Budapesti Műszaki és Gazdaságtudományi Egyetem
Villamosmérnöki és Informatikai Kar
Automatizálási és Alkalmazott Informatikai Tanszék

SLAM algoritmus megvalósítása egyszerű távolságszenzorokkal

TDK Dolgozat

Készítette
Menyhart-Radó Dávid & Varga Adrián

Konzulens
Kiss Domokos

2016. október 27.

Tartalomjegyzék

Kivonat

A technológia fejlődésének köszönhetően ma már könnyedén hozzá lehet jutni jó minőségű szenzorokhoz, valamint komoly teljesítményű hordozható számítógépekhez és mikrokontrollerekhez is. Megbízható és megfizethető robotikai hardverek és szoftverek is jobban elérhetők, mint idáig a múltban valaha. Ezért egy olyan automóm robot megvalósítását tűztük ki célul, ami képes feltérképezni, szimulálni környezetét, ebben a szimulált környezetben meghatározni a saját helyzetét, és végül a saját maga által elkészített térképen önállóan eljutni valamilyen célpontba.

Ezen elképzelésünk megvalósításához saját szenzorrendszert fejlesztünk, ami képes megfelelő minőségű kétdimenziós pontfelhőket szolgáltatni, amelyekből már egy térkép készíthető és a robot lokalizálni tudja magát. Tulajdonképpen egy költéges LIDAR (laser scanner & rangefinder) szenzort helyettesítünk infravörös szenzorokkal és szervomotorokkal. Ezen rendszer, valamint a robotmodell egyéb részeinek is könnyen beszerezhetőnek kell lennie, ami az egyik alapvető feltételünk a téma kidolgozása során.

A navigációs feladat megoldásához szükséges, a szakirodalomban fellelhető algoritmusokat saját magunk implementáljuk, amelyek segítségével feldolgozzuk a robot különböző szenzorai által szolgáltatott adatokat. A mobil robotikában elterjedt SLAM (Simulteneous Localization And Mapping) algoritmus kiterjesztett Kálmán-szűrőn (EKF: Extended Kalman Filter) alapuló változatát használjuk, amely a robotunk szoftveres magját képezi. A Kálmán-szűrős algoritmus bemenetét a megvalósításunkban a szenzoradatokból kinyert vonalak képezik. A vonalak tulajdonképpen a feltérképezendő környezet falait és egyéb akadályait reprezentálják, és felfoghatjuk őket a környezet térképjellemzőinek.

A szenzorok által szolgáltatott pontfelhő adatokból a vonalakat egy már ismert split-and-merge algoritmussal nyerjük ki, aminek bemutatjuk egy általunk továbbfejlesztett változatát, az általunk vizsgált problémára optimalizálva. Dolgozatunk fő célja a rendszer és az algoritmusok szimulációs vizsgálata, a hardveres megvalósítás előkészítése céljából.

A szimulációkat V-REP (Virtual Robot Experimental Platform) segítségével valósítottuk meg. Dolgozatunkkal demonstráljuk, hogy egy egyszerű szenzorokkal felszerelt mobil robot is képes lehet autonóm navigációs funkciók megvalósítására.

Abstract

In today's world it is easy to gain access to quality sensors and very high performance computers and microcontrollers, thanks to recent advancements in technology. Reliable and affordable robot hardware and software components are more widely available than ever before. For these reasons, we have decided to develop a mobile robot that is able to map and simulate its own environment, inside of this environment can determine its own location, and is finally able to autonomously plan its own movement to arrive at certain coordinates on the map.

To accomplish these goals we have developed a sensor system of our own, which is able to provide satisfactory 2D point clouds, from which a map can be created in which the robot can perform its own localization. In other words, we are replacing a costly LIDAR (laser scanner & rangefinder) sensor with infrared distance sensors and servo motors. One of our main principles (and conditions) is that all of our components throughout this development process must be easily obtainable.

To solve the problem of navigation we implement our own versions of the algorithms published in scientific literature, and we use them to process the data of the various sensors we employ. We use the EKF (Extended Kalman Filter) variant of the widespread SLAM (Simultaneous Localization and Mapping) algorithm used in mobile robotics, which conceives the core of the robot's software. The input of the Kalman filter algorithm in our implementation consists of arrays of lines. The lines, in practice, represent the features of the environment, and can be extracted from things such as walls and similar obstacles.

We use a split-and-merge algorithm to extract the features from the point clouds, and we present this algorithm in an improved form that we optimized for the tasks of our robot model. The main goal of this study is to analyze these algorithms in order to prepare them for physical implementation.

We perform the simulations using V-REP (Virtual Robot Experimental Platform), and our final objective is to demonstrate that a mobile robot equipped with only simple sensors is able to perform autonomous navigational functions as well.

Bevezető

A mai világban már sok területen használnak autonóm önjáró robotokat. Ezeknek egy része önállóan képes feltérképezni környezetét, így a robotnak nem szükséges egy előre megadott térképet használnia. Ilyen gyakorlati alkalmazásokra lehet példa az otthonokban önmaguktól takarító kis robotok, katasztrófák helyszínén alkalmazott autonóm drónok, vagy egy gyár területén önmagát navigálni képes targonca.

Ezek a robotok a környezetükről rendszerint LIDAR segítségével szereznek információt, pontfelhők formájában. A robot utána ezekből a precíz pontfelhőkből képes lokalizálnia magát és térképet készíteni. A mi elképzelésünkben LIDAR helyett infravörös szenzorokat és szervomotorokat alkalmazunk a környezet pontfelhővé való leképezéséhez. Ennek a megközelítésnek egy nagy előnye, hogy míg a LIDAR-ok általában igen nagy és nehezen beszerezhető eszközök, a mi szenzoraink iparilag igen elterjedtek, és könnyen beszerezhetőek. A lokalizáció és a térképalkotás problémáját egy SLAM algoritmus implementálásával oldottuk meg, ami kiterjesztett Kálmán szűrőt használ. Ez a Kálmán szűrő vonalak segítségével határozza meg a robot pozícióját, a vonalakat pedig az infravörös szenzorok által szolgáltatott pontfelhőből nyerjük ki. A vonalak kinyeréséhez szintén saját szoftveres megoldást kellett implementálnunk, ami képes a zajos pontfelhőkből megfelelően kinyerni a vonalakat.

Mivel a végső célunk egy önjáró robot megalkotása, ezért még egy útvonaltervező algoritmust is meg kell valósítanunk. Útvonaltervezéshez egy Rapidly Exploring Random Trees nevű algoritmust használtunk fel.

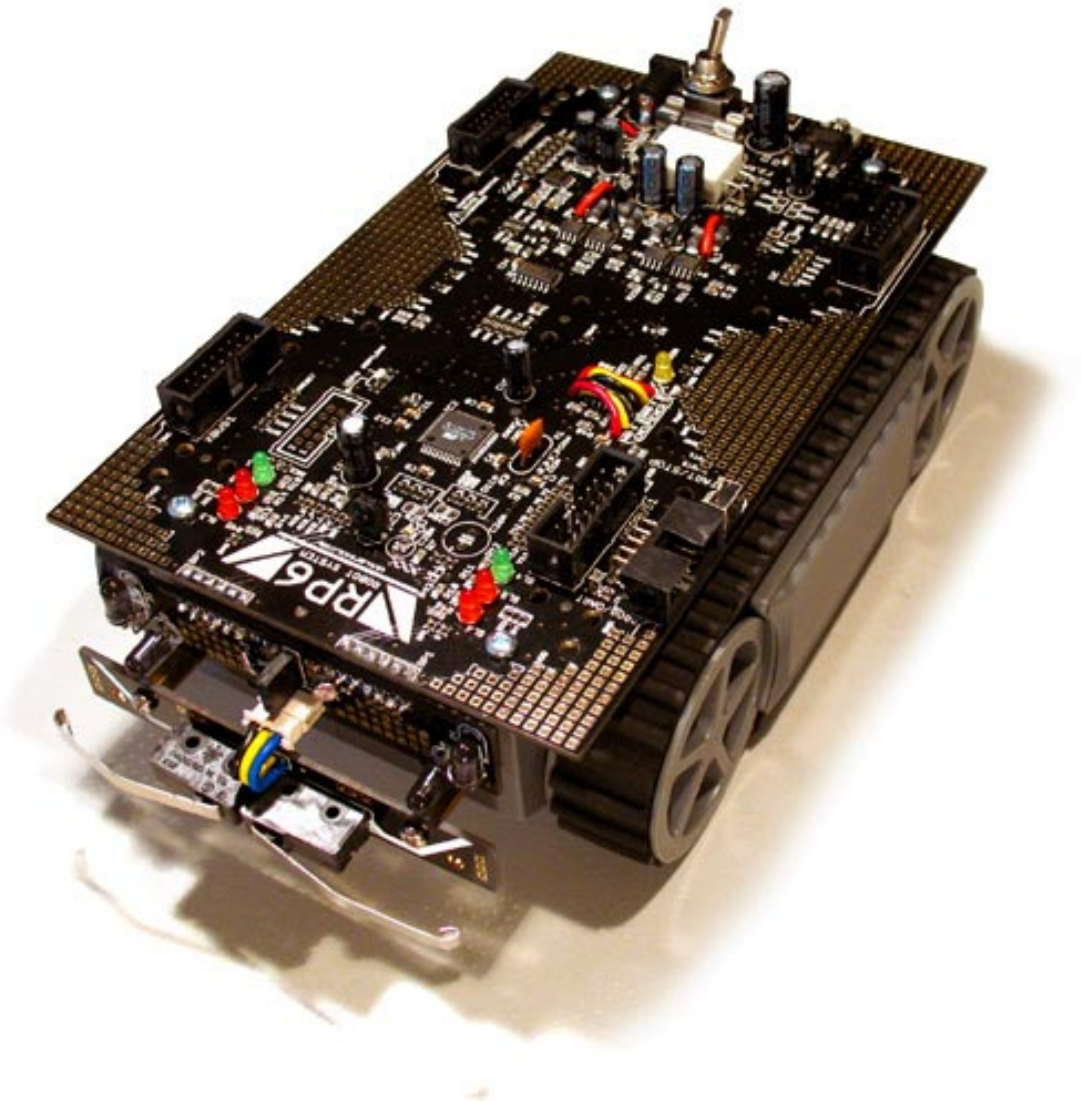
A fent említett szoftverek implementációját első körben szimulációs tesztek alá vetettük, hiszen a tesztelés folyamatát a valós roboton elég nehézkes megvalósítani. Az elkészített algoritmusokat így először is egy V-REP nevű szimulációs környezetben teszteltük, gondosan ügyelve rá hogy a fő algoritmusok implementációi ne igényeljenek változtatásokat valós környezetbe történő áthelyezésnél.

A szimulációhoz a különböző problémák megoldására szolgáló szoftvereknek együtt kell működniük. A szoftverek közti együttműködéshez egy ROS nevű keretrendszert használtunk fel.

1. fejezet

A vizsgált robot

1.1. Differenciális meghajtás szabályzása



1.1. ábra. *A robot.*

A robot amivel megszeretnénk valósítani a terveinket, egy RP6-os lánctalpas kis robot, ahogyan az 1.1-es képen is láthatjuk. Ehhez a robothoz jár egy alapkönyvtár, amiben alapfunkciók érhetőek el, mint például a perifériák kezelése, encoderek értékeinek lekérdezése, motorok sebességének beállítása. Továbbá megvalósítottak olyan alapfüggvényeket, mintpéldául az előre-hátra menet, illetve forgást jobbra balra. Ezek a funkciók azonban nem használnak semmilyen szabályzást, ebből kifolyólag a robot mozgása eléggé pontatlan.

Ahhoz, hogy a robotot megfelelően tudjuk majd használni egy szabályzót kell megvalósítanunk, ami egyszerre szabályozza a kerekek forgási sebességét és azokat szinkronban tartja egymáshoz képest. A kerekek szinkronban tartása arra utal, hogy a kerekek ugyan annyit forogjanak. Erre azért van szükség, hogy a robot előre haladáskor pontosan előre haladjon, forgásnál pedig lehetőleg egy pontban forduljon. Ezeket a feltételeket egy állapotteres szabályzással tudjuk biztosítani.

Elsőnek nézzük meg, egy egyenáramú motornak az állapotteres leírása. Az állapotváltozók a következőek $x = (\Psi \ \Psi' \ i_r)$, ahol a Ψ a motor tengelyének a szögelfordulása, a Ψ' a tengely szögsebessége az i_r pedig a motor tekercsében folyó áram. A kimenet $y = \Psi$ lesz. Az egyenáramú motor modelljét az alábbi mátrix és vektor definiálja, ahogy azt a [3] Szabályozástechnika gyakorlatok című könyvben is olvashatjuk.

$$A = \begin{bmatrix} 0 & 1 & 0 \\ 0 & -f/\Theta & c_2/\Theta \\ 0 & -c_1/L_r & -R_r/L_r \end{bmatrix} \qquad B = \begin{bmatrix} 0 \\ 0 \\ 1/L_r \end{bmatrix}$$

Az f a súrlódás, a Θ a forgórész tehetetlenségi nyomatéka, a c_2 a nyomatékállandó, a c_1 a sebességállandó, az L_r pedig a motor kapcsai között mérhető indukció.

Ez a modell csak egy motort ír le. Ahhoz, hogy a robot teljes szabályzását meg tudjuk valósítani ki kell találnunk egy új modellt magára a robotra. Ennek érdekében létrehozok egy új állapottervet, ami megfelelően lemodellezni a robotot szabályozás szempontjából. Az új állapotterem az $x = (\Delta\Psi, \Psi'_L, i_{Lr}, \Psi'_R, i_{Rr})$, ahol a $\Delta\Psi = \Psi_L - \Psi_R$, vagyis a két motorom szögelfordulásának különbsége. Az új állapottérhez az új mátrixok az alábbiak.

$$A = \begin{bmatrix} 0 & 1 & 0 & -1 & 0 \\ 0 & -f_L/\Theta_L & c_{L2}/\Theta_L & 0 & 0 \\ 0 & -c_{L1}/L_{Lr} & -R_{Lr}/L_{Lr} & 0 & 0 \\ 0 & 0 & 0 & -f_R/\Theta_R & c_{R2}/\Theta_R \\ 0 & 0 & 0 & -c_{R1}/L_{Rr} & -R_{Rr}/L_{Rr} \end{bmatrix} \quad B = \begin{bmatrix} 0 & 0 \\ 0 & 0 \\ 1/L_{Lr} & 0 \\ 0 & 0 \\ 0 & 1/L_{Rr} \end{bmatrix}$$

A kimeneteket pedig a $\Delta\Psi$-nek és a Ψ'_L-t választom, amihez az alábbi mátrix tartozik.

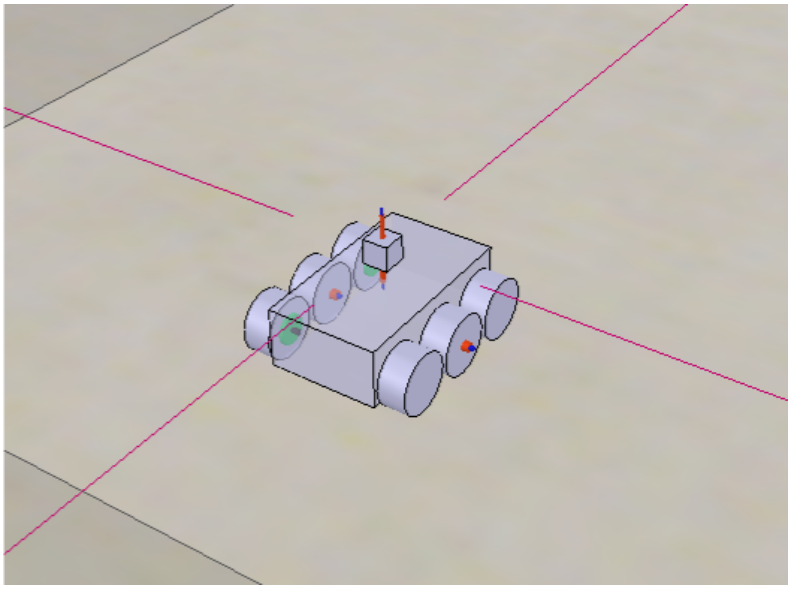
$$C = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \end{bmatrix}$$

Ebből az új rendszermodellből már tudunk megfelelő állapotteres szabályzást alkotni, ami egyszerre szabályozza a motorok sebességét és szinkronban is tartja őket. Az állapotteres szabályzó egy integrálóval is ki lesz egészítve, hogy lehetőleg nulla hibával tartsa a kívánt sebesség és irányt.

Ennek a szabályzónak a megvalósításnak akkor állunk neki, ha a szimulációs környezetben a robot már megfelelően működik és elkezdhetjük tesztelni a valós roboton az algoritmusokat.

1.2. A szimulációhoz használt modell

A 1.2-es ábrán látható a modell, amit a szimulációknál fogunk használni. Az első szembetűnő különbség a valódi robot és a modell között, hogy a modellnek nincsenek lánctalpai. Ez azonban nem baj, hiszen a használt modell és a valós robot mind a ketten differenciális robotként viselkednek, tehát a szimulációhoz használt modellt nyugodtan alkalmazhatjuk.



1.2. ábra. *RP6 modellezése a szimulációs környezetben.*

Következő lépésben meg kell valósítanunk, hogy a modell hasonló módon viselkedjen, mint a valós robot, ehhez meg kell adnunk azon parancsok listáját, amiket majd a valós robotnak is fogunk adni. A mi parancslistánk az alábbi négy parancsot tartalmazza:

- **Haladj előre.** Ennél a parancsnál a robotnak megadjuk, hogy mennyit kell előre mennie.
- **Fordulj jobbra.** Itt a parancs paramétereként a kívánt szögelfordulást adjuk meg.
- **Fordulj balra.** Itt szintén a szögelfordulást adjuk meg.
- **Odometria lekérdezése.** Ennél a parancsnál a robot visszaadja az odometriából a pozícióváltozásra becsült eltolási vektort.

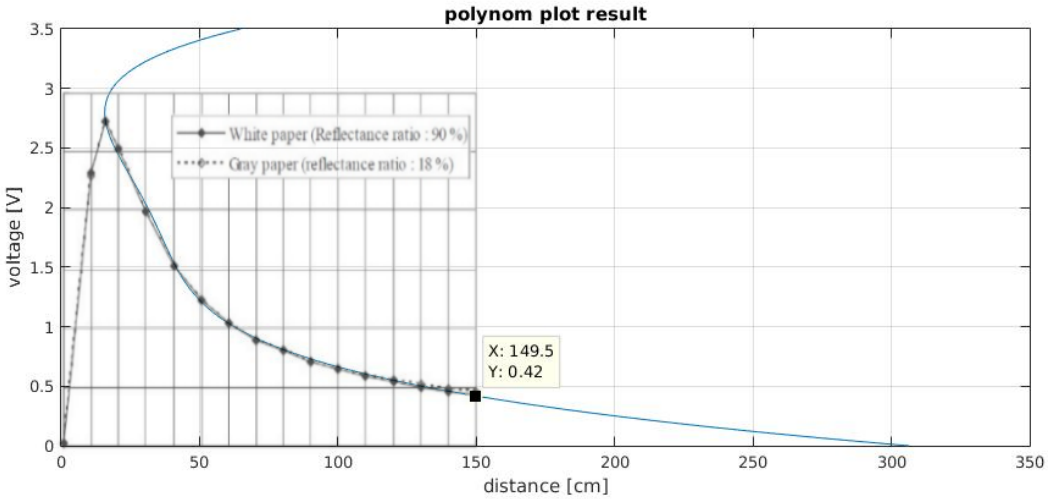
A modellnek ezeket a parancsokat kell felismernie és végrehajtania. A szimulációs környezet, amit használunk (V-REP) lehetővé teszi, hogy egy adott modell viselkedését egy Lua script segítségével meg tudjuk határozni. Ezt a Lua scriptet úgy írtuk meg, hogy a fenti parancsoknak megfelelően viselkedjen.

Ennek az elgondolásnak köszönhetően kapunk egy interface-t amivel a valós robotot és a szimulált robotot is tudjuk irányítani. Ez meg fogja könnyíteni az átállást a szimulált modellről a valós robotra. Ehhez az átálláshoz azonban még meg kell oldanunk egy problémát. A szimulált robot interface-sze socket kommunikációt használ. Ez a szimulációhoz megfelelő, de a valós roboton ezt közvetlenül nem alkalmazhatjuk, hiszen a roboton csak egy mikrokontroller található. Ezen probléma leküzdéséhez közbe kell iktatnunk egy réteget a robot mikrokontrollere és a számítógépen futó szoftverek közzé. Ennek a rétegnek az lesz a feladata, hogy a socket kommunikáción keresztül jövő parancsokat továbbítsa a robotnak. Ehhez a feladathoz mi egy Raspberry Pi 3-at használunk. A Pi 3-nak köszönhetően meg tudjuk oldani, hogy a valós robotot is egy socket kommunikációs interface-en keresztül lehessen irányítani. A Pi 3 a socket-en keresztül megkapott parancsokat UART-on továbbítja a robotnak. Most már könnyedén át tudunk állni a szimulált modellről a valós robotra.

1.3. Szenzorok

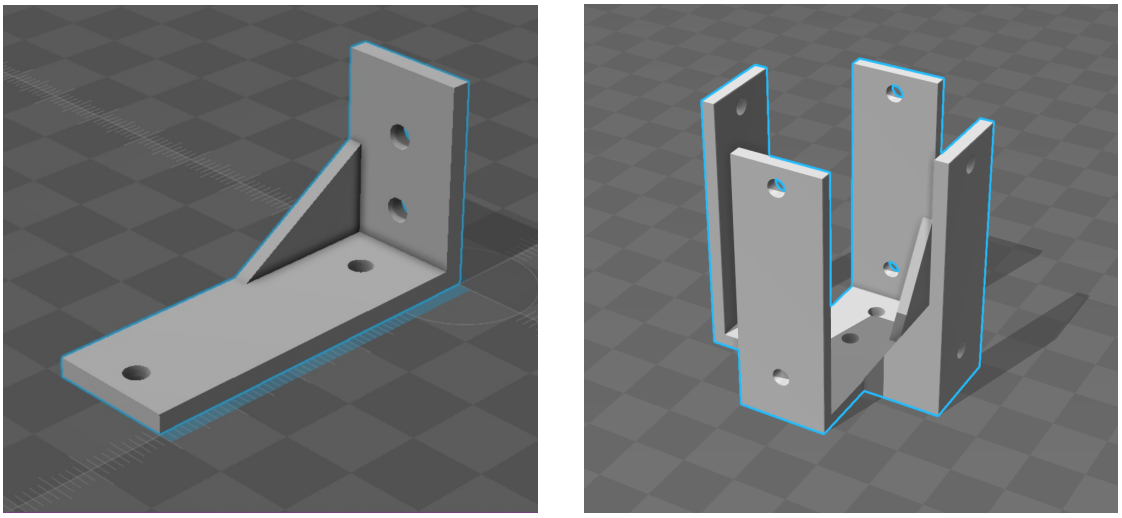
Ahhoz hogy a robot mozgását követni tudjuk, szükségünk volt egy szenzorrendszerre, amivel a környezetről adatokat gyűjthetünk. Mivel egy egyszerű, és könnyen beszerezhető rendszerre törekedtünk, választásunk az infravörös távolságszenzorokra esett. 4 darab ilyen egyszerű szenzor, és egy 90 fokot elfordulni képes szervómotor segítségével a robot környezetét már 360 fokban lehet pásztázni.

Az "r" távolságadatokat a szenzorok, a hozzájuk tartozó "α" szögeket pedig a szervó beállási szöge tudja biztosítani. Mivel az ilyen egyszerű szenzorok csak analóg feszültséget adnak a távolság arányában, elsőként is Matlabbal illesztettünk egy 5-öd fokú polinomot a szenzor karakterisztikájára, természetesen az ADC konverzió megvalósítása után.



1.3. ábra

Ahhoz hogy a szenzorokat használhassuk, szükségünk volt egy állványra amire a 4 szenzort felhelyezhetjük. Ezeknek a 3D modelljeit az AUT tanszék 3D nyomtatójával nyomtattuk ki. A modellek az 1.4 ábrán láthatóak.



1.4. ábra. *A bal ábrán egy kezdeti stádiumú, a jobb oldalon egy későbbi stádiumú állapota látható a kovariancia mátrix bal-felső sarkának. A kék szín a nullához közeli értékeket jelöli.*

2. fejezet

A robot kinematikája

Mielőtt egy robotra bármiféle komolyabb algoritmus implementálható lenne, szükséges hogy a rendszer viselkedése kinematikailag már értelmezhető legyen. Ebből következőleg ez a fejezet, erre a célra, egy általános megközelítést igyekszik bemutatni, majd annak segítségével elvégezni a levezetését a vizsgált differenciális meghajtású robotnak.

Elsődleges célként érdemes megállapítani a robot sebességvektorát a különböző, valós implementáció során is mérhető, paraméterek alapján. A függvény általános alakja:

$$\dot{\xi}_{\mathbf{I}} = \begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \end{bmatrix} = f(l, r, \theta, \dot{\phi_1}, \dot{\phi_2}) \tag{2.1}$$

- ahol "l" a lánctalpak távolsága a középponttól
- "r" a lánctalpak "sugara"
- θ a robot orientációja (2.1 ábra)
- $\dot{\phi_1}$ és $\dot{\phi_2}$ a lánctalpak "szögsebessége"

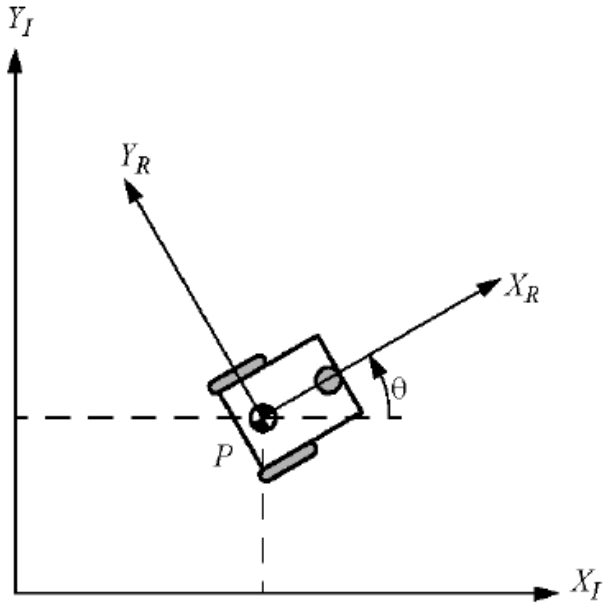
$\dot{\xi}_{\mathbf{I}}$ a robot sebességvektorát jelöli az alsó indexben található "I" pedig a globális koordináta rendszert. A robot sebességvektorát közvetlenül csak a saját koordináta rendszerére tudjuk számolni, így szükségünk lesz egy olyan 2 dimenziós forgatási mátrixra, amellyel két rendszer között transzformáció létesíthető: $\dot{\xi}_{\mathbf{I}} = \mathbf{R}(\theta)^{-1} \cdot \dot{\xi}_{\mathbf{R}}$

$$\mathbf{R}(\theta) = \begin{bmatrix} \cos(\theta) & \sin(\theta) & 0 \\ -\sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{2.2}$$

A következő lépésben fel kell tárnunk azokat a kinematikai korlátozásokat, amelyek a robot mozgását befolyásolják. A modell egyszerűsítése érdekében érdemes a lánctalpakat kerekekként értelmezni. Az előző felsorolás is jóval több értelmet nyer a lánctalp helyett a

"kerék" szó behelyettesítésével, és így áttérhetünk egy általános differenciális meghajtású robot vizsgálására.

Így tehát a következő lépésben a robotunk kerék-modelljeinek viselkedését fogjuk elemezni, annak reményében hogy a későbbiekben ezek segítségével eljuthatunk majd a mozgásegyenlethez.



2.1. ábra. *a globális és a robot koordináta rendszer [6]*

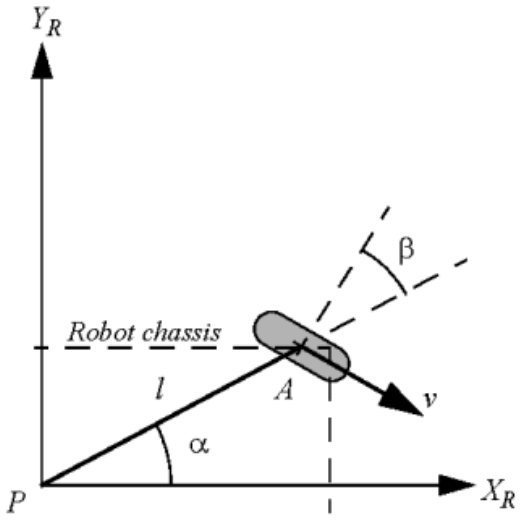
2.1. Kinematikai korlátozások

Többfajta kerék típus is használatos a robotikában, ezek közül az egyik legérdekesebb talán a svéd kerék (mecanum), melynek felhasználásával omnidirekcionális robot is építhető [4]. Ezek leggyakrabban kutatási területeken láthatóak. Egy népszerű választás az elfordítható kerék, ami a személygépjárművek elején megtalálható.

A robotunkon található lánctalpak azonban nem kormányozhatóak, így azokat egy rögzített kerékkel modellezhetjük. Ez azért fontos mert a 3 különböző kerék típusra más-más egyenlet írható fel.

Az első egyszerűsítő feltétel egy rögzített kerék kinematikai korlátjainak leírásában, hogy a kerék nem csúszhat forgás közben. Ez az úgynevezett rolling constraint.

$$\begin{bmatrix} \sin(\alpha_i + \beta_i) \\ -\cos(\alpha_i + \beta_i) \\ (-l_i) \cdot \cos(\beta_i) \end{bmatrix}^{\mathsf{T}} \cdot \mathbf{R}(\theta) \cdot \dot{\xi}_{\mathbf{I}} - r_i \cdot \dot{\phi} = 0 \tag{2.3}$$



2.2. ábra. *egy fix kerékhez tartozó konstansok robot koordináta rendszerben [6]*

A második feltétel pedig azt fejezi ki hogy a kerék nem képes a saját síkjára merőlegesen mozogni, hanem csak a kerék élének irányába. Ez a sliding constraint.

$$\begin{bmatrix} \cos(\alpha_i + \beta_i) \\ \sin(\alpha_i + \beta_i) \\ l_i \cdot \sin(\beta_i) \end{bmatrix}^{\mathsf{T}} \cdot \mathbf{R}(\theta) \cdot \dot{\xi}_{\mathbf{I}} = 0 \tag{2.4}$$

Az α és a β paraméterek jelentése a 2.2 ábrán látható, azaz az első a kerék szögét jelzi a robot tengelyéhez képest, a második pedig a merőleges orientációtól való eltérést. A vizsgált robotra nézve tehát az első kerékre $\alpha_1 = \frac{\pi}{2}$, $\beta_1 = 0$ a második kerékre $\alpha_2 = -\frac{\pi}{2}$, $\beta_2 = 0$.

A két kerékre behelyettesítve így összesen 4 egyenletet kaphatunk, de mivel a kerekek párhuzamosak, ezért a (2.4) egyenlet csak egyszer adhat új információt, és így végül 3 független egyenlethez jutunk. Először is vizsgáljuk meg milyen elméleti következtetésekre juthatunk ezekből az egyenletekből.

Mivel a korlátozási egyenletekben sebességek szerepelnek, létezik közöttük olyan anholonom egyenlet amelyet nem lehet kiintegrálni a pozíciótól és a időtől függő zárt alakba. Ebből megállapíthatjuk hogy a rendszerünk maga is anholonom, ugyanis definíció szerint akkor anholonom egy rendszer, ha a korlátozási egyenletei között található anholonom egyenlet. Tehát végeredményül egy differenciális kinematikai leírást várhatunk csak, ami nem meglepő tekintve hogy a differenciális meghajtású robotoknak ez egy tulajdonsága.

Egy másik intuitívabb megközelítés az, hogyha a szabadsági fokokat elemezzük. A mozgási tér szabadságfoka 3, hiszen a robot pontos pozícióját 3 változóval, x, y és θ-val tudjuk leírni, ahol x, y értelemszerűen a 2 dimenziós hely, θ pedig a robot orientációja a térben. A robot maga viszont pillanatszerűen csak 2 változót tud befolyásolni, az y irányú sebességet, és a $\dot{\theta}$ szögsebességet, mivel közvetlenül a kerekekre merőlegesen mérve nem lehet sebessége (ilyen tulajdonsággal például a svéd kerék rendelkezne, amivel a mi robotunk nem rendelkezik). Mivel a robot 2 csak változót tud közvetlenül változtatni, ezért a differenciális szabadsági foka kettő, ami kisebb, mint a mozgási tér 3 szabadsági foka, és mivel akkor

és csak akkor lehet egy robot holonom, ha a mozgási tér szabadsági fokszáma megegyezik a differenciális szabadsági fokszámával, így okvetlenül anholonom.

Végül pedig a rendszer anholonomitásából származtatva arra a következtetésre tudunk jutni, hogy a robotunk kerekeinek forgását mérve csak közelítő (linearizált) megoldásokat tudunk adni a robot pozíciójának változásáról. Célszerű lesz tehát a kerekeket sűrűn mintavételezni.

2.2. Mozgásegyenlet

Az előző szekcióban tárgyalt 3 egyenletet fel lehet írni egyetlen egyenletként is, mátrixos formában:

$$\begin{bmatrix} \mathbf{J_1} \\ \mathbf{C_1} \end{bmatrix} \cdot \mathbf{R}(\theta) \cdot \dot{\xi}_\mathbf{I} = \begin{bmatrix} \mathbf{J_r} \cdot \phi \\ 0 \end{bmatrix} \tag{2.5}$$

- ahol $\mathbf{J_1}$ a 2 sliding constraint egyenletből kinyerhető 2x3-as mátrix
- $\mathbf{C_1}$ az egyetlen független sliding constraint egyenletből kinyerhető 1x3-as mátrix
- $\mathbf{J_r}$ a kerekek sugarát tartalmazó 2x2 diagonális mátrix
- ϕ pedig a kerekek szögsebességének vektora

A (2.5) egyenlet bal oldali tagjait invertálva belátható, hogy $\dot{\xi}_\mathbf{I}$ kifejezhető a kerekek szögsebességével. A szorzásokat elvégezve $\Delta s_1 = r_1 \cdot \Delta\phi_1$ és $\Delta s_2 = r_2 \cdot \Delta\phi_2$ segítségével, infinitezimálisan kicsi Δ méretű változásokra pedig az új pozíció értéke, a változás és az előző $\mathbf{p_0}$ pozíció összegeként pedig már számítható. Tehát hozzá jutottunk a keresett mozgásegyenlethez.

$$\mathbf{p}' = \begin{bmatrix} x_0 \\ y_0 \\ \theta_0 \end{bmatrix} + \begin{bmatrix} \frac{\Delta s_1 + \Delta s_2}{2} \cdot \cos(\theta_0 + \frac{\Delta s_1 - \Delta s_2}{2 \cdot l}) \\ \frac{\Delta s_1 + \Delta s_2}{2} \cdot \sin(\theta_0 + \frac{\Delta s_1 - \Delta s_2}{2 \cdot l}) \\ \frac{\Delta s_1 - \Delta s_2}{l} \end{bmatrix} \tag{2.6}$$

Érdemes még utoljára megemlíteni, hogy az utolsóként kapott leképzésnek sokszor éppen az inverzére van szükség, azaz arra, hogy egy ismert elmozdulás vektorhoz milyen keresett kerékelforgási értékek tartoznak. Az általános megközelítésnek köszönhetően, ennek megtalálása sem jelentett nagyobb gondot.

Az ebben a fejezetben leírtak szoftveres implementációjának köszönhetően, egy olyan dinamikus megoldáshoz jutottunk, amelynél a robot paramétereinek (mint például a kerekek szögének, távolságának, sugarának) változása nem szükségel komolyabb utánaszámolásokat, vagy változtatásokat a kódon, csupán egy-egy konstans érték átírását.

3. fejezet

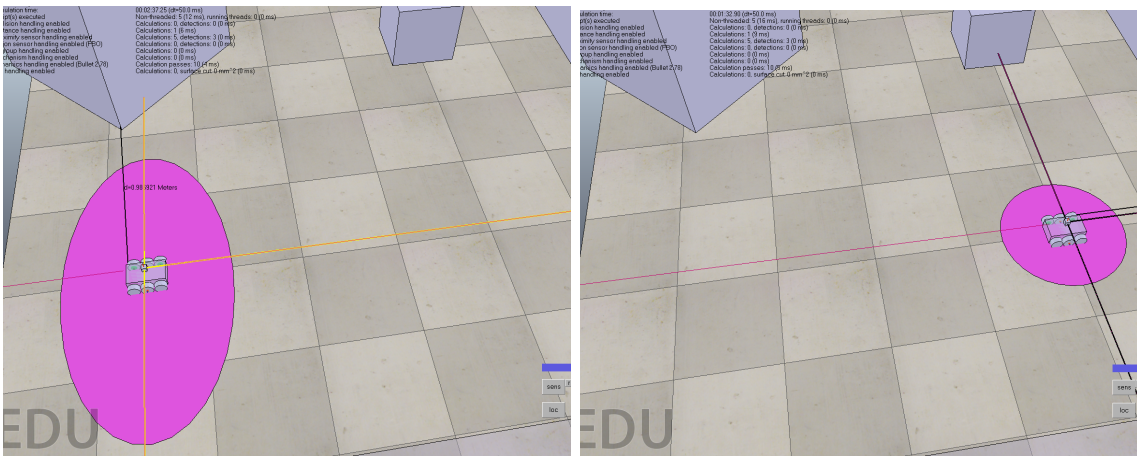
EKF-SLAM

3.1. Áttekintés

A vizsgált robotunk rendelkezik a lánctalpak forgását mérő encoder-ekkel (más néven odometriával), így az előző fejezetben tárgyaltak alapján már képesek vagyunk becsléseket adni mozgásáról. Annak az ötlete, hogy csupán csak erre a módszerre hagyatkozzunk, viszont több sebből is vérzik.

Egyrészt, mint az már említésre került, a robot holonom tulajdonsága miatt, a linearizált becsléseink az elmozdulás vektorokra még elméletileg sem lehetnek tökéletesen pontosak, amennyiben a kerekek egymáshoz viszonyított abszolút sebessége nem egyezik meg (görbe vonalú mozgás), vagy akár két mintavételezés között változik is. Ennek a problémának a kiküszöbölését a sűrű mintavételezésen kívül még azzal lehet orvosolni, hogy a robot mozgását limitáljuk egyenes vonalú, és középpont körül forgó mozgásokra. Mi is ezt a stratégiát választottuk.

A másik, ennél nagyobb probléma, hogy az odometria egy interoceptív szenzor, ami azzal jár, hogy a mérések haladtával a bizonytalanságai halmozódnak (3.1 ábra). Mivel általában a robot orientációjában, és így a mozgás irányában merül fel több hiba, ezért a robot mozgására merőlegesen növekszik jobban a bizonytalanság.



3.1. ábra. *A bizonytalanság halmozódásának szimulációja a robotunkon. [5]*

Erre a problémára egy lehetséges megoldást az első fejezetben tárgyalt infravörös távolságmérő szenzor nyújthat, amivel közvetlenül a környezetről tudunk távolság adatokat gyűjteni. Az ilyen típusú exteroceptív szenzoroknak már nem halmozódik a bizonytalansága.

Felmerül viszont a kérdés, hogy hogyan súlyozzuk az odometria és a távolságmérő szenzor adatait, hogy optimális pozíciót és bizonytalanságot nyerjünk a robot számára. Erre a kérdésre a különböző lokalizációs algoritmusok tudnak választ adni.

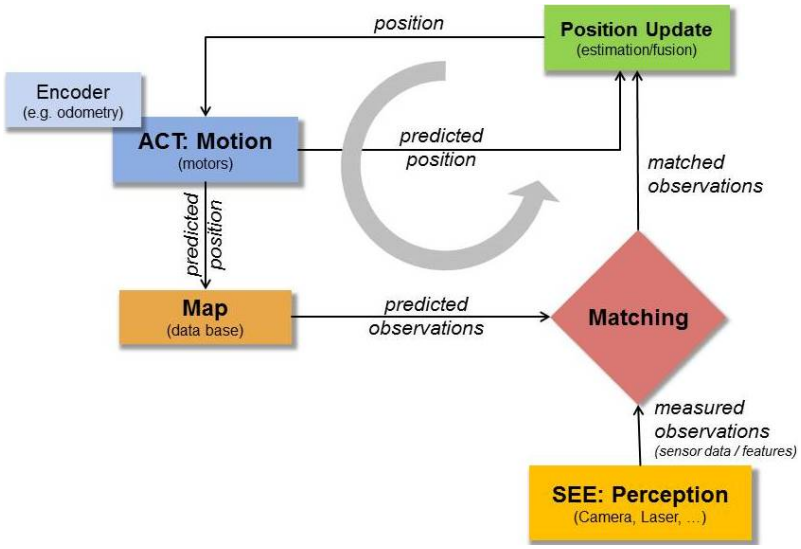
Több ilyen algoritmus is fellelhető a szakirodalomban. Ilyen például a részecske szűrős lokalizáció, amely nagyon jól tudja kezelni a nemlinearitásokat, a Markov lokalizáció, amihez tetszőleges bizonytalansági eloszlások rendelhetőek, és a Kálmán szűrős lokalizáció, amely Gauss eloszlások feltételezésével dolgozik. Tekintve hogy a Gauss eloszlás jól modellezi a szenzoraink hibáinak eloszlását, és hogy a kiterjesztett Kálmán szűrős megoldás a de facto módszer a jelenlegi navigációs rendszerekben, a választásunk erre esett.

3.2. EKF lokalizáció

Ebben az alfejezetben feltételezésre kerül hogy a robot környezete előre ismert, és a robot memóriája tartalmazza, és ennek az egyszerűsítésnek a segítségével fogjuk elemezni a lokalizációs algoritmust. Ez a feltételezés természetesen nem lesz érvényes későbbiekben, és a fejezet későbbi részei már a teljes problémával foglalkoznak.

A kiterjesztett Kálmán szűrő (EKF), és a sima Kálmán szűrő közti különbség, hogy az első a robot pozíciójának várható értéke és kovarianciája köré linearizál, és ezáltal tűrőképes a rendszer bizonyos szintű nemlinearitásaival szemben. Ezen felül viszont mindkettő egy dinamikus rendszer állapotváltozóinak megbecslését célozza meg, miközben ismert hogy a teljes statisztikus rendszerről a mérések segítségével csak részinformációkat kapunk, és azokat is valamilyen hibával.

Az algoritmus a nevét Rudolf Emil Kálmán magyar mérnök-matematikusról kapta, és a robotok navigációján kívül is számos felhasználása van.



3.2. ábra. *Az EKF lokalizációs ciklikus folyamatábrája [6]*

A 3.2 ábrán is megfigyelhetően, a kiterjesztett Kálmán szűrős lokalizáció folyamata az alábbi sorrendben, ciklikusan történik:

1. a robot egy adott pontban foglal helyet, adott bizonytalansággal
2. a robot változtatja pozícióját valamilyen elmozdulásvektor szerint (Motion)
3. a távolságmérő szenzorokkal az előre ismert térkép alapján a későbbiekben beazonosítható pontok (vagy egyéb geometriai adatok) kerülnek eltárolásra (Perception)
4. a robot lokalizálja magát a szenzor adatok fuzionálásával (Matching)
5. a robot új pozíciót számol, és bizonytalansága a felismert pontok szerint csökken (Position Update)

Tehát a robot kiindul egy már ismert pozícióból, és a kerekek forgását mérő encoder segítségével képes megjósolni a következőleg felvett pozícióját. Ez a becslés még meglehetősen nagy bizonytalanságú. Megérkezte után viszont a már előre ismert térkép adataiból azt is képes megbecsülni, hogy mit láthat ebben az új pozícióban a környezetéből. És miután egy újabb távolságképet alkotott a mostani környezetéről, ezeket az adatok összevetve a régi térképi adatokkal, a robot képes tenni egy újabb jóslatot a saját pozíciójára és bizonytalanságára. Ezt a két jóslatot pedig megfelelően súlyozva a szórásaik alapján a Kálmán szűrős lokalizáció képes egy optimálisabb pozíciót és bizonytalanságot rendelni a robothoz.

Most, hogy az algoritmus már koncepcionálisan felvázolásra került, térjünk rá az egész matematikai megvalósításra.

Az előző fejezetben levezetett mozgásegyenletnek köszönhetően fel tudjuk írni a robot elmozdulásvektorának megbecslését, az odometria adatai alapján, egy már egyszerűsített formában.

$$\hat{\mathbf{x}}_k = f(\mathbf{x}_{k-1}, \mathbf{u}_k) = \begin{bmatrix} x_0 \\ y_0 \\ \theta_0 \end{bmatrix} + \begin{bmatrix} \Delta x \cdot \cos(\theta_0 + \frac{\Delta\theta}{2}) \\ \Delta x \cdot \sin(\theta_0 + \frac{\Delta\theta}{2}) \\ \Delta\theta \end{bmatrix} \tag{3.1}$$

Ahol $\hat{\mathbf{x}}_k$ a becsült pozíció vektorát, $\mathbf{x}_{k-1}$ az előző pozíció vektorát, és $\mathbf{u}_k$ az encoder méréseiből kinyerhető (egyelőre még robot koordináta rendszerbeli) elmozdulásvektort jelöli. Az elmozdulásvektornak a robot koordináta rendszerében értelemszerűen nincs y irányú kiterjedése, hiszen a robot a kerekek irányára merőlegesen nem tud mozogni. Szükségünk lesz ennek a függvénynek még néhány Jacobi mátrixára is a későbbiekben. A Jacobi mátrix tömören fogalmazva a gradiens fogalmának kiterjesztése skalármezőkről vektormezőkre. Ezeknek az elemeknek a felhasználásának köszönhetően is csatolódik hozzá a "kiterjesztett" szó a kiterjesztett Kálmán szűrős lokalizációhoz, ugyanis ezeknek a mátrixoknak a segítségével tudunk a várható értékek köré linearizálni.

Először is nézzük a függvénynek a robot kiindulási pozíciója szerint vett Jacobi-ját:

$$\mathbf{F}_x = \frac{df}{d\mathbf{x}} = \begin{bmatrix} \frac{\delta f_1}{\delta x_1} & \frac{\delta f_1}{\delta x_2} & \frac{\delta f_1}{\delta x_3} \\ \frac{\delta f_2}{\delta x_1} & \frac{\delta f_2}{\delta x_2} & \frac{\delta f_2}{\delta x_3} \\ \frac{\delta f_3}{\delta x_1} & \frac{\delta f_3}{\delta x_2} & \frac{\delta f_3}{\delta x_3} \end{bmatrix} = \begin{bmatrix} 1 & 0 & -\Delta x \cdot \sin(\theta_0 + \frac{\Delta\theta}{2}) \\ 0 & 1 & \Delta x \cdot \cos(\theta_0 + \frac{\Delta\theta}{2}) \\ 0 & 0 & 1 \end{bmatrix} \tag{3.2}$$

Másodjára pedig szükségünk lesz még a függvénynek az elmozdulásvektor szerinti Jacobijára is:

$$\mathbf{F}_u = \frac{df}{d\mathbf{u}} = \begin{bmatrix} \cos(\theta_0 + \frac{\Delta\theta}{2}) & 0 & -\Delta x \cdot \frac{1}{2} \cdot \sin(\theta_0 + \frac{\Delta\theta}{2}) \\ \sin(\theta_0 + \frac{\Delta\theta}{2}) & 0 & \Delta x \cdot \frac{1}{2} \cdot \cos(\theta_0 + \frac{\Delta\theta}{2}) \\ 0 & 0 & 1 \end{bmatrix} \tag{3.3}$$

Az $f(\mathbf{x}_{k-1}, \mathbf{u}_k)$ függvénnyel kapcsolatos (azaz az odometriához kötődő) műveletek teljességére a továbbiakban predikcióként fogunk utalni, a szakirodalomnak megfelelően. A predikció utolsó lépéseként számítható egy becslés a robot új kovariancia mátrixára is:

$$\hat{\mathbf{P}}_\mathbf{k} = \mathbf{F}_x \cdot \mathbf{P}_{k-1} \cdot \mathbf{F}_x^\intercal + \mathbf{F}_u \cdot \mathbf{Q} \cdot \mathbf{F}_u^\intercal \tag{3.4}$$

Amiben mint látható felhasználásra került a robot korábbi kovariancia mátrixa, a megfelelő Jacobi mátrixokkal transzformálva, és egy új 3x3 mátrix, $\mathbf{Q}$, ami a zaj modelljét hivatott képviselni, és az encoder méréseinek feltételezett hibájával növelni a bizonytalanságokat, szintén a rá vonatkozó Jacobi-k segítségével.

Ez a kovariancia mátrix röviden összefoglalva a robot pozícióbeli bizonytalanságait foglalja egy adattömbbe. Az átlójában a robot x, y, és θ értékeinek varianciája található, az átlóján kívül pedig a kovarianciák, amik megadják hogy egy változónak a várható értékétől való eltérése hogyan befolyásolja egy másik változó eltérését. A kovariancia más szavakkal egy korreláció, amiből még nem lettek kiszedve a változók mértékegységei. A későbbiekben tárgyalt kovariancia mátrixok is ugyanezzel a struktúrával rendelkeznek.

A predikció végeztével áttérhetünk a algoritmus másik felére, az innovációra. Itt elsőként is szükségünk lesz egy függvényre, amelynek segítségével a robot belső memóriájában található térkép adatok segítségével, jóslatokat tudunk tenni arra, hogy ezek a pontok (térképjellemzők) a robot új pozíciójában hol kéne hogy elhelyezkedjenek, amennyiben a predikciónk becslése tökéletes volt.

$$\hat{\mathbf{z}}^j = h(\hat{\mathbf{x}}_k, \mathbf{m}^j) = \begin{bmatrix} \alpha_j - \hat{\theta}_k \\ r_j - (\hat{x}_k \cdot \cos(\alpha_j) + \hat{y}_k \cdot \sin(\alpha_j)) \end{bmatrix} \tag{3.5}$$

Ahol $\hat{z}^j$ a j-edik ismert térképjellemző jósolt pozícióját jelöli, $\mathbf{m}^j$ pedig a j-edik térképjellemzőnek a még globális koordináta rendszerben tárolt vektorát. Ezeket a geometriai

adatokat nevezhetjük egyelőre még pontoknak is, a tényleges jelentésük és jelentőségük a következő fejezetben fog majd tárgyalásra kerülni. A lényeg hogy ezek az adatok polár koordinátás struktúrában kerülnek eltárolásra, azaz a $[\alpha, r]$ szög, és távolság szerint.

A predikcióhoz hasonlóan itt is szükségünk lesz egy Jacobi mátrixra, méghozzá a h függvénynek a prediktált robot pozíció szerinti Jacobi-jára.

$$\mathbf{H}_x = \frac{dh}{d\hat{\mathbf{x}}} = \begin{bmatrix} 0 & 0 & -1 \\ -\cos(\alpha_j) & -\sin(\alpha_j) & 0 \end{bmatrix} \tag{3.6}$$

Ezek után már nekiláthatunk az innovációs kovariancia mátrix kiszámításához.

$$\mathbf{Z}^{ij} = \mathbf{H}_x^j \cdot \hat{\mathbf{P}}_k \cdot (\mathbf{H}_x^j)^\intercal + \mathbf{R}^i \tag{3.7}$$

Ez az egyenlet struktúráját tekintve nagyon hasonló az előző kovariancia mátrixos egyenlethez. Itt $\mathbf{R}^i$ 2x2 mátrix az i-edik mérés bizonytalanságát veszi figyelembe. $\mathbf{Z}^{ij}$ tehát tulajdonképpen a j-edik felismert térképjellemzőnek a kovariancia mátrixát kell hogy megadja, az i-edik szenzoros mérésből. Következőnek már kiszámíthatjuk a Kálmán erősítést.

$$\mathbf{K} = \hat{\mathbf{P}}_k \cdot \mathbf{H}_x^\intercal \cdot (\mathbf{Z}^{ij})^{-1} \tag{3.8}$$

Ez a 3x2 mátrix fogja megadni hogy az előre ismert térképjellemzők ismételt felismerése milyen súllyal fogja befolyásolni a robotunk új pozícióját, és kovariancia mátrixát, aminek az új értéke

$$\mathbf{P}_k = \hat{\mathbf{P}}_k - \mathbf{K} \cdot \mathbf{Z}^{ij} \cdot \mathbf{K}^\intercal \tag{3.9}$$

lesz. Látható, hogy amennyiben $\mathbf{K}$ is pozitívan súlyozza a $\mathbf{Z}^{ij}$-ben okvetlenül pozitív varianciáit, akkor az egyenletben látható kivonás csökkenteni fogja a robotunk bizonytalanságát, ami pontosan az eredetileg kitűzött célunk is volt.

Természetesen itt még nem állhatunk meg, ugyanis a robotunk pozícióját is frissítenünk kell, hogy ezek a bizonytalanságok érvényesek lehessenek. A súlyozás itt is a Kálmán erősítés segítségével történik, de először is számoljuk ki magát az innovációt, azaz azt a vektort, ami megmondja, hogy mennyivel tért el a predikcióból jósolt térképjellemző pozíciója, a tényleges mérés szerintihez képest.

$$\mathbf{v}^{ij} = \mathbf{z}^i - \hat{\mathbf{z}}^j = \mathbf{z}^i - h(\hat{\mathbf{x}}_k, \mathbf{m}^j) \tag{3.10}$$

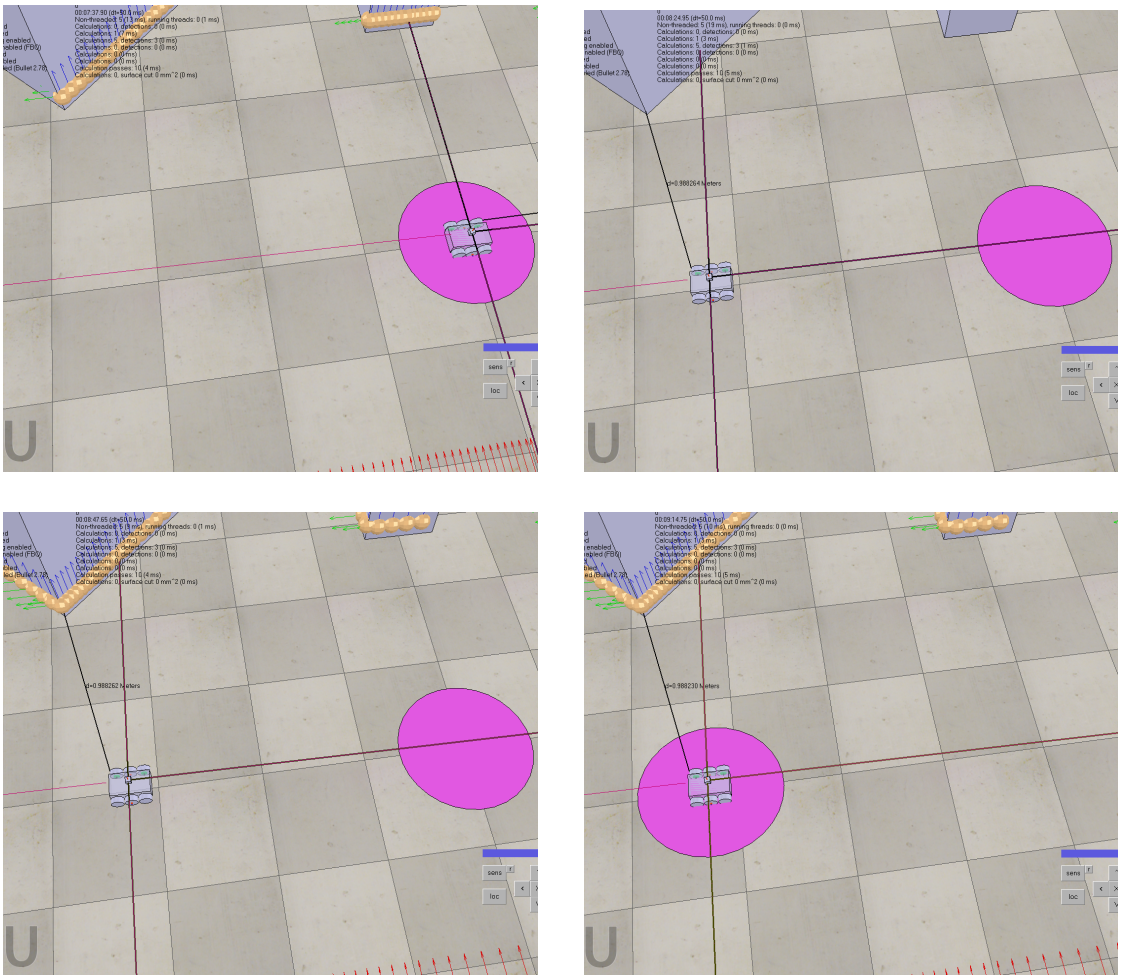
Itt $\mathbf{z}^i$ a tényleges i-edik szenzoros mérés vektorát jelöli. $\mathbf{v}^{ij}$ pedig a j-edik ismert térképadat, és i-edik mérés közötti eltérést adja meg, más néven az innovációt.

A kiterjesztett Kálmán szűrős lokalizáció legutolsó lépésként számoljuk ki a robot új, a megfelelő beállítások mellet optimálisan súlyozott pozícióját.

$$\mathbf{x}_k = \hat{\mathbf{x}}_k + \mathbf{K} \cdot \mathbf{v}^{ij} \tag{3.11}$$

$\mathbf{x}_k$-val tehát megkaptuk az eredményünket, és kezdődhet előröl a ciklus. Az algoritmus futásának V-REP-ben történő szimulációja a 3.4 ábrán látható.

Az utolsó (jobb alsó) képkockán látható update során az algoritmus 3 darab térképjellemzőt ismert fel. Összehasonlítva a 3.1 ábrán lévő szimulációval, könnyen felismerhető, hogy a bizonytalanság csökkent, és eloszlásának alakja is változott.



3.3. ábra. *a kiterjesztett Kálmán szűrős lokalizáció működésének V-REP szimulációja [5]*

3.3. Mahalanobis távolság

Ahhoz, hogy az előző alfejezetben is tárgyalt előre elmentett térképjellemzőket fel tudjuk ismerni, és hozzárendelni egy éppen lefuttatott szenzor méréshez, szükségünk lesz egy matematikai leírásra. Egy módszerre, aminek segítségével meg tudjuk mondani hogy egy térképjellemzőt milyen valószínűséggel találtunk meg újra. Erre a célre alkalmazható a Mahalanobis távolság.

A Mahalanobis távolság egy statisztikai módszer, amivel egy pontnak a hozzá rendelt eloszláshoz viszonyítva mérhető a távolsága. A pontunk jelen esetben a kép térképjellemző vektorának a különbsége lesz, az eloszlás pedig a éppen vizsgált mérésünkhöz tartozó innovációs kovariancia mátrix. A definíció szerint tehát:

Amennyiben $\mathbf{Z}$ identitás mátrix lenne, a képlet egyszerűen csak a térképjellemző és a mérés euklideszi távolságát adná meg. Ennek a módszernek segítségével viszont úgy tudunk súlyozni, hogy amennyiben a térképjellemzőnkhöz és a mérésünkhöz nagy bizonytalanság tartozik, akkor a felismerhetőség könnyű, amennyiben pedig kis bizonytalanság tartozik, nehéz legyen.

$$d = (\mathbf{v}^{ij})^{\intercal} \cdot (\mathbf{Z}^{ij})^{-1} \cdot \mathbf{v}^{ij} \tag{3.12}$$

Ahhoz hogy eldöntsük hogy tovább számoljunk-e ezzel a párosítással, szükségünk lesz egy szórás jellegű konstansra, amivel összehasonlíthatjuk a kapott értékünket.

$$d \lessapprox g^2 \tag{3.13}$$

A g konstans értékére csak empirikus módszerekkel lehet pontosabb becslést adni, miután a felhasznált szenzorok és a célzott tesztkörnyezet bizonytalanságainak kapcsolata már jobban ismert.

3.4. Kiterjesztett predikció

Eljutottunk odáig, hogy most már egy előre ismert környezetben a robot képes legyen lokalizálnia magát, a szenzorok mérésének adataiból. Ez hasznos, de ugyanakkor még nem az előre definiált probléma teljes megoldása. Valós alkalmazásokban is gyakran előfordul, hogy az adott környezetről csak részleges, vagy éppenséggel semmiféle előre megadott adatunk sincs. Ilyen lehet például egy turistát navigálni megkísérlő robot, egy katasztrófa során összedőlt épület beltere, és még sorolhatnánk.

Térjünk át tehát a szoftveresen implementált algoritmusunk egészére a Simultaneous Localization and Mapping (innentől SLAM) azaz az Egyidejű Lokalizációs és térképezős algoritmusra. Ennek segítségével a robotunk már képessé tud válni arra, hogy ismeretlen területen eligazodjon. Elsőre talán nem tűnik nagyon bonyolultnak a probléma, de képzeljük el, hogy pontosan mit is akarunk csinálni.

A robotunk elhelyezkedik valahol, valamilyen bizonytalansággal, majd erre a bizonytalanságra építve információkat szerzünk a környezetről, utána ezeket az információkat egy ismételten bizonytalan szenzoros méréssel visszaolvassuk, és ebből próbáljuk meg javítani a robot pozícióját, és bizonytalanságát, amiből eredetileg kiindultunk. Ez így már akár abszurdnak is tűnhet, de persze mindennapi életünk során, mozgás közben, agyunk is ugyanezt a problémát oldja meg, tehát biztosan megoldható.

Elemezzük elsőként a SLAM-hez kiterjesztett predikciós lépést. Az EKF lokalizáció tárgyalása során, az egyenletek klasszikus alakjait már láthattuk, így csak a szükséges változtatásokra lesz érdemes itt kitérni.

A robot pozíciójának vektorát ki kell egészíteni egy nagyobb állapotvektorrá, ami nem csak a robotnak, hanem a környezet térképjellemzőinek is képes lesz megadni és tárolni az elhelyezkedését.

$$\mathbf{y} = \begin{bmatrix} \mathbf{x} \\ \mathbf{f}_1 \\ \vdots \\ \mathbf{f}_j \\ \vdots \\ \mathbf{f}_N \end{bmatrix} = \begin{bmatrix} x \\ y \\ \theta \\ \alpha_1 \\ r_1 \\ \vdots \\ \alpha_N \\ r_N \end{bmatrix} \tag{3.14}$$

Mint az már említésre került $\mathbf{x}$ a robot pozíciójának vektora, a térképjellemzők pedig polár koordinátás struktúrában tárolhatóak a legkönnyebben. Itt összesen N darab térképjellemzőnk van, és az $\mathbf{f}$ betű az angol "feature" (térképjellemző) szóra utal, és annak polár koordinátás vektorát jelöli.

Ebből kifolyólag a robotunk kovariancia mátrixa is kiegészítésre szorul, ami így már nem csak a robotnak, hanem a térképjellemzőknek is képes lesz tárolni a bizonytalanságát (és természetesen a közöttük lévő összefüggéseket).

$$\mathbf{P} = \begin{bmatrix} \mathbf{P}_{xx} & \mathbf{P}_{xf} \\ \mathbf{P}_{fx} & \mathbf{P}_{ff} \end{bmatrix} = \begin{bmatrix} \mathbf{P}_{xx} & \mathbf{P}_{xf_1} & ... & \mathbf{P}_{xf_N} \\ \mathbf{P}_{f_1x} & \mathbf{P}_{f_1f_1} & ... & \mathbf{P}_{f_1f_N} \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{P}_{f_Nx} & \mathbf{P}_{f_Nf_1} & ... & \mathbf{P}_{f_Nf_N} \end{bmatrix} \tag{3.15}$$

A mátrix így már 4 fő részre osztható, az eredeti kovariancia mátrixra: $\mathbf{P}_{xx}$, a térképjellemzők kovariancia mátrixára: $\mathbf{P}_{ff}$, és a csupán kovarianciákat tartalmazó, a kétfajta adatok között összefüggést teremtő $\mathbf{P}_{fx}$ és $\mathbf{P}_{xf}$ részekre.

$$\mathbf{F}_x = \frac{df(\mathbf{y}, \mathbf{u})}{d\mathbf{y}} = \begin{bmatrix} \frac{\delta f}{\delta \mathbf{x}} & 0 \\ 0 & \mathbf{I} \end{bmatrix} \tag{3.16}$$

$$\mathbf{F}_u = \begin{bmatrix} \frac{\delta f}{\delta \mathbf{u}} \\ 0 \\ \vdots \\ 0 \end{bmatrix} \tag{3.17}$$

Következésképpen a (3.16) és (3.17)-nek megfelelően $f(\mathbf{y}, \mathbf{u})$ szerinti Jacobi mátrixokat is ki kell egészítünk, hogy az egyenleteink tagjai dimenzióikat tekintve továbbra is helyesek legyenek. Ezen felül a $\mathbf{F}_x$-et diagonálisan ki kell egészíteni egy $N \cdot 2$-es egységmátrixszal, hogy a nagy kovariancia mátrixban a térképjellemzőkről tárolt adatok változatlanok maradjanak (a robot pozíció változásának nincs hatása a globális koordináta rendszerben tárolt térképjellemzőkre).

$$\hat{\mathbf{P}}_{xx} = \frac{\delta f}{\delta \mathbf{x}} \cdot \mathbf{P}_{xx} \cdot [\frac{\delta f}{\delta \mathbf{x}}]^\intercal + \frac{\delta f}{\delta \mathbf{u}} \cdot \mathbf{Q} \cdot [\frac{\delta f}{\delta \mathbf{u}}]^\intercal \tag{3.18}$$

$$\hat{\mathbf{P}}_{xf} = \frac{\delta f}{\delta \mathbf{x}} \cdot \mathbf{P}_{xf} \tag{3.19}$$

A predikcióval kapcsolat mátrixok kiterjesztése után akár használhatnánk már a klasszikus EKF lokalizációs egyenleteket. Mivel viszont a Jacobi mátrixaink láthatóan elég ritkásak, lehetséges optimalizációkat végezni, és a kovarianciának csak a ténylegesen frissülő részeire korlátozni a számításokat. A potenciálisan igen nagy mátrixok jelenléte miatt, ez egy beágyazott alkalmazásban fontos lehet, és mivel a későbbiekben mi is hasonló erőforrás szűkösségre számítunk, mi is ennek megfelelően implementáltuk az algoritmusunk.

$$\hat{\mathbf{P}}_{fx} = \hat{\mathbf{P}}_{xf}^{\intercal} \tag{3.20}$$

A mátrix szimmetrikus, így $\hat{\mathbf{P}}_{fx}$, $\hat{\mathbf{P}}_{xf}$ transzponáltja. A (3.18), a (3.19) és a (3.20) eredményeit a (3.15) szerint kell eltárolni, és ezzel a predikció változásait lezárhatjuk.

3.5. Kiterjesztett innováció

Az innovációs lépésben is a predikció során felvázolt kiegészítésekhez hasonló változtatásokat kell tenni, azzal a különbséggel hogy itt a térképjellemzők sor szerinti elhelyezkedése miatt jobban oda kell rá figyelni, hogy a változtatások csak a megfelelő részekre korlátozódjanak.

Megjegyzendő még, hogy az EKF lokalizációhoz hasonlóan itt is egyszerre csak egy darab térképjellemző felismerésével lehet frissíteni az eredményeket.

$$\mathbf{H}^j = \begin{bmatrix} \frac{dh}{d\hat{\mathbf{x}}} & 0 & ... & 0 & \frac{dh}{d\mathbf{f}} & 0 & ... & 0 \end{bmatrix} = \begin{bmatrix} \mathbf{H}_x^j & 0 & ... & 0 & \mathbf{H}_f^j & 0 & ... & 0 \end{bmatrix} \tag{3.21}$$

$$\mathbf{H}_f^j = \begin{bmatrix} 1 & 0 \\ \hat{x} \cdot \sin(\alpha_j) - \hat{y} \cdot \cos(\alpha_j) & 1 \end{bmatrix} \tag{3.22}$$

A most már csak $h(\hat{\mathbf{y}})$ függvény Jacobi mátrixa változtatásra szorul, és ezúttal szükségünk lesz a térképjellemző vektora szerinti deriváltra is. A j index itt is a térképjellemző sorszámára utal, és a (3.21) egyenletben a bal oldali nulla oszlopok száma $(j-1)\cdot 2$ a jobb oldali nulla oszlopok száma pedig $(2\cdot N - j\cdot 2)$, ahol N az összesen elmentett térképjellemzők darabszámát jelöli.

$$\mathbf{Z}^{ij} = \begin{bmatrix} \mathbf{H}_x^j & \mathbf{H}_f^j \end{bmatrix} \cdot \begin{bmatrix} \hat{\mathbf{P}}_{xx} & \hat{\mathbf{P}}_{xf_j} \\ \hat{\mathbf{P}}_{f_j x} & \hat{\mathbf{P}}_{f_j f_j} \end{bmatrix} \cdot \begin{bmatrix} (\mathbf{H}_x^j)^{\intercal} \\ (\mathbf{H}_f^j)^{\intercal} \end{bmatrix} + \mathbf{R}^i \tag{3.23}$$

$$\mathbf{K}^j = \begin{bmatrix} \hat{\mathbf{P}}_{xx} & \hat{\mathbf{P}}_{xf_j} \\ \hat{\mathbf{P}}_{fx} & \hat{\mathbf{P}}_{ff_j} \end{bmatrix} \cdot \begin{bmatrix} (\mathbf{H}_x^j)^{\intercal} \\ (\mathbf{H}_f^j)^{\intercal} \end{bmatrix} \cdot (\mathbf{Z}^{ij})^{-1} \tag{3.24}$$

A már a predikció során említett optimalizálásokat, a (3.23) és a (3.24) egyenletekben is alkalmazni tudjuk. Így meg tudjuk oldani hogy hogy $\mathbf{Z}^{ij}$ ugyanúgy 2x2 dimenziójú legyen, a kiszámításának komplexitása is $O(1)$ maradjon, és $\mathbf{K}^j$ komplexitása is csak $O(n)$ szerint növekedjen $O(n^3)$ helyett. Az egyenleteken való kiigazodásban a (3.15) jelölései nyújthatnak segítséget.

$$\hat{\mathbf{P}}_k = \hat{\mathbf{P}}_k - \mathbf{K}^j \cdot \mathbf{Z}^{ij} \cdot (\mathbf{K}^j)^\intercal \tag{3.25}$$

$$\hat{\mathbf{x}}_k = \hat{\mathbf{x}}_k + \mathbf{K}^j \cdot \mathbf{v}^{ij} \tag{3.26}$$

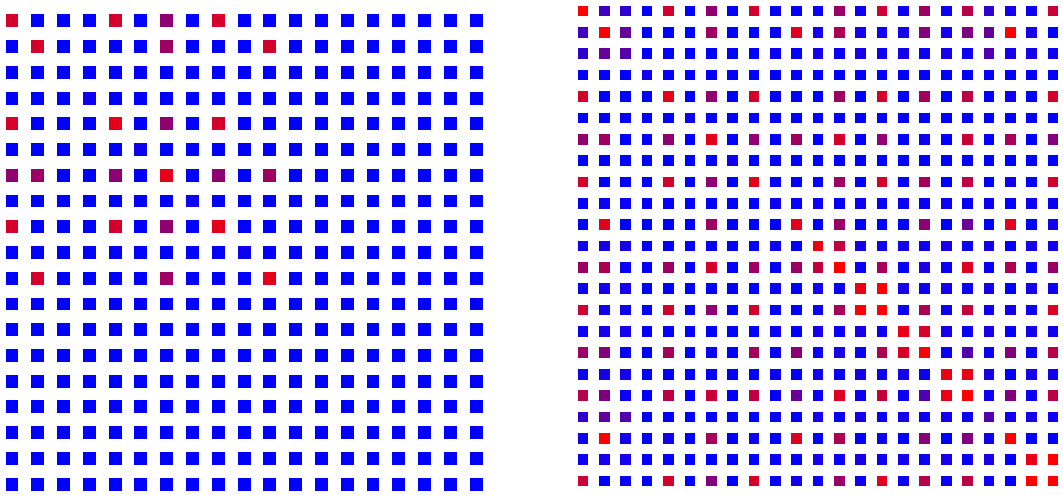
A végső számításokat ezek után már a klasszikus EKF lokalizáció egyenletei szerint elvégezhetjük, csupán a felhasznált mátrixok dimenziói növekedtek.

A végére jutottunk tehát egyetlen térképjellemző és az odometria egymáshoz súlyozásához, de természetesen a többi lehetségesen felismert térképjellemzőt is ajánlott felhasználni. Erre a célra az nem lehet megoldás, hogy az előző innovációs lépésben már az elejétől az összes térképjellemzővel és szenzoros méréssel együtt számolunk, hiszen hogyha például létezik 5 darab beazonosítás (match), amelyek szerint mind 10 centivel kéne odébb tolni a robot pozícióját, akkor ezek a változtatások a végén összegződnének és a robot fél méterrel kerülne odébb, hiszen a változtatások egy innovációs lépés közben nem tudnak egymásról.

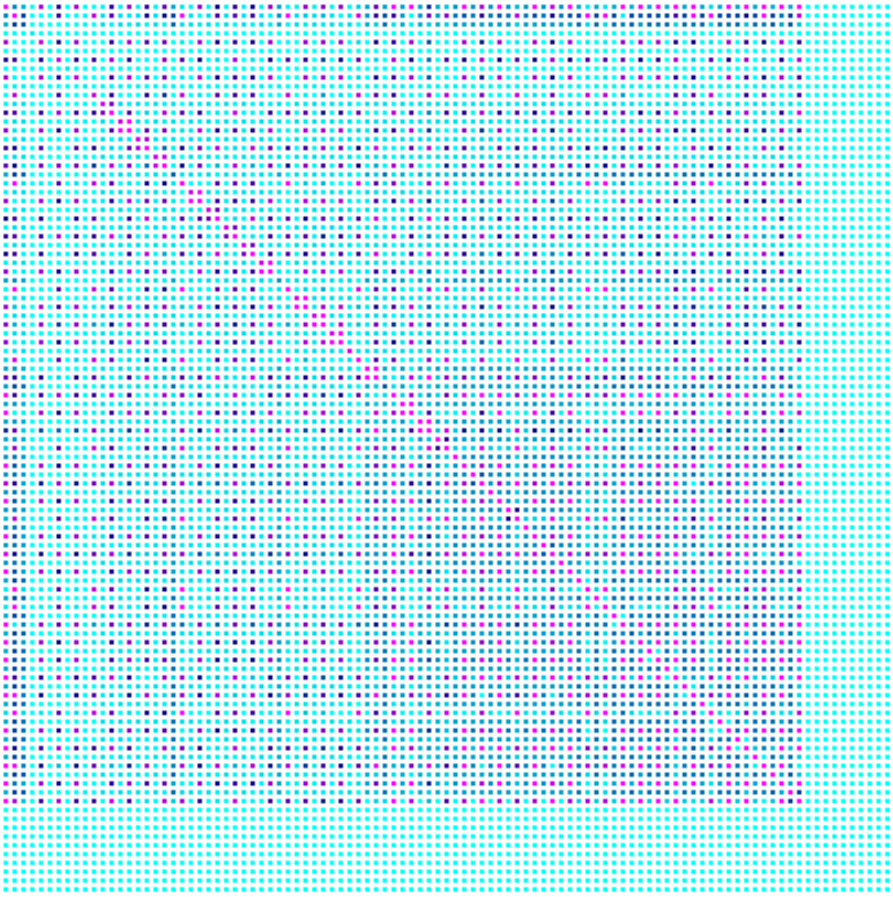
Pontosan ezért a (3.25) és a (3.26) egyenleteknek megfelelően vissza kell csatolnunk az eredményeinket, majd az innovációs lépést megismételni a következő szenzoros mérésre talált felismert térképjellemzőre.

A 3.4 szimulációt ismét elvégezve látszólag megkülönböztethetetlen vizuális eredményekre jutunk, ami algoritmusunk működésének helyességét igazolja.

Erre az algoritmusra azonban már egy másfajta tesztet is le tudunk futtatni, a teljes kovariancia mátrixban található adatok korreláltságának vizuális reprezentálásával. Ennek a tesztnek a létrehozását az OpenCV könyvtár segítségével írtuk. Lásd, Függelék: F.1.1



3.4. ábra. *A bal ábrán egy kezdeti stádiumú, a jobb oldalon egy későbbi stádiumú állapota látható a kovariancia mátrix bal-felső sarkának. A kék szín a nullához közeli értékeket jelöli.*



3.5. ábra. *A második stádium teljes kovariancia mátrixa, a színek kis eltolásával a jobb láthatóság érdekében, megfigyelhető az adatok "csomósodása" a jobban összefüggő térképjellemzőkön.*

3.6. Új térképjellemző felvétele

Az egyetlen olyan létfontosságú dolog amiről még nem esett szó a SLAM kapcsán az, hogy hogyan tudunk új térképjellemzőt felvenni a kiterjesztett állapot vektorunkhoz, és kovariancia mátrixunkhoz. Ezeket a teljes SLAM ciklus legvégén érdemes elmenteni, amikor már a robot új pozíciója és bizonytalansága tisztázódott.

Elsőként is szükségünk van az $f(\mathbf{y})$ függvény inverzére, azaz arra a függvényre ami egy szenzoros mérés alapján el tud menteni egy térképjellemzőt az állapot vektorba.

$$\hat{\mathbf{f}}^{N+1} = g(\mathbf{y}, \mathbf{z}^i) = \begin{bmatrix} \alpha_i + \hat{\theta}_k \\ r_i + (\hat{x} \cdot \cos(\alpha_i) + \hat{y} \cdot \sin(\alpha_i)) \end{bmatrix} \tag{3.27}$$

$$\mathbf{y} = \begin{bmatrix} \mathbf{y} \\ \hat{\mathbf{f}}^{N+1} \end{bmatrix} \tag{3.28}$$

Az eddigiekhez hasonlóan $\mathbf{z}^i$ jelzi az i-edik mérésünket, $N+1$ pedig az idáig elmentett térképjellemzők darabszámánál egyel nagyobb index.

Szükségünk lesz továbbá a függvény robot pozíció szerinti, és a mérés vektora szerinti Jacobi-jára.

$$\mathbf{G}_x^{N+1} = \frac{\delta g(\hat{\mathbf{x}}, \mathbf{z}^i)}{\delta \hat{\mathbf{x}}} = \begin{bmatrix} 0 & 0 & 1 \\ \cos(\alpha_i) & \sin(\alpha_i) & 0 \end{bmatrix} \tag{3.29}$$

$$\mathbf{G}_f^{N+1} = \frac{\delta g(\hat{\mathbf{x}}, \mathbf{z}^i)}{\delta \mathbf{z}^i} = \begin{bmatrix} 1 & 0 \\ \hat{y} \cdot \cos(\alpha_i) - \hat{x} \cdot \sin(\alpha_i) & 1 \end{bmatrix} \tag{3.30}$$

Miután ezekkel az előkészületekkel megvagyunk, a kiterjesztett kovariancia mátrixhoz már hozzá lehet fűzni a mérés 2x2 dimenziójú kovariancia mátrixát,

$$\mathbf{P}_{ff}^{N+1} = \mathbf{G}_x^{N+1} \cdot \mathbf{P}_{xx} \cdot (\mathbf{G}_x^{N+1})^\intercal + \mathbf{G}_f^{N+1} \cdot \mathbf{R}^i \cdot (\mathbf{G}_f^{N+1})^\intercal \tag{3.31}$$

és alulról hozzá lehet csatolni a (3.32) egyenletben kiszámolt 2x(N*2)-es mátrixot. Ennek segítségével lehet elmenteni az új térképjellemzőnek a robottal, és a többi térképjellemzővel létező lehetséges összefüggését.

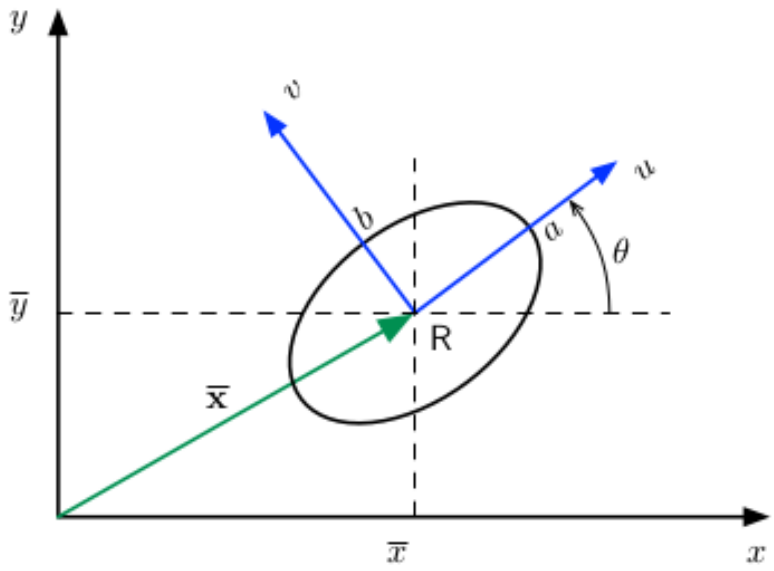
$$\mathbf{P}_{[f_{N+1}f_1 \ldots f_{N+1}f_N]} = \mathbf{G}_x^{N+1} \cdot \begin{bmatrix} \mathbf{P}_{xx} & \mathbf{P}_{xf} \end{bmatrix} \tag{3.32}$$

Végül ennek a transzponáltját jobbról is hozzá kell csatolnunk a most már mindkét dimenziójában 2-vel megnövekedő teljes, szimmetrikus kovariancia mátrixhoz.

[7]-ban található egy rövidebb ismertető arról is hogy hogyan lehet olyan mérési adatokat térképjellemzőként felhasználni, amikor a szenzor csak korlátozott szabadsági fokú információkat tud biztosítani (például monocular SLAM-nél).

3.7. Bizonytalansági ellipszis

Ahhoz, hogy a robotunk bizonytalanságát könnyen és egyszerűen nyomon követhessük a szimulációkon látottak szerint, szükségünk volt egy módszerre amivel annak kovariancia mátrixát vizualizálni tudjuk. Gauss eloszlású változóknál erre jelent megoldást a bizonytalansági ellipszis (3.6 ábra).



3.6. ábra. *a bizonytalansági ellipszis*

Az ellipszis paramétereinek kiszámításához ki kell vágnunk a kovariancia mátrix bal felső 2x2-es részét (azaz az x és y koordinátához tartozó közvetlen értékeket, a robot orientációjának bizonytalanságát nem ábrázoltuk), majd kiszámítani ennek az új mátrixnak a sajátértékeit és sajátvektorait. A nagyobb sajátértékhez tartozó sajátvektor adja meg az ellipszis θ orientációs szögét, amit az $atan2()$ függvénnyel könnyen meg lehet kapni. A fél nagytengely méretét a nagyobb sajátértékből lehet nyerni, 95%-os konfidencia intervallumnál a $2 \cdot \sqrt{5.991 \cdot \lambda_1}$ segítségével, ahol λ a sajátértéket jelöli. A kistengely hosszát ugyanígy λ_2 segítségével kapjuk.

A C++ projektünkben a sajátértékek és sajátvektorok kiszámításához a GNU Scientific Library-t használtuk.

A következő fejezet a már sokszor említett térképjellemzők általunk választott implementációjával foglalkozik, de érdemes megjegyezni hogy a SLAM algoritmus nem függ ettől a módszertől, a vonalakon kívül más térképjellemzőkkel is működhet.

4. fejezet

Vonal, mint térképjellemző

Ahhoz, hogy a robot lokalizálni tudja magát térképjellemzőket használ fel. Mi emberek is használunk egyfajta térképjellemzőket, hogy megtudjuk a pontos helyzetünket. Mi emberek többnyire az utcatáblákat használjuk, mint térképjellemző.

A robotok utcatábla gyanánt vonalakat tudnak használni. Ezt egy egyszerű példán be lehet mutatni. Tegyük fel, hogy a robotunk egy fal mellet halad. Induláskor a szenzoraiból kinyert pontfelhőből kiszámolta a mellette lévő fal paramétereit. A robot előre halad. A kerekek forgásából tud becslést adni, hogy mi lehet az új pozíciója, viszont ez a fajta becslés hibája folyamatosan nő. Ekkor a robotunk újból körbenéz és kiszámolja az új vonalparamétereket. Az elöször kiszámolt vonalparaméterek és az újból kiszámolt vonalparaméterek kicsit eltérnek egymástól, attól függően, hogy a robot menyit mozgott. Ha ez az eltérés kicsi, akkor a robot feltételezheti, hogy ugyan azt a vonalat látja, mint induláskor és ennek segítségével korrigálni tudja a helyzetét. Szóval a robot képes a szenzorokból kinyert pontfelhőkből kiszámolt vonalak segítségével újra pozícionálnia magát.

Ahhoz, hogy a vonalakat kinyerjük a pontfelhőből meg kell oldani a következő problémákat.

1. Mennyi vonalat látók?
2. Melyik pont melyik vonalhoz tartozik?
3. Az egy vonalhoz tartozó pontokból hogyan lehet kinyerni a vonal paramétereit?

A [6] Introduction to Autonomous Mobile Robots című könyvben lévő megoldásokat használjuk fel a fent említett problémák leküzdéséhez.

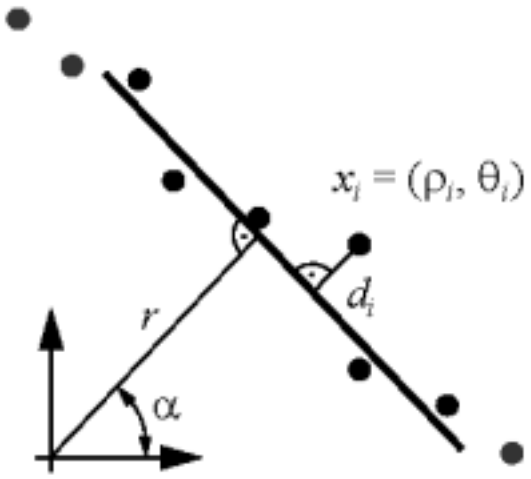
4.1. Vonal paraméterek kiszámolása

Az egy vonalhoz tartozó pontoknak van bizonytalansága, hiszen a mérés során a vonalhoz tartozó pontokat zaj terheli. Egy mérés eredményét a 4.1-es ábra szemlélteti. A pontokat polárkoordinátában használjuk, tehát egy pontot az origótól való távolságával és a vízszintes tengellyel bezárt szögével adjuk meg vagyis $x_i = (\rho_i, \Theta_i)$. Ebben a polárkoordináta rendszerben a vonalakat egy α r paraméterpárossal adunk meg, ahol az r az origót és a vonalat összekötő legrövidebb szakasz hossza, az α pedig az r szakasz vízszintessel bezárt szöge.

Egy mérés során szerzett n ponthoz szeretnénk a lehető legjobb vonalat illeszteni. Feltételezhetjük, hogy a pontok ρ és Θ paramétereinek eloszlása egy Gauss eloszlással jól közelíthető, tehát minden pontot feltudok fogni úgymint két független valószínűség változó.

$$X_i = (P_i, Q_i) \text{ ,ahol } P_i = N(\rho_i, \sigma_{\rho_i}) \text{ és } Q_i = N(\Theta_i, \sigma_{\Theta_i}).$$

Egy vonalat, ahogyan a 4.1-es ábra is mutatja egy α és r párossal adunk meg.



4.1. ábra. *[6]*

A 4.1-es ábrán látható minden x_i ponthoz ki tudjuk számolni a vonaltól való d_i távolságot.

$$\rho_i \cos(\Theta_i - \alpha) - r = d_i$$

Ha azt feltételezzük, hogy miden pontnak ugyan annyi a bizonytalansága, akkor a vonal és a pontok közötti távolságokat, vagyis tulajdonképpen a hibákat össze tudom adni. Mi most a hibák négyzetét adjuk össze. Az így képzett szumma megmutatja, hogy a pontokhoz illesztett vonal mennyire illeszkedik a pontokhoz.

$$S = \sum(\rho_i \cos(\Theta_i - \alpha) - r)^2$$

A célunk természetesen az, hogy megtaláljuk azt az α r paraméter párost, amiknél az S értéke a legkisebb legyen, vagyis úgyis felfoghatjuk, hogy keressük az $S(\alpha, r)$ függvény minimumát.

Ennél a felírásnálban még nem vettük figyelemben, hogy a pontoknak van szórása. A távolság szenzoroknál jellemző, hogy a távolság függvényében nő a pontok szórása. Ezt figyelembe véve [6] könyvre hivatkozva a szummában szereplő távolságokat súlyozzuk, mégpedig úgy, hogy az adott P_i ponthoz tartozó d_i távolság $1/\sigma_i^2$ súllyal számítson bele a szummába. Szóval, ha egy pontnak nagy a szórása, akkor az ahhoz a ponthoz tartozó távolság kisebb súllyal számít bele a szummába. Az így kapott új szummát az alábbi módon írhatjuk fel.

$$S = \sum w_i(\rho_i \cos(\Theta_i - \alpha) - r)^2, \text{ ahol } w_i = 1/\sigma_i^2.$$

Ennek az új szummának keressük a minimum értékét az α r függvényében. A minimum értékét ott veszi fel ez az $S(\alpha, r)$ függvény, ahol a parciális deriváltjai 0. Ezzel a feltétellel fel tudunk írni egy nem lineáris egyenlet rendszert, ahogyan a [6]-ban is teszik.

$$\frac{\partial S}{\partial \alpha} = 0 \qquad \frac{\partial S}{\partial r} = 0$$

Ennek az egyenletrendszernek a megoldása szintén a [6]-ben megtalálható.

$$\alpha = \frac{1}{2} \arctan\left(\frac{\sum w_i \rho_i^2 \sin(2\Theta_i) - \frac{2}{\sum w_j} \sum\sum w_i w_j \rho_i \rho_j \cos(\Theta_i)\sin(\Theta_j)}{\sum w_i \rho_i^2 \cos(2\Theta_i) - \frac{1}{\sum w_j} \sum\sum w_i w_j \rho_i \rho_j \cos(\Theta_i + \Theta_j))}\right.$$

$$r = \frac{\sum w_i \rho_i \cos(\Theta_i - \alpha)}{\sum w_i}$$

Az α r paramétereket már ismerjük, de ezen kívül még ismernünk kell ezeknek a paramétereknek a bizonytalanságát, hogy a már elözőkben említett SLAM fel tudja használni a kiszámolt vonal paramétereit. A [6] -ben megadják, hogy hogyan is tudjuk megkapni egy vonal bizonytalanságát. Mivel itt már két változó bizonytalanságáról beszélünk, amik egymástól nem függetlenek, azért a kiszámolt vonal bizonytalanságát egy kovariancia mátrixszal adjuk meg az alábbi módon.

$$C_{AR} = \begin{bmatrix} \sigma_A^2 & \sigma_{AR} \\ \sigma_{AR} & \sigma_R^2 \end{bmatrix}$$

Ahol a σ_A^2 és a σ_R^2 az α, illetve az r paraméterek szórásnégyzetei a σ_{AR} pedig a köztük fent álló kovariancia. A C_{AR} mátrixot az alábbi módon tudjuk kiszámolni.

$$C_{AR} = F_{PQ} C_X F_{PQ}^T$$

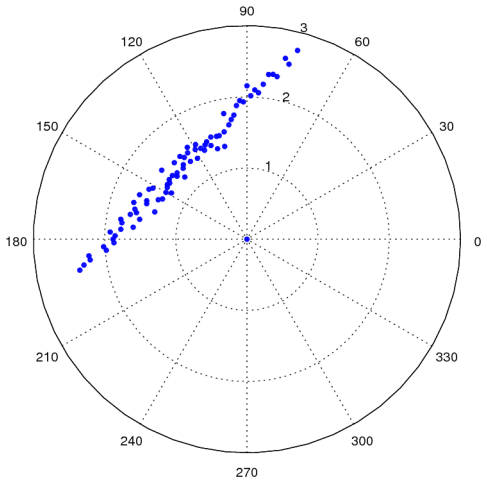
Ahol a C_X mátrix a bemeneti kovariancia, vagyis a pontok P_i és Q_i paraméterek szórásnégyzeteiből képzett diagonális mátrix.

$$C_X = \begin{bmatrix} diag(\sigma_{\rho_i}^2) & 0 \\ 0 & diag(\sigma_{\Theta_i}^2) \end{bmatrix}$$

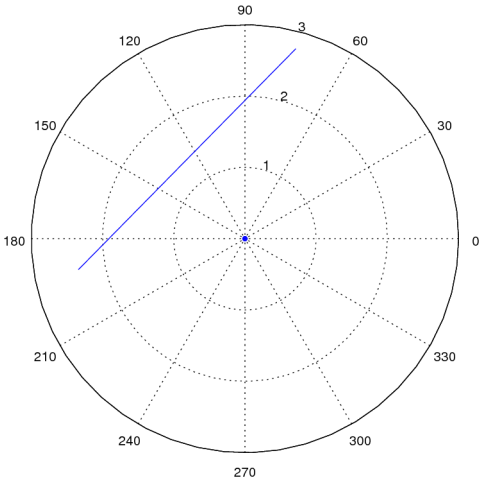
Az F_{PQ} mátrix az α r paraméterek kiszámolásához használt függvények Jacobi mátrix a felhasznált pontokra vonatkoztatva.

$$F_{PQ} = \begin{bmatrix} \frac{\partial \alpha}{\partial P_1} & \frac{\partial \alpha}{\partial P_2} & \cdots & \frac{\partial \alpha}{\partial P_n} & \frac{\partial \alpha}{\partial Q_1} & \frac{\partial \alpha}{\partial Q_2} & \cdots & \frac{\partial \alpha}{\partial Q_n} \\ \frac{\partial r}{\partial P_1} & \frac{\partial r}{\partial P_2} & \cdots & \frac{\partial r}{\partial P_n} & \frac{\partial r}{\partial Q_1} & \frac{\partial r}{\partial Q_2} & \cdots & \frac{\partial r}{\partial Q_n} \end{bmatrix}$$

A fent látható Jacobi mátrix értékeit az implementációban numerikus módon határozzunk meg.



4.2. ábra. *Zajjal terhelt mérésvonal.*



4.3. ábra. *Kiszámolt vonal.*

Az eredmény a 4.3-es ábrán látható, amihez a 4.2-es ábrán található pontokat használtuk fel. A vonal eredeti paraméterei $\alpha = 135°$, illetve $r = 1.2388m$. A zajt egy 0.1m-eres szórású Gauss-zajjal szimuláltuk, ami a pontok távolság paramétereihez adódott hozzá. A kiszámolt paraméterek $\alpha = 135.44°$-nak és $r = 1.26087m$-nek adódtak. Az α paraméternél elkövetett relatív hiba 0.4% az r paraméternél pedig 1.8%. A kiszámolt paraméterekhez az alábbi kovariancia mátrix adódott.

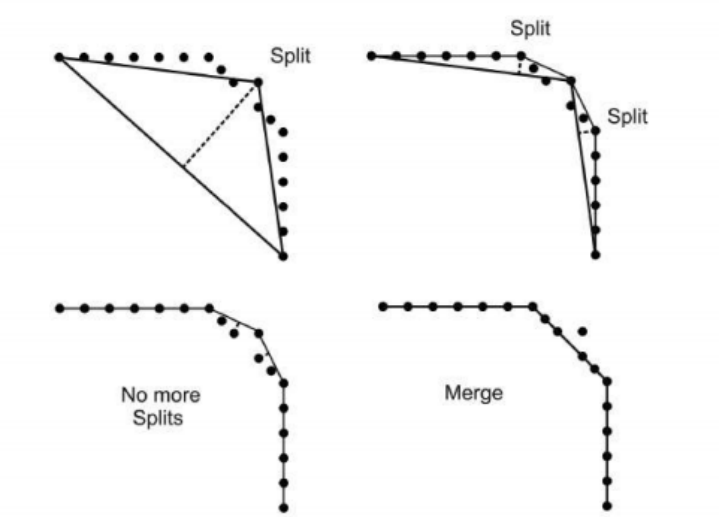
$$C_{AR} = \begin{bmatrix} 5.45293e^{-05} & -1.07108e^{-05} \\ -1.07108e^{-05} & 9.35598e^{-05} \end{bmatrix}$$

Vagyis az α szórása $\sigma_\alpha = 0.4231°$ és az r szórása pedig $\sigma_r = 0.0097m$. Az α számolásakor elkövetett abszolút hiba $0.44°$ ami bőven benne van a $3\sigma_\alpha$ intervallumban. Az r paramétere esetén az elkövetett abszolút hiba $0.02207m$ a $3\sigma_r$ intervallum pedig $0.0291m$, vagyis még az r paraméter esetén is benne vagyunk a 3σ-más intervallumban.

Most már képesek vagyunk egy vonalhoz tartozó pontokból kiszámolni a legoptimálisabb vonalparamétert. Ez azonban feltételezi, hogy az össze pont amit mértünk egy vonalhoz tartozik. Természetesen ez a feltételezés általános esetben nem helytálló. A vonalak kinyeréséhez egy split-and-merge nevű algoritmust használunk, amit a [6]-ben is ajánlanak.

4.2. Módosított Split-and-Merge algoritmus

A split-and-merge az egyik legnépszerűbb algoritmusnak mondható a vonal illesztő algoritmusok közül. Működését egy egyszerű példán könnyen be lehet mutatni.

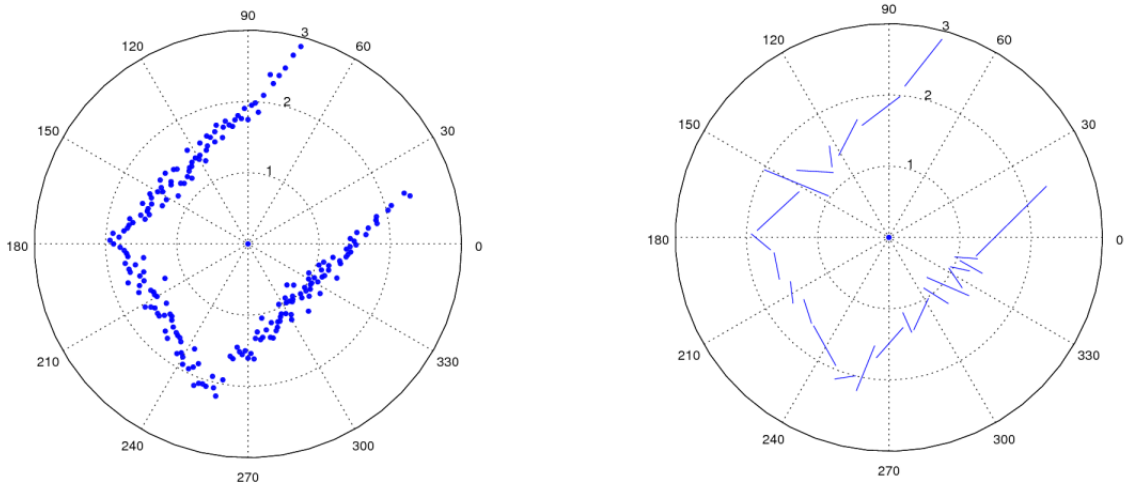


4.4. ábra. *Split-and-Merge [6]*

A 4.4-es ábra jól szemlélteti mit is csinál a split-and-merge. Először a pontokra vonalat illeszt, majd megnézi, hogy az illesztett vonaltól melyik pont van a legtávolabb és milyen távolságra. Ha ez a távolság nagyobb mint egy bizonyos küszöbérték, akkor a legtávolabbi pont mentén ketté választom a ponthalmazomat. Ezeken az új ponthalmazokon megint elvégzem az előző lépéseket. Ezt a ketté választást addig folytatom, amíg minden így képzett ponthalmaz minden eleme kisebb távolságra van a hozzája tartozó vonaltól, mint az adott küszöbérték.

Ez az eljárás nagyon hatékonyan megtalálja a vonalakat, és a lefutási ideje $N \log(N)$-es. A mi esetünkben azonban az eredeti split-and-merge algoritmus nem találja meg megfelelően a vonalakat, hiszen a mi ponthalmazaink erősen zajosak. Ez a fajta zaj megnehezíti a küszöbérték megválasztását. Ha a küszöbérték egy konstans, akkor a split-and-merge a 4.5-ös ábrán látható eredmény adja vissza. Ennek a küszöbértéknek a megfelelő megadására az alábbi ötletet találtuk ki.

Az előző fejezetben definiáltunk egy $S = \sum d_i$ értéket. Ha ennek a szummának megtudnánk mondani a várható értékét és a szórását, akkor definiálni tudnánk egy megfelelő újfajta küszöbértéket a split-and-merge algoritmushoz. Ez az újfajta küszöbérték nem az egyes pontok vonaltól való távolságát venné figyelemben, hanem a már definiált S értékét. A küszöbérték pedig erre az S értékre vonatkozna. Elsőnek vizsgáljuk meg, hogy a d_i értékek hogyan jönnek ki. A pontokat a 4.1-es ábrán látható módon adjuk meg. A Gauss zaj az x_i pont ρ_i távolságparaméteréhez adódik hozzá.



4.5. ábra. *Eredeti Split-and-Merge*

A ρ_i értéke tehát két dologból adódik össze. Az egyik a vonaltól való pontos távolság r_{li} a másik pedig a Gauss zajból adódó távolság r_{ei}. Szóval a $\rho_i = r_{li} + r_{ei}$. Az előző fejezetben már a d_i értékét megadtuk az alábbi módon.

$$\rho_i \cos(\Theta_i - \alpha) - r = d_i$$

Helyettesítsük be a ρ_i helyére a $r_{li}+r_{ei}$ értéket. Ekkor megkapjuk a következő egyenletet.

$$(r_{li} + r_{ei}) \cos(\Theta_i - \alpha) - r = d_i$$

$$r_{li} \cos(\Theta_i - \alpha) + r_{ei} \cos(\Theta_i - \alpha) - r = d_i$$

Észrevehetjük, hogy az $r_{li} \cos(\Theta_i - \alpha)$ értéke az tulajdonképpen az r. Miután kiejtettük ezeket az értékeket ez marad.

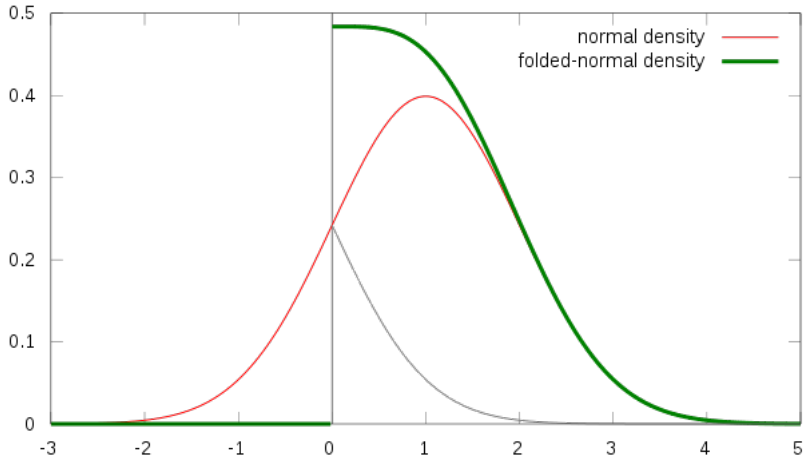
$$r_{ei} \cos(\Theta_i - \alpha) = d_i, \text{ ahol az } r_{ei} \text{ a zajból adódó távolság.}$$

Nézzük meg, hogy az r_{ei} milyen eloszlást követ. A zaja ami létrehozza az r_{ei} értékét egy normál eloszlás, aminek a szórása σ_{ρ_i}. Az r_{ei} csak pozitív értéket vehet fel, hiszen az értéke egyfajta távolságot reprezentál, tehát az r_ei eloszlása jól közelítéssel egy normál eloszlásnak az abszolút értéke. Ezt a fajta eloszlást folded normal eloszlásnak hívják. A 4.6-os ábrán láthatjuk ezt a fajta eloszlást összehasonlítva egy normál eloszlással.

Most már tudjuk, hogy az r_{ei} milyen eloszlást követ, vagyis meg tudjuk határozni r_{ei} várható értékét. $E(r_{ei})$ értéke a [8] alapján a következő.

$$E(r_{ei}) = \sigma_{\rho_i} \sqrt{\frac{2}{\pi}}$$

$E(r_{ei})$ ismeretében fejezzük ki a várható értékét a $\sum d_i$-nek.



4.6. ábra. *Folded normal eloszlás*

$E(\sum d_i) = E(\sum r_{ei}\cos(\Theta_i - \alpha))$, mivel a pontok mérését független eseménynek tekintem, így az összeg várható értéke egyenlő a várható értékek összegével.

$E(\sum d_i) = \sum E(r_{ei}\cos(\Theta_i - \alpha))$ Itt a $\cos(\Theta_i - \alpha)$ kiemelhetem a várható értékből, mivel ez egy konstans érték, tehát végül $E(\sum d_i)$ értéke a következőnek adódik.

$$E(\sum d_i) = \sum \cos(\Theta_i - \alpha)E(r_{ei}), \text{ ahol } E(r_{ei}) = \sigma_{\rho_i}\sqrt{\tfrac{2}{\pi}}.$$

Tehát a végeredmény.

$$E(\sum d_i) = \sum \cos(\Theta_i - \alpha)\sigma_{\rho_i}\sqrt{\tfrac{2}{\pi}}.$$

A $E(\sum d_i)$ mellé még meg kell mondani a $\sum d_i$ szórását is. A [8]-re hivatkozva r_{ei} szórásnégyzete σ_{ρ_i}-vel kifejezve a következő.

$$Var(r_{ei}) = \sigma^2_{\rho_i}(1 - \tfrac{2}{\pi})$$

Most írjuk fel a $\sum d_i$ szórásnégyzetét.

$Var(\sum d_i) = \sum Var(d_i)$, mivel a független események szórásnégyzete összeadható.

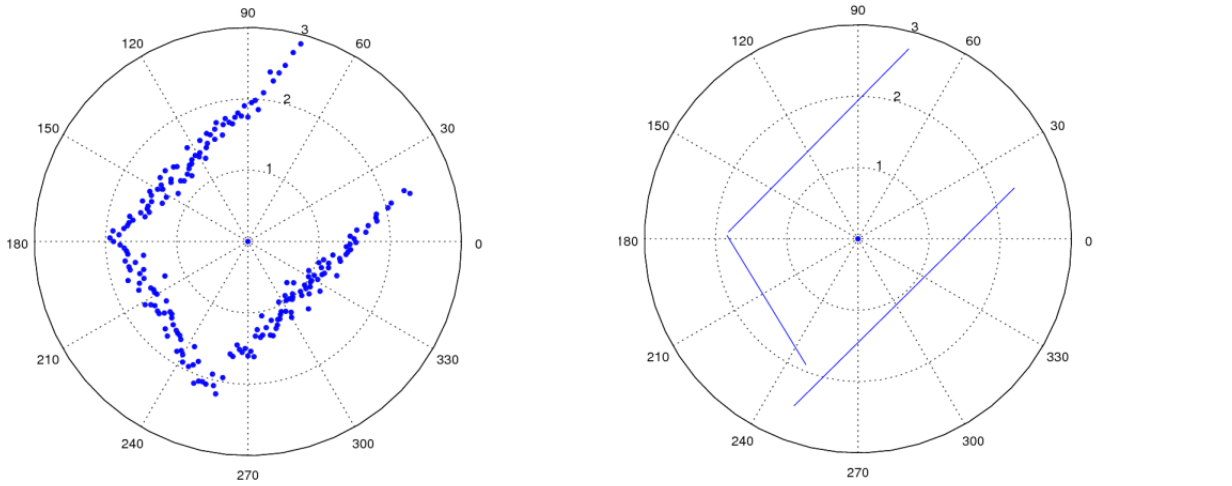
A szórásnégyzet alábbi tulajdonságát kihasználva,

$$Var(aX) = a^2 Var(X)$$

felírhatjuk $\sum Var(d_i)$ értékét $Var(r_{ei})$-vel kifejezve.

$$\sum Var(d_i) = \sum cos(\Theta_i - \alpha)^2\sigma^2_{\rho_i}(1 - \tfrac{2}{\pi})$$

Most már tudunk egy új jobb feltételt mondani, miszerint egy ponthalmazról akkor feltételezem, hogy egy vonalhoz tartoznak, ha teljesül a ponthalmazra az alábbi feltétel.



4.7. ábra. *Módosított split-and-merge algoritmus*

$$\sum d_i < E(\sum d_i) + \sqrt{Var(\sum d_i)}$$

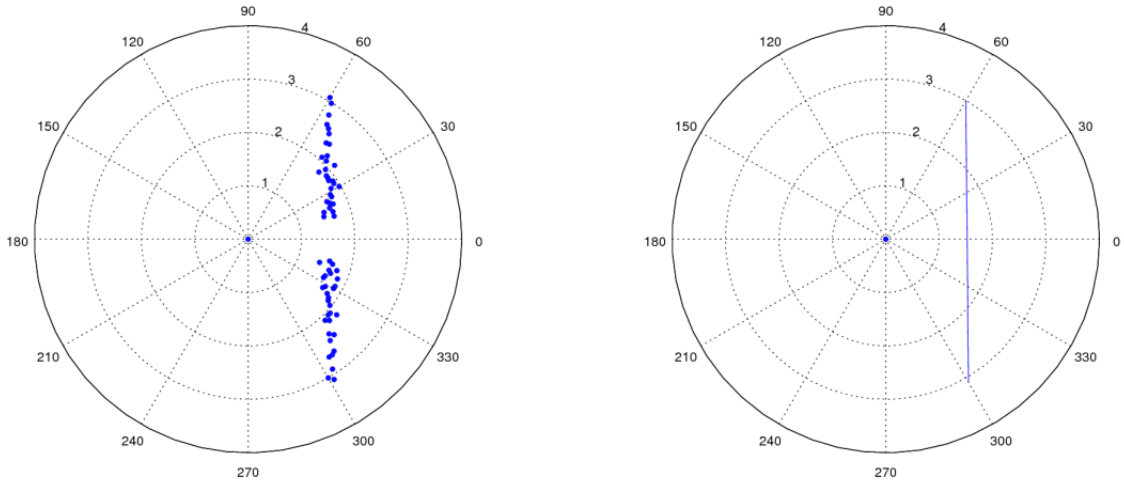
Ezzel az új módosítással a split-and-merge algoritmus az alább eredményt adta.

Mint ahogyan a 4.7-es ábra is mutatja a módosított split-and-merge algoritmus helyesen megtalálta a vonalakat, de ehhez tudnunk kellett a pontok szórását. A σ_{ρ_i} értékeket méréssel meg lehet határozni.

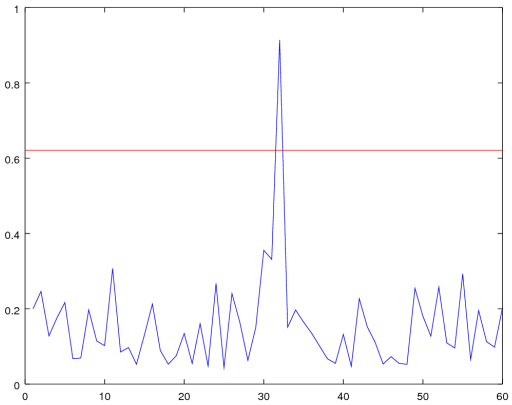
Látszólag most már megfelelően működik a vonal illesztő algoritmusunk, de még mindig van olyan ponthalmaz amire nem megfelelően fut le. Erre példa a 4.8-as ábrán lehet látni. Az ábrán egy tipikus eset látható, amikor a fal vonala megszakad egy ajtó miatt például. A split-and-merge algoritmus nem képes érzékelni a szakadásokat. Ezt a fajta hibát viszont nem engedhetjük meg magunknak, hiszen a robotot alapvetően benti használatra tervezzük, ahol elég gyakori ez a fajta elrendezés.

A fent említett problémára az a megoldás, hogy a ponthalmazokon egyfajta előszegmentálást végzünk. Érezzük, hogy ezt az előszegmentálást a ponthalmaz szomszédos pontjai közötti távolság alapján kéne végeznünk. A fenti ponthalmaz szomszédos pontjai közötti távolságot a 4.9-es ábra szemlélteti. Az ábrán jól láthatóan kitűnik a szakadást jelző, kiemelkedően nagy távolság érték.

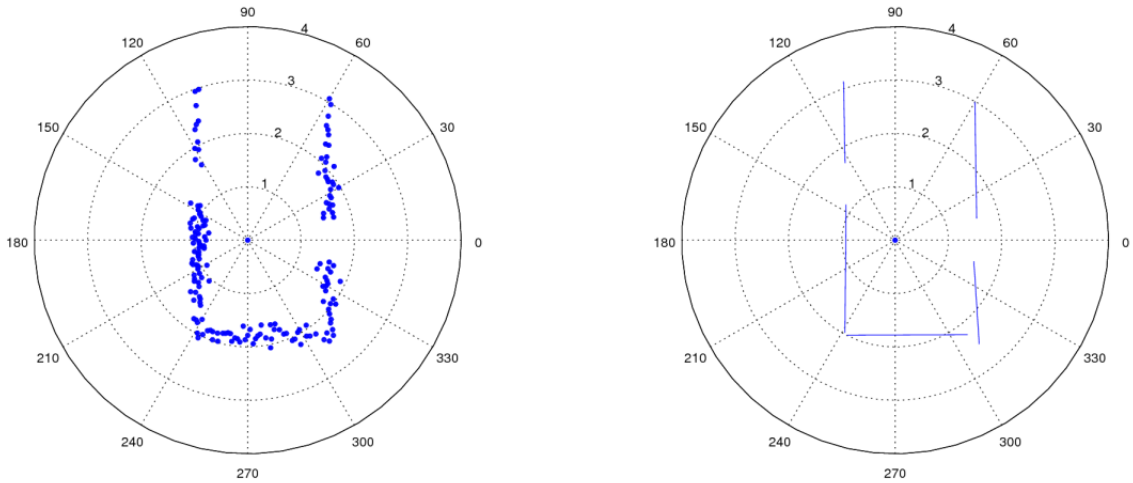
A következő feltétel alapján döntöm el, hogy mikortól mondom egy távolság értékre, hogy azt szakadás okozta. Egy d_i távolság szakadást jelez, ha $d_i > \nu\frac{\sum_1^n d_i}{n}$, vagyis d_i szakadást jelez, ha nagyobb mint a távolságok átlaga szorozva egy ν konstanssal, ahol ν értéke a szimulációk alapján $4-5$ közötti érték. A 4.9-es ábrán a ν-vel módosított átlagot a piros vonal jelzi. Amelyik pontok felette vannak a módosított átlagnak, azon pontok mentén végzem az előszegmentálást. Ennek az előszegmentálásnak az eredményét a 4.10-es ábrán látható.



4.8. ábra. *Az ajtó probléma*



4.9. ábra. *Szomszédos pontok távolsága*



4.10. ábra. *Előszegmentált ponthalmazon végzett vonalillesztés*

5. fejezet

Útvonaltervezés

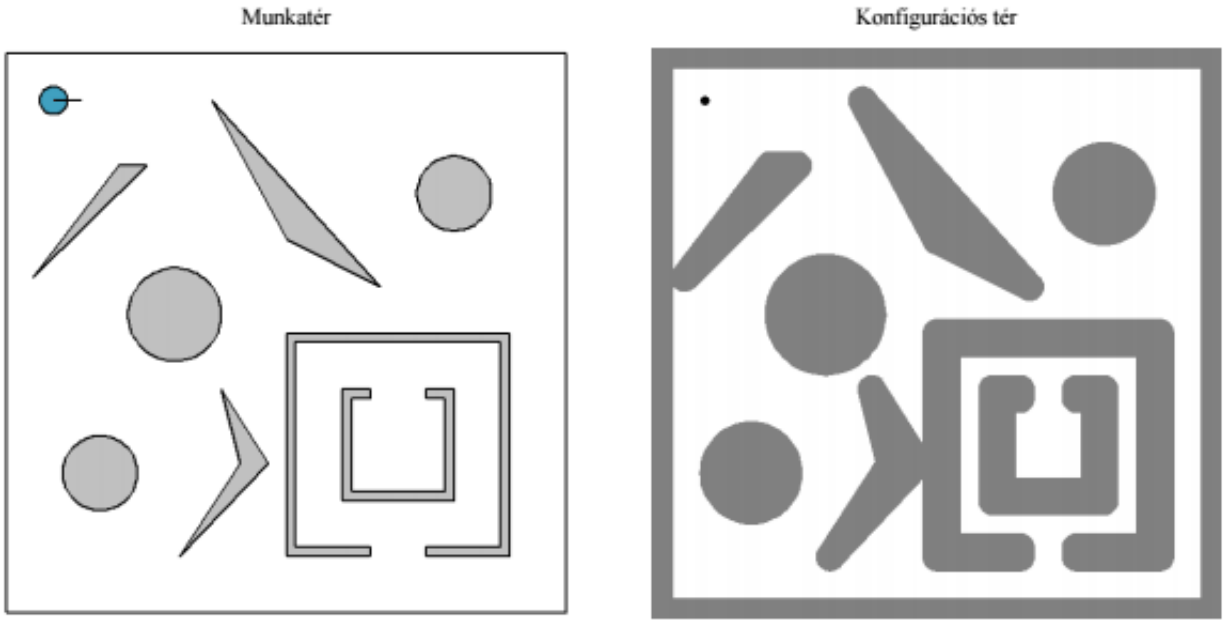
A robotunk már képes meghatározni a helyzetét, és képes a környezetét leképezni egy vonalakból álló térképre, ahol a vonalak az akadályokat reprezentálják. Azonban ez még nem teszi a robotunkat egy önjáró robottá. Ahhoz, hogy egy lépéssel közelebb jussunk a végcélunkhoz, vagyis hogy a robotunk önállóan tudjon működni, implementálnunk kell egy útvonaltervező algoritmust, aminek a segítségével a robot képes elnavigálni a megadott célig.

5.1. A konfigurációs tér

Tételezzük fel, hogy a térképet már ismerem, amiben el kell navigálnom a robotot a célhoz. Ez a térkép vonalakkal írja le a robot környezetét. Ha ezt a térképet használnánk fel az útvonaltervezéshez, akkor érezhetjük, hogy a robotunk elég nagy valószínűséggel neki fog menni az akadályoknak. Ezt a problémát oldja meg a konfigurációs tér használata, ahogyan azt a [2]-ban is olvashatjuk.

Ahhoz, hogy ezt a konfigurációs teret megtudjuk konstruálni vizsgáljuk meg a robotunk állapotait. Ennél a megközelítésnél feltételezzük, hogy a robot mozgását csak a kinematika határozza meg, vagyis, hogy a robot állapotai statikus állapotoknak tekinthetjük. Ez azt jeleni, hogy a robotot bármikor megtudjuk állítani a pályáján, függetlenül attól, hogy éppen milyen gyorsan ment.

A robot állapotát a következő vektor adja meg $\xi = (x, y, \Theta)$. A konfigurációs tér azt a célt szolgálja, hogy a robotot ne engedje olyan állapotba, aminek az eredménye ütközés lenne. Természetesen, hogy ezt a konfigurációs teret létrehozzuk ismernünk kell a robot geometriai paramétereit. A robotunk alakját egy téglatesttel írhatjuk le, de ekkor a konfigurációs terünket egy 3 dimenziós tér írná le. Most képzeljük el, hogy a robotunk henger alakú. Ekkor a robot állapotai közül a Θ elhagyható a konfigurációs tér szempontjából, vagyis ha ezzel a egyszerűsítéssel élünk, akkor a konfigurációs terünk egy 2 dimenziós térré redukálódik. A konfigurációs teret most már létre tudjuk hozni a robot köré írható kör sugarának ismeretével. Az így kapható konfigurációs teret az 5.1-es ábra szemlélteti.



5.1. ábra. *Kör alakú robot konfigurációs tere [2]*

Láthatjuk, hogy ebben a térben a kör alakú robotunk egy pontra képződik le, hiszen ebben a konfigurációs térben a robotot az állapot vektora írja le, míg az akadályok kiszélesednek a kör sugarának megfelelően. Ha ebben az új térben találok egy útvonalat, akkor arról biztosan kijelenthetjük, hogy az útvonal bejárásakor nem fogunk egy akadályba se ütközni.

5.2. Rapidly Exploring Random Trees algoritmus

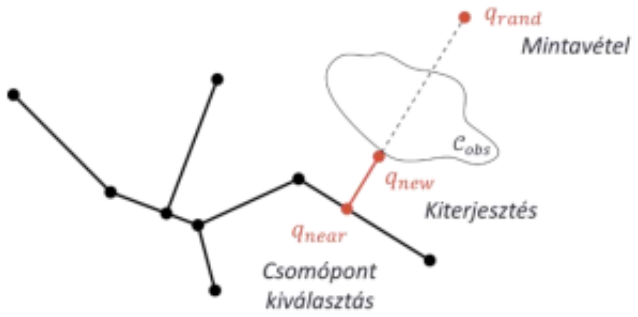
A Rapidly Exploring Random Trees algoritmus egy mintavételezésen alapuló útvonaltervező algoritmus, amiről a [2]-ben olvashatunk. Ahogyan a neve is sugallja ez az algoritmus egy "fával" benövi a szabad teret, és ebben a fában keres majd útvonalat. A fa a robot pozíciójából kezdi el a növekedést. Ha a célpontom a fa egyik levele, akkor az útvonalat egyszerűen megtudom mondani. A fa struktúra miatt az útvonal megismeréséhez elegendő, ha a levéltől kiindulva végig megyek a szülőkön, amíg el nem érek a gyökérhez, vagyis a robot pozíciójához. A fa struktúra ezen tulajdonsága miatt, nem kell költséges útvonalkereső algoritmust használni.

A Rapidly Exploring Random Trees (RRT) algoritmusnak azonban van egy hátránya. Mivel egyenletes eloszlású véletlen mintavételezésen alapszik, a lefutása nem determinisztikus. Szóval ha egy olyan célpontot adok meg neki amihez nem létezik útvonal, akkor végtelen ideig futna. Ezt a problémát egy futási idő korlátozással tudjuk kiküszöbölni. Ezen kívül még egy problémát kell megoldanunk. Az RRT-vel növesztett fában csak akkor tudok eljutni a célomhoz, ha a cél a fa egyik levele. Erre az ad megoldást, ha a célpontot beleteszem a mintasorozatba. Ezt úgy tudom elérni, hogy a mintasorozatot az alábbi módon konstruálom. A célpontot p valószínűséggel választom bele a mintasorozatba, a

véletlen mintákat pedig $1 - p$ valószínűséggel rakom bele. Ennek a p megválasztása igen kritikus. Ha p értéke nagy, akkor az RRT által felhasznált mintasorozatban kevés véletlen minta lesz, ennek következtében az RRT által növesztett fa nem fogja elég gyorsan benőni a teret. Természetesen az se lesz megfelelő, ha a p értékét túl kicsire választjuk, ekkor az elérni kívánt cél kevésszer szerepel a mintasorozatban, ami meg azt eredményezi, hogy a fa túlságosan sokáig nő. A szimulációk alapján $p = 10\%$-nál az RRT lefutása megfelelő.

A fent említett problémák megoldása után nézzük meg az RRT lépéseit a [2] alapján.

- **Mintavétel.** Kijelölünk egy mintát a konfigurációs térből, amely a gráf kiterjesztés célja lesz (q_{rand}).
- **Csomópont kiválasztás.** Megkeressük a gráfban q_{rand}-hoz legközelebb eső pontot. Ha a legközelebb eső pont egy élen helyezkedik el, akkor az élet a pontnál kettéosztjuk, és a pontot felvesszük a gráf csomópontjai közé. Az így kiválasztott legközelebbi gráfbeli (q_{near}) pont lesz a gráf kiterjesztés kiindulópontja.
- **Kiterjesztés.** q_{near} és q_{rand} pontokat egy lokális pályával (pl. egyenessel) összekötjük. Amennyiben ez ütközésmentes, a lokális pályát élként, q_{rand} pontot pedig csomópontként hozzáadjuk a gráfhoz. Ha a lokális pálya akadályon halad keresztül, akkor is kiterjesztjük a gráfot, de q_{rand} helyett csak az utolsó ütközésmentes pontig a lokális pálya mentén (q_{new}).

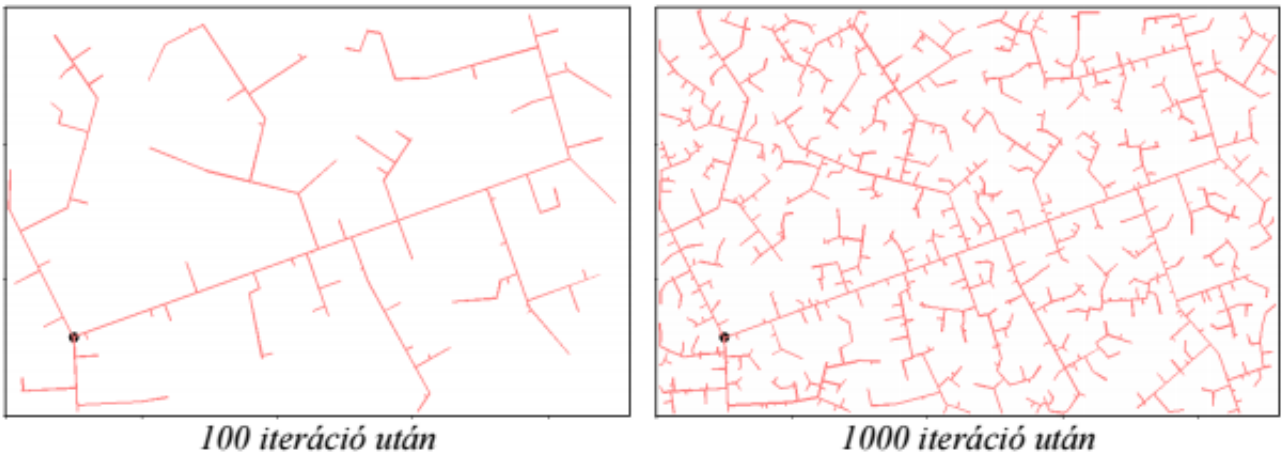


5.2. ábra. *RRT lépései [2]*

Az RRT által legenerált fa az 5.3-mas ábrán látható módon növi be a szabad teret.

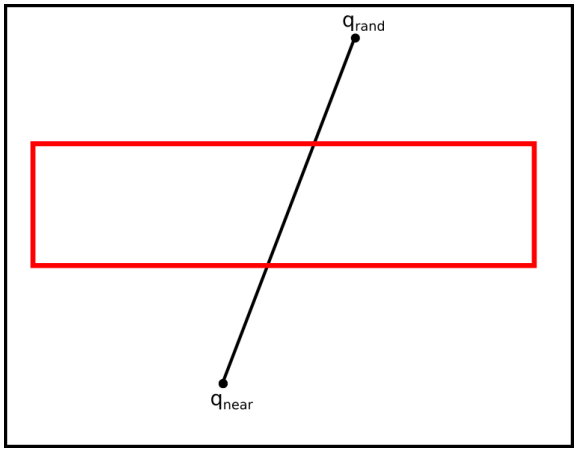
Itt még a tér nem tartalmaz semmilyen akadályt. A mi esetünkben az akadályokat egy vonal reprezentálja, ami a konfigurációs terünkben egy téglalappá fog áttranszformálódni.

Az RRT algoritmus egyik fontos lépése a **Kiterjesztés**. Ebben a lépésben döntjük el, hogy a q_{rand} és a q_{near} között van-e akadály. A mi esetünkben az alábbi módon döntöm el ezt a kérdést.



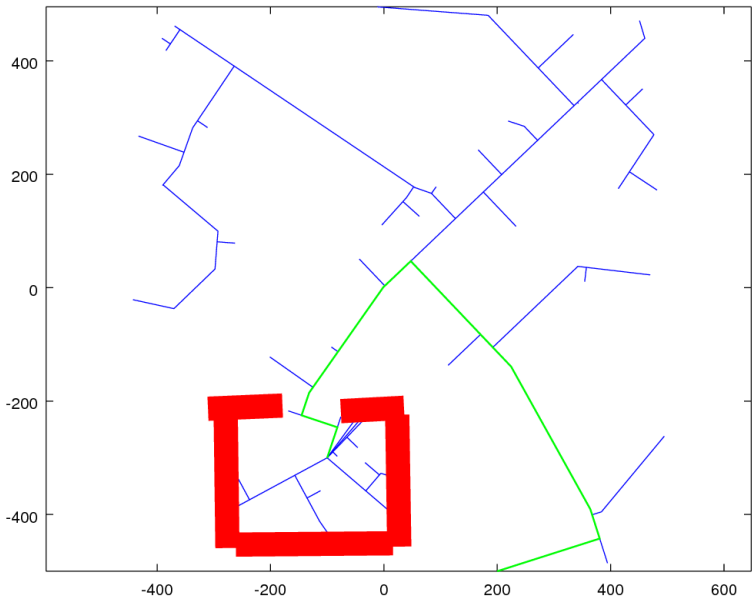
5.3. ábra. *Az RRT szabad tér kitöltő tulajdonságának illusztrációja [2]*

Az 5.4-es ábrán látható egy példa, arra amikor az új q_{rand} és a q_{near} pontok között akadály helyezkedik el. Láthatjuk, hogy ütközés akkor van, ha a q_{rand} és q_{near} által meghatározott egyenes és a téglalap egyik oldalára illesztett egyenes metszéspontja eleme a téglalapnak.



5.4. ábra. *Az akadály szemléltetése a konfigurációs térben.*

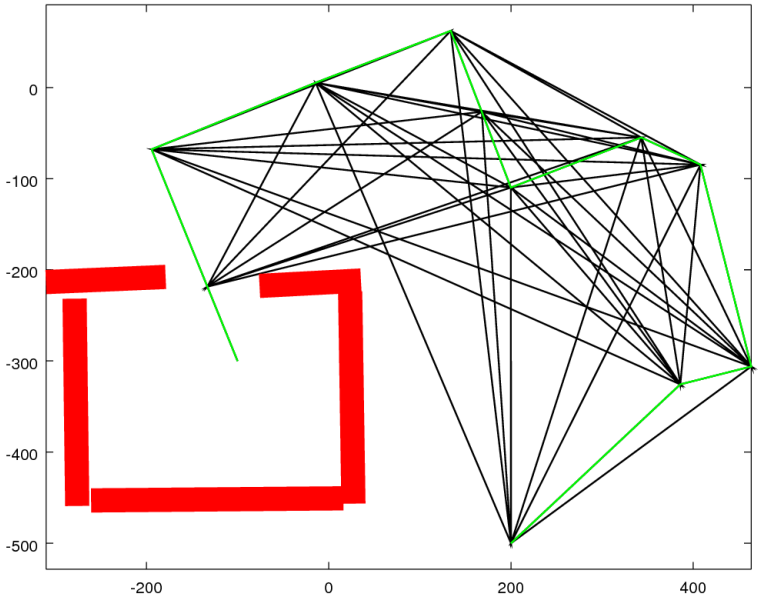
Az akadályokat is tartalmazó konfigurációs térben megadtunk egy célpontot is. A RRT algoritmus az 5.5-ös ábrán látható zölddel jelzett útvonalat adta vissza. Az ábra alapján láthatjuk, hogy az algoritmus talált útvonalat, de érrezük, hogy az algoritmus által adott útvonalat lehet optimalizálni. Erre a megoldást a következő alfejezet adja meg.



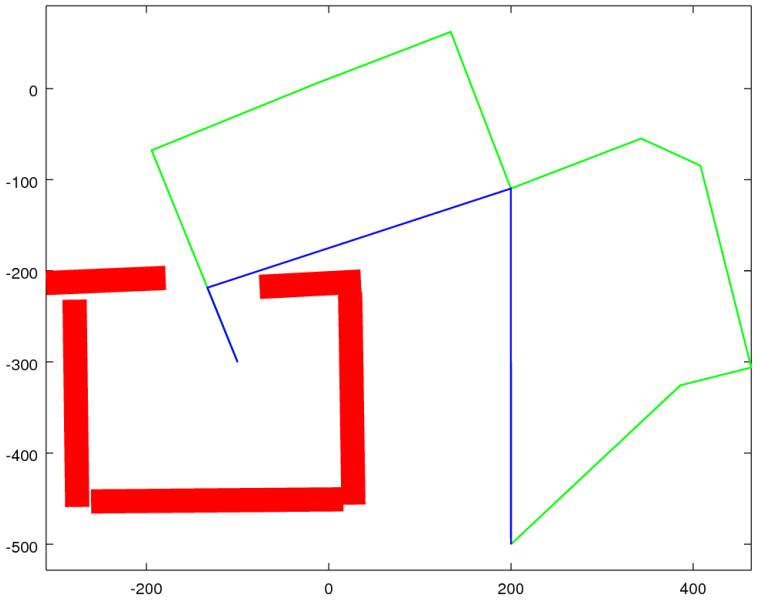
5.5. ábra. *Akadályt tartalmazó tér benövése.*

5.3. Útvonal optimalizálása Dijkstra algoritmussal

Ahogyan az 5.5-ös ábrán is látható az RRT által adott útvonal helyes, de lehet rajta rövidíteni. Például, ha az útvonal két pontját össze tudom kötni úgy, hogy az ne ütközzön akadályba, akkor máris kaptam egy rövidebb utat. Ezen az ötleten elindulva húzzuk be az útvonalon az összes olyan vonalat ami nem ütközik akadályba. Erre példát az 5.6-os ábrán láthatunk. Az így kapott új útvonalakból egy gráfot képzünk. A gráf éleit az élekhez tartó utak hosszával súlyozzuk. Az így kapott gráfban már tudunk legrövidebb útvonalat keresni, mégpedig a Dijkstra algoritmus segítségével. Az eredmény az 5.7-es ábrán látható. Az így kapott új útvonal csakis egyenes szakaszokból áll, vagyis végig tudok menni rajta előre mozgással és forgómozgással, tehát az útvonalból könnyedén le tudom generálni az utasításokat, amiket ha a robot végrehajt eljut a kijelölt célhoz.



5.6. ábra. *Alternatív útvonalak*



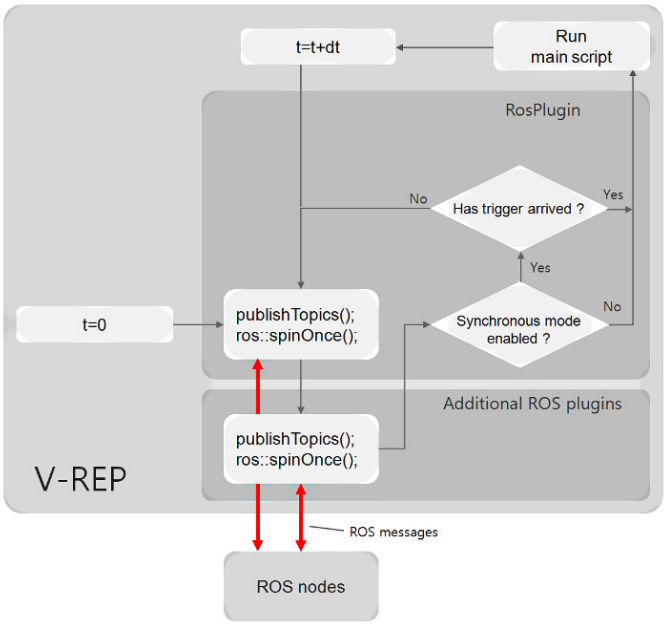
5.7. ábra. *Dijkstra által adott új útvonal és az eredi útvonal*

6. fejezet

Szimuláció

6.1. V-REP

Az algoritmusaink tesztelésére elsősorban a Virtual Robotics Experimentation Platformot [5] (V-REP) használtuk fel. Ebben egy hatkerekű modellt hoztunk létre amelynek kinematikai leírása megegyezik egy lánctalpas robotéval.



6.1. ábra. *ROS folyamatábra*

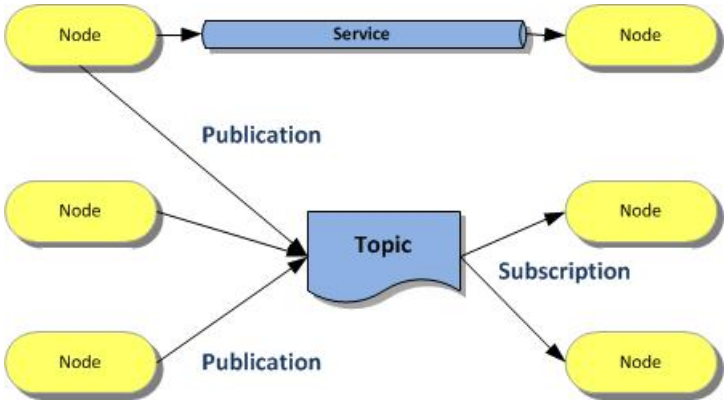
A kezdeti szimulációink során az általunk írt C++ programokat a V-REP szimulációjával párhuzamosan, egy kliens alkalmazásként futtattuk, amik a V-REP remote API plugin-ja által futtatott szerverrel socket kommunikáción keresztül cseréltek információt. Ez egy olyan módszer volt amiben az üzenetek formájára, időzítésére és visszajelzésére sokfajta megközelítés használható volt, de ezt a módszert végül elvetettük egy jobb megoldás javára.

A jobb megoldás ötlete abból ered, hogy a V-REP-nek létezik egy Robot Operating System [1] (ROS) interfésze is (6.1 ábra), azaz a ROS plugin segítségével a szimulációval párhuzamosan futtatható egy beépített V-REP ROS node, amivel a szimulációról szinte bármilyen információ lekérdezhető lesz egyéb ROS node-ok számára. Ez a megoldás egy laza kommunikációs kapcsolatot tud biztosítani a külön node-okon futó algoritmusaink között, ami több szempontból is hasznosabb. Egyrészt így jóval könnyebben bővíthető, szeparálható, és más applikációkba integrálható programokat írhattunk, másrészt pedig egyetlen node összeomlása így nem feltétlenül jelenti a teljes rendszer összeomlását, ami robotikai alkalmazásokban nagy előnyt jelent (például ha egy humanoid robot karja meghibásodik, attól még sétálni ugyanúgy képes lehet). Az implementáció egyetlen kezdeti hátránya az volt, hogy a ROS 2016-os verziójára (Kinetic) még nem volt teljesen felkészítve ez a plugin, így a lefordítása extra munkát igényelt, és ezen kívül pedig jóval több Lua szkript megírására volt szükség, amik a szimuláció közben futnak.

6.2. ROS

A Robot Operating System (ROS) egy rugalmas keretrendszer, ami megkönnyíti a különböző robot szoftverek írását, fejlesztését és tesztelését. A ROS úgynevezett package-ekből épül fel, amiket C++-ban és python-ban is implementálhatunk.

A ROS-nak köszönhetően a különböző feladatokhoz külön-külön package-ket tudunk rendelni. A package-ekhez hozzárendelhetünk úgynevezett node-okat, amik megoldják a hozzájuk tartozó részfeladatot. Ezen node-ok képesek az egymással való kommunikációra, ezáltal megtudnak oldani egy komplex feladatot is. A node-ok két fajta módon képesek kommunikálni egymással.



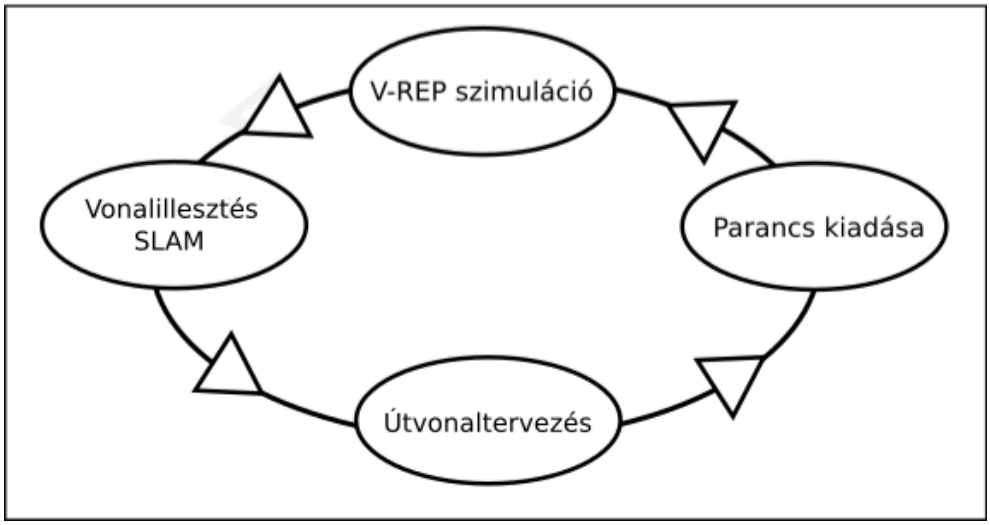
6.2. ábra. *A node-ok kommunikációja.*

A 6.2-es ábra mutatja be a kétféle kommunikációt. Mint láthatjuk az egyik fajta kommunikáció a service. A service kommunikációnál csak két node kommunikál egymással. A másik kommunikációs fajta a topic. Topic kommunikációnál létrehozunk egy topic-ot amire egy node kétfajta módon tud felcsatlakozni. A node vagy publisher-ként csatlakozik fel a topic-ra vagy subscriber-ként. A publisher node-ok adatokat küldenek fel a megfelelő topic-okra, amiről a subscriber-ek értesítést kapnak és letölthetik az újonnan publikált adatokat. Természetesen ahhoz, hogy a fent említett kommunikációk működjenek előre kell definiálni az adatcsomagokat, amiket majd használni fognak a node-ok. A ROS tartalmaz előre definiált message-eket, amiket a node-ok képesek használni a kommunikáláshoz. Ha a ROS által kínált message-ek egyike se lenne megfelelő, akkor saját message-eket is létrehozhatunk.

Ahhoz, hogy a különböző node-ok futni tudjanak, kell egy központi program. Ennek a programnak a feladatát az úgynevezett roscore végzi el. A roscore figyeli a node-ok futását, és irányítja a köztük zajló kommunikációt, továbbá ez a roscore lehetővé teszi, hogy a node-ok egy hálózaton lévő, különböző számítógépeken fussanak. Ebben az esetben a számítógépeken futó roscore-oknak meg kell mondani, hogy melyik roscore végzi majd a teljes felügyeletet.

6.3. Teljes szimulációs ciklus

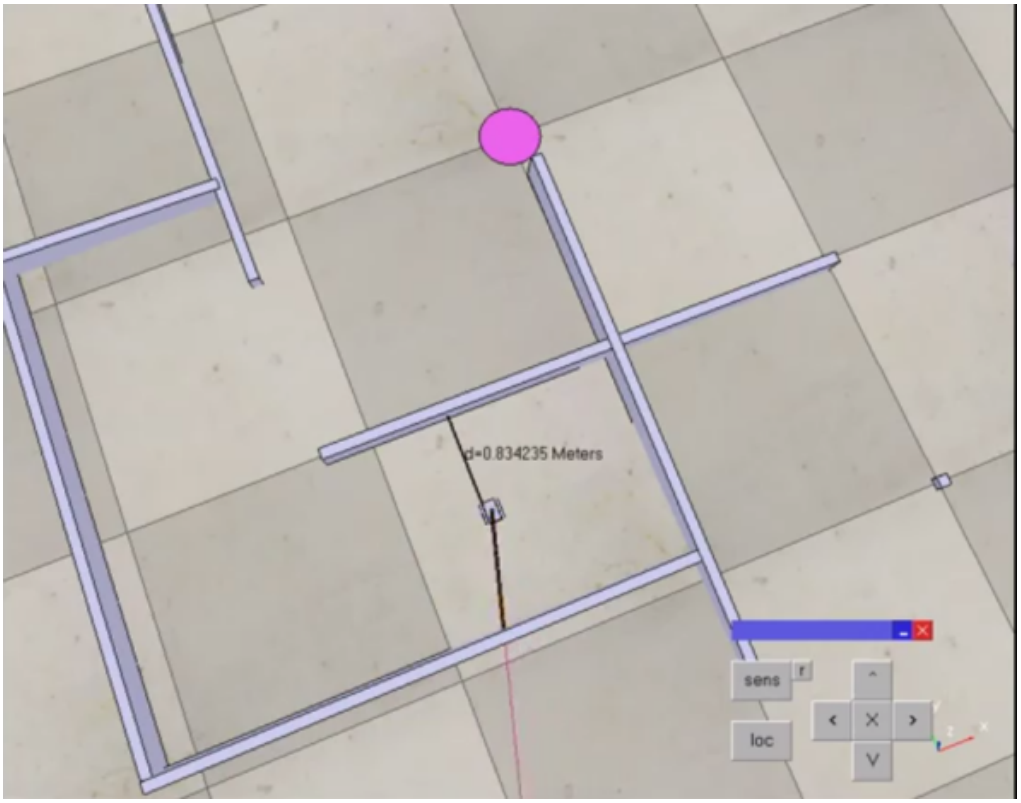
Elsőnek vizsgáljuk meg, hogy milyen node-okra lesz szükségünk a teljes szimulációhoz. Természetesen kell egy node, ami a szimulált környezetben mozgó robotról szolgáltat információkat. Ezek az információk a szenzorokból kinyert pontfelhő és a kerekek elfordulása. Kell egy node, ami a lokalizációt végzi. Ezen felül még szükségünk lesz egy node-ra ami a térképet építi fel és megtervezi az útvonalat, továbbá még egy node ami a parancsokat adja ki a szimulált robotnak. Összesen tehát négy node-ra lesz szükségünk. Ezek a node-ok egy ciklusban fognak működni. Ezt a ciklus a 6.3-mas ábra szemlélteti.



6.3. ábra. *Szimulációs ciklus.*

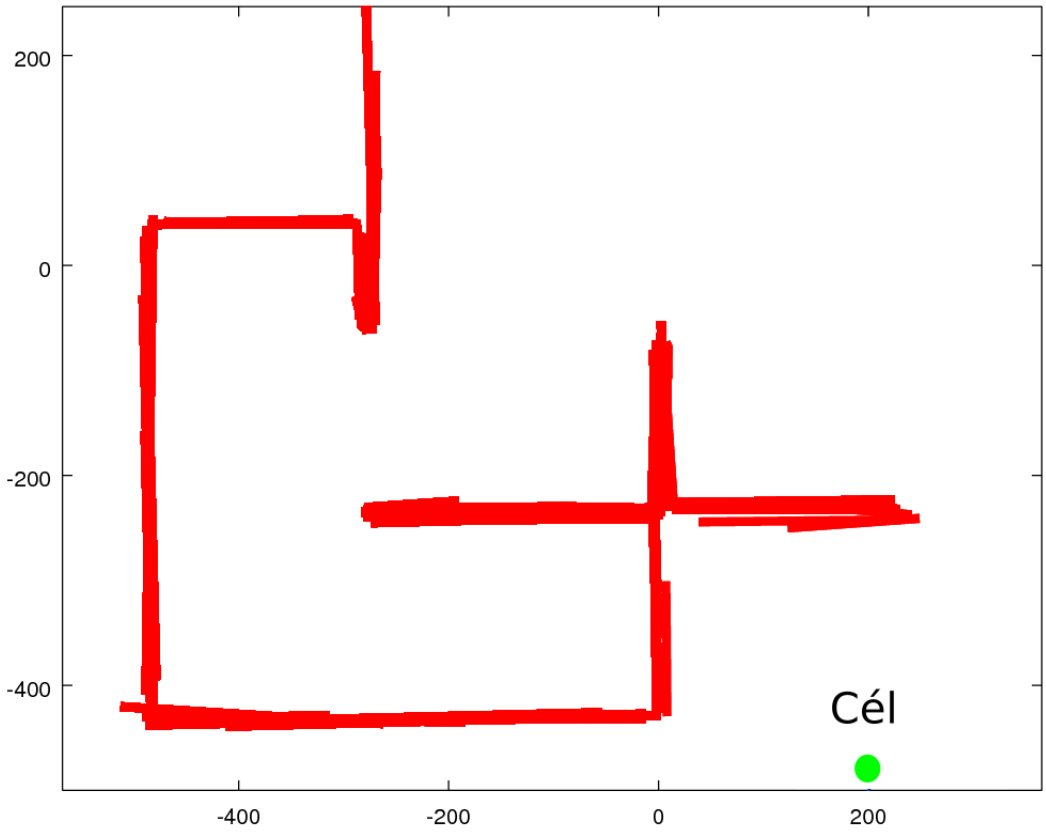
Most vizsgáljuk meg egy szimulációs ciklust. Ez első ciklust a parancs node indítja el egy init paranccsal. Ennek hatására a szimulált robot körbeforgatja a szenzorait. A szenzorokból kinyert pontfelhőt megkapja a lokalizációs node. Mivel még csak az első ciklusban vagyunk, ezért a helyzetünket ismertnek tekintjük. A kezdéskor a pozíciónk a $0, 0$ pontban van és az orientációnk is 0. Annak ellenére, hogy a helyzetünk ismert, a pontfelhőből még ki kell nyerni a vonalak, hiszen mind a lokalizáció, mind a térkép alkotás is a vonalakat használja fel. Miután a lokalizációs node kinyerte a vonalakat a pontfelhőből, a térképalkotással és az útvonaltervezéssel foglalkozó node-nak küldi az adatokat, vagyis a kinyert vonalakat és a pozíciót. A térképalkotásnál természetesen nem az egész vonalat használjuk fel, hanem csak a vonal azon részét, ami az akadályt reprezentálja. Ezzel a kezdetleges térképpel már tudunk útvonalat tervezni a célunkhoz. Ebben a szimulációban egy távoli pontot kell elérni a robotnak. A robot akkor áll meg, ha a célt elérte, vagy ha a cél nem elérhető. Mivel még csak a környezet egy kis részét térképeztük fel, azért a kapott útvonalnak csak azon része megbízható, ami az eddig felépített térképen belül van, tehát a kiszámolt útvonalat cson-

kolni kell. Ugye egy útvonalhoz tartozó parancsok sorozata forgásból és előre menetelből áll. Ebből a parancssorozatból csak az első forgást és az első előremenetelt használjuk fel, úgy hogy az előre menetelnél korlátozzuk a megtehető távolságot. A megtehető távolság maximumát a szenzor érzékelési távolsága adja meg. Ez a korlátozás biztosítja, hogy a robot ne menjen ki az eddig ismert térképről. Most már megvannak azok a parancsok amiket a robotnak ki szeretnénk adni, ezért a parancsok kiadásárt felelős node-nak elküldjük a parancsokat, hogy azokat továbbítsa a szimulált robotnak. Ezzel le is ment az első szimulációs ciklus. Ezt követően amikor a robot véghez vitte a megkapott parancsot kezdődik a ciklus előröl. A szimulációhoz épített pálya a 6.4-es ábrán látható.



6.4. ábra. *Szimulált környezet.*

A szimuláció közben a 6.5-ös ábrán látható térképet építette fel a robot. A tesztelés során készülő videók az alábbi linken érhetőek el `https://www.youtube.com/channel/UC64p_gurJoQFDQoTFPCXZCg`.



6.5. ábra. *A felépített térkép.*

6.4. További tervek

Most még az autonóm rendszerünk képességei abban merülnek ki, hogy eljut egy előre definiált célhoz, és útközben térképet készít a környezetéről. Ez az eljárás természetesen nem garantálja, hogy a robot feltérképezi a teljes környezetét, ezért a további terveink között szerepel egy feltérképező algoritmus implementálása. Ez az algoritmus tulajdonképpen folyamatosan adni fogja az új célpontokat az eddig megvalósított rendszernek, amíg van a robot számára ismeretlen területek, ahova el is tud jutni.

A térképépítés az eddig megvalósított rendszerben igen kezdetleges. Nem használ semmilyen optimalizálást. A mostani térképépítés folyamatosan egészíti ki az új vonalszakaszokkal a térképet. Ez a fajta térképépítés nem veszi figyelembe, hogy egy adott akadályt szerepel az eddigi térképen, ezért a további tervek között szerepel a térképépítés optimalizálása is.

Miután a hiányzó szoftvereket elkészítettük és leteszteltük a szimulációs környezetben, a valós robotra való átállás lesz a következő lépés. A sikeres átállás után a valós robot tesztelésével és fejlesztésével folytatnánk, amíg el nem érjük, hogy a rendszerünk teljesen autonóm módon képes legyen feltérképezni a környezetét és ebben a környezetben utasításokat tudjon végrehajtani.

Köszönetnyilvánítás

Köszönjük a Simonyi Károly Szakkollégium Robotika körének, hogy biztosította nekünk a vizsgált robotot, és Kiss Domokosnak az elgondolkodtató és izgalmas konzultációit és tanácsait.

Irodalomjegyzék

[1] The robot operating system (ros). http://www.ros.org/about-ros/.

[2] Dr. Tevesz Gábor. *Robotirányítás rendszertechnikája.*

[3] Lantos-Kiss-Harmati. *Szabályozástechnika gyakorlatok.*

[4] Tóth Miklós. Lokalizációs és térképezési algoritmusok vizsgálata valós roboton (diplomaterv). https://diplomaterv.vik.bme.hu/hu/Theses/Lokalizacios-es-terkepezesi-algoritmusok1.

[5] Coppelia Robotics. Virtual robot experimentation platform. http://www.coppeliarobotics.com/helpFiles/index.html.

[6] Illah Reza Nourbakhsh Roland Siegwart and Davide Scaramuzza. *Introduction to Autonomous Mobile Robots, Second Edition.* The MIT Press, 2011.

[7] Joan Solŕ. Simulataneous localization and mapping with the extended kalman filter, 2014.

[8] Wikipedia. Folded normal distribution. https://en.wikipedia.org/wiki/Folded_normal_distribution.

Függelék

F.1. OpenCV kódrészlet a kovariancia mátrix vizualizálásához

F.1.1. lista. *Az OpenCV-vel készített teszt forráskódja*

```
using namespace std;

vector <vector <float> > data;
ifstream infile( "covar.txt" );

while (infile){
 string s;
 if (!getline( infile, s )) break;

 istringstream ss( s );
 vector <float> record;

 while (ss){
  string s;
  if (!getline( ss, s, ',' )) break;
  QString tmp = QString::fromStdString(s);
  record.push_back( tmp.toFloat() );
 }
 data.push_back( record );
}
using namespace cv;
Mat cov = Mat(2*206, 2*206, CV_8UC3, Vec3b(255, 255, 255));
for(int i = 8; i < cov.rows; i = i+2*2){
 for(int j = 8; j < cov.cols; j = j+2*2){
  Vec3b color;
  color[2] = std::min(255, static_cast<int>((data[(i-8)/4][(j-8)/4])/0.06*255.0));
  color[0] = 255 - std::min(255, static_cast<int>((data[(i-8)/4][(j-8)/4])/0.06*255));
  cov.at<Vec3b>(i, j) = color;
  cov.at<Vec3b>(i+1, j) = color;
  cov.at<Vec3b>(i, j+1) = color;
  cov.at<Vec3b>(i+1, j+1) = color;
 }
}
std::vector<int> params;
params.push_back(0);
bool success = imwrite("test.png", cov, params);.
```