

BABEŞ-BOLYAI UNIVERSITY CLUJ-NAPOCA
FACULTY OF MATHEMATICS AND INFORMATICS
SPECIALIZATION: COMPUTER SCIENCE

License Thesis

Control and balancing – applied to Lego robots

Abstract

The thesis' goal is to accomplish the remote control of a MINDSTORMS EV3 two wheeled self balancing robot. The control framework is based on network communication between a mobile application and the robot itself. To achieve this goal there are two main components that need to be designed and implemented. Firstly the mobile application that needs to make the communication with the robot, and secondly an algorithm which makes the balancing of the robot possible.

The application and the algorithm which handles the balancing were written in Java. The EV3 control unit is responsible for navigating the robot with the help of the leJOS Linux based operating system. The robot remains in a balanced position by using a PID controller and a one-axis gyroscope sensor. For it to remain in a stable state, its velocity, tilt angle and angular velocity need to approximate zero. Also if it were to remain in a standing balanced position without translation its position needs to approximate zero too. The PID's input argument is the error variable which is the difference between the expected value and the current value of the sensor measurements. In our case we need to minimize this error in order to not lose balance. In conclusion the error is made up of four components: the robot's tilt angle, angular velocity, velocity and position. By properly modifying these components we can manipulate the robot's movement direction and velocity. The mobile application written for the Android platform, connects to the robot through the network and sends eligible commands to it. To accomplish and automate the connection between the two components we used the Cling UPnP library, which relies on the SSDP protocol for device and service discovery.

This work is the result of my own activity. I have neither given nor received unauthorized assistance on this work.

JULY 2016

MÁRTON ZETE-ÖRS

ADVISOR:
ASSIST PROF. DR. HUNOR JAKAB

BABEŞ-BOLYAI UNIVERSITY CLUJ-NAPOCA
FACULTY OF MATHEMATICS AND INFORMATICS
SPECIALIZATION: COMPUTER SCIENCE

License Thesis

**Control and balancing – applied to
Lego robots**



SCIENTIFIC SUPERVISOR:

ASSIST PROF. DR. HUNOR JAKAB

STUDENT:

MÁRTON ZETE-ÖRS

JULY 2016

UNIVERSITATEA BABEŞ-BOLYAI, CLUJ-NAPOCA
FACULTATEA DE MATEMATICĂ ȘI INFORMATICĂ
SPECIALIZAREA INFORMATICĂ

Lucrare de licență

Control și echilibrare – aplicat la roboți Lego



CONDUCĂTOR ȘTIINȚIFIC:
LECTOR DR. HUNOR JAKAB

ABSOLVENT:
MÁRTON ZETE-ÖRS

IULIE 2016

BABEŞ-BOLYAI TUDOMÁNYEGYETEM KOLOZSVÁR
MATEMATIKA ÉS INFORMATIKA KAR
INFORMATIKA SZAK

Államvizsga-dolgozat

**Kontroll és egyensúlyozás –
alkalmazás Lego robotnál**



TÉMAVEZETŐ:

DR. JAKAB HUNOR,
EGYETEMI ADJUNKTUS

SZERZŐ:

MÁRTON ZETE-ÖRS

2016 JÚLIUS

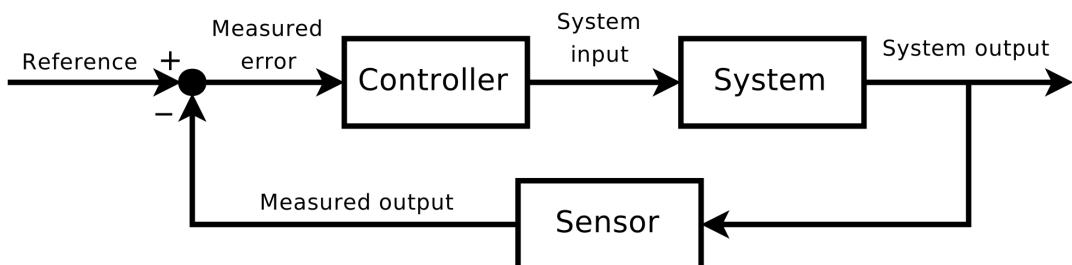
Tartalomjegyzék

| | |
|--|-----------|
| 1. Bevezető | 3 |
| 2. LEGO | 6 |
| 2.1. LEGO megalakulása | 6 |
| 2.2. LEGO MINDSTORMS | 6 |
| 2.2.1. RXC | 6 |
| 2.2.2. NXT | 7 |
| 2.2.3. EV3 | 7 |
| 2.3. Ev3 motorok | 9 |
| 2.3.1. Nagy motor | 9 |
| 2.3.2. Közepes motor | 9 |
| 2.4. Ev3 szenzorok | 9 |
| 2.4.1. Színszenzor | 10 |
| 2.4.2. Giroszkóp szenzor | 10 |
| 2.4.3. Nyomásérzékelő | 10 |
| 2.4.4. Ultrahang szenzor | 11 |
| 2.4.5. Infravörös szenzor | 12 |
| 2.4.6. Az egyensúlyozó robot felépítése | 12 |
| 3. Egyensúlyozás | 14 |
| 3.1. PID szabályzó algoritmus | 14 |
| 4. Megvalósítás | 16 |
| 4.1. EV3 programozása | 16 |
| 4.2. Androidos alkalmazás és kommunikáció | 20 |
| 4.2.1. Google Protocol Buffers | 22 |
| 4.2.2. Robot oldali kommunikáció | 23 |
| 4.2.3. Automatikus kapcsolódás | 23 |
| 4.3. Egyensúlyozás problémái | 24 |
| 4.3.1. Irányítás megvalósítása és egyensúly megtartása | 27 |
| 5. Következtetések és továbbfejlesztési lehetőségek | 31 |

1. fejezet

Bevezető

A szakdolgozat témája robotok egyensúlyozását megvalósító rendszerek tanulmányozása. Ezen rendszereket nevezhetjük zárt vagy nyílt ciklusosnak. Nyílt ciklusos rendszer esetén nincs visszacsatolás, ezért nem megfelelő olyan problémák megoldására, amelyeknél ellenőriznünk kell a rendszer kimenetének eredményét, tehát nem szükséges a visszacsatolás. Egyszerű problémák megoldása esetén használatos. A zárt ciklusos rendszerek visszacsatolással működnek, irányítható a rendszer. Visszacsatolásos működés(1.1 ábra) annyit tesz, hogy a kimenet hatására egy visszajelzést kapunk, amely tulajdonképpen a rendszer bemenete lesz, így a visszajelzés szabályozza a kimenetet és ezáltal komplex feladatok megvalósítására alkalmas.



1.1. ábra. Zárt ciklusos rendszer működési elve https://en.wikipedia.org/wiki/Control_theory

Zárt ciklusos rendszerek esetén szabályzó algoritmusról beszélünk, amelyek használata számos technológiában jelen van. Ilyenek közé sorolható a Swagway¹ illetve a hasonló szerkezetű és működésű Segway².

A dolgozat során olyan projekt kerül bemutatásra, amely a LEGO MINDSTORMS EV3[5] készletből épített kétkerekű egyensúlyozó robot irányítását teszi lehetővé hálózaton keresztül, telefonos alkalmazás segítségével.

A kétkerekű robot a Gyro Boy³ modell alapján készült el. Főbb komponensei közé sorolható az EV3 vezérlőegység⁴, giroszkóp szenzor⁵ és a két nagy szervomotor⁶, amelyek egy tengelyen helyezkednek el. Az előbb említett elemekről bővebben szó lesz a 2.3 illetve 2.4 fejezetben. Mivel a kerekek egy tengelyen

1. <https://swagway.com>

2. <http://www.segwaymagyarorszag.com/>

3. http://robotsquare.com/wp-content/uploads/2013/10/45544_gyroboy.pdf

4. http://lego.wikia.com/wiki/45500_EV3_Intelligent_Brick

5. <http://shop.lego.com/en-US/EV3-Gyro-Sensor-45505>

6. http://lego.wikia.com/wiki/45502_EV3_Large_Servo_Motor

1. FEJEZET: BEVEZETŐ

helyezkednek el ezért instabil a szerkezete és könnyen, rövid idő alatt elveszti egyensúlyi állapotát. E probléma megoldására a robot dőlesi szöge, szög változásának a sebessége és a robot sebessége 0-hoz kell, hogy tartson, valamint annak érdekében, hogy egy helyben próbálja megtartani egyensúlyát a robot pozíciója is 0-hoz kell közelítsen.

A probléma megoldására alkalmas a PID⁷ szabályzó algoritmus használata, amely ipari körökben elterjedt. Viszonylag egyszerű a felépítése, kezelhetősége és az implementálhatósága. A PID szabályzó zárt ciklusos rendszer, melynek a bemeneti értéke a hiba, amely az elvárt érték és az aktuális érték különbsége. Esetünkben mivel 0-t kell közelítsünk annak érdekében, hogy ne veszítse el egyensúlyi állapotát a robot, ezért az elvárt érték örökké 0 lesz, vagyis a hibát négy komponens alkotja: a robot dőlesi szöge, szögsebessége, a robot sebessége és pozíciója, amelyek külön-külön súlyozva vannak. A szög és szögsebesség érték meghatározásához a giroszkóp szenzort használjuk és a szervomotorok beépített szenzorai által lekérhető fordulat szám segítségével számoljuk ki a robot sebességét és pozíóját.

A robot irányításának érdekében szükséges a PID szabályzó algoritmus módosítása és esetleges újabb szabályzók bevezetése annak érdekében, hogy irányítás alatt ne veszítse el az egyensúlyi állapotát. A felhasználónak lehetőséget ad a projekt részeként elkészített Android alkalmazás, hogy hálózaton, socketeken keresztül csatlakozzon a robothoz és az irányításnak megfelelő adatokat továbbítsa. Ezen adatok beviteli módját egy "touch joystick" teszi lehetővé, amellyel négy irányba lehetséges a robot vezérlése. Az adatok védelemét, a tovább bővíthetőséget, illetve a szerializációt a Google Protocol Buffers⁸ biztosítja. A Protocol Buffers platform és nyelvfüggetlen, könnyen kezelhető és gyors. Lehetőséget nyújt az adatok tetszőleges felépítésére, amelynek a forráskódját egy speciális generátor segítségével könnyen kigenerálható. E strukturált adatok írását illetve olvasását biztosítja a generált kód.

Az alkalmazás és a robot közti kapcsolat létrehozásának automatizálására Cling-UPnP(Universal Plug and Play) [15] könyvtárat használjuk, amely SSDP(Simple Service Discovery Protocol)⁹ protokollt használ.

A robot szabályzó algoritmusa Java-ban íródott, amelynek a futtatási környezetét a leJOS firmware biztosítja. Linux alapú és magába foglalja a JVM-t (Java virtual machine), amely lehetővé teszi, hogy a robot programozható legyen Java-ban. Számos firmware-t fejlesztettek ki annak érdekében, hogy a LEGO MINDSTORMS által fejlesztett vezérlőegységek programozhatóak legyenek magas szinten is. Pár ismert firmware: leJOS¹⁰ José Solórzano hozta létre 1999 végén, nyílt forráskódú, támogatja az objektum orientált programozást, ROBOTC [10] támogatja a C programozást, ev3dev [2], amely a szkript nyelveket támogatja (Phyton, NodeJS, Ruby).

Az előbb említett firmware-k teszik lehetővé a komplex feladatok megoldását, amelyek nem lehetégesek az EV3 alapértelmezett rendszerével. E rendszerhez a LEGO MINDSTORMS biztosított egy grafikus felületet, amely a kisebb korosztály számára készült, hogy "programozhassák" a saját kezűleg épített robotokat.

A dolgozat négy fejezetből áll. Az első fejezet által betekintést nyerünk a dolgozat témajába.

7. https://en.wikipedia.org/wiki/PID_controller

8. <https://developers.google.com/protocol-buffers/>

9. https://en.wikipedia.org/wiki/Simple_Service_Discovery_Protocol

10 <http://www.lejos.org>

1. FEJEZET: BEVEZETŐ

A második fejezet röviden bemutatja a LEGO megalakulását, a LEGO MINDSTORMS által kifejlesztett generációkat, majd bemutatja az EV3 készlethez tartozó motorokat és szenzorokat. Tartalmazza azon eszközök részletes leírását, amelyek a projekt során felhasználásra kerültek.

A harmadik fejezet által bemutatásra kerül a PID szabályzó működése és használata e projekt esetén.

A negyedik fejezet célja, hogy bemutassa e szakdolgozat alatt létrehozott projekt működését és az elkészített Android mobil alkalmazást. Valamint a projekt elkészítése során felmerült problémákat, ezek megoldását, illetve lehetséges megoldását.

2. fejezet

LEGO

Összefoglaló: A fejezetben bemutatásra kerül a LEGO megalakulása, fejlődése. E mellett sor kerül a LEGO MINDSTORMS által kifejlesztett technológiák bemutatására, amelyek közül a giroszkóp szenzor és a nagy motorok felhasználásra kerülnek a továbbiakban.

2.1. LEGO megalakulása

A LEGO [3] története 1932-ben kezdődött, Ole Kirk Christiansen¹ asztalos vállalkozást alapított Dánia, Billund nevezetű falujában, amely a fa játékok gyártásával foglalkozott. 1934-ben vette fel a céget a LEGO nevet, amely a LEG GODT dán szóból ered. 1947 után kezdtek el műanyagból előállítani játékokat, amelyek fő célja az volt, hogy könnyedén egymásba illeszthetőek, illetve szétszedhetőek legyenek. Az alapító fia, Godtfred Christian, lett az igazgató 1958-ban, ekkor alakult meg a ma ismert LEGO vállalat és vezetésével fellendült a cégek. Az első számítógép által vezérelt robot 1986-ban jelent meg, amelyet követően 1988-ban a LEGO és az MIT (Massachusetts Institute of Technology) együttműködésével elkezdődött az "inteligens téglák" fejlesztése, mely lehetőséget ad a programozhatóságra.

2.2. LEGO MINDSTORMS

A LEGO első játéka, amit forgalmazott Ole Kirk Christians vezetésével, a fából készült "LEGO Duck". Évekkel később megjelentek a műanyag játékok. Ma már világszerte ismert a LEGO építőjáték, egymással összeilleszthető, kombinálható elemeket tartalmaz, így szinte bármi megépíthető belőle és rendkívül hasznos oktatási eszköz.

A LEGO MINDSTORMS [6] egy programozható robotikai építőkészlet. Lehetőséget ad arra, hogy megépítsd, programozd és irányítsd a robotot. 1998-ban jelent meg az első generáció, az RXC (Robotic Command eXplorers), akkoriban nagy előrelépést számított, mivel teljesen autonóm. Képes számítógép nélkül is működni de, mára már nem gyakori a használata. Az évek során sokat fejlődött és ma már egyre több oktatási intézmény használja a robotika oktatásban.

2.2.1. RXC

Az RXC [11], amelyet a 2.1 ábra szemlélteti, az első generációs LEGO MINDSTORMS. Ma már nem igen használatos, elavult programozható vezérlőegység, mivel csupán 8 bites mikrokontroller és 32 Kbyte

1. http://lego.wikia.com/wiki/Ole_Kirk_Christiansen

2. FEJEZET: LEGO

RAM-al rendelkezik. Tartalmaz három szenzor és három motor csatlakozó portot. Ezek mellett egy LCD kijelzőt, amin látható az akkumulátor töltöttségi szintje, a bemeneti és kimeneti portok állapota. Egyéb információk megjelenítésére is alkalmas. A kapacitásának ellenére számos projektben felhasználták, ilyenekben mint például a Brick Sorter², amely szín szerint szortírozza az adott elemeket, RCX Remote Control³, amely két RCX vezérlőegység közti kommunikációval irányítja a robot.



2.1. ábra. RXC brick
<http://www.noucamp.org/NXT.html>

2.2.2. NXT

A második generáció a 2.2 ábrán látható, az NXT [7][8], 2006-ban jelent meg illetve az NXT 2.0 2009-ben, amely több építőelemet és szenzort tartalmaz. Több mindenben különbözik az NXT az RXC-hez képest. Szembetűnő esztétikai változások történtek, javítottak a kapacitáson, növelték a portok számát és megjelentek a szervomotorok.

Az NXT tartalmaz négy bemeneti portot, három kimeneti portot és egy 2.0 USB portot. Beépített bluetooth kommunikációs adapterrel rendelkezik, megjelent a grafikus kijelző és tartalmazott egy 8Khz hangszórót. Az NXT 32 bites AMTEL ARM7 mikrokontrollerrel, 256 Kbyte flash memoriával és 64 Kbyte RAM-al rendelkezik. Az NXT csomaggal már komplexebb projektek is készültek, mint az RXC-vel. Megemlíteném a Segway with Robot Driver⁴ című projektet, amely megvalósított egy olyan robotot, amely irányítja a szintén NXT vezérlőegység által működtetett Segway-t.

2.2.3. EV3

A harmadik generáció, az EV3 [1][8] 2013-ban jelent meg, a 2.3 ábra szemlélteti. Az "EV" evolúciót jelent és a 3-as azt, hogy harmadik generáció. Az EV3 a legfejlettebb LEGO MINDSTROMS progra-

2. <http://robotsquare.com/2012/02/15/brick-sorter-4/>

3. <http://robotsquare.com/2012/02/14/rcx-remote-control/>

4. <http://robotsquare.com/2012/09/04/segway-with-robot-driver/>

2. FEJEZET: LEGO



2.2. ábra. NXT brick

<http://shop.lego.com/NXT-Intelligent-Brick>

mozható építőelem, amely kezeli a motorokat, a szenzorokat és lehetőséget ad Bluetooth-on keresztül való kommunikációra.

Az EV3 egy ARM9 nevű processzorral van felszerelve, amely 300 Mhz, 16 Mbyte flash memóriával és 64 Mbyte RAM-al rendelkezik. A Linux alapú operációs rendszere nagyban segíti a programozhatóságát. Egy 2.0 mini USB PC porton teszi lehetővé a számítógéppel való kommunikációt, aminek átviteli sebessége 480 Mbit/s, az SD kártya olvasója 32 Gbyte-ot ismer fel. Tartalmaz egy USB portot amely lehetővé teszi, hogy használunk Wi-Fi dongle-t, ezáltal megkönyíti a programok kitelepítését, fájlok kezelését és lehetségesse teszi a kommunikációt okos készülékekkel is.

Az NXT-hez képest sokat fejlődött, nagyobb a kapacitása, lehetőség van Wi-Fi és SD kártya használatára. Ezek mellett négy szenzor port van három helyett és négy motor csatlakozó port. Nagyobb LCD kijelzőt raktak és a felületén hat gomb van négy helyett, könnyítve az operációs rendszer menüpontjainak kezelését.[8]

Mivel a LEGO MINDSTORMS generációk közül a legfejlettebb ezért a projekt által felhasznált vezérlőegység az EV3.



2.3. ábra. EV3 brick

<http://shop.lego.com/EV3-Intelligent-Brick>

2. FEJEZET: LEGO

2.3. Ev3 motorok

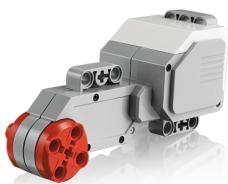
Az NXT generációval megjelentek a szervomotorok, amelyek legfőbb tulajdonsága a pontosság. Két típusa jelent meg a közepes és a nagy szervomotor. Mind a kettőnél megmaradt a pontosság, de a méretük, az erejük és a reagálási idejük eltérnek. Tovább fejlesztették a motorokat, amelyek az EV3 generációval jelentek meg. Az EV3 nagy motorjának teljesítménye megegyezik az NXT-vel de a felépítését optimalizálták, illetve az EV3 közepes motorja teljesen új az NXT-hez képest.

2.3.1. Nagy motor

A nagy motor [13], amelyet a 2.4 ábrázol, erőteljes, ideális az általunk megépített robot irányítására. A motorba beépített forgásérzékelő által információkat lehet lekérni a pillanatnyi állapotáról, amely fokban vagy teljes fordulatban mér.

A nagy motorok sebessége 160-170 rmp, forgatónyomatéka 20 N/cm és, ha blokkolt állapotba kerül, akkor a nyomatéka 40N/cm. A motor beépített forgásérzékelője lehetővé teszi a pontos vezérlést +/- 1 fok pontossággal.

A robot mozgatását a nagy motorok segítségével fogjuk elérni, habár lassabbak, mint a közepes motorok, de nagyobb a nyomatékuk és a precizitásuk azonos.



2.4. ábra. Nagy motor
<http://shop.lego.com/EV3-Large-Servo-Motor>



2.5. ábra. Közepes motor
<http://shop.lego.com/EV3-Medium-Servo-Motor>

2.3.2. Közepes motor

A közepes motor, amelyet a 2.5 ábrázol, tulajdonsági megegyeznek a nagy motor tulajdonságaival. Ugyancsak tartalmaz forgásérzékelőt és lehetséges a helyzetmeghatározás 1 fok eltéréssel, viszont alacsonyabb terhelésre terveztek, forgatónyomatéka 8 N/cm, ha blokkolt állapotba kerül, akkor a nyomatéka 12 N/cm és a sebessége 240-250 rmp.

2.4. Ev3 szenzorok

Az NXT csomagban számos szenzor megtalálható, ilyenek az érintés, a hang, a szín és az ultrahang szenzorok, amelyeket felhasználva több különböző funkcionálisokkal bővíthetjük a megépített robotunkat.

2. FEJEZET: LEGO

Az EV3 megjelenése magával hozott még két új szenzort, a giroszkóp és az infravörös szenzorokat. Ezek mellett fejlesztették a már meglévő szenzorokat is.

2.4.1. Színszenzor

A digitális színszenzor, amelyet a 2.6 ábrázol, hét különböző színt ismer fel. A színpelismerő funkcionáltsán kívül mérhető vele a fényvisszaverődés erőssége vagy a környezeti fény intenzitása. E három üzemmód lehetővé teszi a felhasználóknak például, hogy kövessen egy fekete vonalat vagy megkülönböztessen szín szerint tárgyat és még sok más felhasználási lehetősége van.



2.6. ábra. Szín szenzor
<http://shop.lego.com/EV3-Color-Sensor>

2.4.2. Giroszkóp szenzor

A digitális giroszkóp szenzor [13][12], amely látható a 2.7 ábrán, mi is felhasználunk, az EV3-al egy időben jelent meg. A giroszkóp igen elterjedt szenzor, megtalálható okos telefonokban, illetve különféle navigációs rendszerek vagy akár irányításért felelős vezérlőegységek használják.

Az EV3 giroszkóp szenzora egy tengely mentén tud mérni. Látható a 2.8 ábrán, hogy a giroszkópon fel van tüntetve két nyíl és ennek segítségével be tudjuk állítani a pozícióját, ha jobb oldali nyíl irányába mozdítjuk, akkor mínusz értéket kapunk, ha ellentétesen, akkor egyértelműen pozitívat.

A giroszkóp három módban használható. Mérhető az elfordulási szög fokban -90 és 90 intervallumban, a szög gyorsulása, illetve lehetséges a két mód használata egyszerre. A pontossága szögmérés esetén +/-3 fok, szög gyorsulása esetén pontosabb. Maximális információ megosztási sebessége 440 fok/másodperc és a mintavételezési sebessége 1 kHz. A sebességét kihasználva simítjuk a +/-3 fok szórását oly módon, hogy többször mintavételezünk.

2.4.3. Nyomásérzékelő

Az analóg nyomásérzékelője egy piros gombbal van felszerelve, ami látható a 2.9 ábrán. Működése nem túl bonyolult, viszont annál hasznosabb. Érzékeli a gomb lenyomását illetve felszabadulását, amelyet egy integrált számláló segítségével nyomon lehet követni.

2. FEJEZET: LEGO



2.7. ábra. Giroszkóp
szenzor
<http://shop.lego.com/EV3-Gyro-Sensor>



2.8. ábra. A giroszkóp
szögelfordulási mutatója
<https://education.lego.com/mindstorms-education-ev3>



2.9. ábra. Nyomásérzékelő
<http://shop.lego.com/EV3-Touch-Sensor>

2.4.4. Ultrahang szenzor

A digitális ultrahang szenzor, amely a 2.10 ábrán látható, +/- 1 cm hibával meghatározza, hogy milyen távolságra van az előtte lévő tárgy. A hatótávolsága 250 cm és működése a magas frekvencia kibocsátására alapszik. Számolja az előtte lévő tárgyról visszaverődő frekvenciák érkezésének idejét, ezáltal határozza meg a távolságot. Az akadály pontos irányának meghatározása úgy lehetséges ha több mérést végzünk oldalmozgással. Így lehetőség van arra, hogy kiszámoljuk az akadály irányát.



2.10. ábra. Ultrahang szenzor
<http://shop.lego.com/EV3-Ultrasonic-Sensor>

2. FEJEZET: LEGO

2.4.5. Infravörös szenzor

A digitális infravörös szenzor 2.11 ábrán látható, amelynek a maximális hatótávolsága 70 cm, jelentősen kisebb az ultrahang szenzorhoz képest. Működése inkább vezérlésre alkalmas, 2 m hatótávolságról is érzékeli a LEGO MINDSTORMS saját infravörös jeladóját, amely latható a 2.12 ábrán.



2.11. ábra. Infravörös szenzor
<http://shop.lego.com/EV3-Infrared-Sensor>

2.12. ábra. Infravörös jeladó
<http://shop.lego.com/EV3-Infrared-Beacon>

2.4.6. Az egyensúlyozó robot felépítése

A kétkerekű egyensúlyozó robot, amely a 2.13 ábrán látható, a Gyro Boy⁵ modell alapján készült el és az EV3 csomagból építhető meg. Szimmetrikus súlyelosztása révén elősegíti az egyensúly megtartását. Megemlítenék még más egyensúlyozó robotot is, az NXTway-GS⁶ modellt, amely NXT csomaggal valósítható meg, illetve a Rover Kit⁷, amely mikrovezérlő által irányított egyensúlyozó robot.

A robot főbb komponensei közé sorolható az EV3 vezérlőegység, a giroszkóp szenzor és a két nagy szervomotor. A motorok egy tengelyen helyezkednek el, amelyekre fel vannak erősítve a kerekek. A kerekek átmérője 55mm, ezt az értéket felhasználjuk a pozíció illetve a sebesség kiszámolásakor. A giroszkóp szenzor egy tengely mentén mér és a robot központi részén helyezkedik el, közel az egyensúlyi pontjához.

A robothoz tartozik egy állvány, amely a kiinduló pontja és egyensúlyba tartja amíg az egyensúlyozást megvalósító algoritmus elindul.

Az egyensúlyozás megvalósítására az algoritmus a PID szabályzót használja fel, amely egy zárt ciklusos rendszer. A PID szabályzó bemeneti értéke a hiba és a kimeneti a motorokra leadott erő.

A hibát az elvárt érték és az aktuális érték különbségéből kapjuk meg. Esetünkben, a kiindulási pont a robot egyensúlyi állapota, azaz a robot dőlési szöge, szög változásának a sebessége, a robot sebessége és a pozíciója nulla, amely az elvárt érték.

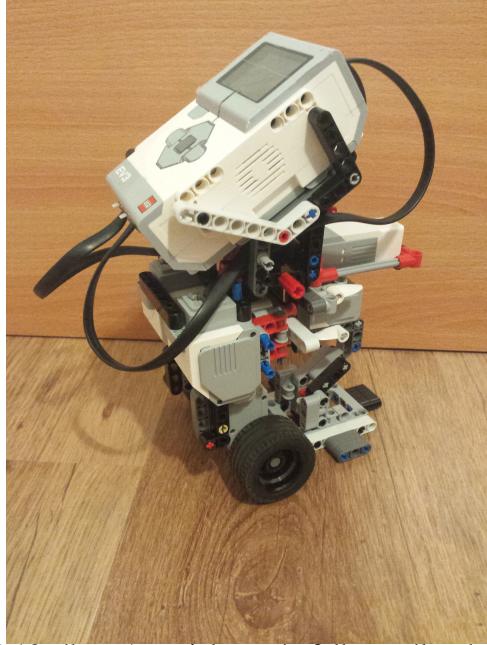
A szög és szög változásának sebességét a giroszkóp szenzor által kérjük le. A leJOS firmware több módot is biztosít a giroszkóp használatára. Esetünkben a rate módot használjuk, amely a szög változásának a sebességét méri. Mérések esetén, mivel a szenzor nem mér pontosan, többször mintavételezünk és ezeket átlagoljuk, hogy kiszűrjük a zajokat. Az így megkapott szög változásának sebességéből kiszámítható a dőlési szög a következő képlettel:

5. http://robotsquare.com/wp-content/uploads/2013/10/45544_gyroboy.pdf

6. http://lejos-osek.sourceforge.net/NXTway-GS_Building_Instructions.pdf

7. <http://www.sainsmart.com/sainsmart-balancing-robot-kit.html>

2. FEJEZET: LEGO



2.13. ábra. A projekt során felhasznált robot

$$\varphi = \varphi_0 + \omega \cdot t, \quad (2.1)$$

ahol az ω jelöli a szög változásának sebességét, φ_0 az eddigi szögek összegét, illetve t az eltelt időt.

A szervomotorok esetén lekérhetjük a fordulatszámát, amelyet a motorba beépített szenzor egy szám-láló segítségével számol. A két motor fordulatszámát átlagoljuk, ezáltal kiszűrjük a zajokat. Az így megkapott értéket átalakítjuk szögbe, majd kiszámítjuk a robot sebességét, valamint pozícióját.

A sebességet és a pozíciót a következő képletekkel határozzuk meg:

$$v = \frac{\alpha - \alpha'}{t} \cdot d \quad (2.2)$$

$$x = \alpha \cdot d, \quad (2.3)$$

ahol az α jelöli a motor elfordulási szögét, az α' az előző időpillanatban a motor elfordulási szögét, a t az eltelt időt, a d a kerék átmérőjét, a v a robot sebességét, illetve az x a robot pozícióját.

Az előbbiekben kiszámított négy értéket kivonjuk a nekik megfelelő elvárt értékekből, amelyek nullaik. Ezt követően konstansok által súlyozva, majd összeadva az értékeket, megkapjuk a PID bemenetét vagyis a hibát.

3. fejezet

Egyensúlyozás

Összefoglaló: E fejezet célja, hogy bemutassa a PID szabályzó algoritmus működését. Emellett sor kerül az egyensúlyozás működésének leírására, valamint, hogy miként lehet felhasználni ez esetbe a PID szabályzó algoritmust.

3.1. PID szabályzó algoritmus

A szabályzó rendszereket kétféleképpen nevezhetjük nyílt vagy zárt ciklusosnak. Nyílt ciklusos rendszerek esetén nincs visszacsatolás, ezért nem megfelelő komplex problémák megoldására. Előnyös a használata, mivel alacsony költségvetésű. A zárt ciklusos rendszerek visszacsatolással működnek, vagyis a rendszer kimenetének hatására visszajelzést kapunk.

A PID (Proportional Integral Derivative) [16] elterjedt zárt ciklusos rendszer az ipari szabályzó körökben, köszönhetően a viszonylag egyszerű félépítésének és a könnyű kezelhetőségének. Számos felhasználási lehetősége van, ilyenek a nyomás, hőmérséklet, sebesség szabályozás. E szabályzó rendszer bementi értéke az aktuális hiba.

A hibát az elvárt érték és az aktuális érték különbsége határozza meg. Esetünkben az aktuális értékek a következők: robot dőlesi szöge, szög változásának sebessége, a robot pozíciója, illetve sebessége. Az előbb felsorolt értékek esetén az elvárt értékek, ahhoz, hogy a robot egyensúlyi állapotba maradjon, 0-hoz kell közelítsenek.

A szabályzó kimeneti értéke a motorokra leadott erő. Esetünkben, tehát a visszacsatolás az előbb felsorolt négy értékben nyilvánul meg, mivel a kerekekre leadott erő befolyásolja ezen értékeket.

A PID, mint a nevéből is látható, három tagból tevődik össze: P (Proportional) az arányos tag, I (Integral) a hibák integrált tagja és a D (Derivative) a hibák időbeli változásának derivált tagja. Abban az esetben, ha az előbb felsorolt tagok közül nem veszünk figyelembe minden tagot, akkor a következő szabályzókról beszélhetünk: P, PI és PD.

PID szabályzó matematikai képlete:

$$u(t) = K_P \cdot e(t) + K_I \cdot \int e(t)dt + K_D \cdot \frac{d}{dt}e(t),$$

ahol t jelöli az időt, $e(t)$ a bemeneti hibát, illetve az $u(t)$ a kimenetet. A K_P, K_I, K_D paraméterek rendre az arányos, az integrál és a derivált tagok súlya, amelyek meghatározzák a szabályzó érzékenységét és teljesítményét.

Kizárálag az arányos tag használata esetén a szabályzó rendszer olyan esetknél használható, ahol

3. FEJEZET: EGYENSÚLYOZÁS

megengedett egy bizonyos nagyságú hiba a kimenetben, miután a rendszer stabilizálódott. Az arányos tag paramétere a K_P , amely egy konstans érték, ha túl nagy instabil rendszert eredményez.

Az integrál tag önmagában nem használható, mint az arányos tag. Az integrál tag esetén a hibák összege mindaddig változik míg az arányos tag nulla nem lesz. Tehát a feladata egy pontosabb és precízebb szabályzó rendszer biztosítása. Numerikus módszer segítségével közelítjük meg, melynek a következő a képlete:

$$u(t) = K_I \cdot \sum e(t),$$

ahol az előbbihez hasonlóan K_I konstans.

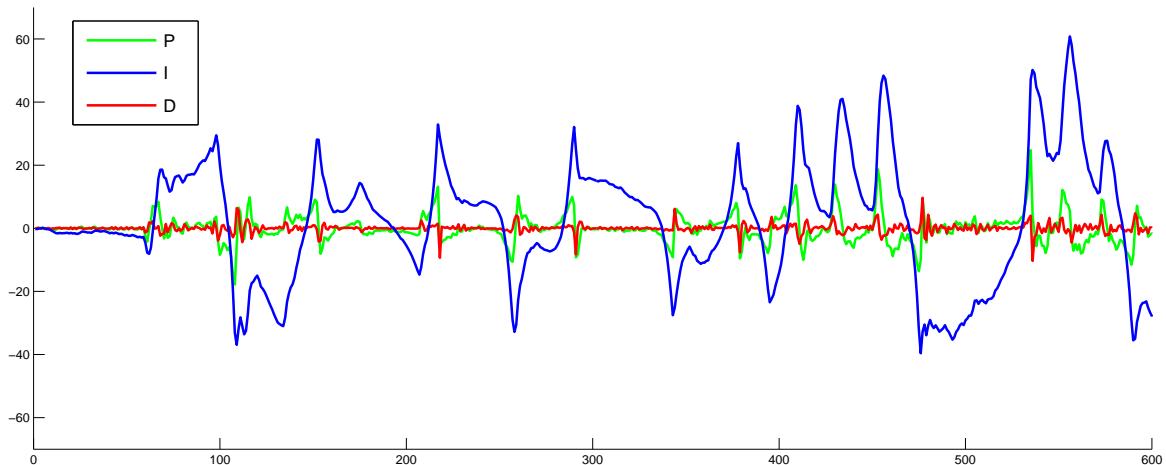
A derivált tag értéke arányos a hiba változási sebességével. Feladata a hirtelen változások késleltetése, ezáltal egy stabilabb rendszert biztosítva. Ez esetben is közelítéssel oldható meg a probléma:

$$u(t) = K_D \cdot (e(t) - e(t-1)),$$

ahol szintén a K_D konstans.

A PID szabályzó működését a K_P, K_I, K_D tagok finomhangolásával tudjuk befolyásolni. Ezáltal feladatra szabható a szabályzó. A PID szabályzó implementálása során a nehézség abban rejlik, hogy megfelelően hangoljuk a tagok értékét. Finomhangolás alatt azt értjük, hogy a megfelelő értékek megválasztásával egy működő rendszert hozzunk létre.

A 3.1 ábrán látható a PID szabályzó tagjainak változása egy teszt esetén. Jól látható az ábrán, hogy a derivált tag ellentétes előjelű az arányos taghoz viszonyítva, ennek köszönhetően a hirtelen változások késleltetve lesznek. Az integrált tag és az arányos tag esetén jól látható, hogy hasonló oszcillációt végeznek.



3.1. ábra. Egy teszt során kapott PID tagjainak értékeit ábrázolja. Az x tengelyen jelöljük az iterációk számát, az y tengelyen a PID tagok értékének mértékét.

4. fejezet

Megvalósítás

Összefoglaló: A dolgozat tárgyát képező projekt egy EV3 készletből épített kétkerekű egyensúlyozó robot irányítását valósítja meg hálózaton keresztül, telefonos alkalmazás segítségével. E fejezetben bemutatásra kerülnek a megvalósítás során felmerült problémák, ezek megoldása és a felhasznált technológiák.

4.1. EV3 programozása

A LEGO MINDSTORMS kifejlesztett egy programozási környezetet, mely célja, hogy a megépített robotot különböző funkcionalitásokkal lehessen felruházni. E környezet lehetővé teszi a kisebb korosztály számára is a robotok programozását. Különböző grafikus elemekből, úgynévezett blokkokból épül fel a program, amely USB-n keresztül kitelepíthető az EV3 vezérlőegységen futó LEGO MINDSTORMS által fejlesztett firmware-re. Az előbb említett programozási környezet nem alkalmas komplexebb problémák megoldására. Ezért több firmware-t is kifejlesztettek, melyek magas szintű programozási nyelvek használatát támogatják, ilyen a ROBOTC¹ Carnegie Mellon egyetem fejlesztette ki, amely támogatja a C programozást, illetve megemlíteném még a Debian² által kifejlesztett ev3dev-et³ amely a szkript nyelveket támogatja (Phyton, NodeJS, Ruby) . Esetünkben a leJOS firmware-t használjuk.

A leJOS firmware-t José Solórzano hozta létre 1999 végén és azóta is folyamatosan fejlesztik. Linux alapú, nyílt forráskódú, magába foglalja a JVM-t (Java virtual machine), a neve is rámutat a Java programozhatóságra JOS(Java Operating System). Futtatási környezetet biztosít a Java programozóknak, támogatja az objektum orientált programozást. Mindezek lehetővé teszik a socket alapú kommunikációt, szinkronizálhatóságot, szálak alkalmazását és Java típusok használatát.

Az EV3 vezérlőegységen található négy bemeneti (S1, S2, S3, S4), illetve négy kimeneti (A, B, C, D) port. A bemeneti portok esetén a szenzorok csatlakoztathatók, valamint a kimeneti portok esetén a motorokat. A leJOS hozzáférést biztosít ezekhez a portokhoz. Konkrétnan meg kell adjuk, hogy milyen porton található a motor, illetve a szenzor. Motorok esetén importálnunk kell `lejos.hardware.port.MotorPort` csomagot és így elérjük a `MotorPort` interface-t. Szenzorok esetén is hasonló az eljárás, azzal a különbséggel, hogy ebben az esetben a `lejos.hardware.port.SensorPort` csomagot kell importáljuk a `SensorPort` interface elérésséhez.

1. <http://www.robotc.net>

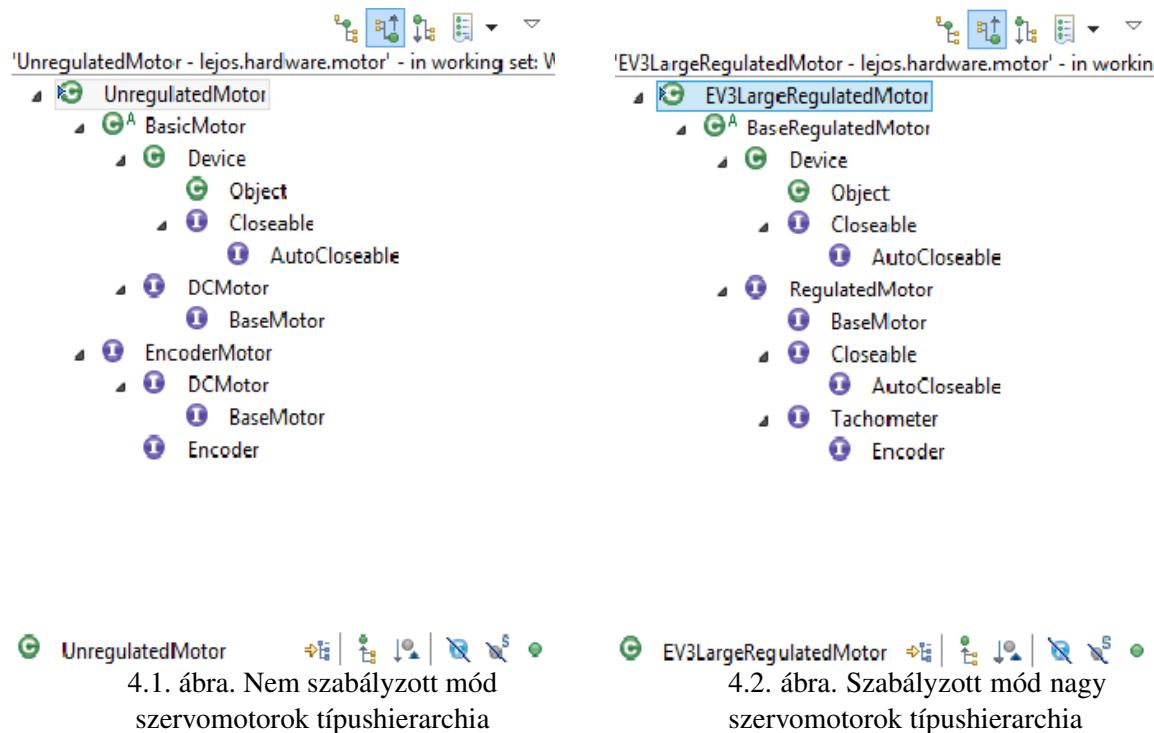
2. <https://www.debian.org/>

3. <http://www.ev3dev.org/>

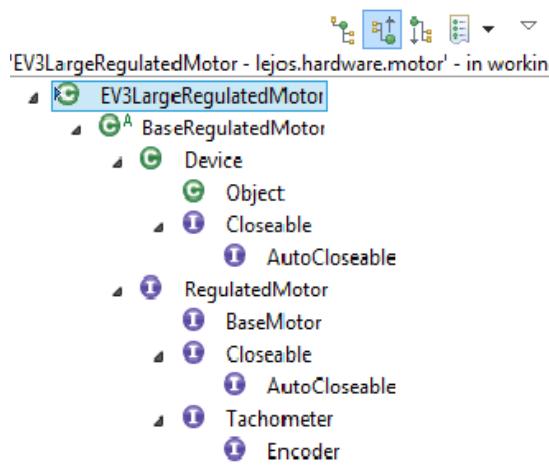
4. FEJEZET: MEGVALÓSÍTÁS

A motorok, illetve a szenzorok példányosításához szükséges megadni a motor vagy szenzor megfelelő portját. A leJOS elrejti a szenzorok implementációját, lehetővé téve a magas szintű absztrakció használatát.

A motorok példányosítása esetén az objektum típusának kiválasztása a motor típusától függ és a mód kiválasztásától. A motor típusát meghatározza a mérete(nagy, illetve közepes motorok). Módok kiválasztása esetén lehetőségünk van szabályzott mód (4.4 ábrán látható), illetve nem szabályzott mód (4.3 ábrán látható) közti választásra. A motorok nem szabályzott módjának típushierarchiája megtekinthető a 4.1 ábrán, illetve a szabályzott mód a 4.2 ábrán.



4.1. ábra. Nem szabályzott mód
szervomotorok típushierarchia

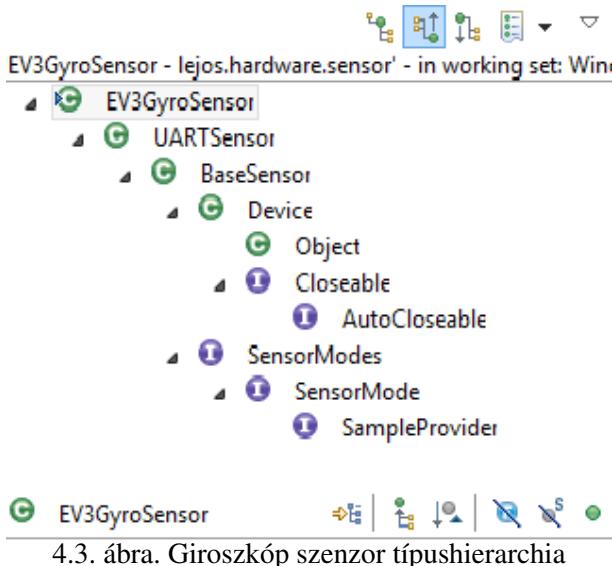


4.2. ábra. Szabályzott mód nagy
szervomotorok típushierarchia

A szenzorok esetén az objektum példányosítását követően választjuk ki a lehetséges módok közül a nekünk megfelelőt. Ehhez szükségünk van a SensorMode objektumra, amely a későbbiekben biztosítja e mód metódusait. A szenzoroknak megfelelő objektum esetén a motorokkal ellentétbe, mindenkoruk származtatja a UARTSensor osztályt, a 4.3 ábrán látható.

A 4.1 és a 4.2 ábrákon látható a leJOS firmware által biztosított két mód, a giroszkóp szenzor használatához. A rate mód a szögsebességet méri, amelyet szög/másodperce -ben kapunk meg. Esetünkben szükségünk van a robot dőlési szögére is, amelyet a 2.4.6 alfejezetben látható 2.1 képlet segítségével kiszámítunk. A 4.1 ábrán látható az előbb említett mód kiválasztása és ezt követően a mintavételezés, melynek eredménye a sample tömb első eleme lesz. Használható angle módban is, amely 4.2 ábrán látható. E mód a szenzor kezdő orientációjához képest mér. A mintavételezés hasonlóan történik, mint az előbb említett módnál, annyi különbséggel, hogy ez esetben szöget mér a kezdő pozíciójához képest. Mindkét mód esetében a reset metódus hívással újra kalibrálhatjuk a szenzort.

4. FEJEZET: MEGVALÓSÍTÁS



4.3. ábra. Giroszkóp szenzor típushierarchia

4.1. Listing. Giroszkóp szenzor rate mód használata

```

1 float[] sample = new float[1];
EV3GyroSensor gyroSensor = new EV3GyroSensor(SensorPort.S2);
SampleProvider gyroMode = gyroSensor.getRateMode();
6 gyro.fetchSample(sample, 0);

```

4.2. Listing. Giroszkóp szenzor angle mód használata

```

2 float[] sample = new float[1];
EV3GyroSensor gyroSensor = new EV3GyroSensor(SensorPort.S2);
SampleProvider gyroMode = gyroSensor.getAngleMode();
7 gyro.fetchSample(sample, 0);

```

A nagy motorok esetében is két módot biztosít a leJOS firmware, a nem szabályzott (4.3 ábra) és a szabályzott (4.4 ábra) módok.

A motor nem szabályzott mód (4.3 ábra) használata esetén az irányítást a `setPower` metódus hívással valósítjuk meg, amelynek -100 és 100 közötti értéket adhatunk át. Az átadott érték határozza meg, hogy mekkora erőt fejtsen ki a motor, ha az átadott érték pozitív, akkor előre forog, illetve, ha negatív akkor hátra. A `resetTachoCount`, valamint `getTachoCount` metódusokkal kezeljük a motorban levő forgásmérő szenzort, melynek számlálójának értékét lekérhetjük vagy lenullázhatjuk. Esetünkben szükséges a pozíció, illetve a sebesség meghatározása, a megvalósításához szükséges az előbb említett metódus által lekérni a két motor fordulat számát, amelyeket átlagoljuk, ezáltal kiszűrjük a szenzor mérsénél keletkezett zajokat. Az így megkapott értéket átalakítjuk szögbe, majd kiszámítjuk a 2.4.6 alfejezetben megjelenő (2.2) és (2.3) képletek segítségével. A `stop` metódus hívása esetén a motor blokkolt állapotba kerül.

A motor szabályzott mód (4.4 ábra) esetén némely funkcionálitás eltér az előbb említett módhoz képest. Ez esetben a motor sebessége fok/másodperc-ben megadható a `setSpeed` metódus által. A forgás sebessége függ az akkumulátor töltöttségi szintjétől, a maximális sebessége 740 fok/másodperc. A

4. FEJEZET: MEGVALÓSÍTÁS

`rotate` metódussal adjuk meg, hogy hány fokot forduljon a motor. E metódusnak két változata van. Megadhatjuk csak a fokot, vagy egy logikai értékkel megadható, hogy miután elérte az adott forgási fokot magától megálljon a motor. Ez esetben ha forgás közben meghívódik egy másik metódus, mely parancsot ad a motornak, akkor az előbb kiadott utasítás végrehajtása leáll. A `waitComplete` metódussal lehetőség van arra, hogy bevárja a forgás befejezését, tehát addig vár, míg végrehajtja a motor az azelőtt kapott utasítást. Az előbb említett mód esetén a pozíciót és a sebességet ki kellet számolni. Ez esetben lehetőség van ezen értékek lekérésére a `getPosition` és `getSpeed` metódusok által.

Esetünkben lényeges különbség a két mód között, hogy a szabályzott mód esetén jóval lassabb a motorok reagálási ideje, mint a nem szabályzott mód esetén. E mellett a szabályzott mód használatakor, ha kiadunk egy utasítást, akkor kell figyeljünk, hogy utána ne adjunk olyan más utasítást, amely miatt az előzőt leállítaná.

4.3. Listing. A nagy motor `unregulated` mód használata

```
2 EncoderMotor motor = new UnregulatedMotor(MotorPort.A)
motor.resetTachoCount();
motor.setPower(40);
7 int tacho = motor.getTachoCount();
motor.stop();
```

4.4. Listing. A nagy motor `regulated` mód használata

```
RegulatedMotor motor = new EV3LargeRegulatedMotor(MotorPort.A);
motor.resetTachoCount();
motor.setSpeed(600);
motor.rotate(720);
motor.waitComplete();
motor.rotate(-460, true);
10 float position = motor.getPosition();
```

Annak érdekében, hogy az EV3 vezérlőegységen futassuk és könnyedén kitelepítsük a programokat, az Eclipse IDE fejlesztői környezetet használjuk és a leJOS plugin-t, amelyek mindezet megkönnyíti. Abban az esetben, ha nem szeretnénk használni az Eclipse fejlesztői környezetet, lehetőség van az Ant⁴ build rendszer használatára, amely `build.xml` konfigurációs állományában megadhatóak a dependenciák. Az Eclipse használata esetén manuálisan át kell másoljuk a dependenciák jar fájljait, az SD kártyára telepített leJOS firmware-t `/home/root/lejos/ejrel.7.0_60/lib/ext` mappába, hogy a roboton futó program használni tudja a függőségeit.

Mivel az EV3 vezérlőegységen az alapértelmezett firmware van telepítve, ezért külön SD kártyára fel kell telepíteni a leJOS-t. Legalább 2GB-os SD kártya szükséges de ne legyen 32GB-nál nagyobb és ne SDXC típusú legyen, mert nem ismeri fel az EV3 hardware. Az SD kártyát szükséges formázni FAT32 típusú partícióra. A leJOS számítógépre való telepítése során szükség lesz az 1.7 JDK-ra(Java Development Kit). Az előkészített program segítségével feltelepíthető a leJOS firmware az SD kártyára, ehhez még kell a JRE(Java Runtime Environment) is. Mivel az EV3 ARM9-es processzorral rendelkezik, ezért fontos, hogy a JRE ARM verzióját töltök le. Sikeres telepítés

4. <http://ant.apache.org/>

4. FEJEZET: MEGVALÓSÍTÁS

után az SD kártyát behelyezve az EV3 vezérlőegységbe, elindítható a firmware. Ha az alapértelmezett rendszer indul el, akkor meg kell ismételni az SD kártyára való telepítést. Ezt követően telepít-sük az Eclipse plugin-t, majd állítsuk be az EV3_HOME környezeti változónak a feltelepített leJOS plugin elérési útvonalát (C:\ProgramFiles\leJOSEV3) és a könyvtárán belüli bin könyvtárat (C:\ProgramFiles\leJOSEV3\bin) a Path-nek. Ezek a beállítások teszik lehetővé az operációs rendszer számára, hogy futtatás esetén megkapja a szükséges fájlok elérési útvonalát.

4.2. Androidos alkalmazás és kommunikáció

A projekt része egy telefonos alkalmazás, mely célja, hogy hálózaton keresztül kapcsolódjon a robothoz egy köztes router segítségével és küldje a megfelelő utasításokat, annak érdekében, hogy a felhasználó tudja irányítani a robot mozgását. Az alkalmazást Android stúdióban készítettük és a kommunikációt Java socketen keresztül valósítottuk meg, amit a leJOS, Linux alapú firmware tesz lehetővé.

Az alkalmazás fejlesztésének célja, hogy könnyen, gyorsan csatlakozni tudjon a felhasználó a robothoz és a sikeres csatlakozás után könnyedén irányítani is tudja. A használatának előfeltétele, hogy a robot és az alkalmazást futtató telefon ugyanarra a router-re legyen rácslakozva a kapcsolódás és kommunikáció érdekében.

Az alkalmazást elindítva megadhatjuk a robot IP és PORT címét, amin keresztül csatlakozik. A csatlakozás során ellenőrizzük a bekért adatok helyes formátumát és azt hogy lehetséges vagy sem a kapcsolat. Az alkalmazás kezelését elősegíti egy általunk létrehozott "Remember me" funkcióval, mely célja, hogy a legutóbbi IP és PORT címet visszatölts az alkalmazás elindításakor. E megvalósítása a SharedPreferences API-n keresztül történik, érték és kulcsok alapján tárolódnak fájlba az adatok. Ezen adatok hozzáférési pontja a SharedPreferences objektum, amely könnyen kezelhető metódusokat biztosít ezek olvasására, illetve írására.

A sikeres kapcsolódást követően egy 2D-s joystick segítségével lehet irányítani a robotot négy irányba. A joystick vizuális megjelenítésére két kör rajzolunk ki. A nagy kör jelöli vizuálisan a határokat a felhasználónak és ezáltal meghatározunk egy konkrét intervallumot az irányításhoz szükséges értékeknek, valamint így kizártuk annak a lehetőségét, hogy olyan értékeket olvassunk a kisebb kör mozgatásának hatására, amelyek nem megfelelők. A kisebbik kör segít a felhasználóval tudatni, hogy a nagy kör peremén belüli rész az általunk értelmezett, illetve figyelembe vett felület a parancsok küldésére. E két kör a telefon képernyőjére canvas segítségével jelenítsük meg (4.4 ábrán látható), amely felületéhez hozzárendeljük a megfelelő eseményfigyelőt(OnTouchListener). Annak érdekében, hogy a felhasználó ne tudja kimozdítani a nagyobb körön belüli kisebb köröt, átalakításokat végezünk koordináták között.

A képernyőt megérintve az eseményfigyelő által, megkapjuk az x és y koordinátákat, ezeket a pontokat átalakítjuk polárkoordinátákba:

$$(x, y) \longrightarrow (r, \varphi)$$

$$r = \sqrt{x^2 + y^2}$$

$$\varphi = \arctg(y, x)$$

4. FEJEZET: MEGVALÓSÍTÁS

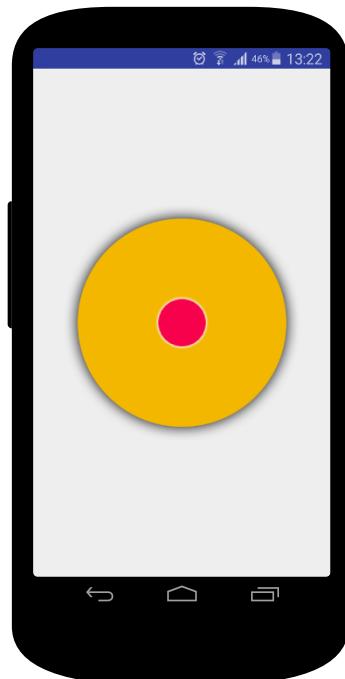
Tudva a két kör sugarának különbségét és a polárkoordinátákat, leellenőrizhető, hogy a kis kör a nagy kör sugarán kívül esik vagy sem. Abban az esetben, ha a nagy körön kívül esik a kirajzolási pont, akkor a sugár mentén rajzoljuk ki a kisebb kört a szög függvényében. Ehhez szükséges polárkoordinátából átalakítani euklideszi koordinátába:

$$(r, \varphi) \longrightarrow (x, y)$$

$$x = R \cdot \cos(\varphi)$$

$$y = R \cdot \sin(\varphi),$$

ahol R a két kör sugarának különbsége.



4.4. ábra. Joystick vizuális megjelenítése

Tudva a mozgatás irányát, socketen keresztül a roboton elindított szervernek küldjük a parancsokat, amelyeket továbbít az egyensúlyozásért felelős algoritmusnak. A kommunikációhoz szükséges szervert robot oldalon, az egyensúlyozásért felelős algoritmussal egy időben indítjuk el, külön-külön szálon.

A felhasználó által kapott irányítási parancsok alatt jól definiált értékekről beszélünk. Vagyis, a parancs függvényében más-más értékek előállításával érjük el a megfelelő értékek halmazát, amely szükséges, hogy a szabályzó megfelelően tudja értelmezni a parancsot, tehát a parancsok összetett értékekből állnak. Ezen értékeket egy modell objektumban tároljuk, amely küldésére szükséges a szerializáció használata.

4. FEJEZET: MEGVALÓSÍTÁS

4.2.1. Google Protocol Buffers

A szerializáció, értelmezése szerint, objektumok állapotának adatfolyamba való kiírása illetve kiolvasása. Ezáltal lementhetőek a későbbi felhasználásra és lehetséges távoli eszközökre való küldése az adatoknak. Tehát biztosítja a modell objektumban lementett értékek állapotát.

Az adatok szerializációját illetve titkosítását a Google Protocol Buffers által biztosítjuk, amely lehetővé teszi, hogy megszerkesszük az adatok struktúráját majd egy speciális generátorral létrehozzuk ezen strukturált adatok kezelésére a hozzá tartozó Java osztályt. E létrehozott osztályon keresztül könnyedén kezelhetjük a strukturált adatokat.

A Google Protocol Buffers [9] használatához létre kell hozzunk egy `.proto` kiterjesztésű állományt (4.5 ábra), amelyben definiáljuk az irányításhoz szükséges adataink struktúráját. A `.proto` állomány útmutatóként szolgál a speciális kód generátornak, amely generálja a struktúrához való hozzáférést biztosító osztályt. Az állomány első sorában deklaráljuk a csomag(package) nevét, második sorban konkrétan megadjuk a Java csomag hierarchiát és a harmadik sorban megadjuk az osztály nevét, amellyel rendelkezni fog a legenerált osztály. Abban az esetben, ha nem adunk meg konkrét nevet a `java_outer_classname` mezőben, akkor a `.proto` fájl nevét fogja megkapni. A további sorokban megadjuk az adattagokat, amelyek rendelkeznek típus névvel. Ez lehet `bool`, `int32`, `float`, `double` és `string`. minden attribútumnak megadunk egy számot, amely egyedi azonosítóként szerepel a bináris kódolás során. Ezekben kívül megadható három típus mező minden attribútumnak, amelyek a következők: `required`, `optional`, `repeated`. A `required` mezővel beállíthatjuk, hogy az adott attribútum kötelezően értéket kapjon, különben `RuntimeException` vagy `IOException` hibát eredményez. Az `optional` mezőt annak az adattagnak állítjuk be, amely nem biztos, hogy értéket kap futási időben, ebben az esetben megadhatunk `.proto` állományban egy alapértelmezett értéket ennek az adattagnak. A `repeated` típussal lehetséges annak a jelzése, hogy az adott adattag ismétlődni fog.

4.5. Listing. Az adatok strukturáját definíáló `.proto` állomány

```
4 package protobuf;
option java_package = "app.cs.ubb.edu.ev3.protobuf";
option java_outer_classname = "DataProtos";
5
6 message Data {
7
8     required int32 left = 1;
9     required int32 right = 2;
    required float forward = 3;
    required float speed = 4;
    required float angle = 5;
    required bool client = 6;
}
```

Összehasonlítva a Protocol Buffers-t az XML formátummal, az vehető észre, hogy egyszerűbb, kezelhetőbb az adathozzáférést biztosító osztály által és átláthatóbb, mint az XML. A Protocol Buffers használatakor az adathozzáférés metódus hívással megvalósítható, míg az XML esetén tag-ek nevei szerint amelynek meg kell adjuk, hogy hányadik elemére van szükségünk, tehát lényegesen különböző adat hozzáférési idő. Ezen tulajdonságok tudatában választottuk a Google Protocols Buffers-t és, mivel platform független, több lehetőség van a továbbfejleszthetőségre.

4. FEJEZET: MEGVALÓSÍTÁS

4.2.2. Robot oldali kommunikáció

A robot irányításához szükséges telefonos alkalmazással való kapcsolat létrehozása és az általa küldött utasítások értelmezése. Ahogy az előbbiekben is említettem, a kommunikációt Java socketeken keresztül valósítottuk meg.

A robot processzorának teljesítményét tekintve a Google Protocol Buffers a legmegfelelőbb az adatok szerializájójára.

Robot oldalon külön szálon indítjuk a klienst fogadó szervert. Tehát a robot indulásakor függetlenül az egyensúlyozást megvalósító algoritmustól indítjuk a szervert, amely mindaddig él, míg a robot egyensúlyban van.

Klienssel való kapcsolat létrehozása után, mindaddig várja az utasításokat, míg a kapcsolat él. A szerver egyszerre egy klienstől fogad utasításokat, tehát mindaddig, amíg egy adott klienssel kapcsolata van, addig más kliens nem tud kapcsolódni a szerverhez.

A strukturált adatok kinyerése érdekében a Google Protocol Buffers által legenerált hozzáférési osztályt használjuk. Az így megkapott értékeket átadjuk a szabályzó algoritmusnak, amely a hiba számításánál az elvárt értékeknek tekinti a robot dőlési szögét, szögsebességét, pozíóját, sebességét illetően.

4.2.3. Automatikus kapcsolódás

A robot és a telefonos alkalmazás közti kapcsolat automatikus létrehozását, amely felgyorsítja, illetve megkönnyíti a felhasználónak a kapcsolódást a robothoz, a Cling UPnP (Universal Plug and Play) [15] könyvtár által valósítottuk meg.

A Cling UPnP könyvtár az SSDP (Simple Service Discovery Protocol)⁵ protokollt használja.

Szerver oldalon annak érdekében, hogy a kliens lássa a hálózaton, úgymond kell "reklámoznia" magát. Ahhoz, hogy a kliens megtalálja az általa keresett szervert, fel kell fedezze a hálózaton levő szervereket, tehát "hallgatóznia" kell. Az ehhez szükséges metódusokat tartalmazza az UPnpService objektum. A szerver létre kell hozzon egy UPnpService-t, amelyet futtatnia kell. Függetlenül attól, hogy szerver vagy kliens oldalon vagyunk, erre szükségünk van. Ezt követően hozza kell adjon egy eszközt (Device-t), amelynek meg kell adni egy egyedi azonosítót, típust, amely a verzió, leírást és a szolgáltatást. Ezek mellett egy icon-t is meg lehet adni. A szolgáltatás definiálása (4.6 ábrán látható) esetén szükséges egy egyedi azonosító és típus megadása, ez esetben a @UpnpServiceId és a @UpnpServiceType meta-annotációkat használhatjuk. A változók, illetve a metódusok esetén is szükséges az annotációk használata. Változók esetében a @UpnpStateVariable használható. Metódusok esetén @UpnpAction-el jelezük, hogy metódus. Ezen kívül, ha metódus, akkor, ha van bemeneti paramétere a @UpnpInputArgument használható, illetve, ha van visszatérési értéke a @UpnpOutputArgument annotáció használható.

Lehetséges a metaadatok megadása annotációk helyett XML formátumba is.

5. https://en.wikipedia.org/wiki/Simple_Service_Discovery_Protocol

4. FEJEZET: MEGVALÓSÍTÁS

4.6. Listing. Szolgáltatást definiáló osztály

```
import org.fourthline.cling.binding.annotations.*;
2 @UpnpService(
    serviceId = @UpnpServiceId("MyService"),
    serviceType = @UpnpServiceType(value = "MyService", version = 1)
)
7 public class Service {
12     @UpnpStateVariable(defaultValue = "0", sendEvents = false)
        private boolean target = false;
17     @UpnpStateVariable(defaultValue = "0")
        private boolean status = false;
22     @UpnpAction
        public void setTarget(@UpnpInputArgument(name = "NewTargetValue")
                                boolean newTargetValue) {
            target = newTargetValue;
            status = newTargetValue;
        }
27     @UpnpAction(out = @UpnpOutputArgument(name = "RetTargetValue"))
        public boolean getTarget() {
            return target;
        }
        @UpnpAction(out = @UpnpOutputArgument(name = "ResultStatus"))
        public boolean getStatus() {
            return status;
        }
}
```

Kliens oldalon létrehozunk egy RegistryListener-t, amely esemény figyelő, értesül, ha valamely szerver állapotát módosult a hálózaton. Ha a számunkra megfelelő állapotba került a szerver, akkor leellenőrizhető, hogy rendelkezik-e az általunk keresett szolgáltatás azonosító id-val. Abban az esetben, ha rendelkezik, akkor a szolgáltatásban a meta-annotációkkal felannotált tevékenységeket futtathatjuk az ActionInvocation API segítségével.

A projektünk során a Cling UPnp-t felhasználva a robot létrehoz egy olyan szolgáltatást, amelytől le tudjuk kérni az IP címét. Az alkalmazás azonosítja ezt a szolgáltatást az ID-ja szerint és lekéri az IP címet, amelyet felhasználva létrehozza a kapcsolatot.

4.3. Egyensúlyozás problémái

A dolgozat alapjául a Gilyen Hunor által elkészített projektet vettük és ebből indultunk ki, hogy megvalósítsuk a robot négy irányba való irányítását úgy, hogy megtartsa közbe az egyensúlyi állapotát.

Az eredeti projekt struktúrája nem volt előnyös számunkra. E struktúra tartalmazott egy központi osztályt, amelyben definiálva volt egy belső osztályt. Ezen osztályon belül került sor az egyensúlyozást megvalósító PID szabályzó implementálása, amely egy szálon volt elindítva, tehát implementálta a Runnable interfészt. A robot szenzoraitól lekérdezett értékek két modell segítségével voltak eltárolva. A program futtatásakor egy Thread⁶ objektum példányosítódik, amely konstruktorának át lesz adva a Runnable interfészt implementáló szabályzó algoritmus.

A projekt struktúrájának javítása érdekében különválasztottuk a PID szabályzó algoritmust és a giroszkóp szenzor olvasásához szükséges metódusokat. E két osztályt külön szálon indítjuk el.

A szinkronizálást a CyclicBarrier⁷ segítségével valósítottuk meg, amely a Java 7 része. A pél-

6. <https://docs.oracle.com/javase/7/docs/api/java/lang/Thread.html>

7. <http://tutorials.jenkov.com/java-util-concurrent/cyclicbarrier.html>

4. FEJEZET: MEGVALÓSÍTÁS

dányosításakor (4.7 ábrán látható) megadható, hogy hány szállal szeretnénk dolgozni. Szinkronizálás során a `await` metódus hívással a szál várakozik mindaddig, míg az összes szál nem hívja meg ezt a metódust. Mikor egy szál meghívja ezt a metódus akkor a `CyclicBarrier` növeli a számlálóját és minden addig várakozik a szál, amíg ez a számláló nem éri el a példányosításkor megadott értéket. Valamint létrehoztunk egy központi osztályt, amely feladata a szálak kezelése és a szenzorokhoz, illetve motorokhoz való hozzáférést biztosító objektumok példányosítása. Az eredeti projekt struktúráját átalakítva, a 4.7 ábrán látható osztály diagrammal ábrázolt struktúra alakult ki.

4.7. Listing. `CyclicBarrier` példányosítása

```
CyclicBarrier barrier = new CyclicBarrier(2);
```

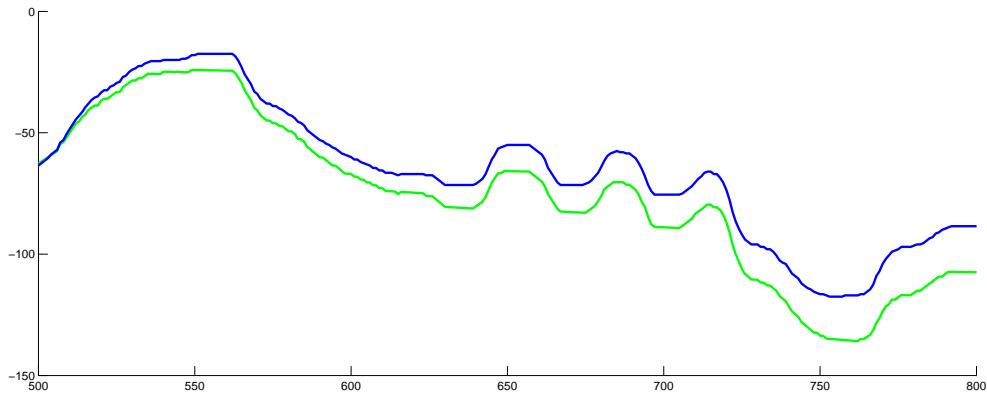
Kezdetben az volt az elképzelésünk, hogy különválasszuk a jobb és bal motorokat irányító algoritmust, ezáltal egymástól függetlenné téve őket. Ehhez szükséges volt a szabályzó algoritmus általánosítása, illetve a két motor szinkronizálásának megoldása. Mivel külön szalon fut a giroszkóp kezelése és külön-külön a két motort szabályzó algoritmusá ezért minden iterációban a giroszkóp leolvasását egyszerre kellett végrehajtsák, különben más-más értékekkel dolgoztak, mely egyensúlyvesztést okozott. Ehhez szükséges volt az előbb létrehozott `CyclicBarrier` konstrukturának átadott értékének növelése eggyel, mivel most már külön-külön kérte le a giroszkóp szenzor értékét a jobb és bal motorokat vezérlő szál. Emellett még egy újabb `CyclicBarrier` bevezetése volt szükséges, hogy a motorokat is szinkronizáljuk egymáshoz. E módosítást követően a robot rövid időn belül elvesztette az egyensúlyát.

Mivel a robot LCD kijelzője nem megfelelő méretű, hogy tesztelésre alkalmas adatokat jelenítsünk meg futás közben, ezért szükség van a fájlba való kiíratásra és ezt követően az értékek értelmezésére Matlab segítségével. E módszer igen csak időigényes és nehézkes, mivel a túl sok fájlba való írás leterheli a robot processzorát és növelődik egy iterációnak a végrehajtási ideje. Az iterációként eltelt idő szerint számítuk a robot sebességét és PID integrált, illetve derivált tagját is. Tehát, ha jelentősen tolódik egy iteráció lefutási ideje, akkor az kritikusan befolyásolja az algoritmus működését.

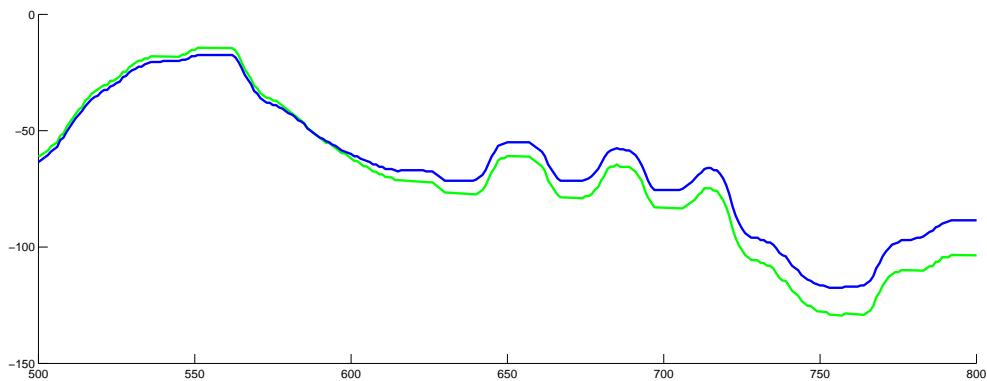
Az előbb említett tesztelés esetén felmerülő probléma megoldására naplázási keretrendszer alkalmaztunk. Az Slf4J-LOG4J naplázási keretrendszer választottuk, az Slf4J [14] egy absztrakciós szint, amely a LOG4J [4] naplázási keretrendszer implementációja fölött áll. A konfigurációs XML állományban, szűrök használatával, lehetőségünk van a naplázási szintek (debug, info, warning, error) elkülönítésére.

A szinkronizálás helyességének tesztelése után, a PID szabályzó bemeneti hibáját meghatározó négy komponensnek a kiszámítását teszteltük. Az eredeti projekt szabályzó algoritmusá esetében a robot sebességének és pozíciójának kiszámításakor átlagolva volt a két motor fordulatszáma, annak érdekében, hogy kiszűrje a zajokat. Esetünkben ez módosult és elhagytuk az átlagolást. Ennek eredményeként, amikor egy adott iterációban minimális különbség lépett fel a két motor fordulatszáma között, akkor ez nagymértékben befolyásolta a sebesség és pozíció kiszámítását. Mivel a fordulatszám különbözőt, ezért a sebesség és a pozíció is. E különbség iterációként növekedett. A sebesség és a pozíció a PID bemeneti hibájának a két komponense, ezért a kimeneti érték is különbözőt, amely meghatározta a motorokra adott erő nagyságát. Tehát a motorok fordulatmérő szenzorjainak zajos mérései miatt olyan eltérés keletkezett

4. FEJEZET: MEGVALÓSÍTÁS



4.5. ábra. Bal motor fordulatszám összehasonlítása a bal és jobb motor fordulatszám átlagával.
Az x tengelyen jelöljük a iterációk számát, az y tengelyen a fordulatszám értékének mértékét.



4.6. ábra. Jobb motor fordulatszám összehasonlítása a bal és jobb motor fordulatszám átlagával.
Az x tengelyen jelöljük a iterációk számát, az y tengelyen a fordulatszám értékének mértékét.

a két motort kezelő szabályzó közt, hogy a robot elvesztette egyensúlyi állapotát.

E probléma megoldására, mivel iterációinként nőtt az eltérés a két párhuzamosan futó szabályzó közt, próbáltuk súlyozni az aktuális és az azelőtti mért fordulatszámot, hogy megközelítsük az átlagolt fordulatszámot. Az így megkapott értékre számoltuk ki a robot sebességét és pozíóját. A legjobb megközelítését, amelyet az átlagolt értékkal hasonlítottunk össze, az látható a 4.5 és a 4.6 ábrákon. A kék szín jelöli az adott motor fordulatszámát, illetve a zöld jelöli a motorok átlagolt fordulatszámot. De még ez a közelítés sem volt elegendő, hogy a robot ne veszítse el az egyensúlyát.

4. FEJEZET: MEGVALÓSÍTÁS

4.3.1. Irányítás megvalósítása és egyensúly megtartása

Miután sikeresen tudtunk kommunikálni a robottal, nekikezdtünk az irányítás megvalósításának. Rövid időn belül problémába is ütközöttünk.

Az első próbálkozásunk során, a motoroknak átadott értéket módosítottuk az irányításnak megfelelően. Jobbra és balra fordulásnál, az iránynak ellentétesen, a megfelelő motornak beállítandó értékét növeltük. Vagyis jobbra fordulásnál növeltük a bal motornak leadott értékét és balra forduláskor a jobb motornak. Ebben az esetben a robot nem a saját tengelye körül fordult, hanem egy hosszabb ívet írt le és eközben a PID szabályzó próbálta kompenzálni a hibát, amit érzékelte. Hiszen hirtelen megnőtt a robot sebessége, hirtelen nagyon változott a pozíciója és megnőtt a dőlési szöge, valamint a szögsebessége. Ennek eredményeként a robot egy-egy pillanatban lelassult, majd felgyorsult. E probléma kiküszöböléssére nem csak a fordulás irányának az ellentettjének megfelelő motor leadott értékét növeltük, hanem ugyan azt az értéket ellentétes előjellel hozzáadtuk a másik motorhoz, vagyis jobbra kanyarodásnál a jobb motor értékéből kivontuk és bal motornak az értékéhez hozzáadtuk. Így a saját tengelye körül fordult. Emellett a szenzorok sem érzékeltek olyan nagy mértékű változást, ami hirtelen befolyásolná a hiba komponenseinek értékét.

Az előre és hátra való irányítás esetén is hasonlóképpen próbálkoztunk, mint a fordulás estében, de ez esetben azonos előjellel módosítottuk a kerekekre leadandó értéket. Ezért kis mértékben elmozdult az iránynak megfelelően, majd a PID szabályzó korrigálta a hibát.

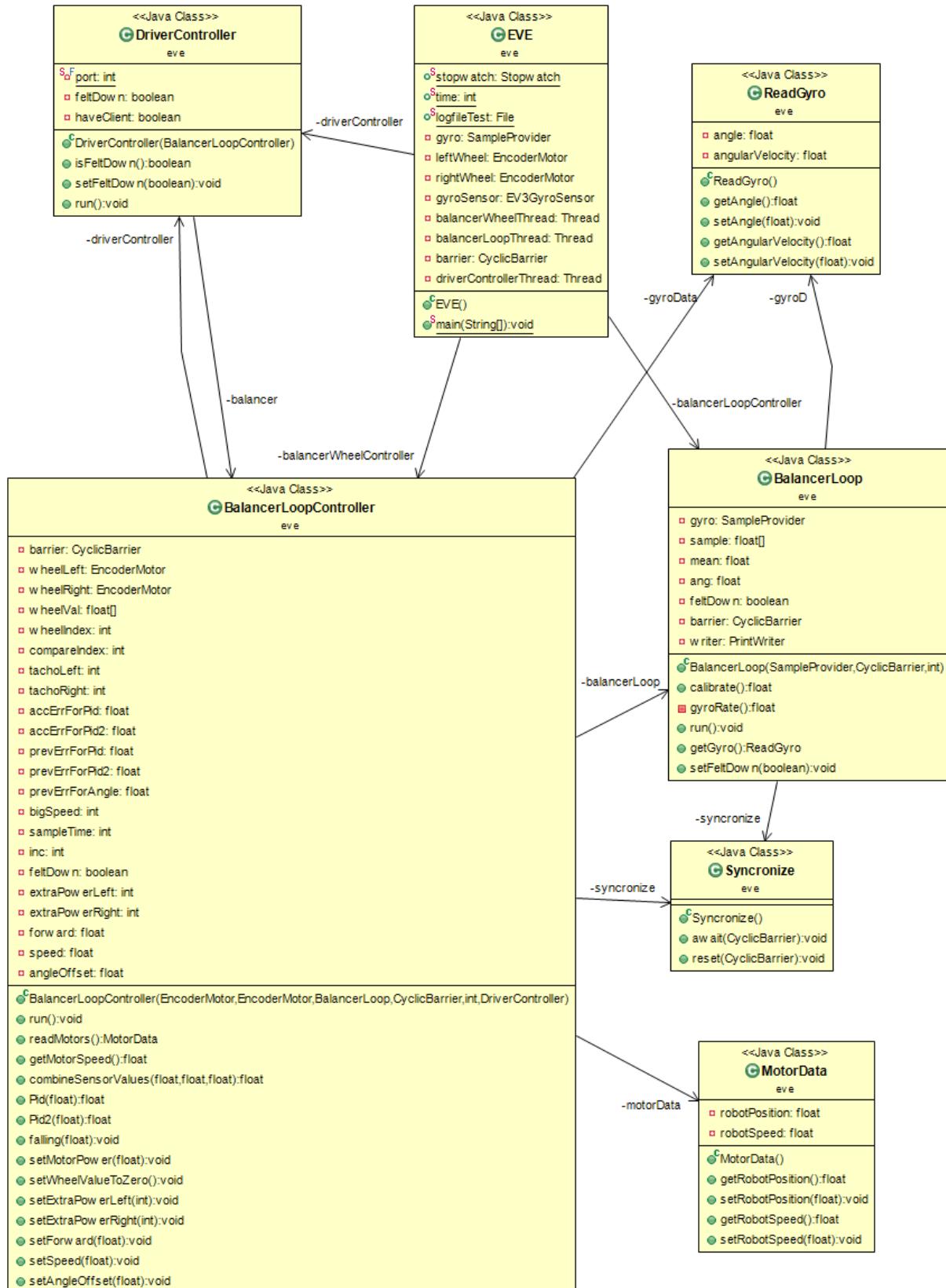
Az első próbálkozásunk után a hiba négy komponensének (szög, szögsebesség, pozíció, robot sebessége) a módosításával próbáltuk elérni, hogy a robot előre vagy hátra fele irányítható legyen. Elképzelésünk szerint próbáltuk a giroszkóp által leolvasott dőlési szög értékét módosítani sikertelenül. Majd a későbbiekben sikerült részleges eredményt elérni. A robot pozíciójának megfelelő változtatásával képesek voltunk a megfelelő irányba elmozdítani a robotot úgy, hogy megtartsa egyensúlyi állapotát. E próbálkozás esetén a pozíció változtatásához bevezettük egy ellensúlyozást (`offset`), amely értékét folyamatosan növeltük vagy csökkentettük és hozzáadtuk az aktuális pozícióhoz. Így az egyensúly megtartásához ellensúlyoznia kellett a pozíció értékét. Tehát, ha növeltük az `offset` értékét a robot hátra ment, ha pedig csökkentettük, akkor előre ment. Ebben az esetben nem mindig volt elég, ha csak egy konstanssal növeltük vagy csökkentettük az `offset` értékét, így nem volt meg az irányítás alatt egy konstans sebesség. Próbáltuk a szenzor által mért sebesség növelését, de nem oldotta meg a felmerült problémát, csak az előbb említett viselkedést gyorsabban hajtotta végre.

A következő próbálkozásunk esetén ugyancsak a hibát módosítottuk, csak nem az aktuális értékeket, hanem bevezettük az elvárt értéket, ami eddig nulla volt. Emellett rájöttünk, hogy előre és hátra irányítás esetén a hibába szereplő pozíció tagot ki kell vegyük. A pozíció tag esetén a szabályzó arra törekszik, hogy egy helyben maradjon a robot. Tehát, ha előre vagy hátra akarjuk mozdítani a robotot, akkor az irányítás idejére ki kell vegyük ezt a tagot. Ehhez egy újabb PID szabályzó bevezetése volt szükséges, amelynek a bemeneti értéke a pozíció volt, értelemszerűen a másik PID szabályzó hibáját meghatározó komponensek közül kivettük. Tehát, most van egy PID1 és egy PID2 szabályzónk. Irányítás hiányában a két szabályzó kimenetét összegeztük (4.8 ára) és ez az érték lett a véleges kimenet, amely a motorokra

4. FEJEZET: MEGVALÓSÍTÁS

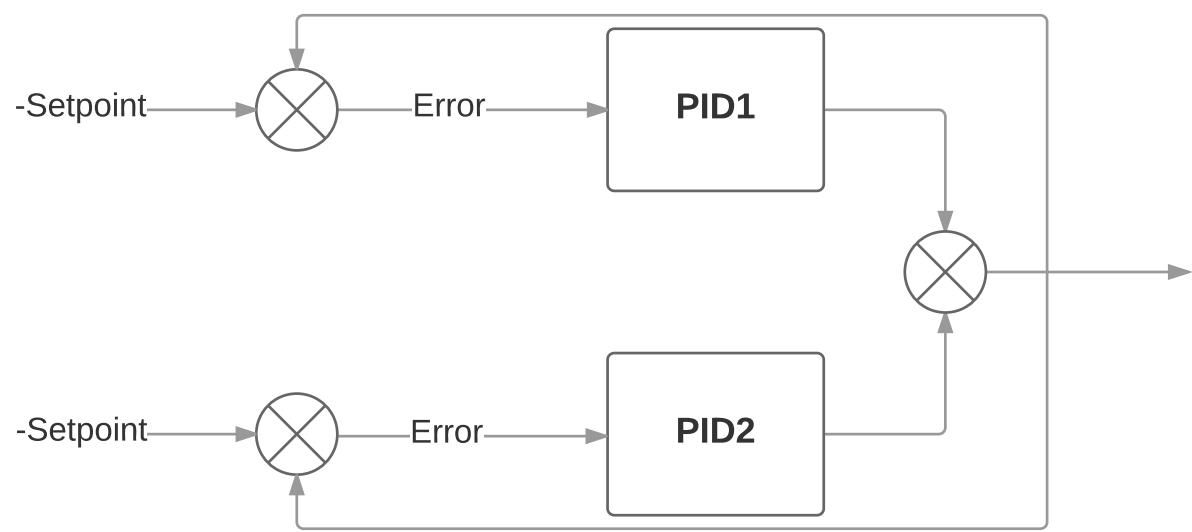
leadott erőt határozta meg. Irányítás alatt a PID2 szabályzót, amelynek a pozíció megtartásának szabályozása a feladata, nem vettük figyelembe. A PID2 elhagyása, illetve visszavétele kisebb akadály volt, mivel hirtelen csökkent vagy nőtt a szabályzó rendszer kimeneti értéke a derivált tag miatt, hiszen a hiba hirtelen megugrott. E probléma megoldására próbáltuk több iterációra lebontani a különbséget, de ez nem volt elégsges. Megoldása, ha egyensúlyi állapotból irányított állapotba váltunk, akkor az aktuális állapotot vehetjük kiindulási pontnak, mivel a robot egyensúlyban van. Visszaváltás esetén a motorok fordulatszám mérőjét szükséges lenullázni, különben, a PID2 szabályzó túllendülést okoz. Túllendülés alatt azt értem, hogy motoroknak leadandó érték, amely a motor erejét határozza meg. Oly nagy lesz, hogy hirtelen nagyon változik a robot dőlési szöge és elveszti egyensúlyát.

4. FEJEZET: MEGVALÓSÍTÁS



4.7. ábra. A roboton futó alkalmazás osztály diagramja

4. FEJEZET: MEGVALÓSÍTÁS



4.8. ábra. PID1 és PID2 szabályzók összekötése

5. fejezet

Következtetések és továbbfejlesztési lehetőségek

A dolgozat során elkészült egy olyan projekt, amely lehetővé teszi a robot irányítását hálózaton keresztül, Android alkalmazás segítségével. Az alkalmazás lehetőséget ad az automatikus kapcsolat létrehozására a robottal.

Az adatok szerializációját megvalósító Google Protocol Buffers több továbbfejlesztési lehetőséget ad. Az általunk definiált adatstruktúra könnyen bővíthető, így lehetőség van arra, hogy a robot adatokat küldjön az alkalmazásnak a jelenlegi állapotáról, amelyeket az alkalmazás megjelenítene a telefonon .

Az EV3 ultrahang szenzorja és infravörös szenzorja lehetővé teszi, hogy távolságot mérjünk .Továbbfejlesztési lehetőségekkel ezek szenzorok egyikét felhasználva lehetséges, hogy a robot irányítás alatt, ha akadályt észlel, akkor megáll, figyelmeztetést küld a felhasználónak és lehetséges irányváltoztatást, annak érdekében, hogy tovább tudjon menni.

Továbbfejlesztés lehetőségekkel még a fordulás módosítását, egy újabb szabályzó rendszer bevezetésével, hasonlóan, mint az előre-hátra irányításnál, valamint az előre-hátra irányítás finomítását és optimalizálását.

Irodalomjegyzék

- [1] EV3 vezérlőegység. <http://www.lego.com/en-us/mindstorms/products/mindstorms-ev3-31313>, . Utolsó megtekintés dátuma: 2016-05-16.
- [2] EV3DEV firmware hivatalos oldala. <http://www.ev3dev.org/>, . Utolsó megtekintés dátuma: 2016-06-06.
- [3] A LEGO történelmi idővonala. http://www.lego.com/en-us/aboutus/lego-group/the_lego_history/1930. Utolsó megtekintés dátuma: 2016-05-09.
- [4] LOG4J hivatalos oldala. <http://logging.apache.org/log4j/2.x/>. Utolsó megtekintés dátuma: 2016-06-04.
- [5] A LEGO MINDSTORMS EV3 hivatalos oldala. <http://www.lego.com/en-US/mindstorms/?domainredirect=mindstorms.lego.com>, . Utolsó megtekintés dátuma: 2016-05-16.
- [6] A LEGO MINDSTORMS történelme. <http://www.lego.com/en-us/mindstorms/history>, . Utolsó megtekintés dátuma: 2016-05-09.
- [7] Az NXT vezérlőegység. <http://shop.lego.com/en-CA/NXT-Intelligent-Brick-9841>, . Utolsó megtekintés dátuma: 2016-05-16.
- [8] NXT összehasonlítása az EV3-al. <http://botbench.com/blog/2013/01/08/comparing-the-nxt-and-ev3-bricks/>, . Utolsó megtekintés dátuma: 2016-05-16.
- [9] A Google Protocol Buffers hivatalos oldala. <https://developers.google.com/protocol-buffers/>. Utolsó megtekintés dátuma: 2016-06-01.
- [10] ROBOTC firmware hivatalos oldala. <http://www.robotc.net>. Utolsó megtekintés dátuma: 2016-06-06.

IRODALOMJEGYZÉK

- [11] Az RXC vezérlőegység. http://lego.wikia.com/wiki/Mindstorms_RCX. Utolsó megtekintés dátuma: 2016-05-16.
- [12] Az EV3 szenzorairól szóló információk. <https://www.intorobotics.com/sensors-lego-mindstorms-ev3-features-comparison/>, . Utolsó megtekintés dátuma: 2016-05-17.
- [13] Az EV3 szenzorairól és motorjairól szóló információk. http://fll.memphisfirstteams.org/Files/Workshop_4-16-16/EV3-Motors-Sensors.pdf, . Utolsó megtekintés dátuma: 2016-05-17.
- [14] Slf4J hivatalos oldala. <http://slf4j.org/>. Utolsó megtekintés dátuma: 2016-06-04.
- [15] Cling UPnP könyvtár hivatalos oldala. <http://4thline.org/projects/cling/>. Utolsó megtekintés dátuma: 2016-06-05.
- [16] Zoltán, S. A digitális PID szabályzó. Technical report, Budapesti Műszaki és Gazdaság tudományi Egyetem, 2008.