

Óbudai Egyetem

Neumann János Informatikai Kar



TUDOMÁNYOS DIÁKKÖRI DOLGOZAT

MOTORSZABÁLYOZÁS NEURÁLIS HÁLÓZATTAL

Szerző:

Mavridisz Vaszilisz

mérnök informatikus alap szak,
IV. évfolyam

Konzulens:

Somlyai László

tanszéki mérnök

Tartalom

TARTALOM	2
MOTORSZABÁLYOZÁS NEURÁLIS HÁLÓZATTAL.....	3
ABSZTRAKT	3
BEVEZETÉS	3
SZABÁLYOZÁS	5
SZEKVENCIA (ALAPJEL KÉPZŐ).....	6
SZABÁLYZÓ	7
VÉGREHAJTÓ ÉS BEAVATKOZÓ	8
SZABÁLYZOTT SZAKASZ	8
SZABÁLYOZÁS BEÁGYAZOTT RENDSZEREKEN	9
FOLYAMATOK IDŐZÍTÉSE	10
KOMMUNIKÁCIÓ	12
MINTAVÉTELES PID-SZABÁLYOZÁS	13
OPTIMALIZÁLÁS	13
AZ OPTIMALIZÁLÁS EREDMÉNYE	14
NEURÁLIS HÁLÓZATOK	14
BEMENETI ADATOK	15
RÉTEGEK.....	16
TANÍTÁS	17
KIMENET JAVÍTÁSA	17
ÖSSZEGZÉS.....	19
FELHASZNÁLT IRODALOM	19
I. MELLÉKLET	21
II. MELLÉKLET	22
III. MELLÉKLET	23
IV. MELLÉKLET.....	26

Motorszabályozás neurális hálózattal

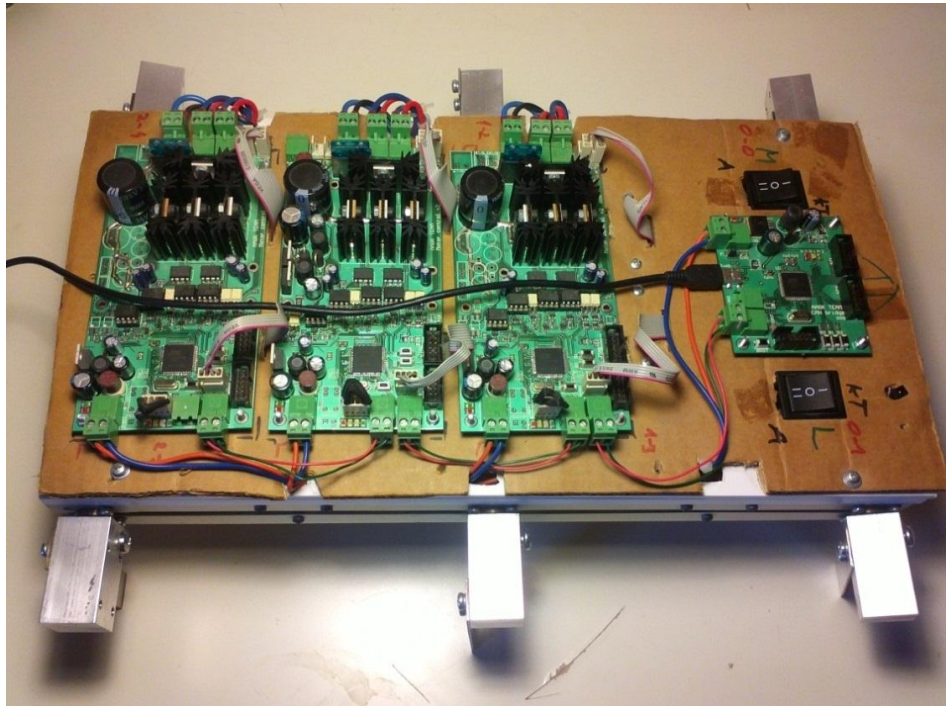
Absztrakt

A dolgozat hatlábú, lépegető roboton megvalósított rendszert mutat be. A kutatás célja a kis teljesítményű, elosztott rendszerek lehetőségeinek feltárása a robotirányítás terén. A kutatás során kidolgozásra került hagyományos valamint neurális elven tanuló és működő vezérlési rendszer.

A dolgozat konklúziójaként e szabályozási megoldásokat vizsgáltuk azonos feltételek mellett több alkalmazási területen. A kapott eredményeket több szempontból is összehasonlítottuk. A megvalósítás során egy többelemű mikrokontrolleres irányító rendszer készült el, amely számítógéppel együttműködve végzi el a tanulási folyamatot. Részletesen bemutatjuk a neurális hálózat felépítését, implementálását mikrokontrollerre és a tanítási folyamat speciális megvalósítását.

Bevezetés

A robot, amelyen a szabályozást szeretnénk megvalósítani hexapod típusú. Ezek a robotok 6 mozgatható lábbal haladnak, miközben legalább 3 folyamatosan érinti a talajt. A robot kialakításának (1. ábra) ötletét a Boston Dynamics által készített rHex robot [1] adta. A megvalósítás során egy már korábbi robot elektronikáját használtuk [5].



1. ábra: A robot

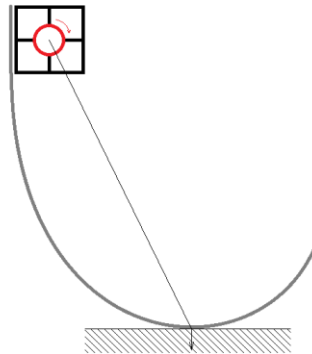
A lábakkal meghajtott robot esetén a lábak összehangolt mozgására van szükség, ezért összetett vezérlést igényel az eszköz. A vezérléshez a robotnak összetett parancsokat kell teljesítenie:

- lábak kalibrálása
- fekvő helyzetből álló helyzetbe való állás
- egy helyben állás
- álló helyzetből fekvő helyzetbe állás
- egyenes irányú haladás
- kanyarodás haladás közben, meghatározott szögben
- elfordulás egy helyben

Ezek az összetett parancsok a lábak forgatásának összehangolt sorozataként írhatók le. A mozgás során olyan összetett erők hatnak a végrehajtókra, amelyek összefüggésben vannak a robot talajhoz viszonyított helyzetével, a robot fizikai paramétereivel [2] és környezeti tényezőkkel [3].

A fizikai paraméterek közé tartozik a lábak formája, egymáshoz viszonyított elhelyezkedésük, a vezérelt motor fizikai paraméterei. Rövidtávon ezek állandónak tekinthetők, hosszútávon azonban a kopás és korrózió miatt változhatnak. Az ívelt formájú láb (2. ábra) egyenletesebb mozgást tesz lehetővé, de speciális szabályozást igényel, mivel szögtől függően eltérő hosszúságú erőként viselkedik, így a motorokra eső terhelés is eltérő.

Ennek érdekes változata a SprawlHex [6] robot, amely a lábakat mozgató tengelyre merőlegesen is képes a robot lábait elforgatni, így lehetővé téve a még összetettebb mozgást. A motorokat vezérlő szabályozásnak a teljes erőtartományban elfogadható viselkedést kell mutatnia. A túl erős vagy a túl gyenge vezérlőjel az összehangolt mozgás folytonosságát megtörheti, a robotot letérítheti a mozgási pályájáról.



2. ábra: Ívelt láb

A környezeti tényezők rövidtávon már sokkal változatosabbak lehetnek. Ezek a tényezők a talaj minőségével és a domborzat mintájával vannak összefüggésben. Adott útvonal bejárása során sokféle tereppel kerülhet szembe, amely akár a normál mozgástól teljesen eltérő mozgatsort igényel (például lépcsőmászás). Továbbá egy mozgatsoron belül is szögtől függően, ha a talajt érinti a láb és a robot támaszkodik is azon, akkor eltérően kell viselkednie, mint amikor a levegőben forog.

Tehát látható, hogy amikor a robot szabályozását szeretnénk megvalósítani, akkor meglehetősen sokoldalú, kellően rugalmas megoldást kell találni. A cél, hogy képes legyen akár a már ismert vagy ismeretlen körülményekhez a lehető leggyorsabban alkalmazkodni, és erre emlékezni is.

Szabályozás

A robot két fő részből álló irányítási rendszer, ezek: az irányítandó rendszer és maga az irányítórendszer.

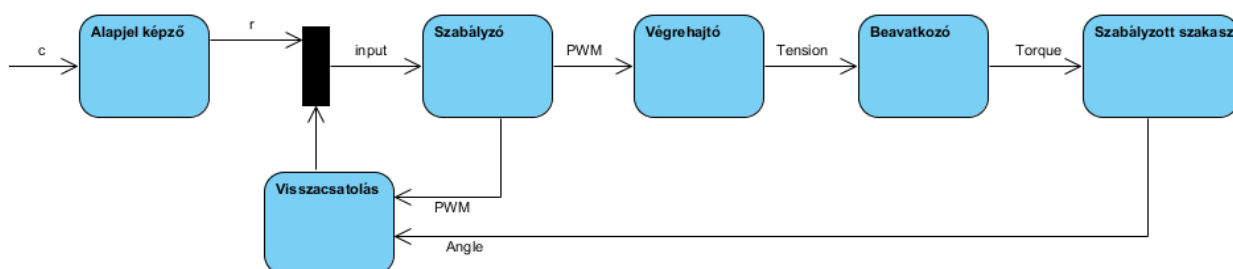
Az irányítandó rendszer az irányítórendszertől függetlenül létező műszaki eszköz, röviden „szakasznak” nevezik. A szakaszok folyamatai alatt a szakaszokban lévő energiák és információk szállítását és átalakítását értjük. Az irányítandó szakaszok különböző folyamatjellemzőkkel, folyamatváltozókkal jellemezhetők. A folyamatjellemzők esetünkben diszkrét, a motorok adott teljesítményét diszkrét értékekből álló tartományban adjuk meg. Az elfordulás visszacsatolását pedig egy relatív távadó valósítja meg, amely szintén diszkrét pozíciókat mér. A folyamat során a lábak szögének egy előre meghatározott mintát kell követniük.

Az irányítórendszer, mindazon szervek, készülékek összessége, amelyek együttműködése révén az irányítandó berendezés vagy rendszer magas színvonalú önműködő irányítása valósul meg. A vezérlés olyan irányítási tevékenység, amely során beavatkozás történik az irányítandó gép technológia folyamataiba. Az irányítási forma zárt hatásláncú, mivel tartalmaz visszacsatolást.

A szabályozás formája követőszabályozás, ebben az esetben, az alapérték üzemszerűen változik. A megvalósított szabályozás programszabályozás, mivel a vezető jelét algoritmus alapján kapja a szabályozás, ez azt jelenti, hogy egy előre definiált vezetőjel szerint történik a szabályozás. A rendszer segédenergiát igényel, mivel az elfordulást mérő távadók optikai elven működnek, aktív fényt használnak a forgás érzékelésére.

A folyamatos szabályozás esetén, a szabályozási részműveletek folyamatosan történnek. A szabályozás akkor folyamatos, ha az irányítás részműveletei (érzékelés, ítéletalkotás, rendelkezés, beavatkozás) megszakítás nélkül folyamatosan mennek végbe.

A szabályozás folytonos, a végrehajtó szerv kimeneti jele (beavatkozó jel) egy értékkészleten belül meghatározott függvénye a bemeneti jelnek (végrehajtó jel). Ez az adott művelettől függően változhat. De lehet akár PID- szabályozás vagy neurális hálózat is. Ahhoz, hogy ilyen sokféle dolgot egy rendszerben megvalósíthassunk, a hagyományos szabályozási rendszerektől annyiban térünk el, hogy a visszacsatolást kicsit tágabban értelmezve nem csupán a szabályzott érték, hanem akár a modellből eredő értékek is a szabályozás bemenetét adhatják.



3. ábra: Szabályozási kör

A 3. ábrán az irányítási rendszer felépítését mutatja be, ahol:

- **c** (vezetőjel – command variable): az összetett parancs külső alapjelként értelmezhető, amely a szabályzó referencia jele.
- **r** (alapjel – reference signal): az összetett parancsokat megvalósító összehangolt mozgássorozat időben vett értéke.
- **Input**: A szabályozás bemenete.
- **PWM** (végrehajtó jel – actuator signal): a mikrokontroller által generált jel, amellyel a motor teljesítményét lehet befolyásolni.
- **Tension** (beavatkozó jel – intervening signal): a PWM-jel kitöltési tényezőjének megfelelő feszültség a motoron.
- **Torque** (módosított jellemző – process manipulated variable): a feszültségből és a motor belső tulajdonságaiból, valamint külső tényezőkből létrejött forgatónyomaték.
- **Angle** (szabályozott jellemző – controlled variable): a láb állása, amely időben vizsgálva elfordulásként mért.

Szekvencia (alapjel képző)

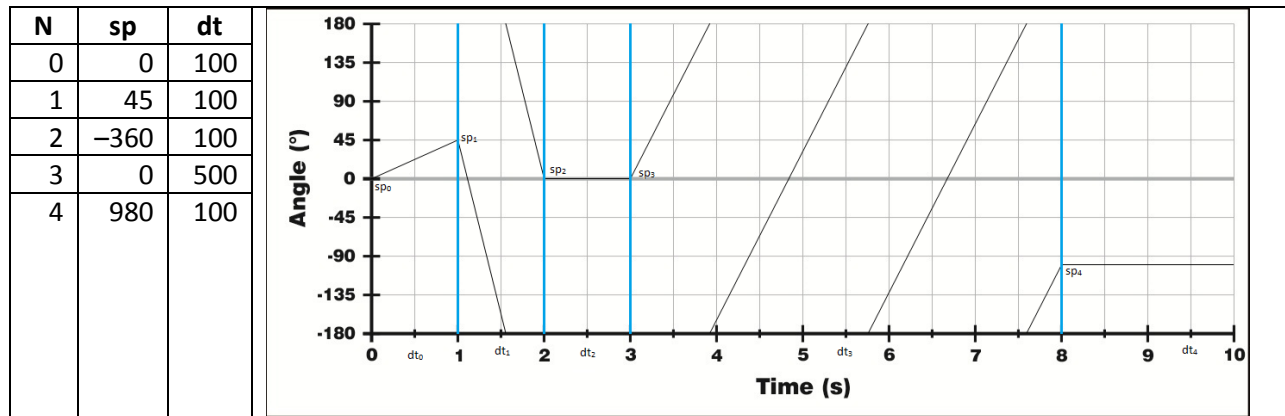
A dolgozatban a forgatási pálya és az ettől való eltérést grafikonon jelink meg. Ennek jelentése: a vízszintes tengely az időt mutatja, a függőleges pedig az elfordulás szögét. Ez a szögtartomány -180° -tól $+180^\circ$ -ig tart, a láb tengelye nincs korlátozva. Képes körbefordulni, így minden szakasz végén, amikor átlépi a határt, az értékét normalizálja a rendszer, így lesz a második lépés végén -360° -ból 0° . A visszacsatolás értékét az alapjel normalizálásával szintén eltolja.

A szekvenciák leírására láncolt listában két értéket tárolunk: az egyik a pont érkezési szöge (sp) a másik az utána következő szakasz hossza (time). Két szekvenciapont között idő alapon határozzuk meg a láb kívánt helyzetét, ezt idővel arányos átmenet alapján tudjuk kiszámolni (1).

$$r(t) = sp_{n+1} * \frac{time_n - dt}{time_{n+1} - time_n} + sp_n * \frac{dt}{time_{n+1} - time_n} \quad (1)$$

Ahol:

- $r(t)$: a láb szögének időben vett értéke
- sp_n : a szakasz elején vett pozíció
- sp_{n+1} : a szakasz végén vett érték
- dt : a szakasz kezdete óta eltelt idő
- $time_n$: a pont abszolút ideje

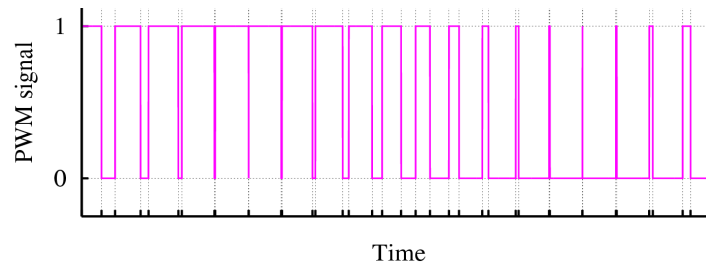


4. ábra: Szekvencia példa

Amikor a pontot kiindulási pontként kezeljük, akkor a $[-180^\circ; +180^\circ]$ tartományban normalizált értéket vesszük alapul. Ha ez a pont célértékként -360° , akkor kiindulási értékként, 0° -ként kezeljük. Célértékként elméletileg bármekkora elfordulást meg lehet adni, akár a 360° többszörösét is. Ebben az esetben a pálya áthalad a grafikon tetején, és az alján tér vissza, ahogy az a 4. ábrán látható.

Szabályzó

A digitális pozíciószabályozás általunk használt eszköze a PWM (Pulse Width Modulation – impulzus szélesség moduláció). Működési elve: a motor állandó frekvenciájú és állandó amplitúdójú négyszögjelet kap, amelynek a kitöltési tényezőjének a megváltoztatásával lehet a motort vezérelni. A 5. ábra mutatja a jelalakot. Minél kisebb a kitöltési tényező, annál kevesebb ideig kap a motor feszültséget, így lassabban forog, de ahogy növeljük a kitöltési tényezőt, tovább kap feszültséget, ezért egyre gyorsabban forog.

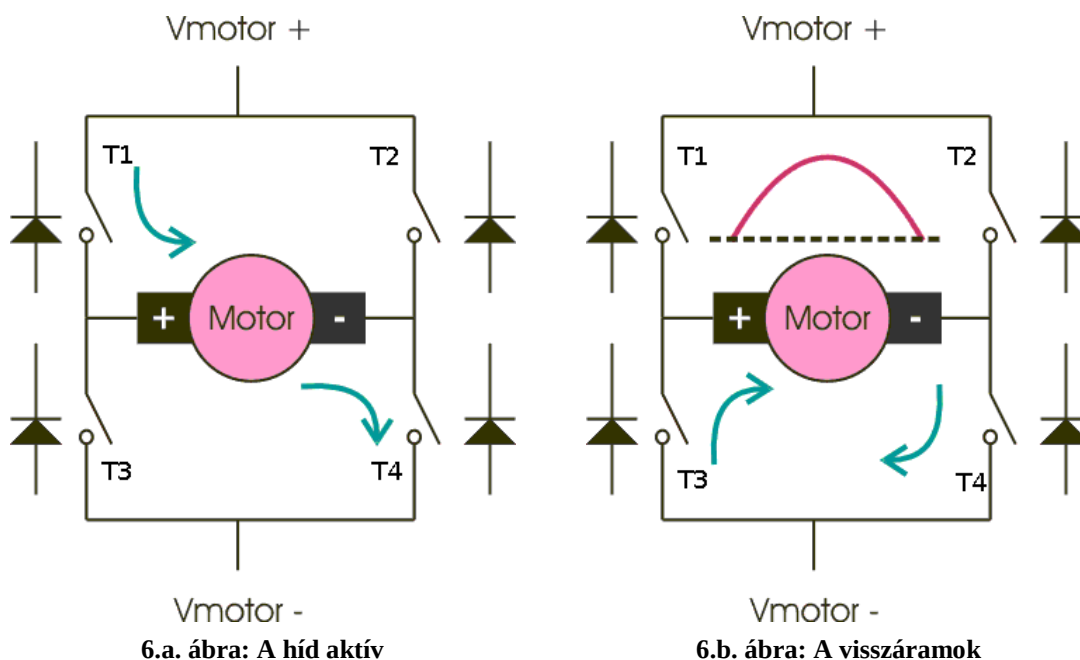


5. ábra: PWM-jel

Végrehajtó és beavatkozó

A generált PWM-jel mellett a feszültség irányát is meg kell határozni, ehhez H-hidakat használunk. A H-híd két engedélyező bittel kapcsolható, ha ellentétes értékűek ezek a bitek, akkor a motor forgási irányát határozzák meg, ahogy az a 6. ábrán látható. Amennyiben mindkettő kikapcsolt állapotban van, akkor a motor kikapcsolva szabadon elforoghat, ha mindkettő be van kapcsolva, akkor a motor a 8. ábrán látható módon zárt állapotba kerül, és fékként viselkedik. Ez utóbbi állapotban tilos PWM-jel kitöltési tényezőjének 0%-tól eltérő értéket adni, ugyanis az a táp rövidre zárását eredményezi.

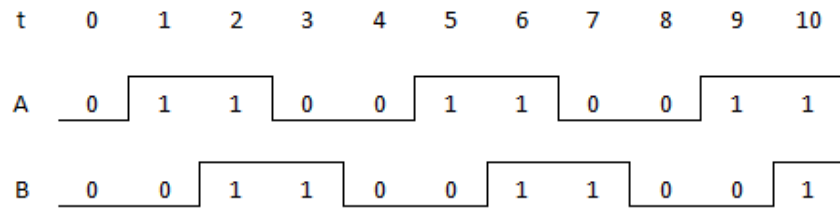
A láb mozgatását Faulhaber motor végzi. A motor tengelye és a láb tengelye közt áttétet keresztül adódik át az erő, így felerősítve a motor forgatónyomatékát.



Szabályzott szakasz

A láb helyzetét a motor tengelyére kötött kétcsatornás távadó követi. Ezen a motoron egy teljes kört 1288 lépésre lehet osztani. Mivel az elfordulás jele kétcsatornás, ezért nem csupán a forgást tudjuk követni, de az irányát is. A távadó két jelét a mikrokontroller olyan lábaira kötöttem be, amelyek logikai változása megszakítást képesek kiváltani a program futása során.

A forgás során egy kétbites, a 7. ábrán látható Gray kódot kapunk az A és a B csatornán. A motor tengelyforgásának visszacsatolásához ezeket a jeleket kell figyelni.



7. ábra: Gray kód

A jelek feldolgozása során mindkét jel logikai változása megszakítást eredményez. Ennek táblája a 8. ábrán látható, például ha az A csatorna változása megszakítást eredményez, akkor mindkét csatorna értékét kiértékeli a rendszer. Amennyiben a csatornák értéke megegyezik, a forgás iránya negatív, ha eltérők, akkor pozitív. Ugyan ilyen feldolgozásra kerül a B csatorna is. Ennek a módszernek köszönhetően minden megszakítás lépteti a számlálót, így elérve a maximális felbontást.

A / B	0	1
0	-	+
1	+	-

a)

A / B	0	1
0	+	-
1	-	+

b)

8 ábra: A megszakítás igazságtáblái

Szabályozás beágyazott rendszereken

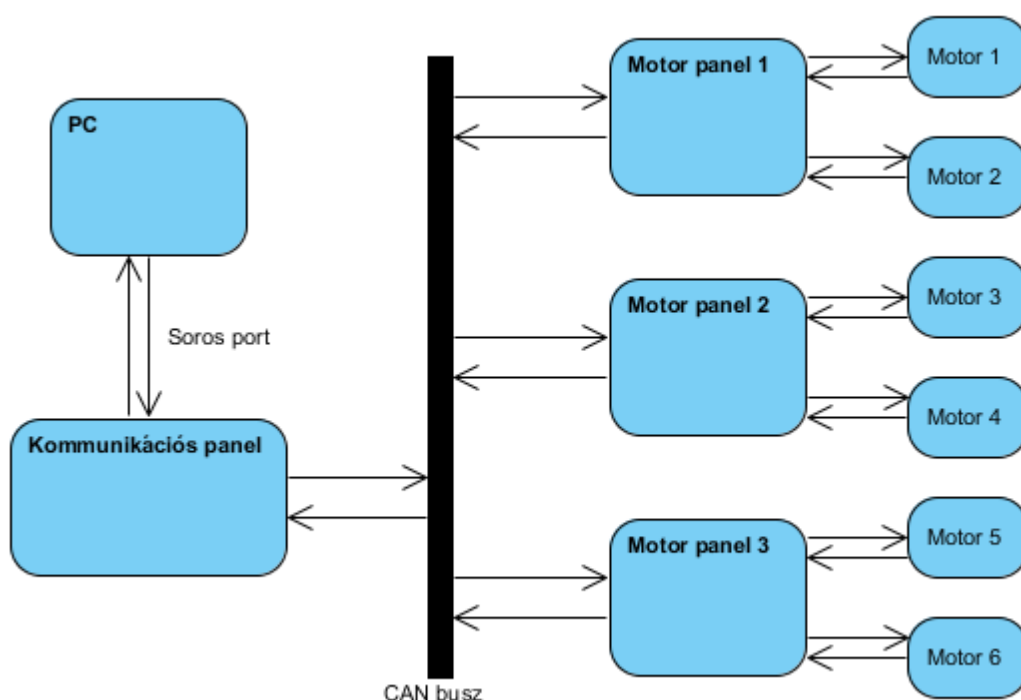
Összesen hat, nagy teljesítményű motort szeretnénk meghajtani, és a visszacsatolást érzékelni. Ez egyetlen processzor számára túl nagy terhelést jelent, a sok megszakítás akadozáshoz vezethet. A sok eszköz sok kimenetet igényelne a processzoron. Ezért célszerű olyan több részegységből álló elosztott rendszert építeni, amelyben minden egység saját vezérlővel rendelkezik és az egyes számító egységek között a feladatok megoszlanak. Ez viszont plusz feladatként azt rója ránk, hogy a kommunikációs csatornákat kell kiépítenünk az egyes egységek között.

A megvalósítás során egy korábbi TDK -ra épített robot elektronikáját használtuk fel. Abban a rendszerben két motor meghajtása volt a feladat, egy kommunikációs és egy motorpanellel oldottuk meg. Itt most hat motor meghajtása a feladat. Mivel a CAN-busznak köszönhetően ilyen irányban jól skálázható a rendszer, így három motorpanel, és egy kommunikációs panel összekötésével megvalósítható a robot irányítási rendszere a 9. ábrán látható módon. A paneleken található mikrokontroller típusa AT90CAN128, amely két beépített CAN-busz kezelőt is tartalmaz. A rendszer három szintre lett osztva, ezek a szintek kommunikációs csatornákon keresztül állnak kapcsolatban.

Az első szinten a PC áll, feladata az összetett parancsok sorozatának előállítása, és a futás folyamatának ellenőrzése, értékelése, az optimalizálás segítése.

A köztes szinten a kommunikációs panel áll. Ez teremti meg a kapcsolatot az irányítási rendszer és a külvilág között. A motorvezérlők egymás közti kommunikációját is ez az elektronika kezeli. A PC-vel soros porton keresztül kommunikál, az elektronikák pedig egymás közt CAN-buszon. Ezen a panelen fut az elsődlegesnek tekinthető időzítő, amelynek értékével bizonyos időközönként a többi panel időzítőjét szinkronizáljuk.

A motorvezérlő-panel két nagy teljesítményű H-hídon keresztül szabályozza a motorokra adott feszültség értékét. A motorok helyzetét távadóval figyeljük, ezek kezelését is a motorvezérlő-panel végzi.



9. ábra: Az irányítási rendszer elvi felépítése

Folyamatok időzítése

A szabályozás megvalósításához valós idejű rendszerre van szükségünk, amely minden egyes megszakítást a körülményektől függetlenül, a lehető leggyorsabban végrehajt. A megszakítások a feldolgozásnak a lehető legkevesebb részét hajtják végre, a tényleges feldolgozás a fő ciklusban történik.

Ilyen funkciók a soros port írása/olvasása, a CAN-busz írása/olvasása, a motor PWM beállítása, a gombok ellenőrzése, egyéb időzített folyamatok (LED villogtatás).

Mivel ezek a funkciók a futási idejüket tekintve teljesen eltérők is lehetnek, csak egy ideális rendszer lenne képes mindent a megfelelő pillanatban végrehajtani. Mivel ilyen nehéz építeni, ezért nem szabad elvárni a rendszertől, hogy a funkciók ténylegesen a megfelelő időpillanatban kerüljenek meghívásra. Viszont ha elfogadjuk, hogy a vártnál több idő is eltelhet két szabályozási lépés közt, a szabályozás során ezt figyelembe is vehetjük.

Ennek a viselkedésnek az eléréséhez bizonyos megkötésekkel kell élni az egyes funkciók megvalósításakor. Osszuk a szükséges műveleteket jellegüktől függően két részre:

- **Aszinkron folyamat:** A műveletek érzékelési, megszakítás alapú része itt történik, csak számlálás és/vagy flagbeállítás történik.
- **Szinkron folyamat:** A művelet számítási része itt történik, a fő ciklusból van meghívva.

A folyamatokat a következő paraméterek jellemzik:

- hívások közti idő (ms)
- legutóbbi hívás ideje
- meghívni kívánt függvény címe

A funkciók ismételt meghívása közt eltelt időt milliszekundumban (ms) kell megadni, ez az idő azt mutatja meg, hogy egy hívás után mennyi idővel kell ismételt meghívni a függvényt. Amennyiben más folyamatok túl sokáig lefoglalják a processzort, és a szinkronciklus nem jut időben el a függvény újrahívásáig, abban az esetben csak késéssel lesz meghívva a függvény. A következő hívásra ettől az időponttól számítva kerül sor. Ez azért fontos, mert amikor a függvényekben a műveletek során az időt is számításba vesszük, akkor figyelembe kell vennünk az esetleges késésből adódó eltéréseket.

Az időt egy 2 bájtos számláló tárolja, túlcsoordulásakor 0-ról folytatja a számlálást. Mivel minden függvényhívásnál számolni kell a relatív és az abszolút idővel, ezért ezeket a paramétereket a függvény hívása során megkapják. Az alábbi példa azt mutatja be, hogyan kell alkalmazni az abszolút időt (t) egy függvényben és ezt a funkciót miként kell átadni az időzítőnek:

```
void BlinkLed(unsigned int t, unsigned int dt) {  
    if (t % 1024 < 256)  
        LedSet(&ledRed);  
    else  
        LedClear(&ledRed);  
}  
  
AddFunction(&BlinkLed, 128);
```

Azt szeretnénk megoldani, hogy a LED másodpercenként villanjon fel 200 ms-ra. Mivel a számláló bináris, ezért érdemes az időzítés számainak kettő valamelyik hatványát választani. Tehát ahogy a példában látható, a függvény 256 ms-onként hívjuk meg. Az abszolút időt maradékos osztással másodperces egységekre bontjuk, és a megfelelő szakaszban ki-bekapcsoljuk a LED-et. Tehát a funkcióhoz tartozó függvény akkor lesz meghívva, ha az aktuális idő és a legutóbbi hívás ideje közti

különbség megegyezik vagy meghaladja a kívánt időt. Ez az időzítés csak a funkciók szinkron részére vonatkoznak, a megszakítások kezeléséről és az ebből adódó ütközések elkerüléséről már minden funkciónak magának kell gondoskodnia. Az időzítés implementációját az *I. melléklet* tartalmazza.

A feldolgozás során lehetőség van az előző hívás óta eltelt időt (dt) számításba venni, valamint az abszolút időt (t), ami az időzítés számlálója. A számítások során minden időzített függvényben rendelkezésre állnak ezek az argumentumok, de csak az időzített feladattól függ, hogy ezek közül melyikre van szükség. A példában csak a „t” változó kerül felhasználásra. A „dt” változóra például a PID szabályzásnál van szükség, melynek kódja a II. mellékletben megtekinthető.

Kommunikáció

A kommunikáció soros porton és CAN-buszon történik. A soros port kétirányú kommunikációt tesz lehetővé, de csak két pont közt. Mivel az újabb számítógépeken már nem mindig található meg a soros port csatlakozási lehetősége, így nem lehetett a hagyományos RS232-es csatlakozási felületet használni. Manapság minden PC tartalmaz USB-portot. Az áramkörre így célszerűen FTDI nevű IC lett felszerelve. Ez egy USB-portra csatlakoztatható eszköz, amely virtuális soros portként működik. Így viszont csak a kommunikációs panellel van kapcsolatunk.

A többi panellel a kommunikáció CAN-buszon keresztül zajlik. A CAN-busz csomagokkal kommunikál, minden csomagnak van egy címzettje, azok az eszközök, amelyek lehallgatják ezt a címet, megszakításon keresztül, aszinkron módon kapják meg ezt a csomagot. A csomag 8 byte-ot tartalmaz, az üzenetek ilyen csomagokra darabolva közlekednek a panelek között. A CAN-busz protokollja gondoskodik az üzenetek hibáinak detektálásáról, az üzenetek vételének visszaigazolásáról.

Mivel a rendszerünk elosztva több egységet is tartalmaz, ezekkel szeretnénk közvetlenül is kommunikálni, de a kommunikációs csatornák sokfélesége miatt nehézségekbe ütközünk. Az üzenetek küldése és fogadása eltérő késleltetéssel, eltérő módon történik. Ahhoz, hogy karbantartható és kezelhető maradjon a rendszer a fejlesztés során, egységes kommunikációs formát vezettünk be.

A kommunikáció stringek/karakter tömbök formájában történik, minden sor egy parancsot jelent a sorokat egy '\n' karakter zárja le. Ha egy motorpanel küld üzenetet, a címzett minden esetben a PC. Ha a PC akar üzenetet küldeni, az üzenet elején a címzett nevével kell kezdeni, és azt egy kettősponttal el kell választani az üzenettől, például egy PID paraméterezést beállító üzenet a következőképpen néz ki:

m0:PID 1.51 2.112 3.335

Amikor egy üzenet vagy karakter beérkezik, aszinkron módon egy bufferben tárolja az adatot a rendszer, amennyiben a beérkezett karakter sorlezáró, akkor egy számlálót – ami a feldolgozatlan sorok számát tárolja – növelünk eggyel. A szinkronfutás során a feldolgozatlan üzeneteket sorról sorra dolgozzuk fel.

Mintavételes PID-szabályozás

A PID-szabályozást azért implementáltuk, mert szükség volt egy kipróbált, működő megoldásra, amellyel a rendszer helyes működését mérhetjük, ellenőrizhetjük, ezzel is felkészítve a neurális hálózat alkalmazására.

A PID-kompenzálás, amely sávszűrőnek felel meg, egyszerre javítja a szabályozási kör alacsony- és nagyfrekvenciás tulajdonságait (például megszünteti a hibajelet és csökkenti a szabályozási időt is). A PID kompenzálású szabályzó alkalmazása esetén, három paraméter beállítása szükséges: az arányos átviteli tényező (P), az integrálási átviteli tényező (I), valamint a differenciálási átviteli tényező (D).

Az ideális mintavételes PID-szabályzó működését az időtartományban a rendelkező jel és a végrehajtójel között az alábbi differenciaegyenlet, a (2) egyenlet írja le.

$$PWM = P * e_n + I * \left(\sum_{i=0}^n e_i * dt_i \right) + D * \frac{e_n - e_{n-1}}{dt_n} \quad (2)$$

Ahol:

- **PWM:** a vezérlőjel
- **e_n :** az aktuális hiba (kívánt érték – aktuális érték)
- **P, I, D:** az átviteli tényezők
- **dt_n :** pedig az előző mintavétel óta eltelt idő

A mintavételes szabályzó berendezés számító egységének a másik fontos feladata, hogy algoritmizált kompenzálassal a szabályozási kört a legjobb minőségi paraméterekkel működtesse, úgy értéktartás-, mint követő szabályozástechnikai alkalmazások esetén. A mintavételes szabályozók képesek PID jellegű szabályozók szerint üzemelni. A PID jellegű mintavételes szabályzókat napjainkban is előszeretettel használják az irányítástechnika területén.

A PID-szabályozás folyamata során a motor forgásának követése aszinkron módon történik, megszakítások formájában, a motor forgásának szabályozása pedig meghatározott időközönként kerül meghívásra szinkron módon.

Optimalizálás

A PID-szabályozást a paramétereinek helyes megválasztásával lehet optimalizálni. A szakirodalom nagyon sokféle szabályozó paraméterezési ajánlást közöl. Ezeket a paraméterezési módszereket tapasztalati úton (empirikusan) határozták meg. A szerzők különböző integrál kritériumokra történő optimalizálásnál kapták meg az eredményeiket.

Az általam alkalmazott optimalizáló eljárás kicsit általánosabb célokra készült. A Nelder-Mead [7] eljárás N-dimenziós térben geometriai alapon keres minimumot. Ennek olyan implementációját alkalmazom, amely Net-re készült F# nyelven. Ez az eszköz egy évtizedek óta, a szakma által folyamatosan tesztelt és finomított Fortran nyelvű implementációja alapján készült. Használatához függvény formájában kell kifejezni a problémát.

Ezek minimum függvények paraméterként megkapják mindegyik dimenzió értékét, és vissza kell térniük a hibával. PID-szabályozás esetén a három átviteli tényező adja a dimenziókat. A futtatáskor kiindulási értéket kell választani mindegyik változónak, illetve értéktartományt lehet meghatározni a minimalizáló függvénynek. PID szabályozásnál, ha bármelyik tag 0 alá esne, úgy garantáltan rossz szabályozást kapnánk, ezért valamennyi változó csak 0 és 10 között vehet fel értéket.

A Nelder-Mead egy háromszöget helyez el az N-dimenziós térben, és a „legmagasabb” pontját mindig átbillenti az ellentétes tengelyen. A háromszög méretét csökkenti, egyre jobban közelítve a lokális minimumot. Az újabb pontok kiértékeléséhez így állít elő mérési pontokat, amelyeket átad az általunk készített függvénynek, ez pedig kiértékeli a hibát. A hiba kiértékelésére itt is integrál kritériumokkal történik. A hiba számítása a (3)-as egyenlet alapján történik, amelyben összegezzük a mérési pontok hibáját.

$$f_{hiba} = \sum_{i=0}^n e_i^2 * dt_i \quad (3)$$

Ahol:

- **f_{hiba}**: a teljes szabályzott lépéssorozat összes hibája
- **e_i**: az i. lépés hibája
- **dt**: az i. lépés ideje

Az optimalizálás eredménye

Több szimuláció futtatása után azt mondhatjuk el, hogy 3 dimenzió esetén a Simplex algoritmus megbízhatóan képes megtalálni a lokális minimumot. Továbbá a megfelelő paraméterek megtalálásához pár tucat szimulációs lépés is elegendőnek bizonyult. A PID-szabályozás optimalizálásának eredményeiből az vonható le, hogy az általunk alkalmazni kívánt szabályozási feladatra nem igazán alkalmas, mivel nehéz olyan paraméterezést találni, amely gyors és lassú mozgás esetén is elfogadható eredményeket ad. Ilyen optimalizálás egyes lépései láthatóak a *IV. mellékletben*.

Neurális hálózatok

A neurális hálózatok [8] neuronok és azokat összekötő axonok összessége. A neuronok ugyanolyan vagy hasonló típusú műveleteket végeznek. A neuronok ezeket a műveleteket egymástól függetlenül, lokálisan végzik. Egy neuron nagyon sok másik neuronnal lehet összekapcsolva axonokkal, ezek a kapcsolatok súlyozva vannak, és a súlyok értékének módosítása útján ezek a hálózatok tanulni is képesek. Általában egy irányított gráffal reprezentáljuk őket, amelyben a csomópontok az egyes neuronok, amíg az irányok a kimenetektől a bemenetek felé mutatnak.

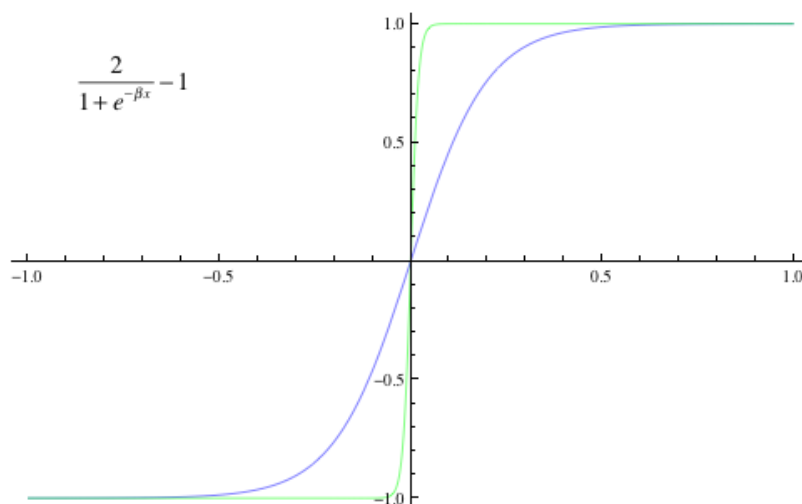
Az mérések során használt hálózat egy MLP (Multi Layer Perceptron) implementáció. A gyakorlatban talán ezt a típust alkalmazzák a leggyakrabban. Az MLP több rétegbe szervezett neuronokból áll. Egy

bemeneti rétegből, amely a bemenő adatok illesztésére szolgál, a kimeneti rétegből, amely az eredményt szolgáltatja. Továbbá tartalmaz egy rejtett réteget is, ennek a rétegnek köszönhetően az MLP tetszőleges folytonos nemlineáris függvény tetszőleges pontosságú approximációjára [9] képes. A hálózat tehát többrétegű, előrecsatolt hálózat.

A rejtett réteg neuronjainak számára általában felső becslést szoktak adni, ám ezek általában sokszorosai a gyakorlatban valóban szükséges méretnek. Mivel a neurális hálózatot beágyazott eszközön implementáljuk, ezért inkább ez a meghatározó a méret megválasztásakor. A memóriába és a futási időbe még beférő hálózatot próbáltam elkészíteni, és kipróbálni, hogy képes-e a pályakövetésre.

A használt átviteli függvény egy a 10. ábrán látható bipoláris szigmoid függvény (4), ahol α egy modellparaméter, amely az átmenet meredekségét határozza meg.

$y = \frac{2}{1 + e^{-\alpha \cdot x}} - 1$	(4)
---	-----



10. ábra: Bipoláris szigmoid függvény

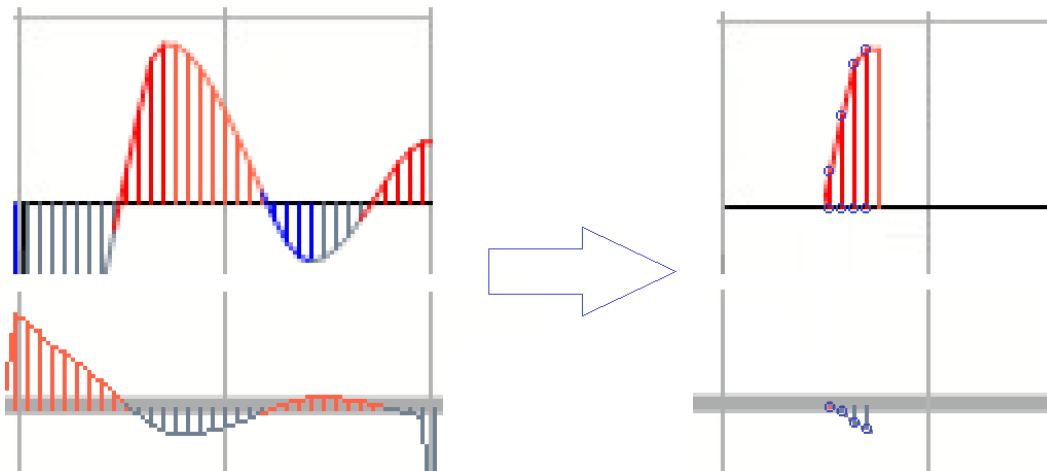
Bemeneti adatok

A feladat meghatározásakor meg kell határozni, milyen kimenetekre van szükségünk, és ez alapján kell megválasztani a bemeneteket. A PID-szabályozás tapasztalatai alapján szükség van az aktuális hibára és ennek változására, tehát a korábbi hibákra. Ehhez még hozzávettem a szabályozás helyzetét és a szabályozás utáni kimenet értékét. Mivel az aktuális helyzet, a kívánt helyzet és a hiba összefüggésben állnak, ezért elég ezekből kettőt a bemenetnek átadni, a harmadik paramétert már magukban hordozzák. Így adódik, hogy a neurális hálózat bemenetei több mérési pont paraméterei lesznek, a paraméterek a következők:

- kívánt helyzete
- aktuális helyzete

- PWM (kimenete)

Vegyük ezeket az aktuális időpillanatban, és az ezt megelőző n darab mérési pontban. Tároljuk a mérési értékeket egy fix méretű kör-bufferbe, amelybe minden lépésnél új adatsort töltünk be és kiolvassuk a teljes tartalmat egy vektorba. Ez a vektor adja a neurális háló bemenetét (11. ábra). A mostani tesztek során a lábat a saját tehetetlenségén kívül nem érte más terhelés, ezért feltételezhetjük, hogy minden irányban közel azonosan viselkedik. Ezt alapul véve, az értékeket az aktuális pozícióhoz mérten eltolom az értékeket a vektorban. A tesztelés során a legutóbbi négy mérési pontot veszem figyelembe, ez összesen 12 bemeneti neuront jelent.



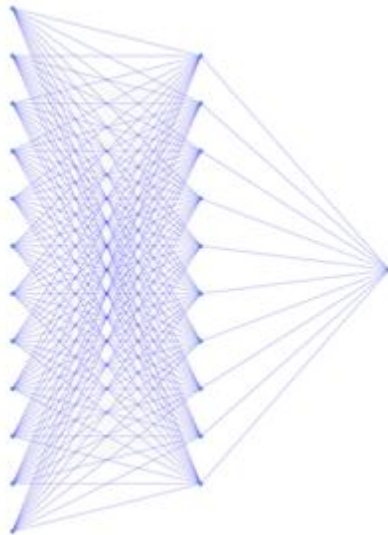
11. ábra: Tanító pontok

Rétegek

A rejtett rétegek száma egy MLP-ben tetszőleges lehet, érdemes lehet többet alkalmazni, mivel a rétegek számának növelésével, bizonyos esetekben csökkenthető a rendszer teljes neuronjainak száma. Viszont ez a rétegek olyan kezelését igényli, amely ilyen számítási kapacitás mellett körülményes lehet. Így egy réteget használunk. Tapasztat alapján a legtöbb probléma legfeljebb $2n + 1$ darab rejtett réteget igényel, ahol n a bemenetek száma.

A kimeneti réteg egyetlen neuronból áll. Ennek a neuronnak kimeneti értéke minden szabályozási lépés során a PWM értékét határozza meg.

Mivel csak egy réteget alkalmazunk, ezért az implementálás során egyszerűsítésekkel élhetünk, minden réteget előre definiálhatunk és ezekkel a megfelelő módon kezelhetjük. Valamint csak egyetlen neuronból áll a kimenet, ezért további egyszerűsítésekre nyílik lehetőség, több ciklus is kiejtethető e tulajdonság miatt. A neurális háló implementációja a *III. mellékletben* található. Az alkalmazott hálózat pedig a 12. ábrán látható.



12. ábra: Neurális hálózat topológiája

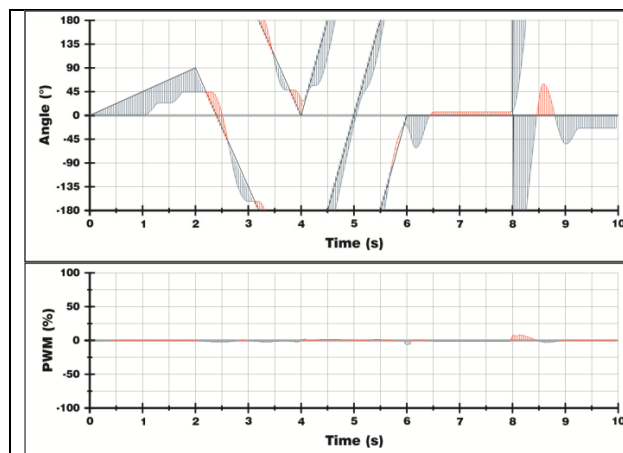
Tanítás

Mivel a neurális háló bemenetét végső soron a motor viselkedése határozza meg, ezért a hálózatot csak a motor vezérlése közben lehet tanítani. Minden szabályozási lépésben egy tanulási lépés is lezajlik. A tanításhoz back propagation algoritmust használtam, amelynek a tanítási lépéshez meg kell határozni egy kívánt értéket. Ez azt jelenti, hogy valamilyen módon meg kell határoznunk milyen értéket is szerettünk volna kapni. Mivel az irányított rendszert a neurális hálózat szempontjából fekete dobozként kezeljük, ezért ez az érték előre nem meghatározható. Az elvárt kimeneti értéket, úgy határozzuk meg, hogy a kapotthoz képest – a korábbi adatok figyelembe vételével – kisebb vagy nagyobb értéket adjon vissza.

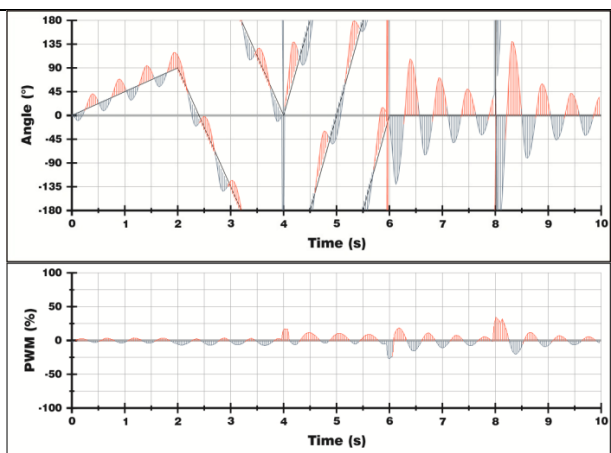
A tanulás során egy tanulási lépést két paraméter határoz meg, a tanulási sebesség (learning rate) és az aktuális pont hatásának mértéke, a momentum. A szabályozás során rengeteg ponton tud tanulni a rendszer ezért aránylag kis tanítási sebességgel is képes gyorsan tanulni. A momentumot a tanulás elején érdemes kisebbre venni, hogy a kezdeti hibák ne tegyék instabillá a tanulási folyamatot, de később az apróbb hibák kiküszöböléséhez érdemes lehet nagyobb értékkel is kipróbálni.

Kimenet javítása

Ha a korábbi néhány lépés hibáinak előjeles összegét veszem alapul, akkor a hálózat nagyon gyorsan megközelíti az ideális mozgási pályát, viszont az eredmény aránylag pontatlan és gyorsan túltanítottá válik a rendszer és oszcillálni kezd, ahogy az a 13. b ábrán látható.

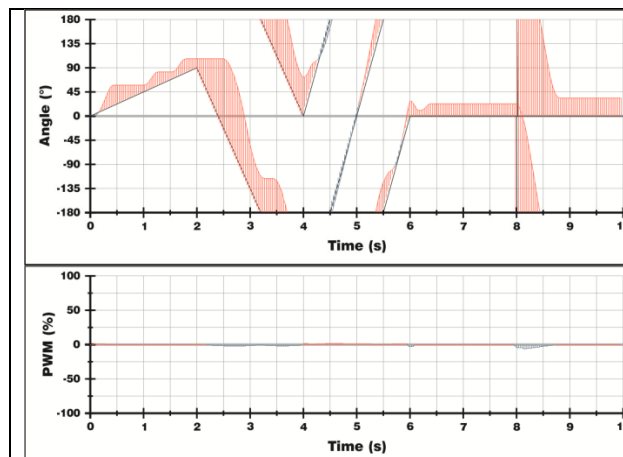


13. a ábra: 9. tanítási sorozat

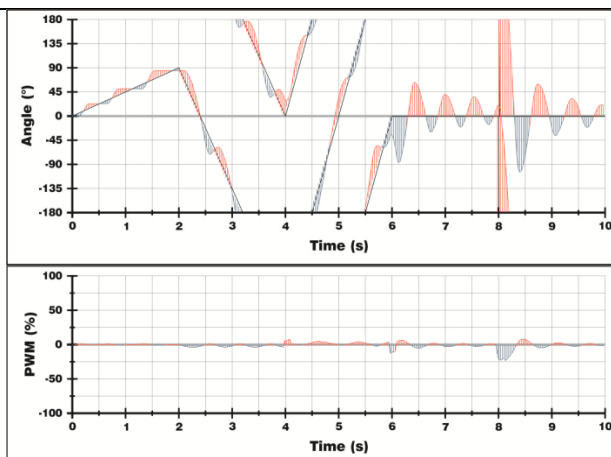


13. b ábra: 90. tanítási sorozat

Ha a hiba változását veszem alapul, és csak akkor tanítok, ha az előző lépéshez képest növekedett a hiba akkor a háló lassabban tanul. Kevésbé közelíti pontosan és idővel egyes hibák továbbra is hajlamosak az erősödre, ami a rendszer ismételt túltanulásához vezet, ahogy az a 14. ábrán látható.

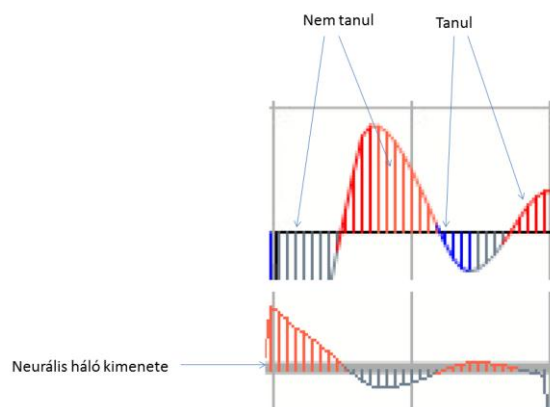


14. a ábra: 9. tanítási sorozat

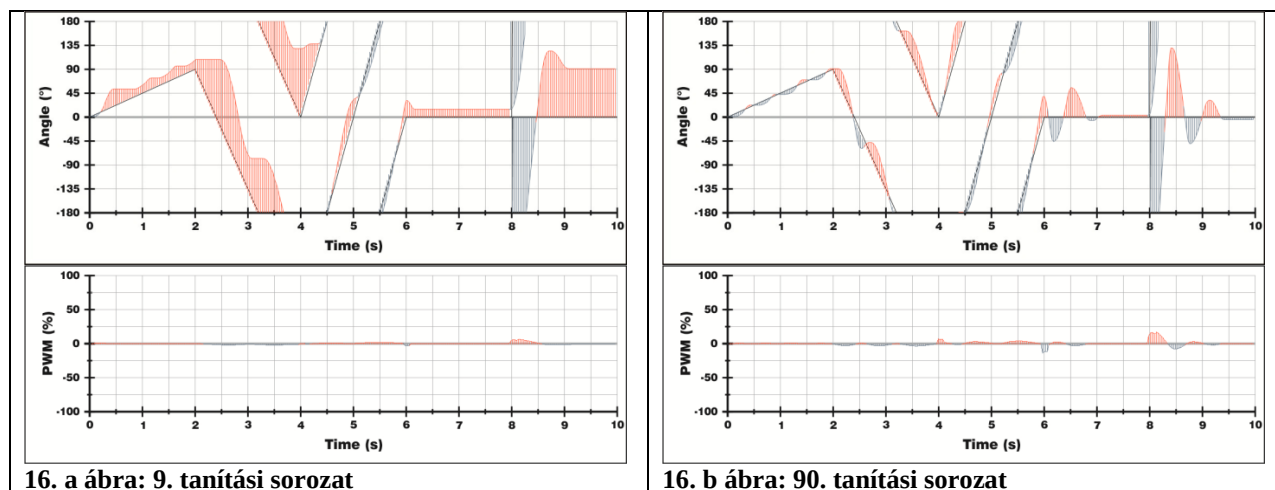


14. b ábra: 90. tanítási sorozat

Vegyük hozzá az ideális görbe meredekségét az eddigiekhez, és tanítás csak akkor legyen, ha a hiba változásának mértéke nagyobb mint a görbe változásának mértéke csak akkor tanítunk(13. ábra). Így már kevésbé hajlamos a hibák generálására és a görbét is jobban közelíti ahogy az a 16.ábrán látható.



13. ábra: Tanított szakaszok



Összegzés

A neurális hálózatok alkalmazhatónak bizonyultak szabályzók helyettesítésére. A kísérletek alapján közel ugyan olyan eredményeket értek el, mint a PID-szabályzó. Neurális hálózatok jól párhuzamosíthatósága miatt a beágyazott rendszerek területén leginkább az FPGA-k hoz áll közel, de mikrokontrollereken szintén megvalósítható. Viszont alkalmazhatóságát nehezíti az, hogy nehéz az egyes feladatokhoz megfelelő hálót vagy tanító algoritmust találni. Viszont ebben segít, hogy az irányítástechnika területén felhalmozódott tapasztalati tudás a neurális hálók alkalmazásakor is felhasználhatóak.

A további lehetőség, is rejlik a neurális hálózatokban. A bemutatott módszerekhez hasonló eljárásokkal saját pontosságát, energiaigényét optimalizálja a rendszer, illetve állapotát diagnosztizálhatja vele.

Felhasznált irodalom

- [1] J. Weingarten, G. Lopes, M. Buehler, R. Groff, and D. Koditschek. Automated gait adaptation for legged robots. volume 3, pages 2153–2158. International Conference on Robotics and Automation, 2004.
- [2] Samuel Burden, Jonathan Clark, Joel Weingarten, Haldun Komsuoglu, Daniel Koditschek. Heterogeneous Leg Stiffness and Roll in Dynamic Running, 2007.
- [3] Chen Li, Paul B. Umbanhowar, Haldun Komsuoglu, Daniel E. Koditschek, Daniel I. Goldman. Sensitive dependence of the motion of a legged robot on granular media
- [4] <http://kodlab.seas.upenn.edu/RHex/EduBot>, Látogatva: 2012.11.07
- [5] Mavridisz V., Gál B., Somlyai L.: „Lézerszenekkel támogatott önjáró robot körbelátó rendszerrel”, BMF NIK, OTDK dolgozat. (2009)

[6] A Physical Model for Dynamical Arthropod Running on Level Ground, Haldun Komsuoglu, Kiwon Sohn, Robert J. Full, Daniel E. Koditschek. 2008

[7] J.A. Nelder, R.A. Mead, A simplex method for function minimization (1965).

[8] Altrichter, Horváth, Pataki, Strausz, Takács, Valyon, Neurális hálózatok, Panem (2006)

[9] Kunihiro Fukushima, A hierarchical neural network model for associative memory, 1984

I. melléklet

Időzítést kezelő algoritmus és a hozzá tartozó adattípus implementációja.

```
typedef struct {
    unsigned int dt; // Futtatások közti idő
    unsigned int LastCall; // Legutóbbi hívás ideje
    void (*Func)(unsigned int, unsigned int); // Meghívandó függvény
} TFunc;

TFunc TimedFunc[4]; // függvények listája
unsigned int NumberOfFunctions; // függvények száma

void StepTimer() {
    unsigned int t;
    unsigned int dt;

    t = GetCount(); // Aktuális idő
    for(int i = 0; i < NumberOfFunctions; i++) {
        if (t > TimedFunc[i].LastCall)
            dt = t - TimedFunc[i].LastCall; // Utolsó hívás óta eltelt idő
        else
            dt = (t - 0x7FFF) - (TimedFunc[i].LastCall + 0x7FFF); // Túlcsordulás elkerülése

        if (dt >= TimedFunc[i].dt) { // Ha itt az ideje, meghívja a függvényt
            TimedFunc[i].LastCall = t;
            TimedFunc[i].Func(t, dt);
        }
    }
}
```

II. melléklet

A motor definíciója, PID-szabályozás implementációja.

```
typedef struct{
    signed int Count;
    float error, prevError, sumError;
    float setPoint, output;
}Motor;
void pid(Motor *m, unsigned int dt)
{
    float derivative;

    //Calculate P,I,D
    m->error = m->setPoint - m->Count;

    //In case of error too small then stop intergration
    if(abs(m->error) > 0.0001)
    {
        m->sumError = m->sumError + m->error * (float)dt;
    }
    derivative = (m->error - m->prevError) / (float)dt;
    m->output = (m->p*m->error + m->i*m->sumError + m->d*derivative);
    if(m->output > maxPWM) //Saturation Filter
    {
        m->output = maxPWM;
    }
    else if(m->output < minPWM)
    {
        m->output = minPWM;
    }
    m->prevError = m->error;           //Update error
}
```

III. melléklet

Az alábbi kód egy egyszerűsített neurális háló implementációja, amely a gyors működésre lett optimalizálva.

```
#define InputSize 12
#define HiddenSize 10
#define OutputSize 1

float input[InputSize];
float hidden[HiddenSize];
float output[OutputSize];

float inputError[InputSize];
float hiddenError[HiddenSize];
float outputError[OutputSize];

float hiddenWeight[InputSize * HiddenSize];
float hiddenWeightUpdate[InputSize * HiddenSize];
float hiddenThershold[HiddenSize];
float hiddenThersholdUpdate[HiddenSize];

float outputWeight[HiddenSize * OutputSize];
float outputWeightUpdate[HiddenSize * OutputSize];
float outputThershold[OutputSize];
float outputThersholdUpdate[OutputSize];

float cachedMomentum;
float cached1mMomentum;

float alpha = 0.3f;

void NeuralInitalize();
float NeuralStep(float inputValues[]);
void NeuralTrain(float target, float lr, float mom);

float sig(float x) {
    return (float)(( 2 / ( 1 + exp( -alpha * x ) ) ) - 1 );
}

float dsig(float y) {
    return ( alpha * ( 1 - y * y ) / 2 );
}

void NeuralInitalize(){
    for (int i = 0; i < InputSize; i++)
    {
        for (int h = 0; h < HiddenSize; h++)
        {
            hiddenWeight[h * InputSize + i] = (float)((rand()) % 1000 - 500) / 10000.0f;
        }
    }

    for (int h = 0; h < HiddenSize; h++)
    {
        hiddenThershold[h] = (float)((rand()) % 1000 - 500) / 5000.0f;
        for (int o = 0; o < OutputSize; o++)
        {
            outputWeight[o * InputSize + h] = (float)((rand()) % 1000 - 500) / 10000.0f;
        }
    }
    outputThershold[0] = (float)((rand()) % 1000 - 500) / 5000.0f;
}

float NeuralStep(float inputValues[]) {
```

```

        for (int i = 0; i < InputSize; i++)
        {
            input[i] = inputValues[i];
        }
        Calculate();

        return output[0];
    }

void NeuralTrain(float target, float lr, float mom) {
    cachedMomentum = lr * mom;
    cached1mMomentum = lr * (1 - mom);

    outputError[0] = (target - output[0]) * dsig(output[0]);

    CalculateError();
    CalculateUpdate();
    Update();
}

void Calculate() {
    float sum;
    int y = 0;

    for (int h = 0; h < HiddenSize; h++)
    {
        sum = 0;
        for(int i = 0; i < InputSize; i++)
        {
            sum += input[i] * hiddenWeight[y + i];
        }

        hidden[h] = sig(sum + hiddenThershold[h]);

        y += InputSize;
    }

    if (hiddenWeight[0] == hiddenWeight[0] + 5)
        return;

    sum = 0;
    for(int h = 0; h < HiddenSize; h++)
    {
        sum += hidden[h] * outputWeight[h];
    }

    output[0] = sig(sum + outputThershold[0]);
}

void CalculateError(float diso){
    float sum;
    int pos;

    for (int h = 0; h < HiddenSize; h++)
    {
        hiddenError[h] = (outputError[0] * outputWeight[h]) * dsig(hidden[h]);
    }

    for (int i = 0; i < InputSize; i++)
    {
        sum = 0;
        pos = i;
        for(int h = 0; h < HiddenSize; h++)
        {
            sum += hiddenError[h] * hiddenWeight[pos];
            pos += InputSize;
        }
    }
}

```

```

        inputError[i] = sum * dsig(input[i]);
    }
}

void CalculateUpdate() {
    float sum;
    int y, h, i;
    float cerror;

    y = 0;
    for (h = 0; h < HiddenSize; h++)
    {
        sum = 0;
        cerror = hiddenError[h] * cached1mMomentum;

        for(i = 0; i < InputSize; i++)
        {
            hiddenWeightUpdate[y + i] = cachedMomentum * hiddenWeightUpdate[y + i] + cerror *
input[i];
        }

        hiddenThersholdUpdate[h] = cachedMomentum * hiddenThersholdUpdate[h] + cerror;

        y+= InputSize;
    }

    cerror = outputError[0] * cached1mMomentum;
    for(h = 0; h < HiddenSize; h++)
    {
        outputWeightUpdate[h] = cachedMomentum * outputWeightUpdate[h] + cerror * hidden[h];
    }

    outputThersholdUpdate[0] = cachedMomentum * outputThersholdUpdate[0] + cerror;
}

void Update()
{
    int y = 0;
    for (int h = 0; h < HiddenSize; h++)
    {
        for(int i = 0; i < InputSize; i++)
        {
            hiddenWeight[y + i] = hiddenWeight[y + i] + hiddenWeightUpdate[y + i];
        }

        hiddenThershold[h] = hiddenThershold[h] + hiddenThersholdUpdate[h];

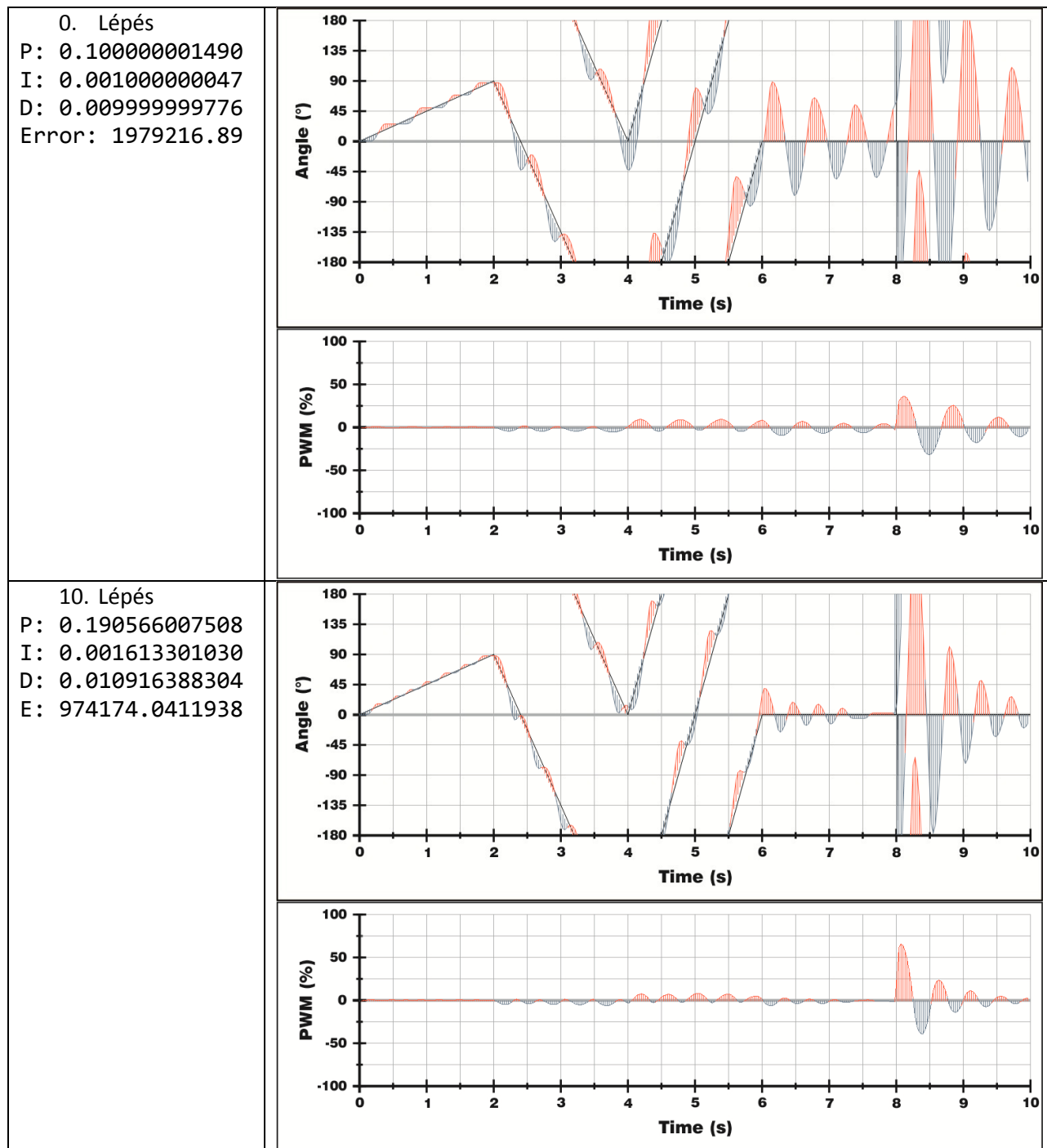
        y+= InputSize;
    }

    for(int h = 0; h < HiddenSize; h++)
    {
        outputWeight[h] = outputWeight[h] + outputWeightUpdate[h];
    }
}

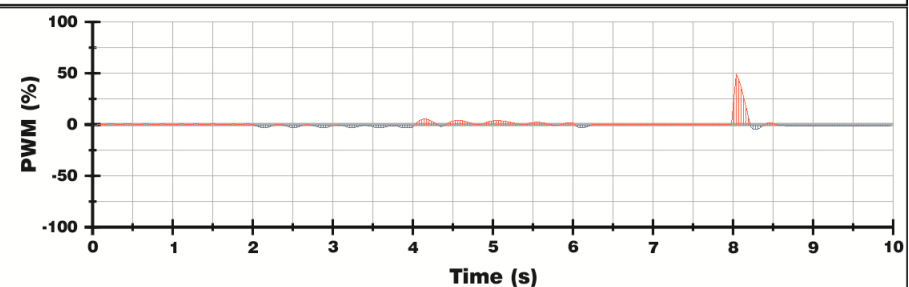
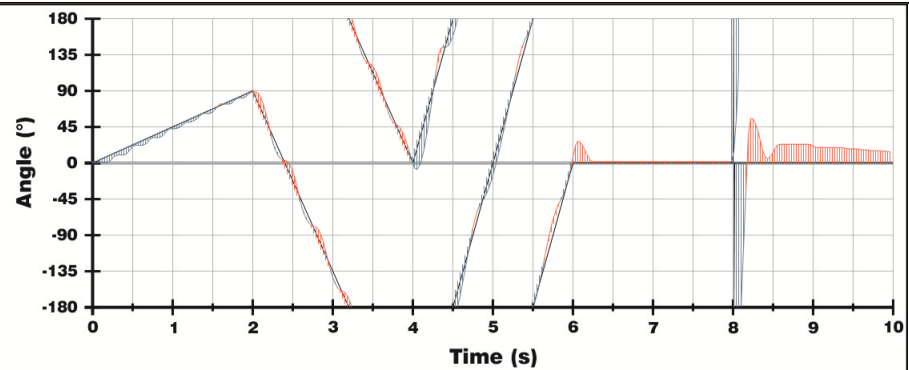
```

IV. melléklet

Az alábbi melléklet a PID-paraméterek Nelder-Mead Simplex optimalizálás a során, az egyes lépéseket és eredményét mutatja be.



30. lépés
P: 0.199606716211
I: 0.000142793977
D: 7.658269E-05
E: 415644.2605104



47. Lépés
(legjobb paraméterek)
P: 0.199618812304
I: 7.96075E-09
D: 0.000109616399
E: 405813.893341

