



Budapesti Műszaki és Gazdaságtudományi Egyetem

Villamosmérnöki és Informatikai Kar

Automatizálási és Alkalmazott Informatikai Tanszék

Pályatervezési és pályakövető szabályozási algoritmusok fejlesztése robotautóhoz

DIPLOMATERV

Készítette
Csorvási Gábor

Konzulens
Kiss Domokos

2014. december 15.

Tartalomjegyzék

Kivonat	4
Abstract	6
1. Bevezető	8
1.1. Problémafelvetés	8
1.2. Pályatervezés elmélete	8
1.2.1. Alapvető fogalmak	8
1.2.2. Pályatervezők osztályozása	10
2. Útvonaltervezés C*CS pályákkal	12
2.1. Reeds-Shepp lokális pályák	12
2.2. C*CS lokális pályák	13
2.3. Globális tervező	14
2.4. C*CS approximációs módszer	16
2.4.1. Működés	17
2.4.2. Lokális tervező	17
2.5. $c\bar{c}S$ lokális tervező algoritmus	19
2.5.1. Topológiai feltétel	19
2.5.2. $c\bar{c}S$	20
2.6. Ütközésdetektálás	21
2.7. Eredmények	22
2.7.1. Fejlesztési lehetőségek	23
2.8. Új globális tervező	23
2.8.1. RRT	24
2.8.2. RTR	25
3. Pálya időparaméterevezése	27
3.1. Időparaméterevezés	27
3.2. Jelölések	28
3.3. Korlátozások	29
3.4. Geometriai sebességprofil	32
3.5. Újramintavételezés	34

4. Pályakövető szabályozás	37
4.1. Pályakövetés	37
4.1.1. Sebességszabályozás	37
4.1.2. Referenciapont-választás	39
4.1.3. Túlhaladás problémája	40
4.2. Virtuális vonalkövető szabályozás	40
4.3. Eredmények	43
5. Algoritmusok megvalósítása	45
5.1. Szimuláció – V-REP	45
5.1.1. Szervert program	46
5.1.2. Kliens program	46
5.1.3. Implementáció	47
5.2. Szimulációs eredmények	48
6. Megvalósítás valós roboton	51
6.1. Felépítés	51
6.2. Nehézségek	52
6.2.1. Helymeghatározás	52
6.2.2. Dead-reckoning	53
6.2.3. Kotyogás	53
6.3. Eredmények	54
6.4. Jövőbeli tervezek	55
7. Összegzés	56
Köszönetnyilvánítás	57
Irodalomjegyzék	59
Függelék	60
F.1. Megjegyzés a CD melléklettel kapcsolatban	60

HALLGATÓI NYILATKOZAT

Alulírott *Csorvási Gábor*, 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 15.

Csorvási Gábor
hallgató

Kivonat

A mobil robotok manapság egyre inkább feltörekvőben vannak. Már nem csak az ipar fedezi fel őket, hanem lassan a minden napjai életünk részévé válnak. Azonban még rengeteg elmeleti és gyakorlati kérdés vár megoldásra, hogy az ilyen robotokkal rendszeresen találkozzunk. A mobil robotika egyik legalapvetőbb kérdése az akadályok jelenlétében történő mozgástervezés és mozgásvéghajtás. A dolgozatban ezt a kérdéskört járom körül, foglalkozom a globális és lokális geometriai pályatervezéssel, pályamenti sebességprofil kialakításával, valamint pályakövető szabályozással. Ezeket egy kerekeken guruló robotmodellre alkalmazom, mind szimulált, mind valós környezetben.

A dolgozatban bemutatom a leggyakrabban használt pályatervezési algoritmusokat, és az ezekhez kapcsolódó előnyöket és problémákat. Külön kitérek az általam vizsgált robotmodellnél felmerülő kinematikai korlátozásokra, és ezek hatásaira a pályatervezésben. Egy approximációs pályatervezési megközelítést mutatok be a dolgozatomban, amely egy globális és egy lokális tervező algoritmus együttes használatán alapszik.

Az irodalomban jól ismert, és gyakran alkalmazott módszer a Reeds-Shepp pályák használata. A módszer segítségével az optimális útvonalat határozhatjuk meg autószerű robotok számára. A dolgozatomban bemutatom a C*CS lokális tervezőt, ami alternatívát nyújthat a Reeds-Shepp pályákkal szemben. Az algoritmus körívek (C) és egyenes szakaszok (S) kombinációjából készíti el a megoldást. Bár ez a tervező nem szolgáltat optimális útvonalat, az eredmény sokkal jobban hasonlít egy valós sofőr által végrehajtott pályához. Ehhez szükség van egy globális tervező által előállított pályára. Bemutatok egy egyszerű celladekompozíciós eljárást, amely egyenes szakaszokból álló pályát ad vissza, majd ennek segítségével szemléltetem a C*CS algoritmus működését. Végül egy új globális tervező algoritmust (RTR) is bemutatok, amely szintén egyenes szakaszokból állítja össze a pályát, de számos olyan tulajdonsága van, amely hatékonyabbá teszi a celladekompozíciós eljárásnál.

A megtervezett geometriai pálya még nem tartalmaz információt a mozgás időparamétereire (a robot sebességére, gyorsulására vagy szögsebességére) nézve. Ebből kifolyólag ismertetem azt az algoritmust is, amely a pályamenti sebességprofil meghatározására lett kitalálva. Ez a profil a robot maximális sebessége, maximális gyorsulása és a robot kerekeinek maximális gyorsulása alapján számolható ki. Az így kialakuló pályát ezután újramintavételezzük, hogy időben egyenletes mintavételű pálya álljon rendelkezésre a pályakövető szabályozás számára.

A pályakövető algoritmus a robot pályamenti sebességét és az orientációját egymástól függetlenül szabályozza. A szétcsatolt rendszer sebesség és kormányszög beavatkozó

jelei, a robot kinematikai egyenletei alapján közvetlenül képzik a beavatkozó jeleket. A sebesség-szabályozási kört a robotban található PI szabályozó valósítja meg. Az orientációszabályozás egy mozgás közbeni korrekciót hajt végre, amelynek alapját a robot későbbi előírt pozíciói képezik.

Az algoritmusokat a V-REP robotszimulációs környezetben implementáltam és teszteltem, majd működésüket valós roboton is vizsgáltam. Végül bemutatom a felhasznált autószerű robotot és annak működését, kitérek az elkészítés közben felmerült nehézségekre, és a jövőbeli tervekre is.

Abstract

The research and application of mobile robots is nowadays increasingly widespread. Beyond industrial applications they are getting popular in personal usage as well. However, there are a lot of theoretical and practical issues to be resolved. One of the most fundamental aspects of mobile robotics is motion planning and control in environments populated with obstacles. In this paper we discuss global and local geometric path planning, velocity profile generation and motion control along the path. We simulated the investigated methods and tested with a real car-like robot as well.

In this paper we present the most commonly used path planning algorithms and their benefits and disadvantages. We discuss the kinematic constraints for the tested robot model and the consequences of these constraints. We present an approximation method for path planning, which is based on a global and a local planner algorithm.

The Reeds-Shepp paths are commonly used in mobile robotics, which are the optimal paths for car-like robots. The C*CS algorithm is presented as an alternative to the Reeds-Shepp paths. This method uses arcs (C) and straight segments (S) to provide a path for car-like robots. Although this planner does not provide optimal solution, the resulted paths are rather similar to the ones used by a real driver. To reach this, a global planner method is presented. This planner uses a cell decomposition algorithm which provides a solution with straight segments. After the global path is generated the C*CS local planner is used to approximate the global path. At the end of this section a new global planner method is presented (RTR), which also uses straight segments, but has many qualities which makes it more efficient than the cell decomposition algorithm.

The path generated previously does not contain any information about the robot's velocity, acceleration and angular velocity. Therefore, we present an algorithm developed to determine the velocity profile. The profile is based on parameters such as the robot's maximum velocity, maximum acceleration along the path and the maximum acceleration of the robot's wheels. After velocity profile generation the geometric path needs to be re-sampled to obtain a path having uniform time-sampling for the motion controller.

The path following algorithm controls the robot translational velocity and orientation in two independent control loops. The velocity and steering angle control signals of the decoupled system are passed directly to the robot, based on the kinematic equations of the robot. The velocity control loops are implemented on the robot by a PI controller. The orientation controller is based on orientation error between the current robot configuration and a future path point.

We have implemented and tested the algorithms in V-REP robot simulation framework and with a real robot as well. Finally we present the structure of the used real robot, the difficulties during the development and implementation, and the future plans.

1. fejezet

Bevezető

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 problémát jelentős mértékben megnehezíti, ha a robot környezetében akadályok is találhatóak.

Dolgozatomban bemutatom azon megközelítéseket és módszereket, amelyek megoldást nyújtanak az autonóm pályatervezés kérdésére. Néhány módszert részletesebben is ismertetek, ezeket szimulátoron és valós robotokon is implementáltam, illetve teszteltem. A pályatervezéshez szorosan kapcsoló téma a mozgásirányítás, amivel szintén foglalkoznunk kell, hogy valós környezetben ténylegesen használható eljárásokat kapjunk.

1.2. Pályatervezés elmélete

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

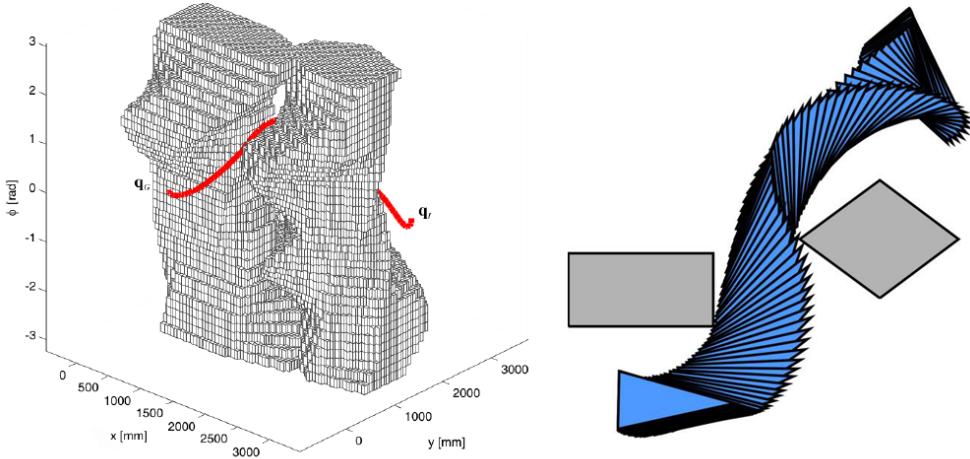
1.2.1. Alapvető fogalmak

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 pozíóját 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}).



1.1. ábra. Konfigurációs tér szemléltetése egy adott útvonal során. A konfigurációs térben a piros vonal a robot útját a célpontja felé [3].

E halmaz komplementere azokat a konfigurációkat tartalmazza, amelyek esetén a robot ütközne az akadályokkal ($C_{obs} = C \setminus C_{free}$). A konfigurációs teret az 1.1. ábrán szemléltetjük.

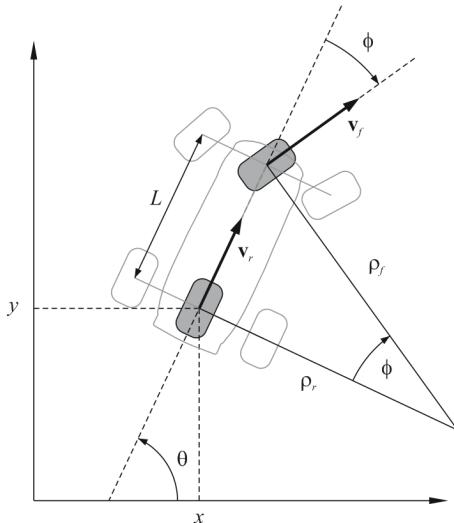
A mozgásirányítás és pályatervezés során kifejezetten fontos ismernünk a korlátozásokat, mivel ezek meghatározzák a követhető pályákat. A környezetben elhelyezkedő akadályokat *globális korlátozásoknak* tekintjük, a robothoz kapcsolódó korlátozásokat pedig *lokális korlátozásoknak* [2]. A lokális korlátozásokat 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). Dolgozatomban csak kinematikai korlátozásokkal foglalkozni, dinamikai korlátokkal nem.

Egy autó esetében differenciális korlát, hogy csak bizonyos íveken tudunk mozogni, egy adott konfigurációból nem tudunk a konfigurációs tér bármely irányába elmozdulni, bár a szabad tér bármely konfigurációjába eljuthatunk. Autónál emiatt nem olyan egyszerű például a párhu zamosan parkolás, vagy az egy helyben való elfordulás. Azokat a robotokat, amelyek ehhez hasonló korlátozásokkal rendelkeznek, *anholonom rendszereknek* nevezzük. Ha a korlátozások olyan differenciálegyenlettel írhatóak le, amelyek nem integrálhatóak, akkor a beszélhetünk anholonom korlátozásról.

Az általam vizsgált robottípus az *autószerű robot*, egy 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

Az általam vizsgált robot típust, az autószerű robot sokak által jól ismert és elterjedtsége megkérőjelezhetetlen. A mozgásegyenletét az 1.2. ábra segítségével könnyedén levezethetjük:



1.2. ábra. Autószerű robot modellje.

$$\begin{aligned}\dot{x} &= v_r \cos \theta \\ \dot{y} &= v_r \sin \theta \\ \dot{\theta} &= \frac{v_r}{L} \tan \phi,\end{aligned}\tag{1.2}$$

ahol L az első és hátsó tengelyek távolsága, ϕ a kormányszög, v pedig a hátsó tengely középpontjának tangenciális sebessége, amelyet a robot referencia pontjának nevezünk.

1.2.2. Pályatervezők osztályozása

Mielőtt belekezdenék az általam megvizsgált pályatervező algoritmusok részletesebb ismeretébe, tekintsük át az irodalomban használatos 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 minimum helyen 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 általam bemutatott algoritmusok is figyelembe veszik 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. Ezen algoritmusok előnye, hogy ha nem létezik megoldás, akkor ezt véges időn belül képesek eldönteni, ám a mintavételes tervezők ezt nem tudják véges időn belül megtenni.

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, de a globális tervező által megtervezett pályát közelíthetjük egy lokális tervezővel is, ha az anholonom robotunk közvetlenül nem tudná lekövetni a globális pályát. Ezt az eljárást, approximációs módszernek nevezik. Egy ilyen algoritmusra látunk példát a következő fejezetben.

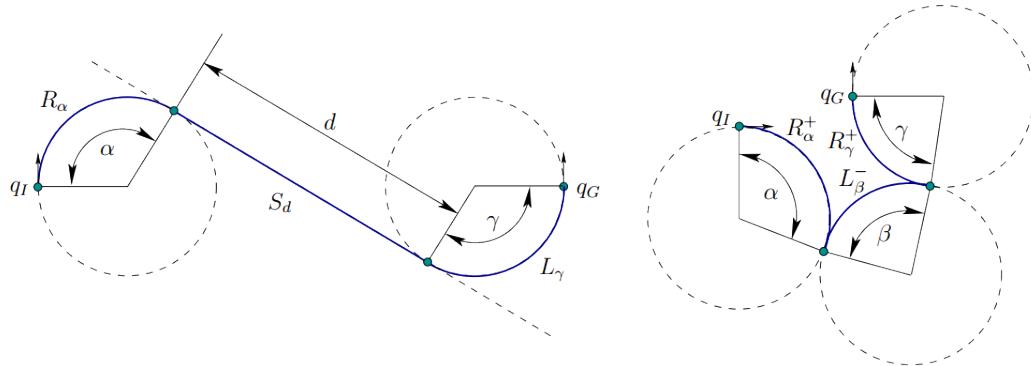
2. fejezet

Útvonaltervezés C*CS pályákkal

A C*CS és a $c\bar{c}S$ algoritmus Kiss Domokos munkája [4]. Az algoritmusok elsődlegesen autószerű robotok számára terveznek pályát, de az így tervezett pálya egy differenciális robot számára is végrehajtható. Feladatom az algoritmus implementálása volt C++ nyelven, majd annak tesztelése szimulációs, illetve valós környezetben. A fejezetet az algoritmus ismertetésével kezdem, majd kitérek az implementációs problémákra és az elért eredményekre is.

2.1. Reeds-Shepp lokális pályák

Az anholonom rendszerek irányítása akadályuktól mentes környezetben is egy igen bonyolult feladat. Sok esetben nem adható meg általános algoritmus, csak néhány speciális rendszer esetén. Szerencsére ilyen rendszerek közé tartoznak a differenciális robotok, az autószerű robotok, amelyek csak előre mozoghatnak (Dubins autó), és azok, amelyek előre és hátra is képesek mozogni.



2.1. ábra. Dubins és Reeds-Shepp megoldások [1]

Az utóbbi típusú robotokat hívjuk Reeds-Shepp autóknak, melyeknél bizonyított, hogy bármely kezdő- és célkonfiguráció között a legrövidebb utat megtalálhatjuk 48 lehetséges megoldás közül, amelyből kettő látható a 2.1 ábrán. Ezek a lehetséges megoldások maximum öt egyenes vagy minimális sugarú körív kombinációjából állhatnak, és a pályák maximum

két csúcsot tartalmazhatnak, azaz ennyiszer lehet irányt változtatni a végrehajtás közben [5].

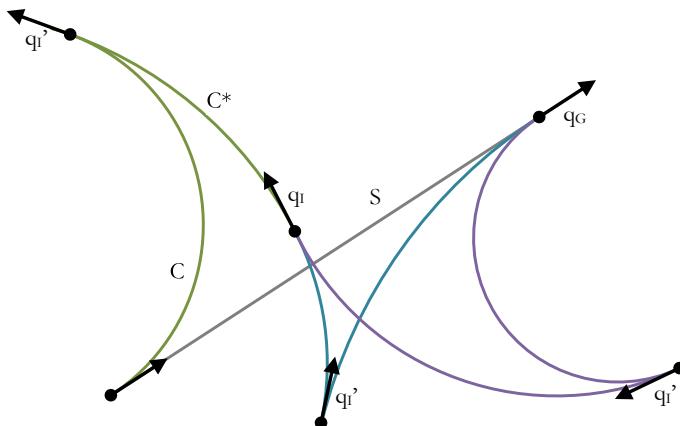
Mint látható, akadályuktól mentes környezetben találhatunk optimális útvonalat, de ennek hátránya, hogy minden minimális sugarú pályákat feltételez, mely egy valós esetben nem életszerű, illetve a pályák lehetnek igen bonyolultak is. Azonban ha elvetjük az optimalitás igényét – amit egyébként is meg kell tennünk, ha egy globális tervező részeként alkalmazzuk a módszert – akkor a lehetséges megoldásokon jelentős mértékben egyszerűsíthetünk.

2.2. C*CS lokális pályák

A lokális tervezők bármely kezdő- és célkonfiguráció páros esetén megoldást kell nyújtanak, de megfelelő koordináta-rendszer választásával egyszerűsíthetünk a számításokon. Tegyük fel, hogy egy ilyen választás mellett adódott $q_I = (x_I, y_I, \theta_I)$ kezdő és $q_G = (0, 0, 0)$ célkonfiguráció. Ha eltekintünk a minimális fordulási sugár korlátozásától, és feltesszük, hogy $\theta_I \neq 0$, akkor könnyen belátható, hogy egy kör és egy egyenes segítségével elérhető a célkonfiguráció. Először egy érintő körön elfordulunk a $\tilde{q}_G = (\tilde{x}_G, 0, 0)$ köztes célkonfigurációba, majd egy egyenes mentén végighaladunk a célig. Az ehhez tartozó kör sugarát a következő egyenlet segítségével számíthatjuk:

$$\rho_{I,\tilde{G}} = \frac{y_I}{1 - \cos \theta_I} \quad (2.1)$$

Ha a kiadódó sugár kisebb, mint a minimálisan megengedett $|\rho_{I,\tilde{G}}| < \rho_{min}$, esetleg $\theta_I = 0$, akkor egy egyenes vagy egy kör segítségével egy köztes kezdőkonfigurációba ($\tilde{q}_I = (\tilde{x}_I, \tilde{y}_I, \tilde{\theta}_I)$) kell eljutnunk, ahol biztosított, hogy $\tilde{\theta}_I \neq 0$ és, hogy $\rho_{\tilde{I},\tilde{G}} \geq \rho_{min}$. Megjegyzendő, hogy az első szakasz nem lehet egyenes, ha $\theta_I = 0$, $\theta_I = \pi$ vagy $|y_I| < 2\rho_{min}$.



2.2. ábra. A köztes konfiguráció megválasztása (q'_I)

Bebizonyítható [4], hogy \tilde{q}_I köztes konfigurációt végtelen sokféleképpen megválaszthatjuk. A 2.2. ábrán látható erre egy példa. A q_I kezdeti konfiguráció túl közel található a q_G

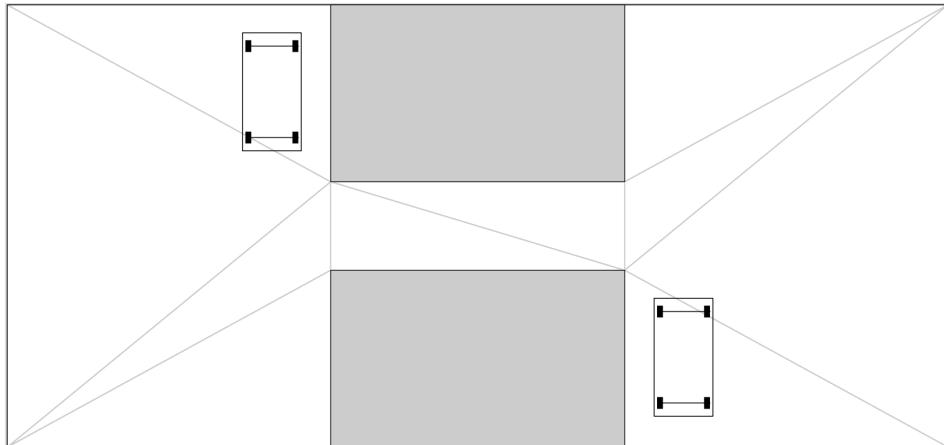
egyeneséhez, ezért előbb egy köríven (C^*) el kell mozdulnunk egy köztes konfigurációba. Ezt megtehetjük előrehaladással vagy tolatással is.

Hogy egyszerűsítsük a jelöléseket, a továbbiakban az egyenes szakaszokra S , a körívekre pedig a C betűk segítségével hivatkozunk. Ezt felhasználva belátható, hogy a célpontba egy SCS , vagy egy CCS pálya segítségével eljuthatunk. Könnyen belátható, hogy ha egy körív (C) sugarával a végtelenbe tartunk, akkor a szakasz az egyeneshez tart. Az olyan speciális köríveket, amelyek sugara végtelen is lehet, C^* -gal jelöljük. Innen a módszer neve, a C^*CS .

2.3. Globális tervező

Ugyan bármilyen globális tervező által nyújtott pályát alapul vehetünk, a C^*CS lokális tervező alapötlete, hogy körívek segítségével egy egyenes szakaszra forduljunk rá. Így sokkal természetesebb megoldásokat eredményez olyan globális pályák használata, melyek egyenes szakaszokból állítják elő az eredményt.

Erre a célra egy celladekompozíció alapuló algoritmust használtam, melynek lényege, hogy a szabad környezetet cellákra osztja. Az egyik legegyszerűbben használható és legtermészetesebb ilyen cella a háromszög. Sokszögek háromszög-felbontására több algoritmus is létezik, én a Fade2D algoritmusát használtam [6]. Az elkészült felbontásra láthatunk egy példát a 2.3. ábrán.



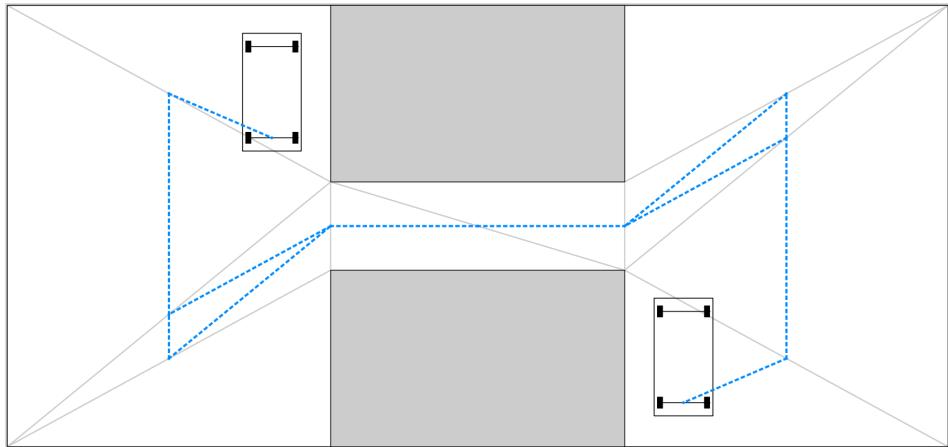
2.3. ábra. Környezet háromszögekre bontása

Miután a felbontás elkészült, egy gráfot alkotunk a következőképpen: Kiválogatjuk azokat a háromszögoldalakat, amelyek nem akadályhoz vagy a környezet határához tartoznak. Ezeknek az oldalaknak felezőpontjai fogják a gráf csomópontjait alkotni. A gráfban két olyan pont között veszünk fel élt, ahol azok egy háromszöghöz tartoznak, ezzel biztosítva, hogy ezek a szakaszok ne metssék egymást. A pontok közé beszúrjuk a kezdő és célkonfiguráció pontjait, meghatározzuk, hogy ezek melyik cellában helyezkednek el, és összekötjük az ahhoz a cellához tartozó csomópontokkal.

Az így elkészített gráfban még vannak olyan éllek, amiket a végső megoldásban nem alkalmazhatunk, mert vagy egy akadályhoz, vagy a pálya széléhez túl közel találhatóak. Hogy ezeket kiszűrjük, végig kell nézzük, hogy ezeken az autóval végig tudunk-e haladni

ütközés nélkül. Az ütközésdetektálásról még később szót ejtek. Fontos megemlíteni, hogy az ellenőrzés során csak az autó szélességét vesszük figyelembe, mivel az a célunk, hogy ezeken a szakaszokon egyenesen haladjon az autó.

A már végleges gráfban (2.4. ábra) minden élhez a hozzá tartozó szakasz hosszát hozzárendeljük, ezek fogják jelenteni az él súlyát. Végül a gráfban egy keresőalgoritmus segítségével kiválasztjuk a legrövidebb utat. Ha a gráf nem összefüggő, az algoritmus hibával tér vissza, mivel a feladat nem megoldható. A kereséshez én a Dijkstra-algoritmust használtam [7], de bármilyen más algoritmust használhatnánk, mivel a fentebb látott módszerrel alkotott gráf jellemzően nem lesz nagy méretű, ezért nem lesz jelentős különbség a futási idők között.



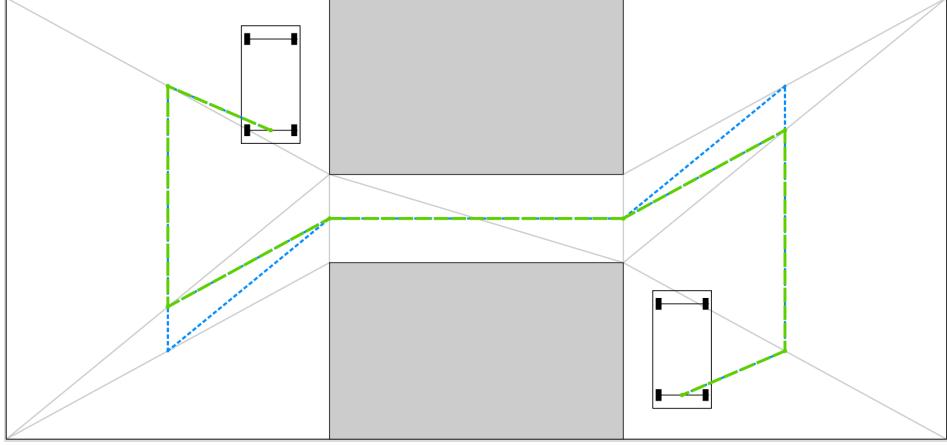
2.4. ábra. A kialakított gráf

A végleges megoldáshoz (2.5. ábra) nekünk nem csak pontokra, hanem konfigurációkra van szükségünk, ezért a legrövidebb út csomópontjaihoz még orientációt kell rendeljünk. Ezt a következőképpen tesszük meg: Az kezdeti és célkonfiguráció pontjainál az előre meghatározott orientációt alkalmazzuk. Köztes csomópontok esetén az aktuális és a következő pont által meghatározott egyenes és x-tengellyel bezárt szög adja az orientációt. Ezt a (2.2) alapján számolhatjuk ki:

$$\theta_i = \arctan \left(\frac{y_{i+1} - y_i}{x_{i+1} - x_i} \right), \quad (2.2)$$

ahol x_i és y_i az aktuális csomópont megfelelő koordinátái.

A megoldásnak az előnye, hogy nagyjából a szabad terület közepén alkot pályát, így ha az autó ezt követi, akkor bármilyen irányú manőverezésre lesz lehetősége. Továbbá ez egy úgynevezett kombinatorikus eljárás, így véges időn belül képes megmondani, hogy létezik-e megoldás. Hátránya, hogy a háromszögelés miatt csak sokszögekkel leírható akadályokkal képes dolgozni



2.5. ábra. Az elkészült globális pálya

2.4. C*CS approximációs módszer

Az így elkészült globális pályát természetesen a töréspontok miatt nem tudnánk lekövetni egy autóval, ezért közelítenünk kell olyan pályákkal, amelyek már nem okoznak gondot az autó számára sem. Ezt a C*CS approximációs módszer segítségével tesszük meg. A módszer a nevét a benne alkalmazott lokális tervezőről kapta, a működését az **Algoritmus 1** mutatja be.

Algoritmus 1 C*CS approximáció

```

1:  $Q := [q_{Q,1} \cdots q_{Q,m}] = GlobalisTervezo(\mathcal{C}_{obs}, q_I, q_G)$ 
2:  $\lambda_{final} := 0, \quad i := 1, \quad j := m$ 
3:  $arm := ARM(\mathcal{C}_{obs}, q_I)$ 
4: while  $(j - i) <> 1$  do
5:    $q_{IL} := q_{Q,i}, \quad q_{GL} := q_{Q,j}$ 
6:   if  $(\lambda_{local} := C^*CS(\mathcal{C}_{obs}, arm, q_{IL}, q_{GL})) <> 0$  then
7:     if  $q_{GL} = q_G$  then
8:        $\lambda_{final} := Osszefuz(\lambda_{final}, \lambda_{local})$ 
9:       return  $\lambda_{final}$ 
10:    else
11:       $\lambda_{final} := Osszefuz(\lambda_{final}, \lambda_{local}^*)$ 
12:       $q_{Q,j} := q'_{GL}$ 
13:       $i := j, \quad j := m$ 
14:       $arm := ARM(\mathcal{C}_{obs}, q_{Q,i})$ 
15:    end if
16:    else
17:       $j := \lceil \frac{j-i}{2} \rceil + i$ 
18:    end if
19: end while
20: return HIBA

```

Mielőtt bemutatnám a működését, érdemes szót ejteni a jelölések ről. A globális tervező által elkészített konfigurációs listát Q jelöli, ezen belül az i . konfigurációt $q_{Q,i}$. A környezet leírására a \mathcal{C}_{obs} jelölést használtam. λ jelöli az elkészült pályát, ahol alsó indexben a $final$ a teljes, míg a $local$ jelző a lokális tervező által készített pályára utal. A q_I és q_G a globálisan

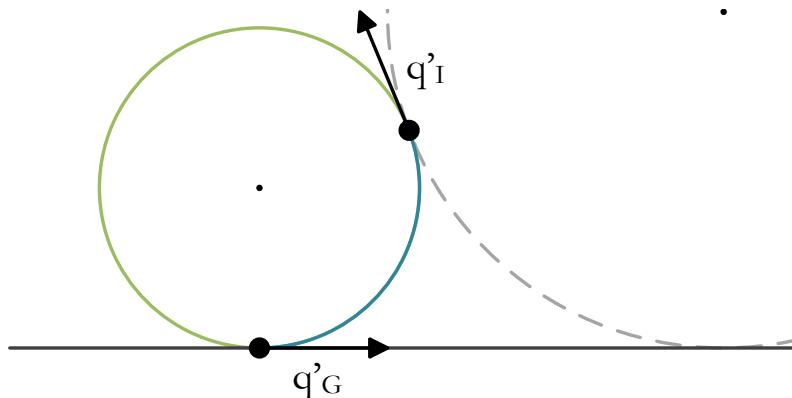
meghatározott kezdő- és célkonfiguráció, míg a q_{IL} és q_{GL} a lokálisan közelíteni kívánt pályarészlethez tartozik.

2.4.1. Működés

Mint látható, a globális pályát iterálva próbáljuk meg C*CS pályákkal közelíteni, az első lépésben rögtön a kezdő és célkonfigurációval kezdve. Ha ez nem sikerül, a 17. sorban látott módon a pályát megfelezzük, és a lokális célkonfigurációnak a középső töréspontot választjuk. Ezt egészen addig folytatjuk, míg nem találunk megoldást, vagy el nem fogynak az újabb köztes töréspontok. Az előnye ennek a módszernek, hogy szerencsés esetben akár rögtön az első iterációban is találhatunk pályát, ami jelentősen felgyorsíthatja a működést.

Egyik fontos tulajdonsága az algoritmusnak, hogy nem kényszeríti az elkészült pályát a globális sarokpontokba, mivel nem igazán fontos számunkra, hogy elérjük őket¹. Elég csak a globális pályaszakaszokhoz „tapadni”, ezért a megtalált pályából ilyenkor elhagyjuk az egyenes szakaszt, és csak a C^*C részletet fűzzük hozzá a végleges pályához. Az újabb szakasz keresését az érintési pontból indítjuk, így ezt be kell szúrjuk a listába.

Egyetlen kivétel ez alól, mikor a q_{GL} lokális célkonfiguráció, az maga a globális célpont (q_G). Ebben az esetben az utolsó szakaszt mindenkor hozzá kell fűzzük a végleges pályához, hogy elérjük a célpontot².



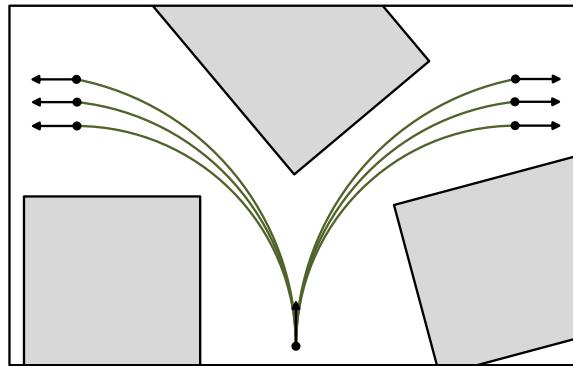
2.6. ábra. Középső körív számítása érintő körrel

2.4.2. Lokális tervező

Az előzőekben láthattuk, hogy a C*CS végtelen sok megoldást nyújt. Lokális esetben ez nem feltétlen hasznos, de akadályok jelenlétében már igen, mivel így sokkal nagyobb valósínűséggel találhatunk végrehajtható pályát. Természetesen az összes megoldást nincs

¹Nem is minden előnyös az utolsó egyenes szakasz használata, gondoljuk például arra az esetre, mikor az érintési pont a célkonfiguráció után található, ebben az esetben felesleges tolatások jelennének meg a pályában

²Fontos megemlíteni, hogy a lokális tervező minden elvégzi a számításokat a teljes C^*CS pályára (üt-közésetektálás, legrövidebb út keresés), az utolsó S szakasz később kerül kitörlésre.

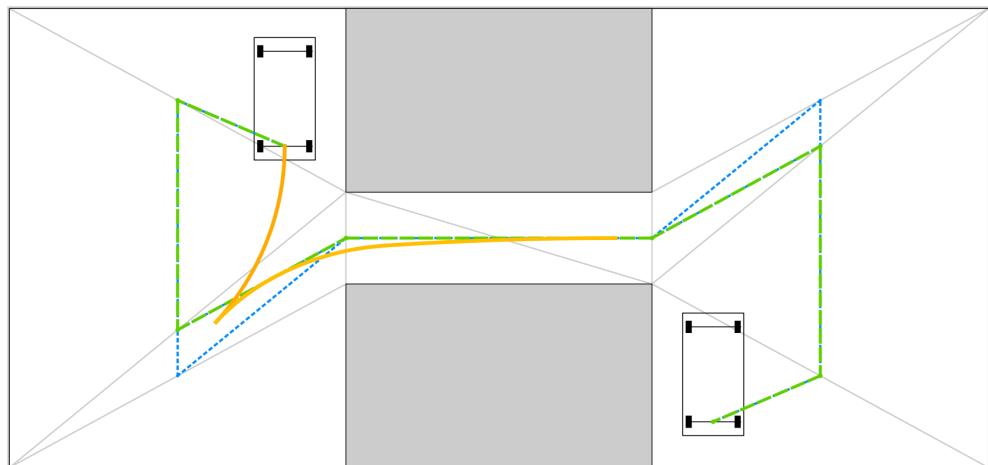


2.7. ábra. Példa a körívekkel elérhető sokaságra (ARM)

lehetőségünk kipróbálni, így \tilde{q}_I -t valamelyen mintavételező eljárással kell kiválasztanunk. Sokféle mintavételezési megoldást fel lehetne használni, de az egyszerűség kedvéért én egy egyszerű fixtávolságú mintavételezést használtam.

A lokális tervező első lépésben összegyűjti azokat a köríveket, amelyeket a robot megadott kezdőkonfigurációból ütközésmentesen el tud érni. Ezeknek a köríveknek a sokaságát *Arc Reachable Manifold*-nak (ARM) nevezzük [8]. Az összes ilyen \tilde{q}_I -ból az algoritmus megkeresi a célkonfigurációhoz tartozó érintő körívet (2.6. ábra), és az egyenes szakaszát. Azokat a szegmenteket, amelyek ütközésmentesen végrehajthatóak, összegyűjti egy listába, majd ebből a listából kikeresi azt, amelyik a legrövidebb az összes közül.

Itt érdemes megemlíteni, hogy a végeredmény akkor fog igazán hasonlítani a valósághoz, ha lecsökkentjük a tolatások számát, mivel az emberek nagy többsége nem szeret tolatva közlekedni. Ahhoz, hogy ezt megtegyük, az algoritmus figyelembe vesz egy súlytényezőt, mellyel a tolató szakaszok „hosszát” tudjuk megnövelni.



2.8. ábra. A *C*CS* algoritmus működés közben, második iterációra található lokális pálya.

2.5. $c\bar{c}S$ lokális tervező algoritmus

Az approximációs algoritmus nem teljes ebben a formájában, mivel ha nem talál újabb töréspontot, akkor hibával tér vissza. Megtehetnénk, hogy az egyenes szakaszokat feldaraboljuk, de ezzel nem nyernék új megoldásokat, mivel minden esetben ugyanarra az S szakaszra próbálnánk érintőt találni. Ezért az új konfigurációkat a töréspontokban kell elhelyezni. Viszont önmagában ez még nem elég, valahogy biztosítanunk kell azt, hogy az algoritmus konvergáljon a megoldás felé.

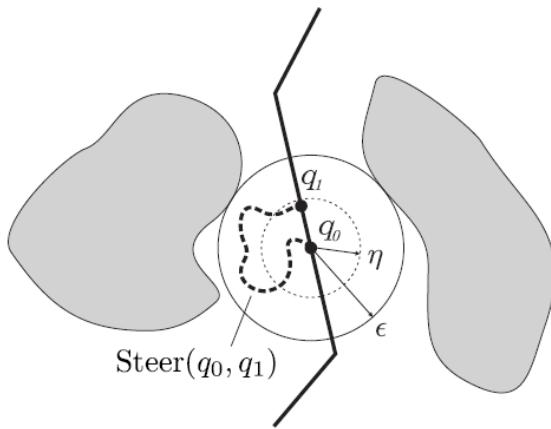
2.5.1. Topológiai feltétel

Olyan lokális tervezőre van szükségünk, mely teljesíti az úgynevezett topológiai feltételt. Ha ezt biztosítani tudjuk, akkor az approximációs algoritmusunk teljes lesz. A teljesség itt azt jelenti, hogy az algoritmus minden olyan esetben megoldással tér vissza, mikor a globális tervező érvényes pályát szolgáltatott. Tehát a körívekkel való közelítés nem csökkenti a megoldás létezésének esélyét.

Egy lokális tervező (szakirodalomban *steering method*) $\text{Steer}(q_0, q_1)$ teljesíti a topológiai feltételt, ha:

$$\begin{aligned} \forall \epsilon > 0, \quad \exists \eta > 0, \quad \forall q_0, q_1 \in \mathcal{C} \\ d_{\mathcal{C}}(q_0, q_1) < \eta \Rightarrow d_{\mathcal{C}}(q_0, \text{Steer}(q_0, q_1)(\sigma)) < \epsilon, \\ \forall \sigma \in [q, S], \end{aligned} \tag{2.3}$$

ahol $d_{\mathcal{C}}$ egy bármilyen metrika \mathcal{C} konfigurációs téren definiálva.



2.9. ábra. Topológiai feltétel illusztrációja $\mathcal{C} = \mathbb{R}^2$ esetén [4]

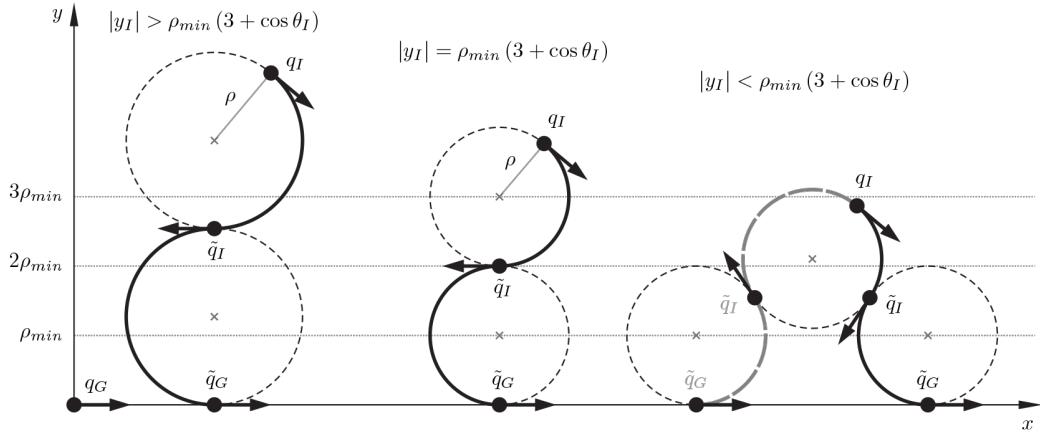
Ez azt jelenti, hogy ha bármilyen pozitív ϵ -hoz található olyan η , hogy q_1 a $q_0 - \eta$ környezetében van, akkor a lokális tervező által generált pálya nem lépi ki a $q_0 - \epsilon$ környezetét. Erre egy segítő példa a 2.9. ábrán látható.

Más szavakkal tehát, ha a globális pályát közelítjük a q_0 konfigurációs pontból, amelynek van egy ϵ sugarú, ütközésmentes környezete, akkor találhatunk olyan q_1 konfigurációt,

amelyre a lokális tervezőnk ütközésmentes és végrehajtható pályát eredményez. Ez azt is magával vonja, hogy ha a globális pálya mellett van egy kis szabad terület, ahol manőverezésre van lehetőség, akkor a lokális tervező talál megoldást.

2.5.2. $c\bar{c}S$

Bizonyított, hogy a $c\bar{c}S$ algoritmus teljesíti ezt a feltételt [4], tehát felhasználható a konfigurációk közelítésekor. Ezek alapján az approximációs algoritmust kiegészíthetjük a következőkkel: Ha nincs új sarokpont a globális pályán, akkor egy egyenes szakasz segítségével elmegyünk a következő sarokpontig. Ezt megtehetjük, mivel ez biztosan ütközésmentes, csak elhagytuk a pályából. Majd a globális pályába beszúrunk egy konfigurációt, ahol a pont a következő sarokpont, és az orientáció az aktuális sarokponthoz tartozó szög. Végül ezek között keresünk megoldást, de már az új – topológiai feltételt biztosító – lokális tervezővel. Ha ezek között sincs végrehajtható pálya, akkor a konfigurációs térben (\mathcal{C}) távolságot csökkentenünk kell. Ezt az orientációkülönbség megfelezésével megtehetjük.



2.10. ábra. A $c\bar{c}S$ algoritmus megoldásai különböző y_I esetén [4]

A $c\bar{c}S$ algoritmusra a C^*CS algoritmus egy módosított változata, mely csak egy megoldást ad egy konfiguráció párra. Az eljárás lényege, hogy az első két kör sugara minimális és megegyező, de ellentétes előjelű, tehát a másik irányba kell forgassuk a kormányt. A komplementer jelölés jelzi az előjel változását. Ahogy a 2.10 ábrán látható, ha q_I túl közel van a q_G egyeneséhez, akkor a körök egymás mellett elcsúsznak, és két megoldást is adnak.

Bár a $c\bar{c}S$ definíció szerint egy megoldást ad, a végrehajtásakor több pályát is „eldob”. A módszer implementációját úgy készítettem el, hogy minden ilyen megoldást ellenőrizzen, ha esetleg a legrövidebb nem lenne végrehajtható, akkor válasszon másikat. Jogosan felmerülhet a kérdés, hogy ha ez az algoritmus minden esetben nyújt megoldást, akkor miért nem ezt használjuk a C^*CS helyett? Bár valóban igaz, a $c\bar{c}S$ minden megoldás elérhető, a C^*CS több lehetséges megoldás közül választ, így a gyakorlatban természetesebb pályákat ad eredményül.

2.6. Ütközésdetektálás

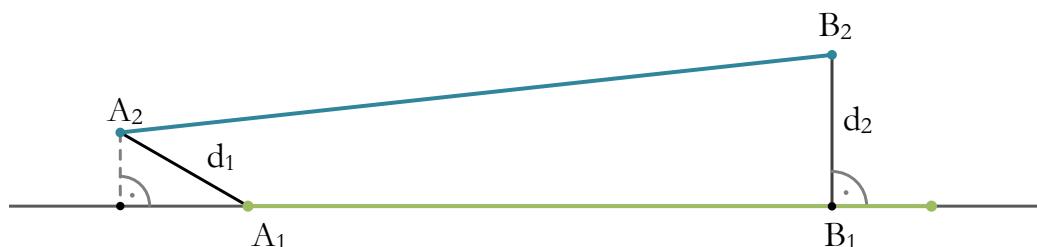
Akadályok jelenlétében nem kerülhető el a tervezett pálya végrehajthatóságának vizsgálata. Azért is fontos ezt megemlíteni, mert a tervezés során ennek a számítása az egyik leginkább időigényes folyamat. Általános esetben ez a probléma igen bonyolult, és még inkább időigényes folyamattá is válhat.

Abban az esetben, ha mind az akadályok, mind a robot teste sokszögekkel leírható, de a tervezett pályával kapcsolatban nincsen semmilyen megkötésünk, csak a mintavételezésre hagyatkozhatunk. Az elkészült pályát fel kell osztani több pontra, és ezeken a helyeken ellenőrizni, hogy a robot teste nem ér-e ki a pályáról vagy lóg-e bele az egyik akadályba. Ilyenkor különös odafigyeléssel kell megválasztanunk a mintavételező eljárást, mivel ha túl távoli pontokat ellenőrzünk, lehet, hogy kihagyunk olyan pontokat, amelyek ütköznének. Túlságosan sűrű mintavételezzel pedig az eljárás működése lelassul.

Az általam használt algoritmusoknál nincs szükség mintavételezésre, mivel az elkészült pályák (körívek, és egyenes szakaszok) zárt alakban felírhatók. Az ütközésdetektáláshoz a robot testét a meghatározott pályán kell végigmozgatni, és ellenőrizni, hogy egyik pontjának pályája se keresztezi bármelyik akadály élét³.

Egyenes szakaszok

Egyenes szakaszok ütközésdetektálását két különböző módon oldottam meg. A globális tervező esetén csak az akadályok és a lehetséges pályaszakaszok távolságát számítja ki az algoritmus. Először is ellenőri, hogy a két szakasz metszi-e egymást. Ha igen, a távolságuk nulla. Ha nem, akkor minden szakasz két végpontját levetíti az ellenőrizni kívánt szakaszra. Ha ez a vetületi pont a szakaszon belül található, akkor ennek és a végpontnak a távolságát vessziük eredményül, ha kívülre esik, akkor a két végpont távolsága az eredmény (2.11.). Ha a szakaszok távolsága kisebb mint a robot maximális szélességének a fele, akkor az élt töröljük.



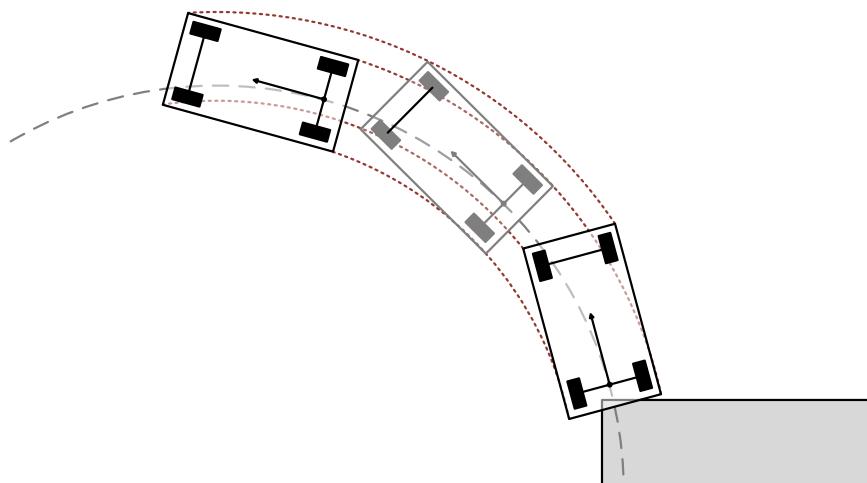
2.11. ábra. Szakaszok távolságának meghatározása

Természetesen ez nem vezet pontos eredményre, de a 2.3 fejezetben ezt a hibát figyelmen kívül hagyjuk. A másik, pontosabb módszert detektáláshoz a körívek esetén mutatom be.

³Ebben az esetben a pálya széle is akadályként tekinthető.

Körívek

Mivel kikötöttük, hogy a robotunk teste is sokszögekkel leírható, az ellenőrzéséhez elegendő a csúcsokat elmozdítani a pályával párhuzamosan. Ezt mind egyenes szakaszok, minden körív esetén hasonlóan kell megtegyük. Kiválasztunk egy csúcsot, majd egy koordinátatranszformáció segítségével a pálya végpontjait eltoljuk a megfelelő helyre, azaz kiszámoljuk, hogy a két végpontban a robot azon pontja hol is lesz pontosan. Egyenes szakaszoknál ez a két pont meghatározza az új szakaszt, körív esetén a két új pont és a pálya körének középpontja segítségével kiszámítjuk az új körívet. Amint ezzel készen vagyunk, metszéspontot keresünk az áttranszformált pálya, és az akadályélek között. Ez körív és egyenes esetén egy kicsit komplikáltabb feladat, mivel itt a megoldást egy másodfokú egyenlet határozza meg.



2.12. ábra. Hibás ütközésdetektálás, nem elég csak a robot pontjait mozgatni

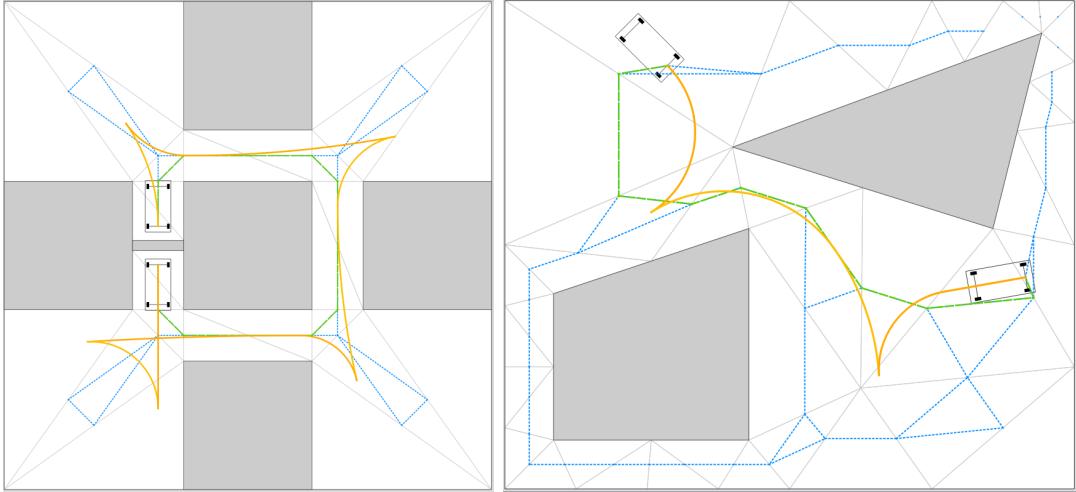
Ha ezt megtettük, még nem vagyok készen, le kell ellenőriznünk a fordított esetet is, amikor az akadályok csúcsait mozgatjuk a pályán, de ellentétes irányban. Erre azért van szükség, mert elképzelhető olyan helyzet, hogy a robot sarokpontjai ugyan nem metszenek egyetlen akadályelt se, mégis ütközünk akadállyal. Erre láthatunk egy példát a 2.12. ábrán.

Ezt úgy tehetjük meg, hogy kiszámítjuk a pálya végpontjából az akadály egyik csúcsába mutató vektort, majd ezt használjuk fel a kezdőpont transzformálásához. Az így keletkezett pontot és a csúcspontot a pályának megfelelő szakasszal összekötjük, és ellenőrizzük, hogy a robot testének éleivel van-e metszéspontja. Ha egyik esetben se volt metszet, akkor a pálya végrehajtható.

2.7. Eredmények

A feladatom megvalósításához rendelkezésre állt a C*CS algoritmus egy MATLAB scriptben megírt változata, ellenben ez csak demonstrációs célokra szolgált, a feladatot lassan hajtotta végre, valószínűleg az interpretált működés következtében, ezért vált szükségesé egy C++ implementáció.

Az így készült programmal nagyságrendileg százszoros gyorsulást sikerült elérnem, és a legbonyolultabb környezetben is egy másodpercen belül sikerült megoldást találnia az algoritmusnak⁴. Ez igen nagy előrelépés, így valószínű, hogy egy kisebb teljesítményű beágyazott számítógépen is elfogadható időn belül végez.



2.13. ábra. A *C*CS* algoritmus megoldása különféle környezetekben

2.7.1. Fejlesztési lehetőségek

A fejlesztés során odafigyeltem, hogy hol lehetne gyorsítani, módosítani a működésen. Ahol ez egyszerűen megvalósítható volt, ott ezeket megtettem, de maradtak további fejlesztési lehetőségek is a programban, például számos helyen lehetne a futást párhuzamosítani.

A tapasztalatok azt mutatták, hogy az algoritmus futásának jelentős hányada az ARM számítása. Mivel a körívek kiszámítása, és azok ütközésdetektálása komplex művelet, így ez az eredmény nem meglepő. A kiszámítás ideje természetesen függ a választott távolság-egységtől és a környezet méretétől. A számítást fel lehet gyorsítani, ha a mintavételeznél használt távolságot megnöveljük, de ezzel lehetséges megoldásokat hagyhatunk el.

Másik javítási lehetőség, ha előre elkészítünk egy foglaltsági mátrixot, ami megmondja az adott pont akadályon beül van-e, így ezekre a pontokra nem kell a számítást elvégezni. Mivel az így kapott körívek egy adott kezdőkonfigurációhoz tartoznak, további gyorsításra ad lehetőséget, ha az approximációs lépésekben inkább a célkonfiguráció pontját mozgatjuk, így nem kell újra és újra kiszámolni az ARM-ot. Az **Algoritmus 1** esetén is ez látható. A harmadik sorban rögtön a kezdő konfigurációra elvégezzük a számítást, és csak akkor ismételjük meg újra, ha a kezdőpontunk is megváltozott (14. sor).

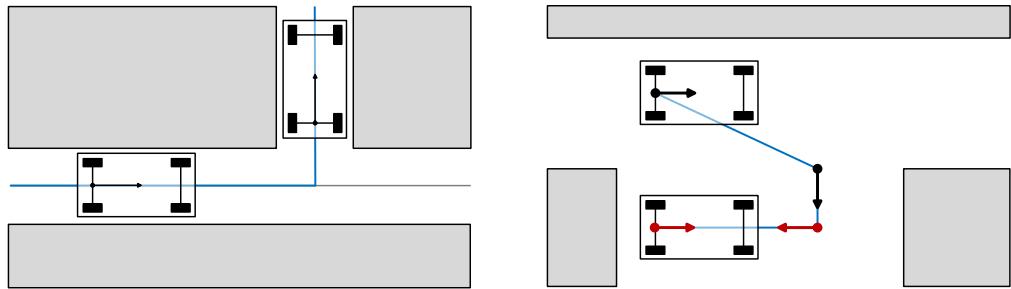
2.8. Új globális tervező

Az esetek nagy többségében a 2.3 fejezetben használt celladekompozíciós eljárás jó eredménnyel szolgál, de több problémával is rendelkezik. Például könnyen észrevehető, hogy az

⁴Intel Core 2 Duo E8400 @ 3.0GHz, 4GB RAM

egyszerű ütközésdetektálásnál még maradhatnak olyan élek a gráfban, melyek nem minden esetben végrehajthatóak.

További hibát okoz, hogy nem ellenőrzi az eljárás, hogy van-e hely manőverezésre, pedig ez a topológiai feltételnél látható, hogy ez a végrehajtható pálya megtalálásának feltétele. Ez a probléma jelentkezik két merőleges folyosónál is. A felhasznált szakaszok távolsága hiába nagyobb mindkét, mint az autó szélessége, nincs hely a megfordulásra.



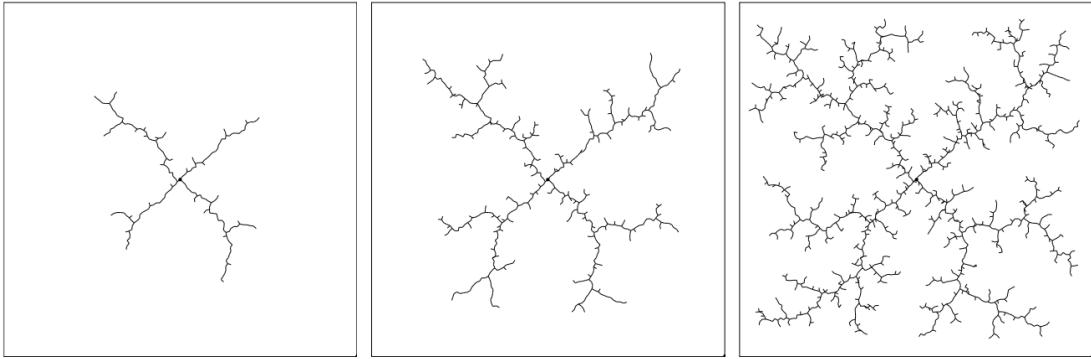
2.14. ábra. *Hibás globális pályák. A bal oldalon merőleges folyosók problémája, ahol nincs lehetőség elfordulni. Jobb oldalon a párhuzamos parkolás problémája, a piros konfigurációk okozzák a bonyolult pályát.*

Egy másik problémás tulajdonsága az orientációk meghatározásakor jelentkezik. Mivel a kiindulási és célkonfiguráció esetén az előre meghatározott orientációkat használjuk, a köztes sarokpontoknál pedig a pálya segítségével határozzuk meg, így kialakulhatnak ellentmondásos (egymással szembe mutató) konfigurációk. Ez jelentkezik párhuzamos parkolás esetén is. Ugyan ilyen esetben található megoldás, de a keletkezett útvonal túl bonyolult lesz, így a valóságban ez használhatatlanná válik.

A fejlesztés során a fentebb felsorolt problémák miatt a globális tervezőt lecseréltem az RTR nevű algoritmusra. Ez is Kiss Domokos munkája [9], az implementációját Nagy Ákos végezte el [10]. Ez egy, a szakirodalomban széles körben használt RRT (Rapidly Exploring Random Trees) módszeren alapuló pályatervezési eljárás.

2.8.1. RRT

A globális tervezők sok esetben topologikus gráfokat (speciális esetben fákat) használnak a konfigurációs tér struktúrájának leírásához [11]. A szakirodalomban egyik leggyakrabban használt ilyen algoritmus a *Rapidly Exploring Random Trees* [12]. Ennek a lényege, hogy a kezdeti konfigurációból egy fát építünk a szabadon bejárható konfigurációs térből. 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.



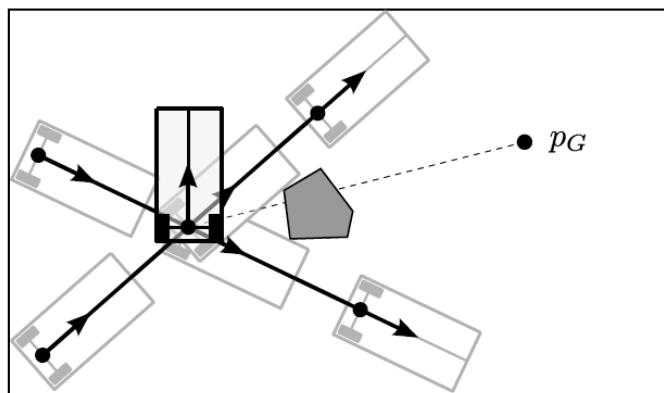
2.15. ábra. Az RRT algoritmus három különböző iterációjánál [12].

A fák építését három fázisra lehet felbontani. Az első fázisban véletlenszerűen kiválasztunk a szabad konfigurációs térből egy pontot (q_{rand}). Ezt nevezzük *mintavételezési fázisnak*. Majd végighaladunk a fán, és kiválasztjuk, melyik konfiguráció található legközelebb a mintavételezett ponthoz (q_{near}). Ezt a *csomópont kiválasztó szakasznak* nevezzük. Majd végül megpróbáljuk a q_{rand} és q_{near} konfigurációkat összekötőni (*összekötési fázis*).

Az utolsó fázisban általában egy előre meghatározott hosszig veszünk fel szakaszt. Anthonolóm rendszerek esetén is lehet használni az RRT algoritmust, ilyenkor az összekötési fázisban egy lokális tervezőt kell használnunk. Természetesen több fa is terjeszthetünk egyszerre, ilyenkor az összes fa esetén végre kell hajtani a fázisokat. A terjesztést addig folytatjuk, míg a fák össze nem kapcsolódnak.

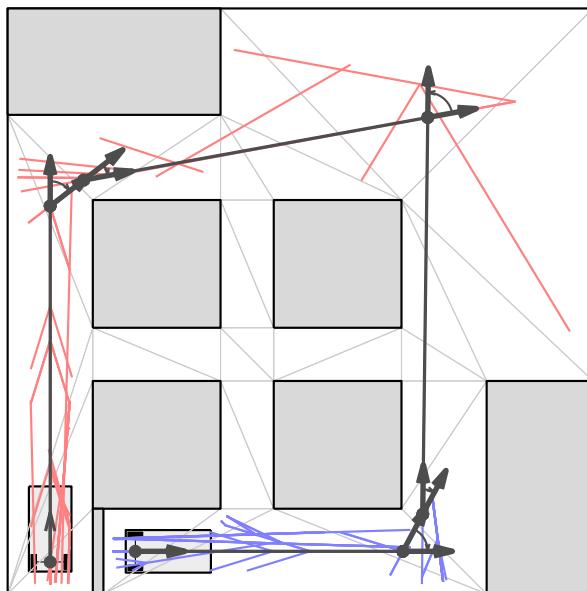
2.8.2. RTR

A *Rotate-Translate-Rotate* algoritmus a fentebb látott RRT algoritmus egy módosított változata. Elsődlegesen differenciális robotok számára tervez egyenes szakaszokból és egy helyben fordulásokból álló pályát, de jó alapot szolgáltat a C*CS algoritmushoz is. Nevét a benne használt mozgási lokális tervezőről kapta, azaz az R , mint fordulást, a T pedig az egyenesen haladást jelöli. Az eljárás az RRT minden fázisában vezet be változásokat, és mind a kezdő, mind a célkonfigurációból növesztünk fákat.



2.16. ábra. A kiterjesztés folyamata. [10]

- A mintavételezés során nem csak egy konfigurációt választunk, hanem egy pozíciót a térben (p_G), és az ehhez tartozó konfigurációs térbeli egyenes bármely pontja felé növesztjük tovább a fákat.
- A kiválasztási fázisban egyszerűbb a dolgunk, mivel csak egy pozícióhoz képest kell figyeljük a távolságokat. Itt figyelembe kell vegyük a szögek távolságát is.
- A kiterjesztés fázisa különbözik leginkább az RRT-ben látottaktól. Itt nem csak a p_G felé terjesztjük a fákat, hanem ellentétes irányban is. Illetve nem csak a pontig, hanem minden esetben addig, míg nem ütközik egy akadályba a robot. Ha a forgás közben ütközik a robot, akkor az ütközési pontból is továbbterjesztünk. Erre látható egy példa a 2.16. ábrán.



2.17. ábra. Az RTR algoritmus; piros a kezdő, kék a célkonfigurációból indított fa. [10]

Ez az algoritmus azért előnyös, mert a celladekompozíciós eljárás hibáját kiküszöböli. Tehát két szűk folyosó találkozásánál csak akkor ad megoldást, ha van hely az elfordulásra. Mivel a keresés során két fát épít, így a párhuzamos parkolás esetén is segíti a megoldás megtalálását.

3. fejezet

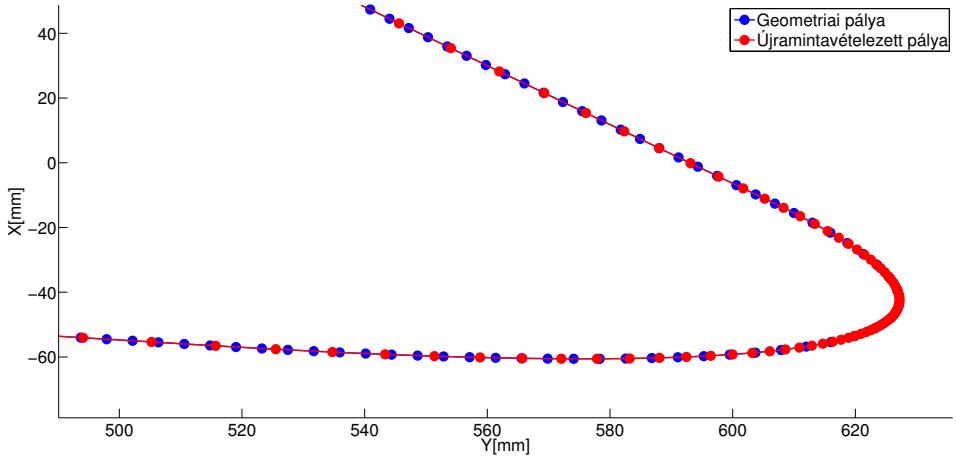
Pálya időparaméterezése

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.

A módszert Nagy Ákos dolgozta ki differenciális robotokhoz, majd módosítottam autószerű robotok számára [10]. Az elveket végül úgy alakítottuk ki közösen, hogy az ne függjen a robot típusától, így alkalmazható legyen bármelyik kerekeken mozgó robottípusra.

3.1. Időparaméterezés

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-szabályozó 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 időparaméterezéssel kapcsolatos munka található. Egy hasonló megközelítést Christoph Sprunk munkájában olvashatunk [13]. A legfontosabb eltérés, hogy Sprunk külön korlátozza a robot tangenciális és centripetális 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 fejezetekben bemutatott pályatervező által tervezett pálya speciális tulajdonságait, a célunk egy olyan algoritmus készítése volt, amely tetszőleges geometriai pályából képes sebesség információval ellátott, időben egyenletes mintavételelű pályát készíteni. Emiatt nem építhetünk a pályatervező által használt geometriai elemekre (körív, egyenes), és ezek speciális tulajdonságaira.

Általános esetben nem tudjuk analitikusan meghatározni a pálya görbületét, így görbület becslést kell alkalmaznunk [14]. Természetesen abban az esetben, ha a pályatervező rendelkezik már a pálya görbületével, az időparaméterező algoritmus azt fogja használni a becslés helyett, mivel így pontosabb eredményt érhetünk el, és gyorsítja is a végrehajtást.

3.2. 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.

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

Azokat az eseteket külön jelöljük, amikor a robot egyik kerekére vonatkozó mennyiségekről beszélünk. Alsó indexben azt, hogy első (f) vagy hátsó (r) kerék, illetve, hogy bal (l) vagy jobb (r), kerékről van szó. Ezenkívül a kerekeknél megkülönböztetjük, hogy tangenciális (a_t), centripetális (a_c) vagy eredő (a) gyorsulásról beszélünk.

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 egyenesen kötnénk össze a pályapontokat, akkor a görbületnek szükségszerűen nullának kellene lennie.

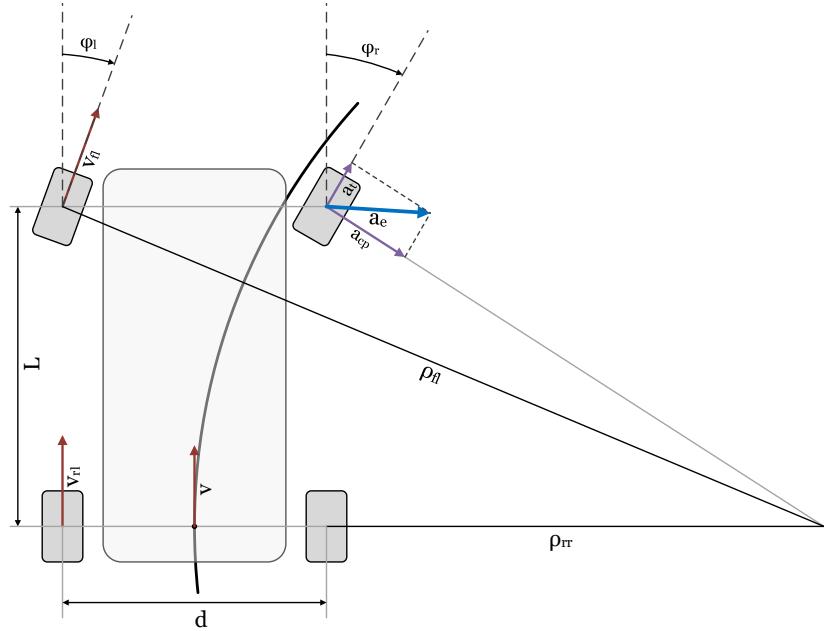
3.3. Korlátozások

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.

A sebességről létrehozásakor rendelkezésünkre áll a pálya görbülete, azaz az autó referenciaPontja által bejárt kör sugara (ρ). Ebből a hátsó kerekek fordulókörének sugara a következőképpen számolható:

$$\begin{aligned}
\rho_{rl} &= \rho - \frac{d}{2} \\
\rho_{rr} &= \rho + \frac{d}{2}. \tag{3.2}
\end{aligned}$$

Fontos megemlíteni, hogy itt ρ előjeles sugarat jelölm, ami esetünkben pozitív körülfordulási irányban pozitív (balra fordulás). Az első kerekek esetén nincs ilyen könnyű dolgunk. Ahhoz hogy az első kerekek fordulás közben ne csússzanak meg oldalirányba, a két keréknél különböző szögben kell elfordulnia, ezt nevezzük Ackermann-hajtásnak. Ez azzal áll kapcsolatban, hogy a kerekeknek különböző köríven kell elfordulniuk. Hogy ezt kiszámítunk, először írjuk fel az összefüggést a sugár és a kormányszög között, kerékpár modellt feltételezve. Ehhez használjuk fel az (1.2) egyenletet:



3.2. ábra. Autószerű robot mozgása köríven

$$\rho = \frac{L}{\tan \phi} \quad (3.3)$$

a sugár előjele konzakvens a kormányszög előjelével. Innen az Ackermann-hajtás szerint a kerekek kormányszöge:

$$\begin{aligned} \phi_l &= \arctan \left(\frac{L}{\rho - \frac{d}{2}} \right) \\ \phi_r &= \arctan \left(\frac{L}{\rho + \frac{d}{2}} \right) \end{aligned} \quad (3.4)$$

ezek és a (3.2) egyenletek segítségével számítható az első kerekek által bejárt körök sugarai:

$$\begin{aligned} \rho_{fl} &= \frac{\rho_{rl}}{\cos \phi_l} \\ \rho_{fr} &= \frac{\rho_{rr}}{\cos \phi_r} \end{aligned} \quad (3.5)$$

A különböző keréksebességek arányosak a referencia pont sebességével, ez az arány pedig a sugarak arányaival írható fel:

$$p(k) = \frac{\rho_w(k)}{\rho(k)} = \frac{v_w(k)}{v(k)} \quad (3.6)$$

ahol w jelzi a kerékre vonatkozó mennyiségeket. Látható, hogy ha a maximális keréksebességet keressük, akkor elegendő csak a maximális abszolút értékű fordulókör sugarát megkeresni. Az (3.2) egyenletek alapján következik, hogy ez mindenkor a külső kerék esetén teljesül. Az (3.5) egyenletek alapján pedig belátható, hogy az első kerekek sebessége mindenkor nagyobb lesz a hátsóéknál, mivel $|\cos \phi| \leq 1$. Azaz jobbra kanyarodás esetén a bal első kerék sebessége a legnagyobb és vice versa.

A sebességprofil készítésekor egy adott robot esetében ezekre a mennyiségekre határozunk meg korlátozásokat:

$$v^{max} : \text{A robot pályamenti sebesség korlátja} \quad (3.7)$$

$$\rho^{min} : \text{A robot minimális fordulóköre}$$

$$\omega^{max} : \text{A robot maximális szögsebessége}$$

$$a_w^{max} : \text{A robot egy kerekének maximális gyorsulás korlátja} \quad (3.8)$$

Autószerű robotok esetén a szögsebesség korlát származtatható a minimális fordulási sugár ból és a maximális sebességből, így ez a korlátozás elhagyható. Differenciális robotok esetén a sugár nincs korlátozva, így ott ezt nem lehetjük meg. A gyorsulás korlátját elég egyetlen kerékre meghatározni, feltéve, hogy a kerekek tapadási tényezője megegyezik. A kerekek maximális eredő gyorsulását a tapadási súrlódási együttható (μ_{tap}) határozza meg, amelynél a robot kerelei még nem csúsznak meg. A maximális gyorsulás és a tapadási együttható között a következő egyszerű összefüggés áll fent:

$$a_{max} = \mu_{tap_{max}} \cdot g, \quad (3.9)$$

ahol g a nehézségi gyorsulás. Írjuk fel egy kerék gyorsulását:

$$a(k) = \sqrt{a_{wc}(k)^2 + a_{wt}(k)^2} \leq g \cdot \mu_{tap}, \quad (3.10)$$

ahol $a_{wc}(k)$ egy kerék centripetalis gyorsulása és $a_{wt}(k)$ a tangenciális gyorsulása. Az (3.10) egyenletben azzal a feltevéssel élünk, hogy a robot kerelei és a talaj között a tapadási súrlódási együttható állandó és nem függ az erő irányától. Az általunk használt robotoknál ez a közelítés megengedhető, mivel a gumikerekek homogénnek tekinthetők.

Fontos megjegyezni, hogy a kerékgyorsulás korlátokat lassulásnál is alkalmazzuk. Tehát a kerék gyorsulásának abszolút értékét korlátozzák ezek a megkötések. Így azt tételezzük fel,

hogy a kerekek viselkedése gyorsulás és lassulás esetében megegyezik. A robot sebességénél viszont nem engedünk negatív értékeket, a robot végig előre haladhat. A tervező viszont megadhat olyan pályát, ahol tolatnia kell a robotnak, de ezt a pályatervező algoritmus kezeli.

3.4. Geometriai sebességprofil

Első lépésként 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. Ezt megtehetjük úgy, hogy a robot kerekeinek tangenciális gyorsulását maximalizáljuk, azonban több hatás miatt nem minden tudjuk ezt a gyorsulást maximalizálni.

Egyrészt a robot sebességekorlátját nem sérthetjük meg, 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:

$$a_{wc}(k) = v_w(k)^2 \cdot c(k) = v(k)^2 \cdot c(k) \cdot p(k), \quad (3.11)$$

ahol p tehát a referenciapont és a maximális fordulási körrel rendelkező kerék sugarának aránya. Fontos ezen kívül 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. Ezután a kerekek tangenciális gyorsulását a (3.12) egyenlet alapján határozhatjuk meg.

$$a_{wt}(k) = \sqrt{(a_{max})^2 - a_{wc}(k)^2} \quad (3.12)$$

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

$$a_t(k) = \frac{a_{wt}(k)}{p(k)} \quad (3.13)$$

$$v(k + 1) = \min \left(v^{max}(k + 1), \sqrt{v(k)^2 + 2 \cdot a_t(k) \cdot \Delta s_c(k)} \right) \quad (3.14)$$

Profil visszaterjesztés

Két esetben előfordulhat, hogy az előző pályaponthoz meghatározott sebességértéket módosítani kell. Egyrészt, ha a centripetális gyorsulás önmagában meghaladja a megengedhető

maximális gyorsulást, másrészt, ha a (3.14) egyenletben megsérjük a robot gyorsuláskorlátját. Az előbbi eset a kanyar előtti fékezést fogja meghatározni, utóbbi a pálya végén lévőt, hisz ott előírjuk, hogy $v^{max}(N) = 0$. Ha nem tennénk meg a módosítást, akkor a fékezés következtében fellépő gyorsulás (lassítás) abszolút értéke meghaladhatná a megengedettet, hiszen az előző pontokban nem tudtuk, hogy lassítani kell a robotnak. Ezt a módosítást hívjuk a profil visszaterjesztésének, mivel itt addig kell visszafelé haladva ellenőrizni, amíg a kiszámolt értékek már nem sértik meg a korlátokat.

Ahhoz hogy ezt megtegyük, kezdetnek számoljuk ki, hogy a megváltozott sebesség következtében mekkora lesz a leginkább terhelt kerék tangenciális gyorsulása.

$$a_{wt}(k) = \frac{v_w(k+1)^2 - v_w(k)^2}{2 \cdot \Delta s_w(k)} = \frac{v(k+1)^2 - v(k)^2}{2 \cdot \Delta s(k)} \cdot p(k) \quad (3.15)$$

Amennyiben a kapott tangenciális gyorsulás megséríti a gyorsulásra vonatkozó korlátot az előző pont sebességét is csökkentenünk kell. Ehhez használjuk fel a (3.12) és a (3.15) egyenleteket:

$$\begin{aligned} a_{wt}(k) &= \frac{v(k+1)^2 - v(k)^2}{2 \cdot \Delta s(k)} \cdot p(k) = \sqrt{(a_{max})^2 - a_{wc}(k)^2} \\ &= \sqrt{(a_{max})^2 - ((v(k) \cdot p(k))^2 \cdot c(k))^2} \end{aligned} \quad (3.16)$$

ezt kifejezve $v(k)$ -ra a lentebb látható negyedfokú egyenletet kapjuk:

$$\begin{aligned} d(k) &= \frac{p(k)^2}{4 \cdot \Delta s(k)^2} + c(k) \cdot (p(k))^2 \\ e(k) &= -\frac{2 \cdot v(k+1)^2 \cdot p(k)^2}{4 \cdot \Delta s(k)^2} \\ f(k) &= \frac{v(k+1)^4 \cdot p(k)^2}{4 \cdot \Delta s(k)^2} - a_{max}^2 \\ 0 &= v(k)^4 \cdot d(k) + v(k)^2 \cdot e(k) + f(k). \end{aligned} \quad (3.17)$$

A (3.17) egyenlet valós, pozitív megoldásait keressük. Felmerülhet a kérdés, hogy mi garantálja, hogy minden lesz ilyen megoldás. A Viète-formula felírásával belátható, hogy minden pozitív megoldása van az egyenletnek, a másodfokú egyenlet diszkriminánsának felírásával pedig, hogy lesz valós megoldás. Amennyiben több pozitív valós megoldása van az egyenletnek, akkor a legnagyobb megoldást választjuk. Végül erre az értékre kell módosítanunk a sebességet, majd visszalépni, hogy ezzel a módosítással nem sértünk-e újabb értéket¹.

¹Általános esetben bevezethető egy tangenciális gyorsulás korlát is, de az autószerű robot esetén ezt nem alkalmaztuk.

A visszaterjesztés során a sebesség és szögsebesség korlátokkal nem kell foglalkoznunk, hiszen minden esetben, mikor módosítjuk a sebességet, csökkentjük az értékét.

3.5. Újramintavételezés

Miután elkészítettük a geometriai pályához tartozó sebességprofilt, létrehozzuk a végleges pályát, amit majd a pályakövető egység bemenetként megkap. Ez a végleges pálya már időben egyenletesen lesz mintavételezve (újramintavételezett pálya). Ehhez először számoljuk ki az eltelt időt a geometriai pálya mentén, amelynek alapja, hogy két pályapont között a robot állandó gyorsulással halad.

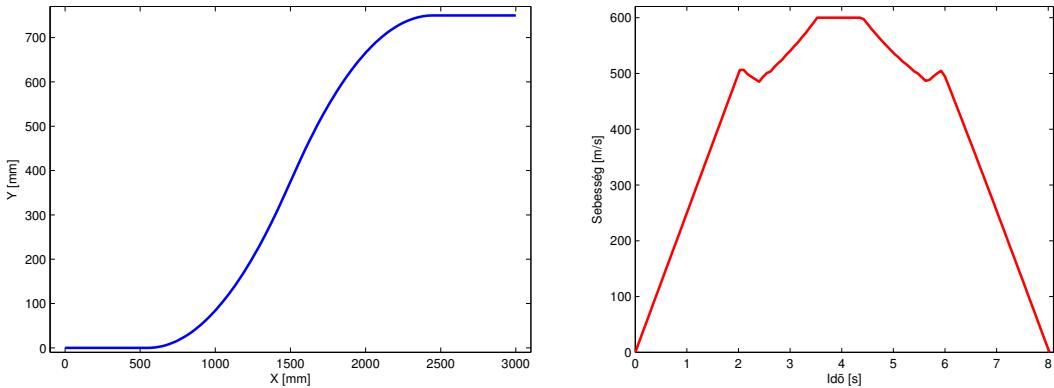
$$\Delta t^g(k) = \frac{2\Delta s^g(k)}{v^g(k) + v^g(k+1)} \quad (3.18)$$

$$t^g(k+1) = t^g(k) + \Delta t^g(k) \quad (3.19)$$

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 pályához tartozó pontok számának meghatározására:

$$N^s = \lceil t^g(N^g)/t_s \rceil + 1 \quad (3.20)$$

A pontok számába beleérjük a kezdő és végpontot is. A (3.20). 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.



3.3. ábra. A pálya és a mintavételezett sebességprofil autószerű robot esetén

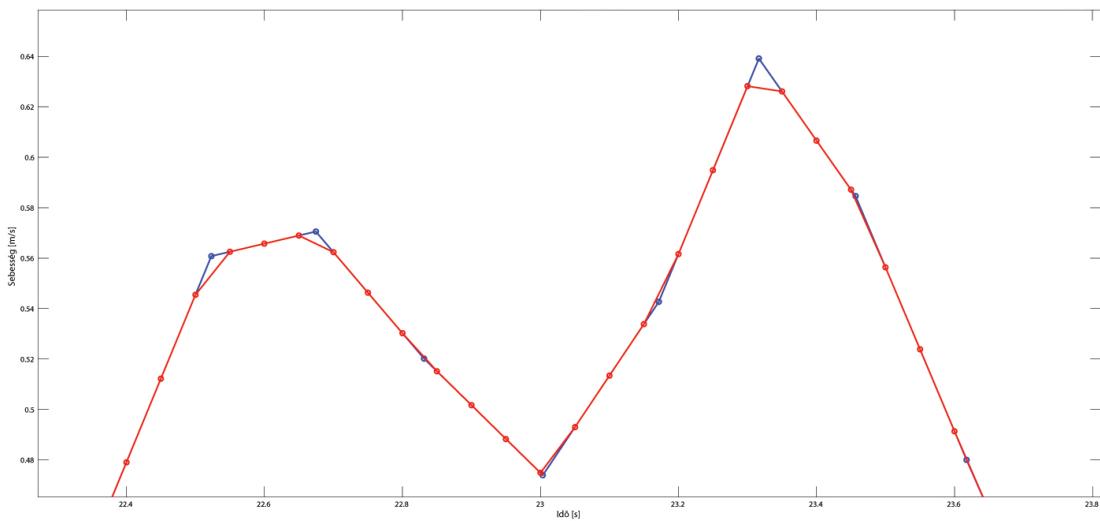
Most pedig határozzuk meg a mintavételezett pálya pontjaiban a sebességet. Ezt a geometriai pálya alapján tessziük, figyelembe véve, hogy a mintavételezett pálya esetén is két

pont között állandó gyorsulást feltételezünk. A számítás egy egyszerű lineáris interpolációt valósít meg:

$$v^s(k) = v^g(j) + v^g(j+1) \cdot it(k) \quad (3.21)$$

$$it(k) = \frac{t^s(k) - t^g(j)}{t^g(j+1) - t^g(j)}, \quad (3.22)$$

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.



3.4. ábra. A geometriai (kék) és újramintavételezett (piros) sebességprofil.

A kiszámított sebességprofil alapján könnyedén adódik a megtett út is:

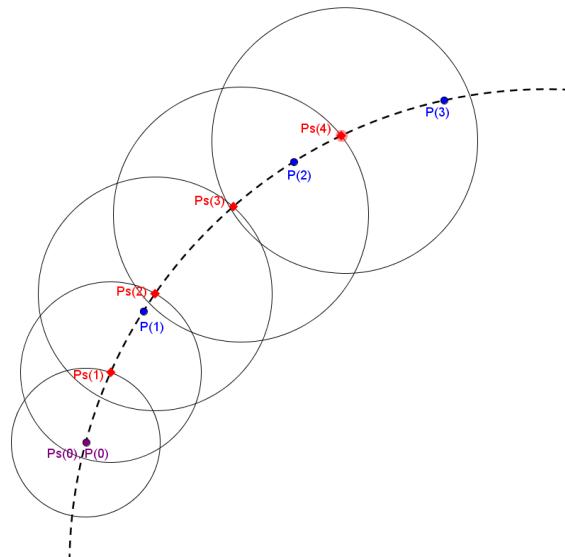
$$\Delta s^s(k) = \frac{v^s(k) + v^s(k+1)}{2} \cdot t_s \quad (3.23)$$

$$s^{s+1}(k) = s^s(k) + \Delta s^s(k) \quad (3.24)$$

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 3.5. ábrán.



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

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 mintavételezet pontot a geometriai pálya első pontjába helyezzük el.

4. fejezet

Pályakövető szabályozás

Ebben a fejezetben az elkészült, időben egyenletesen mintavételezett pálya követésének problémáját, majd az ehhez készült szabályozási algoritmusokat mutatom be.

4.1. Pályakövetés

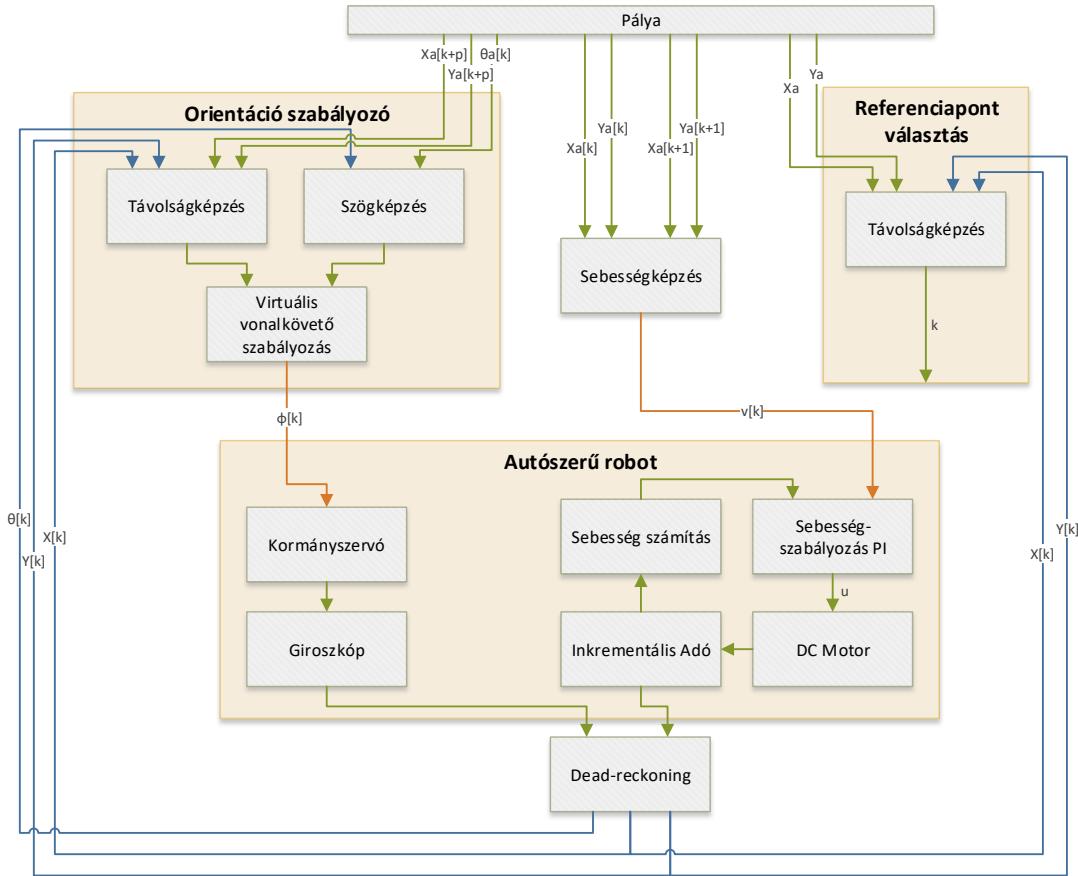
Autószerű robot esetén a követendő pályát szegmensekre bontjuk fel. Egy szegmensen belül a robot megállás nélkül halad előre vagy hátra a pálya mentén. Ebből következik, hogy a szegmensek a haladási irányból, és a pálya időben egyenletesen mintavételezett pontjaiból állnak. Kettő közt a robotnak nem szükséges semmilyen speciális feladatot végrehajtani, közvetlenül folytatja az útját következő szegmens végrehajtásával¹. Ilyenkor van lehetőség az irány módosítására. Ebből következik, hogy a teljes útvonal szegmensekre bontását irányváltoztatások jelzik, az azonos irányú pályaelemeket, mint például amik a C*CS esetén is keletkezhetnek, egy szakaszba egyesíti az algoritmus.

A pályakövető szabályozás alapvetően két szintre oszlik, ahogyan ez a 4.1. ábrán is látható. A felsőbb szinten a pályába kódolt sebesség és pozíció követése a cél, a robot aktuális pozíciója alapján, míg az alsóbb szinten az így számolt sebesség és kormányszög alapjel szabályozása történik. Ez a szabályozás az algoritmusok megvalósításának szintjén is elkölnöl, míg az alacsonyabb szintű szabályozást magán a robot vezérlőkártyáján futtathatjuk, addig a magasabb szintű szabályozást egy nagyobb teljesítményű számítógépre kell elhelyeznünk.

4.1.1. Sebességszabályozás

Alacsony szinten a robotsebességénél történik meg a sebességszabályozás. Az általam használt valós robot négy kerék meghajtású, egyetlen DC motor gondoskodik a robot mozgatásáról. A robot középső erőátviteli tengelyére csatlakozik egy inkrementális adó, mely biztosítja a sebességszabályozás számára a visszacsatolást. Ahogyan a feszültségevezérelt egyenáramú motorok esetén gyakran lenni szokott, én is egy módosított PI szabályozót használtam sebességszabályozásra.

¹Ellentétben a differenciális robottal, amely képes egy helyben való elfordulásra, így akár törtszakaszok követésére is.



4.1. ábra. A pályakövetés áttekintő blokkdiagramja.

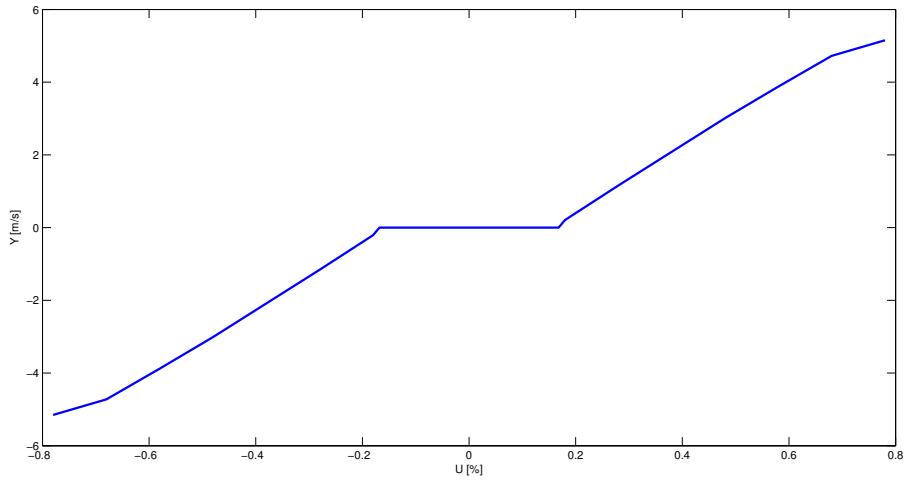
A szabályozótervezés előtt figyelembe kell vegyük a motor és a mechanikából adódó nemlineáris hatásokat, mivel ezek a hatások nagymértékben leronthatják a szabályozási rendszer tulajdonságait. A nemlineáris hatást annak inverzével lehet kiiktatni a rendszerről, mivel így az együttes hatásuk egy egységnyi erősítéssé módosul. Az inverz meghatározásához meg kellett mérnem a nemlineáris hatás karakteristikáját. Ez látható a 4.2. ábrán.

A ábrán a vízszintes tengelyen a motorvezérlő által kiadott kitöltési tényező, a függőleges tengelyen pedig a robot állandósult sebessége látható. Megfigyelhető, hogy a leginkább káros hatása a holtsávnak van. Ez a jelenség alacsony sebességeken érvényesül. Ezt a hatást az okozza, hogy a súrlódások következtében a motor nem tud elfordulni, így a kiadott beavatkozójel nem tud érvényre jutni. A karakterisztikán látható még a telítődés hatása is, azaz hogy magasabb sebességeken már nem érhető el olyan mértékű gyorsulás.

A PI szabályozók esetében gyakran előforduló probléma az elintegrálódás [15]. Az elintegrálódás a rendszerben lévő beavatkozószerv telítése miatt lép fel, kiküszöbölése történhet többféléképpen, szabályozó típusától függően. Esetünkben egy FOXBORO struktúra segítségével előzzük meg az elintegrálódást.

A FOXBORO szabályozás esetén egy pozitívan visszacsatolt rendszer segítségével valósítjuk meg a PI szabályozót. A szabályozás hatásvázlata a 4.3. ábrán látható.

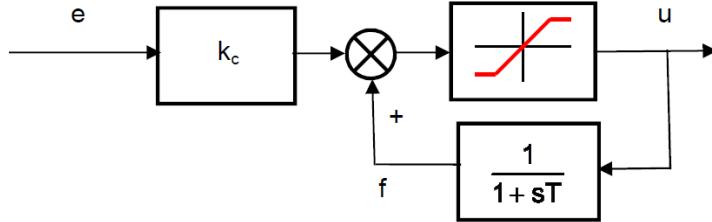
Átviteli függvénye:



4.2. ábra. Motor és mechanika nemlineáris karakterisztikája

$$C(s) = k_C \frac{1}{1 - \frac{1}{1+sT}} = k_C \left(1 + \frac{1}{sT} \right) \quad (4.1)$$

Azaz a lineáris tartományban pontosan megegyezik egy PI szabályozással.



4.3. ábra. A FOXBORO szabályozó hatásvázlata [15]

4.1.2. 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 előírt 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 első közelítésben igen egyszerű, a pálya pontjai közül a robot pozíójához legközelebbi pályapontot választjuk referenciapontnak, és így már egyértelműen adódik a sebesség alapjelünk is.

A fejlesztés egy korai stádiumában 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 szabályozó segítségével korrigáltam a pályába kódolt sebesség alapjelet, hogy a robot elérje a referenciapontot.

Amennyiben nem ideális modellt használtunk, a megoldás nem működött, a rendszer instabillá vált. Később beláttam, hogy a megoldás problémája az volt, hogy nem csak azt írtuk elő a robot számára, hogy mekkora sebességgel haladjon a pálya mentén, hanem a referenciapontron 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 alapból úgy írtuk elő, hogy a lehető leggyorsabban haladjon a robot a pálya mentén. Tehát az alapjel módosító szabályozóval arra kényszerítenénk 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 a roboton valós időben kell működnie, ezért a referenciaPont meghatározásánál nem megyünk végig a pálya összes pontján. Hogy ezt megtégyük, a legközelebbi pont keresését az előző iterációban használt referenciaPontnál kezdjük, és csak egy bizonyos számú pontot vizsgálunk meg. Ha a robot korlátai megfelelően lettek beállítva, akkor az egymás utáni referenciaPontoknak időben sorban kell következniük. Ezért teljesen felesleges a pálya összes pontját megvizsgálnunk.

4.1.3. Túlhaladás problémája

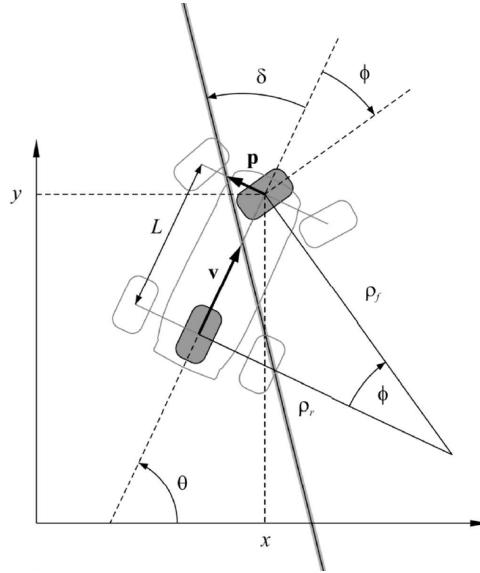
A pályakövetés valós roboton való tesztelése során egy érdekes jelenséget tapasztaltunk, miszerint néha a pálya közepén megállt a robot és nem indult tovább. Későbbi elemzés során kiderült, hogy ez abban az esetben történik, mikor túlhalad egy szegmensen, tehát nem pontosan a szegmens utolsó pontjában áll meg a robot.

A probléma ebben az esetben az, hogy a referenciaPont-választás alapján a következő szegmens legelső pontját találja meg, és az ehhez tartozó sebességről viszont igen alacsony. Míg el nem éri a pálya kezdetét, addig ezt a sebességet tartja a robot. A gondot az alsóbbrendű PI szabályozás beállási ideje, illetve annak lassúsága okozza.

A probléma elkerülését úgy oldottam meg, hogy amíg a pályaszakasz első pontja van legközelebb a robot pozícójához, addig nem a pályába kódolt sebességet továbbítjuk az alsóbb szintű szabályozónak, hanem a gyorsulás és sebességkorlát alapján növeljük a robot sebességét. A valóságban ez csak köztes szakaszok váltásánál jelentkezik, mivel a pályát eleve a robot kezdőpozíciójából tervezük. Ezzel szemben két szegmens közt nem garantált a pálya pontos követése, mivel nem pozíciószabályozást használunk.

4.2. Virtuális vonalkövető szabályozás

Az orientációszabályozás feladata a kormányszög alapjel biztosítása a pályakövetés során. Erre a célra egy virtuális vonalkövetést valósítottam meg. A vonalkövető autók rendszermodelljénél és szabályozásánál azzal a feltételezéssel élünk, hogy az autó első keréktengelye alatt egy keresztirányú, egydimenziós vonalszenzor helyezkedik el. A jelenlegi esetben a pályakövető szabályozás elvét egy ehhez hasonló „virtuális szenzor” segítségével fogalmaztam



4.4. ábra. Ferde vonal és robot modellje [16]

meg. Ez a módszer nagyon hasonló a RobonAUT [16] versenyen is látott vonalkövető autók szabályozására, azzal az előnnyel, hogy itt sokkal pontosabban ismert a „vonal” helye és orientációja.

Mivel egy ilyen vonalkövető autó esetén a vonalszenzor mozgására van szükségünk, így módosítani kell a robotunk modelljét (1.2), amit a következőképpen tehetünk meg:

$$\begin{aligned}\dot{x} &= v_r \frac{\cos(\theta + \phi)}{\cos \phi} \\ \dot{y} &= v_r \frac{\sin(\theta + \phi)}{\cos \phi} \\ \dot{\theta} &= v_r \frac{\tan \phi}{L},\end{aligned}\tag{4.2}$$

ahol a jelölések megegyeznek az 1.2 esetén használtakkal. Fontos megjegyezni, hogy bár az első tengely középpontjára írtuk fel a mozgásegyenletet, a hátsó referencia pont sebességevel számolunk. A továbbiakban ezt a sebességet egyszerűen csak v -vel jelöljük. A 4.4 ábrán látható modellt tételezzük fel, azaz azt, hogy az a vonal, amit követni kívánunk, egyenes. Ebben az esetben meghatározható a vonal és a robot orientációjának különbsége (δ), illetve a vonal és az első tengely középpontjának – szenzorsor közepének – előjeles távolsága (p). Ezen feltételezések és a mozgásegyenlet alapján felírhatjuk a következő következtetéseket [16]:

$$\begin{aligned}\dot{\delta} &= -v \frac{\tan \phi}{L} \\ \dot{p} &= v \cdot \tan \delta - v \cdot \tan \phi - v \cdot \frac{p}{L} \tan \delta \tan \phi\end{aligned}\tag{4.3}$$

Látható, hogy ez egy nemlineáris rendszer, de a szabályzótervezéshez ezt linearizálnunk kell. Mivel az a célunk, hogy a robot a vonalon, ésazzal párhuzamosan helyezkedjen el, így a munkapont, amely körül a linearizálást elvégezzük a $p = 0$, $\phi = 0$ és a $\delta = 0$. Így a következő egyenletekkel számolhatunk:

$$\begin{aligned}\dot{\delta} &= -\frac{v}{L}\phi \\ \dot{p} &= v(\delta - \phi - 0)\end{aligned}\tag{4.4}$$

A linearizálás egyszerű, mert a tangens 0 környezetében jól közelíthető az argumentumával. Látható, hogy a \dot{p} esetén az utolsó tagot elhanyagoljuk, mivel a két kis szög szorzata annyira kis számot eredményez, hogy ez gond nélkül megtehető. Ha ezt kissé más formában írjuk fel, rögtön megkapjuk a linearizált rendszer állapotteres leírását:

$$\begin{aligned}x &= [\delta \quad p]^T \\ \dot{x} &= \begin{bmatrix} 0 & 0 \\ v & 0 \end{bmatrix} x + \begin{bmatrix} -v/L \\ -v \end{bmatrix} \phi \\ p &= [0 \quad 1]x + 0 \cdot \phi\end{aligned}\tag{4.5}$$

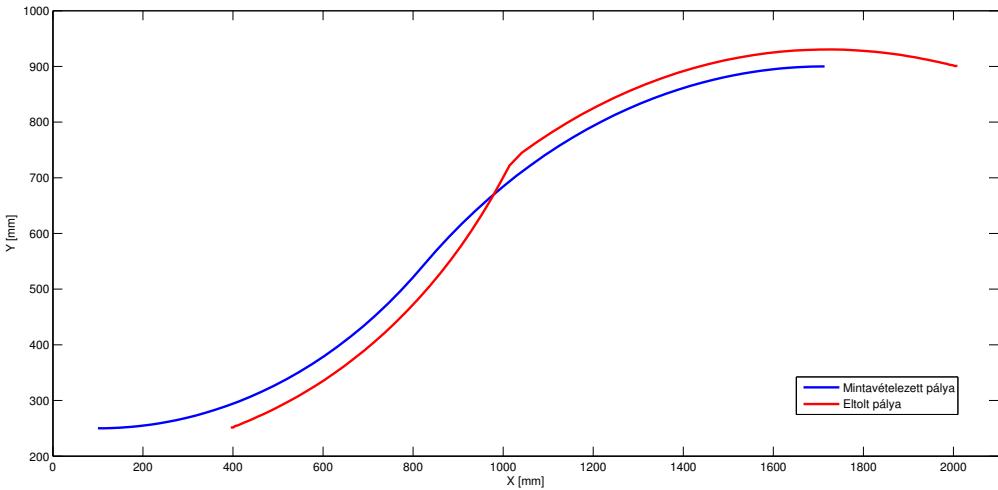
Ellentétben egy valós vonalkövető autóval, esetünkben viszonylag pontosan meg tudjuk határozni a rendszer állapotváltozóit. Így célszerű közvetlenül ezek visszacsatolása, mivel így szabadon megválaszthatóak a visszacsatolt rendszer pólusai. Ezt érdemes úgy megtenni, hogy minimálisra csökkentsük a túllendülést. Ha a rendszer válaszát kéttárolós lengőtaggal közelítjük, akkor annak átviteli függvénye a következő:

$$W(s) = \frac{\omega_0^2}{\omega_0^2 + 2\xi\omega_0 s + s^2},\tag{4.6}$$

ahonnan a pólusok:

$$s_{1,2} = -\omega_0\xi \pm j\omega_0\sqrt{1 - \xi^2},\tag{4.7}$$

ahol ω_0 a rendszer csillapítatlan sajátfrekvenciája és ξ a csillapítási tényező. Ha túllen-dülés mentes rendszert szeretnénk, de a lehető leggyorsabb beállási idővel, akkor $\xi = 1$ -et kell választanunk. Az ω_0 megválasztására nincsen hasonló korlátozásunk, ezt az aktuális pályához tudjuk igazítani.

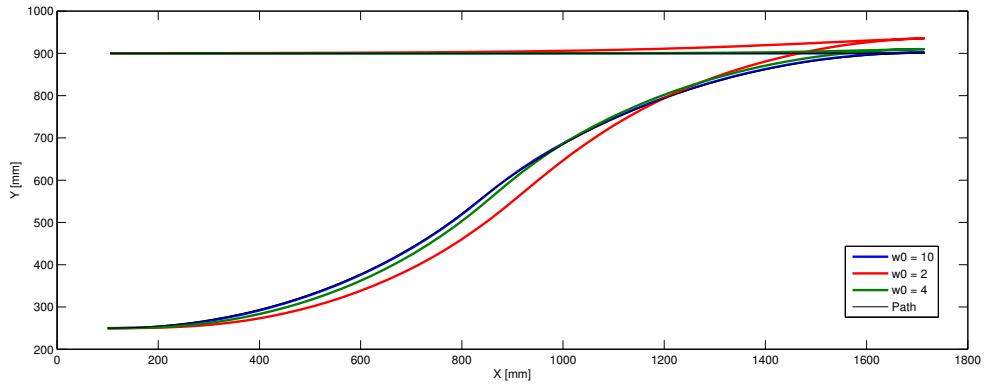


4.5. ábra. Kék az eredeti pálya, piros az virtuális szenzorsor számára elolt pálya

Az algoritmust úgy készítettem el, hogy az inicializálási fázisban a kívánt pólusoknak megfelelően, az Ackermann-képlet [17] segítségével kiszámítja az erősítési tényezőket, és később ezt használja fel a szabályozási fázisban. Az eredmények azt mutatták, hogy az így készült szabályozóval a szimulációban a robot trajektóriája a kanyarokat levágta. Ez az eredmény egyáltalán nem meglepő, mivel a szabályozást úgy írtuk fel, hogy az autó eleje kövesse a pályát, de a pályatervezés során a robot referenciaPontjának pályáját terveztük meg. Szerencsére ezt egyszerűen orvosolhatjuk, ha a mintavételezett pálya minden pontját eltoljuk az autó hosszával (4.5. ábra).

4.3. Eredmények

Most pedig tekintsük át a pályakövetés alakulását különböző paraméterek esetén. A sebességszabályozásnál láttuk, hogy egy újabb szabályozás bevezetése csak hibát okoz a rendszerünkben, így csak az alacsony szintű PI szabályozó paramétereivel tudjuk módosítani a sebességprofil követésének minőségét.



4.6. ábra. Pályakövetés szimulált eredményei különböző ω_0 paraméterekkel

Az orientációszabályozás esetén az előírt pólusok értékein kívül a szenzorsor pozíciójával is lehetőségünk van módosítani a szabályozás minőségét. Bár a követendő pályát, az autó hosszával eltoltuk, az eredmények azt mutatták, hogy hasznos, ha van egy további paraméterünk a követés során, így további előretekintést adhatunk meg. Intuitívan belátthatjuk, hogy minél előrébb tekintünk, annál gyorsabban tudunk reagálni, így alacsonyabb beavatkozójel szükséges a szabályozás során.

Ehhez képzeljük el, hogy az első kerekeket egy d távolsággal előrébb helyezzük. Ebben az esetben a kormányszög és a kerék által bezárt fordulókör sugara közt a következő összefüggést írhatjuk fel:

$$\tan \phi = \frac{L}{\rho}, \quad \tan \gamma = \frac{L+d}{\rho} \quad (4.8)$$

Emlékeztetőül előbb az eredeti összefüggés látható. Ebben az esetben γ az új kormányszögünk. A két egyenletet egyenlővé téve:

$$\frac{L}{\tan \phi} = \frac{L+d}{\tan \gamma}, \quad (4.9)$$

amiből a következő kifejezést kapjuk [16]:

$$\tan \phi = \tan \phi \cdot \frac{L}{L+d}, \quad (4.10)$$

Vegyük észre, hogy az előretekintés távolságának megnövelésével a virtuális kormányszög (γ) megnő. Mivel a valóságos kormányszög mechanikailag korlátozva van, ez telítést jelent a szabályozásunkban, amit így ki tudunk bővíteni. Természetesen az előretekintés mértékének túl nagyra választása azt eredményezi, hogy a kanyarokat hajlamos a robot levágni. Mégis jelentős előnyvel rendelkezik, mivel így gyengíthető a kormányszervó nem nulla beállási idejének hatása.

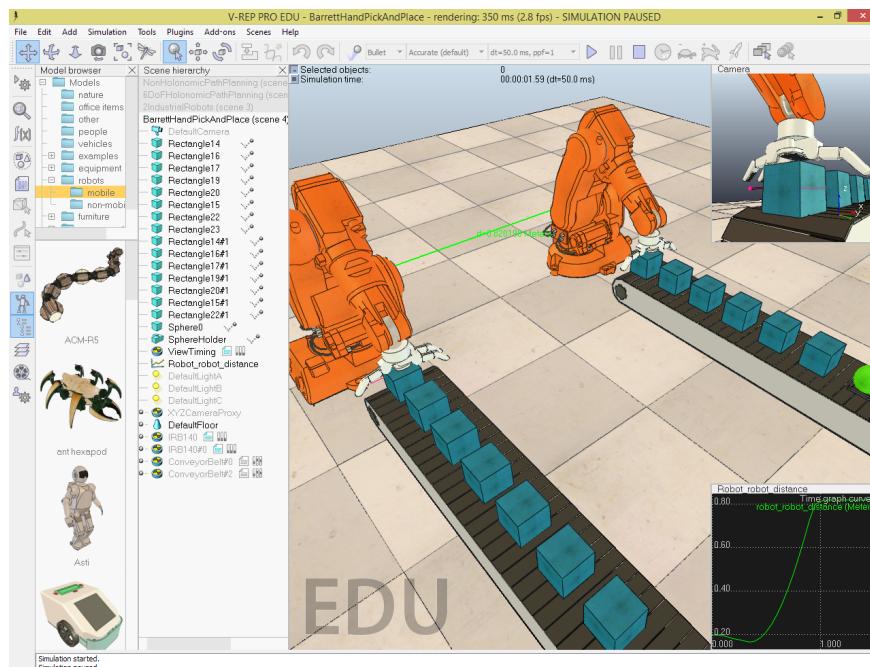
5. fejezet

Algoritmusok megvalósítása

Ebben a fejezetben az algoritmusok megvalósításáról, és az azokhoz használt eszközök-ről beszélek. Bemutatom a használt szimulációs környezetet, és a köré készült programok működését, majd leírom a szimulátorban és a valós robotokon elért eredményeimet.

5.1. Szimuláció – V-REP

A robot mozgásának szimulálására a V-REP robotszimulátort használtam. A program a Coppelia Robotics terméke [18], amely oktatási célból ingyenesen letölthető és használható. Nagyon széleskörűen használható program a robotika minden ágában. Tesztelhető benne ipari szerelőrobotok működése, ahogyan az az 5.1 ábrán is látható, felhasználható ilyen robotok programozásának oktatására is, de a mobil robotok területén is kifejezetten praktikus eszköz. Jól dokumentált, sok oktató anyaggal, példaprogramokkal együtt. Folyamatosan frissítik és új funkciókkal bővítik a programot.

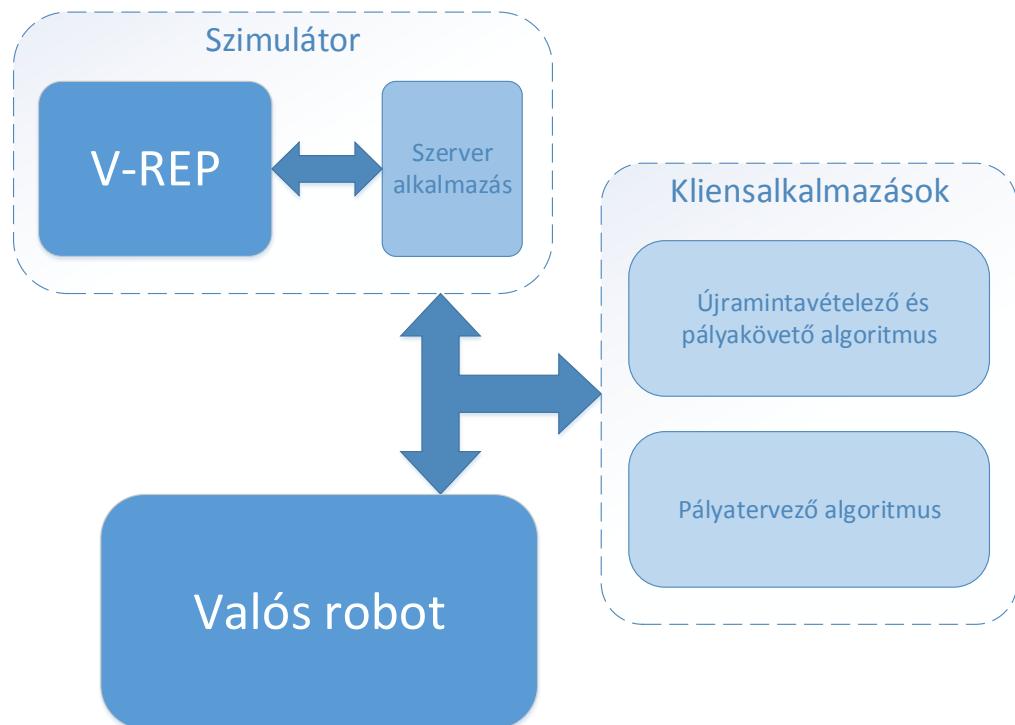


5.1. ábra. A V-REP szimulációs program

5.1.1. Szerver program

Több módon is kiegészíthető a program működése. A megoldásunkban egy szerver programot hoztunk létre, mely kommunikál a V-REP egy lua szkriptjével, majd a kapott üzenet alapján mozgatja a szimulált robotot. A szerver nem csak a szimulátorral áll kapcsolatban, hanem hozzá csatlakozhatnak a különböző egyéb algoritmusok, ahogy az az 5.2 ábrán is látható. A fejlesztés során próbáltuk a szimulációt a robot típusától függetlenné tenni, ezt a következőképpen valósítottuk meg:

A szimuláció indulásakor a lua szkript elküldi a szimuláció módját, és a hozzá tartozó paramétereket. Innen a szerver alkalmazás eldönti, milyen paraméterek és egyéb adatok érkezhetnek, illetve, hogy melyik kliensre kell várjon. Ha a kapcsolat létrejött a kliens alkalmazással, akkor az a paramétereknek megfelelően végzi a feladatát, majd az eredményt a szerver alkalmazáson keresztül elküldi a szimulátornak. Ez a struktúra első ránézésre igen bonyolultnak tűnik, de ez a módszer biztosítja, hogy a szimuláltot egyszerűen lecserélhessük egy valós robotra, vagy akár párhuzamosan is működhessen vele. A szerver működése teljes mértékben transzparens, a későbbiekben ennek működését a szimulátor részének tekinthetem.

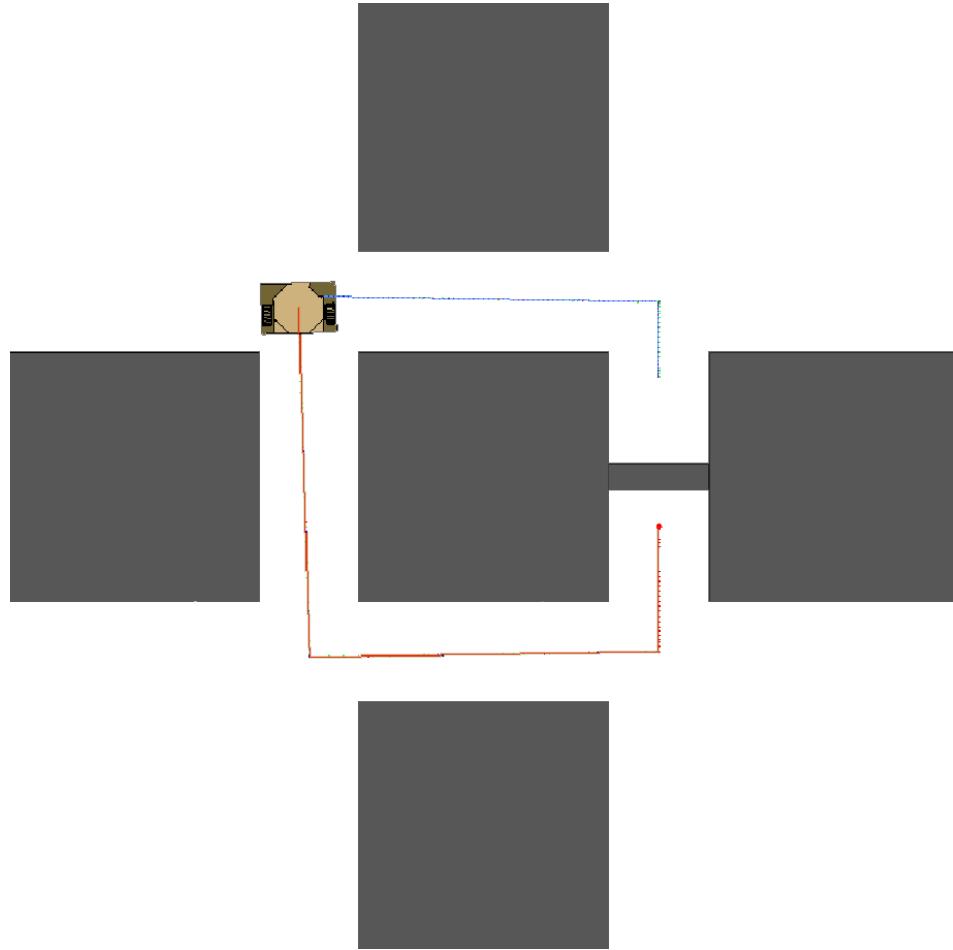


5.2. ábra. Az elkészült keretrendszer blokkvázlata

5.1.2. Kliens program

A különböző kliensprogramok más-más paramétereket várnak, ezt biztosítja a szerver alkalmazás. Az időparaméterező és a pályakövető szabályozás teszteléséhez készült egy kliens, mely a szimuláltot fogad egy előre elkészített pályát, majd ezt újramintavételezi. Az így

készült pályát visszaküldi a szimulátornak, ami kirajzolja az új pályát, majd elküldi a robot aktuális pozíóját. Innen átveszi a működést a pályakövető algoritmus, a kapott pozíciót feldolgozza, és ez alapján az előző fejezetben részletezett módon kiszámítja a beavatkozó jeleket, amit visszaküld a szimulátornak. A szimulátor és a pályakövető alkalmazás működése szinkronizálva van, azaz megvárják egymást a következő lépéssel.

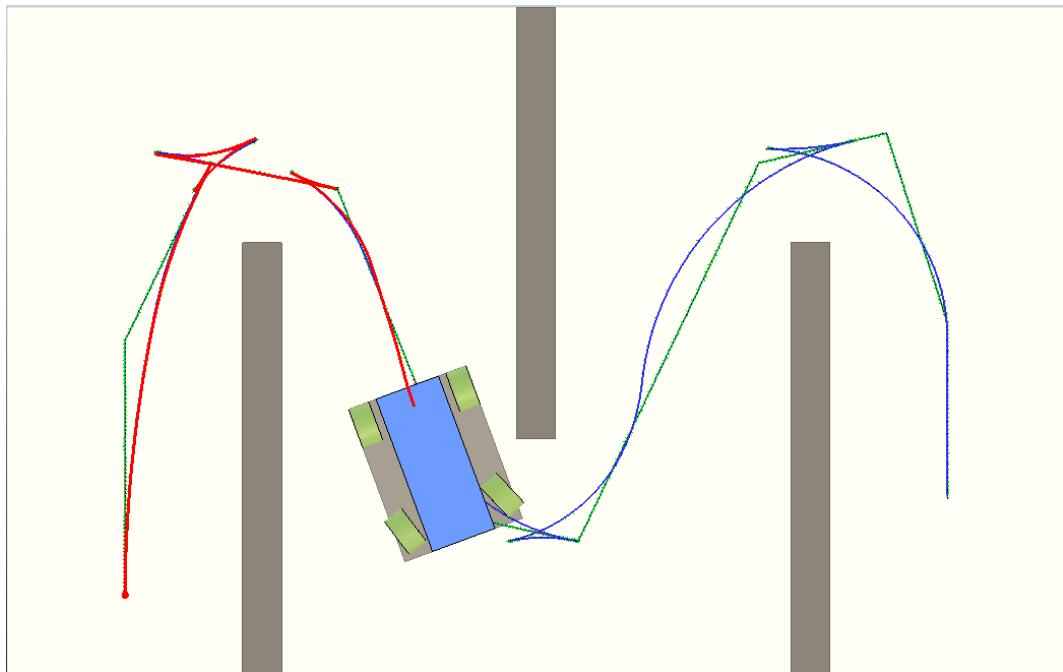


5.3. ábra. Az elkészült keretrendszer nem csak autószerű robotok esetén használható. Az ábrán egy differenciális robot látható követés közben.

A másik elkészült kliens alkalmazás a pályatervező program. Ez nem vár pályára a szimulátortól, csak az előre meghatározott környezet nevére. Itt kompromisszumot kellett kötnünk, mivel a szimulátor speciális fájlformátumot tud csak kezelni, ezért közös fájllokkel dolgozik a két program, de a pályatervező más forrásból is elfogad pályát, így továbbra is lecserélhető marad a szimulátor. A pályatervezés után a működése teljesen megegyezik a pályakövető kliensprogramnál leírtakkal, azzal a különbséggel, hogy a pályát itt a tervező szolgáltatja.

5.1.3. Implementáció

A programokat a C++ nyelvet készítettem el. Azért esett a választásom erre a programozási nyelvre, mert a valós roboton, beágyazott rendszerben is könnyedén használható.



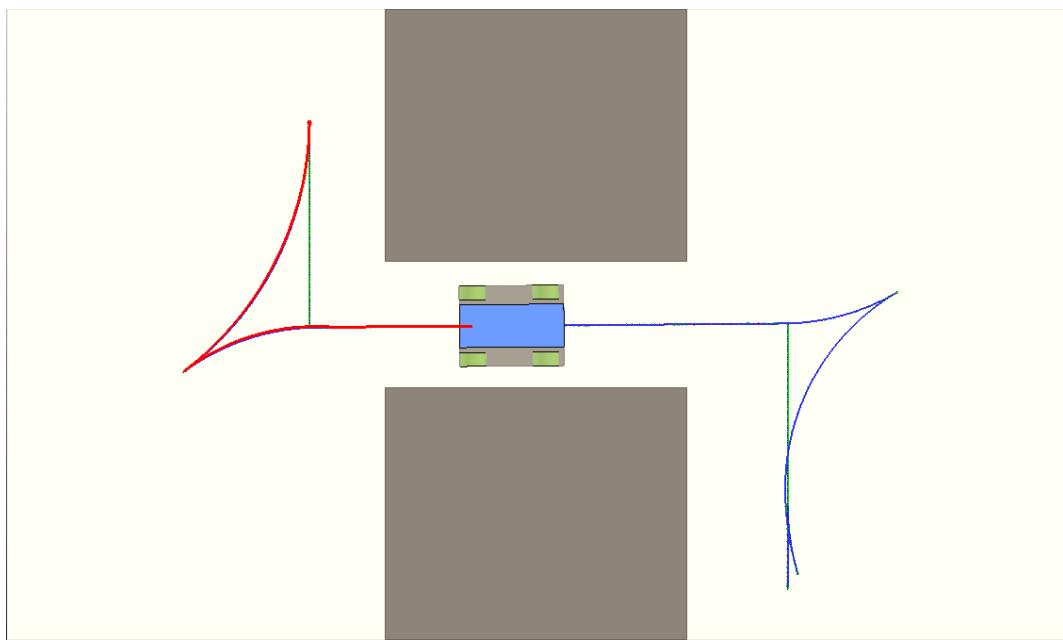
5.4. ábra. A C^*CS alkalmazása autószerű robotok esetén, szimulációs környezetben

A beagyazott környezet miatt igyekeztem kerülni bármiféle olyan külső szoftvercsomag használatát, aminek a használata problémás lehet a valós roboton.

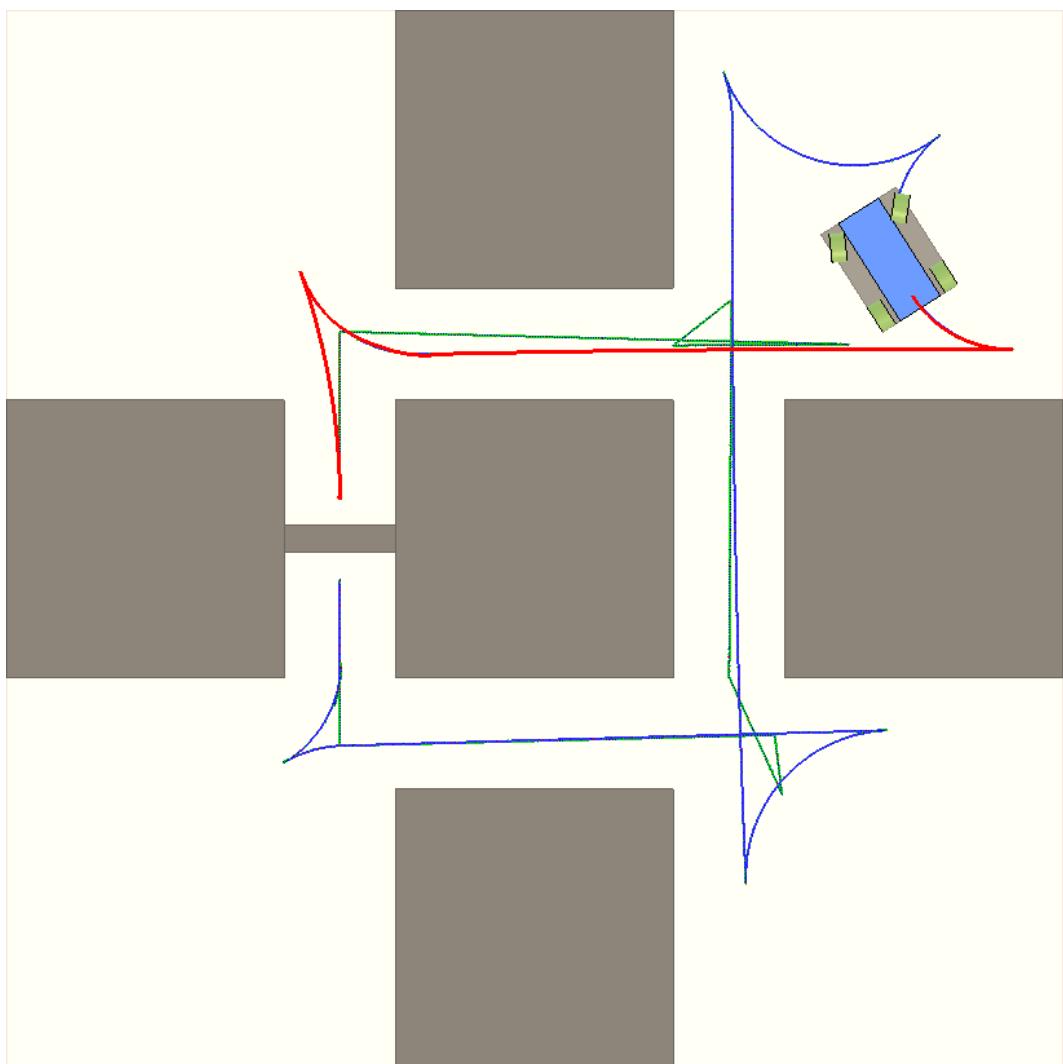
Ezek mellett a fejlesztés során igen fontos volt az objektum-orientált szemléletmód, mivel így biztosítható a legjobban a modularitás, és a későbbi egyszerű fejlesztés, módosítás. Az implementálás során egyéb előnyös tulajdonságát is ki tudtuk használni, ezek közül a legjelentősebb az újrafelhasználhatóság volt. A programozás előrehaladtával a fejlesztés sebessége is nőtt, mivel az előzőleg elkészített kódrészleteket egyszerűen újra tudtuk használni.

5.2. Szimulációs eredmények

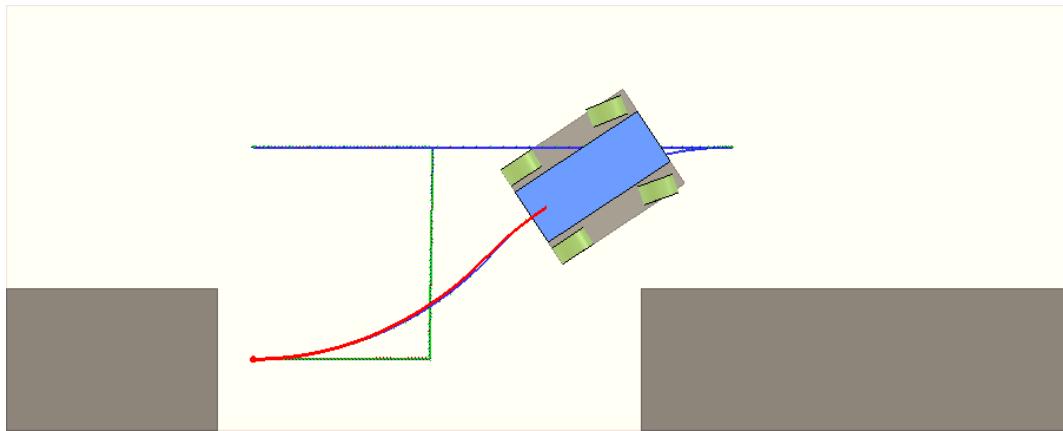
Az általam ismertetett algoritmusok szimulációs környezetben a várakozásomnak megfelelően működtek. Az eredmények a következő ábrákon láthatóak. A képek a V-REP szimulátorról készültek, az akadályokat és a pálya határait színes polygonok jelzik, a pályatervező által generált pályát kék színnel jelölték, míg a robot által bezárt pályát sárgával. A robot alatt található polygon jelzi a tervezés során ténylegesen figyelembe vett robot méretét.



5.5. ábra. A 2.8. ábrán látható pálya RTR globális tervező esetén.



5.6. ábra. Szűk folyosók. Autószerű robot esetén igen bonyolult pálya adódik.



5.7. ábra. A tervezett pálya igen hasonló egy valós esetben végrehajtott parkoláshoz

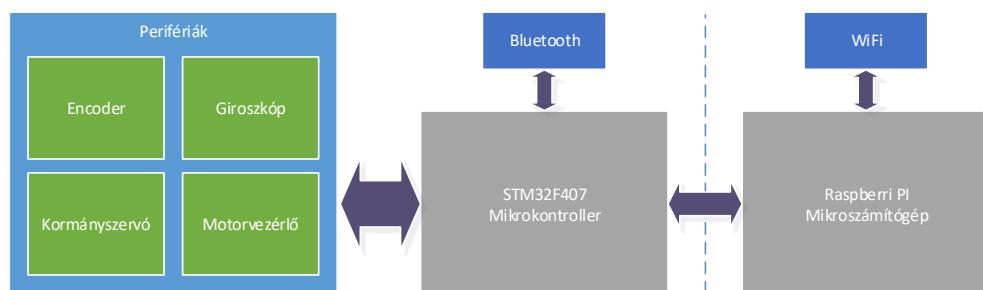
6. fejezet

Megvalósítás valós roboton

Az elkészült algoritmusok elméleti szinten és a szimulátorban is a vártnak megfelelően működtek. A szimuláció során használt modelleket próbáltam minél inkább a valósághoz igazítani, de a valósággal teljes mértékben megegyező modell megalkotása természetesen nem lehetséges, így az elkészült algoritmusokat valós roboton is kipróbáltam. Ebben a fejezetben az elkészült robotról, a fejlesztés és az implementálás közben felmerült nehézségekről és a jövőbeli tervekről fogok beszélni.

6.1. Felépítés

A robot elkészítése a szakmai gyakorlatom keretében valósult meg. A fejlesztés során a célom egy olyan autószerű mobilrobot készítése volt, amely a környezet ismeretében képes a pályatervező algoritmusok futtatására és egy elkészült pálya követésére.



6.1. ábra. A robot rendszermodellje

Az elkészült rendszer két részre bontható a feladat jellegéből fakadóan. Az alacsonyabb szintű feladatok, mint a motor vezérlése, szabályozása, kormányzás és szenzorfúzió egy beágyazott mikrokontroller feladatai közé tartoznak. Emellett beépítésre került egy bluetooth modul, amely segíti a hibakeresést. Ezen felül ez a modul lehetővé teszi, hogy ezt az alacsonyabb szintű alrendszer leválasszuk, az erőforrásigényesebb algoritmusok fejlesztése során.

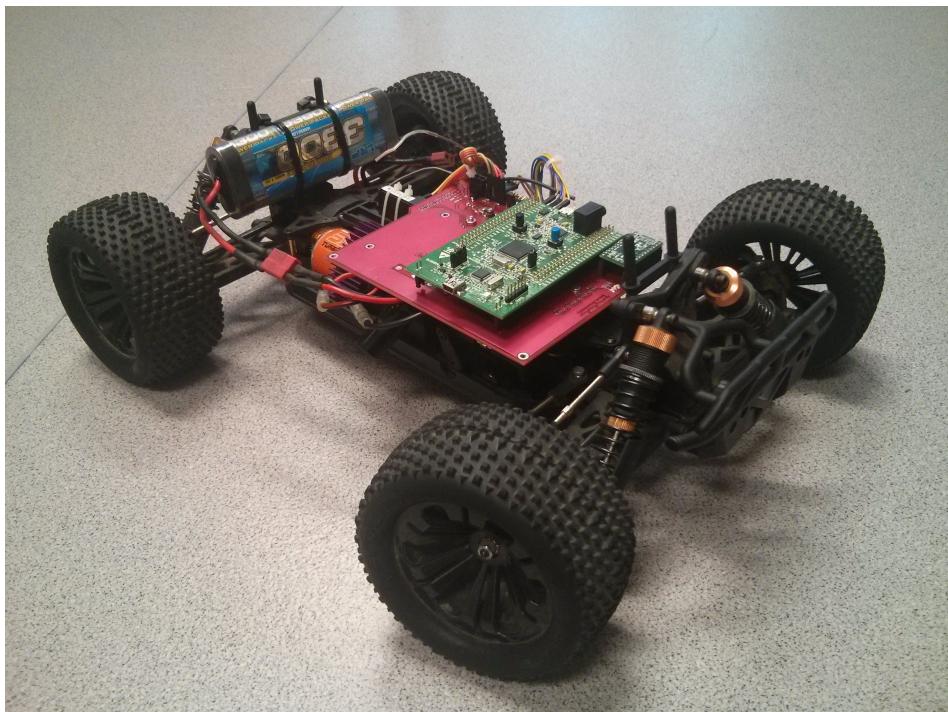
A magasabb szintű feladatok, mint például a pályatervezés és követés, már egy Raspberry PI mikroszámítógép feladata. Ezen a számítógépen működhet a robot felügyeleti szerve is, illetve ez teszi lehetővé vezeték nélküli hálózathoz való csatlakozást is.

6.2. Nehézségek

A robot fejlesztése és az algoritmusok valós roboton való végrehajtása közben számos problémával kellett megküzdenem, az alábbiakban bemutatom a legjelentősebbeket.

6.2.1. Helymeghatározás

Az eredeti tervekben a robot helymeghatározása, így a pályakövetés visszacsatolása egy kamerás látórendszer feladata lett volna. A kamera képfeldolgozásáért egy a RoboCup versenyre készült látórendszert használtam volna. Ennek a módszernek a pozíció meghatározási pontossági igen jó, de az orientáció meghatározása igen nagy hibával történik. Ezen felül, mivel a program kifejezetten a robotfoci pályához készült, kalibrálása igen nehézkes, és a mért adatok kinyerése is bonyolult lett volna.



6.2. ábra. Az elkészült robotautó

Egy későbbi tervben a tanszéken elkészült helymeghatározó rendszer használata merült fel. Ez a rendszer ultrahangos adó és vevőegységek segítségével végzi a pozíció meghatározását. Az orientáció mérése pedig optikai úton történik, az ultrahangos háromszögelő eljáráshoz nagyon hasonló módon. A robot hardver tervébe bekerült egy illesztő áramkör, és egy csatlakozó is, hogy ezzel a rendszerrel együtt lehessen használni. Ez a módszer igen jó pontossággal működik, és abszolút helymeghatározásra képes. Hátránya, hogy csak egy korlátozott területen belül képes az objektum érzékelésére. Végül mielőtt a robot műkö-

désbe lépett volna, a rendszer meghibásodott, és kijavítására nem jutott idő, illetve ez nem is tartozott a feladataim közé, így ezt a módszert is el kellett vetni.

6.2.2. Dead-reckoning

Végső megoldásként a rendelkezésre álló inkrementális adó és a giroszkóp adatait feldolgozva dead-reckoning segítségével határoztam meg a robot pozíóját. Ehhez az (1.2) egyenletet kell felhasználnunk. A pozíció meghatározásához szükségünk van az autó orientációjára (θ), és a sebességére (v_r). A sebességmérést az inkrementális adó segítségével végeztem el. Az adó felbontása megfelelő, az ebből fakadó hiba $1\frac{m}{s}$ sebességnél 1%-os, különbségképzés segítségével számolva.

Mivel a kormányszög mérésére nincs lehetőségünk, az orientációt egy giroszkóppal mérjük, a giroszkóp a szögsebesség mérésére képes. A szögsebességből egy egyszerű integrálás segítségével számítható az orientáció. Sajnos a giroszkóp IC paraméterei nem megfelelőek, így az általa szolgáltatott információt szűrnünk kell. Ehhez bekapcsoláskor egy egyszerű átlagolással kiszámíthatjuk a driftet, azaz a mért értékek konstans eltolódását. Majd az algoritmus futása közben egy szűrő algoritmus segítségével közvetlenül az orientációt kapjuk [19].

Mivel a dead-reckoning során integrálás segítségével számítunk pozíciót, így a maradó hiba időben növekvő mértékű lesz. Az esetünkben a giroszkóp hibája a jelentős, de ez az általam használt pályaméretek és követési idők esetén még elhanyagolható.

6.2.3. Kotyogás

A sebességszabályozás alacsony sebességek esetén nem megfelelően működik. A hibás működés oka ez erőátviteli rendszer kotyogása. Ez abból fakad, hogy a sebesség visszacsatolás a motorral szorosabb összeköttetésben lévő kardántengely segítségével történik, amely a motor elfordulásának hatására könnyen elfordul, de a kerekek még állva maradnak.

Ebben az esetben az inkrementális adó viszonylag nagy sebességet jelez, míg valós elmozdulás nem történik. A hirtelen indítás a PI szabályozó arányos tagjának következménye, mivel ilyenkor az ugrásszerű beavatkozójel nagy hibát okoz. A szabályozás a következő szabályozási ciklusban egy hasonlóan nagy, de ellenkező irányú beavatkozójellel próbálja a hibásan mért nagy sebességet ellensúlyozni. Ez a hatás oszcillációt okoz a szabályozásban.

A kotyogás a rendszer egy nemlineáris hatása, amit nehéz kiküszöbölni. Ipari eszközökben a hajtás lecserélése egyszerű fogaskerekekkel hullámhajtóműre megoldást jelent, de erre jelen esetben nincs lehetőség. A szabályozás szempontjából ezt több módon is lehet kezelni, de tökéletes megoldást nehéz találni.

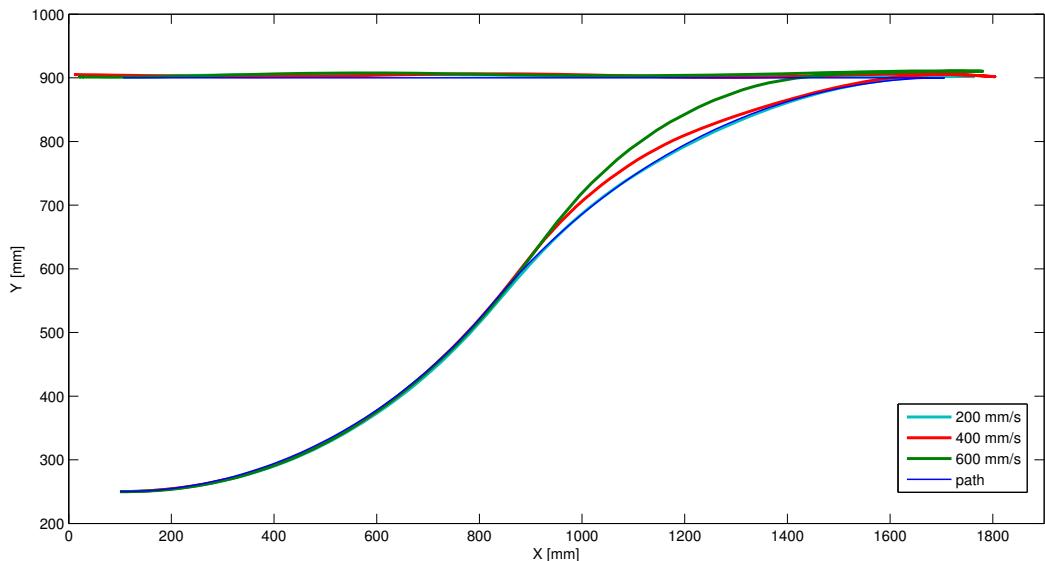
Az egyik lehetséges módszer a PI szabályozó hangolása oly módon, hogy az integráló hatást megnöveljük, míg az arányos tagot lecsökkentjük. A módszer behangolása igen időigényes, és nem is minden esetben érhető el vele a várt eredmény. Ezen felül rendelkezik egy egyértelmű hibával, a rendszer válaszának jellege erősen megváltozik, sok esetben lelassul, ami más problémákat hozhat elő.

Egy másik módszer az inkrementális adó sebességjelének szűrése. Ezt megtehetjük egy egyszerű alul-áteresztő szűrő segítségével, de ez jelentős holtidőt vihet a rendszerbe, vagy bonyolultabb szűrőalgoritmusokat is használhatunk, például egy Kalman-szűrőt, de ennek implementálása és hangolása idő és erőforrásigényes feladat.

Lehetséges a program módosítása, kiegészítése, például csak nemnegatív beavatkozójel kiadása, vagy alacsony sebességeken más szabályozó paraméterek használata, vagy a nemlinearitás figyelembe vétele, mint ahogy azt tettük a holtsáv kiküszöbölésekor, de a kotypogás egy ezzel ellentétes folyamat, ami megbonyolítja az implementálást.

6.3. Eredmények

A nehézségek ellenére összeállt egy kész rendszer, és jó eredmények születtek, de további finomításokra szükség van. A kotypogás kiküszöbölése jelenleg alacsony sebességen jelentősen megnöveli a sebességszabályzó beállási idejét, ezért jelentős a túlhaladás a pálya végén. A 6.3. ábrán látható ez a jelenség. Érdemes összehasonlítani a 4.6. ábrával.



6.3. ábra. Pályakövetés valós robotautóval különböző sebességeken

Látható, hogy minden előrehaladás esetén, minden tolatásnál jelentős a túlszaladás. Alacsony sebességen ($200 \frac{mm}{s}$) a vonalkövetés pontos, a grafikonon a pályával teljes átfedésben van, viszont a sebességet növelte a követés elromlik. Ez a kormányszervó lassú beállása miatt történik. A követésről készült videófelvétel is, ami megtalálható a dolgozatom mellékletében. A felvételen látható, hogy bár a szimulátorban nincs jelentős tévedés, a valóságban a robot elcsúszik a tervezett célponthoz képest. Ez a dead-reckoning hibájából, pontosabban az enkóder elcsúszásából fakad.

6.4. Jövőbeli tervezek

A jövőben szeretnék egy egységes programkönyvtárat kialakítani, ami megkönnyíteni a különböző pályatervező algoritmusok implementálását, azok eredményeinek összevetését. Ezen felül a valós roboton egy általános interfész kialakítása, mely bármely általános tervezőalgoritmus számára elérhetővé tenné a használatát, is céлом.

A dolgozatom elkészítése során több helyen is modelleket használtam. Ezeket a modelleket szeretném kibővíteni, finomítani, ahol ezzel jelentős eredményeket lehet elérni, például az időparaméterezés során újabb korlátozások figyelembe vétele, vagy dinamikai modell alkalmazása, vagy a tervezés és követés során a kormányszög beállási idejének figyelembe vételével.

Továbbá célom újabb robottípusok és algoritmusok megismerése, implementálása. A robotautó átalakítása, hogy a RobonAUT 2015 versenyen mint safety car működjön, és hosszabb távon szeretném, ha a tervező algoritmusokat a versenyen is használhatnák.

7. fejezet

Összegzés

Összegzésképpen elmondható, hogy a feladatban kitűzött céljaimat sikerült elérnem. Megismertem a robotikában leggyakrabban alkalmazott pályatervezési módszereket, majd implementáltam egy pályatervező algoritmust, mely autószerű robotok számára készít útvonalat, de differenciális robotok esetén is kiválóan alkalmazható. Az elkészült pálya nem csak távolságban, de időben is egyenletesen mintavételezhető az elkészült algoritmus segítségével. Implementáltam egy pályakövető algoritmust, ami lehetővé teszi a pálya stabil lekövetését, nem csak szimulátoron, hanem valós robottal is. Megismerkedtem a V-REP robotszimulátor programmal, amelyben elkészítettem egy autószerű robot modelljét. Az algoritmusok működését nem csak szimulátor, hanem valós robot segítségével is sikerült igazolnom.

Köszönetnyilvánítás

Szeretnék köszönetet mondani **Kiss Domokosnak** a folyamatos konzultációkért, tanácsokért és iránymutatásokért. Ezenkívül **Nagy Ákosnak** javaslataiért, tanácsaiért és a segítségért, valamint **Kolumbán Sándornak** a szabályozástechnikában nyújtott segítségeért.

Irodalomjegyzék

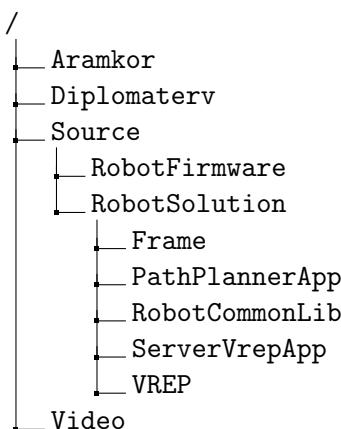
- [1] S. M. LaValle, *Planning Algorithms*. Cambridge, U.K.: Cambridge University Press, 2006. Available online at <http://planning.cs.uiuc.edu/>.
- [2] B. Siciliano and O. Khatib, *Handbook of Robotics*. Springer, 2008.
- [3] D. Kiss and G. Tevesz, „A model predictive navigation approach considering mobile robot shape and dynamics,” *Periodica Polytechnica - Electrical Engineering*, vol. 56, pp. 43–50, 2012.
- [4] D. Kiss and G. Tevesz, „A steering method for the kinematic car using C*CS paths,” in *Proceedings of the 2014 15th International Carpathian Control Conference (ICCC)*, (Velké Karlovice, Czech Republic), pp. 227–232, May 2013.
- [5] J. A. Reeds and L. A. Shepp, *Optimal paths for a car that goes both forward and backwards*. Pacific Journal of Mathematics, 1990.
- [6] B. Kornberger, „Fade2D - an easy to use delaunay triangulation library for C++.” <http://www.geom.at/fade2d/html/>, 2014.
- [7] G. Katona, A. Recski, and C. Szabó, *A számítástudomány alapjai*. Typotex, 2. javított kiadás ed., 2001.
- [8] J. Minguez and L. Montano, „Extending collision avoidance methods to consider the vehicle shape, kinematics, and dynamics of a mobile robot,” *IEEE Trans. Robot.*, vol. 25, pp. 367–381, 2009.
- [9] 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.
- [10] Ákos Nagy, *Pályatervezési és pályakövető szabályozási algoritmusok fejlesztése differenciális robothoz*. BME Automatizálási és Alkalmazott Informatikai Tanszék, 2014.
- [11] 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.

- [12] S. M. LaValle, „Rapidly-exploring random trees: A new tool for path planning,” tech. rep., Computer Science Dept., Iowa State University, 1998.
- [13] C. Sprunk, *Planning Motion Trajectories for Mobile Robots Using Splines*. Albert-Ludwigs-Universitat Freiburg, 2008.
- [14] D.-J. Kroon, „2D line curvature and normals.”
<http://www.mathworks.com/matlabcentral/fileexchange/32696-2d-line-curvature-and-normals/content/LineCurvature2D.m>, 2014.
- [15] I. Bézi, *Robotirányítás rendszertechnikája 3. fejezet*. BME Automatizálási és Alkalmazott Informatikai Tanszék, 2013.
- [16] K. Domokos and K. Sándor, *Segédlet a RoboNAUT verseny szabályozástechnikai szemináriumához*. BME Automatizálási és Alkalmazott Informatikai Tanszék, 2014.
- [17] B. Lantos, *Egy változós szabályozások*. Akadémiai Kiadó, 2009.
- [18] C. Robotics, „V-REP.” <http://www.coppeliarobotics.com/>.
- [19] S. Madgwick, A. Harrison, and R. Vaidyanathan, „Estimation of IMU and MARG orientation using a gradient descent algorithm,” in *Rehabilitation Robotics (ICORR), 2011 IEEE International Conference on*, pp. 1–7, June 2011.

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 könyvtárstruktúrában:



Aramkor:

Az autószerű robot vezérlőáramköre.

Diplomaterv:

Jelen dokumentum forrása.

Source:

Forráskódok.

RobotFirmware:

A robton található beágyazott rendszer programkódja.

RobotSolution:

A keretrendszer forráskódja, Nagy Ákossal 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.