonatkozó korlátot. A 3.18. egyenletet fejezzük ki $v(k)$ -ra minden két kerék esetén:

$$\begin{aligned} v_l^t(k) &= \sqrt{v(k+1)^2 + \frac{2 \cdot a_{lt}^{max} \Delta s_l(k)}{p_l(k)^2}} \\ v_r^t(k) &= \sqrt{v(k+1)^2 + \frac{2 \cdot a_{rt}^{max} \Delta s_r(k)}{p_r(k)^2}}, \end{aligned} \quad (3.19)$$

ahol a $v_l^t(k)$, $v_r^t(k)$ jelölések arra utalnak, hogy a sebességek a tangenciális korlátból adódnak a bal és jobb kerék esetén.

A 3.19. egyenlet alapján minden esetben meg tudjuk határozni a keresett robotsebességet, hiszen a gyök alatt található összeg minden két tagja biztosan nem negatív.

Az eredő gyorsulásra vonatkozó korlát esetén pedig a 3.20. összefüggést használhatjuk. Ehhez felhasználjuk a 3.11 egyenletet (az egyszerűség kedvéért most nem vezetjük le külön a két kerék esetén a számítást, és w -vel jelöljük a kerékre vonatkozó mennyiségeket):

$$\begin{aligned} a_{wt}(k) &= \frac{v(k+1)^2 - v_w^e(k)^2}{2 \cdot \Delta s_w(k)} \cdot p_w(k)^2 = \sqrt{(a^{max})^2 - (a_{wc}^{max})^2} \\ &= \sqrt{(a^{max})^2 - (v_w^e(k) \cdot p_w(k) \cdot c(k))^2} \end{aligned} \quad (3.20)$$

ahol a $v_w^e(k)$ jelölés arra utal, hogy a sebesség az eredő gyorsulásra vonatkozó korlátból adódik.

A keresett robotsebesség érdekében, fejezzük ki a 3.20. egyenletet $v_w^e(k)$ -re. Ekkor egy negyedfokú egyenletet kapunk, ami a következőképpen épül fel:

$$d(k) = \frac{p_w(k)^4}{4 \cdot \Delta s_w(k)^2} + c(k)^2 \cdot p_w(k)^2 \quad (3.21)$$

$$e(k) = -\frac{2 \cdot v(k+1)^2 \cdot p_w(k)^4}{4 \cdot \Delta s_w(k)^2}$$

$$f(k) = \frac{v(k+1)^4 \cdot p_w(k)^4}{4 \cdot \Delta s_w(k)^2} - a_{max}^2$$

$$0 = {v_w}^e(k)^4 \cdot d(k) + {v_w}^e(k)^2 \cdot e(k) + f(k) \quad (3.22)$$

A 3.22. negyedfokú egyenlet visszavezethető másodfokú egyenletre, ha bevezetjük az $x(k) = {v_w}^e(k)^2$ változót:

$$0 = x(k)^2 \cdot d(k) + x(k) \cdot e(k) + f(k) \quad (3.23)$$

A 3.23. egyenlet valós, pozitív megoldásait keressük, hiszen ekkor lesz megoldása az eredeti 3.22. egyenletnek. Felmerülhet a kérdés, hogy mi garantálja, hogy minden lesz ilyen megoldás.

A gyökök összegére felírt Viète-formula segítségével belátható, hogy legalább egy pozitív megoldása van az egyenletnek:

$$x_1(k) + x_2(k) = \frac{-e(k)}{d(k)} \geq 0, \quad (3.24)$$

ahol $x_1(k)$ és $x_2(k)$ a 3.22. egyenlet gyökei, és felhasználtuk, hogy $e(k) \leq 0$ és $d(k) \geq 0$.

A 3.23. egyenlet diszkriminánsának felírásával pedig beláthatjuk, hogy az egyenlet megoldásai minden esetben valósak:

$$D = e(k)^2 - 4 \cdot d(k) \cdot f(k) \geq 0 \quad (3.25)$$

Tehát amennyiben a diszkrimináns nem negatív, akkor az egyenletnek csak valós megoldása van. Az $d(k)$, $e(k)$ és $f(k)$ értékét behelyettesítve, a következő összefüggést kell belátnunk:

$$a_{max}^2 \cdot \frac{p_w(k)^4}{\Delta s_w(k)^2} - \frac{v(k+1)^4 \cdot p_w(k)^6 \cdot c(k)^2}{\Delta s_w(k)^2} + 4 \cdot p_w(k)^2 \cdot c(k)^2 \cdot a_{max}^2 \geq 0 \quad (3.26)$$

Az egyenlőtlenség minden oldalát szorozzuk be $\frac{\Delta s_w(k)^2}{p_w(k)^4}$ -vel, és használjuk fel, hogy $v_w(k+1) = v(k+1) \cdot p_w(k)$:

$$a_{max}^2 - v_w(k+1)^4 \cdot \frac{c(k)^2}{p_w(k)^2} + 4 \cdot p_w(k)^2 \cdot \Delta s_w(k)^2 \cdot a_{max}^2 \cdot \frac{c(k)^2}{p_w(k)^2} \geq 0 \quad (3.27)$$

Mivel $\frac{c(k)}{p_w(k)} = c_w(k)$ és $a_{wc}(k) = v_w(k)^2 \cdot c_w(k)$, ezért a 3.27. egyenlőtlenséget átírhatjuk a következők szerint:

$$a_{max}^2 - a_{wc}(k+1)^2 + 4 \cdot p_w(k)^2 \cdot \Delta s_w(k)^2 \cdot a_{max}^2 \cdot \frac{c(k)^2}{p_w(k)^2} \geq 0 \quad (3.28)$$

A 3.28. egyenlőtlenség első két tagjára a következő feltétel teljesül a gyorsulás korlátok alapján:

$$\begin{aligned} a_{wt}(k+1)^2 + a_{wc}(k+1)^2 &\leq a_{max}^2 \\ 0 \leq a_{wt}(k+1)^2 &\leq a_{max}^2 - a_{wc}(k+1)^2 \end{aligned} \quad (3.29)$$

Tehát a 3.29 feltétel alapján a 3.28. egyenlőtlenség első két tagja biztosan nem negatív, és mivel az egyenlőtlenség többi tagja is biztosan nem negatív, így biztosak lehetünk benne, hogy a 3.22. egyenletnek lesz valós megoldása.

Miután meghatároztuk $v^e(k)$ és $v^t(k)$ értékeit mindenkorrekt kerékre (összesen 4 érték), $v(k)$ értéke ezek közül a legkisebb lesz, hiszen így biztosíthatjuk, hogy a robot egyik kereke sem fogja megsérteni a két gyorsulás korlátot.

A visszaterjesztés során a sebesség és szögsebesség korlátokkal nem kell foglalkoznunk, hiszen mikor visszaterjesztés közben csökkentjük a sebességet, ezektől a korlátoktól távolodunk.

3.5. Újramintavételezés

Miután elkészítettük a geometriai pályához tartozó sebességprofilt, létrehozzuk a véleges pályát, amit majd a pályakövető algoritmus bemenetként megkap. Ez a véleges pálya már időben egyenletesen lesz mintavételezve (mintavételezett pálya).

Először számoljuk ki az eltelt időt a geometriai pálya mentén, amelynek alapja, hogy két pályapont között a robot állandó gyorsulással halad.

$$\Delta t^g(k) = \frac{2\Delta s^g(k)}{v^g(k) + v^g(k+1)} \quad (3.30)$$

$$t^g(k+1) = t^g(k) + \Delta t^g(k) \quad (3.31)$$

A következő lépésben meghatározzuk, hogy az újramintavételezett pályánk hány pontból álljon. Ezt könnyedén megtehetjük, hiszen adott számunkra a kívánt mintavételi idő(t_s). Így a következő képlet adódik a mintavételezett pálya pontjainak számára:

$$N^s = \lceil t^g(N^g)/t_s \rceil + 1 \quad (3.32)$$

A pontok számába beleérjük a kezdő és végpontot is. A 3.32. egyenletből következik, hogy amennyiben $t(N^g)$ és t_s nem egymás többszörösei, a mintavételezett pálya utolsó pontjához olyan időpont tartozik, amely nagyobb mint $t(N^g)$. A pálya végpontját még a későbbiekben tárgyaljuk, akkor visszatérünk erre az eltérésre is.

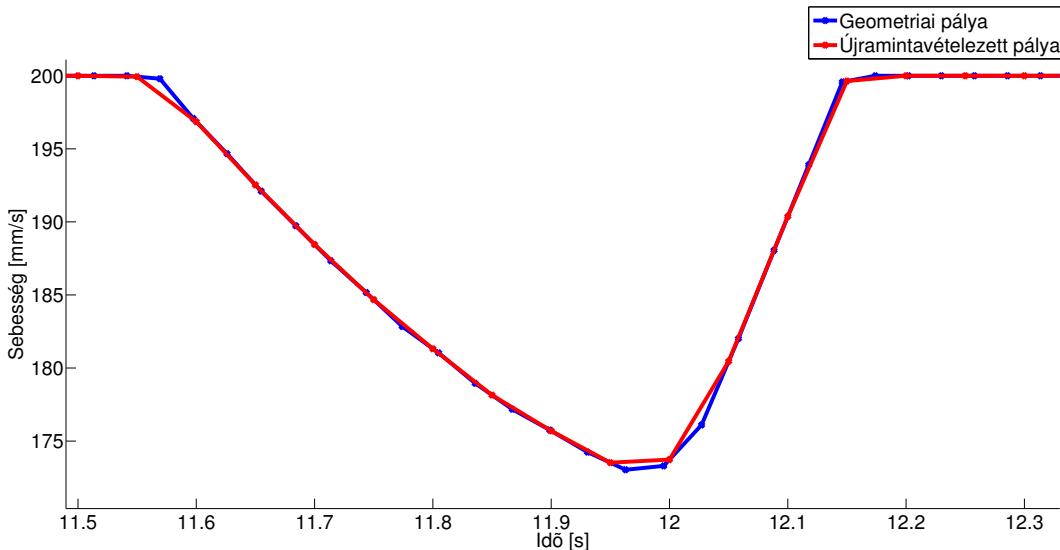
Most pedig meghatározzuk a mintavételezett pálya pontjaiban a sebességet. Ezt a geometriai pálya alapján tesszük, figyelembe véve, hogy a mintavételezett pálya esetén is két pont között állandó gyorsulást feltételezünk. A számítás egy egyszerű lineáris interpolációt valósít meg:

$$v^s(k) = v^g(j) + v^g(j+1) \cdot it(k) \quad (3.33)$$

$$it(k) = \frac{t^s(k) - t^g(j)}{t^g(j+1) - t^g(j)}, \quad (3.34)$$

ahol j jelöli a legkisebb indexet amelyre teljesül, hogy $t^s(k) < t^g(j)$

A lineáris interpoláció miatt teljesül az a feltétel, hogy két pont között állandó gyorsulással mozogjon a robot.



3.3. ábra. A geometriai és mintavételezett sebességprofil egy részlete.

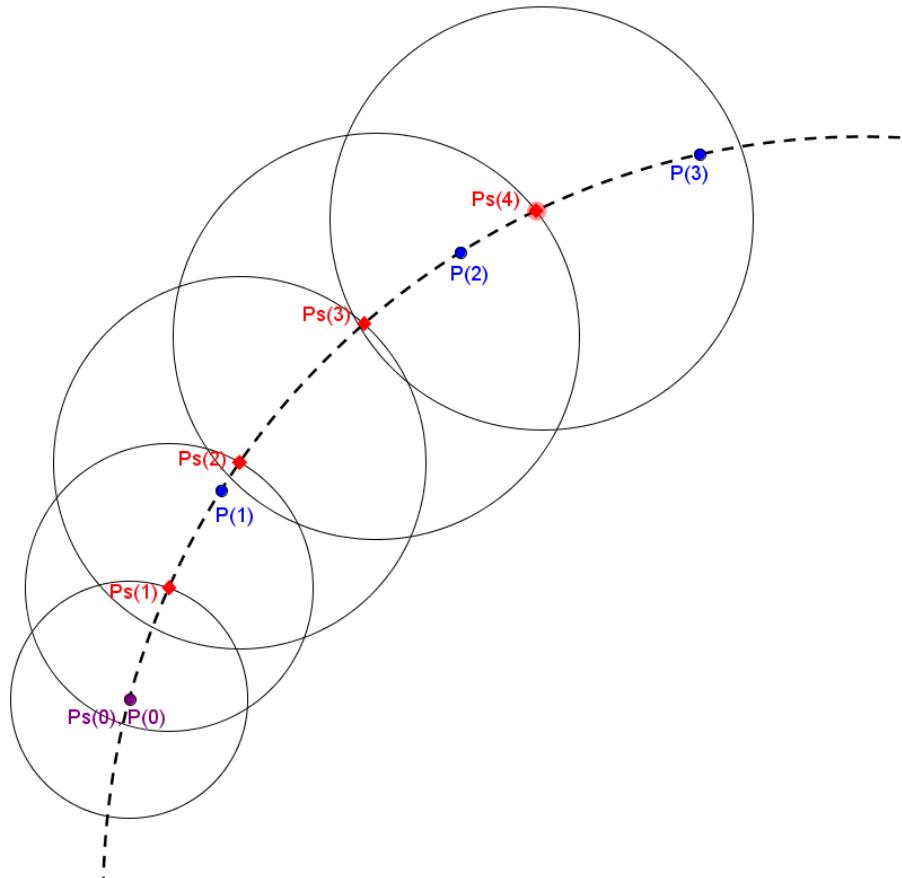
A kiszámított sebességprofil alapján könnyedén adódik a megtett út is:

$$\Delta s^s(k) = \frac{v^s(k) + v^s(k+1)}{2} \cdot t_s \quad (3.35)$$

$$s^{s+1}(k) = s^s(k) + \Delta s^s(k) \quad (3.36)$$

Így már rendelkezésünkre áll a robot kívánt sebessége, a megtett út, valamint az idő a mintavételezett pálya összes pontjában. Már csupán a pálya pontjainak koordinátáit kell ezek alapján meghatároznunk.

Mivel ismerjük a pályapontok közötti távolságot ($\Delta s^s(k)$), iteratív eljárással az előző pályapont koordinátái alapján az aktuális pontról tudjuk, hogy egy körön helyezkedik el. További feltételünk, hogy a pont az eredeti, geometriai pályán rajta legyen. Ha vesszük a geometriai pálya pontjai közötti görbületből adódó köríveket, akkor az ívek és a kör metszéspontjai közül kell kiválasztanunk a keresett pontot. A kiválasztás egyszerű, ha megjegyezzük, hogy az előző pontnál melyik szakasz alapján találtuk meg a pontot, így csak attól a szakasztól kezdve kell keresni a metszéspontokat. Az algoritmus menete látható a 3.4. ábrán. minden vizsgált szakasznál arra kell figyelni, hogy a metszéspont a szakasz határpontjai között helyezkedjen el. Az első szakasz vizsgálatánál még az is fontos, hogy az előző pont előtti metszéspontot ne vegyük figyelembe. Az ábrán a $Ps(1)$ pontban ezért nem választhatjuk a másik metszéspontot. A legelső mintavételezett pontot a geometriai pálya első pontjába helyezzük el.



3.4. ábra. A mintavételezett pontok meghatározása. $P(x)$ a geometriai pálya pontjait jelöli, $Ps(y)$ pedig a keletkező mintavételezett pályát.

3.5.1. Mintavételezett pálya végpontja

Az lenne az optimális eset, ha a mintavételezett pálya utolsó pontja egybeesne az eredeti pálya végpontjával, ahogyan a kezdőpontjaik ténylegesen egybeesnek. Alapvetően úgy hoztuk létre a mintavételezett pályát, hogy az a geometriai pálya sebességprofiljának megfeleljen, ez viszont nem garantálja az előző feltétel teljesülését.

Három hatás azt eredményezi, hogy nem fog teljesülni ez a feltétel a pálya utolsó pontjára:

1. Ahogy már említettük korábban, nem biztos, hogy a két pályát ugyanannyi idő alatt járja be a robot. Ez maximum t_s időkülönbséget okozhat, és minden esetben távolabbi végpontot eredményez, mint az eredeti végpont.
2. A mintavételezett pálya sebességprofiljának elkészítésekor nem tökéletesen követi az eredeti sebességet a robot a mintavételezésből adódóan. Ez látszik a 3.3. ábrán is. A hiba megegyezik a két görbe alatti terület közötti különbséggel, ami okozhat távolabbi és közelebbi végpontot is.
3. A harmadik hiba a koordináták meghatározásánál keletkezik. Ez a hatás is mindig távolabbi végpontot okoz.

A legtöbb esetben célszerű, ha a végpontok egybeesnek, így ezt a mintavételezett pálya meghatározásánál biztosítanunk kell. Ha egyszerűen az utolsó pályapontot az eredeti pálya végpontjába tesszük, nem biztos, hogy betartjuk a robot gyorsulás korlátait, így más módszerhez kell folyamodnunk.

Az általunk használt algoritmus lényege, hogy a sebességprofilnak egy részét egy adott sebességgel eltoljuk úgy, hogy a két pálya végpontja pontosan egybeessen. Az eltolás mértékét (Δv_{corr}) a következő képlettel kapjuk meg:

$$\Delta v_{corr} = \frac{\Delta s_{corr}}{t_s \cdot n}, \quad (3.37)$$

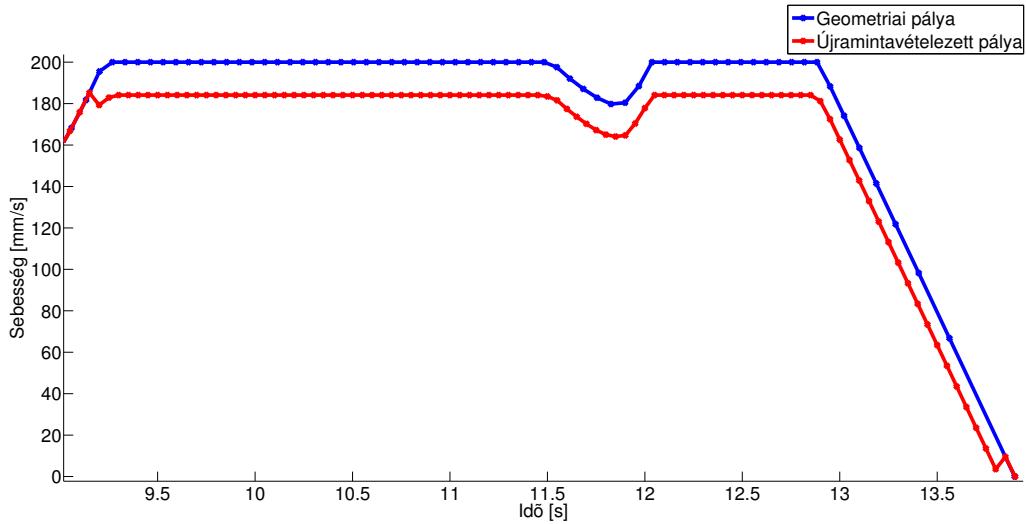
ahol Δs_{corr} a mintavételezett és a geometriai pálya végpontjai közötti távolság előjelesen. Ha a mintavételezett pálya utolsó pontja van távolabb, akkor negatív a távolság, különben pozitív. n pedig azoknak a sebességpontoknak a száma, amiket eltolunk.

A 3.37. egyenlet egyszerűen belátható, ha felírjuk az eltolásból adódó területkülönbséget. A Δs_{corr} útkülönbséget azért kell előjelesen megadnunk, hogy minden esetben használható legyen az algoritmus, akkor is, ha a mintavételezett pálya végpontja van távolabb és akkor is ha a geometriai pályáé.

A továbbiakban meghatározzuk azokat a sebességpontokat, amelyeket Δv_{corr} sebességgel eltolunk. Mivel a megváltozott sebességponthoz tartozó koordinátákat újra ki kell számolnunk, így minél kevesebb pontot szeretnénk eltolni a sebességprofilon. Viszont a sebesség és gyorsulás korlátokat be kell tartanunk, így nem tolhatunk el tetszőlegesen kevés pontot.

Vizsgáljuk külön a két alapesetet Δs_{corr} előjele alapján. Kezdjük azzal az esettel amikor Δs_{corr} negatív, tehát a mintavételezett pálya végpontja van távolabb (3.5. ábra). Ekkor a

módosítandó szakasz kezdő pontjához tartozó gyorsulásnak pozitívnak kell lennie, hiszen mi csökkenteni fogjuk a soron következő pont sebességét, és ha a gyorsulás pozitív vagy nulla, akkor a módosítás hatására csökkeni fog a robot gyorsulása a szakasz kezdőpontjában. Ha a gyorsulás negatív lenne a kezdőpontban, akkor könnyedén előfordulhat olyan eset, hogy a sebességsökkenés után megszegjük a gyorsulás korlátot.

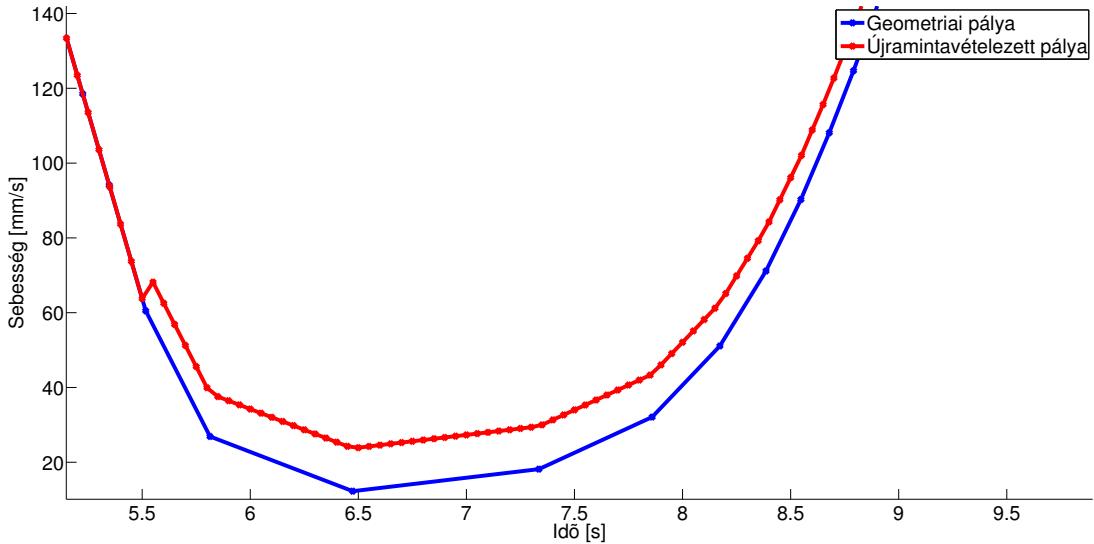


3.5. ábra. A módosított mintavételezett sebességprofil ha Δs_{corr} negatív.

A szakasz végpontjánál pedig negatív gyorsulás szükséges, hiszen a következő pont gyorsulása meg fog nőni a módosítás hatására, és ha pozitív lenne a gyorsulás, a gyorsulásra vonatkozó korlátunkat könnyedén megszegnénk.

Tehát a legegyszerűbb esetben a módosítandó szakasz kezdőpontja a pálya végén található lassító szakasz eleje, mielőtt lassítani kezd a robot és a végpontja a pálya utolsó előtti pontja. Ennek a szakasznak a pontjait fogjuk a 3.37. egyenletből adódó Δv_{corr} sebességgel csökkenteni, és így a robot pontosan a geometriai pálya végpontjában áll meg.

A másik eset, amikor Δs_{corr} pozitív, tehát a mintavételezett pálya végpontja messzebb van a geometriai pálya végpontjához képest. Ekkor, mivel meg fogjuk növelni a szakasz sebességét, pont fordítva kell kiválasztanunk a módosítandó szakaszt, a kezdőpontjánál negatív gyorsulás szükséges, a végpontjánál pedig pozitív. Így kerülhető el leginkább a gyorsulás korlát megszegése. Itt pedig egy megfelelő szakasz a pályán található utolsó gyorsító rész.



3.6. ábra. A módosított mintavételezett sebességprofil ha Δs_{corr} pozitív.

Abban az esetben, ha valamiért az előbb leírt triviális szakaszok mégsem jók, másik szakaszt kell választanunk. Első lépésként válasszunk ki egy megfelelő végpontot a keresendő szakaszhoz. Ha Δs_{corr} negatív, akkor megfelelő választás a pálya utolsó előtti pontja, ha pozitív, akkor pedig a pálya utolsó olyan pontja, ahol a gyorsulás pozitív. Ezután keresünk ehhez a kiválasztott végponthoz egy kezdőpontot, de most már vegyük figyelembe a robot korlátozásait, és természetesen azt, hogy az útkülönbség az előírt Δs_{corr} legyen. Miután megkaptuk a kezdőpontot is, akkor még ellenőriznünk kell, hogy a végpontnál a robot korlátozásait nem sértjük-e meg. Ezt az első lépésben nem tudtuk megtenni, mivel nem ismertük a végpontot, így Δv_{corr} értékét sem. Ha a végpont megséríti a korlátokat, új végpontot kell keresnünk és ahhoz új kezdőpontot. Ezt addig kell folytatnunk, amíg a robot korlátozásait be nem tartjuk.

Miután a módosított sebességprofil elkészült, a szakasz elejétől kezdve újra kell számolnunk a mintavételezett pálya koordinátáit. Ez teljesen ugyanúgy történik, ahogyan már egyszer megkaptuk a mintavételezett pályát. Azért volt fontos, hogy a lehető legkevesebb sebességpontot toljuk el, hogy a koordináták újraszámlálását is kevesebb pontnál kelljen megtenni.

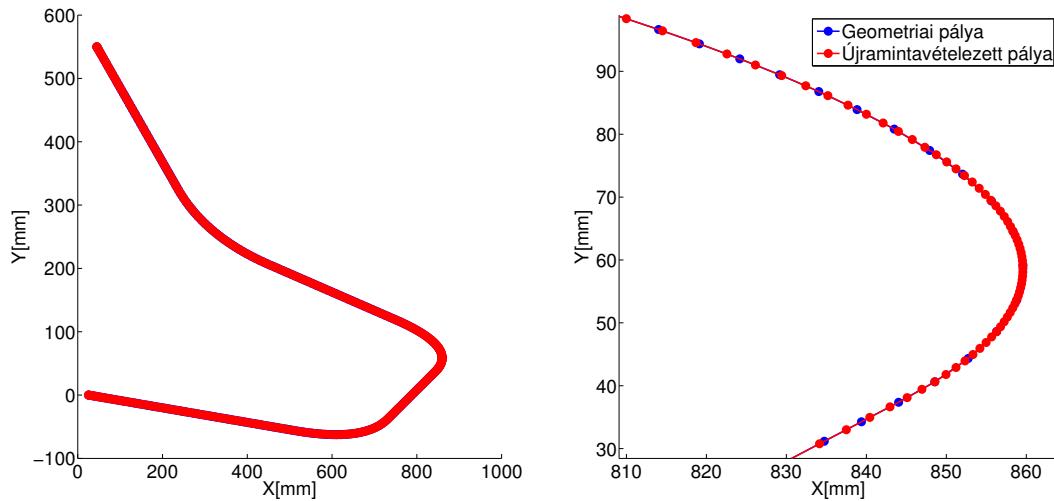
Habár a fenti iteratív eljárás hosszadalmasnak tűnik, vegyük figyelembe, hogy általában kis távolságot kell kompenzálnunk, amihez kis sebessékgülönbség tartozik. Ebből adódóan nagy valószínűséggel a triviális szakasz is megfelelő lesz számunkra.

Szintén fontos megjegyezni, hogy mivel a tárgyalt három hatás elsősorban negatív Δs_{corr} -t eredményezz, így a gyakorlatban ez az eset fordul elő. A gyakorlatot tekintve még megemlíteni kell, hogy a Δs_{corr} nagyságrendje igencsak csekély a pálya teljes hosszához képest, nehezen elképzelhető akárcsak 1%-ot meghaladó arány a teljes pálya hosszához képest. A jobb ábrázolás érdekében ezért a 3.5. és a 3.6. ábrákon megnöveltük az eltérést, a valóságban ekkora eltérés nem fordul elő.

3.6. Eredmények

A következőkben bemutatjuk az időparaméterezés eredményeit egy konkrét pályán. A pályát nem az RTR pályatervező szolgáltatja ebben a példában, hanem egy tetszőleges pályát vizsgálunk.

A 3.7. ábrán látható a geometriai és az újramintavételezett pálya, a jobb oldali ábrán kiemelve a pálya egy részletét. A kiemelt részen látható, hogy a kanyarban, ahol nagy a pálya görbülete, a mintavételezett pálya pontjai besűrűsödnek, mivel itt a robotnak lassítani kell.

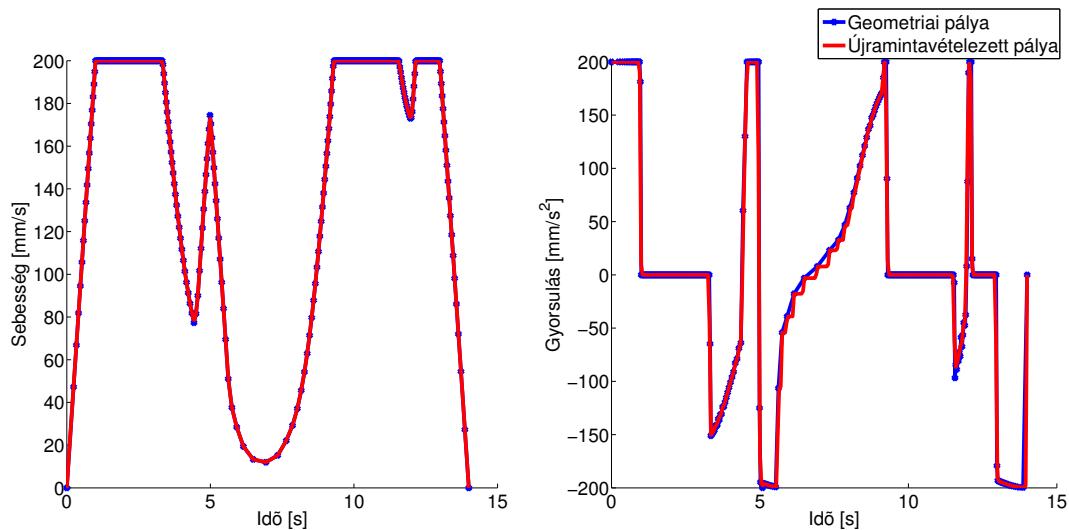


3.7. ábra. A geometriai és mintavételezett pálya.

A 3.8. ábrán látható a geometriai és mintavételezett pályához tartozó sebesség- és gyorsulásprofil. A sebesség itt robotsebességet jelent, a gyorsulás pedig a robot tangenciális gyorsulását.

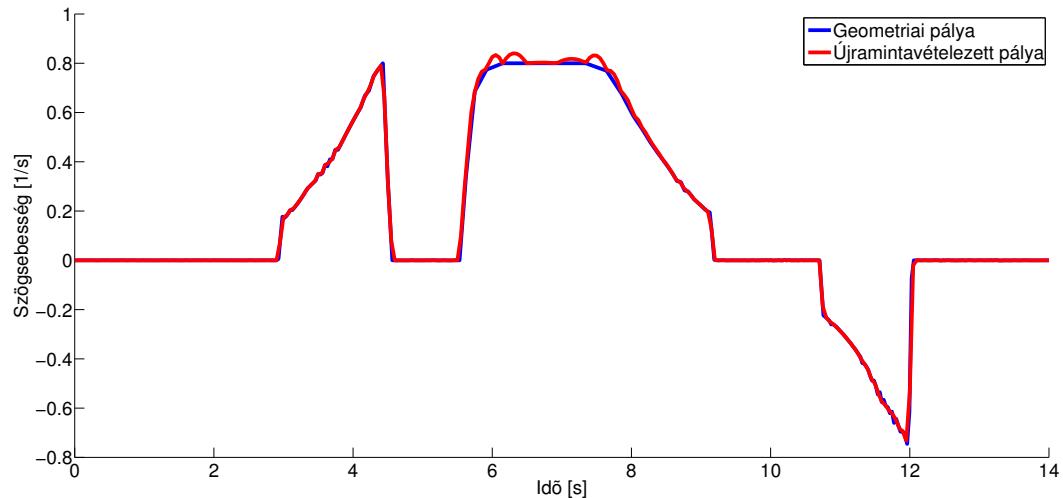
A sebességprofil esetében elmondható, hogy a mintavételezett pálya sebességprofilja jól követi az eredeti, geometriai pályához tartozó sebességeket. Ez nem meglepő, hiszen a lineáris interpolációt a sebességprofilon végeztük el.

A gyorsulásprofil esetében már tapasztalunk némi eltérést a profil közepénél. Ez egyértelműen a mintavételezésből adódik, hiszen ahogy a sebességprofilnál látható, ezen a szakaszon a geometriai pálya felbontása kicsi a sebesség-változás leírására.



3.8. ábra. A két pályához tartozó sebesség- és gyorsulásprofil 200 mm/s sebességhatár és 200 mm/s^2 gyorsuláshatár esetén.

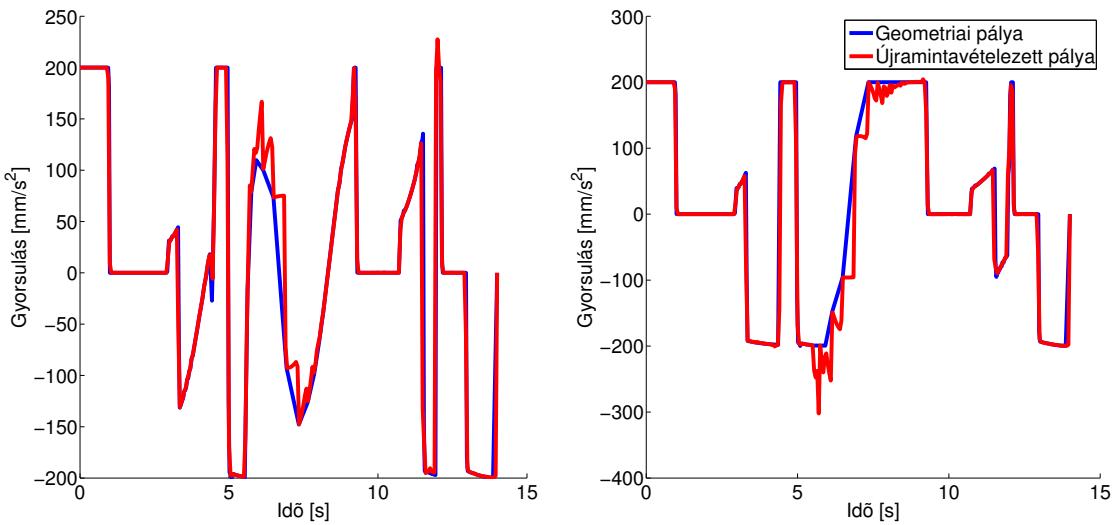
A következő ábra a robot szögsebességszabályzóját mutatja be. Itt is elmondható, hogy a pálya középső részén a mintavételezés miatt eltérés van a két pálya szögsebességében. A mintavételezett szögsebességszabályzó emiatt megséríti az előírt 0.8 1/s -os szögsebességhatárt. (Fontos megjegyezni, hogy a pályakövető algoritmus nem fogja engedni a korlánál nagyobb szögsebesség érték használatát.)



3.9. ábra. A két pályához tartozó szögsebességszabályzó 0.8 1/s szögsebességhatár esetén.

A 3.10. ábra a kerekek eredő gyorsulását ábrázolja minden pálya esetén. Mindkét ábrán az eredő gyorsulás nagyságát ábrázoltuk, a profilok előjelét a tangenciális gyorsulás előjele adja meg, így szemléltethető, hogy mikor lassít és mikor gyorsít az adott kerék.

A mintavételezés hatása itt a legnagyobb, a 200 mm/s^2 gyorsuláshatárt jóval meghaladó tüskék jelennek meg a kerékgyorsulásban a mintavételezett pálya esetén.



3.10. ábra. A robot két kerekének eredő gyorsulása a geometriai és a mintavételezett pálya esetén 200 mm/s^2 gyorsuláskorlát esetén.

Összegzésképpen elmondható, hogy igen fontos az eredeti geometriai pálya felbontása, hiszen az időparaméterezés során csak a geometriai pálya esetén biztosítjuk a korlátok betartását. Viszont lehetőségünk van arra is, hogy a geometriai pálya felbontását a kritikusabb részeken (elsősorban kanyarokban) megnöveljük, hiszen az algoritmus nem feltételezi, hogy a geometriai pálya térbeli felbontása a pálya mentén egyenletes. Így a mintavételezésből adódó eltérések csökkenthetők.

4. fejezet

Pályakövető szabályozás

A pályakövetés ismertetése előtt tekintsük át, hogy pontosan mit értünk pályán, amelyet követni szeretnénk.

A követendő pályát szegmensekre bontjuk fel. Egy szegmensen belül a robot megállás nélkül halad előre vagy hátra a pálya mentén. Ebből következik, hogy egy szegmens a haladás irányából és a pálya időben egyenletesen mintavételezett koordinátáiból áll. Két szegmens között a robot egy helyben fordul a következő szegmens kezdeti irányába. Az előbb leírtak alapján két alapvető mozgásprimitívvel dolgozunk: egy helyben fordulás adott irányba és pályakövetés. Azt, hogy az adott szegmensnél a robot előre vagy hátrafelé halad, a pályatervező algoritmus dönti el, ahogyan a szegmenshatárokat is a pályatervező adja meg.

4.1. Egy helyben fordulás

Az egy helyben fordulásnál nem szabad megsérteni a robot maximális szögsebesség (ω^{max}) és szögggyorsulás (β^{max}) korlátját. Habár a 3.3. részben nem soroltuk fel a β^{max} -ot mint korlátot, azonban az eddig meghatározott korlátokból következik a szögggyorsulás korlát is:

$$\begin{aligned}\beta(i) &= \frac{a_{rt}(i) - a_{lt}(i)}{W} \\ \beta^{max} &= \frac{{a_{rt}}^{max} + {a_{lt}}^{max}}{W},\end{aligned}\tag{4.1}$$

ahol kihasználtuk, hogy a lassuláskorlát abszolútértéke megegyezik a gyorsuláskorlát abszolutétkével. Az i index időpontot jelöl szemben az előző fejezetben használt k indexsel, amely pályaponthoz tartozott.

Mielőtt a robot elkezdené a forgást, megvizsgáljuk, hogy melyik körüljárás szerint érdekes fordulni, és azt az irányt választjuk, amerre kisebb a szögkülönbség.

Alapvetően a fordulás a maximális szögggyorsulással történik, ha ez nem sérti a maximális szögsebességre vonatkozó korlátot. Ezenkívül azt szeretnénk, hogy a robot a fordulás végén pontosan a kívánt irányba álljon, nem lenne szerencsés, ha a robot túlfordulna, és utána ezt kellene kompenzálnunk. Ezért a legfontosabb kérdés, hogy mikor kell elkezdenünk a

szögsebességet csökkenteni, hogy a robot a korlátozás betartása mellett a kívánt irányban álljon meg.

A kérdés megválaszolásához kövessük végig az alábbi gondolatmenetet. Jelenleg a i . időpontban vagyunk és meghatároztunk egy $\omega(i)$ szögsebességet a korlátoknak megfelelően. Ekkor a következő időpontban a robot iránya:

$$\theta(i+1) = \theta(i) + \omega(i) \cdot \Delta t, \quad (4.2)$$

ahol $\theta(i)$ a robot orientációja i . időpontban.

Vizsgáljuk meg, hogy mi történne, ha a következő mintavételkor ($i+1$). időpontban a maximális szöggysorsulással elkezdenénk lassítani a fordulást:

$$\begin{aligned} \omega(i+1) &= \omega(i) - \beta^{max} \cdot \Delta t \\ \theta(i+2) &= \theta(i) + \omega(i) \cdot \Delta t + (\omega(i) - \beta^{max} \cdot \Delta t) \cdot \Delta t \\ &= \theta(i) + 2 \cdot \omega(i) \cdot \Delta t - \beta^{max} \cdot \Delta t^2 \end{aligned} \quad (4.3)$$

Általános esetben $\theta(i+n)$ értéke a következő szerint alakul, felhasználva a számtani sorozat összegképletét:

$$\theta(i+n) = \theta(i) + n \cdot \omega(i) \cdot \Delta t - n \cdot \frac{n-1}{2} \cdot \beta^{max} \cdot \Delta t^2 \quad (4.4)$$

Az is tudjuk, hogy mekkora a szögsebesség értéke $i+n$. időpontban:

$$\omega(i+n) = \omega(i) - n \cdot \beta^{max} \cdot \Delta t \quad (4.5)$$

Mi arra az állapotra vagyunk kíváncsiak, amikor a robot megállt, tehát amikor $\omega(i+n) = 0$. Ezt a 4.5. képlet alapján könnyedén megkapjuk ügyelve arra, hogy n egész szám:

$$n = \lceil \frac{\omega(i)}{\beta^{max} \cdot \Delta t} \rceil \quad (4.6)$$

Tehát az i . időpontban (felhasználva a 4.4. és a 4.6. egyenleteket) meg tudjuk határozni, hogy ha a következő mintavételkor elkezdünk maximális szöggysorsulással lassítani, akkor a forgás kívánt orientációját meghaladjuk-e. Amennyiben meghaladnánk, akkor nem adjuk ki az $\omega(i)$ beavatkozójelet, hanem már az i . időpontban elkezdünk lassítani.

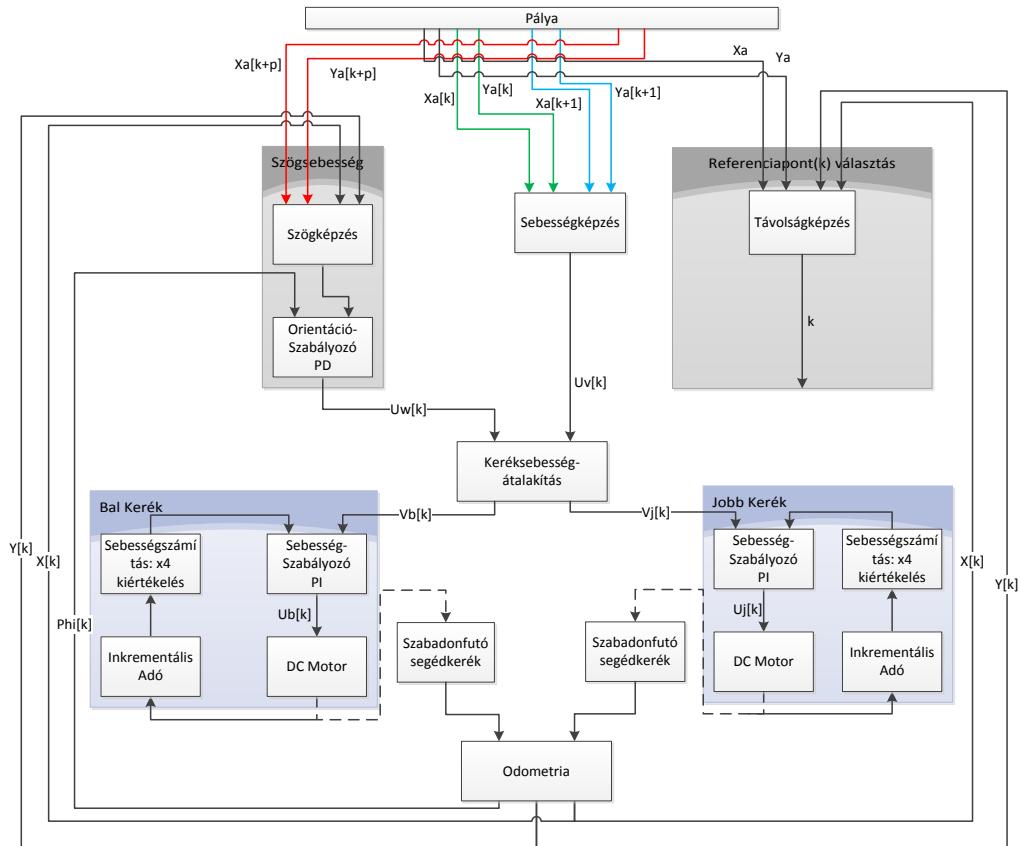
A lassítás közben minden mintavételkor újra megvizsgáljuk, hogy szükséges-e tovább lassítani. Előfordulhat, hogy azt kapjuk az előbb ismertetett eljárás alapján, hogy már nem kell. Ilyenkor viszont már nem kezdünk el maximális szöggysorsulással gyorsítani, hanem az előző mintavételkor használt szögsebességgel avatkozunk be ismét (0 szöggysorsulással).

Tehát az egy helyben fordulás szögsebesség-idő profilja alapvetően trapéz alakú, de a lefutó ágán vízszintes szakaszok is lehetnek.

4.2. Pályakövetés

A pályakövetés alapja, hogy szétcsatolt sebesség és szögsebesség-szabályozást hajtunk végre. A szétcsatolás következménye, hogy egyszerű, lineáris szabályozókat használhatunk a pályakövetés során. A pályakövetés felépítése a 4.1. ábrán látható.

Az egy helyben fordulásnál szögsebesség beavatkozójelet határozunk meg, míg a pályakövetésnél szögsebesség és sebesség beavatkozójelet. Az algoritmus végén ezeket az értékeket átszámoljuk keréksebességekre a 3.2. egyenlet alapján, tehát közvetlenül keréksebességek lesznek a beavatkozójeleink. Ez a 4.1. ábra középen látható *Keréksebesség-átalakítás* néven.



4.1. ábra. A pályakövetés áttekintő blokk diagramja.

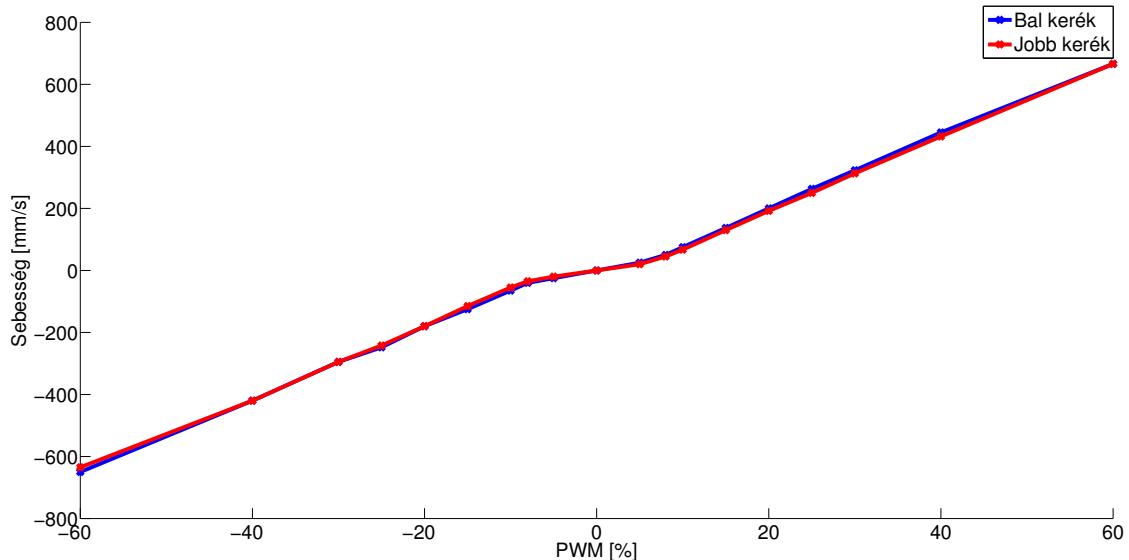
A 4.1. ábrán látható még két szabadonfutó segédkerék, ezeknek a pozíció visszacsatolás a szerepük. A pozíció visszacsatolás odometria segítségével történik, de a dolgozatomnak ez nem témaja.

4.2.1. Sebességszabályozás

Alacsony szinten, a keréksebességeknél történik a sebességszabályozás. Az általam használt valós robot két hajtott kerékkel rendelkezik, minden kerék esetén egy-egy DC motor gondoskodik a robot mozgatásáról. A DC motorok tengelyéhez egy-egy inkrementális adó csatlakozik, amely biztosítja a sebességszabályozás számára a visszacsatolást. Feszültségvezérelt egyenáramú motorok esetében széleskörűen alkalmazott gyakorlat PI szabályozók használata, így sebességszabályozásra én is PI szabályozókat használok.

A PI szabályozók esetében gyakran előforduló probléma az elintegrálódás [12]. Az elintegrálódás a rendszerben lévő beavatkozószerv telítései miatt lép fel, kiküszöbölése történhet többféleképpen szabályozó típusától függően. Esetben az integrátor visszaállításával előzöm meg az elintegrálódást.

Az általam implementált sebességszabályozók figyelembe veszik a motor és az áttetelek nemlinearitását is. Ehhez felvettek minden kerék esetén a rendszer karakteristikáját, tehát, hogy adott feszültség mellett mekkora sebességet ér el a kerék (4.2. ábra). Ennek a karakteristikának az inverzét beépítettem a rendszer modelljébe, így elméletileg a nemlineáritást kiejtettem.



4.2. ábra. A két kerék esetén a rendszer karakteristikája.

A karakteristikát a következők szerint vettetem fel. A robot két kerekére ugyanazt a feszültséget adtam ki beavatkozójelnek majd megmértem, hogy mekkora állandósult sebességet érnek el a kerekek. Ezt több feszültségnél megismételve, kiadódik a rendszer karakteristikája a két kerék esetén. A feszültségvezérlés impulzusszélesség-modulációval (PWM) történik, a 4.2. ábra x tengelyén a PWM jel kitöltési tényezője látható.

4.2.2. Referenciapont-választás

A sebességszabályozók számára a sebesség alapjelet a pálya biztosítja, hiszen az időparaméterezés során olyan pálya készült, amely időben egyenletesen mintavételezett, és így a

pályapontok közötti távolságból a robot sebessége kiszámolható.

Már csak azt kell eldöntenünk, hogy a pálya melyik pontjához tartozó sebesség alapjelet alkalmazzuk az adott mintavételnél. Ezt hívjuk *referenciapont-választásnak*. Az eljárás igen egyszerű, a pálya pontjai közül a robot pozíciójához legközelebbi pályapontot választjuk referenciapontnak, és így már egyértelműen adódik a sebesség alapjelünk is.

A fejlesztés egy korai stádiumban felmerült, hogy ezt a referenciapontot ne így határozzam meg, hanem folyamatosan léptessem a pálya mentén. Ezzel kvázi előírtam, hogy a robot adott időpontban a pálya mely pontjában tartózkodjon. Mivel nem biztos, hogy a robot ténylegesen a kívánt pozícióban található, egy külön alapjelmódosító szabályozó segítségével korrigáltam a pályába kódolt sebesség alapjelet, hogy a robot elérje a referenciapontot.

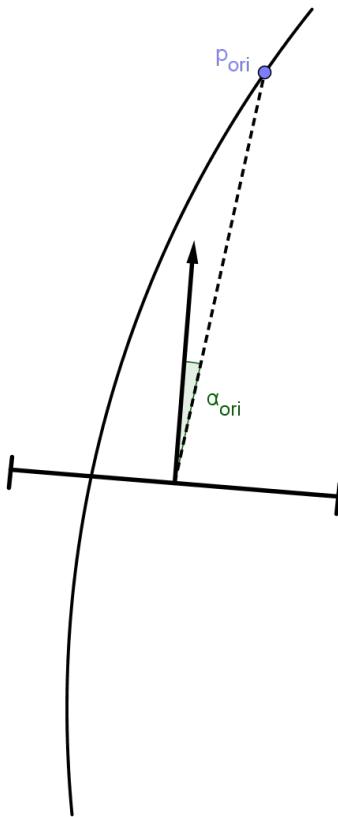
Amennyiben nem ideális modellt használtam, a megoldás nem műköött, a rendszer instabillá vált. Később kiderült, hogy a megoldás problémája az volt, hogy egyrészt előírtam a robot számára, hogy mekkora sebességgel haladjon a pálya mentén, másrészt a referenciaponton keresztül azt is, hogy hol tartózkodjon az adott időpontban. Ez már azért sem lehetséges, mivel, ha a robot a referenciaponthoz képest lemaradásban van (általában ez történik), akkor a sebességalapjel korrekció növelné a sebességet, pedig azt már eleve úgy írtuk elő, hogy a lehető leggyorsabban haladjon a robot a pálya mentén. Tehát az alapjelmódosító szabályozóval arra kényszeríteném a rendszert, hogy szegje meg a saját korlátozásait.

A végleges megoldásnál ezzel szemben a referenciapontot alakítjuk a robothoz, nem pedig fordítva. Ez azt jelenti, hogy nem írjuk elő, hogy a robot a pályát mennyi idő alatt járja be, csak azt, hogy a pálya adott pontjában mekkora sebességgel avatkozzunk be.

A pályakövető algoritmusnál lényeges szempont a futási idő, mivel valós roboton is működnie kell. Ezért a referenciapont meghatározásánál nem megyünk végig a pálya összes pontján. A legközelebbi pont keresését az előző iterációban használt referenciapontnál kezdjük, és csak egy bizonyos számú pontot vizsgálunk meg. Ha a robot korlátai megfelően lettek beállítva, akkor az egymás utáni referenciapontok között körülbelül egy pályapont különbségnek kell lennie. Ezért teljesen felesleges a pálya összes pontját megvizsgálnunk.

4.2.3. Orientációszabályozás

Az orientációszabályozás feladata szögsebesség alapjel biztosítása a pályakövetés során. Az orientációszabályozáshoz felhasználjuk a robot aktuális pozícióját és a pálya egy pontját (p_{ori}). A szabályozó alapjelét a robot pozíciója és p_{ori} pont közötti irány és a robot aktuális orientációjának különbsége adja meg (α_{ori} a 4.3. ábrán). A konkrét orientációszabályozó egy PD szabályozó.



4.3. ábra. Az orientáció-szabályozás referenciaPontja.

Fontos kérdés, hogy a pálya mely pontját választjuk az alapjelszámoláshoz. mindenépp a sebességszabályozónál használt referenciaPontnál távolabbi pontot keresünk, hiszen a referenciaPont van a robothoz legközelebb, és nem akarjuk, hogy az orientációszabályozás a pályán visszafelé irányítsa a robotot.

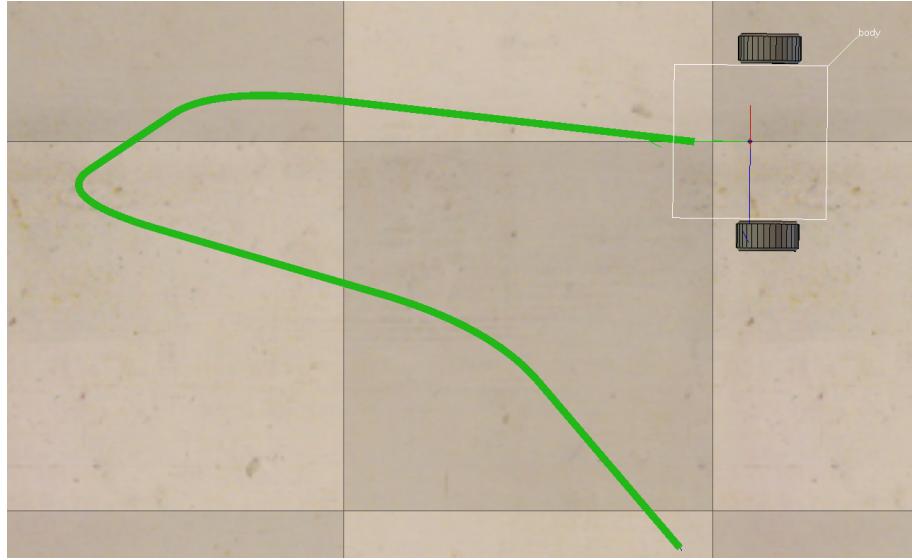
Alapvetően két megközelítést alkalmazhatunk. Egyszer csak használhatunk konstans távolságú előretrekintést, ekkor mindenkor a referenciaPonttól egy adott távolságra lévő pályapon-tot használunk az orientációszabályzáshoz. A másik megközelítésnél pedig konstans idejű előretrekintést alkalmazunk. Ennél a módszernél a referenciaPonthoz képest adott számú mintavétellel előbbre lévő pályapont lesz p_{ori} .

Az algoritmust úgy készítettem el, hogy mindenkor a módszert lehet alkalmazni, akár a kettőt egyszerre is, a következő módon. A konstans távolságú előretrekintéssel elérhető, hogy a robothoz ne kerüljön túlságosan közel p_{ori} , a konstans idejű előretrekintés pedig lehetővé teszi, hogy egyenes részeken, ahol nagyobb sebességgel mozog a robot, távolabb tekintsünk.

4.2.4. Túlhaladás problémája

A pályakövetés valós roboton történő tesztelése során egy eddig nem tárgyalt problémát vettet észre. Amennyiben a robot egy pályaszegmensen túlmegy, tehát nem pontosan a szegmens utolsó pontjánál áll meg, akkor a sebességalapjel kiválasztása nem megfelelő. A probléma akkor is jelentkezik, ha a robot kezdőpontja hátrébb található, mint a pályaszeg-

mens első pontja. Ez az eset látható a 4.4. ábrán.



4.4. ábra. A túlhaladás problémáját bemutató szituáció szimulátorban. A robot alakját nem mutatja az ábra, csak a határolónonalait. A zöld görbe a követendő pálya.

A probléma az ezekben a szituációkban, hogy a referenciapont-választás alapján a szegmens első pontjához tartozó sebesség értékét választja a pályakövető algoritmus alapjelnek, mivel ez a pont található a legközelebb a robot aktuális pozíciójához. Ez a sebesség viszont igen alacsony, mivel azt feltételezi, hogy a robot éppen elindul és amíg a robot nem éri el a pálya második pontját, végig ezzel az alacsony sebességgel fog haladni.

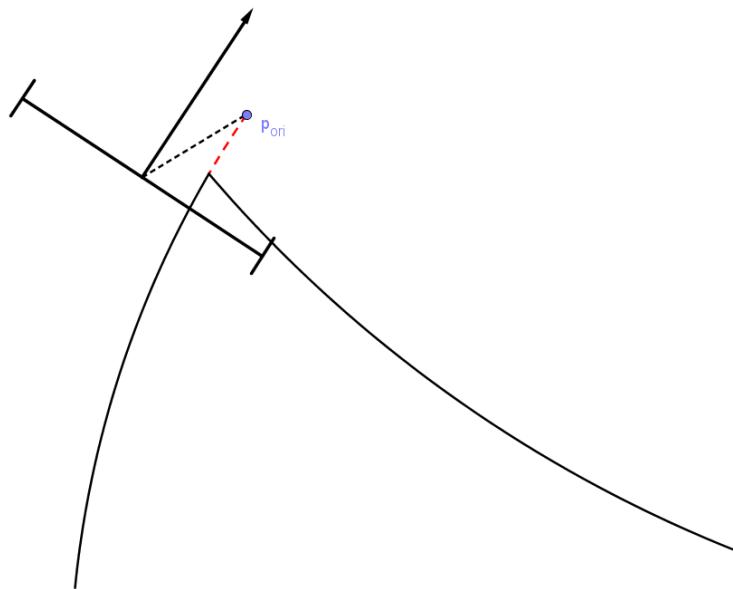
A probléma elkerülése érdekében amíg a pályaszegmens első pontja van a legközelebb a robot aktuális pozíciójához, addig nem a pályába kódolt sebességet használom, hanem a robot gyorsulás- és sebességkorlátja alapján választok robotsebességet.

A valóságban nem a 4.4. ábrán látható esett szokott előfordulni, mivel a pályát eleve a robot pozíciójából tervezük. Ezzel szemben két szegmens között nem garantált, hogy a második szegmens kezdőpontja megegyezik a robot aktuális pozíciójával. Fontos megjegyezni, hogy valós robotnál a túlhaladás mértéke, megfelelő korlátok esetén, nem akkora, mint ahogyan a 4.4. ábrán látható..

4.2.5. Orientációsabályozás szegmens végpontjánál

A pályaszegmensek végpontjában jelentkezhet egy másik probléma is. A probléma az orientációsabályozás számára a p_{ori} pont meghatározása, mivel a szegmens végén p_{ori} közel kerülhet a robot aktuális pozíciójához, hiszen p_{ori} maximum a szegmens utolsó pontja lehet. Túlhaladás esetén pedig akár meg is előzheti a robot p_{ori} pontot, és így visszafordulna a robot, miután túlhaladt a szegmensen. Ezt az esetet mindenképp el kell kerülni.

A probléma megoldása érdekében a szegmens végét kiterjesztjük abban az esetben, ha p_{ori} a szegmens utolsó pontja lenne. A kiterjesztés azt jelenti, hogy a pályaszegmens utolsó két pontja által meghatározott egyenes pontját választjuk p_{ori} -nak. A kiterjesztett egyenes a 4.5. ábrán piros színnel látható.

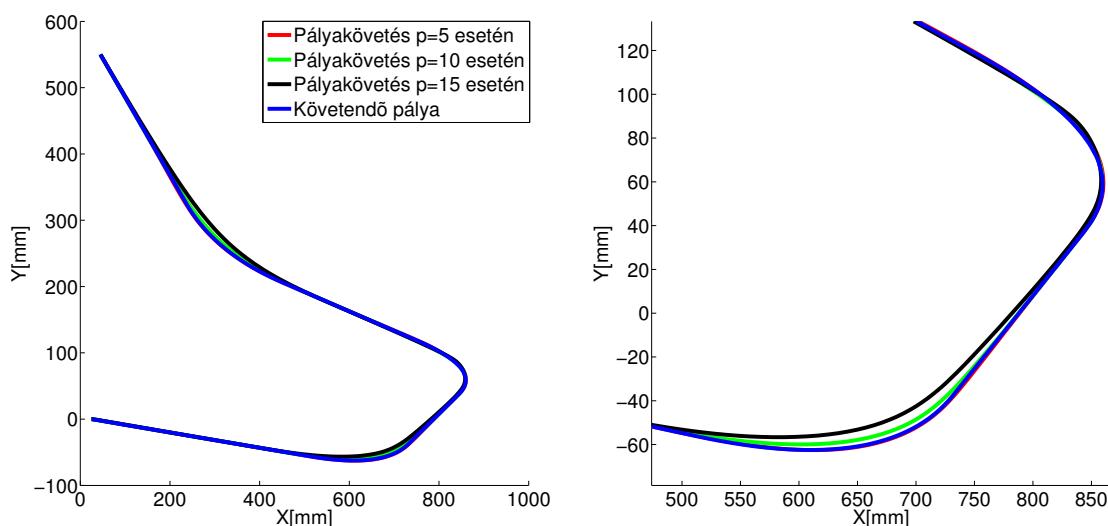


4.5. ábra. A pályaszegmens kiterjesztése.

4.3. Eredmények

Most pedig tekintsük át a pályakövetés alakulását különböző paraméterek esetén. A sebességszabályozó esetében a PI szabályozó paraméterein kívül nincsen más paraméterünk, a robotnak a pályában kódolt sebességet módosítás nélkül kell lekövetnie.

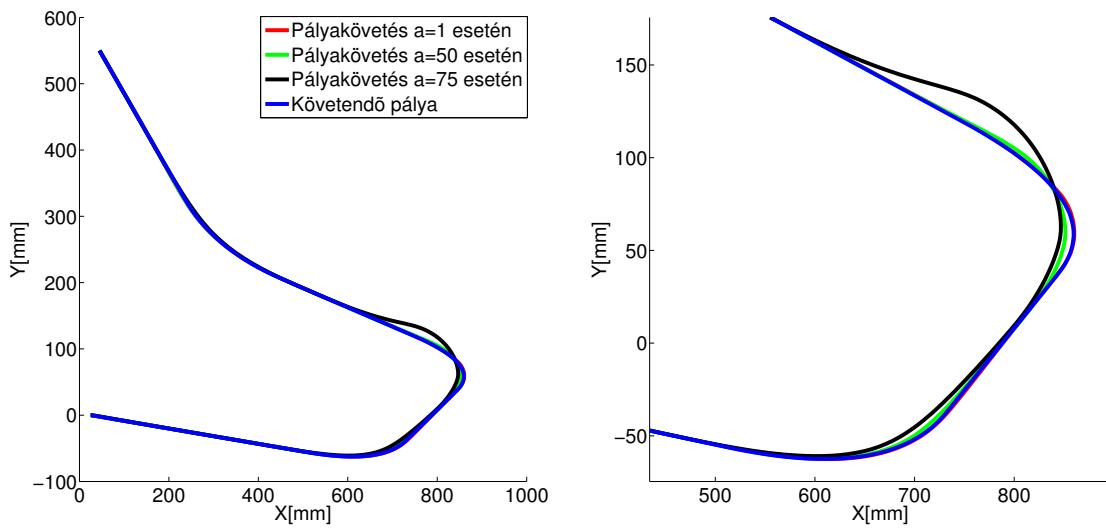
Az orientációs szabályozó esetében p_{ori} meghatározásához két paraméter áll a rendelkezésünkre. Egyrészt a konstans idejű előretékintéshez meghatározhatunk egy p paramétert, amely megadja, hogy p_{ori} hány mintavétellel legyen távolabb a robot aktuális referencia-pontjához képest. A p paraméter hatása a 4.6. ábrán látható három különböző esetben. Az ábrán vázolt esetekben a konstans távolságú előretékintés értéke 2 milliméter.



4.6. ábra. A konstans idejű előretékintés hatása.

A 4.6. ábrán látható, hogy p paraméter segítségével határozhatjuk meg, hogy a robot a pálya kanyarjait milyen mértékben vágja le. Amennyiben túlságosan kicsi p értéket választunk, akkor a robot mozgásában lengés jelenhet meg. Ez inkább a valós roboton történő mérésekkel fordul el. Valós roboton végzett méréseket a következő fejezetben láthatjuk majd.

A 4.7. ábrán pedig a konstans távolságú előretrekintés hatását láthatjuk három esetben. Ebben az esetben az a paraméter azt adja meg, hogy p_{ori} pont mekkora távolságra (milliméterben) legyen a robot referenciaPontjától. Itt p értéke 5 mintavétel.



4.7. ábra. A konstans távolságú előretrekintés hatása.

Éles kanyar esetén túlságosan nagy a paraméter esetén a pályakövetés pontatlan. Mivel a gyakorlatban ezt a paramétert arra használtuk, hogy csökkentsük a lehetőséget, hogy a robot megelőzi p_{ori} pontot, ezért nem alkalmaztunk 20 milliméternél nagyobb értéket.

5. fejezet

Rendszer implementációja

A fejezet elején röviden bemutatunk egy újabb pályatervező algoritmust, majd a szimulációs környezetet, amelyben a dolgozatban ismertetett algoritmusokat teszteltem. Végül áttérünk a valós robot bemutatására. Alapvető céлом volt, hogy a szimulációnál használt programokat minél kevesebb változtatással tudjam a valós roboton is alkalmazni. Ezenkívül a fejezetben megvizsgáljuk a szimulátorban és valós roboton elért eredményeket.

5.1. C*CS pályatervező

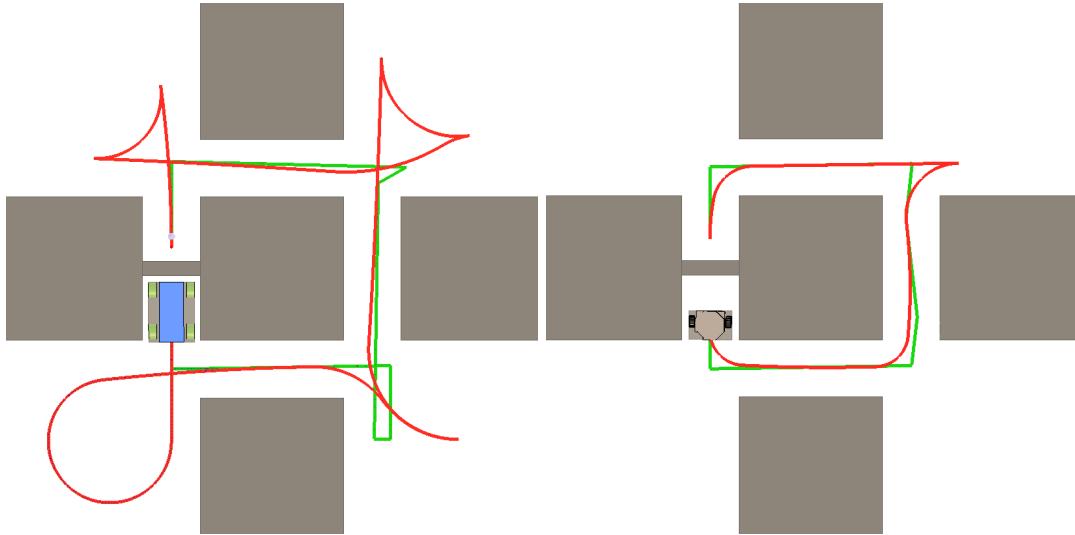
Mivel az RTR tervező egyenes és egyhelyben fordulás primitívekkel dolgozik, ezért sok esetben a differenciális robot gyorsabban is végre tudná hajtani a pályát, amennyiben nem kellene minden egyenes szakasz végén megállnia. A most bemutatásra kerülő lokális pályatervező segítségével gyorsabban végrehajtható pályát kapunk.

Az 1.2.2. fejezetben már ismertetük az úgynevezett approximációs pályatervezési módszert. A módszer lényege, hogy a globális tervező által tervezett pályát egy lokális tervező segítségével közelítjük. Jelen esetben a bemutatott RTR tervezőt használjuk, mint globális tervezőt. Lokális tervezőnek pedig az úgynevezett *C*CS* tervezőt használjuk, amelyet szintén Kiss Domokos dolgozott ki, ahogyan az RTR tervezőt is. A *C*CS* algoritmust Csorvási Gábor implementálta C++ nyelven, az ő megoldását használtam fel a dolgozatomban [13].

A *C*CS* egy autók számára kifejlesztett algoritmus. Az autókra jellemző, hogy van minimális fordulási sugaruk. Mivel a differenciális robotok fordulási sugara nulla (képesek egyhelyben fordulni), végre tudják hajtani az autók számára tervezett pályát is.

A *C*CS* algoritmus lényege, hogy körívek és egyenesek segítségével közelítjük a globális tervező által tervezett pályát. Az algoritmus nevében szereplő *C* betű körívre utal, az *S* egyenesre és a *C** pedig olyan körívre, amelynek sugara végtelen is lehet (egyenes).

Az RTR tervező és a *C*CS* tervező együttes alkalmazását az 5.1. ábrán láthatjuk.



5.1. ábra. A bal oldalon autószerű robot esetében látható egy szituáció, míg a jobb oldalon differenciális robot esetében. A C^*CS algoritmus megoldása piros színnel látható, míg a globális pálya (RTR) zöld színnel.

5.2. V-REP robotszimulátor

A pályatervező, időparaméterező és pályakövető algoritmusok tesztelésére a V-REP robotszimulátort használtam. A programot a Coppelia Robotics cég fejleszti. A szimulátor oktatási célra ingyenesen letölthető, használható [14]. A program széles körűen használható a robotika minden ágában. Szimulálhatók benne ipari szerelőrobotok, ahogyan az 5.2. ábrán is látható, de mobil robotok esetében is igen hasznos eszköz. Valós robotok programozására is alkalmas. A szimulátor működése jól dokumentált, sok oktató anyagot, példaprogramot készítettek a fejlesztők hozzá, és ezenkívül folyamatosan frissítik, új funkciókkal bővítik.

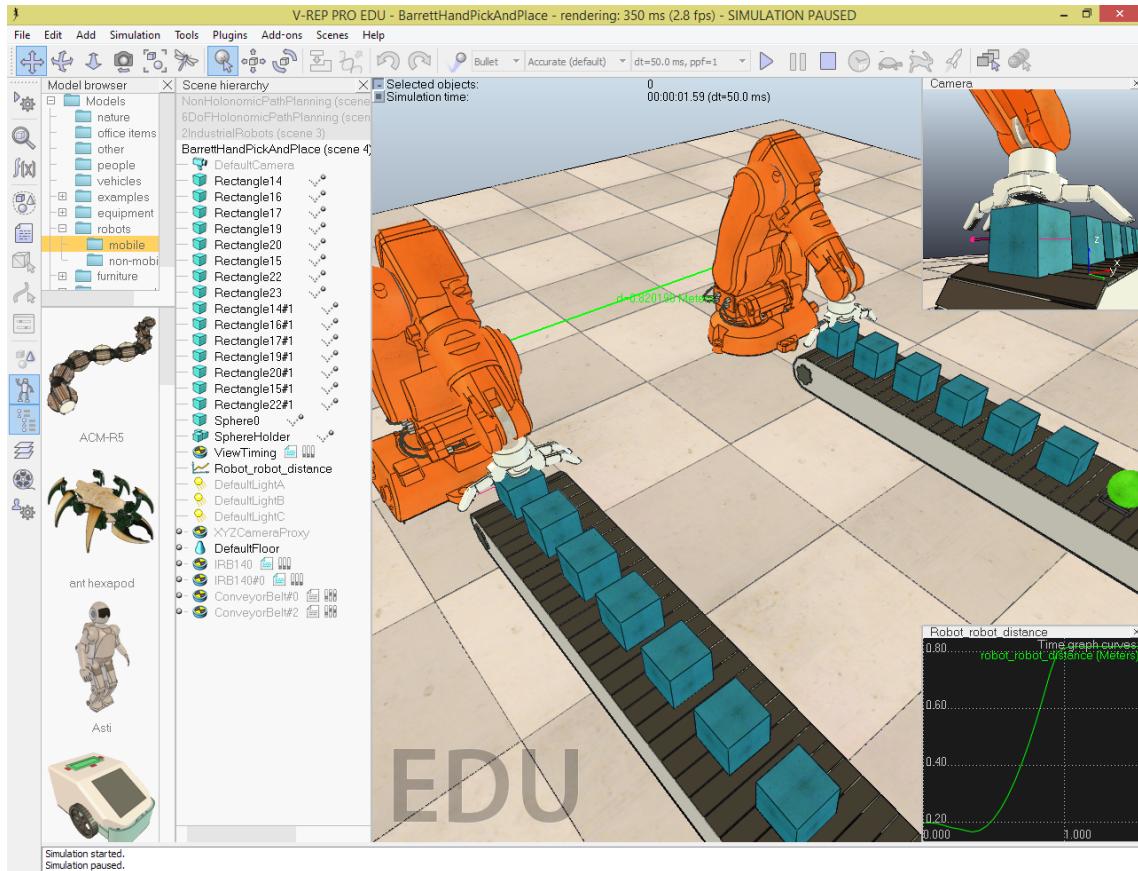
5.2.1. Keretrendszer

A V-REP működése több módon is kiegészíthető. Csorvási Gáborral közösen egy keretrendszer dolgoztuk ki. A keretrendszer része az általunk készített szerver és a kliensprogram. A V-REP-hez készítettünk egy Lua szkriptet, amely kapcsolódik a szerver programhoz, elküldi számára a szimuláció paramétereit, majd a szerverről kapott adatok alapján mozgatja a robotot, kirajzolja a megtervezett pályát, esetleg egyéb hasznos információkat jelenít meg. A szerver programhoz kapcsolódik a kliensprogramunk is. A kliensprogram futtatja a pályatervező, időparaméterező és a pályatervező algoritmusokat. Az algoritmusok paramétereit a Lua szkript határozza meg a szerveren keresztül. A szerver működése teljesen transzparens, az alkalmazás tekinthető a szimulátor részének. A keretrendszer felépítését az 5.3. ábra szemlélteti.

A keretrendszer felépítése azért is előnyös, mert a kliensprogram helyett akár a valós robot vezérlőszoftvere is kapcsolódhat a szerverhez. Így a V-REP-ben a valós robot aktuális konfigurációját láthatjuk. A valós roboton futó algoritmusok beállításai szintén a Lua szkriptből megadhatóak.

Mivel Csorvási Gábor témaja hasonlóan pályatervezéssel és pályakövetéssel kapcsolatos,

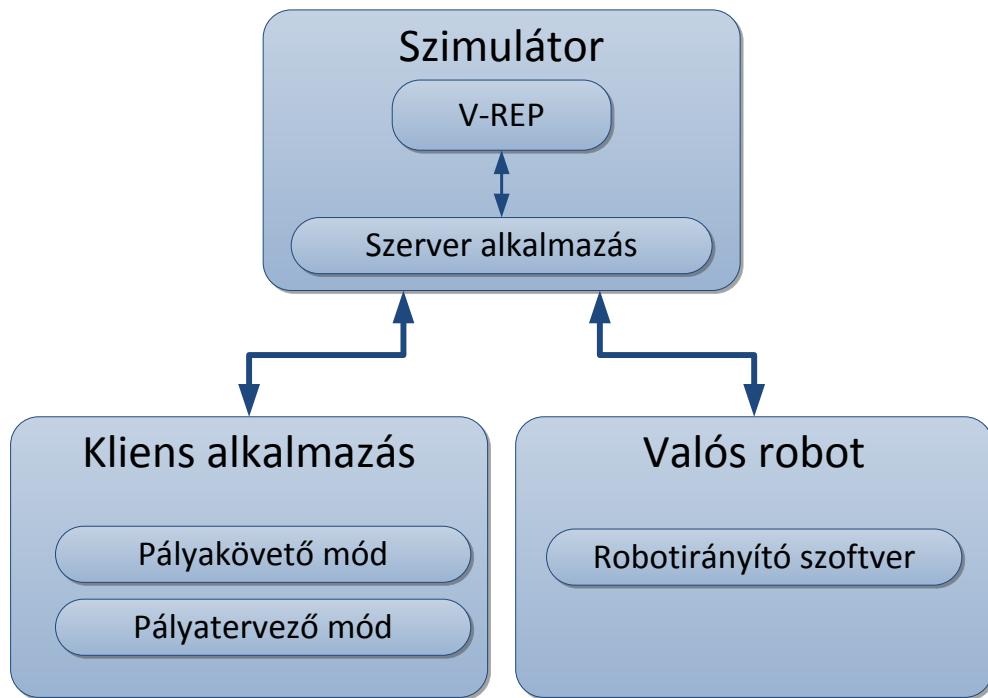
de nem differenciális robot, hanem autószerű robot esetén, így fontosnak tartottuk úgy elkészíteni a keretrendszer elemeit, hogy függetlenek legyenek a robot típusától. A Lua szkriptben megadható, hogy a szimuláció vagy a valós roboton végzett mérés differenciális vagy autószerű roboton fusson. A két robottípushoz más paraméterek tartoznak, és csak az adott típushoz tartozó paramétereket kapja meg a szerver és a kliensalkalmazás.



5.2. ábra. A V-REP szimulációs program

Kliensprogram

A kliensprogram alapvetően két módban futtatható. Az egyik módban az időparaméterező és a pályakövető szabályozás tesztelhető. Ekkor a kliens a szimulátortól fogad egy előre elkészített pályát, majd ezt az időparaméterező algoritmus segítségével újramintavételezi és a kapott pályát visszaküldi a V-REP-nek, amely megjeleníti azt. Innentől a pályakövető algoritmus kerül előtérbe. A szimulátor kezdetben elküldi a robot aktuális pozíóját a kliensnek, majd minden szabályozási ciklusban a pályakövető eljárás elküldi a szervernek az aktuális beavatkozó jeleket, amely alapján megtörténik a robot konfigurációjának kijelzése. A szimulátor és a kliens alkalmazás működése szinkronizálva van, azaz megvárják egymást a következő lépéssel.



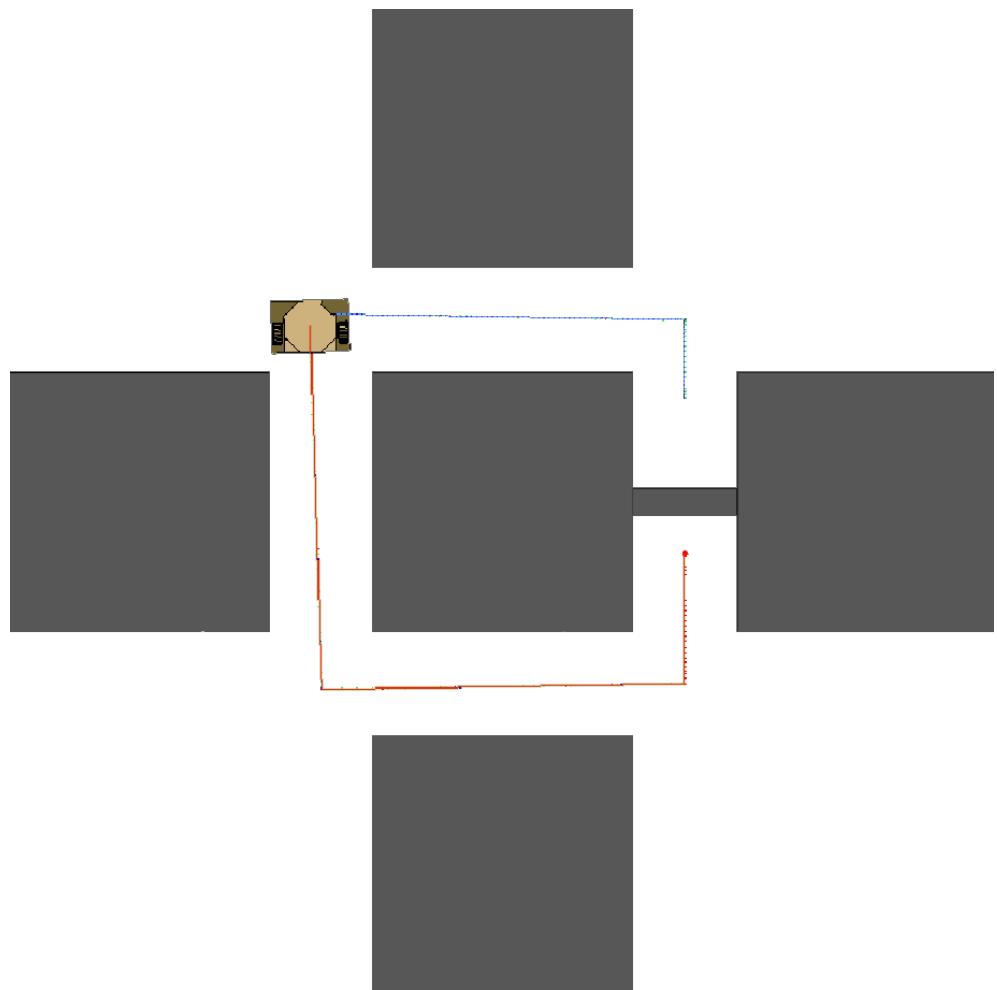
5.3. ábra. Az elkészült keretrendszer blokkvázlata

A kliens másik módja a pályatervezés teszteléséhez használható. Ebben az esetben a kliens nem kap előre elkészített pályát, hanem egy környezetet kap. Környezeten értjük az akadályok leírását, beleértve a környezet határvonalait is. Emellett a pályatervezéshez szükséges egy kezdő- és célkonfiguráció, és a robot alakjának leírása is. Ezeket szintén megkapja a kliens a szimuláltortól. A pályatervezés után a kliens működése megegyezik a pályakövető módnál leírtakkal, csak a követendő pályát most nem a V-REP szolgáltatja, hanem a pályatervező algoritmus.

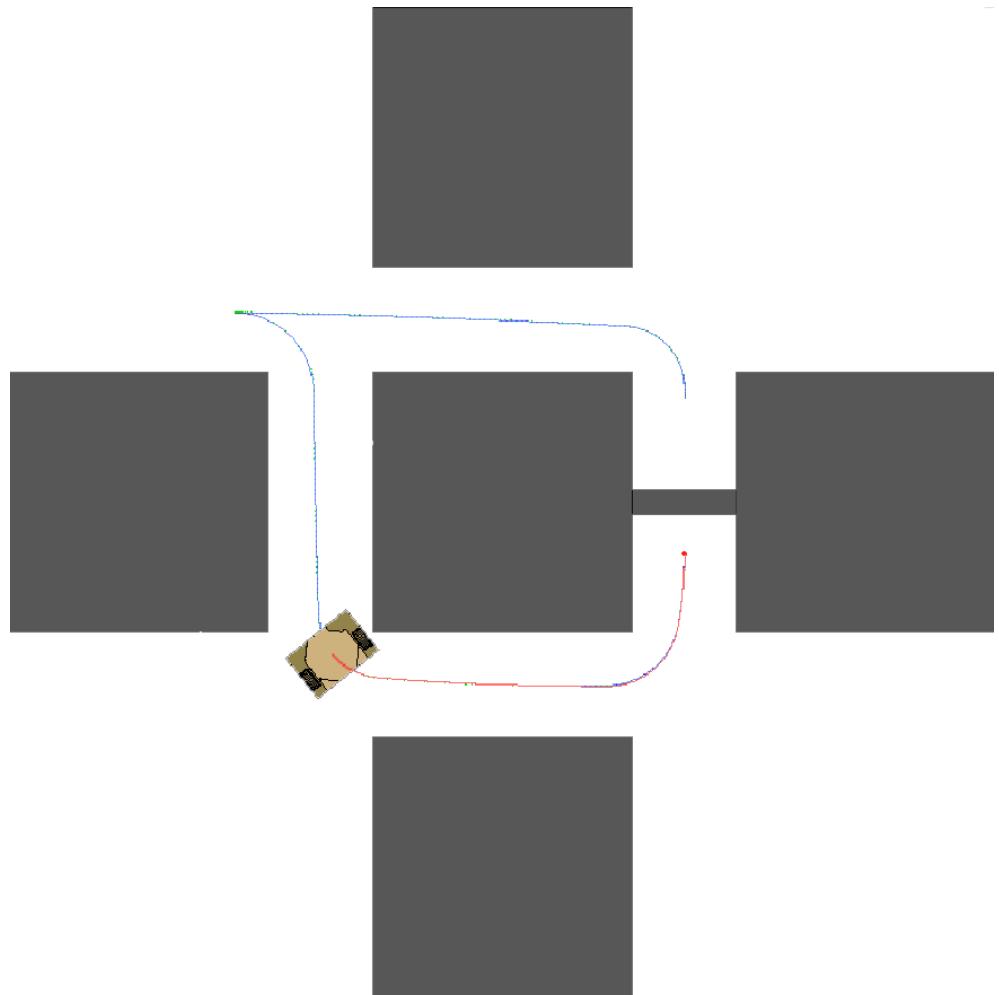
Ahogyan az algoritmuskosokat, úgy a klienst is C++ nyelven implementáltuk. A C++ nyelv azért is volt előnyös választás, mivel így valós roboton, beágyazott környezetben is könnyedén használhatóak az algoritmuskosok. A beágyazott környezet miatt igyekeztünk kerülni bármiféle olyan külső szoftvercsomag használatát, aminek a használata problémás lehet a valós roboton.

5.2.2. Szimulációs eredmények

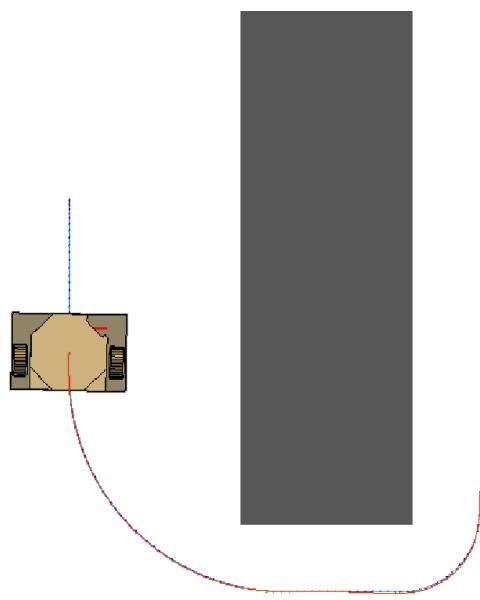
A szimulációs környezetben bemutatott algoritmuskosok a várakozásaimnak megfelelően működtek. Az eredmények a következő ábrákon láthatóak. A képek a V-REP szimulátorról készültek, az akadályokat szürke sokszögek jelzik, a pályatervező által generált pályát kék színnel jelöljük, míg a robot által bezárt pályát pirossal. A robot alatt található sokszög jelzi a tervezés során ténylegesen figyelembe vett robot alakot.



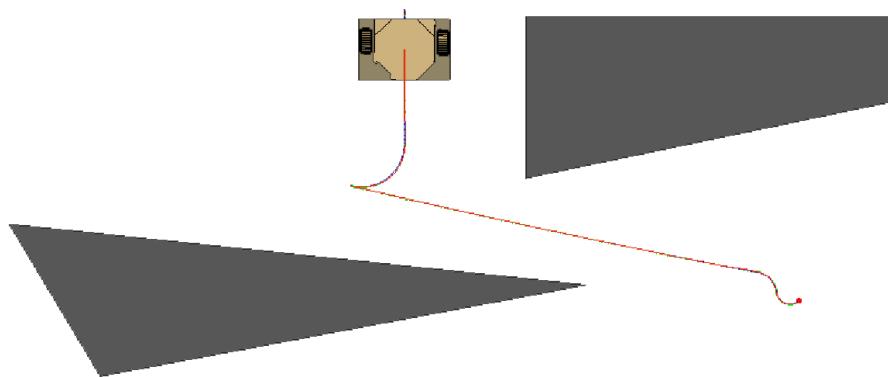
5.4. ábra. Ez a pálya differenciális robot esetében viszonylag egyszerűen teljesíthető, mivel a robot képes egyhelyben fordulni. A pályát az RTR tervező generálta.



5.5. ábra. Az előző pálya RTR és C*CS tervező együttes használatával.



5.6. ábra. Itt is szép megoldást eredményezett az RTR és C*CS tervezők közös használta.



5.7. ábra. Más alakú akadályok sem jelentenek problémát.

5.3. Valós robot

A dolgozatban bemutatott algoritmusokat az Eurobot 2013 nemzetközi robotversenyre elkészített differenciális roboton teszteltem. A robotot az Automatizálási és Alkalmazott Informatikai Tanszék vezetésével működő Robo2BME csapat tagjai tervezték.



5.8. ábra. A dolgozatban használt valós differenciális robot.

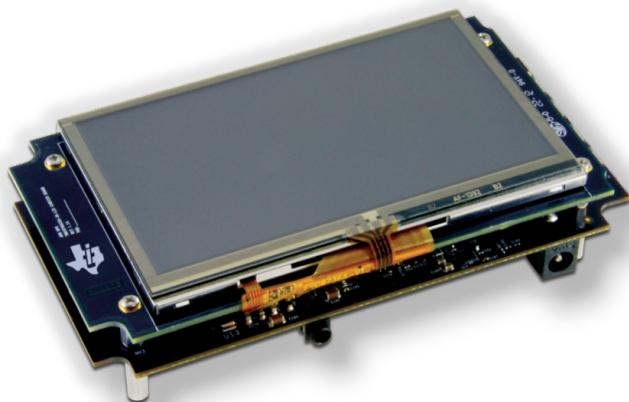
5.3.1. A robot irányítórendszere

A robot irányítórendszere alapvetően két szintből áll. A robot alsó szintű vezérlése elosztott modulokkal történik. Ezek a modulok felelnek a robot tápellátásáért, a robot különböző beavatkozószerveinek vezérléséért és a robot mérő- és érzékelőszerveitől érkező jelek fogadásáért (beleértve a robot pozíciójának mérését, odometria segítségével). A modulok CAN buszon kommunikálnak egymással és egy CAN-Ethernet átjáró segítségével a felső szintű irányítórendszerrel. minden modul két részből áll, egy feladat specifikus kártyából és egy általános DSP kártyából, amely a modul szoftverét tartalmazza.

Az egyik modul felelős a robot mozgatásáért, így a dolgozatban bemutatott algoritmusok egy részét a modulon található DSP processzoron kellett implementálnom. Mivel a DSP processzorokat elsősorban jelfeldolgozási feladatokra használják, alapvetően a pályakövető algoritmust implementáltam ezen az alsó szinten. A pályatervező és a hozzá kapcsolódó időparaméterező eljárást értelmetlen lett volna a mozgatásáért felelős modulon implementálni, mivel a modul nem rendelkezik minden információval a pályatervezéshez, és a modulon használt DSP teljesítménye nem megfelelő a pályatervezés megvalósításához. A DSP a Texas Instruments által gyártott TMS320F2809 32 bites fixpontos processzor.

A robot felső szintű vezérlőegysége felelős a robot stratégiájának autonóm megvalósításáért. A stratégián elsősorban valamilyen összetett feladat elvégzését értem, amelynek szerves része a pályatervezés, akadályelkerülés. A valós robot esetében, a pályatervező és időparaméterező algoritmusokat ezért ezen a szinten implementáltam.

A robot felső szintű vezérlőegysége egy C++ program, amely TCP/IP protokollon keresztül kapcsolódik az alsó szintű kártyához. Ebből következik, hogy ez az úgynevezett RobotPilot szoftver akár különböző architektúrákon is futhat, hiszen csak egy TCP kapcsolatra van szüksége a robot vezérléséhez. Én egyszer csak számítógépen futtattam a szoftvert, és ekkor egy UTP kábel biztosította a kapcsolatot a robot alsó szintjéhez. Másrészt egy beágyazott számítógépre is lefordítottam a RobotPilot szoftvert, amelyet a roboton helyeztem el. A beágyazott számítógép a Texas Instruments AM335x Starter Kit kártyája volt. Ez a kártya nagy teljesítményű ARM Cortex-A8 processzort, érintőképernyőt és igen sokféle perifériát tartalmaz. A perifériák közül az Ethernet és a WIFI modult használtam elsősorban. Az Ethernet modul segítségével kapcsolódott a robot alsó szintjéhez a kártya, WIFI-n keresztül pedig a V-REP-et futtató számítógéphez.



5.9. ábra. A robot felső szintű szoftverét futtató beágyazott számítógép.

Mind asztali, mind beágyazott számítógép esetében a RobotPilot csatlakozik a V-REP-hez tartozó szerver alkalmazáshoz, így a V-REP-ben megjelenik a robot aktuális konfigurációja. Ugyanakkor a V-REP segítségével a pályatervező és pályakövető algoritmusok paramétere megadhatóak, ahogyan szimuláció esetén is.

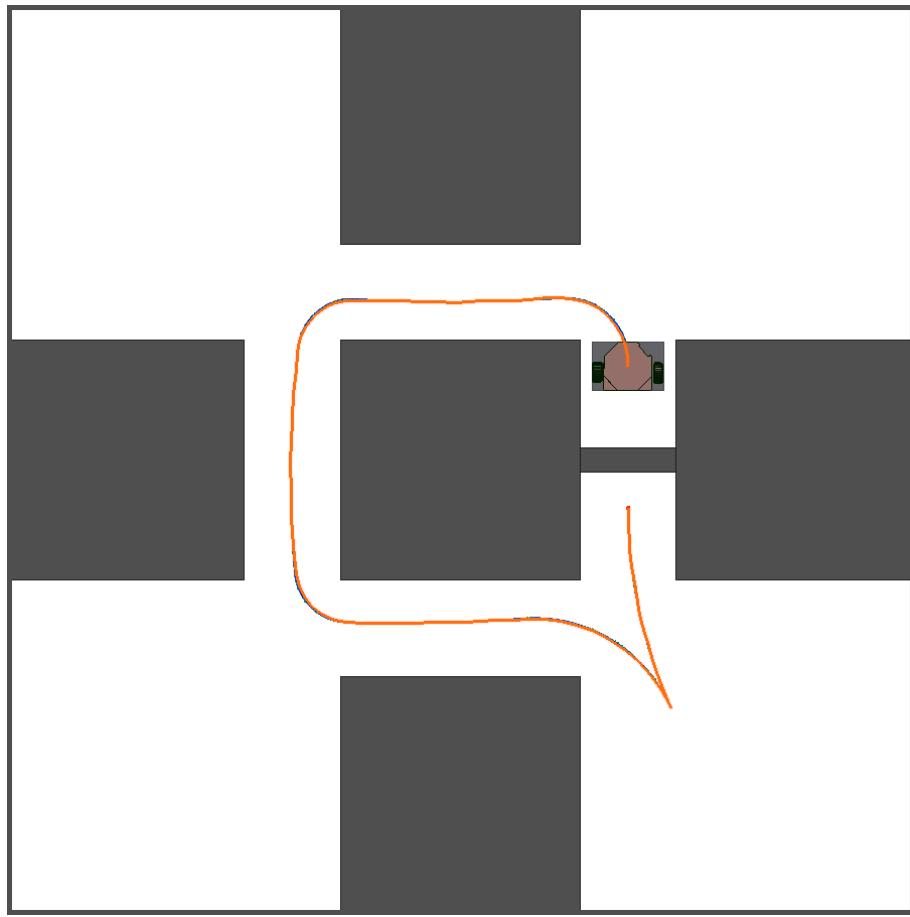
5.3.2. Valós eredmények

A most következő ábrákon a valós roboton végzett tesztelés eredményei láthatóak. Ezek a képek is a V-REP robotszimulátorról készültek, ahogyan a szimulációs eredmények is. Az akadályokat szürke sokszögek jelzik, a pályatervező által generált pályát zöld színnel jelöljük, míg a robot által bezárt pályát narancssárgával. A robot kiindulási pozícióját egy piros pont jelöli a pályán. Itt is a robot alatt található sokszög jelzi a tervezés során ténylegesen figyelembe vett robot alakot, amely közelítőleg megegyezik a valós robot alakjával.

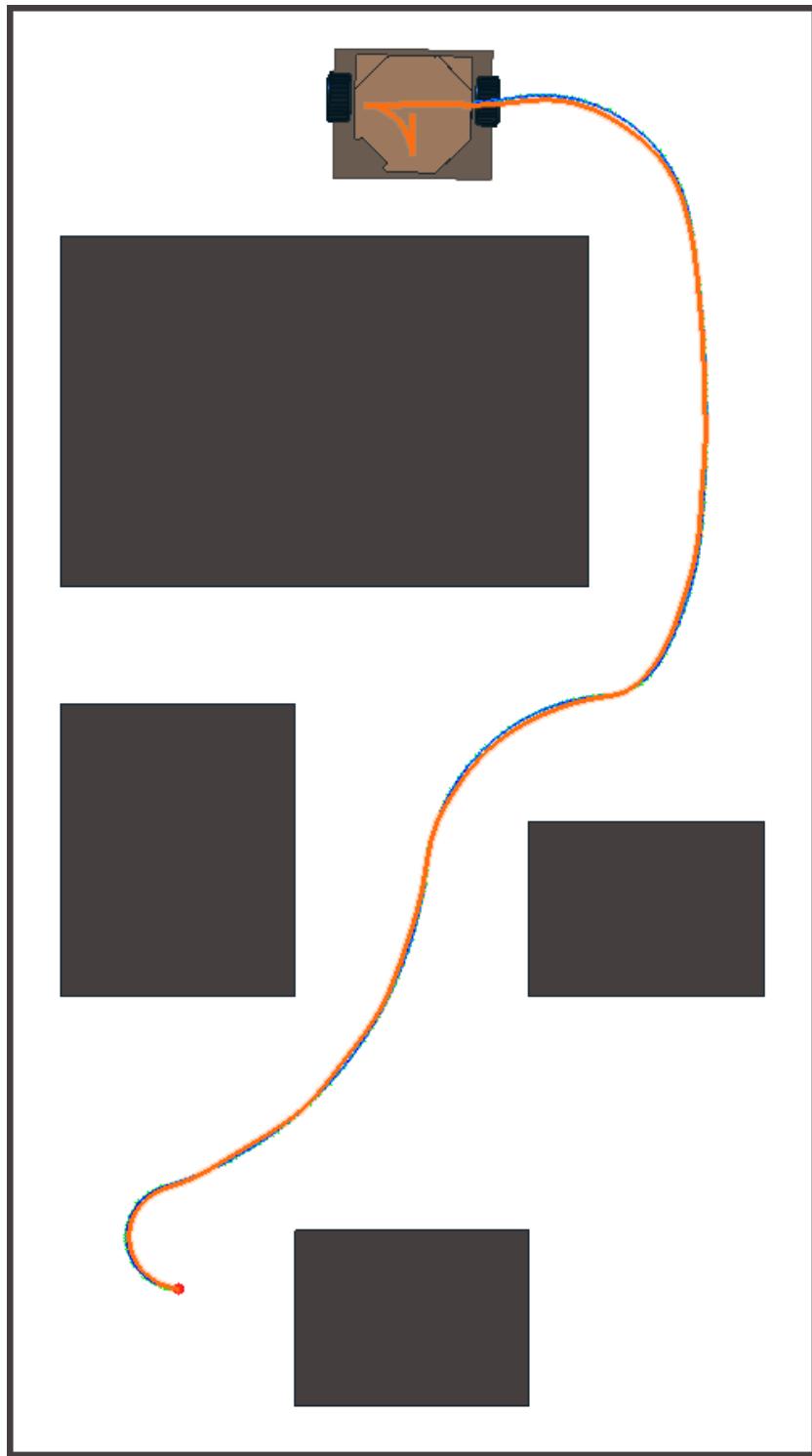
A pályákat a robot a következő paraméterekkel hajtotta végre:

- 200 mm/s robot pályamentri sebesség korlát
- 1.4 rad/s robot szögggyorsulás korlát
- 200 mm/s^2 eredő kerékgyorsulás korlát
- 200 mm/s^2 tangenciális kerékgyorsulás korlát

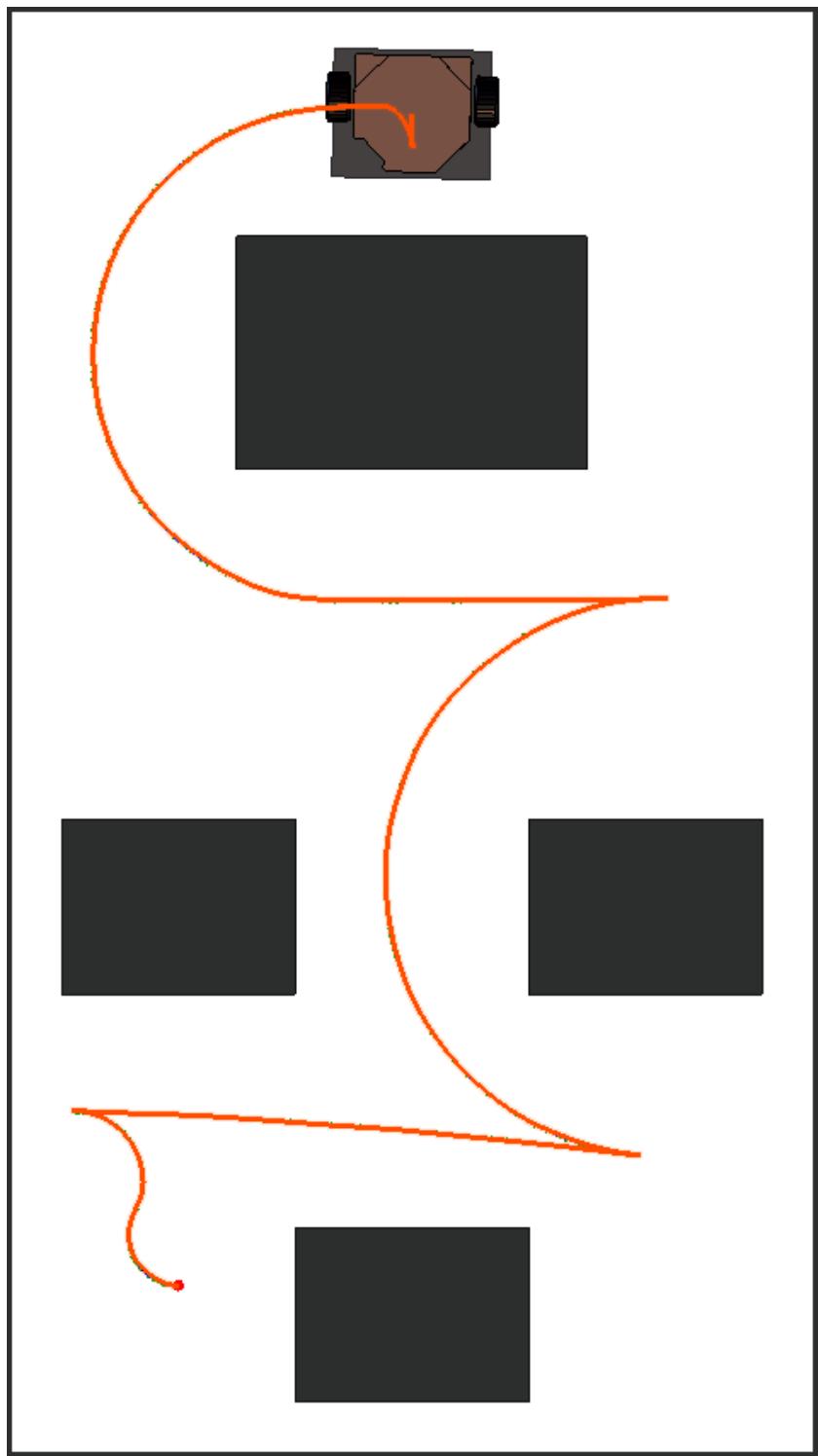
Az 5.13. pályánál viszont a sebesség és gyorsuláskorlátok kétszer ekkora értékűek voltak.



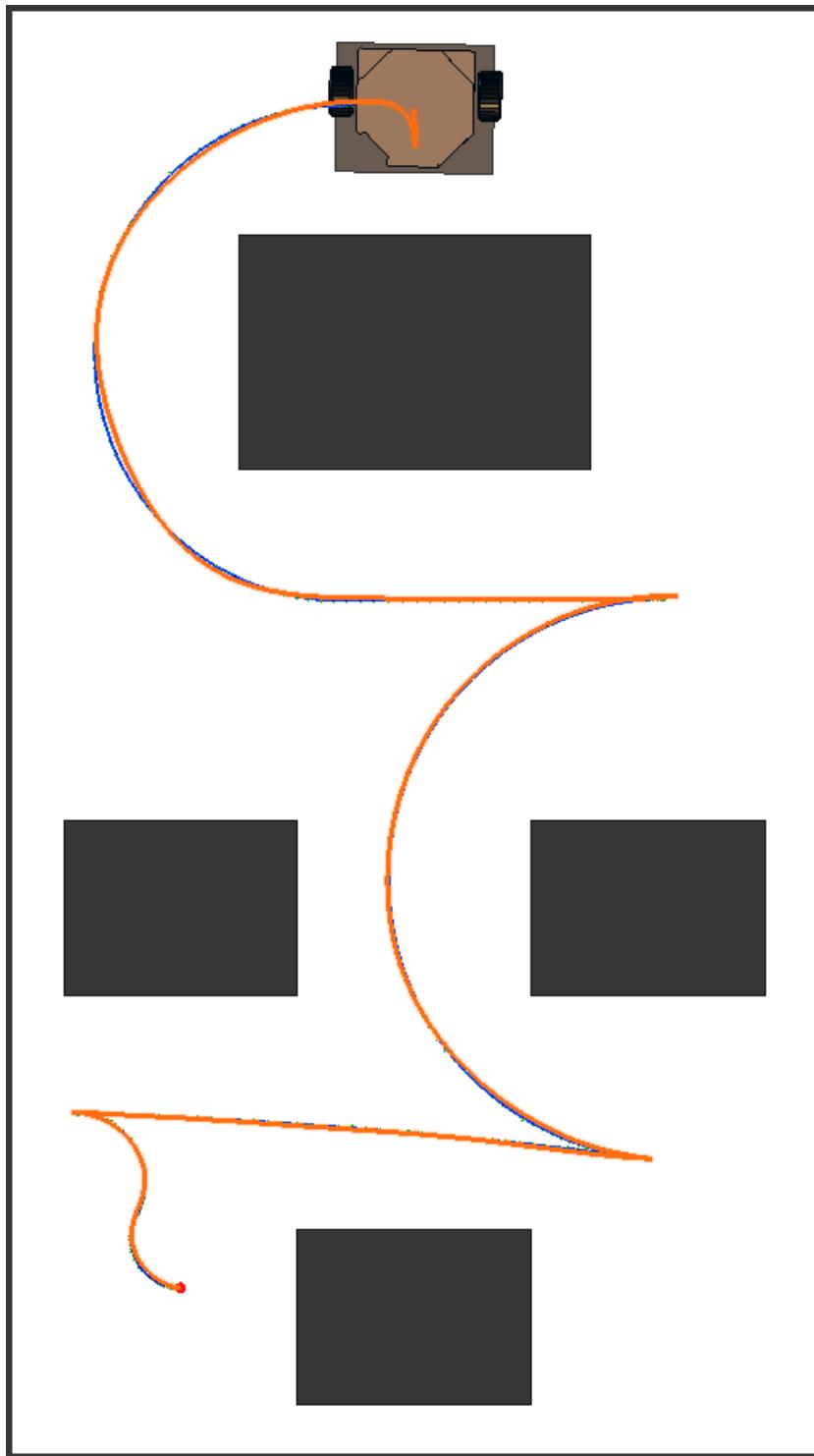
5.10. ábra. Ez a környezet már szerepelt a szimulációs eredmények között is.
Valós robot számára sem okoz problémát a megtervezett pálya lekövetése.



5.11. ábra. Ebben az esetben a C^*CS tervező igen szép pályát tervezett.



5.12. ábra. Erről a pályakövetésről egy videófelvétel is található a dolgozat mellékletében.



5.13. ábra. A pálya megegyezik az előzővel, viszont a robot kétszer akkora sebességgel, gyorsulással hajtotta végre (400 mm/s , 400 mm/s^2).

A pályakövetés valós robot esetén is megfelelően működött, ahogyan a szimulációnál is. Az adott robot esetében, a mérésekhez használt paraméterek mellett biztonságosan végre tudta hajtani a pályákat, a mérés során egyszer sem ütközött akadályba.

Az a véleményem, hogy amennyiben ezzel a robottal nagyobb sebességgel és gyorsulás-sal szeretnénk végrehajtani a megtervezett pályát, elsősorban a robot mechanikáján kellene

változtatni. A robot hajtásrendszere, beleértve a kerekek kialakítását, nem megfelelők nagyobb sebességű pályakövetéshez. Gyakran előfordult, hogy mozgás során a robot kerekei megakadtak, amit a sebességszabályozók ugyan kompenzáltak, de ez ettől függetlenül lengést eredményezett a pályakövetésben. Előfordult, hogy nagyobb gyorsulás esetén a kerék gumiborítása lejött a kerékről. Ezeken a zavaró hatásokon sikerült javítanom a fejlesztés során, de csak a robot hajtásrendszerének módosításával lehetne megközelíteni az 1 m/s sebességhatárt, amelyre a robot motorjai és meghajtása egyébként képes volna.

6. fejezet

Összegzés

Összegzésképpen elmondhatjuk, hogy a diplomatervem célkitűzéseit sikerült elérnem. Megismertem a robotikában alkalmazott pályatervezési megközelítéseket. A pályatervező algoritmusok közül egy módszert részletesen is megvizsgáltam, valamint implementáltam. Kifejlesztettem egy a pályatervezéshez kapcsolódó időparaméterezési, és egy pályakövető eljárást. Megismertem a V-REP robotszimulátor programot, amelyben elkészítettem a differenciális robot modelljét. Az algoritmusok működését szimulátorban sikerült igazolnom. A szimulációnak megfelelő eredményeket kaptam valós roboton történő vizsgálatok során is.

Úgy látom, a jövőben a következő fejlesztésekkel lehetne továbblépni a pályatervezés és mozgásirányítás témaiban. További pályatervező módszerek implementálása, és ezen algoritmusok összehasonlítása szimulátorban végzett mérések alapján. Kinematikai robotmodell lecserélése dinamikai modellre. Időparaméterezsnél további korlátozások figyelembe vétele. Az algoritmusok tesztelése, elsősorban a tapasztalt mechanikai problémák miatt, más differenciális, valós roboton. Keretrendszer fejlesztése, elsősorban a könnyebb használat érdekében. Valós robot irányítószoftverének továbbfejlesztése, a bemutatott algoritmusok alkalmazása az Eurobot 2015 robotversenyen.

Köszönetnyilvánítás

Szeretnék köszönetet mondani **Kiss Domokosnak** a folyamatos konzultációkért, tanácso-kért és iránymutatásokért. Ezenkívül **Csorvási Gábornak** javaslataiért, tanácsaiért, és **Kolumbán Sándornak** a szabályozástechnikában nyújtott segítségéért.

Irodalomjegyzék

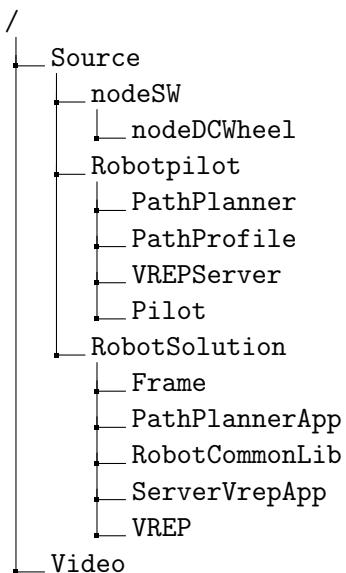
- [1] S. M. LaValle, *Planning Algorithms*. Cambridge, U.K.: Cambridge University Press, 2006. Available online at <http://planning.cs.uiuc.edu/>.
- [2] B. Siciliano and O. Khatib, *Handbook of Robotics*. Springer, 2008.
- [3] D. Kiss and G. Tevesz, „A model predictive navigation approach considering mobile robot shape and dynamics,” *Periodica Polytechnica - Electrical Engineering*, vol. 56, pp. 43–50, 2012.
- [4] Kiss Domokos, *Autonóm robot fejlesztése az Eurobot 2007 versenyre*, Diplomaterv. BME-VIK Automatizálási és Alkalmazott Informatikai Tanszék, 2007.
- [5] L. E. Kavraki, P. Svetska, J.-C. Latombe, and M. H. Overmars, „Probabilistic roadmaps for path planning in high-dimensional configuration spaces,” *IEEE Trans. Robot. Autom.*, vol. 12, pp. 566–580, 1996.
- [6] S. M. LaValle, „Rapidly-exploring random trees: A new tool for path planning,” tech. rep., Computer Science Dept., Iowa State University, 1998.
- [7] B. Kornberger, „Fade2d - An easy to use Delaunay Triangulation Library for C++,” 2014. <http://www.geom.at/fade2d/html/>.
- [8] D. Kiss and G. Tevesz, „The RTR path planner for differential drive robots,” in *Proceedings of the 16th International Workshop on Computer Science and Information Technologies CSIT’2014*, (Sheffield, England), September 2014.
- [9] Katona Gyula Y., Recski András és Szabó Csaba, *A számítástudomány alapjai*. Typotex, 2. javított kiadás ed., 2001.
- [10] C. Sprunk, *Planning Motion Trajectories for Mobile Robots Using Splines*. Albert-Ludwigs-Universitat Freiburg, 2008.
- [11] Dirk-Jan Kroon, „2D Line Curvature and Normals.” <http://www.mathworks.com/matlabcentral/fileexchange/32696-2d-line-curvature-and-normals/content/LineCurvature2D.m>.
- [12] Bézi István, *Robotirányítás rendszerteknikája 3. fejezet*. BME-VIK Automatizálási és Alkalmazott Informatikai Tanszék, 2013.

- [13] Csorvási Gábor, *Pályatervezési és pályakövető szabályozási algoritmusok fejlesztése robotautóhoz*, Diplomaterv. BME-VIK Automatizálási és Alkalmazott Informatikai Tanszék, 2014.
- [14] Coppelia Robotics, „V-REP.” <http://www.coppeliarobotics.com/>.

Függelék

F.1. Megjegyzés a CD melléklettel kapcsolatban

A dolgozat mellékletében megtalálhatóak az általam készített forráskódok, és egy videó-felvétel is a valós robotról. A következő könyvtárak találhatóak a dolgozat mellékletében:



Source:

Forráskódok.

nodeSW:

TMS320F2809 DSP-n futó programkód, az általam fejlesztett node-ot tartalmazza és néhány közös függvényt.

nodeDCWheel:

DCWheel node felel a hajtásért, ezt teljes egészében én fejlesztettem.

Robotpilot:

A robot felső irányítórendszere, teljes programkód. Robo2BME csapattal közösen fejlesztve.

PathPlanner:

Az RTR, C*CS pályatervezők forráskódja.

PathProfile:

Az időparaméterező forráskódja.

VREPServer:

V-REP Server-hez kapcsolódó kliens kódja.

Pilot:

A robot stratégiáját leíró Lua szkriptek.

RobotSolution:

A keretrendszer forráskódja, szimulációhoz elsősorban. Csorvási Gáborral közösen fejlesztve.

Frame:

Példa környezetek.

PathPlannerApp:

Kliens program.

RobotCommonLib:

Több program által használt közös kódok.

ServerVrepApp:

Szerver program.

VREP:

V-REP környezetek.

Video:

Egy adott környezetről készült videófelvétel.