Budapesti Műszaki és Gazdaságtudományi Egyetem
Villamosmérnöki és Informatikai Kar
Irányítástechnika és Informatika Tanszék

Mobilis robot ütközésmentes mozgástervezése LIDAR használatával

TDK dolgozat

Készítette:

Lévay Mátyás

Konzulens:

Gincsainé Dr. Szádeczky-Kardoss Emese

2020

Tartalomjegyzék

Kivonat i

Abstract ii

1. Bevezetés 1

2. Irodalmi módszerek 2
2.1. Akadálykerülési algoritmusok 2
2.1.1. APF 2
2.1.2. Bug 3
2.1.3. Velocity Space és Velocity Obstacle 4
2.1.4. Gap-alapú megközelítés 4

3. A kiindulási algoritmus 5
3.1. Robotmodell és koordinátarendszerek 5
3.2. Leírás 6
3.3. Implementálás 7
3.4. Eredmények 8
3.5. Hibák 8
3.6. Konklúzió 11

4. Az új robot modellje 12
4.1. Differenciális meghajtás modellje 12
4.2. LIDAR modell 13

5. Saját algoritmus I 16
5.1. Leírás 16
5.2. Implementálás 18
5.3. Eredmények 19
5.4. Futási idő összehasonlítás 22
5.5. Konklúzió 22

6. Tangent Bug algoritmus 24
6.1. Leírás 24
6.2. Módosítások 25
6.3. Implementálás 26
6.4. Eredmények 28
6.5. Konklúzió 30

7. Saját algoritmus II 32
7.1. Leírás 32
7.2. Implementálás 32

7.3. Eredmények 33
7.4. Dinamikus tulajdonságok 37
7.5. Futási idők és egyéb mérőszámok 39
7.6. Konklúzió 40

8. Összefoglalás **41**
8.1. A kész módszerek összevetése 41
8.2. Értékelés 41
8.3. Továbbfejlesztési lehetőségek 42

Köszönetnyilvánítás **43**

Irodalomjegyzék **44**

Kivonat

Az autonóm járművek fejlődésével érthető módon egyre fontosabbá válnak a robotok navigációját és akadályelkerülését megvalósító algoritmusok. Ezeknek biztonságosnak, megbízhatónak és gyorsnak kell lenniük, annak érdekében, hogy a haladásuk során ne ütközzenek, jussanak el a célba, és lehetőleg ezt minél gyorsabban és hatékonyabban tegyék, akár kis számítási kapacitás mellett is. Jelen TDK dolgozatom egy, ezen területen folyó kutatás eredménye.

Kiindulásképpen egy, a szakirodalomban fellelhető cikket választottunk ([5]), mely állítása szerint statikus és mobilis akadályokra, omnidirekcionális meghajtású robot feltételezésével képes ütközésmentes útvonalat tervezni. (Az omnidirekcionális jelentése az, hogy bármerre képes elmozdulni.) A munkám első lépéseként ezt az algoritmust tanulmányoztam, implementáltam és vizsgáltam, azzal a szándékkal, hogy a megismert algoritmust átültessem differenciális meghajtású, LIDAR érzékelővel ellátott robotra, és szükség esetén ezt továbbfejlesszem a felmerülő követelményeknek megfelelően. Munkám során MATLAB környezetben dolgoztam.

A kiindulási algoritmus implementálása során több olyan problémát is észrevettem, amelyek kizárják a megbízható működést, és ütközéshez vezetnek. Az első ilyen megfigyelés dinamikus környezet (mozgó akadályok) esetén merült fel, azonban konstruálható olyan statikus környezet, melyben szintén ütközni fog a járművünk. Ez, és az a megfigyelés, hogy az omnidirekcionalitásra alapozott pályatervezés nem volt kompatibilis a differenciális meghajtásból adódó kényszerekkel, vezetett el odáig, hogy elkezdtem kidolgozni egy saját algoritmust, amely nem feltételezi az omnidirekcionalitást, és biztonságos működést eredményez, a számítási igény számottevő növekedése nélkül. A saját algoritmus elsődleges célja a statikus környezetben való megbízható navigálás, kijavítva [5] hibáit, de az algoritmus dinamikus környezetben mutatott tulajdonságaira is kitérek a dolgozatomban.

Az algoritmusom alapgondolata az, hogy a robot haladási iránya alapján létrehozunk egy biztonsági zónát, és ha abban észlelünk akadályt, akkor változtatunk a haladási irányon, úgy, hogy elfordulunk a kisebb fordulást szükségessé tevő szabad út felé. Ha nem található akadály a biztonsági zónában, akkor viszont az ideális úton megyünk a cél irányába. Ezt az irányt az első megközelítés szerint dead-reckoning segítségével tartja nyilván a robot, mivel a kiindulási helyzetünk és a cél koordinátáit, illetve a meghajtószerveink elmozdulását ismerjük.

A munkám során tanulmányoztam más útvonaltervezési módszereket is, ezek közül legfőképpen a Bug-algoritmusok családját emelném ki, ugyanis e család egyik tagjával, az úgynevezett Tangent Bug algoritmussal közelebbről is megismerkedtem. Ezt szintén implementáltam, és átültettem differenciális meghajtású robotra. Ennek megfelelően összehasonlítom a két algoritmust, és a holtpont-probléma megoldására példát keresek az algoritmusom végső variánsának megalkotásához.

Abstract

With the development of autonomous vehicles, the navigation and the obstacle avoidance algorithms of these robots are increasingly important. These algorithms have to be secure, reliable, and quick to ensure the collision-free and efficient navigation to the goal, even with small computational capacity. This thesis summarizes the current developments of ongoing research in this field.

A published paper [5] was the starting point of the research. It claimed to be able to plan a collision-free path for an omnidirectional robot in a static or dynamic environment. (Omnidirectional means that the robot can start to move in an arbitrary direction at any moment.) As the first step of the research, I studied, implemented, and examined the algorithm of [5], intending to make it a viable option for a differential drive robot with a LIDAR sensor and further develop it according to the exact needs of the robot. I implemented the algorithm in the MATLAB environment.

During the implementation of [5], multiple problems emerged. Most of them lead to collisions, so their existence makes the algorithm unsafe. The first problem was noticed while examining the algorithm in a dynamic environment, but it is possible to construct a variety of static environments where our vehicle will collide with the obstacles. These facts, and the observation that the main idea behind this method is tied to the omnidirectionality, gave rise to the need for a novel algorithm. This algorithm does not assume an omnidirectional robot, but it is reliable while maintaining the same computational need. The primary objective of this novel algorithm is the navigation in a static environment (correcting the problems of [5]), but I will present the properties of the method in a dynamic environment too.

The fundamental idea of the novel algorithm is the creation of a safe zone. When the robot senses an obstacle inside this zone, it has to modify its course to the nearest collision-free path. If there is no obstacle in the safe zone, the robot takes the ideal route to the direction of the destination. In the first approach, the robot has the coordinates of the start and the goal points and computes its current location and orientation using dead reckoning.

During this research, I examined other path planning methods, such as the family of the Bug-algorithms. One of these is the Tangent Bug algorithm, which I implemented and modified to suit the features of a robot with differential drives. I compared the two algorithms and examined the Bug algorithm to solve a deadlock problem in the novel method.

1. fejezet

Bevezetés

Miért is szükséges a robotok mozgástervezése, és miért érdemes kutatni ezt a témát? Ha azt szeretnénk, hogy a robotok ne csak emberi tervezés és beavatkozás mellett tudjanak navigálni a környezetükben, akkor szükségünk van ilyen algoritmusokra. Ezeknek biztonságosnak, megbízhatónak és gyorsnak kell lenniük, abban az értelemben, hogy ne ütközzenek, jussanak el a célba, és lehetőleg minél gyorsabban tegyék ezt, minél kisebb számítási kapacitás mellett.

A felsorolt érveknek megfelelően a témával való ismerkedésem az [5] cikkel kezdődött. Ez állítása szerint kis számítási kapacitás mellett képes feldolgozni a statikus vagy dinamikus akadályokat tartalmazó környezetéből érkező jeleket, és megbízhatóan és biztonságosan képes elnavigálni a robotot a kiindulási pozícióból a célpozícióba. Mindezeket ráadásul úgy kivitelezi, hogy a globális útvonaltervezést a két különböző irányú mozgás alapján ötvözi a lokális akadályelkerüléssel, így a sikeres pályatervezés két különböző, de egyenlően fontos összetevőjét külön-külön is tetten lehet érni benne.

A kutatás célja kezdetben ennek a módszernek a megértése és implementálása, illetve a megismert eljárás differenciális meghajtású robotra való átültetése volt. Célként szerepelt még az is, hogy az új robotmodell LIDAR érzékelővel jusson információhoz a környezetéből. Később ez a célkitűzés az új eredmények hatására változott, de ezeket a változásokat mindig a megfelelő tartalmi résznél írom majd le.

A dolgozat felépítése a következő: a bevezető után szakirodalmi áttekintés következik, majd bemutatom a kiindulási algoritmust, illetve a meghatározott új robotmodell után az ezzel kompatibilis új algoritmust is. Sor kerül az úgynevezett *Tangent Bug* algoritmus bővebb ismertetésére, és végül a továbbfejlesztett saját algoritmus leírását olvashatjuk. Végezetül összegzem a dolgozatot, és áttekintem az elért eredményeket.

2. fejezet

Irodalmi módszerek

Robotok útvonaltervezésének és akadályelkerülésének tekintetében meglehetősen széleskörű szakirodalom áll már rendelkezésünkre, a téma fontosságának megfelelően. A kutatás során ezért úgy gondoltam, hogy érdemes lenne körülnézni a szakirodalomban a kiindulási cikken kívül is, hogy egyrészt egy átfogóbb képet szerezzek a területen folyó kutatásokról, illetve inspirációt nyerjek a felmerült problémák megoldásához. Ebben a fejezetben ennek az irodalomkutatásnak az összefoglalóját olvashatjuk.

Az irodalomkutatás többcélú volt: cél volt utánanézni a bevezetőben is említett LIDAR szenzor megfelelő kezelésének és az adatfeldolgozásának is, illetve annak, hogy milyen elterjedt algoritmusok léteznek az akadályok kikerülésére az [5] kiindulási algoritmuson kívül, és ezeknek mik az erősségei és gyengeségei.

Ennek ellenére tartalmi szempontból jobbnak tartom a LIDAR-hoz tartozó szakirodalmat nem most, hanem a 4.2. fejezetben leírni, így ez ott található.

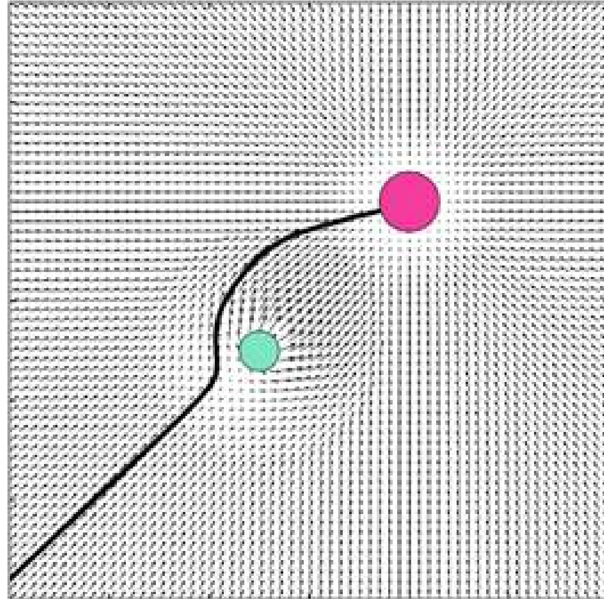
2.1. Akadálykerülési algoritmusok

Ebben az alfejezetben tömören bemutatok több különböző ütközés-elkerülési megközelítést, azokat forráshivatkozásokkal ellátva. A hangsúly főleg az adott metódusok erősségein és gyengeségein lesz, annak érdekében, hogy könnyen átlátható legyen a több különböző algoritmus.

2.1.1. APF

Az *APF*, vagyis *Artificial Potential Field* azon a gondolaton alapul, hogy ha a robotot egy olyan töltésként képzeljük el, amit a célpozíció helyére képzelt töltés vonz, illetve az akadályok helyére képzeltek taszítanak, akkor egyszerűen és elegánsan elég kiszámolnunk a robotra ható erők vektori eredőjét, és megkapjuk azt az irányt és nagyságot, amelyhez hasonlóan kell a robotunknak is mozognia. A módszer egy vizualizációja látható a 2.1. ábrán.

Ez a módszer - bár tényleg egyszerű implementálni - feltételezi a robot környezetének teljes ismeretét, illetve könnyen beragadhat egy lokális minimumba, vagy végezhet oszcilláló mozgást. (Az eredeti cikk: [7].) A probléma megoldására több ötlet is született, például a [2], ahol egy véletlenszerű sétát alkalmaztak a lokális minimumon, illetve az oszcilláción való túllendülésre. Később az APF módszer kiterjesztésre került dinamikus környezet esetére is: [8].



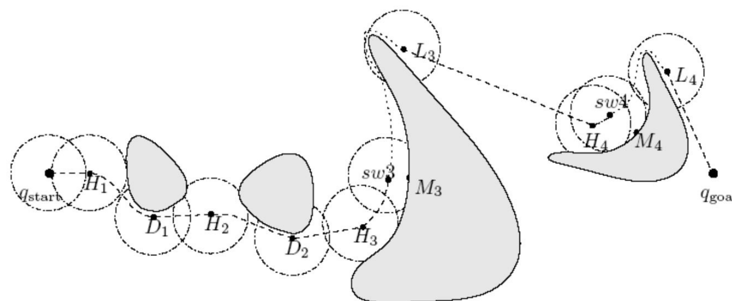
2.1. ábra. Szemléltető ábra az *APF* módszerről [11]

2.1.2. Bug

A másik híres és elterjedt módszer a *Bug-algoritmusok* családja, közülük pedig a *Tangent Bug*. Ennek alapgondolata az, hogy egészen addig mozogjunk a cél felé, amíg nem találkozunk akadállyal, egyébként pedig kövessük az akadály körvonalát addig, amíg újra mehetünk a cél felé. Természetesen nem csak akkor megyünk a cél irányába, ha arra nem látunk akadályt, hanem van ennek az eldöntésére egy konvergens kritériumunk (ha a látott akadályvégpontokhoz tartozó költségfüggvény nőni kezd, akkor követjük az akadályokat, hogy elkerüljük a lokális minimumokat, ha pedig a követés kezdeténél megfigyelt szint alá csökken, akkor újra a végpontok segítségével haladunk), és ennek megfelelően akkor váltunk a két viselkedési minta között.

Az említett költségfüggvényt úgy kell kiszámolni, hogy a kérdéses, az érzékelés határán lévő pont (tehát vagy egy akadálypont, vagy az adott irányban a legtávolabbi érzékelhető pont) a robottól vett távolságát összeadjuk a pont céltól vett távolságával.

Ezt a gondolatmenetet (az algoritmus működését és a költségfüggvény szemléltetését) bővebben kifejtem a 6.1-es fejezetben, de megtalálható a [6] és a [12] cikkben is. Egy szemléltető kép látható a működésről a 2.2. ábrán.



2.2. ábra. Szemléltető ábra a *Bug*-algoritmusról [3]

A módszer hátránya, hogy statikus környezetre próbálták csak ki, illetve a robotot pontszerűnek feltételezi. Mindemellett a fizikai megvalósítást nehezíti az, hogy erősen függ a végeredmény az alkalmazott szenzor pontosságától.

2.1.3. Velocity Space és Velocity Obstacle

A *Velocity Space* algoritmusok közül a *Dynamic Window Approach* vagy *DWA*-t említeném meg [4]. Ez az akadályok elkerülését egy, a sebességek terében elvégzendő feladattá alakítja, aminek a megoldásához figyelembe veszi a robot dinamikai leírását is, mint kényszerfeltételeket.

Segítségével viszonylag gyors és „sima” útvonalakat sikerül létrehozni, viszont ez a metódus szintén érzékeny a lokális minimumokra, és paramétereit hangolni kell.

Ehhez hasonlóan a *Velocity Obstacle* módszer lényege az, hogy a sebességtérben kerüli ki az akadályokat, előrejelezve és elkerülve a lehetséges ütközéseket, álló és mozgó akadályokra egyaránt.

Hátránya, hogy feltételezi a teljes környezet pontos ismeretét, és az, hogy a környezethez illő előrejelzési időléptéket kell mindig választani, és ez zsúfolt környezetben nehéz feladat. Előnye viszont a dinamikus és statikus akadályelkerülés, és az, hogy például több robot mozgástervezése is megoldható vele.

2.1.4. Gap-alapú megközelítés

A *gap* szó jelentése *rés*. Ebből sejthető, hogy ezek a módszerek főleg az akadályok közötti olyan térrel foglalkoznak, ahol a robot probléma nélkül elfér. Egy ilyen elven működő módszer például a *Nearness Diagram* [9], ami előre definiált helyzetek segítségével választja ki a követendő útvonalat.

A *Gap*-alapú algoritmusok és az *ND* előnye az, hogy ígéretes megoldási lehetőségeket nyújtanak (figyelembe véve a ráépülő további algoritmusok számát és minőségét), viszont hátrányuk, hogy szintén hajlamosak oszcilláló viselkedésre, és sokszor eltérülnek a nagyméretű szabad helyek felé, ami nem ideális útvonalat eredményez. Ennek a módszernek a továbbfejlesztése a [10], ami az irodalomkutatásom kiindulási pontját is képezte.

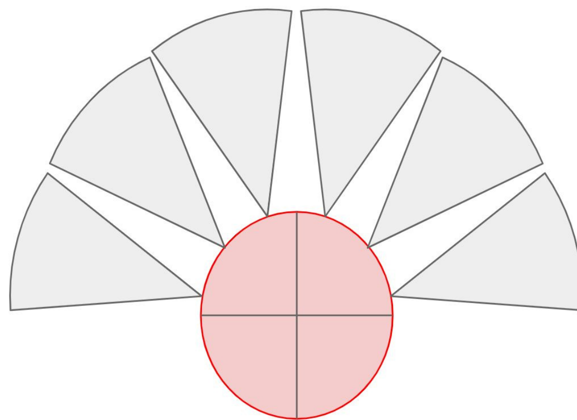
3. fejezet

A kiindulási algoritmus

Ebben a fejezetben leírom az [5] cikkben feltételezett robotmodellt, a fellelhető algoritmust, és ezek implementálását, és a futtatott szimulációkból származó eredményeket, hibákat és tanulságokat.

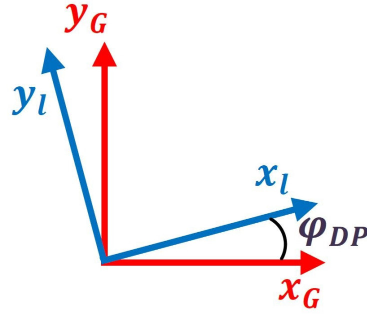
3.1. Robotmodell és koordinátarendszerek

Ebben a fejezetben egy kör alakú, omnidirekcionális robot számára tervezünk pályát. A robot a haladás irányában hat ultrahangos távolságérzékelővel rendelkezik, melyek egyenként 30°-os látószöggel bírnak, mint a 3.1. ábrán is látható. Az ábrán a robotot a piros kör, az elején szimmetrikusan elhelyezkedő szenzorok látótereit a szürke körcikkek szemléltetik. Fontos, de kimondatlan feltételezés az, hogy amennyiben két szomszédos ilyen zónában nincs akadály, akkor ott elfér a robot ütközés nélkül.



3.1. ábra. A robot elején lévő szenzorok látóterének illusztrációja

A pozíciókat tekintve ismertek a kiindulási hely és az elérendő cél koordinátái egy rögzített koordinátatérben. Az algoritmus végrehajtása során áttérünk lokálkoordinátákra, vagyis egy olyan vonatkoztatási rendszerre, amely origója a kezdeti rendszer origója, de a vízszintes tengely pozitív iránya egyirányú a robot kiindulási állapotából a célba mutató vektorral. Innentől - hacsak másképp nincs feltüntetve – ez a koordinátarendszer az alapértelmezett. Ezt az áttérést a 3.2. ábrán szemléltetem, ahol az x_l tengely egyirányú a robot kezdeti pozícióját a céllal összekötő egyenessel, így az áttérés lényegében egy forgatással megoldható.



3.2. ábra. Áttérés a globális és lokális koordináták között [5]

3.2. Leírás

A pályatervezési módszer szöveges leírása mellett a 3.3. ábrával is szemléltetem a végrehajtandó algoritmus azon részét, mely az Y irányú sebesség kialakításáért felel.

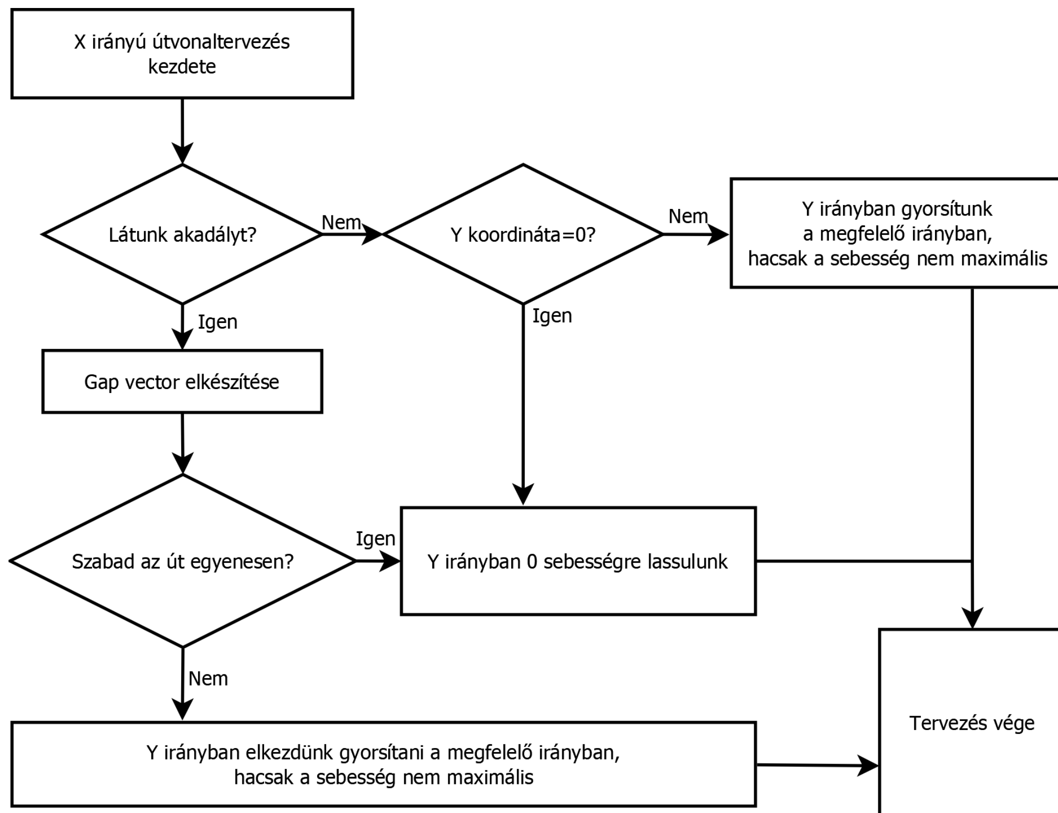
Az algoritmus fundamentuma az a feltételezés, hogy külön kezelhető az omnidirekcionalitás miatt az X és az Y irányú mozgás, és erre építve mondják azt a cikk szerzői, hogy az algoritmus globális megközelítésben közelít a célhoz, miközben lokálisan elkerüli az akadályokat. Ez annyit jelent, hogy az X irányú sebességet egy előre meghatározott módon számoljuk úgy, hogy ebben nem reagálunk a környezeti változásokra, viszont cserébe az Y irányú mozgást csak az akadályelkerülésre kell felhasználnunk. Sejthető, hogy ez így önmagában nem feltétlenül elegendő egy dinamikus környezetben, csak akkor, ha vannak bizonyos egyéb megkötéseink is, ez a cikkben az, hogy az akadályok maximális sebessége egyenlő vagy kisebb a robotunk maximális sebességénél, illetve annak hallgatólagos feltételezése, hogy csak kör alakú akadályok fordulhatnak elő a pályán. (Ezekre a feltevésekre még később kitérek.)

Minden időlépésben az érzékelők kimeneti adatait egy tömbbe (az úgynevezett *sense vector*-ba) rendezzük. Ez egy olyan hatelemű tömb, amely egyes elemei az 1 értéket veszik fel, ha van az ütközési távolságon belül akadály, egyébként pedig a nullát. Az ütközési távolság a maximális sebesség feltételezésével kiszámolható a legrosszabb esetben, így ezt használjuk. Ha bármikor ezen a távolságon belül érzékelünk akadályt, akkor a *gap vector*-nak nevezett tömböt konstruáljuk a *sense vector*-ból a következő módon: a *gap vector* k-adik eleme ($V_G[k]$) azt mutatja meg, hogy a *sense vector* k-adik és $k+1$-edik elemei ($V_S[k]$ illetve $V_S[k+1]$) mind nulla értékűek-e, ekkor 0, egyébként pedig 1 az értéke. Lényegében: $V_G[k] = V_S[k] \vee V_S[k+1]$

A bemutatott számolás mögött az az általam már említett feltételezés áll, hogy ha a robot bármelyik két szomszédos érzékelő által lefedett sávban nem lát akadályt, akkor ott biztosan elfér. Ez nem biztos, hogy teljesül bizonyos határesetekben (például ha túlságosan közel indul a robot az akadályokhoz), de többnyire feltehető. Szintén feltételeztük a kör alakú akadályokat, ez pedig akkor hasznos, ha a robot pályáját úgy szeretnénk vizsgálni, mint egy pont mozgását a síkon, ugyanis akkor a fizikai kiterjedés kezelhető úgy, hogy a robot sugarával nagyobbnak képzeljük el az összes akadály sugarát. Innentől ezt a módszert feltételezzük.

Ennyi bevezetés után térjünk rá a pályatervezésre! Az X irányú gyorsulás számítása másodfokú modellben történik: megszabhatjuk a gyorsulás nagyságát, és a megtett út, illetve a sebesség már ebből adódik. Igyekszünk a lehető legrövidebb idő alatt eljutni X koordináta tekintetében a célba, ennek két korlátja van: a robot sebességének és gyorsulásának is adott a maximális értéke.

Az akadályelkerülésért felelős Y irányú gyorsulás számítása is viszonylag egyszerű: ha van a látótérben akadály, akkor először meghatározzuk, hogy melyik a haladási irányhoz legközelebbi luk a *gap vector*-ban a középső, jobb oldali, bal oldali, távoli jobb és távoli bal oldali prioritási sorrendben, és úgy avatkozunk be a maximális Y irányú gyorsulással, hogy az először megtalált lyuk irányába mozduljon el a robot. Ha sehol sincs luk, akkor automatikusan balra indulunk el. Ha viszont nincs a látótérben akadály az adott iterációban, akkor visszatérünk a *desired path*-nak nevezett egyeneshez, amely a robot indulási pontját köti össze a végcéllal.



3.3. ábra. Az Y irányú sebességért felelős folyamat

Az olvasónak ekkor bizonyára feltűnik, hogy nem vettük figyelembe azt, hogy a robot célba ért-e. Erre az a magyarázat, hogy azt feltételezték a cikk szerzői, hogy amikor az időben fix hosszúságú X menti mozgás véget ér, akkor a robotnak Y irány szerint is a célban kell lennie. Ez általában működik, de könnyen konstruálható olyan eset, amikor a robot nem ér be a célba. Ilyen például a cél tetején álló akadály, vagy a *desired path*-ot keresztező, akadályokból álló vonal, amely a robot számára balra túlnyúlik a célba érés fix ideje alatt megtehető maximális Y irányú távolság felén. Ezektől az esetektől a szerzők eltekintettek (vagy nem vették észre őket), helyette olyan eseteket vizsgáltak, amikor a robot meg tudja tenni a célig vezető utat ezzel az algoritmussal.

3.3. Implementálás

Az algoritmus implementálását MATLAB-ban végeztem. A szimuláció struktúrája a következő: a főprogram kezdetén definiáljuk a teljes szimulációt befolyásoló paramétereket, például a szenzorok maximális érzékelési távolságát, és a célbaérés elfogadható numerikus hibáját, majd utána következik a robot és az akadályok létrehozása, és csak ezután következik maga a szimuláció magja, egy ciklus, amely akkor ér véget, ha kifut a szimuláció az

előre definiált maximális lépésszámból, vagy ha a robot célba ért. A cikluson belül minden iterációban feljegyezzük minden szereplő (robot vagy akadály) aktuális pozícióját, és az így feljegyzett adatok alapján tudjuk később szemléltetni a szimuláció eredményeit, az útvonalakat.

A ciklus magjában öt függvény kapott helyet, amiből három a robot és az akadályok útvonalának tervezésével, és kettő a tervezés végén a lépések végrehajtásával foglalkozik. Ezek közül a legérdekesebbek a robot útvonalát tervező függvények, amelyek az algoritmus leírásánál szereplő okok miatt az X és az Y irányban terveznek. Az X irányban tervezés a szimuláció szerint felettébb egyszerű, csak az állandó maximális gyorsulás mellett előálló egyenleteket kell leírni benne. Ezek bővebben megtalálhatóak az [5] cikkben, ezért nem ismétlem őket meg.

Az Y irányban némileg összetettebb a tervezés. Először képezzük az akadályok ismert pozíciója alapján előálló *sense vector*-t, majd ez szolgál az algoritmus bemeneteként. Ha látunk akadályt, akkor képezzük a *gap vector*-t, és ennek megfelelően teljes Y irányú gyorsulással indulunk el a legközelebbi szabad út felé. Ha nem látunk akadályt, akkor elindulunk vissza az $Y = 0$ egyenes irányába, kivéve, ha célba értünk. Emellett annyit még megjegyeznék, hogy a cikk eltér az eredetileg saját maga által felállított követelménytől, miszerint másodfokú közelítést alkalmaz a robot pályáját illetően, ugyanis az Y irányú pályatervezésben két helyen is szerepel olyan pont, amikor közvetlenül a robot sebességét változtatja annak gyorsulása helyett. Ebben eltértem a cikktől az implementálás során, és a kívánt állapot eléréséhez kiszámoltam a távolság megtételéhez szükséges idő, gyorsulás és sebesség paramétereket, és ennek megfelelően avatkoztam be.

3.4. Eredmények

Az implementált algoritmus futtatása során nyert szimulációk eredményei bíztatóak voltak, sikerült reprodukálni az [5] cikkben szereplő teszteseteket, és több, véletlenszerűen kitalált tesztesetre is működött az algoritmus, ezekről néhány ábrát mellékeltem ízelítőül, például a 3.4. ábra, ahol egy statikus akadályt kerül ki a robot, vagy a 3.5. ábra, ahol sok álló akadály között találja meg az utat a robot.

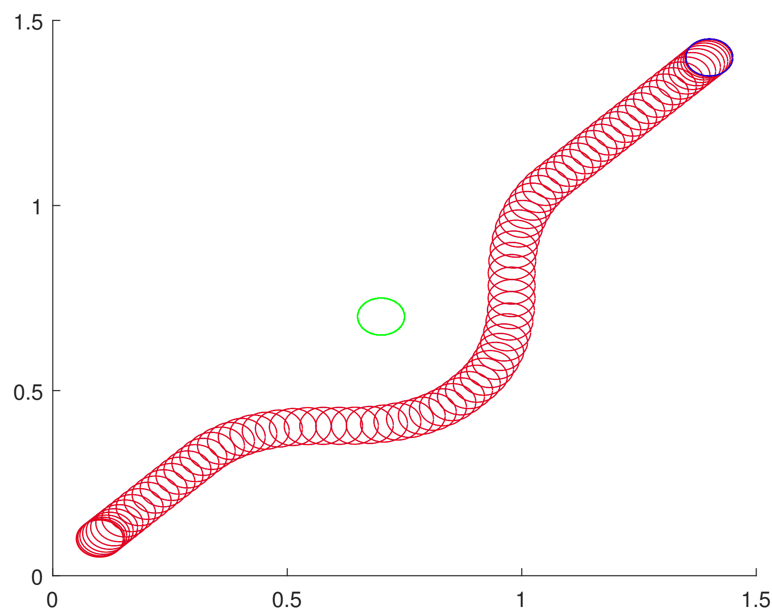
A képek jelmagyarázata a következő: a zöld körök jelzik az akadályokat, a piros körökből álló vonal pedig a robot útját. A robot az ábra bal alsó sarkából indul, és a cél pedig a piros vonal jobb felső végénél található kék kör. A tengelyek a világkoordinátarendszert jelölik, és méterben skálázottak.

Kipróbáltam mozgó akadályra is az algoritmust, és arra is sikerrel működött, a robot az útját keresztező jármű előtt sikeresen átment az útvonalon, lásd a 3.6. ábrát. Ez a teszteset azonban elgondolkoztatott, és közelebbi vizsgálat után arra jutottam, hogy a mozgó akadályok jelenlegi kezelése nem biztonságos. Ezt részletesebben a következő (3.5-ös) fejezetben fejtem ki.

3.5. Hibák

Az algoritmus menetéből adódóan következik, hogy ha a középső *gap* szabad, akkor nem kerüljük ki az akadályt, még akkor sem, ha az keresztezi az útvonalunkat. Amikor először egy ilyen, a 3.7. ábrán is megfigyelhető ütközést tapasztaltam, akkor arra gyanakodtam, hogy elrontottam valamilyen feltételt az algoritmus implementálása során, de rá kellett jöjjek arra, hogy a fentebb leírt jelenség következik be.

A 3.7. ábrával kapcsolatban szólnék még pár szót. Ezen a szimuláció látható 25 időlépésre bontva, a lépések pedig időrendi sorrendben balról jobbra és fentről lefelé követ-



3.4. ábra. Statikus akadály elkerülése

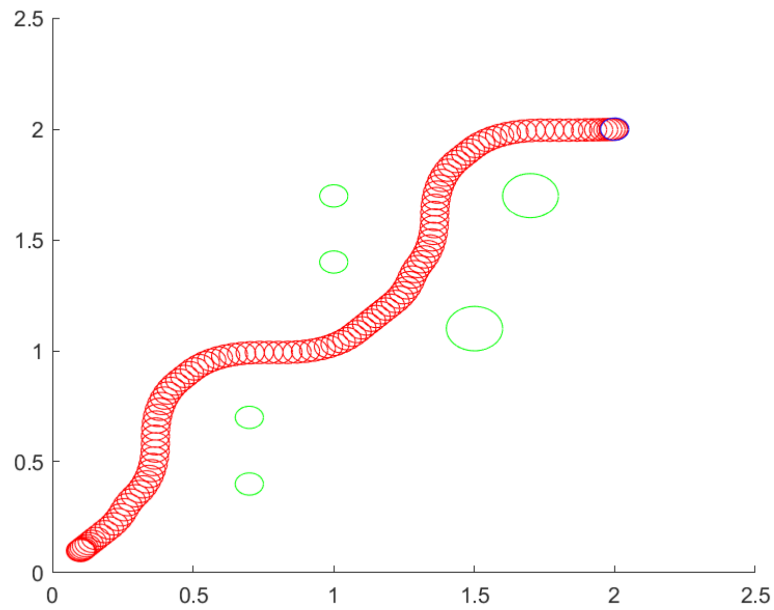
keznek egymás után. Ezen az ábrán jól látható, ahogy ütköznek az akadályok a megfelelő kiindulási feltételek mellett.

Az esemény tanulmányozása közben meggyőződtem arról, hogy a robot érzékelte az akadályt, és ahogy debugoltam, úgy vált világossá előttem a hiba lefolyása: a robot érzékeli az akadályt, de mivel az nem lép be a középső *gap*-be, ezért nem foglalkozik vele. Az akadály sebessége nem lépi túl az előírt maximális nagyságot ($v_{obstacleMAX} \leq v_{robotMAX}$), egyszerűen egy olyan esetről van szó, amely kezelésére nem gondoltak az algoritmus készítői. Ez egy probléma az algoritmus alkalmazhatósága szempontjából, ugyanis a robot így könnyen ütközés áldozatává válhat ezen a „vakfolton" keresztül.

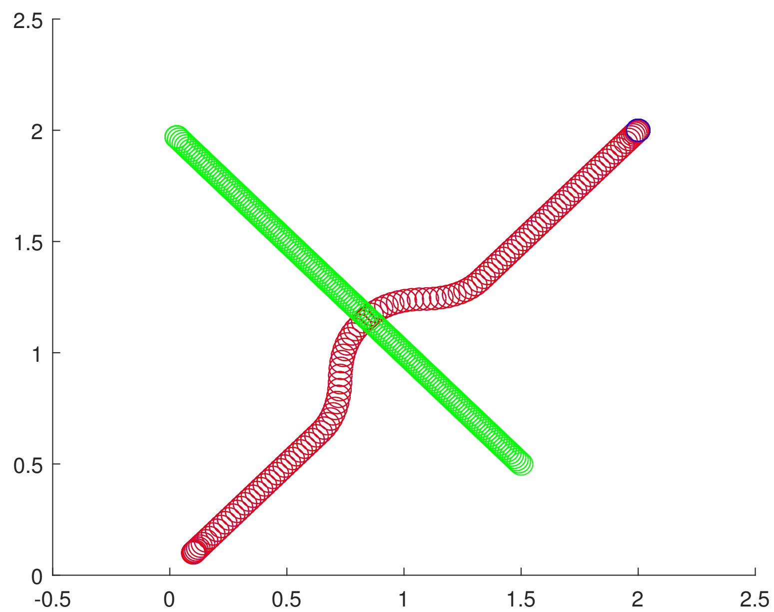
Az említett mellett még van több apróbb probléma is, ami miatt nem ajánlanám a módosítatlan algoritmust használni egy valós roboton. Például ilyen a kör alakú akadályok feltételezése, azzal a kimondatlan mellékes információval, hogy sok kis kör alakú akadály nem alkothat egy nagyobb struktúrát. Ez ahhoz vezethet, hogy például az ideális útvonalra merőlegesen egy akkora vonalat alkotunk akadályokból, hogy a robot a lokálisan X irányú útvonal megtételéhez szükséges időben ne tudja megkerülni lokális Y irányban azt. Ez az elrendezés pedig ütközéshez vezet, mert az X irányú tervezést nem tudja befolyásolni az Y irányú. Hasonlóan könnyen ki lehet alakítani akár egy „zsákot" formázó, vagy egyéb félkör alakú akadályhalmazt, ahol a robot lokális minimumba jut, ha a félkör szélei az érzékelési távolságán kívül vannak.

Érdekes még átgondolni azt a lehetőséget is, hogy mit csinál az algoritmus, ha a célon túl, de látótávolságban egy akadályt helyezünk el. Ebben az esetben, bár a cél kerülés nélkül elérhető lenne, a robot mégis elkezdi kikerülni az akadályt, és így nem éri el a célt.

Az utolsó technikai észrevételem az volt, hogy a metódus kimondatlanul feltételezte azt, hogy a robot átfér egy résen, ha a két középső szenzor egyszerre átlát rajta. Természetesen ez nem feltétlenül van így, de kellően nagy érzékelési távolság esetén ez teljesül.



3.5. ábra. Több álló akadály elkerülése

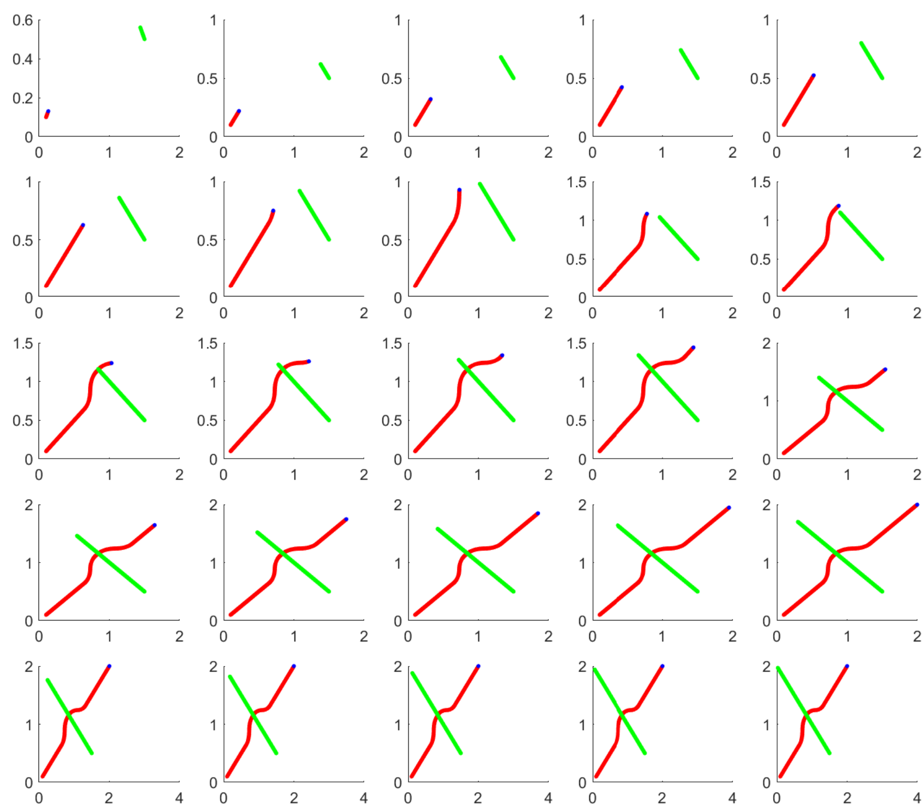


3.6. ábra. Mozgó akadály elkerülése

A „kellően nagy érzékelési távolság” minimálisan a következőt jelenti:

$$r_{sense} > \frac{2}{\sqrt{3}} r_{robot},$$

ahol r_{sense} a maximális érzékelési távolság, r_{robot} pedig a robot köré írható kör sugara.



3.7. ábra. Mozgó akadállyal való ütközés

3.6. Konklúzió

Az algoritmus álló akadályokra, és az azokra vonatkozó megszorításokkal működik, viszont a vakfoltos ütközések miatt semmiképpen nem biztonságos ebben a formában mozgó akadályok kezelésére. Álló akadályok esetén viszont annyi megkötéssel és szigorítással alkalmazható, hogy nem használnám ezt az algoritmust teljesen önálló, felügyelet nélküli pályatervezésre.

A leírtak miatt felmerült az ötlet arra, hogy kijavítva az algoritmus hibáit, egy saját algoritmust készítsek. Ezt a célt a szemem előtt tartva viszont előbb a meglévő algoritmust kellett differenciális meghajtású robotra átültetni, ugyanis az omnidirekcionális meghajtás nagyon speciális robotot igényelt volna, és a tanszéken elérhető lett volna számomra egy differenciális meghajtású robot. Sajnos ez a lehetőség a tavasztól hazánkban fennálló egészségügyi helyzet miatt meghiúsult, de a feladat továbbra sem változott, így szimulációs szinten megvalósítottam az említett átültetést, de erről bővebben az 5. fejezetben számolok be.

4. fejezet

Az új robot modellje

Ebben a fejezetben leírom az eddigi omnidirekcionálisnak feltételezett és hat darab ultrahangos távolságmérővel ellátott robotmodellhez képest az új algoritmus robotmodelljének változásait, mind annak meghajtását, mind érzékelőjét tekintve.

4.1. Differenciális meghajtás modellje

Az a feltételezés, hogy a robot kör alakú, nem változott, hiszen ha nem is lenne kör alakú a robot, azt kezelhetjük úgy, hogy a robottestnek a köré írható körét tekintjük a robot alaprajzának. Ennek ellenére viszont a differenciális meghajtás feltételezésével már nagyobb szerepet kapott a robot elejének kitüntetett volta, ugyanis az nem képes késlekedés nélkül bármerre elmozdulni álló helyzetből.

Mint a 4.1. ábrán is látható, a differenciális meghajtást a robot oldalán elhelyezkedő két kerék segítségével modellezzük. Ezek a robot középpontjától egyaránt b távolságra vannak. Ennek a középpontnak a koordinátái szolgálnak a robot koordinátáiként (jelölésük (x, y)), és a kerekek által létrehozott ω szögsebesség és a robot v sebessége is erre a pontra értelmezett. A robot középpontjától a kerekekkel párhuzamos, a robothoz képest az előre irány szerint felvett félegyenes és az X tengely pozitív fele által bezárt szög pedig a robot θ orientációja. A két kerék szögsebességeit külön-külön is értelmezzük (ω_R és ω_L), viszont a sugaruk (r) azonos.

A fentebb leírt meghajtásból következően a robotra érvényes mozgásegyenletek a következők:

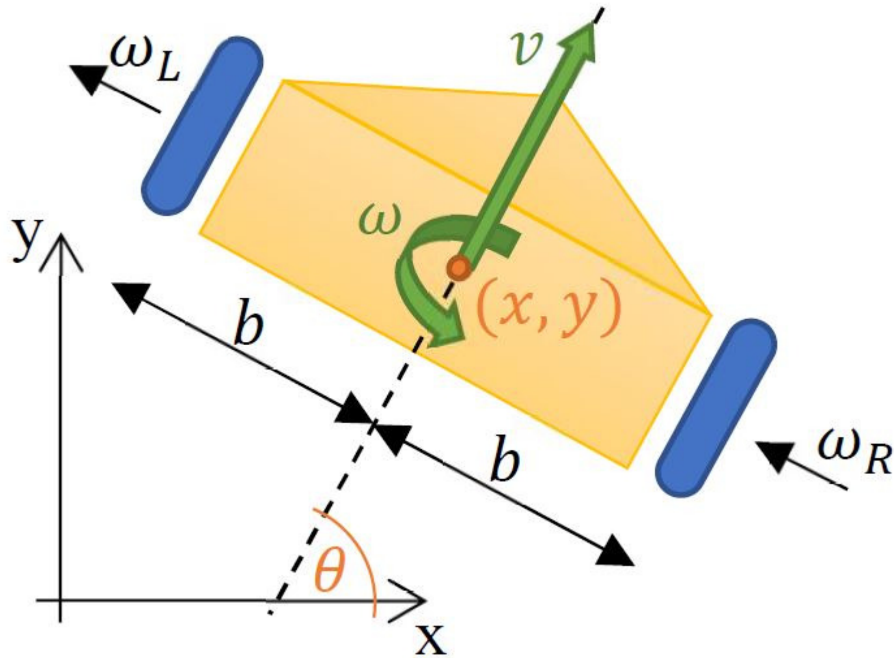
$$v = \frac{\omega_R + \omega_L}{2} r \tag{4.1}$$

$$\omega = \frac{\omega_R - \omega_L}{2b} r \tag{4.2}$$

$$\begin{pmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \end{pmatrix} = \begin{pmatrix} v\ cos\theta \\ v\ sin\theta \\ \omega \end{pmatrix} \tag{4.3}$$

Ezekből az egyenletekből jól megfigyelhető a robot x és y irányú mozgása közötti összefüggés. Ezt látva realizáltam, hogy az omnidirekcionalitásról a differenciális meghajtásra való áttérés nem csupán egy olyan függvény létrehozását jelenti, amely a kiszámolt omnidirekcionalitásra megfelelő beavatkozójeleket átszámolja az új meghajtás szerinti beavatkozójelekre, hiszen a két irány nem választható már szét ilyen módon. Ez ahhoz vezet tehát, hogy át kell dolgozni az irányválasztási stratégiát.

Érdemesnek tartom még megemlíteni a dolgozat absztraktjában is említett *dead reckoning* módszert. Ez annak a módszernek a neve, amely arra a feltevésre építve tartja



4.1. ábra. Differenciális meghajtású robot modellje

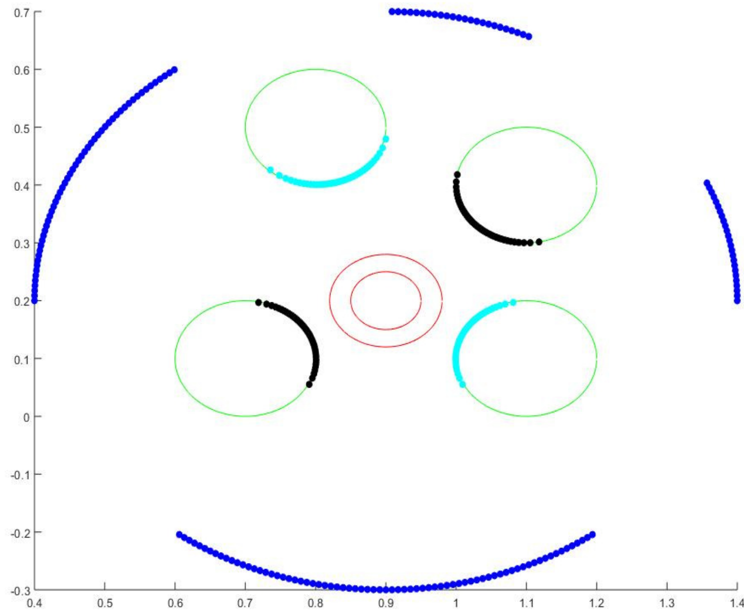
nyilván a robot pozícióját a globális koordinátatérben, hogy a robot meghajtószervei (egyszerűség kedvéért a kerekei) nem csúsznak meg, hanem tisztán gördülnek. Ekkor pedig a beavatkozójelek ismeretében kiszámolható a robot helyzete.

4.2. LIDAR modell

A LIDAR (*Light Detection and Ranging*) egy olyan szenzor, amely a körülötte lévő tér érzékelésének feladatát látja el úgy, hogy fénynyalábokat bocsát ki magából, és a visszaérkezés idejéből kiszámolja az adott irányban lévő legközelebbi tereptárgy távolságát [1]. Ezt folyamatosan körbeforogva ismétli, és így a szenzor teljes, alapvető esetben kétdimenzionális környezetéről információt szolgáltat.

Ezt az információszerzést az algoritmus kidolgozásához szintén szimulálnom kellett. A LIDAR-modellt úgy készítettem el, hogy később könnyen helyettesíthető legyen a programban egy valódi szenzor kimeneti adataival. Ez kézenfekvő módon azt jelenti, hogy a robot középpontjában elképzelt érzékelő fokonkénti bontásban adja vissza a pálya térképe alapján generált adatokat. Erről a szimulációról egy szemléltető képet (a 4.2. ábrát) is elhelyeztem a dokumentumban, ahol az azonos színű pontok az egy akadálynak felfogott tereptárgyakhoz tartozó pontok. Ahol a szenzor nem lát akadályt, ott a visszatérési értéket a maximális mérési távolságnak rögzítettem. Természetesen ez az érték a szimuláció előtt beállítható.

A LIDAR adatainak kezelése szintén összetettebb feladat, mint a hat ultrahangos távolságmérőé, azonban segítségével pontosabb és átgondoltabb stratégiák alakíthatóak ki, mint amik pusztán a bináris, igen-nem jellegű információból lehetségesek lennének. Ennek megfelelően az algoritmus beavatkozási logikája is átdolgozásra szorult a rendelkezésre álló mérések alapján. Az első felmerülő kérdés az, hogy mitől lesz a LIDAR által szolgáltatott pontok halmaza egy, vagy több akadály képe, és hogyan különböztethetőek meg a takarásban álló akadályok egymástól?



4.2. ábra. A LIDAR szenzor szimulációja

Először vegyük figyelembe azt, hogy mitől lesz egy tárgy az emberi szem számára is egy: attól, hogy szerkezetileg folytonos, nem férünk át rajta. Ez kézenfekvő módon azt jelenti, hogy ha két szomszédos mérési pont között érzékelt távolság kisebb, mintsem a robot átmérője, akkor könnyedén tekinthetjük a két mérési pontot egy akadályhoz tartozónak, azonban ha ez a távolság nagyobb, akkor valószínűleg két különböző, de a nézőpontunkból takarásban álló tárgyról van szó. Hasonlóan ha a két pont közül az egyik akadályhoz tartozó pont, a másik azonban a háttér pontja (vagyis a maximálisan érzékelhető távolságot kapjuk vissza róla), akkor az egy akadály szélső pontjának tekinthető. Egészen addig, amíg pontszerű robottal dolgozunk, addig ez a megközelítés elég, azonban mi történik akkor, ha a robotot kiterjedtnek vesszük, és a pályáról alkotott előzetes tudásunkat (például azt, hogy kör alakú akadályokat használtunk), nem használjuk ki az általános működés biztosítása érdekében? Ekkor kénytelenek vagyunk észrevenni azt, hogy ha két akadály közelebb van egymáshoz, mint az a hely, ahol a robot már elférne, akkor célravezetőbb egy akadályként kezelni a kettőt, még úgy is, ha közöttük adott esetben egészen a látóhatárig szabad lenne az út. Ezt a megközelítést először a 6. fejezet algoritmusához alkalmaztam, mert ott ez a meglátás elengedhetetlenül fontossá vált a helyes működéshez.

A LIDAR bevezetésekor utánanéztem a szakirodalomban is a szenzornak, illetve a kezelésének. Kiindulási alapként a [10] cikkben leírtakat vettem figyelembe, annak ellenére, hogy itt nem kifejezetten LIDAR-t használtak, hanem az általánosság megtartásához csak annyit feltételeztek, hogy a szenzor által kinyert adatok mélységi mérésként legyenek figyelembe vehetők. Ezt jobban végiggondolva pontosan a LIDAR által szolgáltatott pontfelhőnek megfelelő adatokból indultak ki.

Mint már ebben a fejezetben szó volt róla, ebben a cikkben is fontosnak tartották a mérési adatokból az egybefüggő akadályokhoz tartozókat csoportosítani, és ráadásul ezt hasonlóan végezték el, mint ahogyan ez általam is leírásra került. Folytonosnak ítéltek két szomszédos mérési pont közötti akadályrészt, ha a közöttük lévő euklideuszi távolság kisebb volt, mint a robot átmérője (vagyis közöttük a robot akkor sem férhet el, ha különböző akadályok), és mindketten akadálypontok voltak, tehát nem a maximális érzékelési

távolságot adta vissza a szenzor egyik pont esetében sem. Ha ezek a feltételek fennálltak, akkor a két pontot egy akadály két szomszédos pontjának tekintették. Amennyiben pontosan egy feltétel teljesült (csak az egyik akadálypont, vagy mindegyik az, de a közöttük érzékelt távolság elegendően nagy), akkor azon a helyen feljegyeztek egy úgynevezett *mélységi folytonossági hiány*-t, és később ezek közül válogatták ki a lehetséges réseket (a továbbiakban *gap*), ahol a robot átfért.

Megvizsgáltam még további, LIDAR alkalmazásával készült cikkeket is, azonban ezek nagyrészt könyvtári függvényeket használtak, vagy megvalósíthatósági tanulmányok voltak, mint például [1], így a kezelés módjában nem szolgáltak új információval. Ennek ellenére azonban érdemesnek tartom megemlíteni a [13] cikket, melyben az érdemi elgondolás az, hogy a LIDAR szenzor kétdimenziós pontos távolsági adatait ötvözzék egy kamera képével, és így azon akadályok detektálására is képessé válik a robot, melyek nincsenek a LIDAR érzékelési magasságában, hanem fölötte „nyúlnak” be a robot előtti térrészbe, mint például egy sorompó, aminek a LIDAR csak a két tartóoszlopát látja.

5. fejezet

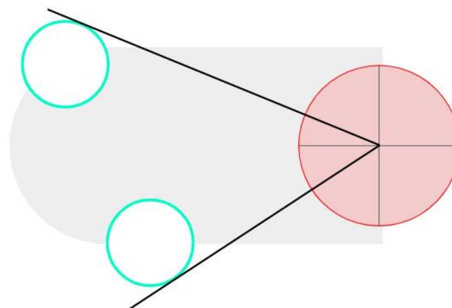
Saját algoritmus I

Ebben a fejezetben bemutatom a differenciális robot irányítására kigondolt első működő algoritmusomat. Ez az előző fejezetben (4.) leírt robotmodellt feltételezi, és ennek megfelelően alakítottam ki a pályatervezést.

Fontosnak tartom megjegyezni, hogy az [5] cikk implementálása során szerzett tapasztalatok fényében célravezetőnek gondoltam először csak statikus akadályokkal foglalkozni, és amikor változatlan környezetben teljesíti az algoritmus a vele szemben támasztott követelményeket (megbízható, biztonságos, és gyors működés), csak utána kiterjeszteni a tesztelést és az eljárást a dinamikus környezet által nyújtott kihívásokra.

5.1. Leírás

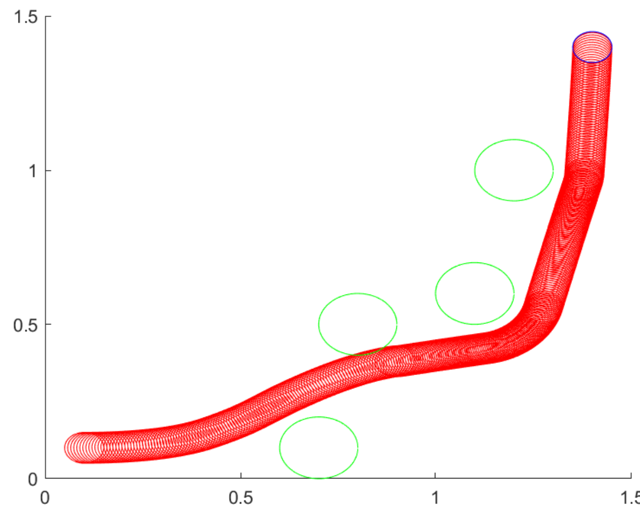
Az alapvető gondolat a pályatervezés mögött az, hogy biztonságosan kell célba érni, tehát a lehető legbiztonságosabb működés előnyt élvez a gyors útvonallal szemben. Ez a megfontolás, és az [5] cikk nyújtotta tapasztalat (konkrétabban leírva az abban kimondatlanul is szereplő biztonsági zóna) vezettek el addig, hogy érdemes lenne a LIDAR esetében is egy biztonsági zónát definiálni. A megalkotott biztonsági zóna hasonló meggondolást követ, mint az előző implementációban látott, vagyis az az érzékelt terület a biztonsági zóna, ahol ütközés következne be egyhelyben álló akadályokat feltételezve, ha a robot nem változtatna a mozgásán. Ez rövid meggondolás után egyszerűen felrajzolható, lásd az 5.1. ábrán szürkével jelölt részt. A robotot a piros kör szemlélteti, illetve két türkiz körvonalú, fehér körrel ábrázolt akadály is látható az ábrán. A két fekete vonal a pályatervezéshez hasznos, ezek adják majd a fordulási szöget. A leírás alapján a biztonsági zónának előrefelé sosem lenne vége, azonban a valóságban (és a szimulációban is) a szenzor maximális érzékelési távolsága korlátozza a zóna méretét.



5.1. ábra. A biztonsági zóna szemléltetése (az ábrán szürkével látható)

Amennyiben a biztonsági zónában nem látunk akadályt, úgy könnyű dolgunk van: egyenes útvonalon tartunk a cél felé, vagy éppen visszafordulunk erre az útvonalra. (Az akadály itt használt definíciójának megértéséhez lásd a 4.2. fejezetet.) Ha viszont akadály kerül az utunkba, akkor ezen változtatni kell, amennyiben az akadály közelebbi, mint a cél pozíciója. Ekkor pedig a robot megvizsgálja, hogy a jelenlegi haladási irányhoz képest merre érdemes fordulni a lehető legkisebb fordulás érdekében, vagyis hogy az akadály érzékelt végpontjai közül melyik látszik az egyeneshez képest kisebb szög alatt.

Az ütközés sikeres elhárításának érdekében a szükséges három lépés közül kettőt már a szélső pontok meghatározásával meg is tettünk, vagyis felismertük az ütközés tényét és döntési alternatívákhoz jutottunk. A harmadik lépést részben elvégeztük: döntés született a követendő irányról. Ez azonban még nem jelent teljes megoldást, mert a követendő kanyar ívét is meg kell határozni, ebből pedig a végső beavatkozójeleket.



5.2. ábra. Az egymást fedő akadályok kezdetben rosszul kezelt esete

Kézenfekvő ötlet lenne az ütközési végpont távolságára szabályozni a kanyar ívét, azonban ez veszélyes működéshez vezethet: ha két akadályt látunk félig fedő helyzetben (az 5.2. ábrán a $[0.8; 0.5]$ és a $[1.1; 0.6]$), és a távolabbihoz kevesebbet kell fordulnunk, akkor ha ahhoz szabályozunk, akkor előfordulhat, hogy ütközünk a közelebbi (a $[0.8; 0.5]$) akadállyal. Erre kezdetben nem figyeltem, de később észrevettem ezt a figyelmetlenséget, és korrigáltam arra, hogy a legközelebbi látott akadályponthoz képest számoljuk ki a fordulási sugarat. Ezt első közelítésben a következő módon tettem:

$$R_{max} = \frac{d^2 - c^2}{2 * c} \tag{5.1}$$

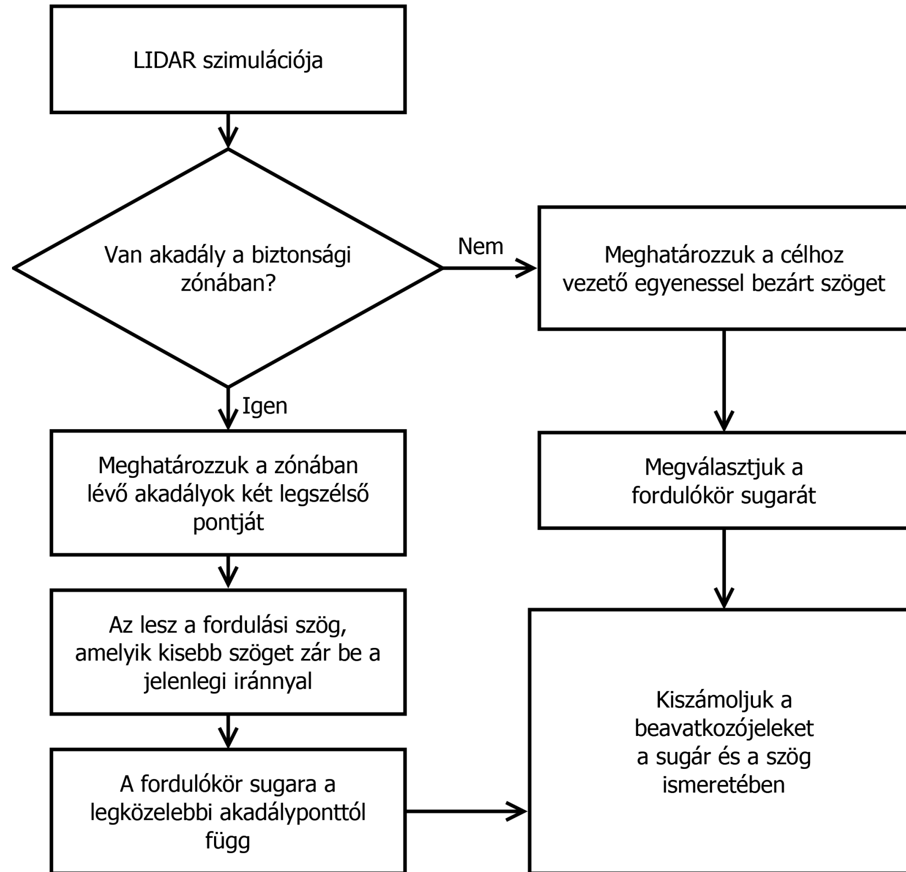
Ahol d az ütközési távolság, c a robot sugarának és egy biztonsági távolságnak az összege, R_{max} pedig a maximális fordulási sugár. A formula levezetésétől eltekintek, de azt bemutatom, hogy hogyan kapható meg R értékének megválasztása után a két beavatkozójel (ω_L és ω_R) nagysága, ha v_{max} jelöli a robot maximális sebességét. Ez a következő módon vezethető le:

$$R_{max} = \frac{v_{max}}{\omega} = \frac{b(\omega_R + \omega_L)}{\omega_R - \omega_L} \tag{5.2}$$

Innen pedig az ω_R és az ω_L közül a nagyobbik szögsebesség $\omega_{nagyobb} = \omega_{MAX}$ választással adódik, és ebből a kisebbik szögsebességű kerék szögsebessége is számolható. Érdekes, bár természetes észrevétel az, hogy R a fordulás irányának megfelelően előjeles, de ahogy értéke b-t közelíti, úgy kezd el a robot egyre inkább elmozdulás nélkül forogni a középpontja körül.

Jelenleg a robot általában körívek darabjain mozog, ám a teljesség kedvéért megemlítendő, hogy egyenesen előre haladás esetén természetesen mindkét kerék azonos, ideális esetben maximális meghajtása a cél.

A könnyebb átláthatóság kedvéért készítettem egy folyamatábrát is az eljárás logikai magjáról, ez az 5.3. ábrán látható.



5.3. ábra. Az algoritmus szemléltetése

5.2. Implementálás

Az implementáció értelemszerűen az algoritmus leírását követi, de hasonlóan, mint a 3.3 fejezetben, itt is a keretrendszer az inicializáció utáni fő ciklusból áll, amely addig fut, amíg a szimuláció eljut a maximális definiált lépésszámig, vagy a robot célba éréséig.

A fő ciklusban itt négy függvény kapott helyet, ezekből kettő-kettő a robot illetve az akadályok útvonalának tervezésével, illetve a pályán való léptetésével foglalkozik. A robot útvonalának tervezése ebben az esetben viszont már nem volt két külön irányra választható, ezért a LIDAR bemeneti adatainak szimulálása után lévő két függvény közül az első a megfelelő fordulási szög kiszámításáról, míg a másik a kiszámolt szög alapján a megfelelő beavatkozójelek előállításáról gondoskodik.

A fordulási szög kiszámításához először az érzékelt adatok alapján eldöntjük, hogy a robot biztonsági zónájában van-e akadály. Ez úgy történik, hogy kiszámoljuk az adott

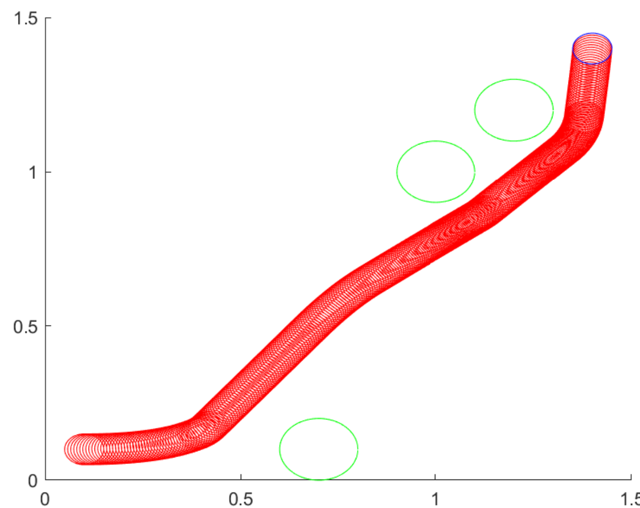
szögben a legkisebb biztonságos távolságot (a biztonsági zónával vett metszéspontot), és összehasonlítjuk a LIDAR-tól az adott szög alatt beérkezett adattal. Ha ütközést érzékelünk, akkor ezt a tényt, illetve az ütközés adatait (szög és távolság) feljegyezzük, amennyiben az nincs messzebb, mint a cél.

Ezután azon két szélső szög meghatározása következik, amelyeknél már nem látunk akadályt a biztonsági zónában. Ha ezek is rendelkezésre állnak, akkor készen vagyunk a fordulási szög kiszámításával, hiszen amennyiben van akadály előttünk, annyiban ezen szögek közül a kisebb abszolútértékűt választjuk, amenyiben pedig nincs, úgy a szögbeli eltérést a cél irányához képest.

Miután ezt meghatároztuk, az előző fejezetben (5.1.) leírtak alapján választjuk ki a beavatkozójelek nagyságát, és a fordulás irányának megfelelően azok előjelét, majd utolsó lépésként alkalmazzuk a kiszámoltakat, és a robotot léptetjük a pályán. Mivel csúszás nélküli gördülést feltételezünk a meghajtásnál, ezért feltételezhetjük, hogy a robot sikeresen el is ér ebbe a pontba, így ezt az információt a robot eltárolja a következő időlépésig (tehát *dead reckoning* módszert használunk).

5.3. Eredmények

Az algoritmus implementálása az ebben a szakaszban bemutatott eredményeket nyújtotta a szimulációk során. Az ábrák továbbra is a mozgás felülnézeti képét ábrázolják, a színhasználat változatlan (az akadályok zöld, a robot pedig piros színnel került ábrázolásra). A robot az ábra bal alsó sarkából indul, az X tengely pozitív feléhez képest bezárt nulla orientációs szöggel. A tengelyek skálázása méterben értendő.

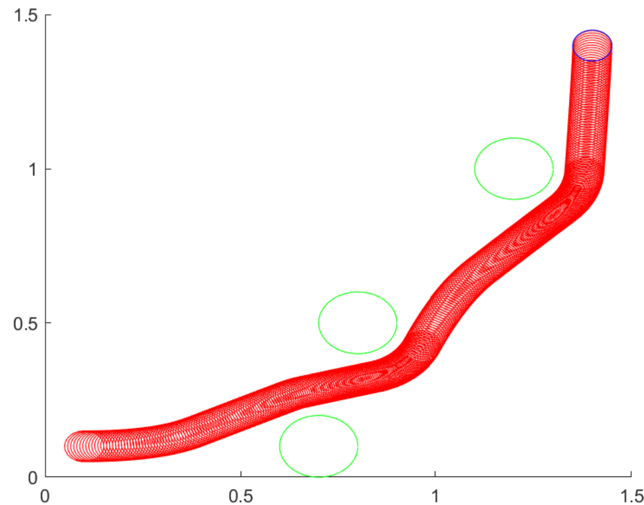


5.4. ábra. Statikus akadályok elkerülése

Az 5.4. ábrán a robot kiinduláskor már érzékelte az első akadályt, így annak a bal széle felé kanyarodott. Amikor ez kikerült a potenciális ütközési zónából, akkor elfordult a cél irányába, és egészen addig haladt egyenesen arra, amíg nem került be a látóterébe a második akadály. Ekkor annak széle felé, majd a harmadik akadály széle felé mozgott, végül pedig a célra fordulva egyenesen odahajtott.

A következő 5.5. ábrán az akadályok pozícióját megváltoztattam, mert arra voltam kíváncsi, hogy sikeresen át tud-e menni a robot két akadály között. Az ábrán látható módon ez sikerrel teljesült. Az első akadály elkerülése alatt bekerült a második akadály

is az ütközési zónába, és így egészen addig haladt a robot a távolabbi akadály elkerülése felé, amíg nem került ki az első akadály ebből a zónából, és ekkor lehetősége nyílt a két akadály közötti rés megtalálásával a második akadályt jobbról megkerülni. Mivel abban a pozícióban ez pont a kisebb elfordulást eredményezte, ezért emellett döntött az algoritmus. Ezután a már megismert séma szerint követte a robot a harmadik akadály mellett egy biztonsági sáv határát, majd amint lehetett, a cél felé folytatta útját.



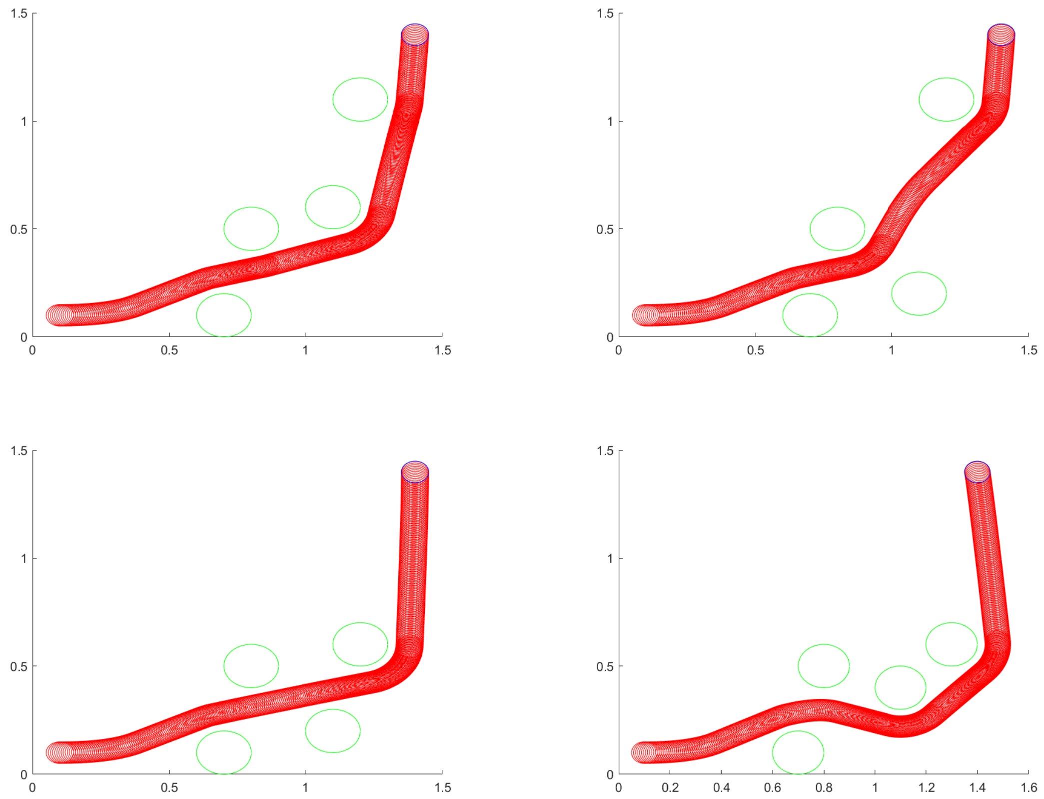
5.5. ábra. Statikus akadályok elkerülése 2.

A már látott 5.2. ábrán még mindig statikus akadályok szerepelnek, azonban még eggyel több. Jól látható, hogy ebben az esetben a robot nem teljesítette jól a feladatát, nekiment a második akadálynak. Ez annak köszönhető, hogy a látótávolság olyan nagy volt, hogy bár a robot érzékelte a második és a harmadik akadályt is, mivel a harmadik jobb széléhez kevesebbet kellett fordulni, ezért a fordulási sugarat is ahhoz tervezte meg, nem pedig a közelebbi ütközési ponthoz.

Ennek a javítására annyit változtattam a fordulási szöget számoló függvényen, hogy az ütközési távolság számolásához (amihez a fordulókör sugarát igazítottam) nem az éppen figyelembe vett cél távolságát használtam, hanem az ütközési távolságok minimumát. Ennek megfelelően meredekebbek lettek a kanyarok. A 21. oldalon az 5.6. ábrán találhatóak további képek a működésről, a teljesség igénye nélkül. Ezek közül az első az 5.2. ábrán látható hibás eset, amely itt már jól működik. A meredekebb kanyarodás közben hamarabb érzékelte a robot a rést az első kettő akadály között, és így könnyebben ment át közöttük. Ezután mindkét akadályt látva, a közelebbi alapján módosította pályáját, és így szinte egyenesen a harmadik akadály felé tudott menni. Ebben a szakaszban minden bemutatott ábrához tartozó maximális érzékelési távolság 0.5 méter volt.

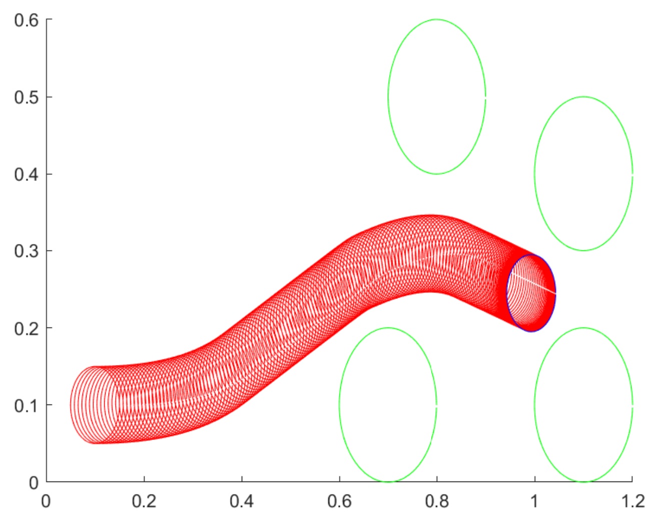
Mint megfigyelhető volt, ez az algoritmus jól kezelte a statikus akadályok esetén felmerülő esetek legnagyobb részét. Ennek ellenére találtam benne hibát, ez pedig a memóriamentességgel kapcsolatos: bizonyos esetekben képes oszcillációra az algoritmus. Ezt az 5.7. ábra esetében figyelhetjük meg, mert ott csak annyi történik, hogy a robot egyhelyben állva forog jobbra-balra.

Az oszcilláció magyarázata a következő: a robot amikor meglátja az $(1.1; 0.4)$-es akadályt, akkor még az ütközési zónájában nem szerepel az $(1.1; 0.1)$-es akadály, és így elindul az éppen meglátott akadály széle felé. Amikor bekerül a képbe az újabb akadály, akkor viszont elfordulna az új akadály alsó széle felé, azonban amint kikerül az ütközési zónából



5.6. ábra. További példák a statikus környezetben való navigálásra

a (1.1; 0.4)-es akadály, az algoritmus szerint elkezd visszafelé fordulni, és így egy hurokban ragad.

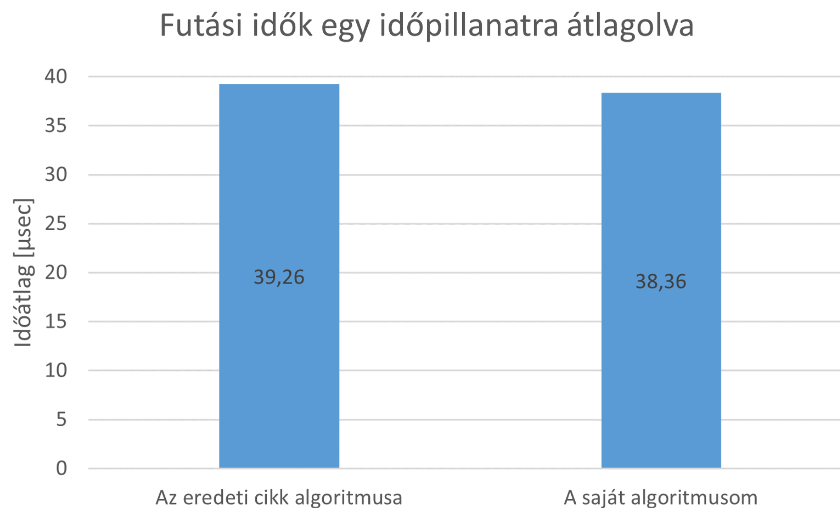


5.7. ábra. Oszcilláció kialakulása

5.4. Futási idő összehasonlítás

A célkitűzések között szerepelt az is, hogy az algoritmus továbbfejlesztésével a futási idő ne változzon számottevően. Ezért amikor készen lett az implementáció, akkor időméréseket is elhelyeztem a kódban, így felállítva egy egyszerű mérőszámot, amely azt mutatja, hogy a számítógépemnek mennyi időre volt szüksége egy időlépésnyi (ez 1 ms volt) kód futtatásához.

Az említett számítógép egy i5-7200U CPU-val és 8 GB memóriával ellátott eszköz, amely Windows 10 operációs rendszer alatt futtatta a szimulációt a MATLAB R2017 környezetben.



5.8. ábra. Futási idők összehasonlítása

A méréseket MATLAB-ban végeztem a *tic* és a *toc* függvények használatával. Mindegyik implementáció a saját tesztesetein futott, ezek közel azonos nehézségűek voltak. Mivel nem végeztem el rengetegszer a szimulációkat, csak ötször, ezért messzemenő következtetéseket nem lehet levonni az eredményből, csak tendenciák figyelhetőek meg. Egymás után futtattam a méréseket, ezért feltételezhetően az éppen a számítógépemre jellemző állapotok nem változtak meg érzékelhetően két mérés között.

Az 5.8. ábrát és a fentebb leírtakat figyelembe véve kijelenthető, hogy mind az új, mind a régi algoritmus közel azonos terhelést jelent számítási kapacitás tekintetében, nem jelenthető ki, hogy bármelyik gyorsabb lenne a másiknál, viszont mindkettő elég alacsony érték ahhoz, hogy bármelyik módszer valós időben alkalmazható legyen.

5.5. Konklúzió

Összefoglalásképpen kijelenthető, hogy a bemutatott új algoritmus sikeresen továbbfejlesztette az elődjét. Sikerült statikus akadályok tekintetében biztonságosabbá tenni, és így már nem produkál ütközéseket, azonban egyrészt elvesztette a dinamikus akadályok kezelésének képességét (amely jogosan az elődjét sem illette volna meg), másrészt viszont még nem vált teljesen megbízhatóvá a holtpont-probléma következtében kialakuló oszcilláció miatt. Sikeresen teljesült továbbá a differenciális meghajtás és a LIDAR-alapú érzékelés alkalmazása a bemutatott módszerben.

A leírt fő probléma miatt szükségesnek tartottam az algoritmus ezen verzióját továbbfejleszteni, azonban a fejlesztéshez vezető lépések között felmerült a meglévő szakirodalom

vizsgálata, és az ott látott jó ötletek alkalmazásának igénye is. Ennek megfelelően utánanéztem több algoritmusnak is, illetve a LIDAR használatára is igyekeztem példákat találni. Ezt a kutatást a 2. fejezetben mutattam be.

Az ott említett módszerek közül végül a *Tangent Bug* algoritmust tanulmányoztam behatóbban, mert működése nagyban hasonlít az 5-ödik fejezetben felvázolt első algoritmusomhoz, de segítségével megoldható a lokális minimumban ragadás problémája. Érdemes még megemlíteni, hogy a most bemutatott módszerem a LIDAR kezelésénél sok dolgot úgy vesz figyelembe, mint a Gap-alapú rendszerek, így lényegében az irodalomkutatásban említett két módszer ötvözete. Mindezek ellenére találtam olyan részeket mindegyik algoritmusban, amelyek illettek a saját algoritmusom céljaihoz vagy módszereihez, így a kutatásnak ez a része is hasznos volt.

6. fejezet

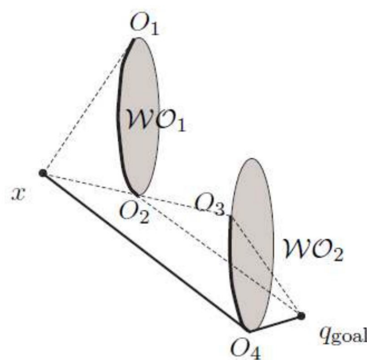
Tangent Bug algoritmus

Ebben a fejezetben bemutatom az eredeti Tangent Bug algoritmust, és azt, hogy ezt hogyan fejlesztettem tovább annak érdekében, hogy differenciális meghajtású és kiterjedéssel rendelkező robot esetén is működjön.

6.1. Leírás

Az első fontos dolog az algoritmushoz alkalmazott robotmodell tisztázása. Eredetileg ez egy pontszerű, omnidirekcionális meghajtású, teljes környezetében távolságot érzékelni képes robot. Később beszélek arról is, hogy mennyiben tértem el ettől, de az algoritmus megértéséhez maradjunk a most ismertetett feltevésnél.

Mint már a 2.1.2. részben láthattuk, az algoritmus két fő részből áll, ezek az akadálykövető és a szabadon futó üzemmód. A kettő közötti váltást egy költségfüggvény (továbbiakban *heurisztika*) segítségével végezzük, amit úgy kell kiszámolni, hogy a kérdéses, az érzékelés határán lévő pont (tehát vagy egy akadálypont, vagy az adott irányban a legtávolabbi érzékelhető pont) a robottól vett távolságát összeadjuk a pont céltól vett távolságával, ahogy a 6.1. ábrán is látható az érzékelt négy akadályvégpont esetében. Ilyen módon megkapjuk a ponthoz tartozó heurisztikát. Annyi megkötés van még, hogy a heurisztikával való munkában az akadályok körvonalainak éppen érzékelhető végpontjaival számolunk, mert arra tud jó eséllyel továbbhaladni a robot.



6.1. ábra. Példa a heurisztika számolására [3]

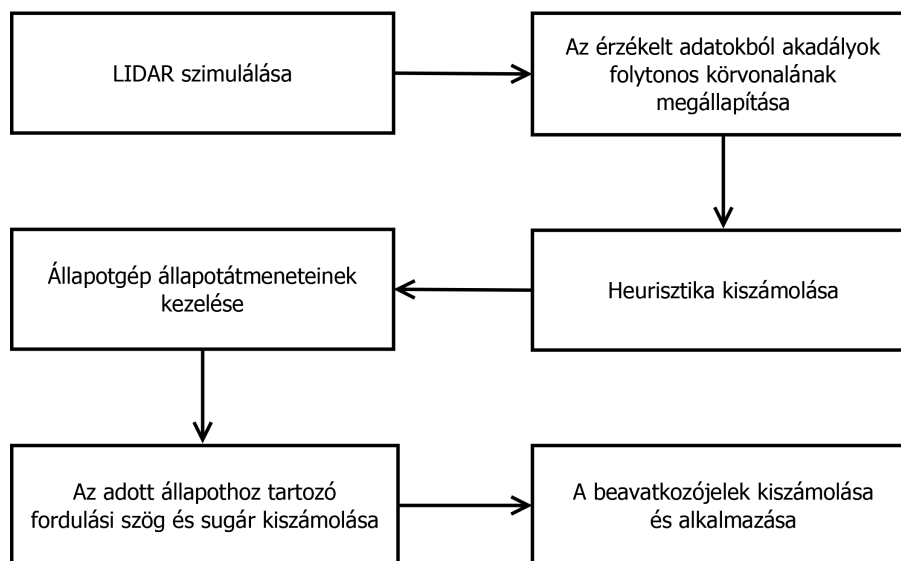
Kezdetben a szabadon futó üzemmód van érvényben, ez azt jelenti, hogy a robot a heurisztika minimumértéke felé mozog egészen addig, amíg az új lépésben kiszámolt heurisztika minimuma nagyobb lesz az előző körben számoltnál. Ez vizuálisan azt jelenti, hogy a robot az akadályokból újonnan érzékelt részeket a céltól távolabbinak látja a sa-

ját szemszögéből, tehát lokális minimumba került. Ekkor pedig átvált az akadálykövető üzemmódba.

Az akadályok követése úgy működik, hogy választunk egy irányt, és arra elfordulva haladunk az akadály mentén úgy, hogy mindig frissítjük két változó – a d_{leave} és a d_{min} – értékét és egymáshoz való viszonyát. A d_{min} az eddig észlelt legrövidebb távolság az akadály körvonala és a cél között, míg d_{leave} a legrövidebb távolság az akadály jelenleg észlelt körvonala és a cél között. Ez alapján kijelenthető, hogy akkor hagyhatjuk abba az akadálykövető viselkedést, amikor

$$d_{leave} < d_{min}.$$

Az algoritmus összefoglalásaként készítettem el a 6.2. ábrát is.



6.2. ábra. Az algoritmus folyamatábrája

6.2. Módosítások

Az előző fejezetben bemutatott algoritmus elég egyszerű, intuitív és megbízható, de egy olyan algoritmust szerettem volna implementálni, amely fizikai kiterjedéssel bíró, differenciális meghajtású robotot vesz figyelembe. Ennek megfelelően átalakítottam az eredeti algoritmust, de ez több buktatót rejtett magában, mint amennyit egy ilyen rövid célkitűzésből sejthettem.

A differenciális meghajtás annyiban változtatott csak az algoritmuson, hogy az eddig bemutatottakhoz hasonlóan meg kellett határozni a fordulási szöget és a fordulási sugarat, azonban ezen kívül nem befolyásolta számottevően a végrehajtás menetét.

Ezzel szemben a robot fizikai kiterjedése sok ponton beleszólt a metódusba. Ez már azt is módosította, hogy a robot nem minden szabadnak érzékelt úton fért el, hanem vizsgálni kellett azt is, hogy a szabad hely (vagy a különböző akadálypontok között érzékelt távolság) elegendő-e a robot biztonságos áthaladására, vagy éppen sok pici akadályból áll-e egy nagyobb. Ennek megoldására egy, a már megismert „akadályok kiterjesztése” nevű módszert alkalmaztam, vagyis ha két akadály túl közel volt egymáshoz, akkor úgy vettem, mintha közöttük egy virtuális akadályt érzékelnék.

Kezdetben azt hittem, hogy ezzel megoldottam a problémát, azonban kiderült, hogy amikor a robot látótávolságából kikerül az utolsó valós akadálypont, akkor hirtelen több

virtuális pont is elveszhet, így hirtelen ezen az – immár torzítatlan – adatsoron a robot érzékel egy olyan minimális heurisztikájú pontot is, melyről eddig úgy tudta, hogy az akadálypont. Ezt a problémát úgy oldottam meg, hogy figyelem az előző lépésben látott akadálypontok számát, és ha ez ugrásszerűen megváltozik, akkor az előző lépésben meghatározott irány felé haladok továbbra is. Az eddigieknek megfelelően szükségessé vált az is, hogy megállapítsuk, hogy az előző lépésben követett akadály melyik jelenlegi akadálynak felel meg. Ezt úgy valósítottam meg, hogy a legközelebbi akadálypont (amihez képest az érintőt számolom) irányát eltároltam, és az új lépésben ennek közelében keresem ugyanazt az akadályt.

Az akadálykövetés egy aprónak tűnő, de fontos lépése a kezdeti érintő irányának meghatározása, ugyanis nem mindegy, hogy ha például a robot maga előtt egy tárgyat érzékel, akkor merről kezdje el követni, jobbra vagy balra fordulva. Általánosságban viszont elmondható, hogy ha a követendő akadály legközelebbi pontja jobbra van, akkor érdemes balra fordulva követnünk a körvonalát, ha viszont ez a pont balra van, akkor jobbra érdemes fordulnunk. Az eredeti Bug-algoritmushoz képest ez is egy kisebb módosítás, kiegészítés volt.

Ezeken kívül implementáltam még egy egyszerű szabályzást arra, hogy a robot ne az érzékelés határán, hanem annál közelebbről kövesse az akadályokat, azért, hogy robosztusabb legyen az érzékelés, illetve azért, hogy közben ne ütközzön más akadályokkal. Ezt úgy valósítottam meg, hogy a robotot nem az érintő irányában indítottam tovább automatikusan, hanem az akadálytól való távolság függvényében torzítottam ezt a célirányt egy negatív visszacsatolással az akadály felé, így végül a robot a beállított távolságot vette fel az akadály körvonalától. Ezt részletesebben kifejtem a következő fejezetben is, de az eredmények között például a 6.4. ábrán is jól látszik a fix követési távolság.

6.3. Implementálás

Az algoritmus implementációja az eddig látottakhoz hasonló volt struktúrájában, ugyanis egy fő ciklusban lépegetett a szimuláció előre egy-egy időlépést (ez továbbra is 1 ms volt), és ebben kellett a robotnak és a – jelenleg statikus – akadályoknak az útvonalát és a beavatkozási logikáját tervezni.

A robot útvonaltervezése most is a LIDAR szimulációjával kezdődött, és utána ebből elkészítettem a látott akadályok folytonosnak érzékelt szakaszait, majd ezen információk segítségével mozgattam a robotot. Az érzékelő adatainak feldolgozása azzal kezdődött, hogy minden pontról eldöntöttem, hogy akadály-e vagy sem, és ha akadály volt, akkor megszámoztam, hogy hányadik akadályhoz tartozik. Erre az osztályozásra „színezésként” fogok hivatkozni, mert az sokkal szemléletesebb.

Miután az akadálypontok ilyen módon színezésre kerültek, gondoskodni kellett arról, hogy ha két akadály között át lehet látni, de a robot nem fér közöttük át, akkor az ilyen kisebb akadályokat egy „akadályszigetként” kezeljük. Ezt úgy tettem, hogy minden akadálypontba rajzoltam egy körívet, ahol a robot középpontja volt a kör középpontja és a körív hossza mindkét irányban a robot sugara volt, majd az ívek azon szakaszain, ahol ezek elfedték más akadályok valódi, vagy ilyen „ívvel rajzolt” képét, ott átszíneztem a pontokat. Minden olyan helyen, ami így két valós akadály között volt, ott feljegyeztem az ívpontok távolságát is a robottól, így hozva létre legalább kettő közeli akadályból egy nagyobbat.

Természetesen elképzelhető volt, hogy így egy akadályt csak részlegesen színeztem át, így még ezt is ellenőrizni kellett. Ennek megfelelően végül a körívek számolása miatt is kétszer futottam át az érzékelési tömbön, és utána a színezés korrekciója miatt is. Azért kellett kétszer átfutni a tömbön, hogy mindkét irányban (balról-jobbra és jobbról-balra

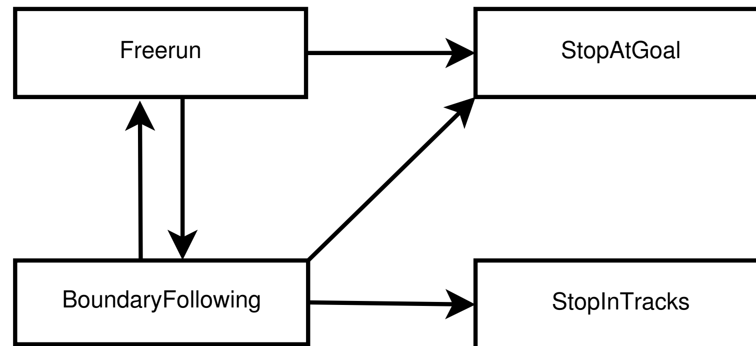
is) biztosítva legyen a számolás. Ez a sok tömbindexelés adott esetben (sok pici akadály esetén például) nagy számítási kapacitásbeli növekedést okoz, ezt majd a futási időknél látni is fogjuk.

A színezés végeztével ki kellett számolni a végpontok heurisztikáját. Ehhez először a végpontok helyzetét állapítottam meg a kész színezés alapján, majd megnéztem azt, hogy a cél irányában van-e akadály. Ha nem volt, akkor természetesen ebben az irányban volt a legkisebb a heurisztika, viszont ha volt, akkor minden végpontra a

$$heuristic = \mathrm{d}(\mathrm{robot}, \mathrm{obstaclePoint}) + \mathrm{d}(\mathrm{obstaclePoint}, \mathrm{goal})$$

képlet segítségével, a 6.1. fejezet alapján ki kellett számolni a heurisztikát, majd ezt a vektort, illetve a minimális heurisztikát, és annak indexét átadni a mozgást tervező függvénynek.

A mozgástervezés logikáját egy állapotgéppel valósítottam meg, ugyanis az algoritmus jól érezhetően különböző logikával jár el egy-egy jól körülhatárolható esetben, például az akadálykövetés vagy a szabadon futás tekintetében, és ezt kézenfekvően egy állapotgép írja le jól. A különböző állapotok közötti állapotátmeneteket a 6.3. ábra szemlélteti, az átmenetek feltételeit viszont leírom itt szövegesen.



6.3. ábra. Az implementáció állapotgépe

A kiindulási *Freerun* állapotból odaérhetünk a célba (és ezzel együtt a *StopAtGoal* állapotba), ha a cél koordinátáitól való távolságunk kisebb, mint egy kis hibahatár (a numerikus pontosság miatt ez soha nem lesz pontosan nulla). Amennyiben viszont nem értünk be a célba, de az előző időlépéshez képest a heurisztika minimuma nőtt, akkor átváltunk a *BoundaryFollowing*, avagy akadálykövető állapotba.

Ha már az akadálykövető állapotban voltunk, akkor szintén megállhatunk a célban, ha odaértünk, vagy ha megkerültük az akadályt teljesen, és nem tudtunk kilépni az akadálykövetésből, akkor a *StopInTracks* nevű állapotban állunk meg, amely lényegében azt fejezi ki, hogy a cél nem elérhető. Ha viszont a már említett $d_{leave} < d_{min}$ teljesül, és nem szembesültünk a virtuális akadálypontok eltűnésének problémájával (lásd az előző, 6.2. fejezetben), akkor visszaléphetünk a szabadon futó állapotba.

Megemlítem még, bár nagyrészt magától értetődik, hogy a célban vagy az akadály elérhetetlensége miatt történő megállás állapotából nem lépünk ki.

Ami az adott állapotokra jellemző számolásokat és döntéseket illeti, azok legfontosabb feladata a fordulási szög és a fordulási sugár meghatározása. Ez a szabadon futó állapotban elég egyszerű: a minimális heurisztikájú végpontot a robottal összekötő szakasz X tengellyel bezárt szögéből kivonjuk a robot aktuális orientációját az X tengelyhez képest. A fordulási sugár meghatározása már nehezebb, de ezt úgy oldottam meg, hogy a legközelebbi akadálypont távolságánál jelentősen kisebb sugárral (például a távolság tizedével) fordul a robot, azért, hogy biztosan ne ütközzön semmivel, és hogy gyorsan és hatéko-

nyan át tudjon térni például egy akadálykövetési orientációból egy – akár nagy orientációs különbséget jelentő – nyílásra, amit a robot ebből a szögből vett észre először.

Az akadálykövetésnél a központi kérdés, ami a fordulási szög meghatározásából következik, az az érintő kiszámolása. Erre egy viszonylag egyszerű módszert alkalmaztam: a legközelebbi megfelelő színű akadálypont irányához $\pm 90°$-ot adva megkapjuk az érintő irányát, és a körüljárási irányt ennek a $90°$-nak az előjele dönti el. Ezt annak megfelelően választjuk, hogy az akadály legközelebbi pontja a robot melyik oldalán helyezkedik el (lásd a 6.2. fejezetet).

Ennek a $90°$-nak az előjelén kívül a nagyságát is változtattam, ezzel megvalósítva egy egyszerű szabályzást. Ha ugyanis a robot túl közel megy az akadályhoz, az balesetveszélyes lehet, viszont ha túl messziről követi, az nem kellően robosztus, és még a többi akadálynak is neki tudna menni közben. Ezért végül egy hiszterézises szabályzást készítettem, amely a robotot közelebb vitte az akadályhoz az érintőhöz szükséges additív tag nagyságának változtatásával amennyiben az a robot biztonsági távolsággal megnövelt sugaránál messzebb volt az akadályhoz képest, illetve eltávolította, ha közelebb volt. Közelítés esetében ez a módosító szorzótényező a

$$\frac{d_{lowest}}{d_{maxSense}}$$

volt, ahol a számlálóban a legközelebbi akadálypont, a nevezőben pedig a maximális érzékelési távolság szerepel. Természetesen ez az érték pontosabban hangolható a $d_{maxSense}$ ismeretében.

Amennyiben az akadály túl közel volt (például a robot biztonsági sugarának 1.2-szeresénél közelebb), akkor viszont ezt a fordulási szöget megnöveltem (jelen esetben 10%-al). Ez egy egyszerű, de hatékony eszköznek bizonyult, hatása megfigyelhető például a már említett 6.4. ábrán is.

Ebben az állapotban még az említett „simítást” is végrehajtottam, ez azt jelenti, hogy az újonnan kiszámolt orientáció 1 : 4-hez súlyozott átlagát vettem az előző lépés orientációjával, így kevésbé zajos fordulásokhoz jutottam. Még egyszer kiemelném, hogy ezek a tényezők nem elengedhetetlenül szükségesek, és nem tökéletesen pontosan beállítottak, de az általam megadott értékek esetén jó eredménnyel fut az algoritmus.

A *StopAtGoal* és a *StopInTracks* állapot neve leírja a működést is: itt a robot csak a lehető legrövidebb úton megáll, ezen kívül nincs más tennivalója.

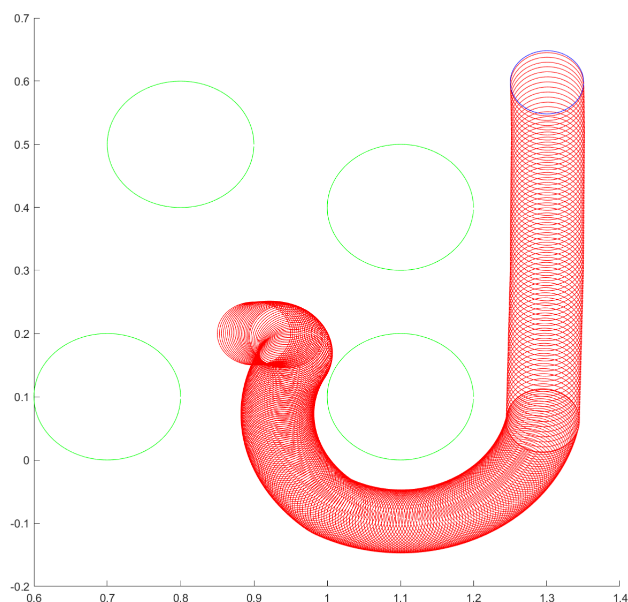
Miután a fordulási szöget és a sugarat kiszámoltuk, a szükséges beavatkozójelek nagyságát és előjelét az (5.2.) egyenlet alkalmazásával határoztuk meg, figyelve arra, hogy a robot megfelelő meghajtó szervei a jó irányba indítsák meg a robotot.

6.4. Eredmények

Az algoritmus tesztelése során döntően olyan eseteket vettem figyelembe, melyek az előző algoritmusnak gondot okoztak volna, vagyis olyan eseteket, ahol a robot belemegy egy olyan térbe, amely több oldalról akadályokkal határolt, és ami a legfontosabb, lokális minimumot tartalmaz. Az első ilyen teszteset rögtön az a szituáció, amelynél rádöbbentem az 5.3. fejezet végén a lokális minimumban ragadás problémájára. Ez a 6.4. ábrán látható.

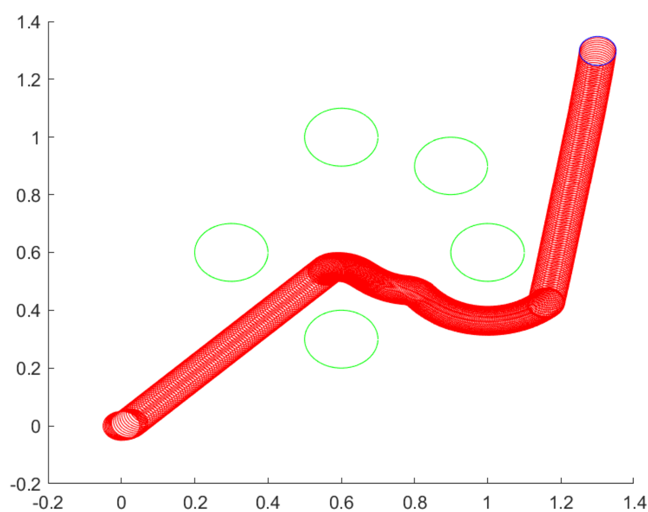
Ebben az esetben a robot jól kezelte a kezdeti, nagyon zsúfolt környezetét, és a jobb alsó akadály követésével végül kitalált a „csapdából”. Jól látható, hogy milyen pontosan követi az akadály körvonalát egészen addig, amíg a látott heurisztika csökkenésével ki nem jut az akadálykövető üzemmódból. Ekkor viszont már egyenesen a cél felé fordulva odahajt.

Hasonló szituációt képvisel a következő, 6.5. ábra is, ahol az akadályok elhelyezésével és az érzékelési távolság csökkentésével a robotot beirányítottuk két akadály között az



6.4. ábra. Az előző algoritmusnak hibát okozó szituáció

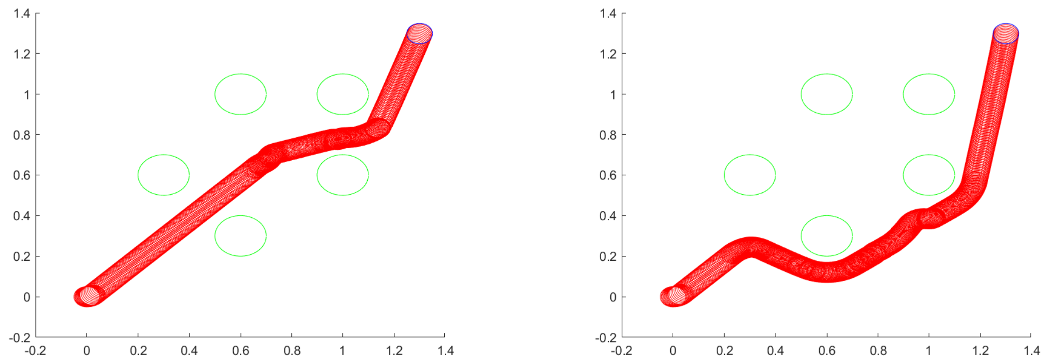
előzővel közel azonos struktúrába, ahonnan szintén az előző szituációban leírtak szerint találta meg az útját.



6.5. ábra. Akadályokból álló „zsák”

Ezek után szeretnék példát mutatni a maximális érzékelési távolság útvonaltervezésre gyakorolt hatására. Ehhez ugyanazt a kissé módosított akadályelrendezést egy $R_{sense} = 1$ és egy $R_{sense} = 0.4$ beállítású esetben futtattam. Ennek az eredménye a 6.6. ábrán látható, ahol a bal oldali esetben 0.4, és a jobb oldali esetben 1 az értéke az említett paraméternek.

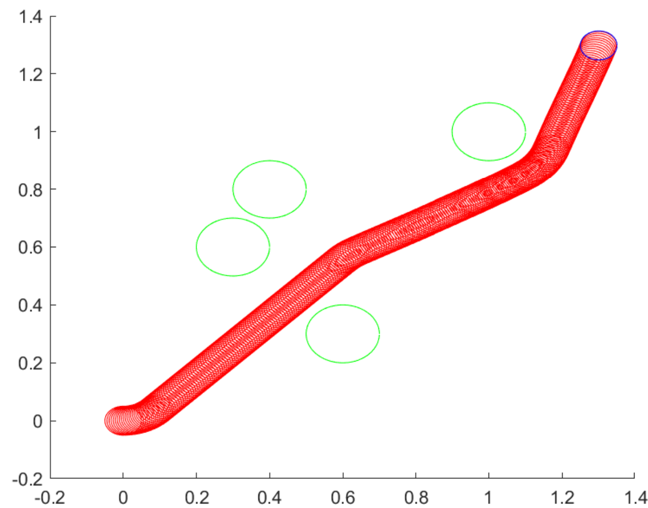
A hivatkozott ábrán jól látható, hogy a kisebb érzékelési távolsággal ellátott robot belement a kezdetben legrövidebb utat ígérő irányban a két akadály közé, majd végül



6.6. ábra. Példa az érzékelési távolság hatására. Ez a bal oldalon 0,4 méter, a jobb oldalon 1 méter volt

egy résen át távozott, míg a nagyobbik érzékelési távolságú esetben a robot „látta” az út túloldalát akkor eltakaró akadályt, és inkább megkerülte az egész akadályhalmazt.

Egy kevésbé speciális esetet is teszteltem, ezt azonban különösebb magyarázat nélkül mutatom be a 6.7. ábrán, mert az eddigiekben megismertekhez képest nem nyújtott meglepetést, hiszen a robot ebben végig a szabadon futó állapot szerint halad.



6.7. ábra. A szabadon futó állapot

6.5. Konklúzió

Az előző fejezetekben láthattuk az eredeti és a módosított Tangent Bug algoritmus bemutatását, illetve a módosított verzió implementációját, és annak eredményeit is. Ezekre alapozva kijelenthető, hogy ez az algoritmus sikeresen elhárította az 5. fejezetben bemutatott első saját algoritmus legfőbb limitációját, a lokális minimumban ragadás problémáját azzal a módszerrel, hogy akadályok követésével váltogat az algoritmus működési módozatai között. Ez az ötlet, és az egységes heurisztikakezelés biztosítja a helyes pályatervezést a robot számára.

Az implementációnál részletesen bemutatott módosítások segítségével sikerült a Tangent Bug algoritmust differenciális meghajtású, LIDAR-ral ellátott robotra kiterjeszteni, azonban a számítási kapacitás megnőtt a korábbi algoritmusokhoz képest, döntően az érzékelő által szolgáltatott adatok megfelelő feldolgozása miatt. Erről a növekedésről kvantitatív információt a 7.5. fejezetben szolgáltatok, ahol összehasonlítom az eddigi összes bemutatott és mélyebben vizsgált algoritmust ebből a szempontból.

Végül az elért eredmények és a megszerzett tapasztalat birtokában a saját algoritmusom új változatának elkészítése mellett döntöttem, amelyben az eddigi jó tulajdonságokat szeretném ötvözni a Bug által használt akadálykövetési és állapotgépvezérelt technikák felhasználásával.

7. fejezet

Saját algoritmus II

Ebben a fejezetben bemutatom az általam készített algoritmus második változatát, amely már képes a lokális minimumok elkerülésére is. Az algoritmus leírásán és implementációján kívül bemutatásra kerülnek a szimulációs eredmények és az algoritmus dinamikus tulajdonságai is.

7.1. Leírás

A Tangent Bug megismerésével példát láttam egy olyan működő költségfüggvényre, és annak felhasználására a különböző üzemállapotok közötti váltásra, amely segítségével megoldhatóvá vált a lokális minimumok elkerülése. Ebből okulva úgy határoztam, hogy az eredeti algoritmusom akár lehetne az egyik üzemállapota is a most készülő második algoritmusomnak.

Átgondolva az 5. fejezetben bemutatott algoritmus működését, úgy láttam, hogy az döntően a Bug akadályelkerülő üzemmódjának a funkcionalitását tudja ellátni, úgyhogy ennek a helyére illesztettem az első algoritmusomat egy modulként.

Ettől a felépítéstől azt vártam, hogy továbbra is problémamentesen képes legyen a robot navigálni a lokális minimumot nem tartalmazó környezetben, illetve azt, hogy a Tangent Bug jó tulajdonságát megőrizve már a lokális minimumok kezelésére is képes legyen, hiszen a LIDAR-tól származó adatok kezelése már a Tangent Bug mintájára történt, tehát a túl közeli akadályok összevonásra kerültek. Emellett a Tangent Bug szabadon futó állapotának megőrzésétől egy, önmagában csak a biztonsági zónás tervezésnél optimálisabb útvonal megtalálását vártam.

A robotmodell továbbra is a 4. fejezetben látott, és feltételezzük (kezdetben), hogy a környezet statikus.

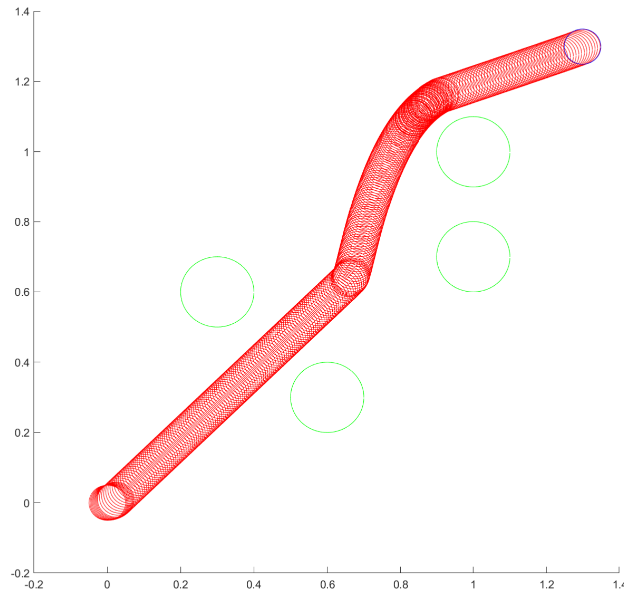
7.2. Implementálás

Az implementáció váza a Tangent Bug általam módosított változatának implementációja, annyi eltéréssel, hogy az állapotgép *FollowBoundary* állapota lerövidült a biztonsági zónával vett ütközések keresésévé, és az ennek megfelelően történő beavatkozássá. Minden egyéb számolás, amely a keretprogram (az állapotgép) működéséhez szükséges, azt továbbra is elvégezzük ebben az állapotban is (például a követett akadályt is azonosítjuk, noha ez a saját algoritmusom működéséhez nem volt előzőleg szükséges).

Még egy változtatásra volt szükség, azonban ez nem az akadálykövető, hanem a szabadon futó részben történt. Azzal a jelenséggel szembesültem, hogy a szabadon futás során néha a biztonsági zónához képest túl közel kerültünk az akadályokhoz, és így amikor az akadályok kiterjesztése a közelségnek megfelelően túl nagy lett, és a robot átváltott aka-

dálykövetésbe, akkor az egyetlen szabad út az akadály érintőjére merőlegesen, az akadállyal ellentétesen volt, és ez a bejárt pálya nagy torzulásával járt.

Ennek kezelésére úgy jártam el, hogy ha szabadon futó állapotban a robottól a cél felé van akadály, akkor a meghatározott legkisebb heurisztikájú ponttól (az akadály legszélső pontjától) messzebb kanyarodtam. Ennek segítségével az akadályoktól távolabb kerülve tudunk átváltani az akadálykövető viselkedésmódra. Egy ilyen kép látható a 7.1. ábrán.



7.1. ábra. Statikus akadályok elkerülése

7.3. Eredmények

Az algoritmus implementálása után az első legfontosabb teszt annak a helyzetnek a kezelése volt, amely az 5.7. ábrán láthatóan a legnagyobb gondot nyújtotta az első saját algoritmusomnak.

Ennek megfelelően a 7.2. ábrán látható az új algoritmus által szolgáltatott válasz. Ezen jól láthatóan a robot elfordul a lent látott gap irányába, és megkerülve az akadályt visszavált a *Freerun* üzemmódba, és beér a célba. Ezzel tehát példát is láttunk arra, hogy a robot immár képessé vált a felmerülő lokális minimumok kezelésére akkor is, ha azok két különálló akadály által jöttek létre.

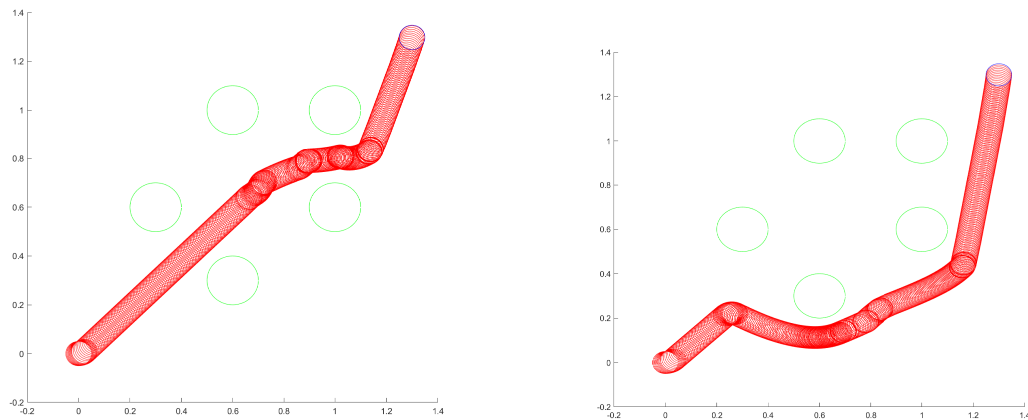
7.2. ábra. Oszcilláció és lokális minimum elkerülése

Itt fontosnak tartom megjegyezni azt, hogy ha egy folytonos akadályban lett volna lokális minimum, azt az első saját algoritmus is le tudta volna kezelni, hiszen csak kikerüli egészen addig, amíg a cél felé fordulhat, azonban a fő problémát az egymáshoz közeli, de mégis különálló akadályok alkották. Ekkor viszont felmerül a jogos kérdés: Miért kellett a teljes Tangent Bug keretprogramját felhasználni, mikor elég lett volna csak az új adatfeldolgozást használni?

7.3. ábra. A szabadon futó üzemmód nélküli program egyik limitációja

A válasz a feltett kérdésre az, hogy vannak olyan esetek, amikor a Tangent Bug által nyújtott állapotváltás teszi lehetővé, hogy elérjük a célt. Például a 7.3. ábrán az akadályban lévő nyílást a robot kellően nagy érzékelési távolság esetén nem venné számításba, hiszen még követi az akadályt, amikor már feltűnik a gap túloldala, és ezzel továbbra is köröz az akadály körül, azonban a heurisztika kiszámolása megszakítja az akadálykövető magatartást, és sikeresen beér a robot a célba. Az ábrán a robotot a piros kör, a célt a zöld kör, a biztonsági zónát a szürke alakzat, az akadályt pedig a fekete ív jelzi.

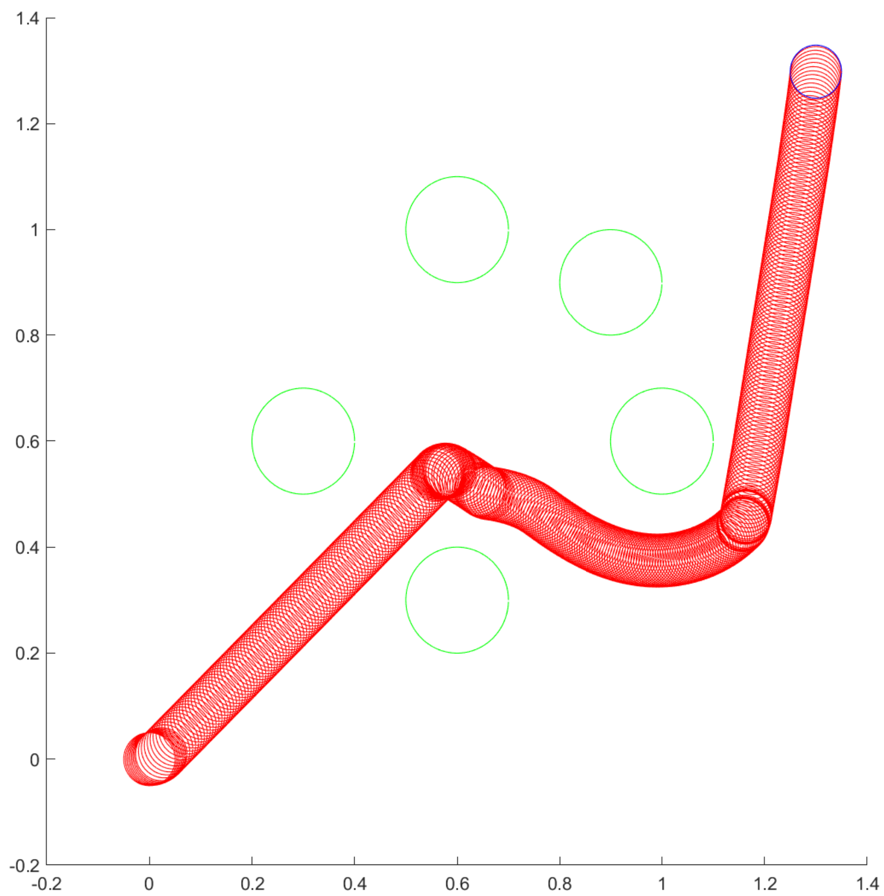
Ennél az algoritmusnál is szeretném megmutatni a maximális érzékelési távolság okozta különbségeket, ugyanazzal az akadályelhelyezéssel, mellyel a Tangent Bug esetén is tettem. Ez a 7.4. ábrán látható.



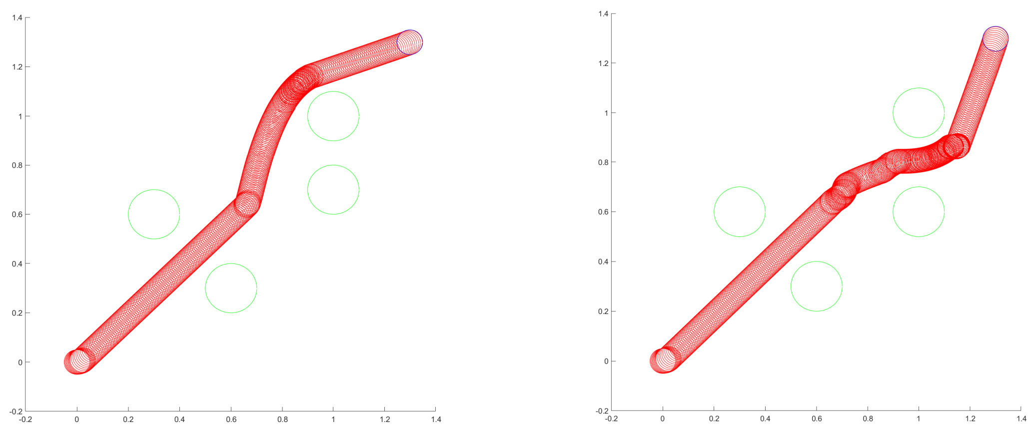
7.4. ábra. Példa az érzékelési távolság hatására

Ennél a tesztelésnél is elkészítettem az általam csak „zsáknak” nevezett akadályelosztást, melynél a robot elhalad két akadály között, és rögtön utána észreveszi, hogy a továbbhaladás útja zárva van, de létezik kijárat is, amit az akadályt követve megtalál. Ezt a 7.5. ábrán láthatjuk.

Szeretnék még egy példát mutatni az útvonaltervezésre, ahol a bal oldalon a már látott 7.1. ábrát, a jobb oldalon viszont annak egy változatát láthatjuk, az útvonalba eső két olyan akadállyal, amely között már átfér a robot. Ez a 7.6. ábrán figyelhető meg.



7.5. ábra. Lokális minimum elkerülése

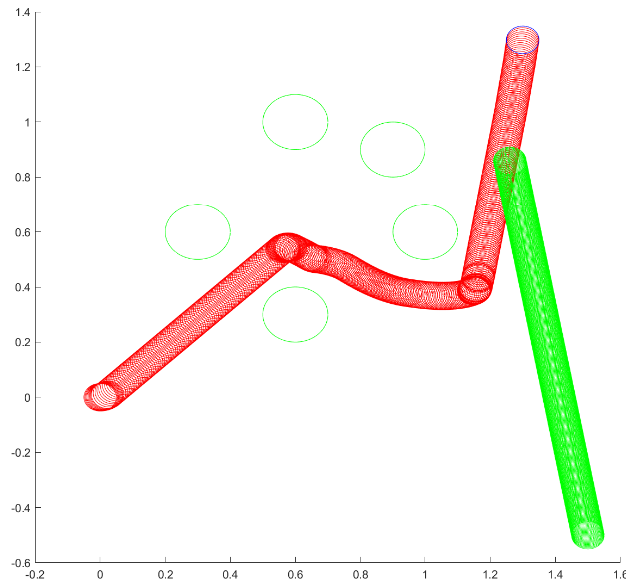


7.6. ábra. Példa az akadályok közötti rések méretének fontosságára

7.4. Dinamikus tulajdonságok

Az algoritmust statikus környezetre terveztem, azonban az elkészítés során kíváncsi lettem a viselkedésére változó környezetben, így ezt is teszteltem. A tesztjeim során az egyszerűség kedvéért csak egy akadályt használtam, és az is egyenes vonalú egyenletes mozgást végzett.

Ízelítőül nézzük is meg a 7.7. ábrát! Ezen az látható, ahogy a robot a célegyeneshez közeledve érzékeli a közeledő akadályt, majd eredeti „tervének" megfelelően továbbhalad.

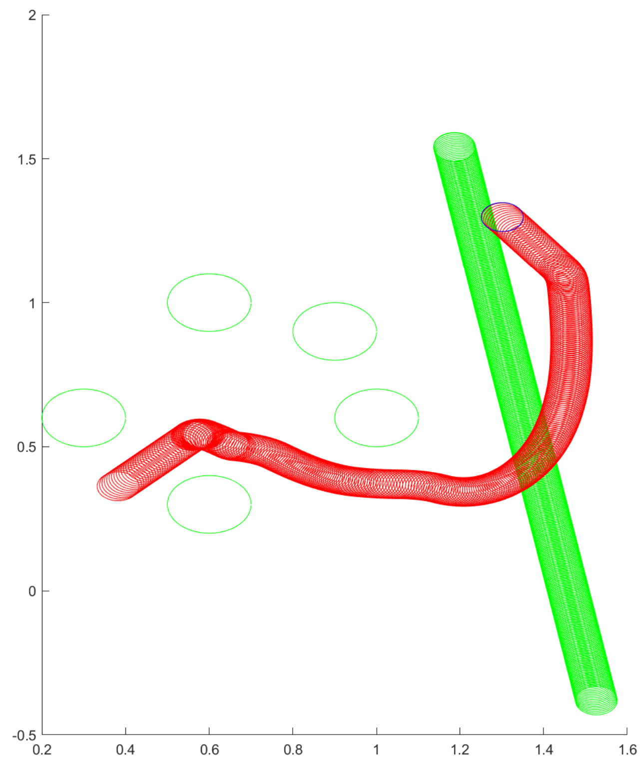


7.7. ábra. Elhaladás a közeledő akadály előtt

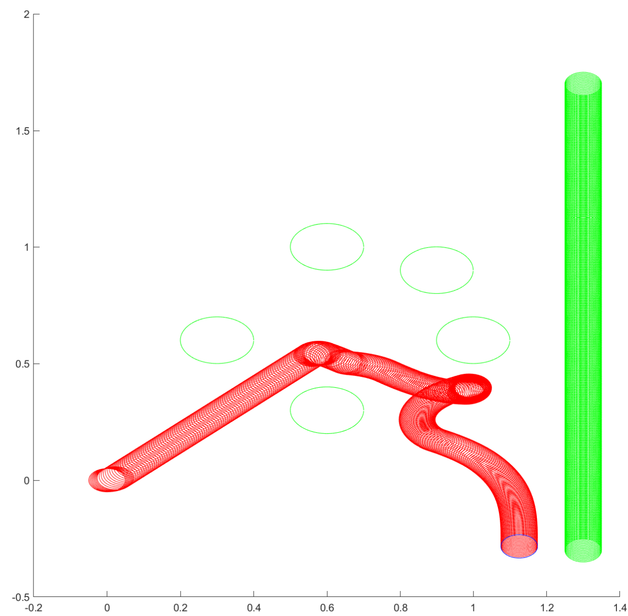
A következő, 7.8. ábrán nézzük meg azt a szituációt, ahol az akadály lentről felfelé keresztezi a robot útját, úgy, hogy elhalad előtte. A robot ennek megfelelően keresztezi az akadály útját, majd amikor úgy érzékeli, hogy az már messzebb van, mint a cél, akkor elindul egyenesen a cél irányába.

Noha a fent leírt esetekben az algoritmus teljesen jól kezelte a mozgó akadályokat, ez nem általános érvényű dolog. Ennek egy példája látható a 7.9. ábrán, ahol a robot a fentről közeledő akadály előtt próbál átvágni, de a kis sebességkülönbség miatt végül azt követni kényszerül, és mivel a heurisztika nincs felkészítve arra, hogy így elmozogjon az akadály, a robot nem fordul meg, és így nem ér célba.

Összegzésül tehát elmondható, hogy az algoritmus nem megbízható dinamikus környezet esetén. Ez bár nem kellemes, de közel sem meglepő megfigyelés.



7.8. ábra. Cél felé távolodó akadály követése

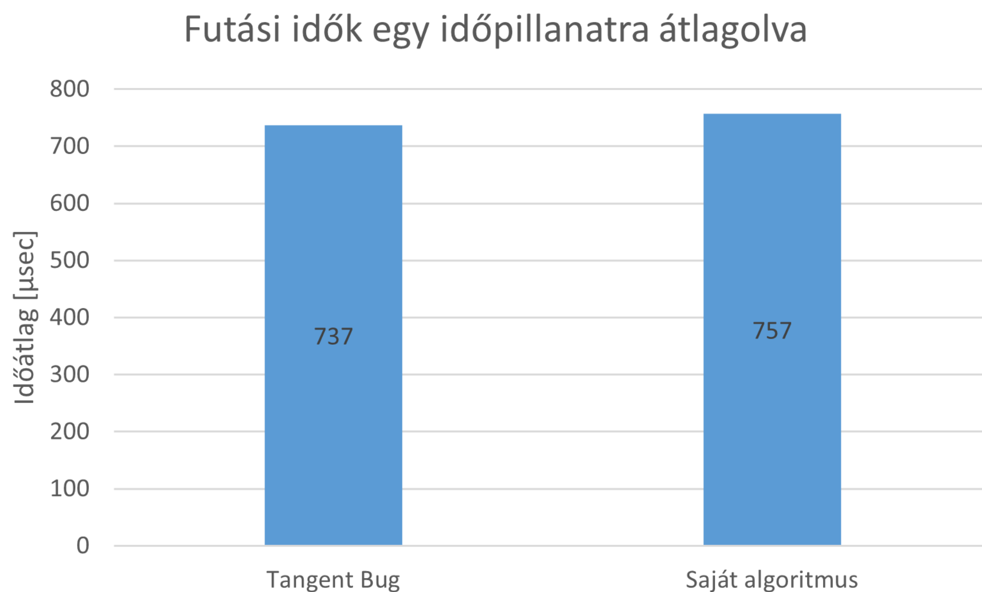


7.9. ábra. Eltérülés azonos sebesség esetén

7.5. Futási idők és egyéb mérőszámok

Ebben a pontban összehasonlítom a most ismertetett algoritmus és a Tangent Bug implementációjának futtatásához szükséges időt. A mérési módszer megegyezett az 5.4. fejezetben látottal, vagyis az ábrázolt mérőszám azt mutatja, hogy átlagosan mennyi idő szükséges egy időlépésnyi (1 ms) szimuláció végrehajtásához.

Mindkét algoritmust azonos pályán, egymás után teszteltem MATLAB-ban a *tic* és a *toc* függvények használatával. A pálya mindkét esetben a 7.5. ábrán látható akadályelhelyezkedés volt. Hasonlóan az előző méréshez, most is inkább tendenciák megállapítására szolgál az eredmény. A méréseket egymás után futtattam, ezért feltételezhetően az éppen a számítógépemre jellemző állapotok nem változtak meg érzékelhetően két mérés között. Az említett számítógép egy i5-7200U CPU-val és 8 GB memóriával ellátott eszköz, amely Windows 10 operációs rendszer alatt futtatta a szimulációt a MATLAB R2017 környezetben, tehát ugyanaz, mint amin az 5.8. ábrán látható mérés készült.

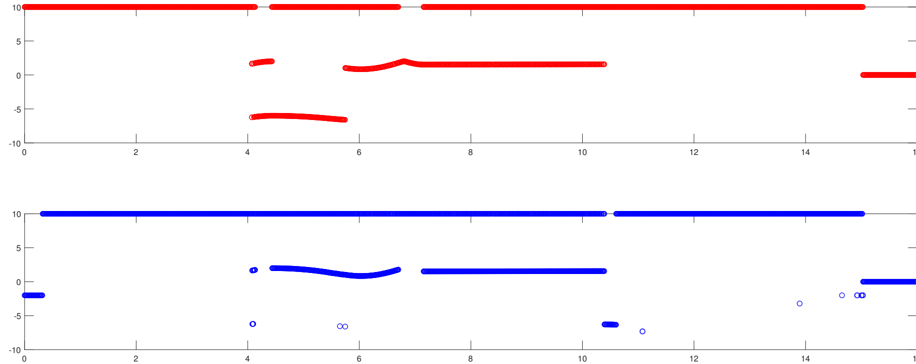


7.10. ábra. Futási idők összehasonlítása

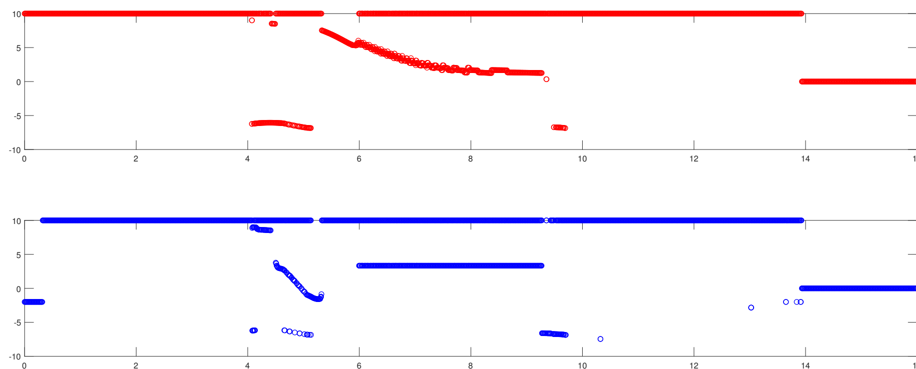
A Tangent Bug algoritmusnál utaltam a számítási igény megnövekedésére, ezt most számszerűsíteni is tudjuk a 7.10. ábra alapján. Visszalapozva az előző mérési eredményhez (5.8. ábra), megfigyelhető, hogy az új, erősebb algoritmusok közel 20-szor annyi ideig futottak, így jelentősen számításigényesebbek, mint a régebben összevetett két algoritmus. Ez érthető, hiszen sokkal komplexebb számításokat kell elvégezniük, és végeredményben bővebb funkcionalitást is nyújtanak.

A futási idő számítása mellett fontos mérőszám a robot célba érkezéséhez szükséges idő is. Hasonlóan a futási időkhöz, a mérés alapjául szolgáló pálya mindkét esetben a 7.5. ábrán látható akadályelhelyezkedés volt. Itt arra lettem figyelmes, hogy a saját algoritmusom ugyanazokkal a beállításokkal (végsebesség, maximális gyorsulás, látótávolság...) gyorsabban, összesen $13,935$ másodperc alatt végzett, míg a Tangent Bug-nak ugyanehhez $15,032$ szimulációs másodpercre volt szüksége. Ez az időbeli különbség annak tudható be, hogy a Tangent Bug esetén az akadálykövetésnél kisebb fordulási sugarak adódtak a számolásnál, és így lassabban tudott a cél felé haladni a robot, mint a másik esetben.

Egy másik megszokott mérőszám algoritmusok esetén a beavatkozójelek nagysága az idő függvényében. Ezt is elkészítettem mindkét algoritmus mindkét beavatkozójelének esetére, ezek a 7.11. ábrán láthatóak a Tangent Bug, és a 7.12. ábrán a saját algoritmusom esetén. Mindegyik ábrán kék színnel jelöltem a bal oldali kerék szögsebességét és piros színnel a jobb oldali kerék szögsebességét. A vízszintes tengelyen az eltelt idő látható másodpercben, míg a függőleges tengelyen a szögsebesség $\frac{\text{rad}}{s}$-ban.



7.11. ábra. A Tangent Bug által előállított beavatkozójelek



7.12. ábra. A saját algoritmus által előállított beavatkozójelek

A 7.11. és a 7.12. ábra alapján elmondható, hogy általában mindkét algoritmus a gyorsítás érdekében legalább az egyik beavatkozójelét a maximális értéken tartja, azonban ez, mint látjuk, ahhoz vezet, hogy folyamatosan váltakozik a maximális és egy kisebb beavatkozójel a robot mindkét kerekén. Ez fizikai rendszerek esetén gyorsabb meghibásodáshoz vezet, így további finomításra van szükség e téren.

7.6. Konklúzió

Ebben a fejezetben bemutattam a saját algoritmusom továbbfejlesztett változatát és összehasonlítottam azt a Tangent Bug algoritmus szintén általam továbbfejlesztett verziójával.

Az eredményeket tekintve kijelenthető, hogy az új algoritmus sikeresen megoldotta azt a problémát, amely őt életre hívta: az oszcillációt, és ezzel a lokális minimumban ragadást is. Ez döntően a bemeneti adatok újfajta kiértékelésének köszönhető, azonban a megbízható és biztonságos működéshez elengedhetetlen az is, hogy a Tangent Bug-nál megismert szabadon futó állapot is alkalmazásra került.

8. fejezet

Összefoglalás

Ebben a fejezetben összefoglalom a dolgozat tartalmát, és bemutatom az alapvető problémafelvetés mellett az elért eredményeket is.

8.1. A kész módszerek összevetése

Munkám során tanulmányoztam, implementáltam és teszteltem négy különböző algoritmust, amik közül kettő-kettő jól párba állítható a megvalósított feladat és az ahhoz szükséges erőforrások tekintetében. Ezek a párok a következők: a kiindulási algoritmus (3. fejezet) és az első saját algoritmus (5. fejezet), illetve a Tangent Bug algoritmus (6. fejezet) és a második saját fejlesztésű algoritmus (7. fejezet).

Az első párból a saját algoritmusom kijelenthetően jobban teljesített, mint az irodalomból tanult párja, hiszen megbízhatóbb és biztonságosabb. A második pár tekintetében nem állítható fel ilyen egyértelmű reláció, ugyanis a két algoritmus ugyanazt a funkcionalitást nyújtja (mióta a Tangent Bug algoritmust is továbbfejlesztettem például az akadálykövetési változtatásokkal), noha a Tangent Bug-nál általános esetben az akadálykövetésnél kismértékben gyorsabb futás figyelhető meg. Az első és a második pár között megfigyelhető egy nagy eltérés: az utóbbi egy nagyságrenddel lassabb, azonban jóval biztonságosabb és megbízhatóbb.

8.2. Értékelés

A munka kezdetén célként szerepelt a kezdeti módszer megértése és implementálása, illetve egy olyan algoritmus létrehozása, amely differenciális meghajtású, LIDAR érzékelővel ellátott robot vezérlésére képes.

A kezdeti módszer implementálása során észrevettem, hogy az valójában csak statikus akadályok kezelésére képes, illetve nem ültethető át hatékonyan differenciális meghajtásra a módszer, mert benne a tervezés fontos összetevője az omnidirekcionális robotmodell, így ennek megfelelően kialakítottam a saját algoritmusomat a feladat megoldására.

A saját algoritmus gyengeségeit a Tangent Bug algoritmus átalakítása után annak mintájára oldottam meg, így végül a kapott algoritmus megbízható és biztonságos lett, úgy, hogy egy kategóriában van számítási kapacitás terén a mintát adó Tangent Buggal. Ezt az eredményt én jónak értékelem, és elégedett vagyok a saját algoritmusokkal. A dinamikus környezet kezelését nem sikerült az eddigi munka időtartama alatt megbízhatóbban megoldanom, mint a kiindulási algoritmusnak, azonban ezt legalább kimutattam és dokumentáltam is.

8.3. Továbbfejlesztési lehetőségek

Érdemes lehet az implementáció optimalizálása a számítási idő csökkentésére, és akkor még alacsonyabb igénnyel kezelhető a pályatervezés. Hasonlóan fontos egy valódi roboton kipróbálni az algoritmust, és az ekkor felmerülő problémákat (zajos mérések, hibák a *dead reckoning*-ban...) is megoldani.

A továbbfejlesztési lehetőségek között megemlíteném az algoritmus egy nagy hátrányát: annak ellenére, hogy magabiztosan megbirkózik a statikus környezet nyújtotta kihívásokkal, és az akadályok alakjától is függetlenül navigál az ismeretlen környezetben, mindössze annyi előzetes információval, hogy mik az ő geometriai méretei, és a célhoz képest milyen koordinátán és milyen orientációval helyezkedik el, a dinamikus környezet esetében nem nyújt hasonlóan jó eredményeket. Ennek megfelelően érdemes kifejleszteni egy olyan algoritmust, amely megtartva a statikus területen elért jó eredményeket, mozgó tereptárgyak esetén is hasonló sikerrel képes elvezetni a robotot a célba.

Köszönetnyilvánítás

Ebben a fejezetben szeretnék köszönetet mondani konzulensemnek, Gincsainé Dr. Szádeczky-Kardoss Emesének, aki útmutatásával, irodalmi és képanyagaival és folyamatos segítségével lehetővé tette a dolgozat elkészültét.

Irodalomjegyzék

[1] Ifte Khairul Alam Bhuiyan: Lidar sensor for autonomous vehicle. Jelentés, 2017. 09, Technische Universität Chemnitz.

[2] Jérôme Barraquand – Jean-Claude Latombe: Robot motion planning: A distributed representation approach. *Int. J. Robotics Res.*, 10. évf. (1991) 6. sz., 628–649. p. URL https://doi.org/10.1177/027836499101000604.

[3] Howie Choset: Robotic motion planning: Bug algorithms. http://www.cs.cmu.edu/~motionplanning/lecture/Chap2-Bug-Alg_howie.pdf.

[4] Dieter Fox – Wolfram Burgard – Sebastian Thrun: The dynamic window approach to collision avoidance. *IEEE Robotics Autom. Mag.*, 4. évf. (1997) 1. sz., 23–33. p. URL https://doi.org/10.1109/100.580977.

[5] Nacer Hacene – B. Mendil: Autonomous navigation and obstacle avoidance for a wheeled mobile robots: A hybrid approach. *International Journal of Computer Applications*, 81. évf. (2013), 34–37. p.

[6] Ishay Kamon – Elon Rimon – Ehud Rivlin: Tangentbug: A range-sensor-based navigation algorithm. *Int. J. Robotics Res.*, 17. évf. (1998) 9. sz., 934–953. p. URL https://doi.org/10.1177/027836499801700903.

[7] Yoram Koren – Johann Borenstein: Potential field methods and their inherent limitations for mobile robot navigation. In *Proceedings of the 1991 IEEE International Conference on Robotics and Automation, Sacramento, CA, USA, 9-11 April 1991* (konferenciaanyag). 1991, IEEE Computer Society, 1398–1404. p. URL https://doi.org/10.1109/ROBOT.1991.131810.

[8] Nick Malone – Hao-Tien Chiang – Kendra Lesser – Meeko Oishi – Lydia Tapia: Hybrid dynamic moving obstacle avoidance using a stochastic reachable set-based potential field. *IEEE Trans. Robotics*, 33. évf. (2017) 5. sz., 1124–1138. p. URL https://doi.org/10.1109/TRO.2017.2705034.

[9] Javier Minguez – Luis Montano: Nearness diagram (ND) navigation: collision avoidance in troublesome scenarios. *IEEE Trans. Robotics Autom.*, 20. évf. (2004) 1. sz., 45–59. p. URL https://doi.org/10.1109/TRA.2003.820849.

[10] Muhannad Mujahed – Dirk Fischer – Bärbel Mertsching: Admissible gap navigation: A new collision avoidance approach. *Robotics Auton. Syst.*, 103. évf. (2018), 93–110. p. URL https://doi.org/10.1016/j.robot.2018.02.008.

[11] Hani Safadi: Local path planning using virtual potential field. https://www.cs.mcgill.ca/~hsafad/robotics/, 2007. Apr.

[12] Andrey V. Savkin – Michael Hoy: Reactive and the shortest path navigation of a wheeled mobile robot in cluttered environments. *Robotica*, 31. évf. (2013) 2. sz., 323–330. p. URL https://doi.org/10.1017/S0263574712000331.

[13] Jeffery Young – Milan Simic: LIDAR and monocular based overhanging obstacle detection. In Liya Ding – Charles Pang – Mun-Kew Leong – Lakhmi C. Jain – Robert J. Howlett (szerk.): *19th International Conference in Knowledge Based and Intelligent Information and Engineering Systems, KES 2015, Singapore, 7-9 September 2015*, Procedia Computer Science konferenciasorozat, 60. köt. 2015, Elsevier, 1423–1432. p. URL https://doi.org/10.1016/j.procs.2015.08.218.