SZAKDOLGOZAT

Szabó Márk

Debrecen

2007

Debreceni Egyetem Informatikai Kar

Vándor, az autonóm felfedező

Témavezetők:

Dr. Ispány Márton és *Dr. Istenes Zoltán*
egyetemi adjunktus , *egyetemi docens*
DE-IK , *ELTE-IK*

Készítette:

Szabó Márk
Programtervező
Informatikus B.s.c.

Debrecen

2007

Bevezetés

A SLAM kialakulása

A SLAM (**S**imultaneous **L**ocalization **A**nd **M**apping), azaz a Szimultán Helymeghatározás És Térképezés a robotika „Szent Grál”-ja volt egészen a legutóbbi évekig. A probléma egyszerű, de egy autonóm mobil robot számára annál fontosabb; egy ismeretlen környezetben ismeretlen kezdő pozícióból indulva inkrementálisan a hely egy térképének a létrehozása (mapping), valamint ezen térkép felhasználásával az aktuális helyzetének meghatározása (localization).

A feladatot már 1986-ban meghatározta a kutatói közösség, ekkortól kezdtek valószínűségi módszerekkel kísérletezni a robotikában is. A munka eredményesen indult; sikerült bizonyítani, hogy a különböző tájékozódási pontok (landmark) erősen korreláltak, és ez a korreláció a megfigyelések számának növekedésével egyre nő. Megtörténtek az első próbálkozások a Kalman-szűrő alkalmazására is. A későbbi megoldások azonban kettéválasztották a helymeghatározás és térképezés problémáját. Ez azon általánosan elfogadott vélemény következménye volt, hogy a térkép becsült hibái nem konvergálnak, hanem inkább egy határtalanul növekvő hibahatáron belüli véletlen bolyongásra hasonlít természetük. A kutatók vagy az egyik, vagy a másik részproblémára fókuszáltak, közelítő megoldásokat próbáltak adni. Ez a tájékozódási pontok közötti korreláció minimalizálásával, illetve figyelmen kívül hagyásával járt.

Az áttörés 1995-ben következett be, amikor rájöttek, hogy a kombinált térképezési és helymeghatározási probléma egyetlen becslési feladatként tekintve valójában konvergens, és felismerték azt is, hogy a tájékozódási pontok közötti korreláció, amit sok kutató minimalizálni igyekezett, kritikus fontosságú a probléma megoldása szempontjából; minél erősebb, annál jobb a megoldás. Azóta megnőtt az érdeklődés a terület iránt; elsősorban az algoritmusok számítási komplexitását igyekeznek csökkenteni, illetve az adat hozzárendelés (data association), és hurok lezárás (loop closure) finomítható még tovább.

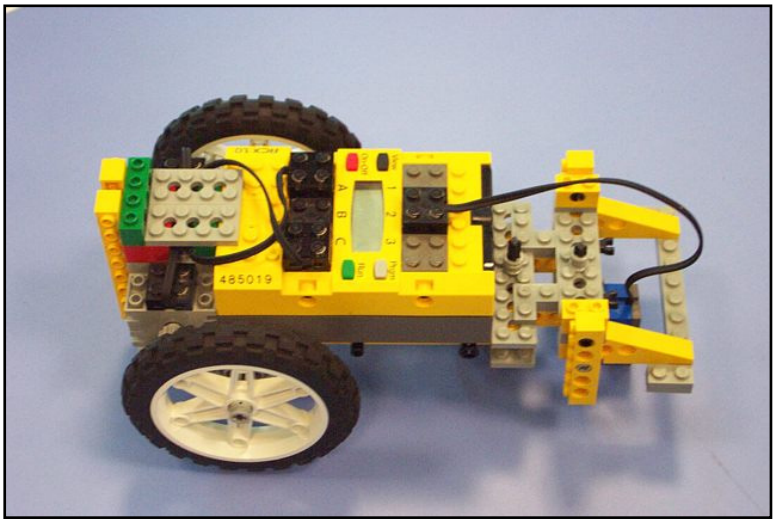
Mostanra a SLAM megoldottnak tekinthető, számos kutatócsoport épített épületben, természetes környezetben, víz alatt, illetve levegőben működő robotokat különféle, akár kombinált szenzorok felhasználásával.

A szakdolgozatomban ezen térképezési eljárást tanulmányoztam, és annak egy fontos részproblémájára, a környezeti objektumok felismerésére készíttetem egy implementációt. Az implementáció egy LEGO Mindstorm NXT roboton alapul, ennek rövid leírását közlöm a következő részben.

A LEGO Mindstorm NXT

Mi is a LEGO Mindstorm? A legelső verziója 1998-ból származik; ez RIS (**R**obotics **I**nvention **S**ystem) néven került piacra. Eredetileg a MIT Media Laboratory és a LEGO közötti együttműködés keretében, oktatási céllal jött létre. A LEGO moduláris építőkockái ideális platformot adtak egy egyszerűen használható, sokcélú készlet létrehozásához és ezzel lehetőséget adtak akár ipari robotok, akár más beágyazott rendszerek modellezéséhez.

A LEGO Mindstorm RIS az RCX elnevezésű speciális építőkocka köré épült. Ez egy (igen kisteljesítményű) számítógépet tartalmazott. Rendelkezett továbbá egy IR porttal; ezen keresztül nyílt lehetősége a PC-vel, vagy akár más RCX elemekkel történő kommunikációra. A készülék 3 input portjára érzékelők csatlakoztathatók; a készlethez két érintésérzékelő és egy fényérzékelő szenzort csomagoltak. (Később készültek nem hivatalos szenzorok is.) A szintén 3 output portra szervomotorokat lehetett kötni. A rendszert programozni eredetileg a készlethez mellékelt grafikus programozási nyelven (ROBOLAB) lehetett, de készült NQC (**N**ot **Q**uite **C**) néven egy C-szerű imperatív nyelv, illetve leJOS néven egy alap JVM az RCX platformra, ami lehetővé tette a robotok Java nyelven történő programozását.



1. ábra Egy Lego RCX robot

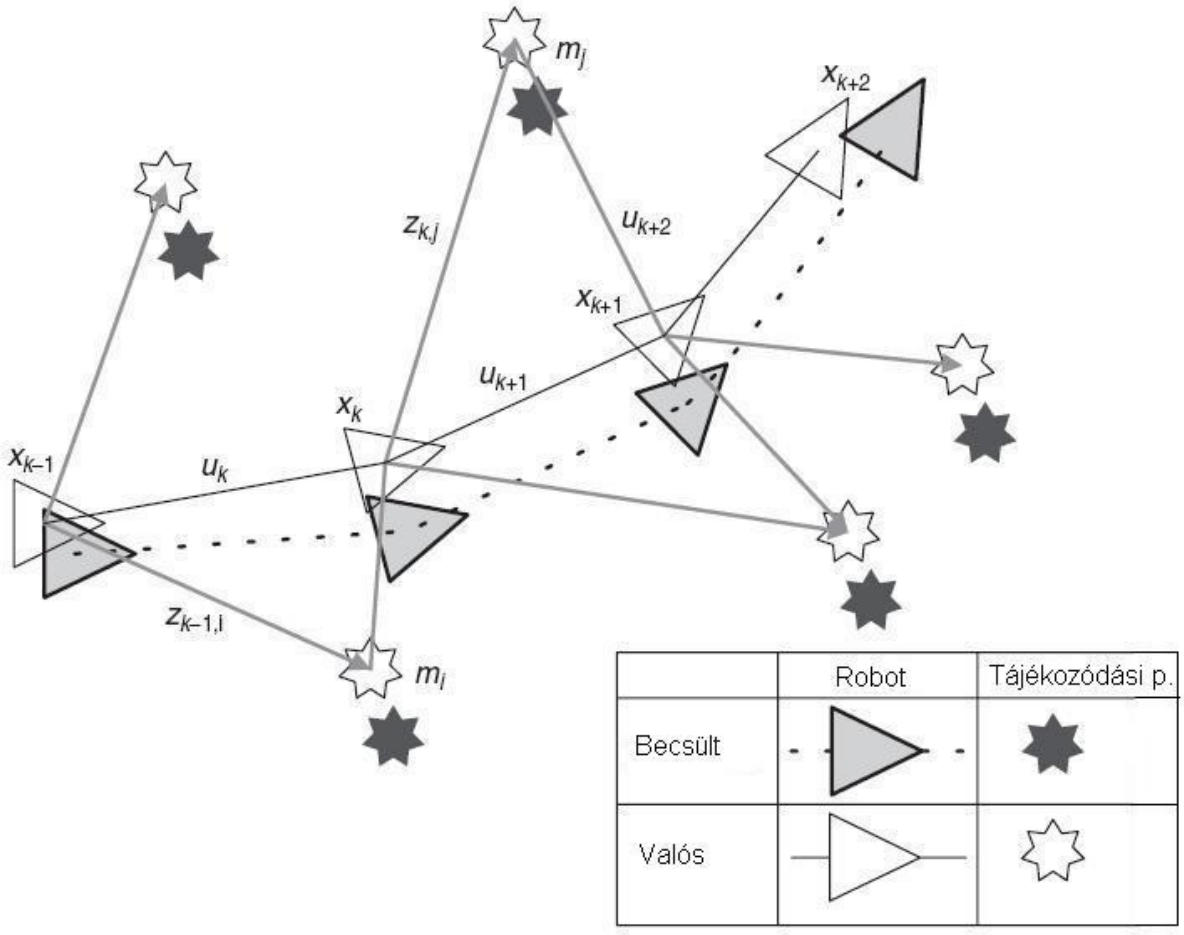
Kisebb lehetőségekkel rendelkező; integrált motorokkal, szenzorokkal, kevesebb I/O porttal rendelkező változatok után (Cybermaster, Scout, Micro Scout, Spybotics) 2006-ban jelent meg a Lego Mindstrom NXT a RIS 2.0 örököseként. Számos irányban bővültek képességei az elődhöz képest. Erősebb mikroprocesszort (32 bites RISC, 48 MHZ), nagyobb memóriát (256 KB Flash, 64 KB Ram), szabványos Bluetooth vezetéknélküli adóvevőt, 4 input portot kapott. A szenzorok mára teljesen digitálisak lettek, visszafelé kompatibilitás kiegészítő átalakítóval érhetőek el. Megjelentek új típusok is; az alapkészlet része lett egy hangérzékelő, és egy ultrahangos távolságérzékelő szenzor is. A motorok is megújultak; beépített forgásérzékelő szenzorral rendelkeznek a pontosabb irányítás érdekében. Erősebb támogatást kapnak a platformot továbbfejlesztők is. Munkájukat nyilvános dokumentációk és nyíltforráskódú firmware segíti. Nem sokkal a robot piacra kerülése után megjelent a leJOS Bluetooth kommunikációt alkalmazó távirányító API-ja iCommand néven, valamint röviddel ezután az NXJ, azaz a NXT-re írt JVM is. A szakdolgozatomban ezen API-kat szándékozom alkalmazni a robot programozására.

2. ábra Az NXT egyik „hivatalos” robotja, az Alpha Rex

Elméleti háttér

A SLAM formalizálása és struktúrája

Tekintsük át a térképezés alapproblémáját! A következő ábra egy tipikus SLAM feladatot ábrázol. A mobil robot egy ismeretlen területen halad keresztül, melyről semmilyen a priori tudással nem rendelkezik. A robot mozgása során beépített szenzorával (az aktuális pozícióhoz képest) relatív megfigyeléseket tesz az útközben előforduló tájékozódási pontokra. A 3. ábra alapján megadom a SLAM általános állapottér leírását.



3. ábra A SLAM probléma. Jelölések magyarázata a következő oldalon.

Minden **k** időpillanatban a következő mennyiségeket értelmezzük:

- $\boldsymbol{x_k}$: A robot aktuális helyzetét és orientációját leíró állapot vektor.
- $\boldsymbol{u_k}$: A vezérlő vektor. A k-1. időpillanatban az X_k pozícióba mozgást előíró utasítás.
- $\boldsymbol{m_i}$: Az *i*-ik tájékozódási pont helyzetét leíró vektor. (Itt kell megjegyeznem, hogy az alap SLAM-ben a tájékozódási pontok helye nem változik. Azaz nem mozognak!)
- $\boldsymbol{z_{ik}}$: A *k*-ik időpillanatban az *i*-ik tájékozódási pontra tett megfigyelés. (Az irodalom z_k-val jelzi, ha egy időpillanatban több tájékozódási pont észlelés történt, illetve ha az adott megfigyelés nem releváns az adott esetben.)
- $\boldsymbol{X_{0:k}} = \{x_0 , \dots , x_k\} = \{ X_{0:k-1} , x_k \}$: A robot mozgásának története. (Az eddigi lokációinak halmaza)
- $\boldsymbol{U_{0:k}} = \{u_0 , \dots , u_k\} = \{ U_{0:k-1} , u_k \}$: A robot mozgási parancsainak története. (Az eddigi vezérlő vektorok halmaza)
- $\boldsymbol{m} = \{m_1, \dots, m_n\}$: Az eddig megfigyelt összes tájékozódási pontot tartalmazó halmaz.
- $\boldsymbol{Z_{0:k}} = \{z_0 , \dots , z_k\} = \{ Z_{0:k-1} , z_k \}$: Az eddigi megfigyelések története.

A valószínűségi modell

A SLAM valószínűségi reprezentációban a feladat minden *k* időpillanatban a robot és a tájékozódási pontok helyzetének ***együttes eloszlását*** kiszámolni, feltéve ha adott a vezérlő vektorok és a tájékozódási pontok megfigyeléseinek története, valamint a robot kiinduló pozíciója. Azaz formálisan a következő feltételes valószínűséget kell meghatározni:

$$P(x_k, m \mid Z_{0:k}, U_{0:k,} x_0)$$

Egy ilyen feltételes valószínűség kiszámolása a legtöbbször nem egyszerű; jó lenne egy rekurzív módszert meghatározni a kiszámítására. Kiindulva a

$$P(x_{k-1}, m \mid Z_{0:k-1}, U_{0:k-1,} x_0)$$

értékből, mint az együttes eloszlás *k-1*-ik időpillanatbeli becsléséből, valamint az u_k vezérlő vektorból, és a z_k megfigyelésekből az együttes eloszlás a *k*-ik időpillanatban a Bayes-tétel alkalmazásával számolható. Ehhez szükségünk van még az állapot átmenet modellre (state transition model; motion model), amely leírja a vezérlő vektor hatását, valamint az észlelési modellre (observation model), amely a megfigyeléseket írja le.

Az észlelési modell azt a valószínűséget adja meg, amellyel az adott pozícióból a z_k megfigyelés tehető a robot és a tájékozódási pontok helyének ismeretében. Ennek formális leírása a következő:

$$P(z_k \mid x_{k-1}, m)$$

A mozgás modell valószínűségi kontextusban a következő módon írható le:

$$P(x_k \mid x_{k-1}, u_k)$$

Azaz az állapot átmenet egy ***Markov folyamat***, ugyanis az x_k állapot csak az őt közvetlenül megelőző x_{k-1} állapottól és az u_k vezérlő vektortól függ, amelyek mind a megfigyelésektől, mind a térképtől függetlenek.

Ezen eredmények ismeretében bevezethető a SLAM kétlépéses rekurzív jóslás (idő frissítés)-korrekció (mérés frissítés) alakja:

Idő frissítés (Time Update):

$$P(x_k, m \mid Z_{0:k-1}, U_{0:k,} x_0)=\int P(x_k \mid x_{k-1}, u_k)*P(x_{k-1}, m \mid Z_{0:k-1}, U_{0:k-1,} x_0)dx_{k-1}$$

Mérés frissítés (Measurement Update):

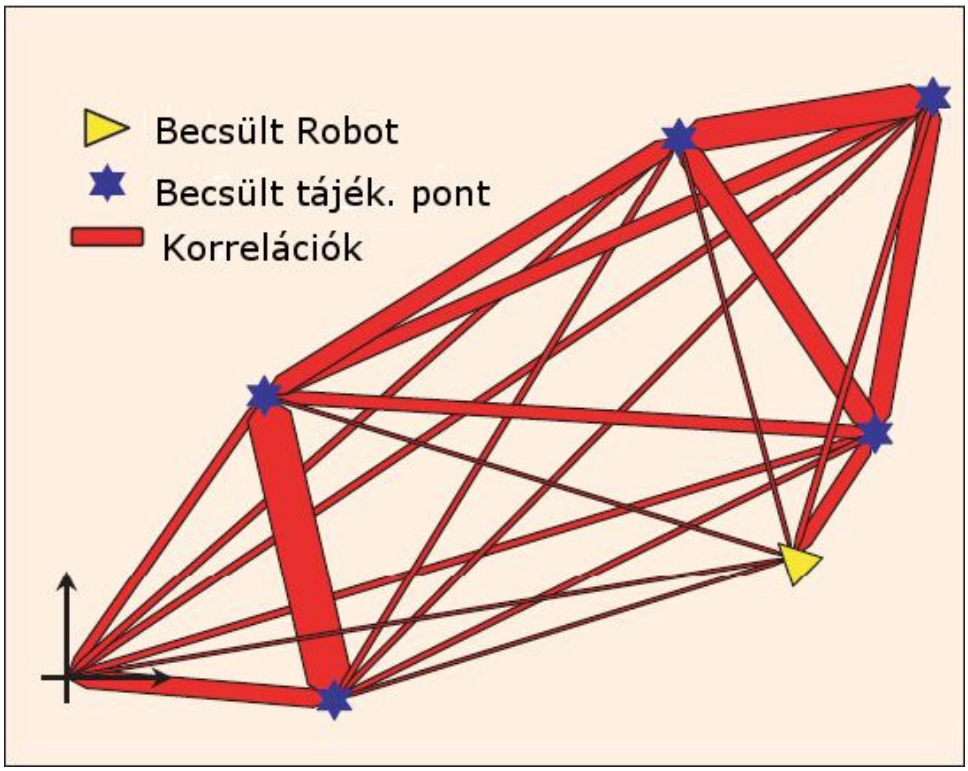
$$P(x_{k-1}, m \mid Z_{0:k-1}, U_{0:k-1,} x_0) = \frac{P(z_k \mid x_k, m)* P(x_k, m \mid Z_{0:k-1}, U_{0:k,} x_0)}{P(z_k \mid Z_{0:k-1}, U_{0:k})}$$

A valószínűségi modell működésének megértéséhez tekintsük ismét a 3. ábrát. Megfigyelhető, hogy a tájékozódási pontok becsült és valós helye közötti eltérés minden pontnál nagyjából azonos; ennek az az oka, hogy a robot helyzete a megfigyelések idején is csak hibákkal terhelten ismert. Ennek következménye, hogy a tájékozódási pontok erősen korreláltak. Továbbá ez azt jelenti, hogy a tájékozódási pontok egymáshoz viszonyított elhelyezkedése (m_i-m_j) nagy pontossággal ismert, habár az egyes tájékozódási pontok abszolút pozíciója csak pontatlanul. Fontos felfedezés volt az, hogy a megfigyelések számával monoton nő ez a korreláció. ([1]) Ez gyakorlatilag azt jelenti, hogy a tájékozódási pontok relatív elhelyezkedésére vonatkozó tudás mindig javul és sosem romlik; a robot mozgásától függetlenül.

A konvergenciát az okozza, hogy a robot megfigyelései „közel függetlennek" tekinthetőek. Nézzük meg ismét az ábrát! A robot az x_k pozícióban észleli az m_i és m_j tájékozódási pontokat. Amint a robot továbbhalad az x_{k+1} pozícióba, újra észleli az m_j pontot. Ez lehetővé teszi a robot és a tájékozódási pont helyzetének pontosítását az előző mérési ponthoz, az x_k-hoz képest. Ennek eredménye továbbterjed visszafelé is, és javul m_i helyzetének ismerete is, habár az nem is látszik az új pozícióból. Ez azért történik meg, mert a két tájékozódási pont erősen korrelált a megelőző mérések eredményeként (a relatív pozíciójuk jól ismert), továbbá amiatt, hogy ugyanazon mérés adatai alapján frissült helyzetük becslése, még erősebben korreláltak lettek. Figyeljük meg, hogy az x_{k+1} pozícióban a robot két új tájékozódási pontot észlel m_j-hez relatívan. Ezek azonnal kapcsolódnak (korrelálódnak) a térkép maradékához. Ezen tájékozódási pontok helyzetének későbbi frissítései frissítik m_j-t, és ezzel együtt m_i-t; és így tovább. Végül a tájékozódási pontok egy hálózattá állnak össze, ahol a kapcsolatok a pontok relatív elhelyezkedései (korrelációk), és a háló pontossága (értéke) növekszik, amikor egy új megfigyelést végez a robot.

A folyamatot vizuálisan jól szemlélteti egy olyan hálózat, ahol a csomópontokat rugók, avagy gumiszalagok kötik össze. Egy megfigyelés hatása a szomszédokra olyan, mintha a rugó vagy gumiszalag rendszert elmozdítanánk. Ennek hatása nagy a szomszéd csomópontok esetén, de a lokális merevségi (korrelációs) mutatóktól függően a távolsággal arányosan csökken a többi csomópontnál. Ahogy a robot felfedezi a környezetét, a rugók egyre inkább merevek lesznek. Végül tájékozódási pontok egy merev térképét kapjuk; a környezet egy pontos relatív térképét. Miután a térkép elkészült, a lokalizáció pontosságát csak a térkép és a használt szenzor minősége korlátozza. Elméleti szinten a relatív lokalizáció

pontossága megegyezik azzal, mintha egy kapott térkép alapján kellene meghatározni a robot helyzetét.



4. ábra A rugó-hálózat analógia. A tájékozódási pontok közötti korrelációt rugókként képzeljük el, így a térkép pontok és az őket összekötő „rugók” hálózatából áll. A rajzon a vastagabb vonalak erősebb korrelációt jelentenek.

A SLAM feladat megoldására legáltalánosabban használt eljárás a kiterjesztett Kalman-szűrő, az **EKF** ([1], [2], [4], [5], [6], [14]) és a **FastSlam** ([2], [7]). Léteznek ezeknél újabb megoldások, valamint ezek továbbfejlesztett változatai. ([8], [9], [10], [11])

Az alkalmazott eljárástól függetlenül az algoritmus működéséhez szükséges még megadnunk:

- Az észlelési modellt
- Az mozgási modellt
- A térkép reprezentációját

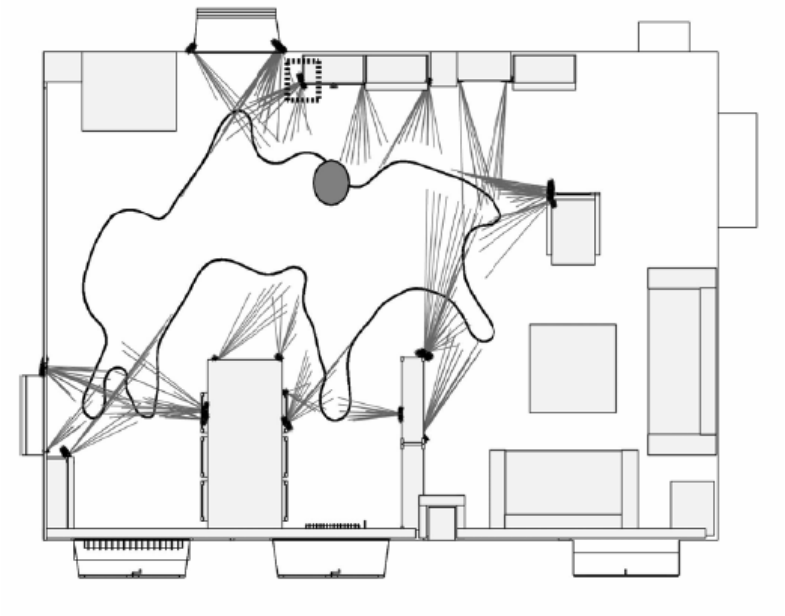
Az észlelési és mozgási modell megadására általános egyenletek több helyen fellelhetők.([1], [14]) A térkép reprezentálására is több technika alakult ki. Ilyenek például a

grid-alapú (grid-based, [12], [15]), a mintaillesztő (scan matching, [13], [15]), topologikus (topological, [15]) illetve a jellegzetesség-alapú reprezentációk (feature-based, [5], [15]).

A dolgozatom írása során igyekeztem megismerkedni a térképezés és a robotika alapfogalmaival, valamint az elmélet mellett igen fontosnak tartottam az eljárások gyakorlati kipróbálását. Sajnos a megadott időkeretben nem nyílt lehetőség egy teljes SLAM implementáció elkészítésére, mert bár a szükséges eljárások több helyen is fellelhetők, de azok konkrét hardverre implementálása korántsem egyszerű feladat. Ugyanis bár a SLAM feladatot megoldják a fentebb említett módszerek, de a robot valós világgal történő interakciója, a térkép tényleges reprezentációja mind egyedi megoldást kívánnak. Ezért a standard EKF algoritmus egy igen fontos részfeladatát igyekeztem implementálni; ezt a következő részben mutatom be.

A jellegzetesség-alapú térkép reprezentáció

Az EKF eljárás jellegzetesség-alapú térkép reprezentációt használ. A reprezentáció a környezetet jellegzetes (gyakran a szenzor által legkönyebben/legbiztosabban érzékelhető) alakzatok halmazaként képzeli el. Ilyen alakzatok lehetnek például épületen belüli környezet esetén a sarkok és az élek.



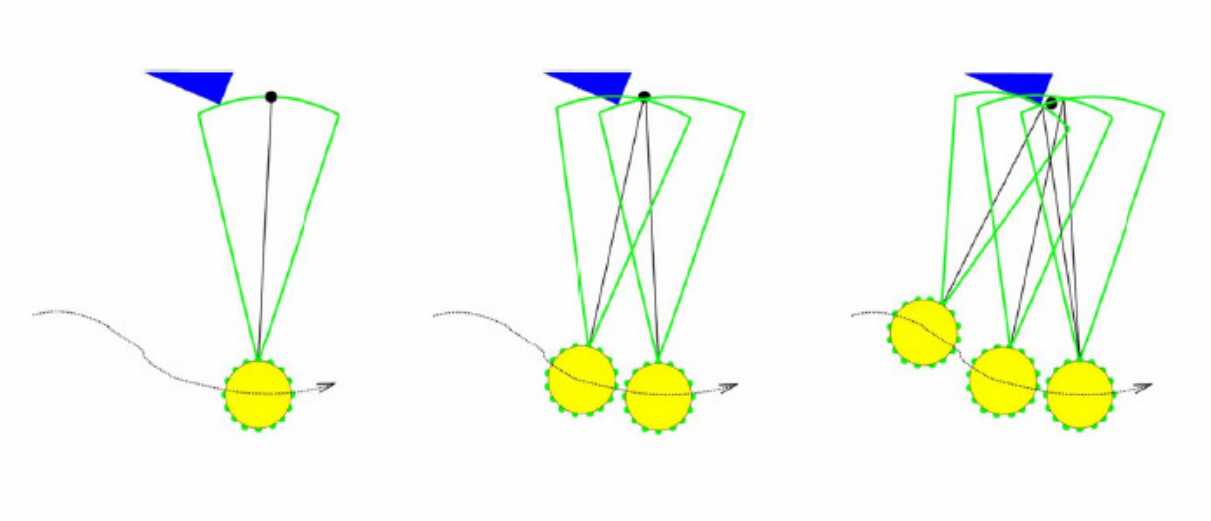
5.ábra Egy nappali jellegzetesség-alapú térképe. A jellegzetességek a sarkok, a felismerési algoritmus TBF.

A jellegzetesség-alapú térkép építésénél meg kell határoznunk azt, hogy mik alkotják majd a térképet, és valamilyen eljárást kell adnunk ezen jellegzetességeknek a szenzor adatokból történő kinyerésére. Az implementációmnál sarkokat választottam felismerendő jellegzetességként, míg a **TBF** (**T**riangulation-**B**ased **F**usion) algoritmust a sarkok felismerésére. (Gyakran választják még az él felismerést is; erre a Hough-transzformáció és a RANSAC algoritmus is alkalmas. Találkozhatunk továbbá kombinált megoldásokkal is.) A továbbiakban megvizsgáljuk a TBF algoritmust.

A TBF algoritmus alapjai

A TBF algoritmust kifejezetten ultrahanggal működő szenzorokkal (szonár) történő együttműködésre fejlesztették ki. A robotikában elterjedtebb lézeres szenzorokkal szemben általában kisebb hatótávval és pontossággal rendelkeznek, valamint hajlamosak fals mérési adatok generálására. Ráadásul működésüket befolyásolják bizonyos környezeti tényezők; például a szél, a levegő nedvességtartalma és a hőmérséklet. Előnyükre válhat ellenben olcsóságuk, valamint az, hogy több szenzor elhelyezésével a robot egész környezete megfigyelhető. Én azért választottam ezt az eljárást, mert a Lego NXT készlethez egy ultrahangos szenzor is tartozik; ennek részleteit később mutatom be.

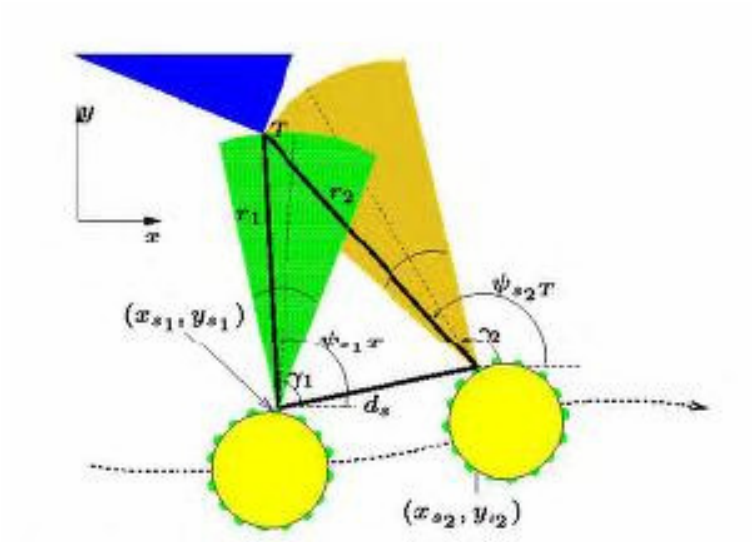
A TBF algoritmus alapelve jól ismert a repülőgépek radarral történő követéséből; ez az úgynevezett háromszögeléses technika. Ennek szemléltetésére figyeljük meg a 6. ábrát!



6. ábra A háromszögeléses technika. A robot szonárja egy visszaverődést

érzékel. Következő méréskor ismét visszaverődés érkezik abból az irányból. Ekkor a két körszelet metszéspontja az objektum helyzetének jobb becslését adja, mint a szenzor egyes mérései önmagukban. (A szonár érzékelés önmagában csak azt állítja, hogy a visszaverődést keltő objektum a mérő nyalábban van.) Több háromszögelési lépés után az objektum helyzetének egy „elég jó” becslését kapjuk.

Röviden összefoglalva a TBF algoritmus működése: a robot mérési adatait egy csúszó ablakban (sliding window) tárolja egy meghatározott ideig. (A mérések története.) Amikor új mérést végez a robot, akkor az ablak legrégebben meglévő mérési adatait kidobja (legbaloldalibb elemek), a többit egyel balra csúsztatja, és végül az új adatokat a legjobboldalibb helyre illeszti. Ezután az új elemeket egyenként megpróbálja összepárosítani az összes eddigi elemmel; arra a kérdésre keresi a választ, hogy a két elem által észlelt visszaverődést okozhatta-e ugyanaz az objektum. (Egyfajta szavazási algoritmus.) Ha igen, akkor megpróbálja elvégezni a háromszögelést, és ha az eredmény kielégítő, akkor sikerült egy objektumot találni. Minél több háromszögelés erősíti meg az objektum létezését (az irányából érkező visszaverődés), annál pontosabbá válik az objektum helyzetének becslése. A feltételeket megadó egyenletek megadása előtt nézzük meg a 7. ábrát!



7. ábra A háromszögelési feladat és jelölései

Tehát a jelölések:

- $(\mathbf{x_{si}}, \mathbf{y_{si}})$: A robot pozíciója az i-ik méréskor.
- $(\mathbf{x_T}, \mathbf{y_T})$: A körszeletek metszépontja.
- $\mathbf{r_i}$: A metszépont távolsága a robottól.
- δ : A szenzor mérő nyalábjának nyilási szöge.
- γ_i : A szenzor mérő nyalábjának tekintési szöge az abszolút koordinátarendszerben megadva.

Az előző jelöléseket felhasználva a következő módon számolható a metszéspont:

$$(x_T - x_{si})^2 + (y_T - y_{si})^2 = r_i^2$$

$$\arctan(y_T - y_{si} / x_T - x_{si}) \in [\, y_i - \delta/2 \,, y_i + \delta/2]$$

Ekkor az első egyenlet megoldásai i=1,2 mellett:

$$x'_T = x_{s1} + (1/d^2_s)(d_{xs}d^2_r \pm |d_{ys}|(r^2_2 d^2_s - d^4_r)^{1/2})$$
$$y'_T = y_{s1} + (1/d^2_s)(d_{ys}d^2_r \pm |d_{xs}|(r^2_2 d^2_s - d^4_r)^{1/2})$$

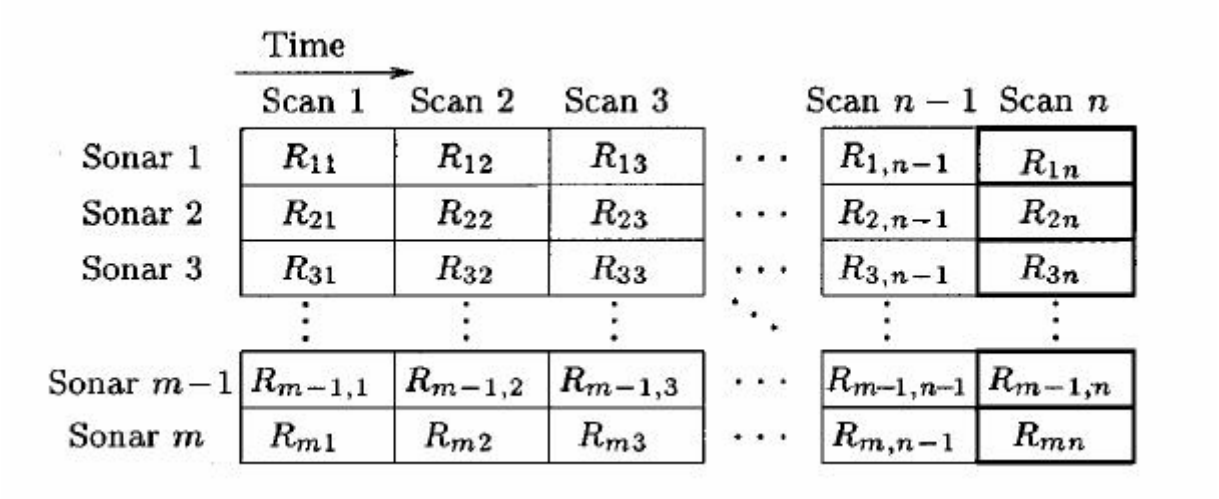
ahol:

$$d_{xs} = x_{s1} - x_{s2}$$
$$d_{ys} = y_{s1} - y_{s2}$$
$$d^2_s = d^2_{xs} + d^2_{ys}$$
$$d^2_r = (r^2_1 - r^2_2 - d^2_s)/2$$

A hibás (x'_t, y'_t) megoldásokat az eredeti egyenletekbe történő behelyettesítéssel szűrhetjük ki.

A TBF algoritmus implementálása

A TBF algoritmus szonárral felszerelt robot hardverrel használható. Képes több, tetszőleges pozícióban, de egy síkban elhelyezett szonár adatai alapján működni. Legyen a rendelkezésre álló szonárok száma *m*, és a csúszó ablak hossza *n*. Ekkor a csúszó ablak egy mxn-es mátrix, ami az egyes szonárok adott időpontbeli méréseit tartalmazza. A következőképpen képzelhetjük el:



8. ábra A csúszó ablak (sliding window) általános leírása

ahol:

$$R_{ij} = (x_{sij}, y_{sij} , \gamma_{ij}, r_{ij})$$

Azaz egy bejegyzés tartalmazza a robot méréskori pozícióját, a szenzor tekintési szögét valamint a szenzor által mért értéket. Minden oszlop egy teljes, körkörös mérést reprezentál, amelyben minden szenzor részt vesz. A sorok az egyes szenzorok mérési bejegyzéseit tárolják a megadott számú mérésre visszamenőleg. A mérések között tipikus beltéri/irodai környezetben általában 5-25 cm-t érdemes mozgatni a robotot. A legrégebbi (első oszlopbeli) értékeket egy új mérés végzésekor töröljük, míg az összes többit egyel balra mozgatjuk. Az új bejegyzések az n-ik oszlopba kerülnek minden esetben. Ezen paraméter megválasztása nyilvánvaló hatást gyakorol az algoritmusra; minél több oszlopunk van, annál több háromszögelési kísérletet végezhetünk el, és annál több egyezést találhatunk,

természetesen ezzel párhuzamosan növekszik a táblázat memória igénye. Tekintsük meg most a 9. ábrát, ami a TBF algoritmus pszeudokódját mutatja be!

```
for i = 1 → m{                                                              (1)
  if r_in < r_max{                                                          (2)
    G := {R_in}                                                             (3)
    n_t := 0,   x̂_T := x_{s_in} + r_in cos(γ_in),   ŷ_T := y_{s_in} + r_in sin(γ_in)   (4)
    x_min := y_min := maxInt,   x_max := y_max := −maxInt                   (5)
    for j = (n − 1) → 1                                                     (6)
      for k = 1 → m                                                         (6)
        if r_kj < r_max                                                     (7)
          if (x̂_T, ŷ_T) ∈ beam of R_kj                                      (7)
            r_e := sqrt((x̂_T − x_{s_kj})^2 + (ŷ_T − y_{s_kj})^2)            (7)
            if |r_e − r_kj| < d_1/(n_t + 1)                                 (7)
              if TriangulationPossible (In:R_in, R_kj; Out:x_T^tri, y_T^tri)   (8)
                G := {G, R_kj}                                              (9)
                x̂_T := 1/(n_t+1) (n_t x̂_T + x_T^tri)                        (9)
                ŷ_T := 1/(n_t+1) (n_t ŷ_T + y_T^tri)                        (9)
                n_t := n_t + 1                                              (9)
                if x_T^tri < x_min then x_min := x_T^tri                    (10)
                if x_T^tri > x_max then x_max := x_T^tri                    (10)
                if y_T^tri < y_min then y_min := y_T^tri                    (10)
                if y_T^tri > y_max then y_max := y_T^tri                    (10)
  }
  if n_t ≥ 1                                                                (11)
    if (x_max − x_min) + (y_max − y_min) > d_2 then n_t := −n_t              (12)
    RefineTriangulationPoint(In: G, Out: x̂_T, ŷ_T, P_T)                     (13)
    Store n_t, T̂, P_T and possibly G                                        (14)
}                                                                           (15)
```

9. ábra A TBF algoritmus. A részleteket lásd lentebb.

(1.) A legkülső ciklusban végigvesszük a csúszó ablak legújabb bejegyzéseit.

(2.) Ha a bejegyzés távolság értéke kisebb, mint egy adott konstans, akkor származhat a mérés egy éltől. (Tapasztalatok szerint a szenzorok -típustól függően- csak egy bizonyos távolságig képesek biztonsággal érzékelni az éleket.)

(3.) Egy halmazt hozzunk létre; ebbe az adott bejegyzéssel összekapcsolt bejegyzések kerülnek. A halmazt R_{in}-el inicializáljuk.

(4.) Felállítjuk az alaphipotézist. A T'=(x'_t, y'_t)-t úgy inicializáljuk, mintha az objektum, amelyről a visszaverődés érkezett a nyaláb közepén lenne.

(5.) Inicializáljuk azon változókat, amelyek a háromszögelési eredmények közötti eltéréseket követik.

(6.) A belső ciklusban az összes többi oszlopon végiglépkedünk és igyekszünk az adott bejegyzéshez háromszögelési partnert találni. (7.)

(7.) Háromszögelési partner lehet egy bejegyzés, ha teljesíti a következő feltételeket.

1) A bejegyzés távolság értéke kisebb, mint egy adott konstans. (2.)
2) T' látható volt a bejegyzés mérésekor a szonár nyalábban.
3) A mért távolság nem különbözik nagyon a T' koordinátái alapján számolttól. (A küszöb, $d_1/(n_t+1)$ a sikeres háromszögelések számával csökken. A d_1 szenzor függő konstans.)

(8.) Ha ezt a pontot elértük, akkor a két bejegyzés nagy valószínűséggel ugyanarról az objektumról tárol mérési eredményt. Ekkor az előző részben megadott egyenletekkel megpróbáljuk meghatározni az (x^{tri}_T , y^{tri}_T) metszéspontot.

(9.) Sikeres háromszögelés esetén hozzáadjuk a bejegyzést a G halmazhoz, a T' értéket finomítjuk, valamint növeljük a hipotézis számláló (n_t) értékét. (Figyeljük meg, hogy $n_t=0$ esetén csupán lecseréljük a T' a kiszámolt metszéspontra.)

(10.) Frissítjük a háromszögelési eltérést követő változókat.

(11.) Végül ha megpróbáltuk már az összes bejegyzéssel a párosítást, és n_t legalább 1, akkor elképzelhető, hogy találtunk egy sarokpontot.

(12.) Ha a sikeres háromszögelések között túl nagy az eltérés (szintén szenzor függő konstanshoz, a d_2-höz viszonyítva.), akkor az objektum *nem jól reprezentálható* pontszerűen, ezért n_t értékét $-n_t$-re változtatjuk.

(13.) A háromszögelések „elég jó" közelítést adnak, de további finomítás is végezhető. [16.]

(14.) A kapott pont és a hozzátartozó adatok elmentése.

(15.) A külső ciklus folytatása.

A munkám pontos elméleti hátterét ezzel megadtam. A téma, mint ahogy a sok irodalmi utalás is mutatja, nagyon széles. A felmerült problémákra sok megoldás létezik. Ezek közül próbáltam meg kiválasztani egy várhatóan pontatlanabb, nem professzionális tudású hardver lehetőségeihez mérten a legjobbat. A következő részben áttérek mind a hardver, mind a szoftver implementációjának leírására, valamint az implementációmmal végzett mérésekre, azok értékelésére.

Implementáció

A robot hardver

Az implementációs részletek tárgyalását kezdjük magával a hardverrel. Már korábban is utaltam arra, hogy szakdolgozatom célja nem csupán a fellelhető elméleti eredmények megismerése volt, hanem fontosnak tartottam azok kipróbálását a gyakorlatban is. Alkalmazhattam volna saját készítésű, esetleg piaci szimulátort az algoritmusok futtatására; de ezek minden komplexitásuk ellenére sem képesek hűen modellezni a valóságban előforduló, sokszor meglepő eseményeket. A valós világgal történő interakciónál tipikusan teljesül a közismert paradoxon: „2*2 néha 5”. Ezt saját magam is tapasztalhattam; bár a szoftveres oldalon is lesz róla szó, de mindenképpen ideillik és tanulságos a következő eset. Röviden a probléma: az NXT firmware a motormozgatással párhuzamosan más cselekvést is végezhet. Ez időnként nem kívánatos lehet; gondolták az iCommand API tervezői. Éppen ezért olyan motormozgató metódust írtak, amely kivárta azon időt, ameddig a motor forog. Ennek megoldása roppant egyszerű; egy ciklus, amely addig várakoztatja a programot, amíg a motor annyit nem fordult, amennyit kell. (Ehhez segítséget a motorok tachométere ad. Lásd később.) Viszont a körülmények összejátszása nyomán a motor sokszor korántsem annyit fordul, mint amennyit megkívántunk tőle! Az eredmény: egy végtelen ciklus. Tehát a hagyományos programozásban bevált gondolkodásunk sok esetben hibás megoldást generálhat!

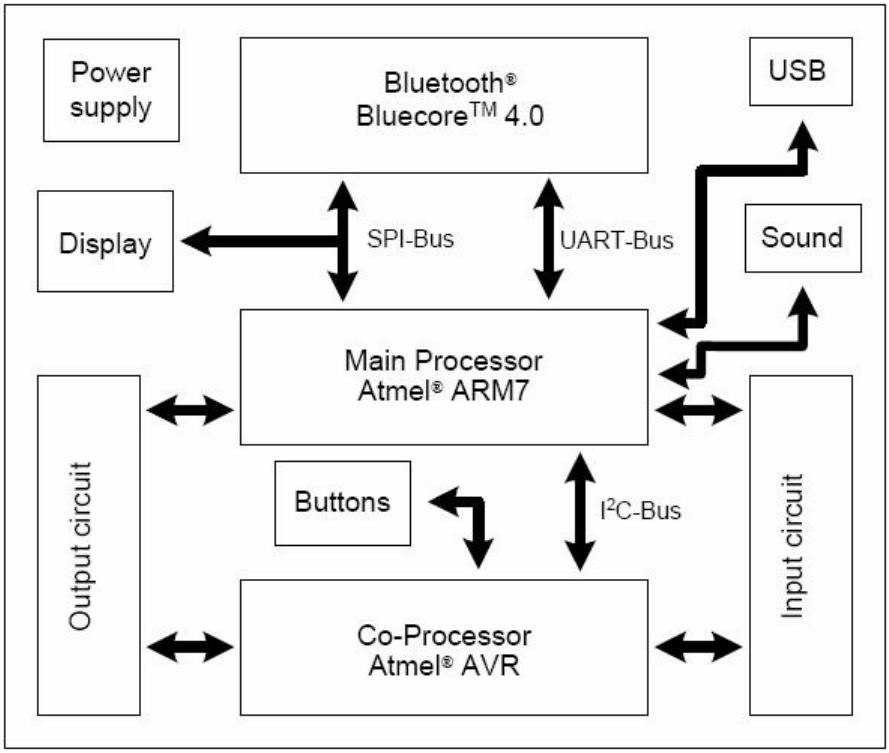
Azt hiszem sikerült ezzel a történettel világossá tennem, hogy miért is ragaszkodtam a hardveres implementáláshoz. Sajnos tanulási, kutatási célra alkalmazható, kereskedelmi forgalomban kapható robot nem sok van. Ráadásul ezek többsége hazánkban hozzáférhetetlen, és meglehetősen drága. Továbbá a robot egy adott feladatra történő adaptálása nehézkes. Ezért bizonyult jó választásnak a Lego NXT hardver.

A Lego-t azt hiszem minden Olvasó ismeri, különösebb bemutatást nem igényel. Különböző méretű és funkcióval rendelkező, ún. téglák alkalmazásával gyakorlatilag bármilyen szerkezet létrehozható. A fejlettebb készletek tartalmaznak továbbá fogaskerekeket, pneumatikus alkatrészeket, és elektromotorokat. Ezt a széles „arzenált” kiegészítve hozta létre a Lego RCX és NXT robot hardvereit. Ezen készletek legfőbb

újdonsága az „intelligens tégla” (intelligent brick), azaz egy kompakt, kis teljesítményű számítógép, valamint a különféle szenzorok és fejlettebb motorok. A bevezetőben említettek alátámasztására röviden bemutatom a jelenleg legfejlettebb készlet, az NXT legfontosabb elemeit.

Az „intelligens tégla” adatai

A rendszer lelke egy Atmel® AT91SAM7S256 32 bites, ARM® technológiájú, 48 MHZ órajelű RISC processzor. A központi memória mérete 64 Kb, valamint háttértárként rendelkezik 256 Kb flash memóriával. A központi processzort támogatja egy Atmel® ATmega48 típusú, 8 bites, 8 MHZ órajelű koprocesszor. Az eszköz képes Bluetooth v2.0 + EDR System szabványú Bluetooth vezetéknélküli csatlakozásra, és ezen keresztül SPP (**S**erial **P**ort **P**rofile) kapcsolat létrehozására. Ezt egy dedikált chip, egy CSR BlueCore™ 4 biztosítja. Az eszköz képes továbbá USB 2.0 szabványú kommunikációra is. I/O rendszere 3 input és 4 output portot támogat. Rendelkezik továbbá egy 100 x 64 pixeles fekete-fehér LCD kijelzővel, és egy egyszerű hangszóróval. Táplálásáról 6 AA méretű elem gondoskodik.



10. ábra Az NXT „intelligens tégla” sematikus felépítése.

Az adatokat tekintve méretéhez és céljához képest impresszív a tégla képessége. A háttértár mérete lehetne nagyobb (ráadásul a firmware is ezen foglal helyet csökkentve a szabadon felhasználható hely méretét) , mint ahogy a memória mérete is. Összességében azonban nagy előrelépés az elődhöz képest, és az I/O kezelése is elfogadható sebességű.

A szenzorok

Szenzorokból négyet tartalmaz az alap készlet:

1) Fényérzékelő szenzor: 255 szürkeárnyalatot képes érzékelni, alapvetően a vizsgált felület fényességének meghatározására való.
2) Hangérzékelő szenzor: Környezeti hangokat képes érzékelni db skálán.
3) Nyomásérzékelő szenzor: Egyszerű szenzor; valamilyen tárgyal kapcsolatba kerülve ütközést, illetve lenyomást képes érzékelni.
4) Ultrahangos szenzor (szonár): A készlet legfejlettebb, digitális szenzora. Egy magas frekvenciás hanghullámot bocsát ki, majd a visszhangot felfogva képes érzékelni a szenzor előtt lévő objektumot, és annak távolságát. Hatékonysága függ a környezeti tényezőktől, valamint az objektum anyagától és felszínétől. (Egy érdes, kerekded tárgyat nehezen észlel, akár láthatatlan is lehet a szenzor számára. Legjobban sima, egyszerű felszínű felületeket ismer fel.) A távolságot egy 255 fokozatú skálán méri; tehát egy objektum legfeljebb 254 cm távolságról ismerhet fel a szenzor.

A nyíltan elérhető hardver specifikációknak köszönhetően ([19]) sok külső cég is gyárt a készlettel alkalmazható szenzorokat. Találhatunk ezek között digitális iránytűt, giroszkópot, színeket is felismerő fényszenzort, stb. Átalakító segítségével használhatóak az RCX szenzorai is a visszafelé kompatibilitásnak köszönhetően.

A motor

A Lego NXT készlet 3 motort tartalmaz. Ezek újdonsága, hogy rendelkeznek egy beépített tacométerrel, így pontosabban vezérelhetőek a motorok, mint az elődök egyszerűbb modelljei.

A szoftver környezet

A Lego ezen jó képességű robotika készletet elsősorban gyerekjátékként gyártja. Ennek megfelelően a készlethez mellékelt programozási eszköz meglehetősen fapados, csupán alapszolgáltatásokat nyújt, elsősorban programozásban kevéssé jártasak számára lehet használható. Ismerve azonban a piac igényeit, a Lego elérhetővé tette mind a tégla firmware-jét, mind a hozzátartozó szoftver specifikációkat ([21], [22]). Így akár teljesen új programozási nyelvet és eszközt is fejleszthet bárki a készlethez, de megoldható a gyári firmware lecserélése is. Ezen téren is jól szerepel a Lego NXT; sok fejlesztőkörnyezet és nyelv áll rendelkezésre.

Már az elődökhöz is elérhető volt egy egyszerűsített C nyelv, az NQC (**N**ot **Q**uite **C**). Ennek leszármazottjának tekinthető az NXC (**N**ot e**X**actly **C**, [23]). Az NXC egy C szintakszisú nyelv [25], amely szintén az ezen csoport által fejlesztett NBC–re (**N**ext **B**yte **C**ode) fordít [24]. Ez az NBC nyelv pedig az ARM processzor gépi nyelvén alapul; akár közvetlenül ezzel is programozható a robot, de a magasszintű NXC alkalmasabb rá.

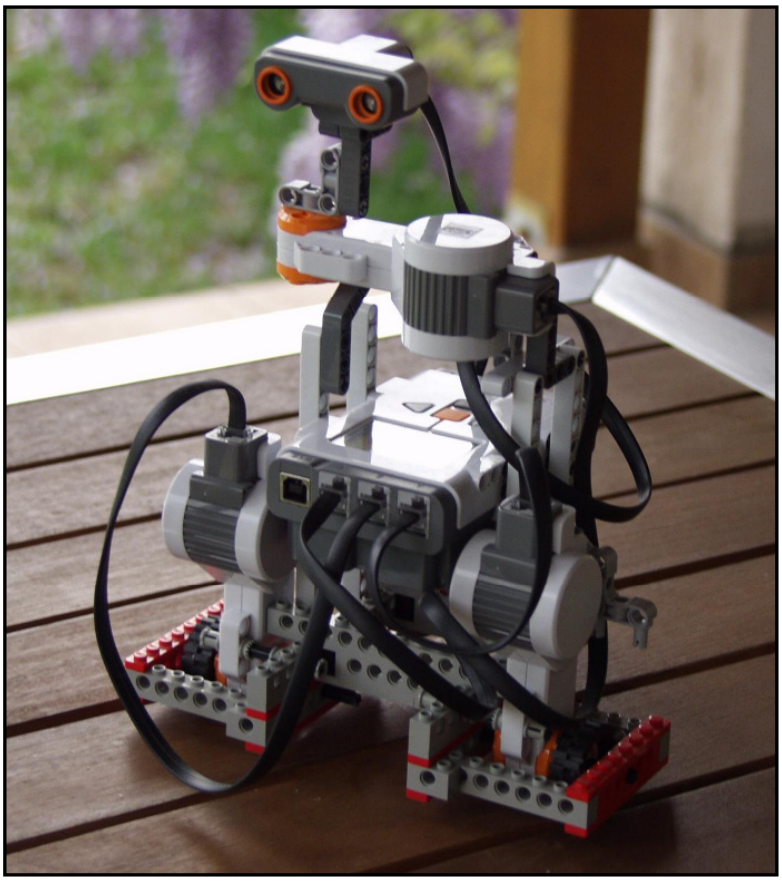
Szintén nem újdonság, hogy elérhető Java nyelvű fejlesztőkörnyezet [26] is. Létezik egy egyszerű JVM, azaz egy új firmware a platformra, ez az NXJ [28]. Emellett a Bluetooth kommunikációs protokollra ([20]) építve egy számítógépes távirányítást lehetővé tevő csomag is elkészült iCommand néven.

Végül alkalmazható a National Instruments Labview® szoftvere is a készlet programozására. A Labview® egy grafikus programozási nyelv; elsősorban speciális mérőkártyákkal végzett méréseknél alkalmazzák, de általános feladatok megoldására is felhasználható. A készlethez mellékelt szoftver is egy erősen „butított” Labview®. Az eredeti változathoz letölthető egy toolkit ([29]), amely segítségével ezen IDE-ből programozható a robot (természetesen eszközök sokkal szélesebb tárházából válogathatunk), illetve létrehozhatóak új elemek is a „butított” változatban történő felhasználásra ([30], [31]).

Tehát sok környezet és nyelv áll rendelkezésünkre a Lego NXT készlet programozásához. Ezek ráadásul különféle paradigmákat is képviselnek, emellett akad mind magas-, mind alacsonyszintű nyelv; így mindenki megtalálhatja a számára legmegfelelőbbet, vagy a feladathoz leginkább hozzáillőt.

A Vándor I.

A dolgozatomhoz összeállított robot csak a feladathoz szükséges funkcionalitást biztosítja. Képes a síkon mozogni, illetve forgó szonárjának köszönhetően a környezetét körkörösen felderíteni. A robotot a 11. ábra mutatja be.



11. ábra A Vándor I.

A robotra tekintve feltűnő lehet annak magas felépítménye. Ennek oka a motorok függőleges beépítése. A készlet motorjai, és az „intelligens tégla” sajnos nagy kiterjedésű, így kissé nehezebben kezelhető. Ez a felépítmény viszont előnyös lett azért, mert így a szenzor viszonylag magasan helyezkedik el, így tipikus irodai környezetben általában több objektum látható. (Jó példa erre a radiátor, ami talaj közelben nem látszik.) Szintén szembetűnő lehet a masszív alváz. Ezt az elemekkel feltöltött „intelligens tégla” tömege tette szükségessé.

A robot helyváltoztatásra 2 motort használ 2 meghajtott kerékkel. A felépítésnek, és a hátsó csúszótalpnak köszönhetően a robot képes helyben fordulni. A robot legmagasabb

pontján került elhelyezésre az ultrahangos szenzor. Ez egy motor segítségével teljesen körbeforgatható. Azért kellett így beépíteni a szonárt, mert a TBF algoritmus sok szenzor alkalmazásával ad jó eredményt. A készlethez azonban csak 1 szenzort adnak, továbbá a Lego felhívja a figyelmet arra, hogy több szenzor együttes alkalmazása növelheti a fals érzékelések számát. Így egyetlen jó megoldásként a szenzor forgatása maradt. Ez viszont olyan előnnyel is jár, hogy a forgatás szögétől függően szimulálható több szenzor megléte. (A szonárt adott szöggel elforgatjuk, majd végzünk egy mérést. Az eljárást addig ismételjük, amíg egy teljes fordulatot meg nem tettünk.)

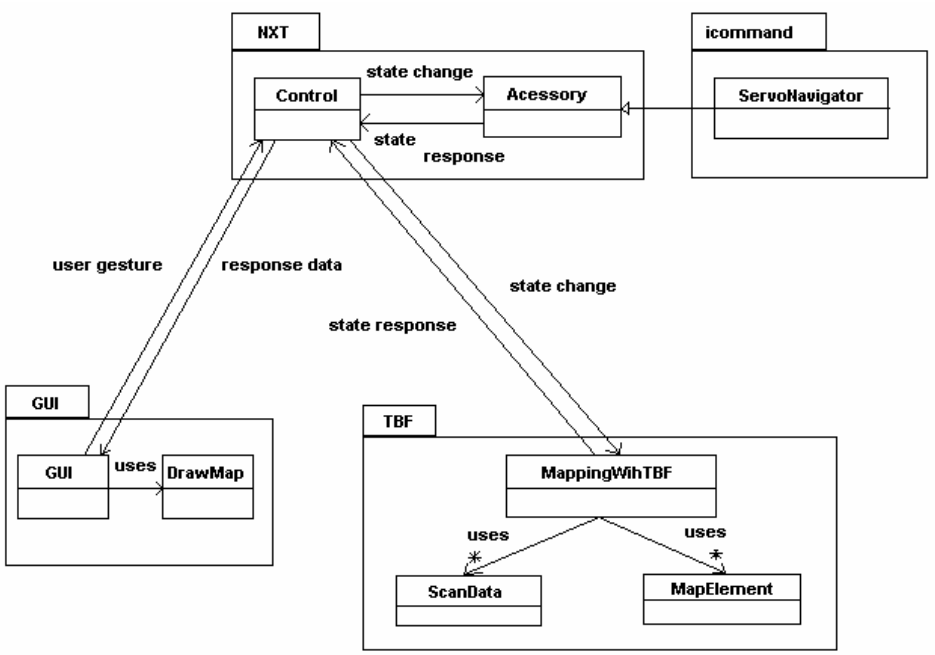
A Vándor I. struktúrája lehetővé teszi a mérsékelt továbbfejlesztést. Sajnos az alapfeladat megoldása a készlet aktorjait kimerítette, így további funcionalitás (pl. robotkar) beépítésre csak egy további készlet felhasználásával nyílik lehetőség.

A térképező szoftver

Elérkeztünk ezzel a térképező alkalmazás leírásához. Amint azt már a bevezetőben is említettem, én a készlet programozására a Java iCommand API-t választottam. A választásomnak több oka volt; mindenekelőtt ragaszkodtam az OO módszertanhoz, és ez jelenleg csak a Java API-n át érhető el. Másrészt a programozás kezdetén még csak készült az időközben megjelent NXJ firmware. (Ráadásul ez a dolgozat írása idején még csak alpha stádiumban van, és nem használható vele az ultrahangos szenzor.) Továbbá az indokolta még a választásom, hogy a várhatóan komoly I/O feldolgozás mellett a pontosság érdekében nagy tárhelyet igénylő számreprezentációt kellett alkalmaznom, ráadásul a TBF módszer ezek egy táblázatát hozza létre. Ezáltal a roboton futtatva az egész programot memória- és sebesség problémákkal kerültem volna szembe. Az iCommand API felhasználásával lehető vált a számításigényes feladatok asztali számítógépen történő végrehajtása; hátrányként jelentkezik viszont a végrehajtás során a Bluetooth átviteli késleltetése. Szerencsére a kapcsolat kiépülése után a jelenség még az elfogadható kategóriában marad.

A program célja a TBF algoritmus implementálása, és az adatok felhasználásával egy egyszerű térkép építése. Nyilvánvalóan a kész térképpel szemben nem lehetnek szigorú elvárásaink, mivel célja csupán a TBF eredményeink jobb bemutatása. Egy teljes térképező rendszer esetén a térképet a SLAM módszer hozná létre, a TBF csupán a tájékozódási pontok kiszűrését végezné. Ezt figyelembe kell vennünk az eredmények értékelésénél is.

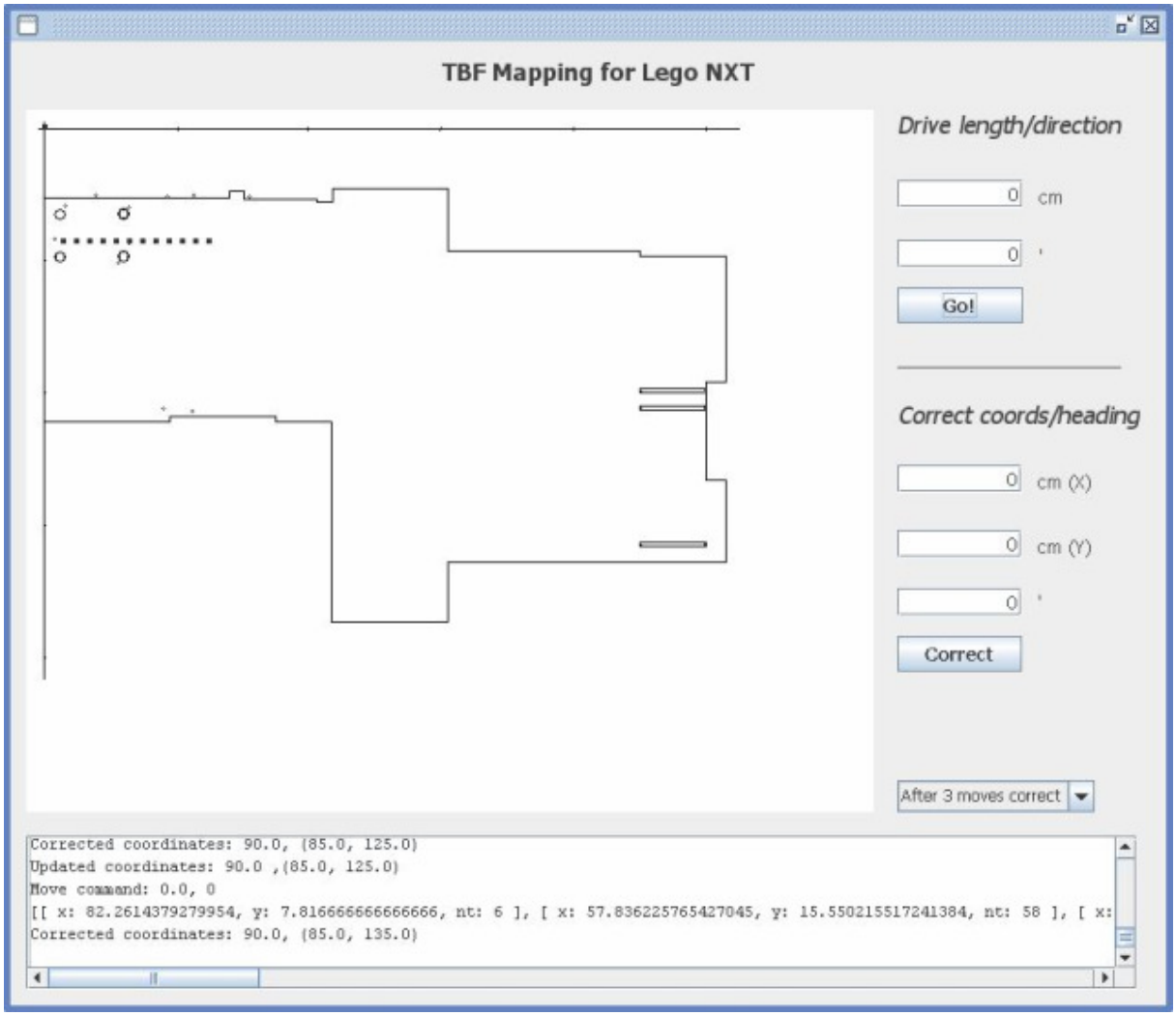
Röviden összefoglalva a program lényegi felépítése: implementáltam a TBF algoritmust, valamint kiterjesztettem a robotmozgatás iCommand által nyújtott formáit egy speciális metódussal, ami a szenzor előzőekben tárgyalt mérési módszerét valósítja meg. Ezt az MVC tervezési mintának megfelelően kiegészítettem még egy irányító osztállyal, valamint a megjelenítésért felelős grafikus interfésszel. A program egyszerűsített osztály diagrammját a 12. ábra mutatja be.



12. ábra A program egyszerűsített osztály diagrammja

Tulajdonképpen nem a hagyományos MVC modell szerint dolgoztam, ugyanis mivel két modellt is kezelni kell, ezért a View számára a Control biztosítja az összetett állapotinformációt. A két modell abból adódik, hogy a robot helyzetével és paramétereivel kapcsolatos információkat az Acessory, a térképezéssel kapcsolatos információkat pedig a MapppingWithTBF osztály tartalmazza.

Az eredmények látványos megjelenítése és az egyszerű kezelhetőség érdekében egy grafikus felhasználói felületet is létrehoztam. Ezen keresztül lehet mozgási parancsokat kiadni, valamint korrigálni a jármű helyzetét, és ezen jelenítődnek meg a futási eredmények és üzenetek. Ezt a következő ábrán szemlélhetjük.



13. ábra A program grafikus felhasználói felülete

A robot mozgatásához meg kell adni egy utazási távolságot, valamint egy fordulási szöget. A robot ekkor először a megadott szöggel elfordul (előjeltől függően a megfelelő irányba), majd a megadott távolságot megteszi. (Szintén előjeltől függően előre, vagy hátrafelé.) A (0,0) mozgási parancs kiadása esetén a robot csak a mérési fázist hajtja végre.

A korrekciós mezőkben megadható a robot aktuális, kézzel mért valós pozíciójának adatai. Erre azért van szükség, mert bár a robot minden lépése után újraszámolja a pozícióját, de ez hibákkal terhelt, ami hosszú távon jelentős eltéréseket okozhat. (Emlékezzünk arra, hogy a jelenlegi program célja a TBF algoritmus implementálása, így *nem* nyújtja a SLAM teljes funkcionalitását; azaz a mérési és mozgási hibákból adódó eltéréseket képtelen kezelni.

Ezért szükséges a pozíció manuális karbantartása.) Megadható még az is, hogy a program milyen időközönként kérje a pozíció kézi korrigálását.

Az eredményeket kétféleképpen jelenítettem meg; egyrészt a nyers koordináták és egyéb információk elérhetők az alsó részen található szövegdobozban; másrészt a jobb szemléltetés érdekében a tájékozódási pontok grafikusan is megjelennek egy 2D térképen. Ezen a térképen kis körök jelölik a tájékozási pontok, míg fekete négyzetek a robot mérési pontjait.

Tapasztalatok és megjegyzések

A hardverről határozottan pozitív véleményem alakult ki. Az előnyeit a megfelelő részben már elsoroltam; ezt nem teszem meg még egyszer. Jó használhatósága ellenére azonban nem szabad elfelejteni, hogy egy *nem professzionális* hardver, így bizonyos feladatok megoldására alkalmatlan. Nekem is szembesülnöm kellett azzal, hogy bár az ultrahangos szenzor maximális észlelési távolsága 254 cm (ez más hardverekkel összehasonlítva nem sok), az algoritmusban 160 cm-es korlátot kellett alkalmaznom, mert nagyobb távolságokon már megbízhatatlan mérési eredményeket adott. Emellett problémái akadtak a mérő nyalábra nem merőleges felületekkel. Mindezek eredményeképpen sok fals mérési adatot generált. (A fenti módosítás után sem tűntek el teljesen a hibás objektum felismerések, de számuk kezelhető szintre csökkent.) További következménye ennek az lett, hogy viszonylag sok mérést kell végezni a megfelelő eredmény eléréséhez.

Negatívum sajnos, hogy a dolgozat írása idején a programozási környezetek meglehetősen félkész állapotúak voltak. Ez nyilván idővel változni fog, de a jelenlegi gondok érzékeltetésére lentebb elmondom az iCommand API néhány lényeges hibáját.

Sok vesződséggel járt a Bluetooth használata is. Feltehetően szintén az iCommand API miatt is (a lefagyó program nem engedte el a kommunikációs portot, így azt nem lehet újra használatba venni) eleinte a programom két futtatása közötti idő elérte a 30 percet (!) is. A problémán sokat javított egy Bluetooth kommunikációs célszoftver (IVT BlueSoleil) alkalmazása. Ennek, és a lefagyások számának mérséklődésének köszönhetően ez az érték a tesztelés során 5 percre redukálódott.

Rengeteg problémám adódott az iCommand API-val. Az általam használt verzió a 0.5-ös volt. Az osztályhierarchia felépítése véleményem szerint nem rossz, de az implementáció ezernyi hibával rendelkezik.

Tesztelnem nem sikerült, hogy ez okozza-e a rejtélyes fagyásokat, de a kód tanulmányozása során azt vettem észre, hogy az NXT Bluetooth kommunikációs protokolljának hibajelzéseit az API egyszerűen figyelmen kívül hagyja. Nincs semmiféle kivételkezelés, nem bővíthető az API. (Jobban mondva csak a forráskód átírásával, osztály kiterjesztését sok helyen megakadályozzák a privát láthatóságú tagok.)

Az implementációs rész bevezetésében említett példa az alapváltozatban lehetetlenné teszi a robot fordulását; ráadásul a robot kívánt fordulási szögével kerül újraszámolásra a pozíció a mozgás után, függetlenül attól ténylegesen mennyit is fordult a robot. Szintén sajnálatos, hogy a motorok kis fordulatszámon rángatnak; pedig a Labview®-ban ilyet nem tapasztaltam. A javítható programozási hibák mellett kifejezetten idegesítő az, hogy a robot mozgása után az NXT firmware firssítést kérő folyamatos csipogó hangot ad ki. Próbáltam ennek utánanézni a hivatalos weblapon és fórumon, de nem rá találtam megoldást. (Sajnos, ugyanis meglehetősen zavaró.)

Szerencsére a forráskód hozzáférhető a Lejos ([26]) honlapján, így lehetőségem nyílt bizonyos hibák kijavítására. Igazából nem is a jelenleg elérhető 0.5 változatot javítgattam, hanem már én is egy másik felhasználó által frissített verziót kezdtem használni. Azt kell mondanom, hogy a jelenlegi Lejos iCommand API meglehetősen „bugos”, akár egy teljes újraírást igényel még.

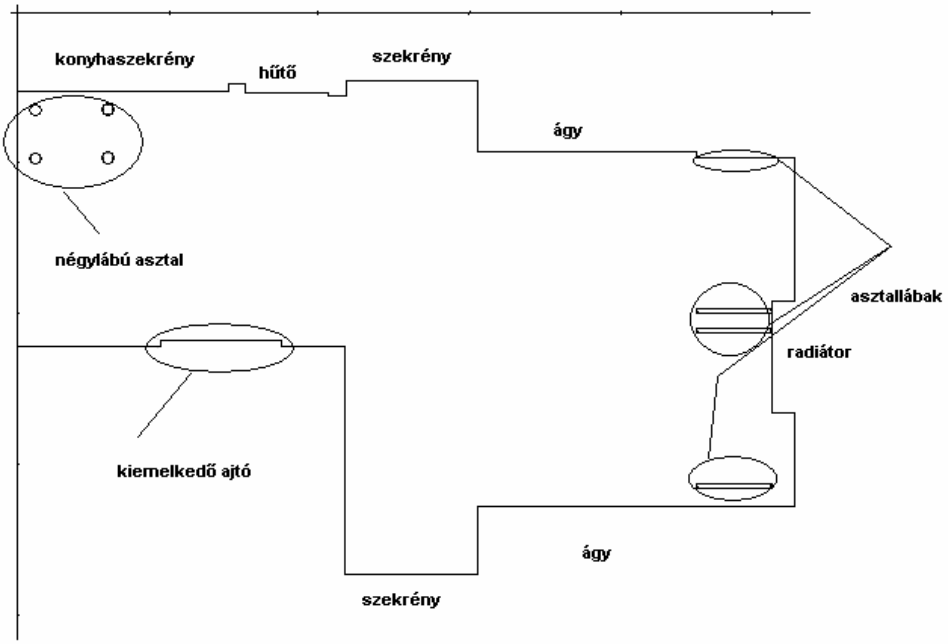
Az eredmények értékelése előtt néhány dolgot el kell mondanom az algoritmusról. Mint korábban említettem, a fals mérési eredmények következtében időnként nem létező tájékozódási pontok kerülnek a térképre. Ezek kiszűrésére a túl gyengén támogatott pontokat törli az algoritmus a térképről. Ez időnként jó pontok eltűnését is okozta. A másik tudnivaló az, hogy az egyértelműség biztosítására az egymáshoz közeli térkép pontok összevonásra kerülnek. (Súlyozott átlag számítással.) Ez időnként jó pozícióban lévő pontok „elcsúszását” okozza. Ismert probléma továbbá a mérési területként használt szoba alaprajzának torzítása. Sajnos az 1 pixel 1 cm méretarány miatt így az összeadódott hibák látványosak lehetnek. Mivel a TBF algoritmus *nem* térképezési módszer, ezért nem kerestem jobb megoldásokat; az algoritmus működése így is jól megfigyelhető.

A következő fejezetben áttérek a szobában végzett mérések elemzésére és értékelésére.

Mérések

A terep

A térképező működését a kollégiumi szobámban teszteltem. A grafikus megjelenítéshez meg kellett rajzolnom a szoba alaprajzát. Ennek elkészítéséhez kézzel bemértem minden jelentős objektum pozícióját. Sajnos ahogy már említettem a felmérési munkába egy kis hiba csúszott, de ez a tesztelésre használt útvonalon csak egy helyen játszott szerepet. A következő ábra ezt az alaprajzot mutatja.



14. ábra A Szoba alaprajza a bemért objektumokkal

Két méréssorozatot végeztem. Ezek során a robot érzékelhette a szoba összes jelentős tereptárgyát. A robotot kézzel mozgattam át egyik mérési pontból a másikba a pontosság biztosítása érdekében. (Korábban is volt szó arról, hogy az API pozíció újraszámolása rövidtávon is sok hibát okoz. A jobb eredmény érdekében ezért választottam a kézi áthelyezést, így az algoritmus tényleges működését jobban mutatják a tesztek.) A következő két részben az egyes méréssorozatokat írom le, lépésenként.

Az első tesztmenet

Ebben a menetben a robot a bejárati ajtótól indult, és a szemközti ágyig jutott el. Az első melléklet mutatja a térkép állását a 8. lépés után. Látható, hogy az asztallábak felismerése szinte tökéletes; köszönhetően a kis érzékelési távolságnak, sok mérésnek, és ez megvilágítja az algoritmus azon tulajdonságát is, hogy olyan objektumokra működik a legjobban, ami mellett a robot elhalad, és közben többször is érzékeli. Az alaprajzon nem jelöltem be a konyhaszekrény ajtóréseit, pedig a robot ezeket is képes volt felismerni.

A második melléklet a 12. lépés utáni állapotot ábrázolja. A térképről törlése került az asztallábak egyike. Sajnos nem volt látható sok lépésben, így a támogatottsága alacsony maradt. Szerencsére szintén eltűnt a térképről több hibás észlelés is. Nagyjából megfelelő, bár kicsit bizonytalan az ajtó felismerése, és szinte tökéletes a hűtő sarkának megtalálása.

A harmadik mellékleten látható a 21. lépés utáni térkép. Az ajtó felismerése még mindig bizonytalan, a hűtő bal oldali éle kicsit arrébb csúszott. Jól pozícionált ellenben a hűtő melletti kiemelkedés. Ismét megfigyelhetőek több helyen fals mérési eredményeknek köszönhető objektumok. (Pl. a szoba közepén)

A negyedik mellékleten az utolsó, 29. mérés utáni állapot jelenik meg. A hűtő melletti kiemelkedés a korábbi állapothoz képest kissé elcsúszott. Felismerésre került a szekrény két sarka, és az ajtónyílás is. Az ágy sarkának pozíciója még nem tökéletes.

A második tesztmenet

A második menetben az ágyak között épített térképet a robot. A hatodik melléklet a 7. lépés utáni állapotot mutatja be. A robot sikeresen felismerte az ágy távolabbi sarkát. Az ágy közelebbi sarkának jó, de elcsúsztatott felismerése ismeretlen hiba oka. A radiátor sarkának közelítése itt még gyenge, valamint a számítógépem kábelrengetege hibás objektum érzékeléseket generált. Megfigyelhető 2 fals tárgy is; ezeket az adott pozícióból nem is láthatta a robot. (Az asztalláb mögött helyezkednek el.)

A hetedik melléklet mutatja a 16. mérés utáni helyzetet. A korábban jól felismert ágy sarok alacsony támogatottság miatt eltávolításra került. (Távolságkorláton hamar „túlesett”.) Figyelemre méltóan pontos 2 asztalláb felismerése is. Szintén elég jó az asztal, és a radiátor

közötti sarok felismerése is. Magát a radiátor több pontként azonosította nagy kiterjedése miatt. A szemközti ágynál még nagy a bizonytalanság.

A nyolcadik melléklet az utolsó lépés utáni térképet ábrázolja. Ekkor felfedezésre került nagy biztonsággal szemközti ágy két sarka, valamint a radiátor közelebbi sarka. Elcsúszott viszont az egyik asztalláb, és még mindig nagy a bizonytalanság a robottal szembe, az ágy körül. Megfigyelhető két „falon túli" objektum is; ismét a fals mérési eredmények számlájára írható megjelenésük.

Eredmények értékelése

Alapvetően mind a hardver, mind az algoritmus jól vizsgázott. Bár az egyszerű térképkészítés miatt néhány jó pont elveszett, illetve elcsúszott, és a hardver limit miatt kisebb volt az érzékelési távolság, mégis az algoritmus sokszor gyakorlatilag teljesen pontos becsléseket adott. Sajnos az ultrahangos szenzor (szonár) hibáival együtt kell élnie; megoldást kell találni a hibás objektumok biztonságos kiszűrésére, illetve ki kell tapasztalni, hogy a felületek anyaga és elhelyezkedése, formája hogyan hat azok érzékelhetőségére.

Későbbiekben a rendszer bővíthető a SLAM irányába, az alapok adottak hozzá. Ehhez azonban mindenképen növelni kell a robot irányításának, illetve pozíció újraszámolásának pontosságát.

Összegzés

Napjainkban autonóm robotok még ritkán fordulnak elő közvetlen környezetünkben. Néhány multinacionális cég már készít háztartási célú robotokat; léteznek kezdetleges önjáró fűnyírók és porszívók; vagy gondoljuk csak a játékos Aibo robotkutyára. Azonban a technika még csak gyermekcipőben jár. A tengerentúlon már komoly fejlesztések folynak; természetesen a hadsereg támogatásával. A DARPA Grand Challenge verseny keretében 2006-ban egy sivatagi pályán már több résztvevő célba ért; teljesen autonóm, emberi segítség nélkül közlekedő autókkal! A fejlődés várhatóan tovább gyorsul a közeljövőben is, és intelligens segítőtársakra találhatunk majd a fejlettebb robotok személyében.

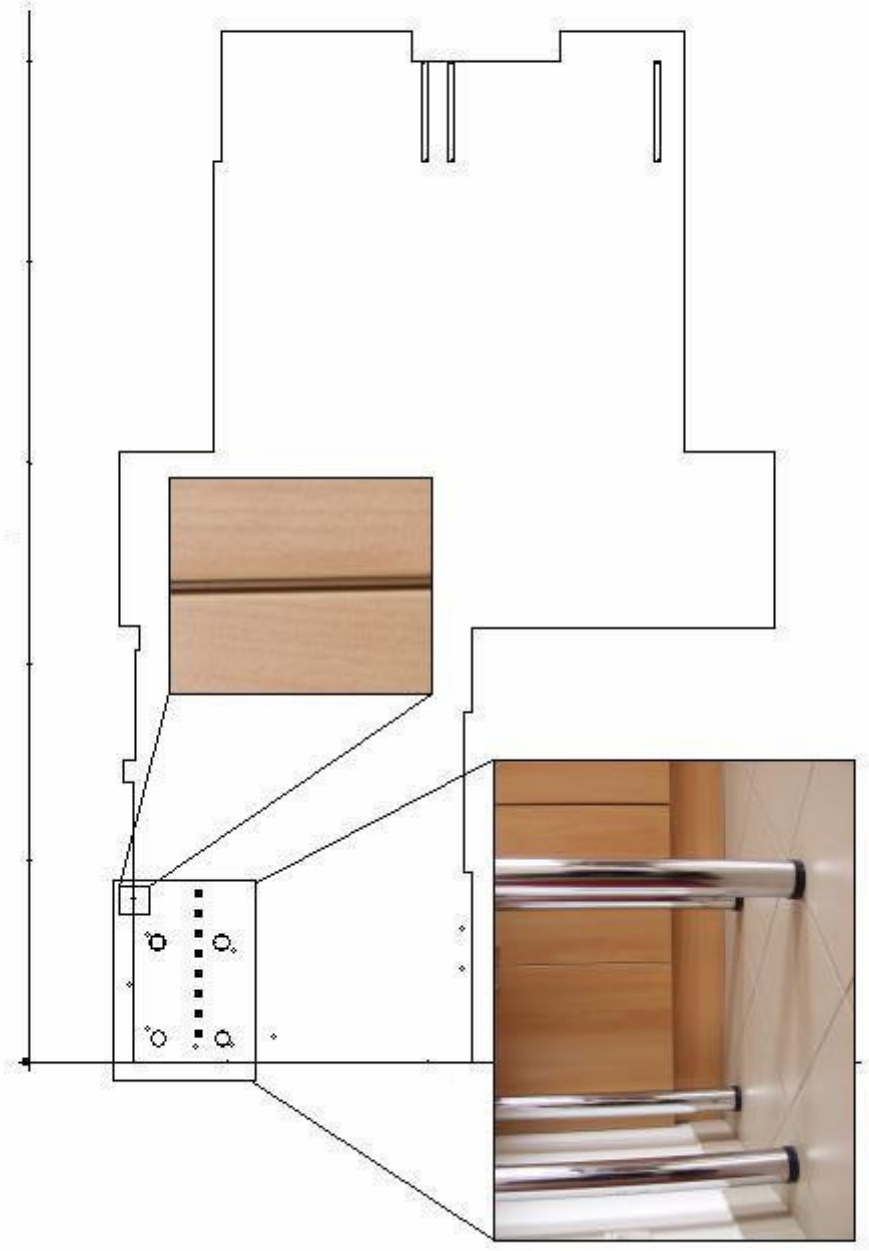
Minden autonóm robot kulcsfontosságú eleme a navigáció, és ezzel szoros összefüggésben a térképkészítés. Dolgozatomban ezen kérdésre igyekeztem felderíteni a jelenleg elérhető válaszokat.

A probléma fontossága miatt jelentős erőfeszítéseket tettek kutatók a megoldás megtalálására; ez meg is hozta az eredményét, és a SLAM ma megoldottnak tekinthető. Eredetileg egy teljes SLAM implementációt szerettem volna megvalósítani; de sajnos ezt idő hiányában nem tudtam elkészítenem. Viszont egy igen fontos részfeladatra, a tájékozódási pontok felismerésére implementáltam a TBF algoritmust egy Lego NXT hardver alkalmazásával. Teszteket is végeztem az algoritmus vizsgálatára, amin az jól szerepelt, és bizonyította életképességét.

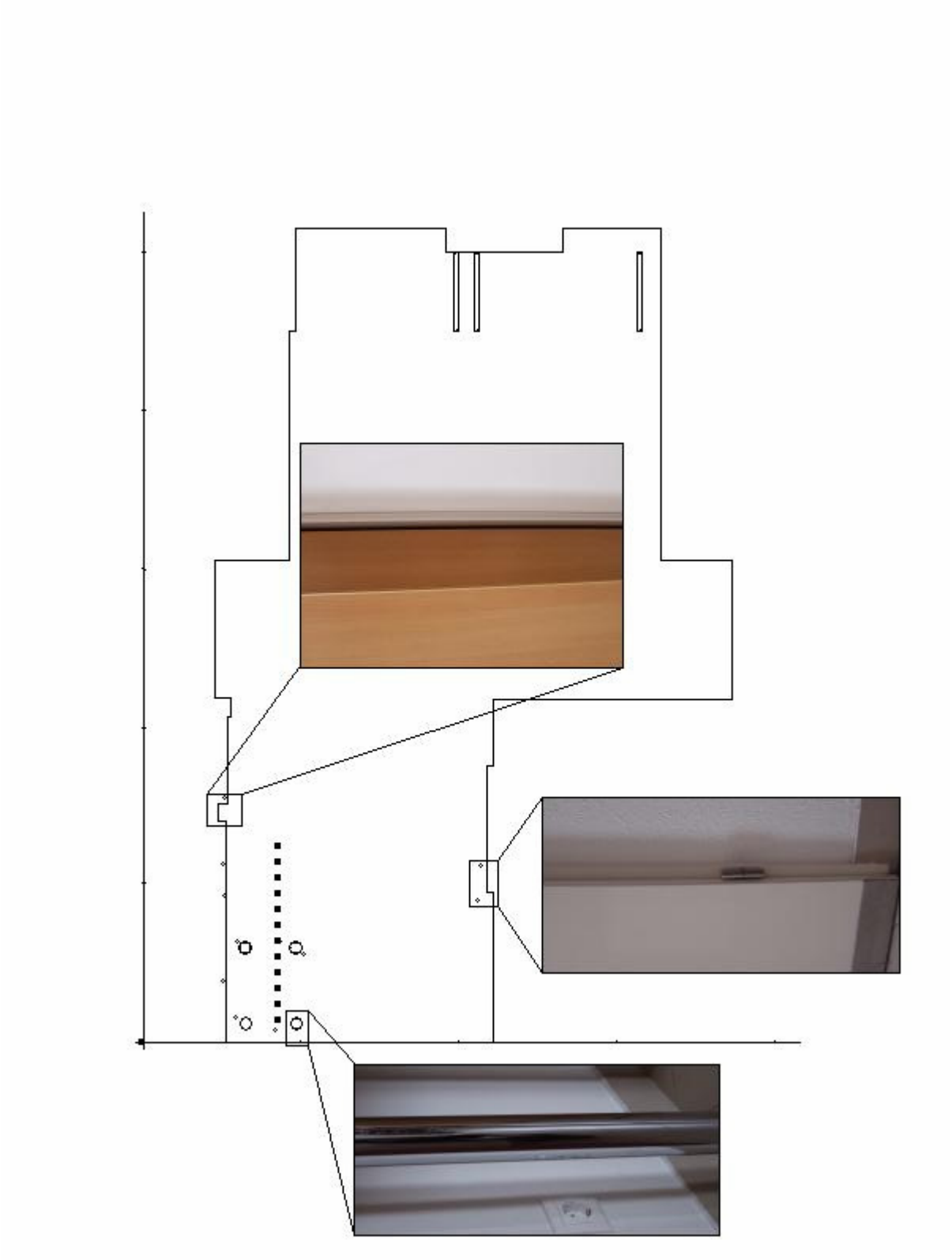
Az elmélet mellett nagy hangsúlyt kapott a hardveres implementáció is. Arra a kérdésre is kerestem a választ, hogy a hazánkban elérhető gyakorlatilag egyetlen robotika készlet mire képes, érdemes-e használni? A válasz egyértelműen igen. Bár nem professzionális, de sok, jellegzetes robotikai feladat megoldására igenis alkalmazható.

Összességében a robotika témaköre nagyon izgalmas, és meglehetősen széles; több informatikai és nem informatikai tudomány területet is összeköt. A hardver eszközök technikai paraméterei, a számítógépek egyre növekvő teljesítménye már ma is lehetővé teszi bonyolult, összetett algoritmusok valós idejű futtatását; a hardver korlátai egyre kevésbé jelentenek tényleges korlátozó tényezőt a programozók számára. Véleményem szerint érdemes a területet kutatni és tanulmányozni, mert adottak a lehetőségek a sikerre és várhatóan a jövőben egyre nő majd mind a terület iránti érdeklődés, mind a robotok iránti igény is.

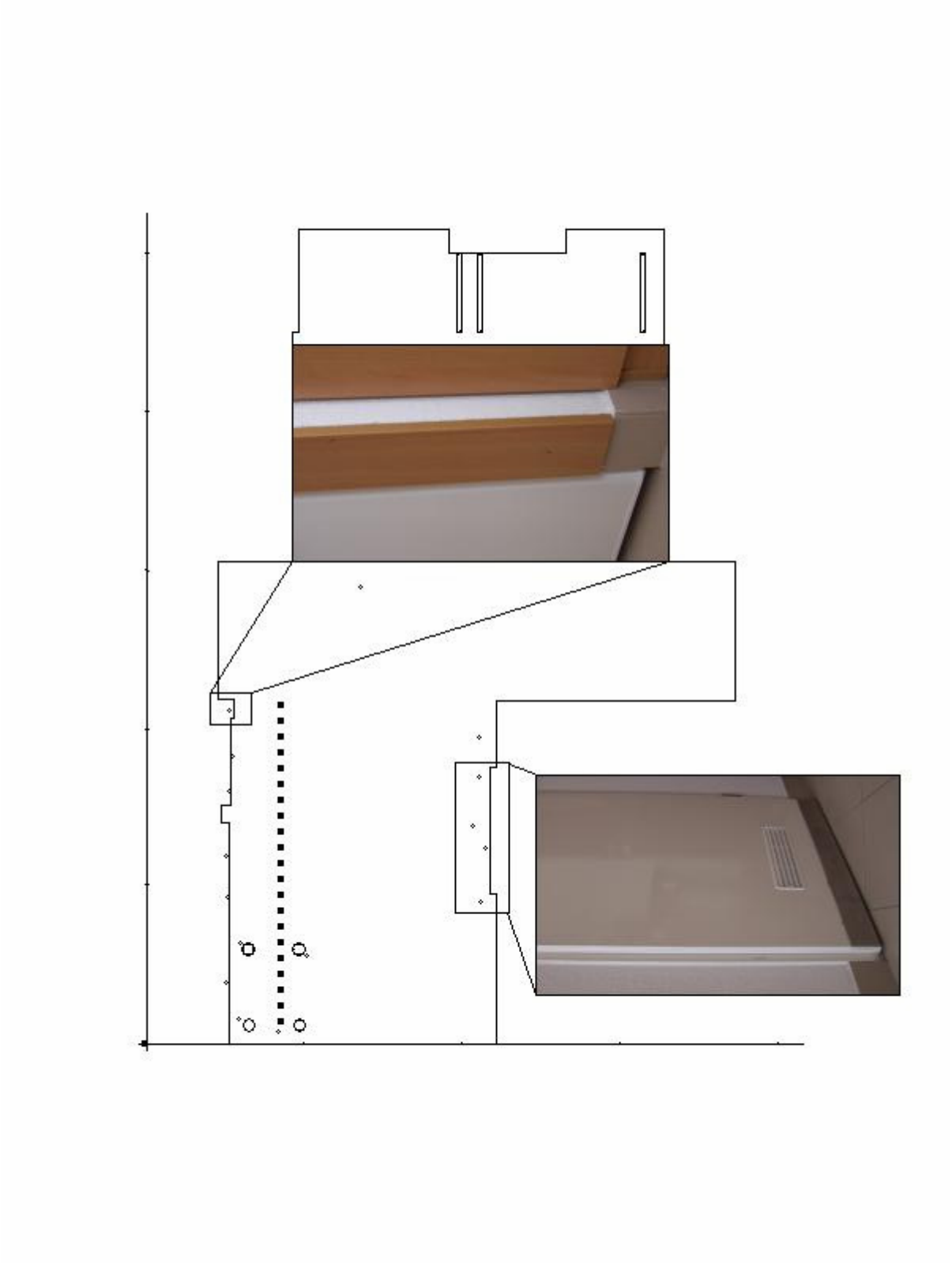
Függelék



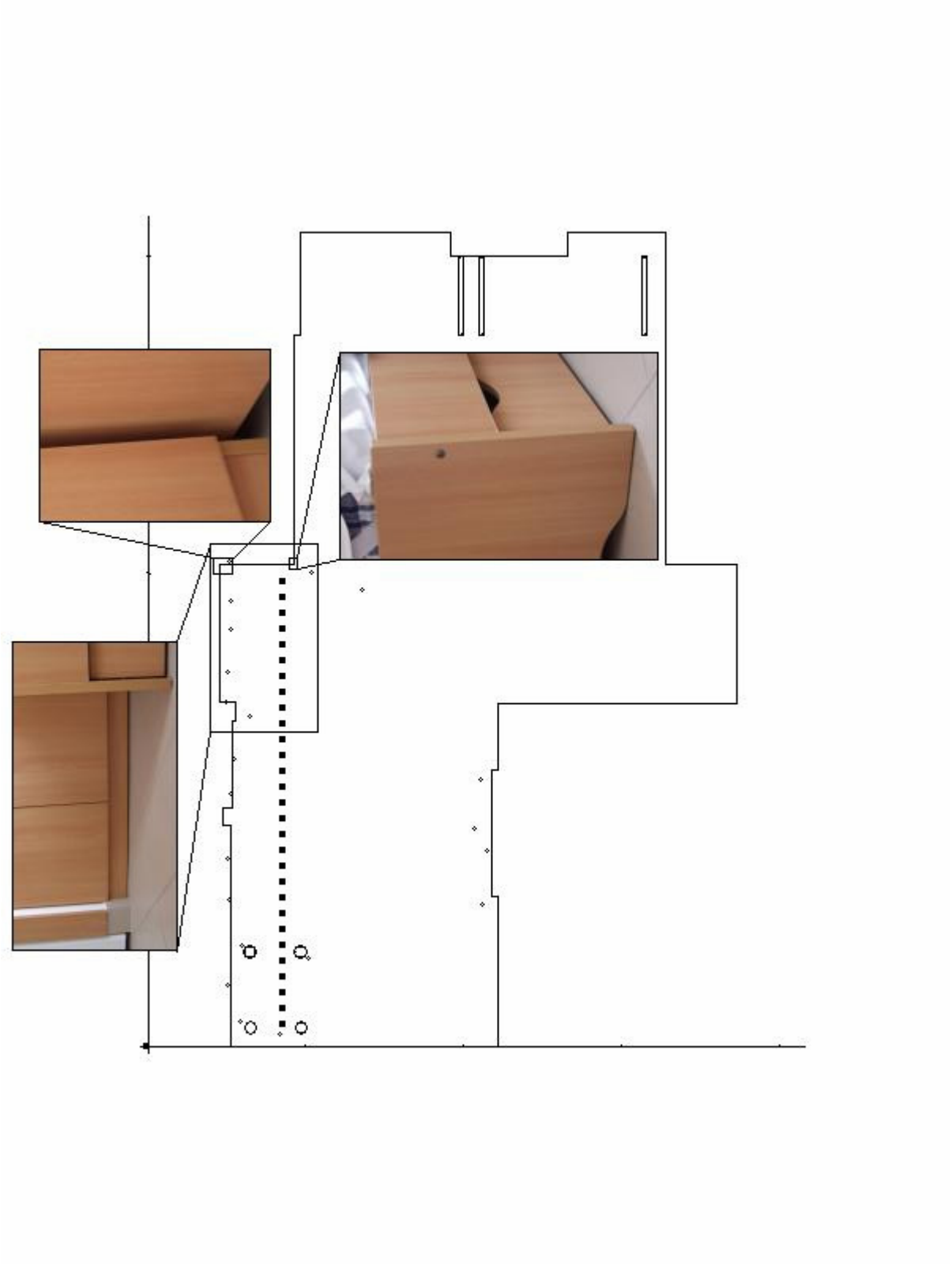
1. melléklet Az első menet a 8. mérés után



2. melléklet Az első menet a 12. mérés után

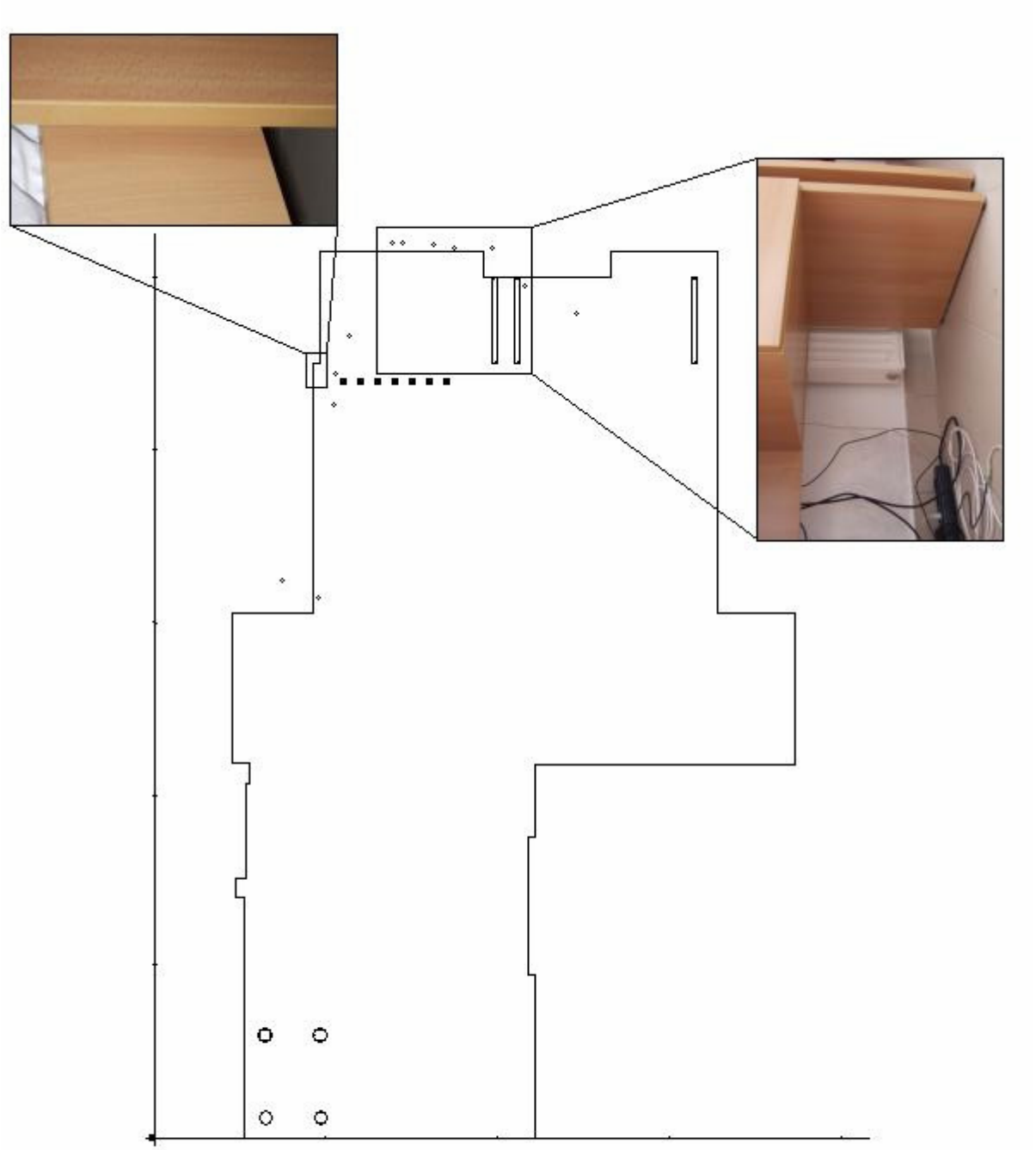


3. melléklet Az első menet a 21. mérés után

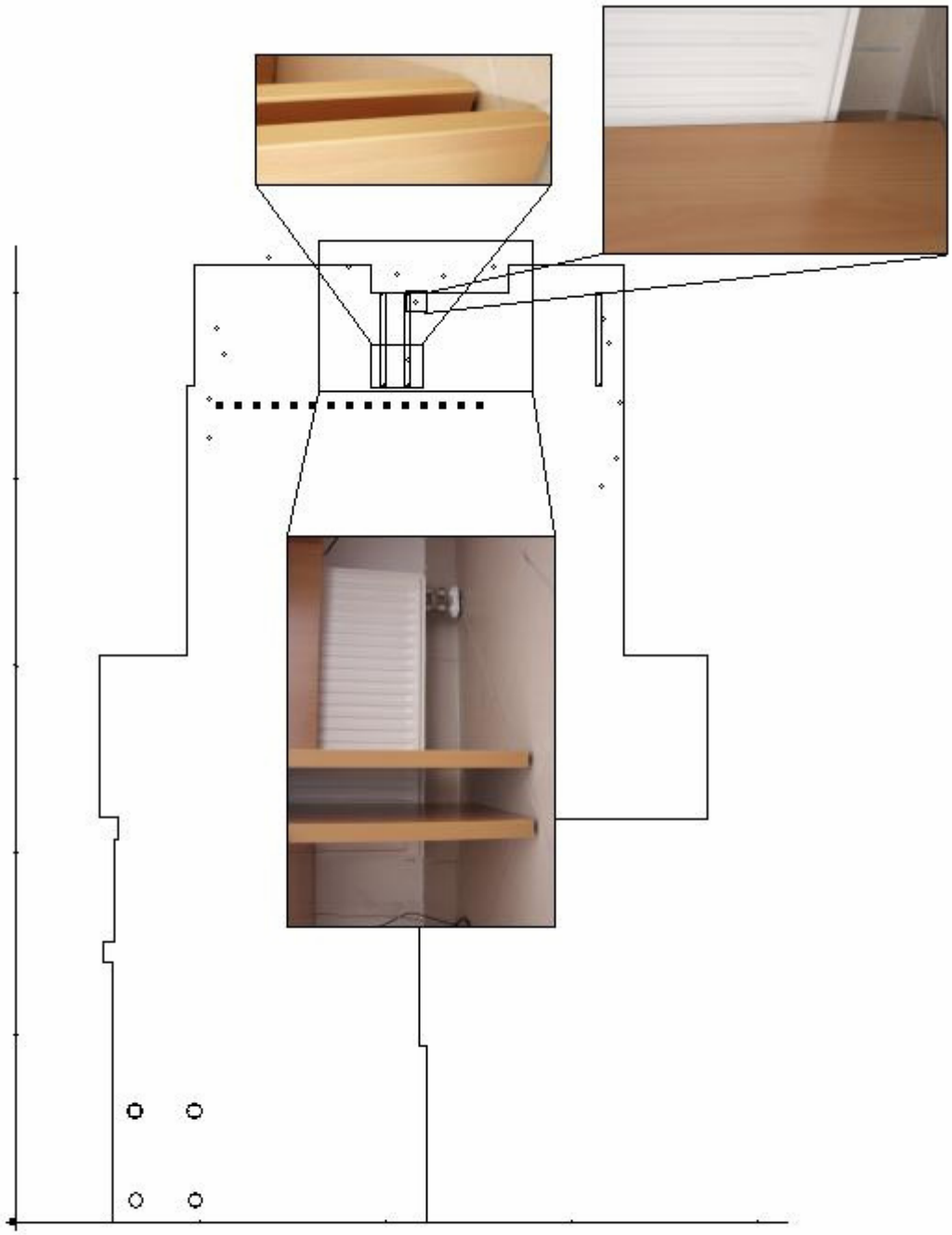


4. melléklet Az első menet a 29. mérés után

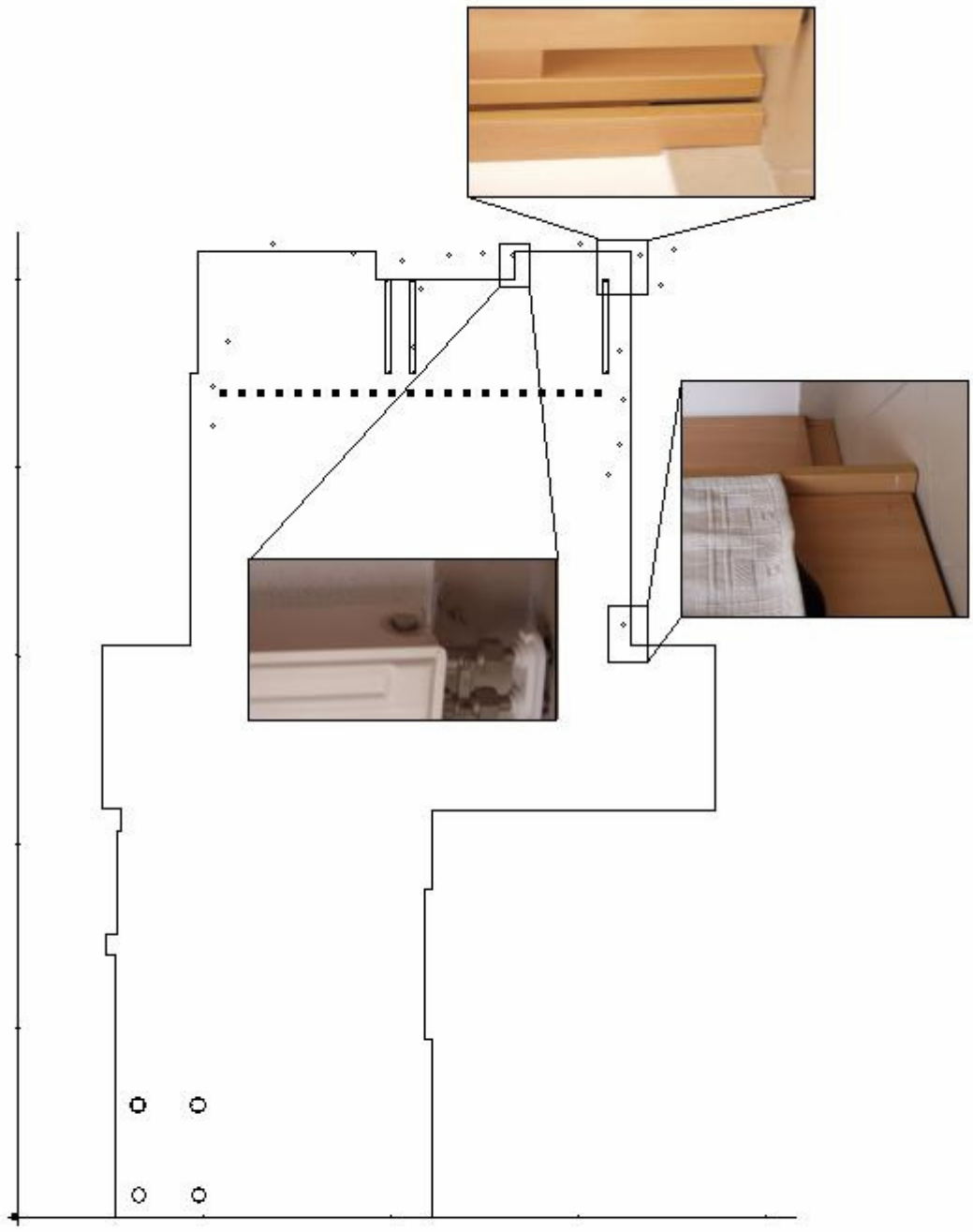
5. melléklet Az első menet teljes útvonala



6. melléklet A második menet a 7. mérés után



7. melléklet A második a menet 16. mérés után



8. melléklet A második menet 22. mérés után

9. melléklet A második menet teljes útvonala

Irodalomjegyzék

[1] Dissanayaka, M., Newman, P., Clark, S., Durrant-Whyte, H., and Csorba, M. : A solution to the simultaneous localisation and map building problem.
IEEE Robotics and Automation 17, 3 (June 2001), 229-241.

[2] Hugh Durrant-Whyte , Tim Bailey :
Simultaneous Localization and Mapping: Part I
IEEE Robotics and Automation *(*June *2006), 99-108.*

[3] Hugh Durrant-Whyte , Tim Bailey :
Simultaneous Localization and Mapping: Part II
IEEE Robotics and Automation *(*September *2006), 108-117.*

[4] José A. Castellanos, José Neira, Juan D. Tardós:
Map Building and SLAM Algorithm

[5] Castellanos, J. A., Montiel, J. M. M., Neira, J., and Tardós, J. D. :
The SPmap: A probabilistic framework for simultaneous localization and map building.
IEEE Robotics and Automation 15, 5 (October 1999), 948-953.

[6] Tardós, J. D., Neira, J., Newman, P. M., and Leonard, J. J. :
Robust mapping and localization in indoor environments using sonar data.
The International Journal of Robotics Research 21, 4 (2002), 311-330.

[7] Montemerlo, M., Thrun, S., Koller, D., and Wegbreit, B. :
Fastslam: A factored solution to the simultaneous localization and mapping problem.
In Proceedings of the AAAI National Conference on Artificial Intelligence,
Edmonton, Canada, 2002. AAAI.

[8] Michael Montemerlo, Sebastian Thrun, Daphne Koller, Ben Wegbrei:
FastSLAM 2.0: An Improved Particle Filtering Algorithm for Simultaneous
Localization and Mapping that Provably Converges
IJCAI 2003

[9] Leonard, J., and Newman, P. :
Consistent, convergent, and constant-time slam.
IJCAI (Acapulco, August 2003)

[10] S. Thrun, D. Koller, Z. Ghahmarani, H. Durrant-Whyte, A. Ng. :
Simultaneous localization and mapping with sparse extended information filters

Carnegie Mellon University , Tech. Rep., 2002

[11] M.A. Paskin:

Thin junction tree filters for simultaneous localization and mapping

Comput. Sci. Div. Univ. California, Tech. Rep. , 2002

[12] Elfes A. :

Sonar-Based Real-World Mapping and Navigation.

IEEE Journal of Robotics and Automation, RA-3(3), 1987 , 249–265.

[13] Gutmann, S., and Schlegel, C. :

Amos: Comparison of scan-matching approaches for self-localization in indoor environments.

In 1st Euromicro Conf on Adv. Mobile Robotics (1996), IEEE.

[14] Hugh Durrant-Whyte:

Localisation, Mapping and the Simultaneous Localisation and Mapping (SLAM) Problem

SLAM Summer School 2002

[15] Raja Chatila :

SLAM: Representation issues

SLAM Summer School 2002

[16] Wijk, O., and Christensen, H. :

Triangulation based fusion of sonar data with application in robot pose tracking

IEEE Transaction on Robotics and Automation 16, 6 (Dec. 2000), 740-752.

[17] Wijk, O., and Christensen, H. , P. Jensfelt :

Triangulation based fusion of ultrasonic sensor data

ICRA 1998, Leuven, Belgium

[18] Wijk, O., and Christensen, H. :

Extraction of Natural Landmarks and Localization using Sonars

[19] http://cache.lego.com/upload/contentTemplating/MindstormsOverview/otherfiles/2057/LEGO%20MINDSTORMS%20NXT%20Hardware%20Developer%20Kit(3)_7A0CF630-CCE5-4AAF-91FA-D1E7C911817C.zip

[20] http://cache.lego.com/upload/contentTemplating/MindstormsOverview/otherfiles/2057/LEGO%20MINDSTORMS%20NXT%20Bluetooth%20Developer%20Kit_58CE458E-5292-4CB0-93D2-4BEC821C13C2.zip

[21] http://cache.lego.com/upload/contentTemplating/MindstormsOverview/otherfiles/2057/fantomSDK1.0.2f0(1)_1935F2F2-3052-406C-8F39-AE2C80C58BAF.zip

[22] http://cache.lego.com/upload/contentTemplating/MindstormsOverview/otherfiles/2057/LEGO%20MINDSTORMS%20NXT%20Executable%20File%20Specification(1)_9143FED4-3FF8-40B1-A06F-78B530347A59.zip

[23] http://bricxcc.sourceforge.net/nbc/

[24] http://bricxcc.sourceforge.net/nbc/doc/NBCManual.zip

[25] http://bricxcc.sourceforge.net/nbc/nxcdoc/NXC_Guide.pdf

[26] http://lejos.sourceforge.net/index.php

[27] http://lejos.sourceforge.net/p_technologies/nxt/icommand/api/index.html

[28] http://lejos.sourceforge.net/p_technologies/nxt/nxj/api/index.html

[29] http://zone.ni.com/devzone/cda/tut/p/id/4435

[30] ftp://ftp.ni.com/evaluation/mindstorms/How_To_Create_NXT_Blocks_with_NI_LabVIEW.pdf

[31] ftp://ftp.ni.com/evaluation/mindstorms/LabVIEW_for_NXT_Advanced_Programming_Guide.pdf