SAPIENTIA ERDÉLYI MAGYAR TUDOMÁNYEGYETEM
MAROSVÁSÁRHELYI KAR

AUTOMATIKA ÉS ALKALMAZOTT INFORMATIKA

Quadruped mobilis robot mozgását megvalósító algoritmusok tervezése

TUDOMÁNYOS DIÁKKÖRI KONFERENCIA
DOLGOZAT
2024

Hallgató:
Csillag Alex

Témavezetők:
Dr. Dávid László (docens), *Sapientia EMTE, Marosvásárhely*
Dr. Márton Lőrinc (docens), *Sapientia EMTE, Marosvásárhely*
Köllő Zsolt (szoftverfejlesztő), *Codespring, Marosvásárhely*
Györfi Endre (szoftverfejlesztő), *Codespring, Marosvásárhely*

Kivonat

Napjainkban a négylábú mobilis robotok, vagy más néven quadruped robotok, egyre népszerűbbek és egyre nagyobb szerepet töltenek be. Az ilyen típusú robotok számos területen, mint például a mezőgazdaság, építőipar, katasztrófavédelem, egészségügy, valamint háztartásokban is alkalmazzák. A mobilis robotok olyan feladatok elvégzésére vannak tervezve, ahol helyváltoztatás is szükséges, azaz az adott feladat térbeli helyzete a robotra felszerelt szerszám (robotkar, amely manipulációs feladatok elvégzésére képes) munkateréből kívül esik. Azonban, hogy ezek a robotok hatékonyan és biztonságosan működjenek, számos komplexitással kell szembenézni a tervezés során. A quadruped robotok mobilitása kulcsfontosságú tényező, mivel ezeknek a gépeknek képesnek kell lenniük különböző tereptípusokon való mozgásra és akadályok áthidalására. A navigációs rendszerek kifejlesztése elengedhetetlen a robotok hatékony és pontos mozgásának biztosításához, beleértve az akadályfelismerést és -kerülést is. Emellett a quadruped robotoknak dinamikusan kell alkalmazkodniuk a környezetükhöz és a változó terephez.

Dolgozatom fő célja egy ilyen quadruped robot mozgásának vizsgálata és elemzése. Ennek érdekében a robot szerkezetének és működésének ismertetésére kerül sor, beleértve a mozgást megvalósító vezérlő algoritmusokat és a stabilitás vizsgálatát is. Ez az elemzés matematikai modellek illetve szimulációs módszerek segítségével történik.

A quadruped robotok tervezése és fejlesztése a jövőben egyre fontosabbá válik az automatizálás és a robotika területén. Dolgozatom által nyújtott eredmények és megközelítések hozzájárulhatnak ezeknek a rendszereknek a hatékonyságának és funkcionalitásának növeléséhez, elősegítve ezzel a quadruped robotok széleskörű alkalmazását és elfogadását a mindennapi életben.

Abstract

Nowadays, four legged mobile robots, also known as quadruped robots, are becoming more and more popular and are playing an increasingly important role. These types of robots are used in many fields, such as agriculture, construction, disaster management, healthcare, as well as in households. Mobile robots are designed to perform tasks where a change of position is required, i.e. the spatial location of the task is outside the working space of the tool (robot arm, which can perform manipulation tasks) mounted on the robot. However, for these robots to operate efficiently and safely, a number of complexities need to be addressed in their design. The mobility of quadruped robots is a key factor, as these machines need to be able to move over different types of terrain and negotiate obstacles. The development of navigation systems is essential to ensure efficient and accurate robot movement, including obstacle detection and avoidance. In addition, quadruped robots must be able to adapt dynamically to their environment and to changing terrain.

The main goal of my thesis is to study and analyse the motion of such a quadruped robot. To this end, the structure and operation of the robot will be described, including the control algorithms that implement the motion and the stability analysis. This analysis is carried out using mathematical models and simulation methods.

The design and development of quadruped robots will become increasingly important in the future in the field of automation and robotics. The results and approaches provided in my thesis can contribute to increasing the efficiency and functionality of these systems, thus promoting the widespread use and adoption of quadruped robots in everyday life.

Extras

În prezent, roboții mobili patrupede, cunoscuți și sub numele de roboți quadrupede, devin din ce în ce mai populari și joacă un rol din ce în ce mai important. Aceste tipuri de roboți sunt utilizate în multe domenii, cum ar fi agricultura, construcțiile, gestionarea dezastrelor, asistența medicală, precum și în gospodării. Roboții mobili sunt concepuți pentru a îndeplini sarcini în care este necesară o schimbare de poziție, adică locația spațială a sarcinii se află în afara spațiului de lucru al instrumentului (brațul robotului, care poate efectua sarcini de manipulare) montat pe robot. Cu toate acestea, pentru ca acești roboți să funcționeze în mod eficient și sigur, este necesar să se abordeze o serie de complexități în proiectarea lor. Mobilitatea roboților quadrupede este un factor-cheie, deoarece aceste mașini trebuie să se poată deplasa pe diferite tipuri de teren și să poată negocia obstacole. Dezvoltarea sistemelor de navigație este esențială pentru a asigura o deplasare eficientă și precisă a roboților, inclusiv detectarea și evitarea obstacolelor. În plus, roboții quadrupede trebuie să fie capabili să se adapteze în mod dinamic la mediul înconjurător și la schimbările de teren.

Scopul principal al tezei mele este de a studia și analiza mișcarea unui astfel de robot patruped. În acest scop, vor fi descrise structura și funcționarea robotului, inclusiv algoritmii de control care implementează mișcarea și analiza stabilității. Această analiză se realizează cu ajutorul modelelor matematice și al metodelor de simulare.

Proiectarea și dezvoltarea de roboți quadrupede va deveni din ce în ce mai importantă în viitor în domeniul automatizării și roboticii. Rezultatele și abordările furnizate în teza mea pot contribui la creșterea eficienței și funcționalității acestor sisteme, promovând astfel utilizarea și adoptarea pe scară largă a roboților quadrupede în viața de zi cu zi.

Köszönetnyilvánítás

A dolgozat sok személy hozzájárulásának köszönhetően valósult meg. Nagy segítséggel voltak a Sapientia EMTE marosvásárhelyi karán, a villamosmérnöki tanszéken oktató tanárok, különösebben **Dr. Dávid László** valamint **Dr. Márton Lőrinc**. A mesterséges intelligencia illetve robotika iránti érdeklődésemet nekik tudom köszönni. Sikerült bebízonyítaniuk, hogy a sok megrémisztő matematika mögött ha a problémákat intuitívebb szemszögből közelítjuk meg, akkor egy nagyon érdekes és megdöbbentő világ rejlik. A projekt ötletének létrejöttét a Codespring cégnek köszönhetem, pontosabban a mentoraimnak, **Köllő Zsoltnak** illetve **Györfi Endrének**. Ők voltak azok a személyek a dolgozat elkészítése közben akiknek sikerült a motivációt bennem tartsák valamint egy szoftverfejlesztő nézőpontjából tanácsokat adjanak akár szoftver szintjén, dokumentáció szintjén, munka szintjén vagy általánosan életre szóló problémákkal kapcsolatban.

Tartalomjegyzék

1. Projekt leírása és bevezető

Az ipari robotok alkalmazása egyre szélesebb körben elterjedt napjainkban, és a jövőben várhatóan még nagyobb szerepet fognak játszani a gazdaságban. Kétféle robot családot különböztetünk meg funkcionalitás szempontjából, *manipulátor robotok* illetve *mobilis robotok*. A manipulátor robotok elsősorban ipari célokra használják fel, például hegesztésre, összeszerelésre, festésre vagy anyagmozgatásra. A mobilis robotok pedig olyan feladatok elvégzésére alkalmasak, ahol a helyváltoztatás is szükséges, például takarításra, háztartási feladatok elvégzésére, szállító feladatokra, vagy akár idős emberek gondozásában is sokat képes segíteni. Ez azt jelenti, hogy nemcsak egy adott helyen képesek elvégezni a feladatukat, hanem képesek eljutni a feladat végrehajtásához szükséges helyre. A mobilis robotok általában kerekekkel, lábakkal vagy más járművekkel rendelkeznek, amelyek lehetővé teszik számukra a mozgást.

Dolgozatom fő célja egy ilyen quadruped robot mozgásának vizsgálata és elemzése. Ennek érdekében a robot szerkezetének és működésének ismertetésére kerül sor, beleértve a mozgást megvalósító vezérlő algoritmusokat és a stabilitás vizsgálatát is. Ez az elemzés matematikai modellek illetve szimulációs módszerek segítségével történik.

Egy mobilis robot fejlesztése számos kihívással jár, ilyen például a mobilitás megvalósítása, azaz a robotnak képes kell lennie a kihívó környezetében való navigációra és az akadályok elkerülésére [2]; a mobilis robot biztonságossá tétele az emberek, a tárgyak illetve a környezetére nézve; a mobilis robot intelligenciájának megvalósítása, azaz képes kell, hogy legyen a környezete felismerésére és a feladatok lehető legoptimálisabb elvégzésére.

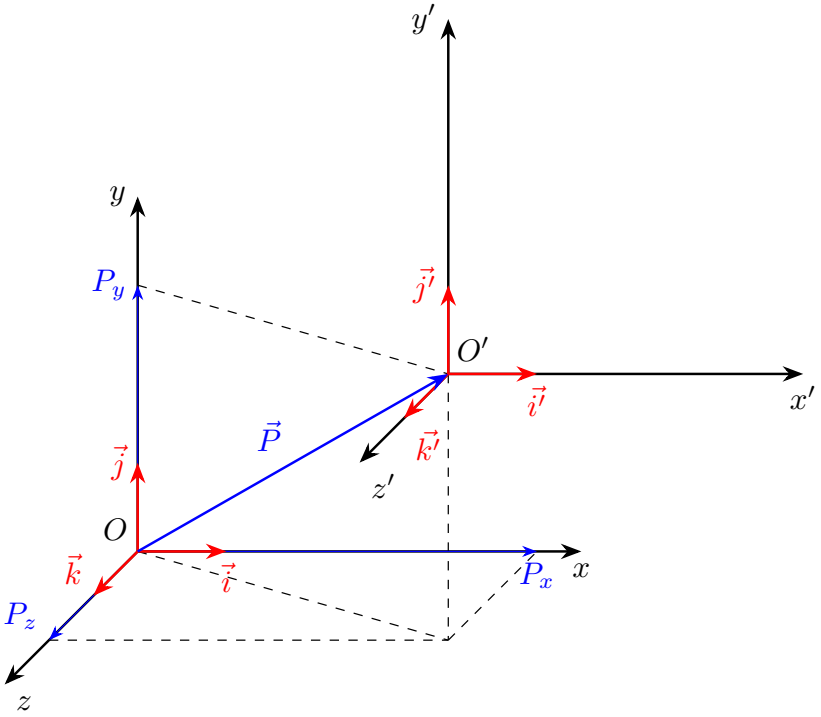
A projekt ötletét a 2022/2023-as tanév alatt elvégzett szakmai gyakorlatom adta. Ezt a **Codespring** [4] cég segítségével tudtam teljesíteni. Eredetileg a projektet egy két fős csapatban végeztem Babos Dávid társaságában. A konkrét feladatom egy quadruped robot vezérlése és programozása volt. Ez több feladatot is magába foglal. Egyik ilyen probléma a robot mozgásának megértése (pozitív- illetve inverz kinematika), a kommunikációs csatorna felépítése, motrok pozíció szabályozása, mesterséges intelligencia alapú taníása (megerősíéses tanulás *[reinforcement learning]*), képfelismerés illetve pályakövetés. A szakmai gyakorlat teljesítése után egy kicsit másféle megközelítésből újra tervezem a projektet teljes mértékben egyedi és önálló munkával.

2. Elméleti bevezető

2.1. Robotika alapfogalmak

2.1.1. Koordináta transzláció (eltolás)

Vegyünk két koordináta-rendszert két különböző origóval, ez esetben O és O'. Az O koordináta-rendszerhez képest fel tudjuk írni az O' koordináta-rendszer relatív pozícióját.



1. ábra. Koordináta transzformáció (transzláció)

Transzlációs vektor

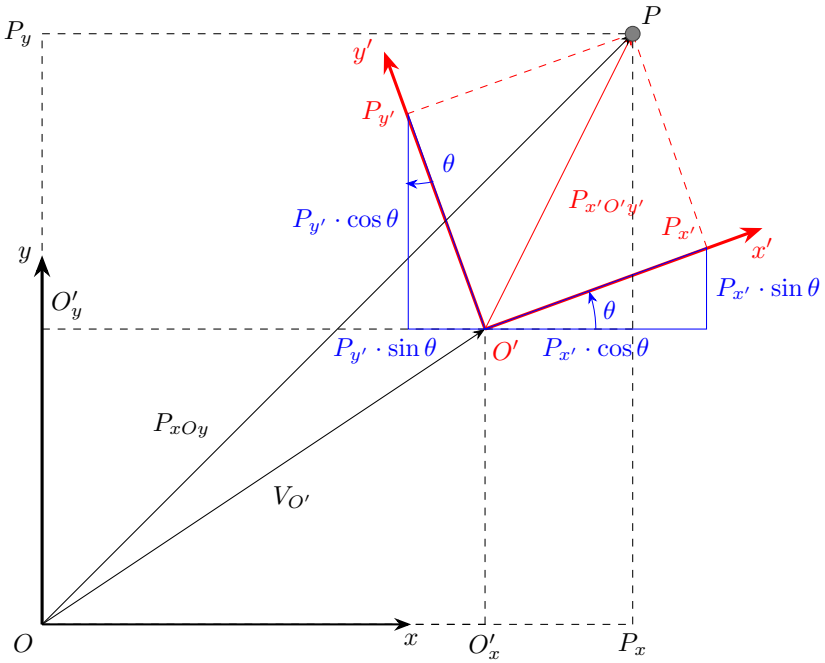
A fenti ábra alapján a következő egyenletet fel tudjuk írni:

$$\vec{P} = P_x \cdot \vec{i} + P_y \cdot \vec{j} + P_z \cdot \vec{k} = \begin{pmatrix} P_x \\ P_y \\ P_z \end{pmatrix} \tag{1}$$

Az $\vec{i}$, $\vec{j}$, $\vec{k}$ vektor az O koordináta-rendszer egységvektorai.

2.1.2. Koordináta elforgatás + eltolás (rotáció + transzláció)

Vegyünk két koordináta-rendszert két különböző origóval (két-dimenzióra egyszerűsítve), ez esetben O és O'. Az O koordináta-rendszerhez képest fel tudjuk írni az O' koordináta-rendszerben a relatív rotációt.



2. ábra. Koordináta transzformáció (rotáció)

A fenti ábra alapján fel tudjuk írni a P pontot az O koordináta-rendszerben a következőképpen:

$$P = P_x \cdot \vec{i} + P_y \cdot \vec{j} = \begin{pmatrix} P_x \\ P_y \end{pmatrix}$$

Ha át szeretnénk térni az O' koordináta-rendszerbe, akkor a következő képpen tudjuk megtenni ezt:

$$P' = P_{x'} \cdot \vec{i} + P_{y'} \cdot \vec{j} = \begin{pmatrix} P_{x'} \\ P_{y'} \end{pmatrix}$$

A P pont koordinátái az O koordináta-rendszerben, ismerve az O' koordináta-rendszerbéli koordinátákat.

$$P = \begin{pmatrix} P_x \\ P_y \end{pmatrix} = \begin{pmatrix} P_{x'} \cdot \cos(\theta) - P_{y'} \cdot \sin(\theta) + O'_x \\ P_{x'} \cdot \sin(\theta) + P_{y'} \cdot \cos(\theta) + O'_y \end{pmatrix} \tag{2}$$

Ezt a következő képpen tudjuk egyszerüsíteni.

$$P_{xOy} = \begin{pmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{pmatrix} \cdot P_{x'O'y'} + V_{O'} \tag{3}$$

Rotációs mátrix

Ezek alapján fel tudjuk írni a **rotációs mátrix** meghatározását két-dimenzióban.

$$R_\theta = \begin{pmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{pmatrix} = \begin{pmatrix} i_x & j_x \\ i_y & j_y \end{pmatrix} \tag{4}$$

Három-dimenzióra általánosítva a következőt kapjuk:

$$R_{\theta\phi\gamma} = \begin{pmatrix} i_x & j_x & k_x \\ i_y & j_y & k_y \\ i_z & j_z & k_z \end{pmatrix} \tag{5}$$

2.1.3. Homogén transzformációs mátrix

Ahhoz, hogy egyszerre tudjuk kezelni a transzlációs illetve rotációs műveleteket, össze tudjuk rakni őket egy matematikai struktúrába, ez a **transzformációs mátrix**.

Transzformációs mátrix

Szükséges kibővítenünk egy nullvektorral, illetve egy skaláris 1-el, a transzformációs mátrixot ahhoz, hogy egy struktúrába tudjuk elhelyezni a rotációs mátrixot illetve transzlációs vektort. Így megkapjuk az általános alakját.

$$T = \left(\begin{array}{c|c} R_{3x3} & P_{3x1} \\ \hline 0^T_{1x3} & 1 \end{array}\right) = \left(\begin{array}{ccc|c} i_x & j_x & k_x & P_x \\ i_y & j_y & k_y & P_y \\ i_z & j_z & k_z & P_z \\ \hline 0 & 0 & 0 & 1 \end{array}\right) \tag{6}$$

2.1.4. Transzformációs mátrix felhasználása a robotikában

A robotikában minden egyes csuklóhoz illetve szegmenshez tudunk rendelni egy-egy koordináta-rendszert. Ha egyik koordináta-rendszerből át szeretnénk térni egy másik koordináta-rendszerbe, akkor azt transzformációs mátrixok segítségével tudjuk megtenni. Soros robot architektúra esetén, a szegmenseket egy darab csukló csatolja egymáshoz, mindegyiknek megvan a saját koordináta rendszere. Ha a végberendezés pozícióját meg szeretnénk határozni a báziskoordináta-rendszerhez képest, akkor a csuklóknak felvett koordináta-rendszerekből ezt meg tudjuk határozni a transzformációs mátrixok szorzásával. Ezt direkt geometriai feladatnak is nevezzük [10].

Direkt geometriai feladat

Legyena K_{i-1} és K_i koordináta-rendszerek közötti transzformáció mátrixa T_{i-1}^i. Ez esetben ha a bázis koordináta-rendszer (K_0) alapján meg szeretnénk határozni a végpontban található koordináta rendszert (K_E), akkor ezt a következő képpen tudjuk megtenni:

$$T_E^0 = T_1^0 \cdot T_2^1 \cdot \dots \cdot T_n^{n-1} \cdot T_n^E = \prod_{i=0}^{n+1} T_{i-1}^i \tag{7}$$

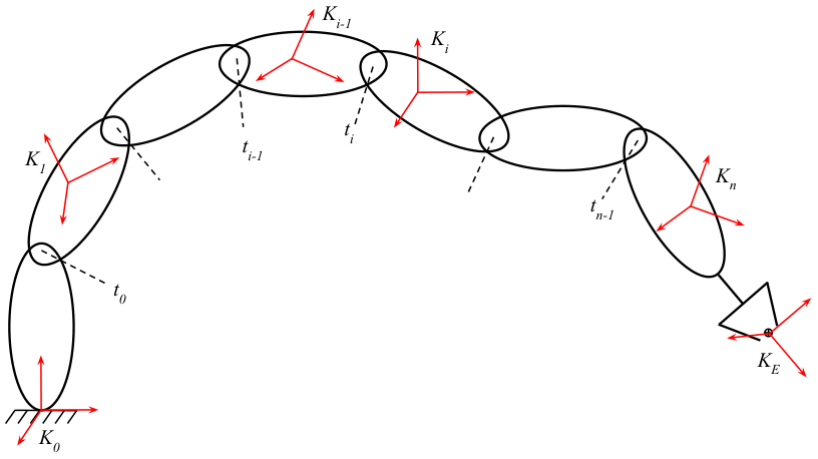
2.1.5. Denavit-Hartenberg (D-H) konvenció

A Denavit-Hartenberg konvenció egy módszert ad egy adott szegmens koordináta rendszerének definiálására, valamint egymást követő koordináta-rendszerek közötti homogén transzformációs mátrixok meghatározásához. Ezt a csuklóváltozók tere, illetve a robot geometriai paraméterei függvényében tudjuk megtenni.

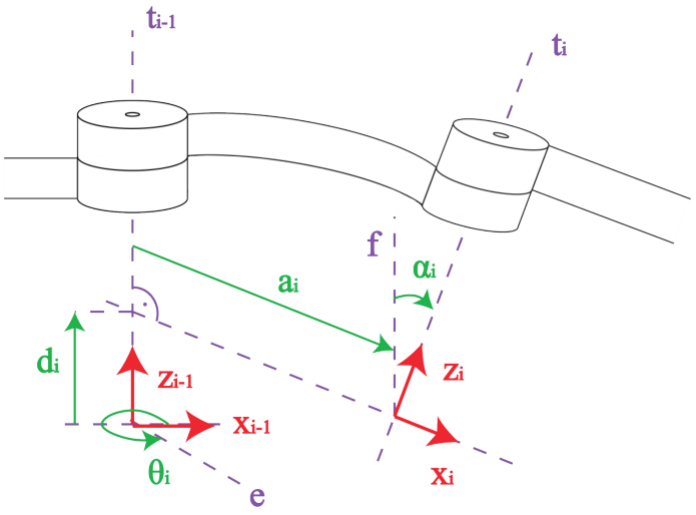
Denavit-Hartenberg paraméterek

Négyfajta paramétert különböztethetünk meg:

1. **Csuklószög** (θ_i): $Rot(z_{i-1}, \theta_i)$;
2. **Csukló offszet** (d_i): $Trans(z_{i-1}, d_i)$;
3. **Szegmens hossz** (a_i): $Trans(x_i, a_i)$;
4. **Szegmens csavarodás** (α_i): $Rot(x_i, \alpha_i)$.



3. ábra. Egy robot manipulátor elvi rajza [10].



4. ábra. Denavit-Hartenberg paraméterek [10].

A koordináta-rendszerek meghatározásának lépései:

1. K_i koordináta-rendszer z_i-tengelye egybe kell essen t_i-vel;
2. K_i koordináta-rendszer x_i-tengelyét úgy vesszük fel, hogy merőleges legyen z_{i-1}-tengelyre;
3. K_i koordináta-rendszer origója ott van ahol az x_i-tengely metszi a t_i-tengelyt.

Homogén transzformációs mátrix D-H paraméterek alapján

A D-H paraméterek segítségével fel tudjuk írni a homogén transzformációs mátrixot két koordináta-rendszer között a következő egyenlet segítségével:

$$T_i^{i-1} = Rot(z_{i-1}) \cdot Trans(z_{i-1}, d_i) \cdot Trans(x_i, a_i) \cdot Rot(x_i, \alpha_i) \qquad (8)$$

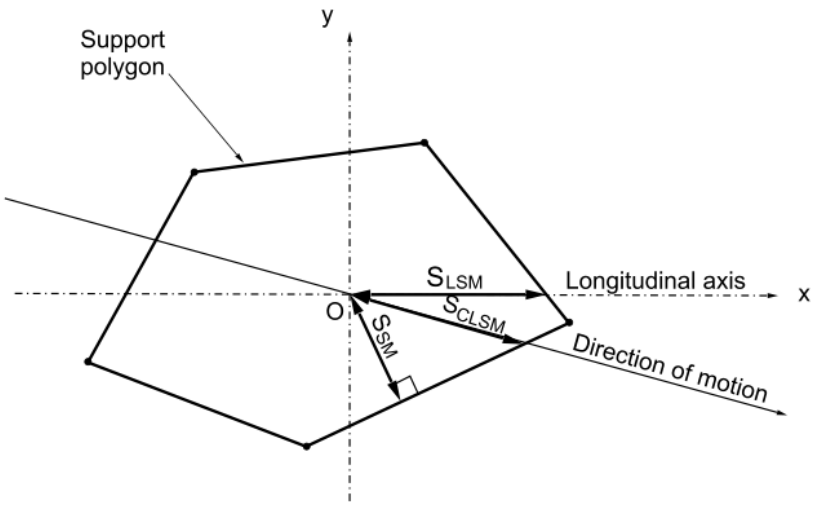
2.2. Mobilis robot stabilitása

Nagyon fontos, hogy számoljunk a robot stabilitásával amikor pályatervezést végzünk, akár egy adott lábra, ugyanis ha a robot kibillen a stabil állapotából, hiába a robot testéhez képest egy adott láb végezni tudja a pályakövetést, de a robot nem fogja az elvárt feladatot tudni elvégezni. Fontos letisztázzuk a stabilitás fogalmát mobilis robotok esetében, amely egy párhuzamos architektúrájú robot, a soros manipulátor robotokkal ellentétben.

Egy robot statikusan stabil akkor amikor a tömegközéppont *(Center of Gravity - COG)* horizontális vetítése a támasztó-felületre egy adott támasztó-minta *(support-pattern)* területén belül található, mi esetünkben egy konvex támasztó-poligon *(support polygon)* [7] [12].

A támasztó-poligont úgy határozzuk meg, hogy figyelembe vesszük a robot lábainak a kontaktusát a támasztó-felülettel, és egy egyenest húzunk két egymás melletti kontaktban levő láb között. n kontaktban levő láb esetében n oldala lesz a támasztó-poligonnak. Ezze könnyen tudjuk vizsgálni azt, hogy perturbáció (tömegközéppont elmozdulása) esetében hogyan viselkedik a robot. Ha a támasztó-poligon területén kívül esik a tömegközéppont akkor is képes kell legyen a robot megtartani saját magát vagy stabil állapotba visszatérni.

A tömegközéppont változásának figyelése érdekében szükséges, hogy tudjuk a robot kinematikai modelljét is amelyhez szükséges a Jakobi mátrix meghatározása is [14].



5. ábra. Támasztó-poligon és különböző stabilitási határok [7].

A robot stabilitásába még beleszólhat a szegmensek, csuklók, maga akár a test interciája (tehetetlensége) is amelyet zavarként tekintünk ugyanis nem várt nyomatékokat hozhatnak létre a csuklókban. Ezek mellett fontos még a surlódás (tapadó-surlódás, viszkózus-surlódás) kompenzálása is. E zavarok jelenlétében a robot vezérlése korlátozva van alacsony és konstans sebességekre. Az egyedüli megoldás a sebességek növelésére az, hogyha figyelembe vesszük a robot dinamikai tulajdonságait.

Többféle stabilitási határt tudunk definiálni, az egyik a már említett horizontális projekciója a tömegközéppontnak, de ez csak egyenletes felületek esetében érvényes. Később ezt a határ kibővítésre került egyenletlen felületekre is.

A *statikus stabilitási határ (Static Stability Margin - S_{SM})* nem más mint a COG projektiójának helyzete és a konvex támasztó-poligon legközelebbi oldalától való távolság. A *longitudinális stabilitási határ (Longitudinal Stability Margin - S_{LSM})* nem más mint a legkisebb távolságok a COG helyzetétől az elülső illetve hátsó oldalaitól a támasztó-poligonnak. Az S_{LSM} egy jó approximációja az S_{SM}-nek, ugyanis könnyebb kiszámolni, mint az S_{SM} stabilitási határt. A probléma ezekkel, hogy a fellépő gyorsulásokat nem veszik figyelembe, ezért definiálták a *rák longitudinális stabilitási határt (Crab Longitudinal Stability MarginS_{CLSM})* amely hasonlít egy rák mozgására.

3. Alkalmazott eszközök

3.1. Yahboom Dogzilla S1 Quadruped Robotkutya

A tanulmány létrejötte egy **YahboomTechnology** [16][17] által fejlesztett robotnak köszönhető mely a **Dogzilla S1** nevet birtokolja.

A **Dogzilla S1** robotkutya rövid leírása:

1. A járása szinte egy valós kutyához hasonlít, 12 szabadságfokkal *(Degrees of Freedom - DOF)*;
2. 12 darab nagy precizitású kormáynmű található rajta;
3. Biztonságos és nem-mérgező aluminium ötvözetből van a váza kialakítva.
4. Egy **Raspberry Pi Model 4B** a fő vezérlőegység, mely támogatja a **Python** alapú programozást, **RViz** illetve **Gazebo** szimulációs környezeteket valamint a távoli számítógépes vezérlést is megvalósítja;
5. Többféle mesterséges intelligencia alapú funkcionalitásokkal rendelkezik (szín felismerés, akadály elkerülés, gépi látás, stb...);
6. Egy fedélzeten található kamera mely felhasználható a fent említett dolgokra;

6. ábra. Dogzila S1 Robotkutya [16]

3.2. Operációs rendszer és távoli elérés

A robot fő vezérlő egysége az említett **Raspberry Pi Model 4B**. Ezen egy **Ubunut 20.04** operációs rendszer fut, melyen támogatott a **VNC Remote Desktop** távoli-asztal elérés, valamint fut egy JupyterLab szerver ahol el tudjuk érni az előre megírt programokat és futtatni tudjuk azt. E mellett SSH *(Secure Sockets Shell)* elérést is biztosít. Érkezik egy alapértelmezett felhasználónévvel, valamint jeszóval, ez mind publikus adat. Ezek nem voltak változtatva a fejlesztés során. A hozzáféréshez szükséges hitelesítő adatokat a következő táblában láthatjuk:

Eszköz	**Felhasználónév**	**Jelszó**
SSH	*pi*	*yahboom*
VNC Remote Desktop	*pi*	*yahboom*
JupyterLab Szerver	*pi*	*yahboom*

1. táblázat. A rendszerhez való csatlakozáshoz szükséges hitelesítési adatok

Ahhoz, hogy tudjunk csatlakozni a robothoz, rendszerindítás *(bootolás)* során létrehoz egy WiFi elérési pontot *(hotspot)* melynek az azonosítója *(SSID)* **DOGZILLA_WIFI** és a jelszava **12345678**.

Az SSH-val való csatlakozáshoz szükséges egy kulcsot generáljunk amely segítségével titkosítva lesz a robottal való kommunikáció. Ez egy fájlba kerül mentésre, mely muszáj ott legyen a számítógépen mellyel csatlakozni akarunk, illetve a Raspberry Pi mikroszámítógépen.

Az SSH csatlakozás a következő parancs futtatásával segítséges egy Shell környezetben:

```
1 ssh pi@192.168.8.88
```

1. kódrészlet. SSH-val való csatlakozás konfiguráció nélkül

Ez után kérni fogja a jelszavat. Ha ezt sikeresen beírtuk, akkor hozzáférésünk lesz egy *tty (TeleTYpewriter)* szesszióhoz.

Ahhoz, hogy könnyen tudjunk csatlakozni, ajánlott egy SSH konfigurációs fájlban elmenteni a csatlakozási adatokat. Tudván, hogy a létrehozott hálózaton a robot lokális lokális címe *(IPv4)* 192.168.8.88, el tudjuk menteni ezt egy SSH konfigurációs fájlba a rendszeren:

```
Host dogzilla
  HostName 192.168.8.88
  User pi
```

2. kódrészlet. SSH konfigurációs fájl tartalma

Ha ez megvan, akkor egy Shell környezetben könnyen tudunk csatlakozni a robotra a következő parancs futtatása segítségével:

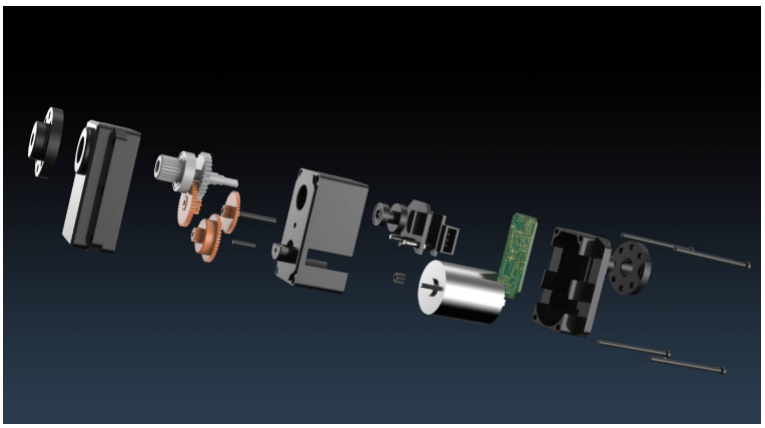
```
ssh dogzilla
```

3. kódrészlet. SSH-val való csatlakozás konfigurációval

Ez esetben jelszavat sem kell megadjunk ugyanis a generált kulcs biztosítja a hitelesítést és automatikusan hozzáférésünk lesz egy *tty* szesszióhoz.

3.3. Használt DC szervo motrok

A Dogzilla S1 moduláris szervo csuklóiban található egy DC motor, reduktor, 12-bites mágneses enkóder és egy integrált motor vezérlő áramkör saját zárt-hurkú vezérléssel, nagysebességű buszrendszer. Ezek mellett a motor pozíciója 360°-ban irányítható, támogatja a sebesség, pozíció, áram illetve hőmérséklet visszacsatolást is. A személyreszabott csuklók esetében változtathatjuk a PID szabályozó paramétereit is.



7. ábra. A használt DC szervo motor ábrázolása [17].

Az említett DC motor paraméterei a következő táblázatban található.

Paraméter	**Érték**
Kimeneti nyomaték	$4.5kg \cdot cm$
Maximális forgási sebesség	$600°/S$
Enkóder pontossága	$0.087°$
Működési feszültség intervallum	$4.8V \sim 7.4V$
Működési hőmérséklet intervallum	$-20°C \sim +60°C$
Elfordulási szög	$0° \sim 360°$

2. táblázat. A felhasznát DC motor paraméterei

Fontos megemlíteni, hogy az I^2C protokollnak köszönhetően, ezek a motrok egymással sorosan vannak kapcsolva, azaz *daisy-chain*-elve vannak egymással. Ez javít hardver szempontjából a komplexitáson, ugyanis nem szükséges annyi ki-/bemenet az STM32-es mikrovezérlőbe ahány motor van, hanem így elég csak 4.

3.4. Fontosabb óvintézkedések és akkumulátor specifikációk

3.4.1. Általános óvintézkedések

1. Ha a robotot lefeküdve találjuk a földön az azt jelenti, hogy az akkumulátor töltöttsége alacsony és már nem tudja azt a szükséges elektromos energiát biztosítani amely szükséges lenne a szervo motrok illetve szenzorok működtetéséhez. A töltő interfész a robot alján található.

2. A töltő csatlakoztatása a robot beavatkozó egységeit (motrok) automatikusan letiltja és kikapcsolja, emiatt a robot vezérlésére nem vagyunk képesek. Ahhoz, hogy az adatvesztést el tudjuk kerülni ajánlott, hogy mentsünk el minden fontos fájlt és csak utána helyezzük töltés alá a robotot.

3. Mielőtt egy *singleton* programot futtatnánk, ajánlott az **APP** nevű program bezárása, amely minden egyes rendszerindítás után automatikusan elindul, ez biztosítja a Bluetooth által való vezérlést a mobil applikáción keresztül. Ha ezt nem végezzü el, akkor zavar léphet fel a robot vezérlése során.

4. A robotkutya teherbírása korlátozott, ezért nem ajánlott külső súlyos perifériákat vagy eszközöket rá elhelyezni.

3.4.2. Akkumulátorral kapcsolatos óvintézkedések

1. Szigorúan tilos olyan berendezéshez csatlakozni, amely meghaladja a termék használati terhelését.

2. Ha az akkumulátor töltöttségi szintje kevesebb mint 9%, a robotkutya lefekszik a földre, és a bekapcsológombon található LED villog. Ebben az állapotban nem vagyunk képesek a robotkutyát irányítani.

3. Ajánlott a kikapcsoló gomb megnyomása töltés előtt biztonsági okok miatt. Alapjáraton töltés alatt a robot egy biztonságos állapotba helyezi magát és ez idő alatt nem tudjuk használni.

4. Tartsuk távol hőforrásoktól, tűztől, folyadékoktól és ne használjuk nedves vagy esős környezetben. Egy nagy páratartalmú környezetben való működtetés könnyen rövidzárlathoz vezethet.

5. Az optimális megengedett hőmérséklet a környezetben nem más mint $0° \sim 35°$ között van. Az akkumulátor, vagy akkumulátortöltő, stabilitása ezen az intervallumon kívül jelentősen csökkenhet.

6. Nem ajánlott a robot működtetése egy olyan környezetben ahol jelentősen nagya a statikus elektromos tér, vagy erős mágneses tér, amely befolyásolhatja a robot működését illetve stabilitását. Hosszú ideig való tartózkodás egy ilyen környezetben jelentősen károsíthatja a robotot.

3.5. A robot szoftver architektúrája

A Raspberry Pi mikroszámítógépen egy Ubuntu 20.04 operációs rendszer fut mely biztosítja a robot működését. Lehetséges mobiltelefonos alkalmazáson keresztül is vezérelni, valamit saját Python, C++ illetve Java programok segítségével.

Többféle programozási lehetőséget is megvalósít, ilyen például a **ROS 2 Foxy** platformon alapuló programozás. A háttérben ez vezérel minden szükséges folyamatot illetve a hardverrel is ezen keresztül tudunk kommunikálni. A ROS 2 kommunikációs architektúra a *Feliratkozó-Téma-Feladó (Topic-Subscriber-Publisher)* architekrúrán alapszik, ahol a létrehozott csomópontok *(node)* egymással kommunikálnak valamilyen kommunikációs protokoll segítségével, ilyen a *Téma (Topic)*, *Akció (Action)* valamint *Szolgáltatás (Service)* protokollok. Az adatokat egy úgynevezett *interfészbe (Interface)* csomagoljuk és ez adódik át csomópontok között.

Egy másik programozási lehetőség kicsivel egyszerűbb, a már meglévő **DOGZILLALib**[16] Python könyvtár felhasználása, amellyel ugyan azt tudjuk elérni mint a fennebb említett ROS 2 architektúrával, csak sokkal egyszerűbben és direkt módon történik a hardverrel való kommunikáció, nincs részfeladatokra/részcsomópontokra osztva.

3.5.1. Kommunikációs protokoll

A hardverrel való kommunikáció soros portokon keresztül történik. A standard TTL soros kommunikációs módszer van felhasználva.

A következő táblázat a szoftver interfészt írja le.

Baud ráta	**Adat bit**	**Stop bit**	**Paritás bit**
115200	8	1	Nincs

3. táblázat. Szoftver interfész szerkezete.

Az adat keret *(data frame)* formátuma a következő.

Keret fejléc	Fix: $0x55\ 0x00$
Keret hossz	Az egész keret hossza byte-ban kifejezve.
Adat	Az utasítás típusától függ, változó utasításonként.
Ellenörző összeg (checksum)	Az adat bájtjainak összege, a legkisebb byte megfordítva.
Keret végét jelző byte	Fix: $0x00\ 0xAA$

4. táblázat. Adat keret szerkezete.

Két típusú utasítást különböztethetünk meg: *írás* és *olvasás*. Az írás esetében nem szükséges sikeres írás esetén válasz generálása **(NO ACK)**, viszont az olvasás esetében szükséges **(ACK)**.

Írás *(0x01)*: Az írás *(WRITE)* parancs az első címtől kezdve folyamatosan módosítja az adatokat anélkül, hogy választ generálna.

Fejléc	**Hossz**	**Típus**	**Első cím**	**Adat**	**Checksum**	**Vége**
$0x55\ 0x00$	(hossz)	$0x01$	(cím)	(adat)	(checksum)	$0x00\ 0xAA$

5. táblázat. Írás típusú utasítás szerkezete.

Olvasás *(0x02)*: Az olvasás *(READ)* parancs folyamatosan olvassa az adatokat az első címről. A következő az elküldött adatcsomag szerkezete.

Fejléc	**Hossz**	**Típus**	**Első cím**	**Olvasás hossza**	**Checksum**	**Vége**
$0x55\ 0x00$	(hossz)	$0x02$	(cím)	*uint_8*	(checksum)	$0x00\ 0xAA$

6. táblázat. Olvasás típusú elküldött csomag szerkezete.

Ha sikeres a szükséges adat lekérése/beolvasása, a következő adatcsomagot kapjuk válaszként.

Fejléc	**Hossz**	**Típus**	**Első cím**	**Adat**	**Checksum**	**Vége**
$0x55\ 0x00$	(hossz)	$0x012$	(cím)	(adat)	(checksum)	$0x00\ 0xAA$

7. táblázat. Olvasás típusú utasítás válaszaként érkező adatcsomag szerkezete.

3.5.2. A Dogzilla S1 robot utasításkészlete *(Memory Table)*

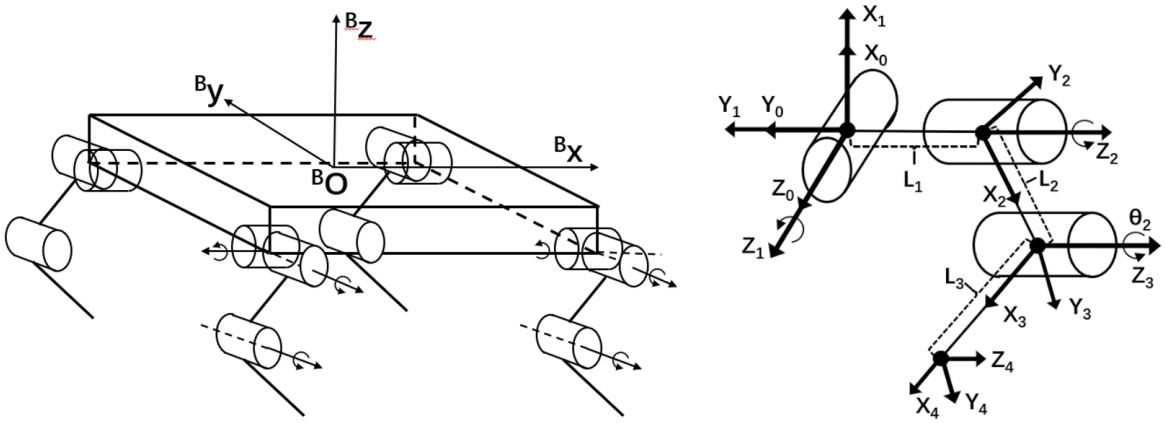
Ahhoz, hogy egy műveletet végre tudjunk hajtani, szükséges ismerjük, hogy a memóriában hogyan, illetve hol van eltárolva az adott utasítás. Ezek mellett szükséges ismerjük a hozzá tartozó paramétereket is. A fontosabb műveletek leírása a következő táblázatban található.[17]

Cím	**Művelet**	**Írás/Olvasás**	**Kezdeti érték**
$0x00$	Munkafeltétel	Olvasás	$0x00$
$0x01$	Akkumulátor szint	Olvasás	$0xFF$
$0x03$	Teljesítmény üzemmód	Írás	$0x00$
$0x04$	Kalibrációs üzemmód	Írás	$0x00$
$0x21$	Motrok nullpozícióba állítása	Írás	$0x00$
$0x30$	Előre/Hátra való mozgás	Írás/Olvasás	$0x80$
$0x31$	Balra/Jobbra való mozgás	Írás/Olvasás	$0x80$
$0x32$	CW és CCW forgási sebesség	Írás/Olvasás	$0x80$
$0x33$	Test transzlációs távolsága X irányban	Írás/Olvasás	$0x80$
$0x34$	Test transzlációs távolsága Y irányban	Írás/olvasás	$0x80$
$0x35$	Test magassága	Írás/Olvasás	$0x80$
$03C$	Egy helyben állás	Írás/Olvasás	$0x00$
$0x3D$	Mozgás üzemmód	Írás/Olvasás	$0x00$
$0x61$	IMU szenzor állapota	Írás/Olvasás	$0x00$
$0x62$	Billenés (Roll) szög	Olvasás	$0x01$
$0x63$	Bólintás (Pitch) szög	Olvasás	$0x02$
$0x64$	Elfordulás (Yaw) szög	Olvasás	$0x03$

8. táblázat. Dogzilla S1 Robot utasításkészlete. (1)

3.6. A robot geometriája és koordináta rendszere

A Dogzilla S1 robotnál használt koordináta-rendszerek a következő ábrán található.



8. ábra. Robot koordináta-rendszerei. [17]

Mielőtt bővebben kifejtésre kerülne a robot matematikai leírása, szükséges néhány fogalom tisztázása.

3.6.1. A robot méretezése

A Dogzilla S1 egyszerűsített modellje a következő, Webots szimulációs környezetben megvalósítva.

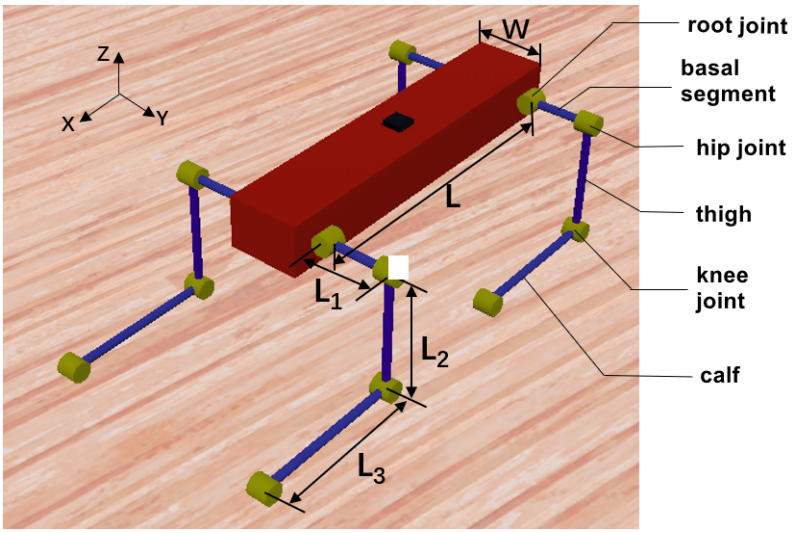
Láthatjuk, hogy a robot merev testekből *(rigidbody/solid)* áll amelyek csuklók segítségével csatlakoznak egymáshoz. A robotikában több fajta csukló létezik, például rotációs *(rotational)*, csavarodó *(twisting)*, *revolute* csuklók, illetve transzlációs csuklók (lineáris valamint ortogonális).

Megfigyelhetjük, hogy csakis rotációs csuklók találhatók a robot architektúrájában. Ezek primitív csuklók, szóval csak egyetlen egy szabadságfokot képesek hozzáadni a robot szerkezetéhez, a forgásirány a csuklótengely körül történik. Ez lehetővé teszi egy (vagy több) szegmens relatív elforgatását bizonyos szöggel és szögsebességgel.

A robot méretezése a következő táblázatban található:

Elnevezés	**Jelölés**	**Érték**
A test szélessége	W	$45[mm]$
A test hosszúsága	L	$150[mm]$
Gyökér csukló *(root joint)*	q_1	$\theta_1[rad]$
Bázis szegmens *(basal segment)*	L_1	$31.6[mm]$
Csípő csukló *(hip joint)*	q_2	$\theta_2[rad]$
Comb szegmens *(thigh segment)*	L_2	$60[mm]$
Térd csukló *(knee joint)*	q_3	$\theta_3[rad]$
Láb szegmens *(calf segment)*	L_3	$75[mm]$

9. táblázat. Dogzilla S1 méretezése



9. ábra. Dogzilla S1 egyszerűsített modellje Webots szimulációs környezetben. [17]

4. Tervezés

4.1. Direkt geometriai feladat megoldása

Mostmár minden szükséges eszköz megvan a robot architektúrájának tárgyalásához robotikai szempontból. Egy ilyen szempont a direkt geometriai feladat, azaz a csuklóváltozók teréből meghatározzuk a végberendezés, ez esetben láb végpontjának a pontos pozícióját és orientációját. Vegyük figyelembe az Ábra 8-et. A B koordináta-rendszer a robot báziskoordináta-rendszerét jelöli, amely a robot tömegközéppontjában található. A jobb oldali ábrán megfigyelhetjük egy láb architektúráját, amely három darab egymásra merőleges rotációs csuklóból áll. A Denavit-Hartenberg tálba felírásával meghatározhatjuk egy láb végpontjának a pozícióját.

K_i	θ_i	d_i	a_i	α_i
1	θ_1	0	0	0
2	θ_2	L_1	$\pi/2$	0
3	θ_3	0	L_2	0
4	θ_4	0	L_3	0

10. táblázat. Egy láb D-H táblázata.

Homogén transzformáció általánosítása a robotra

A (8) képlet alapján fel tudjuk írni az általános homogén transzformációs mátrixot.

$$T_i^{i-1} = \left(\begin{array}{ccc:c} c\theta_i & -s\theta_i & 0 & a_{i-1} \\ s\theta_i c\alpha_{i-1} & c\theta_i c\alpha_{i-1} & -s\alpha_{i-1} & -d_i s\alpha_{i-1} \\ s\theta_i s\alpha_{i-1} & c\theta_i s\alpha_{i-1} & c\alpha_{i-1} & d_i c\alpha_{i-1} \\ \hdashline 0 & 0 & 0 & 1 \end{array}\right) \tag{9}$$

A (9) egyenletbe behelyettesítve a D-H paramétereket megkapjuk a következő

homogén transzformációs mátrixokat [17]:

$$T_1^0 = \begin{pmatrix} c\theta_1 & -s\theta_1 & 0 & 0 \\ s\theta_1 & c\theta_1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \qquad T_2^1 = \begin{pmatrix} c\theta_2 & -s\theta_2 & 0 & 0 \\ 0 & 0 & -1 & -L_1 \\ s\theta_2 & c\theta_2 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$T_3^2 = \begin{pmatrix} c\theta_3 & -s\theta_3 & 0 & L_2 \\ s\theta_3 & c\theta_3 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \qquad T_4^3 = \begin{pmatrix} 1 & 0 & 0 & L_3 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$T_B^0 = \begin{pmatrix} 0 & 0 & 1 & L/2 \\ 0 & -1 & 0 & W/2 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

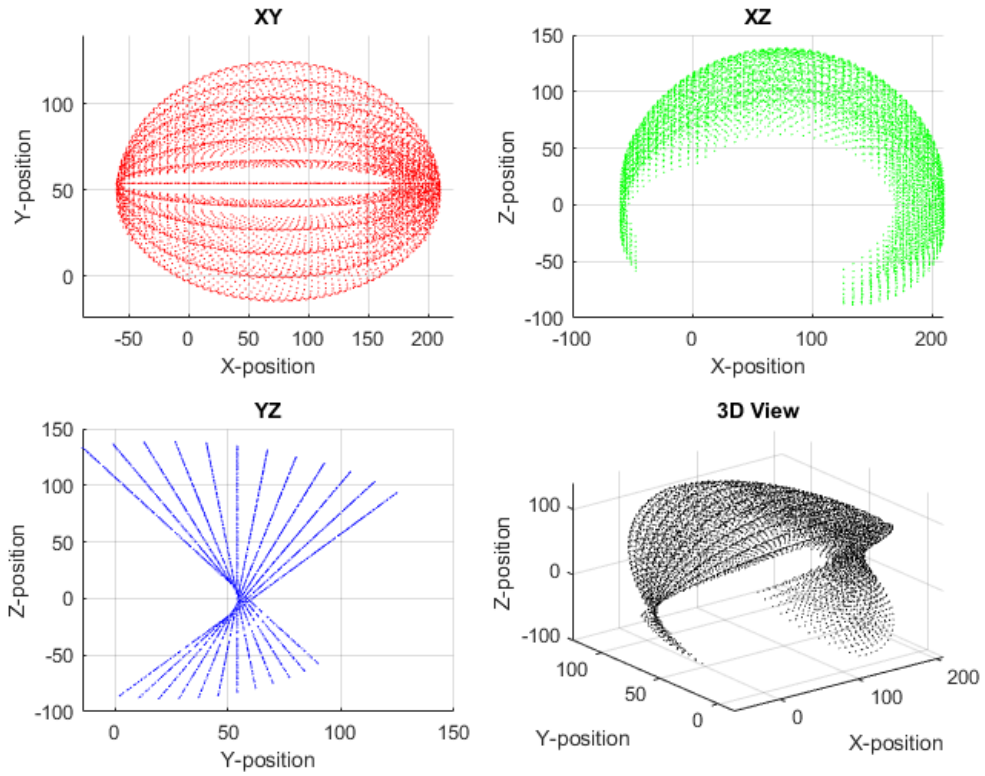
$$\Rightarrow T_B^4 = T_B^0 \cdot T_1^0 \cdot T_2^1 \cdot T_3^2 \cdot T_4^3 = \begin{pmatrix} & & & P_x \\ & R & & P_y \\ & & & P_z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Fontos megjegyezni, hogy a homogén transzformációk a szegmensek hossza illetve a csuklókonfigurációk függvényében történnek. A T_B^4 homogén transzformációs mátrixból fel tudjuk írni a rotációs mátrixot, illetve a transzlációs vektort.

$$R = \begin{pmatrix} S_{23} & C_{23} & 0 \\ -S_1C_{23} & S_1S_{23} & C_1 \\ C_1C_{23} & -C_1S_{23} & S_1 \end{pmatrix} \tag{10}$$

$$P = \begin{pmatrix} P_x \\ P_y \\ P_z \end{pmatrix} = \begin{pmatrix} L/2 + L_3 \cdot S_{23} + L_2 \cdot S_2) \\ W/2 + L_3 \cdot (S_1 \cdot S_2 \cdot S_3 - C_2 \cdot C_3 \cdot S_1) + L_1 \cdot C_1 - L_2 \cdot C_2 \cdot S_1 \\ L_1 \cdot S_1 - L_3 \cdot (C_1 \cdot S_2 \cdot S_3 - C_1 \cdot C_2 \cdot C_3) + L_2 \cdot C_1 \cdot C_2 \end{pmatrix} \tag{11}$$

Ezeknek ismeretében, képesek vagyunk meghatározni egy adott láb munkaterét.



10. ábra. Egy láb munkatere MATLAB segítségével generálva.

Megjegyzés: Fontos megjegyezni, hogy a megadott transzformációs mátrixok csak egy lábra (bal első lábra) érvényesek. A láb elhelyezkedése függvényében ezek változnak.

4.2. Inverz geometriai feladat megoldása

A direkt geometriai feladat megoldásával ellentétben, az inverz geometriai feladat választ ad arra a kérdésre, hogy hogyan kell megválasszuk a csuklókonfigurációt ismerve a végberendezés, ez esetben a láb végpontjának a világ-koordinátákban, vagy a robot testéhez relatív koordináta-rendszerben a pozícióját illetve orientációját.

$$\underline{P} \xrightarrow{\text{Inverz geometriai feladat}} \underline{q}$$
$$\underline{P} \xleftarrow[\text{Direkt geometriai feladat}]{} \underline{q}$$

Az inverz geometriai feladat megoldása nem triviális, nem tudjuk általánosítani a megoldást mint a direkt geometriai feladat esetében. Szükséges geometriai intuíció és logika.

Csuklókonfiguráció meghatározása

Többféle megoldási módszer létezik az inverz geometriai feladat megoldására. Egy kevésbé egyszerű megoldás a direkt geometriai feladatból kiindulva algebrai módszerekkel a θ_1, θ_2 illetve θ_3 szögek meghatározása [1].

Egy ennél egyszerűbb megoldás, ha kiindulunk a direkt geometriai feladatból és megpróbálunk egy egyenletrendszert felírni amely leírja a lábvég térbeli koordinátáit és ez alapján már könnyen, numerikus módszerekkel, meg tudjuk határozni a szögeket.

A következő egyenletrendszer leírja a lábvég térbeli változását a szögek függvényében.

$$\underline{P} = \begin{cases} P_x = L/2 + L_3 \cdot S_{23} + L_2 \cdot S_2 \\ P_y = W/2 + L_3 \cdot (S_1 \cdot S_2 \cdot S_3 - C_2 \cdot C_3 \cdot S_1) + L_1 \cdot C_1 - L_2 \cdot C_2 \cdot S_1 \\ P_z = L_1 \cdot S_1 - L_3 \cdot (C_1 \cdot S_2 \cdot S_3 - C_1 \cdot C_2 \cdot C_3) + L_2 \cdot C_1 \cdot C_2 \end{cases} \tag{12}$$

A megoldás MATLAB környezetben az *fsolve* függvény segítségével történik, valamint Python környezetben a *SciPy* könyvtárból az *Optimize* almodulból a *least_squares* függvény segítségével.

4.3. Periodikus járás generálása

Mielőtt a lépések generálása kerülhetne tárgyalásra, fontos megérteni azt, hogy igazából mi is az a lépés. A robotikában alkalmazott lépés fogalma szorosan kötődik azzal amelyet az állatvilágban megfigyelhetünk több lábú állatok esetében, és sok esetben a valóság alapján van modellezve robotokra is. Többféle lépés létezik. A *periodikus* lépés magától értetődő, ezzel ellentétben az aperiodikus lépés már nem. Akkor beszélünk aperiodikus lépésről amikor más testtartásra kell váltson a robot egy egyenletlen támasztó-felületnek köszönhetően és a periódikusság megszakad.

Járás (Gait): A járást nem más mint egy láb felemelési helye és ideje határozza meg, koordinálva a test mozgásával 6 szabadságfok szerint, annak céljából, hogy a test elmozdításra kerüljön egyik helyzetből a másikba [11].

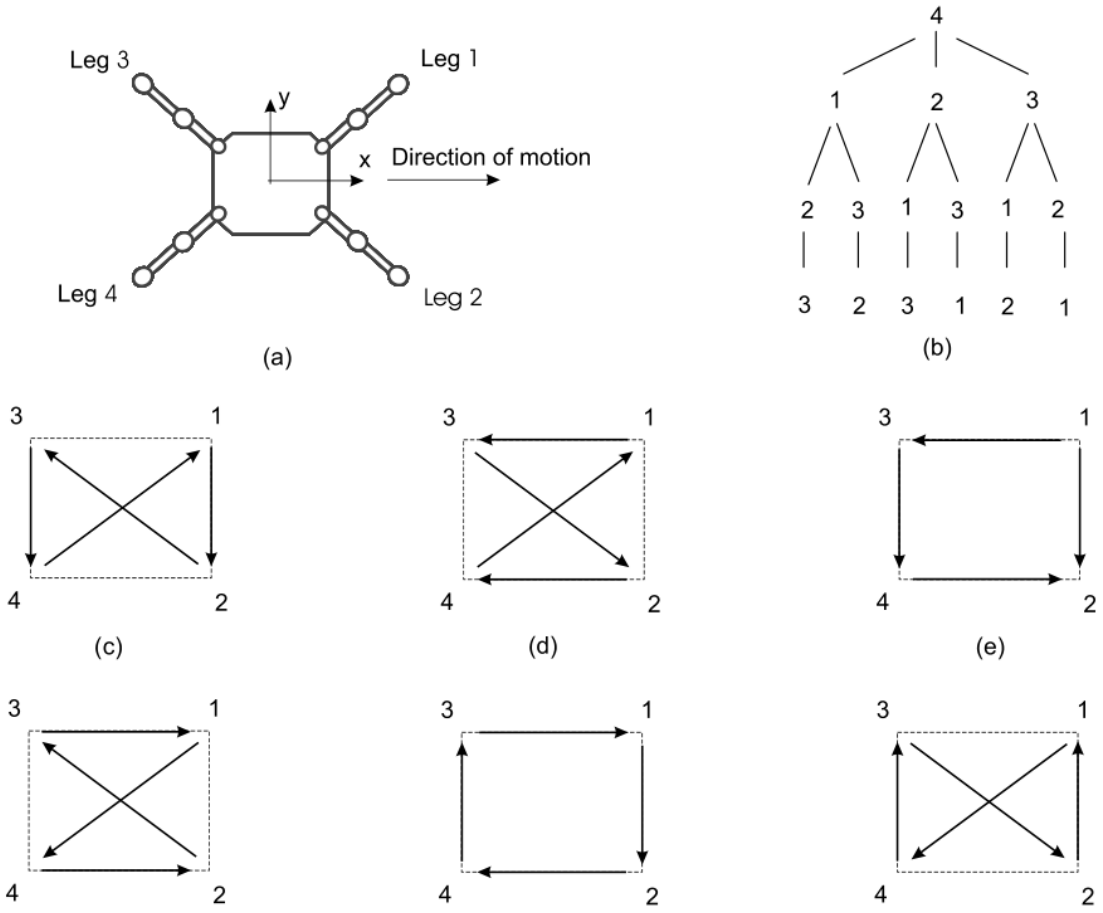
Esemény (Event): Egy esemény nem más mint egy láb felemelése, vagy lerakása. Egy n lábú robot esetében az i.-ik láb lerakása az i.-ik eseménnyel van meghatározva, valamint az i.-ik láb felemelése pedig $i+n$.-ik eseménnyel. Összesen $2n$ esemény létezik egy n lábú robot esetében. Ha két esemény időben párhuzamosan történik (egy időben) akkor azt *szinguláris esemény/lépés*nek *(singular gait)* nevezzük.

Esemény szekvencia (Event sequence): Egymást követő események sorozata.

A mi esetünkben folytonos és szaggatott lépések generálásáról lesz szó. Szükséges a következő kritériumokat szem előtt tartsuk a tervezéssorán [3].

Technikai kritériumok	**Teljesítési szükséglet**
Manőverezőképesség	✓
Keresztirányú képesség	✓
Irányíthatóság	X
Terepföld	✓
Hatékonyság	✓
Stabilitás állandó biztosítása	X
Költséghatékonyság	X
Akadályelkerülés	✓

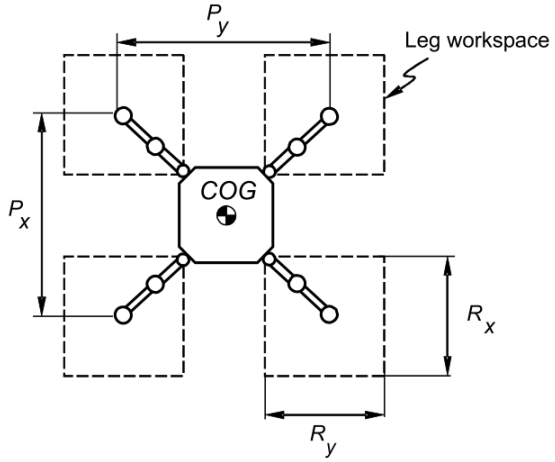
11. táblázat. Tervezési kritériumok



11. ábra. Lehetséges lépés esemény-szekvencia egy négylábú robot esetében [7].
a.) A robot felülről nézve.
b.) Lépés szekvencia gráfja
c.) - **h.)** Esemény szekvencia

A fenti ábrán láthatunk egy lehetséges lépés esemény-szekvenciát egy négylábú robot esetében [7]. Az x-tengely irányába történik a mozgás. Többféle lehetséges ilyen szekvencia létezik. Hat darab osztályra tudjuk osztani a lépéseket: $\pm X$ és $\pm Y$, a mozgást irányának függvényében és $\pm O$ az origó körüli forgás esetében.

4.4. Folytonos lépések geometriai definíciója



12. ábra. Lépés geometriai meghatározása [7].

Vegyük figyelembe a fenti ábrát. Ideális esetben dolgozunk, feltételezzük, hogy a robot lábainak tömege elhanyagolható, így az emiatt fellépő zajok is (inercia hatásai, nyomatékok, surlódások) eltűnnek.

Aktív ciklusidő (Duty Cycle): β_i, az az idő amennyit az i.-ik láb kontaktusban van a támasztó-felülettel az egész ciklus ideje alatt. Ha β_i egyenlő minden láb esetében, akkor szabályos lépésről *(regular gait)* beszélünk.

Láb fázisa (Leg phase): Φ_i, az a normalizált idő amennyi idővel az i.-ik láb elhelyezése késik az 1-es lábhoz képest (az 1-es láb általában a referencia láb által van kezelve).

Láb löket (Leg stroke): R, az a távolság amellyel a láb elmozdul a testhez relatívan a támasztó fázisban (kontaktban van a támasztó-felülettel). R muszáj a láb munkaterén belül tartózkodjon amely R_x és R_y értékekkel van definiálva.

Löket távolság (Stroke pitch): P, a távolság két egymással szomszédos láb stroke középpontjai között. P_x a távolság az egymással szemben található *(collateral)* középpontok között, míg P_y az egymással szemben található *(contra-lateral)* középpontok között.

Lépés hossz (Stride length): λ, az a távolság amelyet a tömegközéppont *(COG)* megtesz egy lépés ciklusban. Ha a lépés periodikus, akkor:

$$\lambda = \frac{R}{\beta} \tag{13}$$

A stabilitásra visszatérve, fontos megemlítenünk a longitudinális stabilitási margót (S_{LSM_C}). A longitudinális stabilitási margó megmondja a quadruped robot stabilitását a ciklusidő függvényében egy adott lépéshosszon. Intuitívan átgondolva, minél több időt van egy láb támasztó módban, annál stabilabb lesz a robot.

Longitudinális stabilitási margó (folytonos és szaggatott)

Általánosítva, legyen a $+X$ irányú láb löket (R_x) hossza 1 méter illetve legyen az aktív ciklusidő a $[0.75, 1]$ intervallumon értelmezve. A longitudinális stabilitási margót a szaggatott (*discontinuous/wave*) járásra a következő képlet írja le:

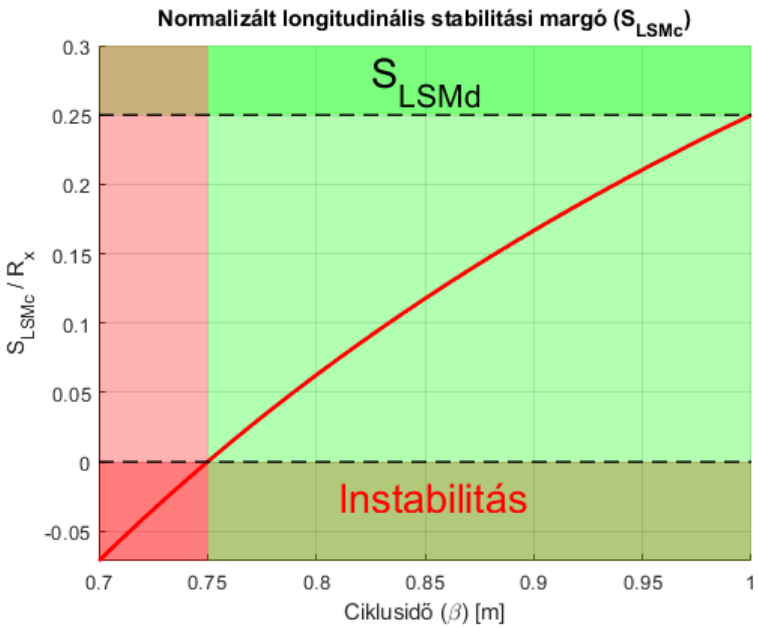
$$S_{LSM_C} = \left(\beta - \frac{3}{4}\right) \cdot \frac{R_x}{4} \tag{14}$$

Ha normalizálni szeretnénk a folytonos longitudinális stabilitási margót, akkor elosztjuk a láb lökettel.

$$S_{LSM_{CN}} = \frac{S_{LSM_C}}{R_x} \tag{15}$$

A szaggatott típusú járás longitudinális stabilitási margót a következő képlet írja le:

$$S_{LSM_D} = \frac{R_x}{4} \tag{16}$$



13. ábra. Longitudinális stabilitási margó

4.5. Két fázisú szaggatott járás periódusa

A nem folytonos járást végrehajtó robot átlagos sebességét az R_x láblöket és a T periódus határozza meg. Nem folytonos járás esetén a periódust az egyes lábak részfázisainak és a testmozgások időinek összege adja. Legyen a járás $+X$ irányú.

Szaggatott járás periódusa

Feltételezzük, hogy a lábvég egy négyszög alakú pályát ír le (lásd 20. ábra), akkor a következő összefüggés megadja egy ilyen szaggatott járásmódnak a periódusát [7].

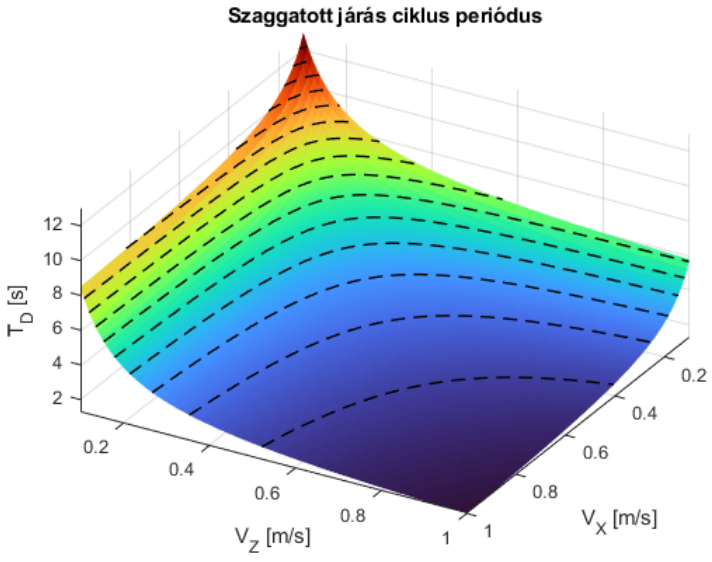
$$T_D = 4\left(t_L + t_F + t_P\right) + 2t_{BP} \tag{17}$$

Ahol t_L az az idő amely alatt a láb a támasztó felülettől emelkedik, t_F az az idő ahol a lábvég előre $(+X)$ irányba halad, t_P az az idő amely alatt a lábvég a támasztó felületre kerül és t_{BP} az az idő ami alatt a robot váza előre mozog.

Ha ismert a lépés magassága (h vagy Z_m), a láblöket (R_x) és az X és Z irányú sebességek (V_X illetve V_Z), akkor fel tudjuk írni a járásnak a periódusát mint:

$$T_D = 4\left(2\frac{h}{V_z} + \frac{R_x}{V_x}\right) + \frac{R_x}{V_x} = \frac{8hV_x + 5R_xV_z}{V_xV_z} \tag{18}$$

Az X illetve Z irányú sebességek függvényében a járási periódus idő a 14. ábrán látható. Várhatóan, minél kisebb a sebesség, annál nagyobb járási periódust kapunk.



14. ábra. Szaggatott járás periódusa X és Z sebességek függvényében.

4.6. Két fázisú folytonos járás periódusa

Folytonos járásban, egy láb $t_s = \beta T_C$ ideig van támasztó fázisban, illetve $t_t = (1-\beta)T_C$ ideig transzfer fázisban, ahol T_C nem más mint a járás ciklus periódusa (lásd 21. ábra). Legyen a járás $+X$ irányú.

Folytonos járás periódusa

Tételezzük fel azt, hogy a folytonos illetve szaggatott járásmódban a lábvég ugyan azt a pályát írja le, valamint azonos ciklusidővel rendelkeznek ($\beta = 0.75$). Fel tudjuk írni egy láb transzfer idejét mint:

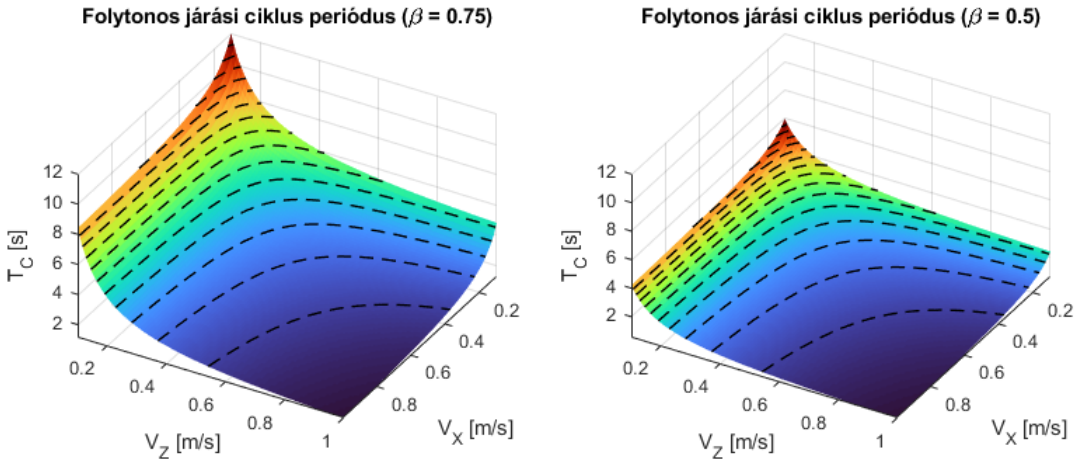
$$t_t = \left(\frac{2h}{V_z} + \frac{R_x}{V_x}\right) \tag{19}$$

Amely segítségével ki tudjuk fejezni a folytonos járás ciklus periódusát, melyet a következő összefüggés határoz meg:

$$T_C = \frac{1}{1-\beta} t_t = \frac{1}{1-\beta}\left(\frac{2h}{V_z} + \frac{R_x}{V_x}\right) \tag{20}$$

A feltételezés, hogy ugyan azt a pályát követi a lábvég folytonos illetve szaggatott járás közben azt eredményezi ($\beta = 0.75$), hogy a folytonos illetve szaggatott járások periódusai között nincs eltérés (lásd 14. ábra illetve 15.a. ábra), azaz $T_D = T_C$.

$$T_D = T_C \iff \frac{8hV_x + 5R_xV_z}{V_xV_z} = \frac{1}{1-\beta}\left(\frac{2h}{V_z} + \frac{R_x}{V_x}\right) \tag{21}$$



15. ábra. Folytonos járás periódusa X és Z sebességek függvényében változó aktív ciklusidőre. a.) $\beta = 0.75$, b.) $\beta = 0.5$

4.7. Folytonos illetve szaggatott járás sebessége

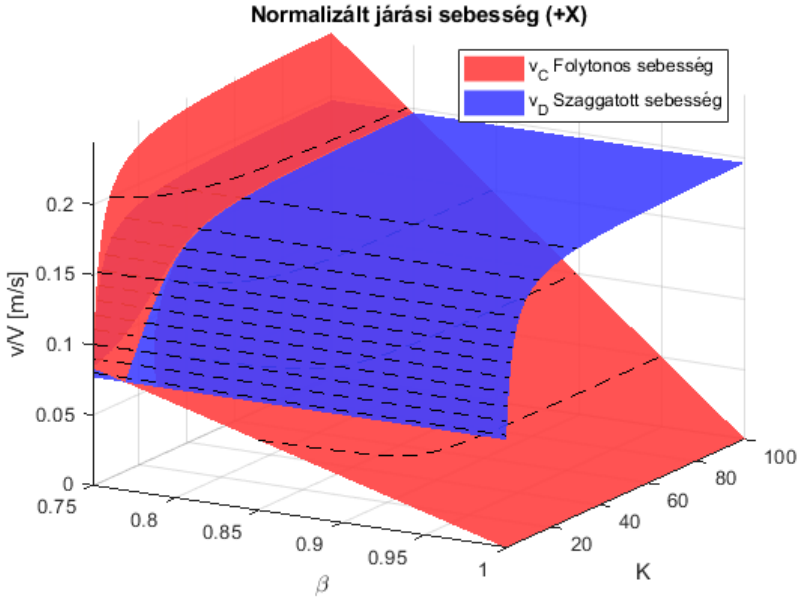
Folytonos illetve szaggatott járás sebessége

A 18. illetve 20. egyenletek megmondják a két különböző járásmód ciklus periódusát. Legyen az elmozdulás folytonos járás esetén λ, valamint R_x szaggatott járás esetén. Legyen a sebesség folytonos járás esetén v_C, valamint szaggatott járás esetén pedig v_D. Ennek tudatában fel tudjuk írni a következő összefüggéseket:

$$v_C = \frac{\lambda}{T_C} = \frac{\lambda(1-\beta)V_xV_z}{2hV_x + R_xV_z} \tag{22}$$

$$v_D = \frac{R_x}{T_D} = \frac{R_xV_xV_z}{8hV_x + 5R_xV_z} \tag{23}$$

Ha elvégezzük a következő feltételezéseket, akkor 16. ábrán megfigyelhetjük a normalizált sebességet ($+X$ irány) az aktív ciklusidő valamint a K paraméter függvényében: legyen $V_x = V_z = V$ illetve $h = R_x/K$, ahol a K paraméter a h illetve R_x paraméterek között teremt összefüggést. Ez két felületet eredményez amely egy görbe mentén metszi egymást. Ezen a görbén a folytonos illetve szaggatott járási sebességek azonosak, azaz nem tudunk megkülönböztetést tenni folytonos vagy szaggatott járás között.

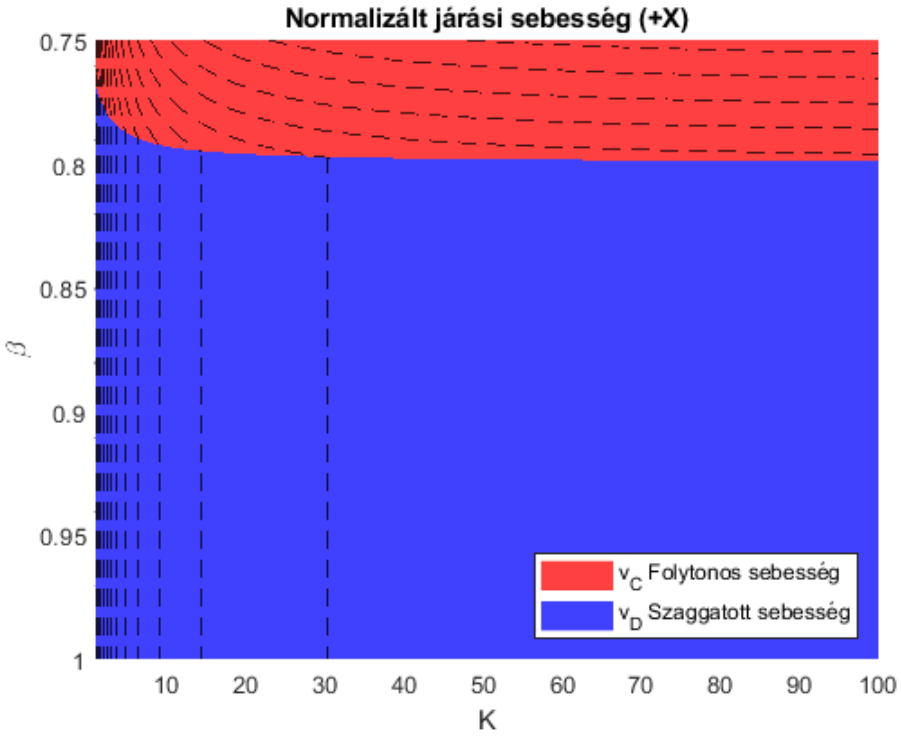


16. ábra. Folytonos illetve szaggatott járási sebességek

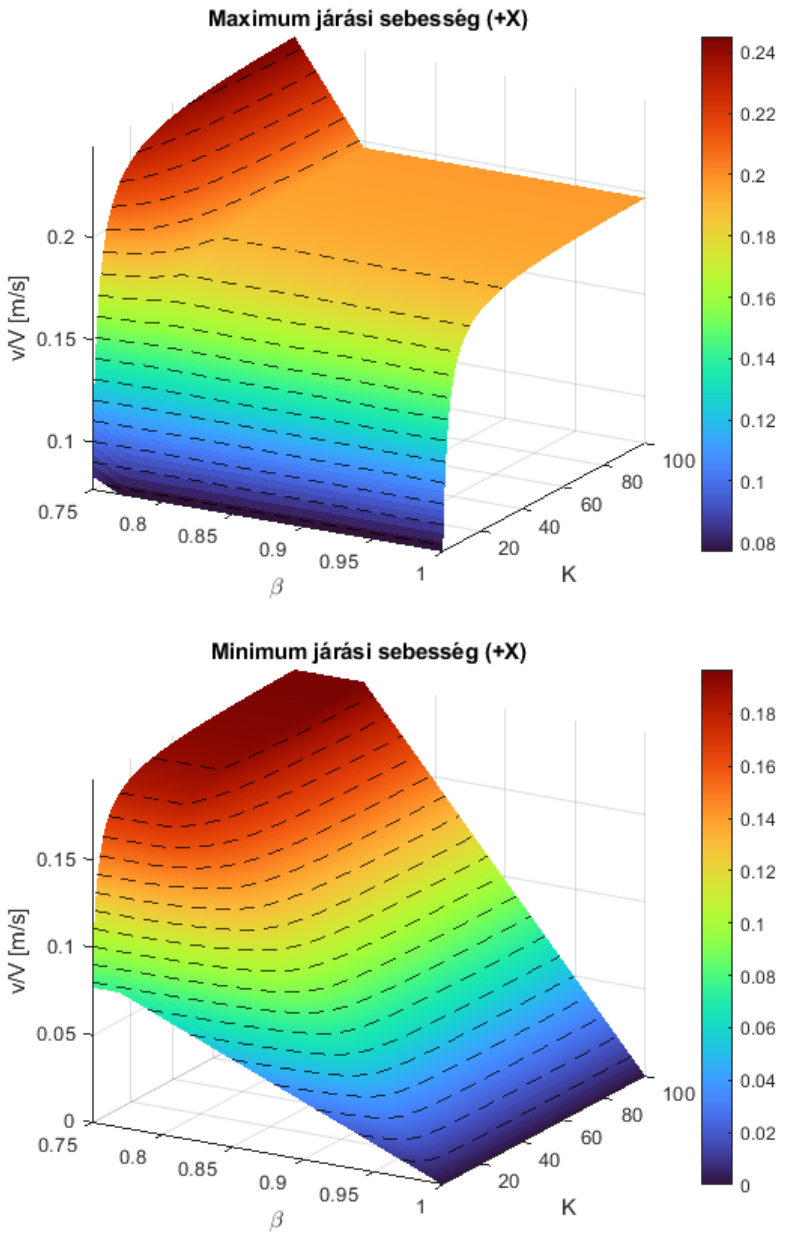
A 16. ábrán megfigyelhetjük azt, hogy a folytonos járás sebessége az aktív ciklusidő függvényében csökken, valamint a szaggatott járás sebessége a ciklusidőtől független. Ez azt jelenti, hogy akármilyen aktív ciklusidőt választunk az adott lábra, nem fogja befolyásolni a szaggatott járás sebességét.

A 17. ábra szemlélteti azt, hogy milyen paraméterkonfigurációk $((\beta, K) \rightarrow V)$ esetében ajánlott folytonos vagy szaggatott járási algoritmust választani. Amennyiben nagyobb sebességet akarunk elérni, mint $\sim 0.2m/s$, ez esetben a piros régióban kell megválasztani a paramétereket és folytonos járási algoritmust megválasztani. Emellett, ha állandó sebességet akarunk elérni, akkor a kék régióban kell megválasztani a paramétereket és szaggatott járási algoritmust tervezni (ajánlott ha $K \geq 30$ ugyanis abban a régióban a leglaposabb a felület).

Azonban, ha kisebb sebességeket akarunk elérni, akkor szintén ajánlott a piros régióban mozogni amely a kék felület alatt található (lásd 16. ábra) és folytonos járási algoritmust választani, ugyanis ha szaggatott járást akarunk megvalósítani, akkor a kék régió meredeksége K paraméter szerint megnehezíti a paraméterkonfiguráció megválasztását ugyanis nagyon érzékeny lesz a K paraméter értékére.



17. ábra. Folytonos illetve szaggatott járási sebességek felülnézeből



18. ábra. Minimum illetve maximum felülete a folytonos és szaggatott járási sebességeknek

Ha azt szeretnénk elérni, hogy a járási sebesség a lehető legnagyobb legyen, akkor a 18. ábrán a felső felületen kell paramétert megválasztani. Viszont, ha kisebb sebességeket akarunk elérni, akkor szintén a 18. ábrán az alsó felületen kell, hogy mozogjunk. A 18. ábra ugyan nem mondja meg, hogy szaggatott vagy folytonos járási módszert kell választani, csak egy referencia sebességhez szükséges paraméterek megválasztásában segít.

4.8. Események időzítése egy lépés ciklusban

Vegyük figyelembe a következő egyenletet.

$$\phi_1 = 0 \tag{24}$$

$$\phi_2 = \frac{1}{2} \tag{25}$$

$$\phi_3 = \beta \tag{26}$$

$$\phi_4 = F\left(\beta - \frac{1}{2}\right) \tag{27}$$

Ahol F egy törtfüggvény. Egy törtfüggvény $Y = F(x)$ egy valós $X \in \mathbb{R}$ szám esetében a következőképp van definiálva:

$$Y = \begin{cases} \text{Törtrésze } X\text{-nek, ha } X \geq 0, \\ 1 - \text{Törtrésze } X\text{-nek, ha } X < 0. \end{cases} \tag{28}$$

Ez egy $+X$ típusú lépést valósít meg.

4.9. Lépésfüggvény számítása

A négylábú robot lábvégének pályatervezése fontos része a járástervezésnek. A lábvég pályája befolyásolja a négylábú robot lépésének pontosságát, a járás stabilitását, valamint a lábvég ütőerejét, és befolyásolja a négylábú robot akadályleküzdési képességét is. A periodikus járásban a négylábú robot mozgása gyorsaságra törekszik, így a lábvég sebessége viszonylag nagy, és a lábvégi pályatervezés optimális célpályájának simasága érhető el.

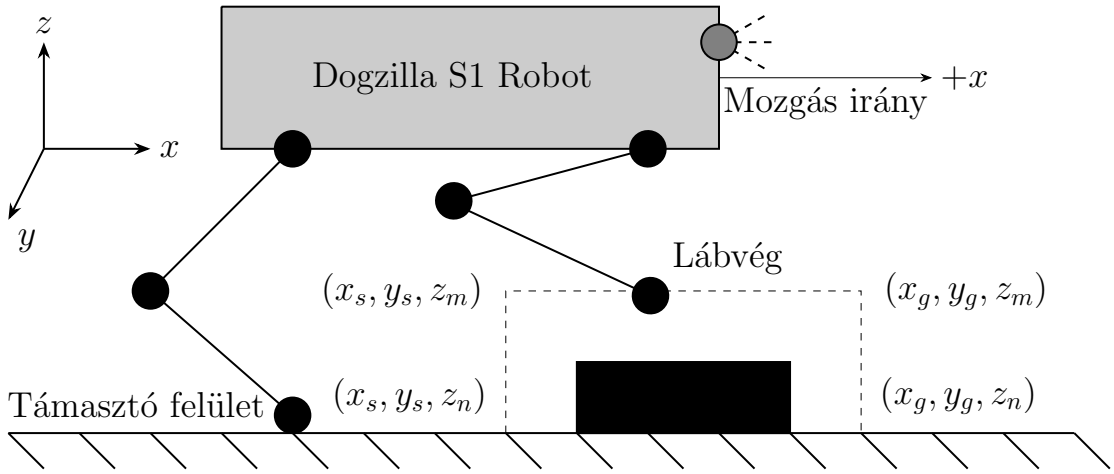
Tételezzük fel, hogy a robot egy akadályt akar kikerülni. Ugyan az az eljárás mint a sima lépésfüggvény meghatározásánál, csak a paraméterek változnak. Vizsgáljuk meg a következő függvényeket.

$$F_x(t) = \begin{cases} x_s & , t \in \left[0, \frac{T_m}{4}\right) \\ x_s + \frac{4 \cdot (x_g - x_s)}{T_m} \cdot \left(t - \frac{T_m}{4}\right) & , t \in \left[\frac{T_m}{4}, \frac{T_m}{2}\right) \\ x_g & , t \in \left[\frac{T_m}{2}, T_m\right) \end{cases} \tag{29}$$

$$F_y(t) = \begin{cases} y_s & ,t \in \left[0, \frac{T_m}{4}\right) \\ y_s + \frac{4 \cdot (y_g - y_s)}{T_m} \cdot \left(t - \frac{T_m}{4}\right) & ,t \in \left[\frac{T_m}{4}, \frac{T_m}{2}\right) \\ y_g & ,t \in \left[\frac{T_m}{2}, T_m\right) \end{cases} \tag{30}$$

$$F_z(t) = \begin{cases} z_n + \frac{4 \cdot z_m}{T_m} \cdot t & ,t \in \left[0, \frac{T_m}{4}\right) \\ z_n + z_m & ,t \in \left[\frac{T_m}{4}, \frac{T_m}{2}\right) \\ z_n + 2 \cdot z_m - \frac{2 \cdot z_m}{T_m} \cdot t & ,t \in \left[\frac{T_m}{2}, T_m\right) \end{cases} \tag{31}$$

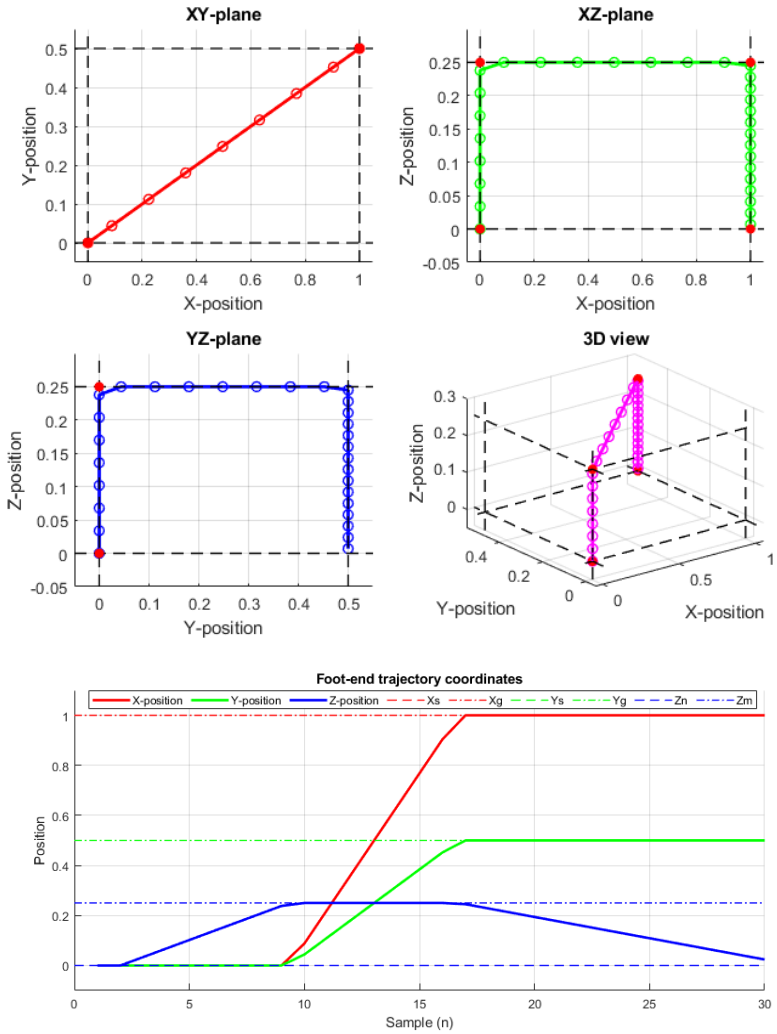
Az $F_x(t)$, $F_y(t)$ és $F_z(t)$ függvények megmondják a lábvég térbeli pozícióját idő függvényében. Az x_s, x_g, y_s, y_g illetve z_n és z_m a lépésfüggvény paraméterei amelyek segítségével pontosan meg tudjuk mondani, hogy hogyan nézzen ki a lépés. Ez a következő ábrán jól látható.



19. ábra. Lépésfüggvény meghatározása akár akadálykerülés céljából, 2-dimenzióra leegyszerűsítve.

Az akadálykerülés csak akkor lehetséges, ha ismerjük, vagy valamilyen módon fel tudjuk térképezni az akadály méreteit és annak függvényében választjuk meg a lépésfüggvény paramétereit. Ideális esetben, egyenletes felületen ezek a paraméterek egy neuronháló segítségével hangolhatók és tanítható a lépésfüggvény, ez persze feltételezi, hogy a lábak nem rendelkeznek tömeggel és az inerciájuk nem befolyásolja a szabályozást, egy valós robottal ellentétben, így gyorsabb sebességeket is el tudunk érni mint a valóságban.

4.10. Lépésfüggvény tervezése



20. ábra. Lábvég pályájának generálása MATLAB környezetben.

A 20. ábrán megfigyelhetünk egy ilyen lépésfüggvényt, azaz a lábvég térbeli (világkoordináták) pozíciójának a változását idő függvényében a (20) - (22) képleteket alkalmazva.

A lépésfüggvény paramétereinek megválasztása a céltól függ. Például ha azt szeretnénk, hogy a robot egyszerűen előre haladjon, akkor az Y pozíció nem változik, konstans nulla. Ezzel ellentétben, ha fordulást szeretnénk elérni, akkor szükséges a lábaknak oldalirányú mozgás is, itt már az Y pozíció nem lehet nulla.

Ahhoz, hogy egy láb képes legyen követni egy ilyen pályát, elengedhetetlen az inverz kinematikai feladat megoldása a pályán található minden egyes mintára. Ezek mellett, szükséges arra is figyeljünk a pályatervezésnél, hogy a robot ne

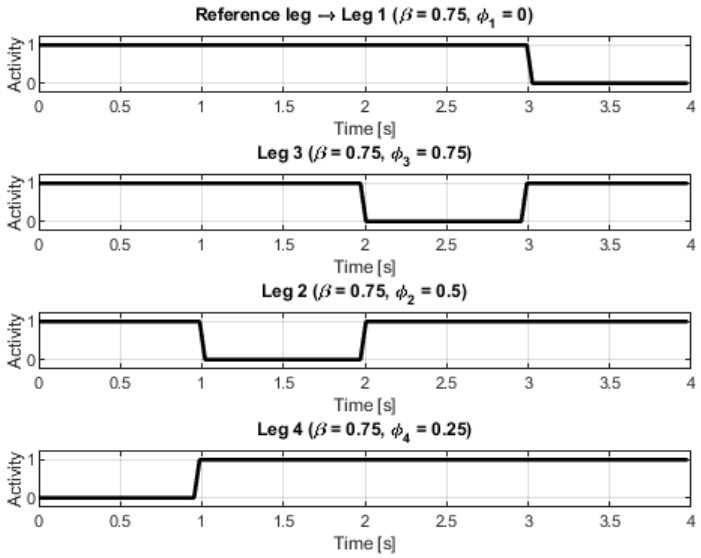
kerüljön szinguláris konfigurációba, ami képes csökkenteni az adott láb szabadságfokát, így egy tengely mentén képtelen a láb elmozdulni. Szerencsére a robot felépítésének köszönhetően, erre a lehetőség nagyon kicsi.

Pályatervezés során szükséges, hogy a robot fizikai határain belül dolgozzunk, azaz ne lépjünk ki egy adott láb munkateréből, valamint kell figyeljünk a robot stabilitására is. Ez a paraméterek kiválasztását megnehezítheti.

4.11. Lépés eseménysorozat tervezése

Egy eseményszekvencia nem más mint a lábak aktivitásának időzítése egy adott referencialábhoz képest, amely általában az 1. láb. Az eseményszekvencia megválasztása szorosan összefügg a robot stabilitásával is, ugyanis minden egyes alkalommal akárhányszor egy, vagy több, láb épp lépésben van instabilitást hoz be a rendszerbe. Úgy kell megtervezzük, hogy ha a robot súlypontjának vetülete a támasztó felületre a támasztó poligonon kívül esik, akkor is legyen képes megtartani saját magát a robot. Ez a szekvencia követi a 4.8 alfejezetben tárgyalt képleteket.

A 21. ábrán látható a lábak aktivitása amely követi a $4-2-3-1$ eseményszekvenciát. Ha egy láb aktivitása 1, akkor azt jelenti, hogy abban a pillanatban a támasztó láb szerepét tölti be, ezzel ellenben ha a láb aktivitása 0, akkor épp lépésben van *(transfer)*.



21. ábra. Eseményszekvencia tervezése.

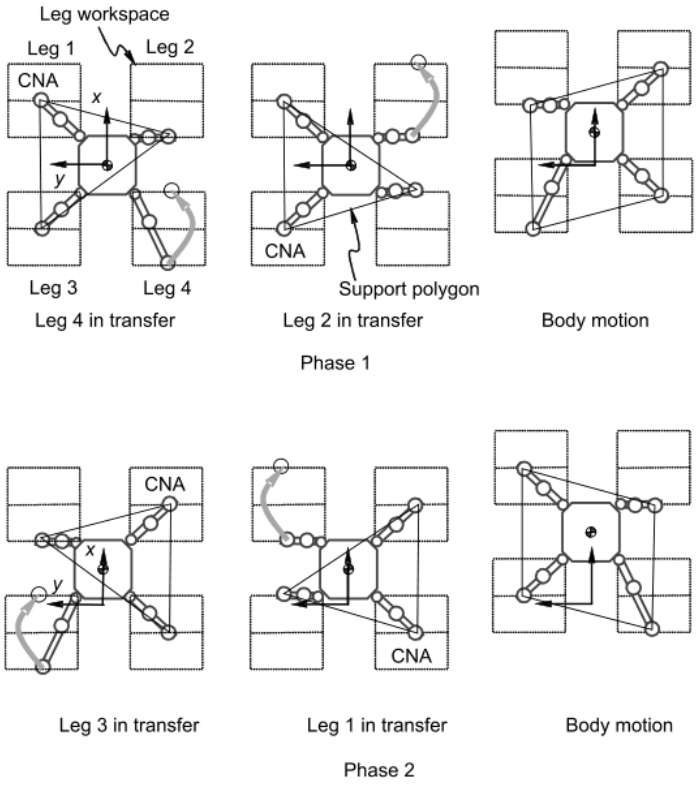
Ahhoz, hogy szaggatott periodikus járást tudjunk generálni, a következő szempontokat kell figyelembe vegyük [7]:

1. Ha az egyik láb a támasztófázisában eléri a munkaterületének hátsó határát (kinematikai határ), ennek a lábnak át kell váltania átviteli fázisba úgy, hogy az elülső kinematikai határába kerüljön;

2. A test az összes láb segítségével kell előre haladjon. Egy testmozgás után legalább egy lábnak a hátsó kinematikai határon kell maradnia a következő lábmozgásba történő átmenethez.

3. A jelenlegi átviteli lábnál kontralaterális és nem szomszédos (*contralateral and non-adjacent, CNA*) láb annyira kell legyen elhelyezve, hogy az átvitt láb elhelyezése után a testsúlypont a CNA láb és az átviteli láb közötti vonal másik oldalán maradjon (lásd 22 ábra). Ezen módon lehetséges lesz egy másik lábat felemelni a gép stabilitásának fenntartása mellett.

4. A lábak sorrendjének periodikusnak kell lennie; ez lehetővé teszi több mozgási ciklus összekapcsolását egy útvonal követése érdekében.



22. ábra. Két fázisú szaggatott járás szekvenciája [7].

5. Implementáció (Hardver)

5.1. Szoftver architektúra

A projekt tervezési fázisához elégséges volt a MATLAB környezet, viszont ha valós hardverrel dolgozunk akkor szükséges egy más környezet mely ezt lehetővé teszi. Erre áll segítségre a Python, amellyel nagyon könnyen tudjuk vezérelni a robotot. Fontos megjegyezni, hogy a fedélzeten található Raspberry Pi Model 4B soros kommunikációt valósít meg egy STM32 mikrovezérlővel, amely felel a motor vezérlésért.

Az osztályok egy-egy funkcionalitást valósítanak meg, például egy osztály felel maga a robot kezeléséért, egy másik osztály a pályatervezésért, egy harmadik osztály a lábakért, stb...

5.1.1. Robot osztály

Ez az osztály felel maga a robot működtetéséért, lábak vezérléséért, adatok begyűjtéséért valamit térbeli orientációjának meghatározásáért. Fontos megjegyezni, hogy már egy létező osztályból van származtatva amely specifikusan a Dogzilla S1 robotra lett tervezve.

```
class Robot(DOGZILLA):
    def __init__(self):
        # Call superconstructor for DOGZILLA
        super().__init__()

        # Body size
        self.L = 0.150
        self.W = 0.045

        # Legs
        self.front_left = Leg(robot=self, leg="front_left")
        self.front_right = Leg(robot=self, leg="front_right")
        self.back_left = Leg(robot=self, leg="back_left")
        self.back_right = Leg(robot=self, leg="back_right")

        # Body attitude (RPY-angles)
        self.roll = 0
        self.pitch = 0
        self.yaw = 0
```

4. kódrészlet. Robot osztály implementációja.

Megfigyelhetjük, hogy a **DOGZILLA** osztályból van származtatva amely a **DOGZILLALib** könyvtárban található. Az osztályváltozók tartalmazzák a robot méretét, a lábakat illetve a *roll*, *pitch* illetve *yaw* szögeket.

A fontosabb örökölt metódusok a következők:
1. `__send(key, index, len)`: Ez a metódus felel az STM32-es mikroprocesszoral való soros kommunikációval megvalósított adatok küldéséért.

```
def __send(self, key, index=1, len=1):
    mode = 0x01
    order = ORDER[key][0] + index - 1
    value = []
    value_sum = 0
    for i in range(0, len):
        value.append(ORDER[key][index + i])
        value_sum = value_sum + ORDER[key][index + i]
    sum_data = ((len + 0x08) + mode + order + value_sum) % 256
    sum_data = 255 - sum_data
    tx = [0x55, 0x00, (len + 0x08), mode, order]
    tx.extend(value)
    tx.extend([sum_data, 0x00, 0xAA])
    self.ser.write(tx)
```

5. kódrészlet. Adatok kiküldése az STM32-es processzornak.

2. `__read(addr, read_len)`: Ez a metódus felel az adatok beolvasásáért soros kommunikációval az STM32-es processzor segítségével.

```
def __read(self, addr, read_len=1):
    mode = 0x02
    sum_data = (0x09 + mode + addr + read_len) % 256
    sum_data = 255 - sum_data
    tx = [0x55, 0x00, 0x09, mode, addr, read_len, sum_data, 0x00,
   0xAA]
    # time.sleep(0.1)
    self.ser.flushInput()
    self.ser.write(tx)
```

6. kódrészlet. Adatok beolvasása az STM32-es processzor segítségével.

4. `motor(motor_id, data)`: Egy adott motor pozíció vezérlését valósítja meg. A motor azonosítók a következők: $11, 12, 13, 21, 22, 23, 31, 32, 33, 41, 42, 43$, ahol az első számjegy maga a láb azonosító és az utolsó számjegy pedig az adott lábon található motorszám a lábvégtől visszafele számolva a bázisig. A `data` paraméter tartalmazza a referencia pozícióját a motornak.

5. `leg(leg_id, data)`: Egy adott láb térbeli pozíció vezérlése a saját koordinátarendszeréhez képest. A láb azonosítók a következők: $1, 2, 3, 4$. A `data` paraméter tartalmazza a referencia koordinátákat.
6. `reset()`: A robotot alaphelyzetbe állítja.
7. `stop()`: A vészhelyzet esetén való leállítást biztosítja, minden irányú mozgást leállít illetve a motrokat alaphelyzetbe állítja.

Vannak műveletek amelyek eltárolják az utolsó aktuális értéket akár későbbi felhasználásra is, ezt az `ORDER` dictionary segítségével valósítsa meg. Minden egyes kulcs megfelel egy adott mért vagy kiszámolt tulajdonságnak, illetve a kulcsokhoz tartozó érték nem más mint egy lista amely tartalmazza a hozzá tartozó művelet címét, valamint az értékeket amelyek a művelet elvégzése után vissza térítődnek. A fontosabb kulcs-érték párok a következők:

```
ORDER = {
    "BATTERY":           [0x01, 100], # Battery percentage
    "FIRMWARE_VERSION": [0x07],
    "UNLOAD_MOTOR":      [0x20, 0],   # Turn off the motors
    "LOAD_MOTOR":        [0x20, 0],   # Turn on the motors
    "VX":                [0x30, 0],   # Velocity on X-axis
    "VY":                [0x31, 0],   # Velocity on Y-axis
    "MOTOR_ANGLE":       [0x50, 128, 128, ...],
    "LEG_POS":           [0x40, 0, 0, ...],
    "IMU":               [0x61, 0],   # Self-stability mode
    "ROLL":              [0x62, 0],
    "PITCH":             [0x63, 0],
    "YAW":               [0x64, 0]
}
```

7. kódrészlet. ORDER dictionary.

Ezek az értékek I^2C (soros) protokoll segítségével kerülnek beolvasásra az STM32 mikroprocesszorból.

Megjegyzés: Az ORDER dictionary-hez hasonlóan, a robot paramétereinek határértékeit egy PARAM nevű dictionary tárolja.

5.1.2. Leg osztály

Ez az osztály felel egy adott láb mozgásának megvalósításáért. Fontos elkülöníteni a lábak vezérlésének kezelését ugyanis fizikailag is négy külön alkatrészről beszélünk, a robot bázis koordináta-rendszeréhez képest más paraméterekkel. Viszont, nem szükséges mind a négy lábra külön-külön elvégezni a pályatervezést, direkt- illetve inverz kinematikai feladatokat, hanem elégséges ha csak egy láb esetében határozzuk meg, ugyanis a referencia lábhoz képest a többi láb szimmetrikus és könnyen tudunk két láb között transzformációt felírni.

Egy adott láb szükséges ismerje a robot objektumot amelyre csatlakozik ugyanis a robot objektumon keresztül vagyunk képesek beolvasni, illetve kiküldeni adatokat az STM32 mikroprocesszornak. Az osztály felépítése a következő:

```
1  class Leg:
2      def __init__(self, robot, leg):
3          self.robot = robot
4
5          # Leg link lengths
6          self.L_1 = 0.0316
7          self.L_2 = 0.0600
8          self.L_3 = 0.0750
9
10         # Angle limits
11         self.t_1_lim = [-PARAM["LEG_LIMIT"][0], PARAM["LEG_LIMIT"
       ][0]]
12         self.t_2_lim = [-PARAM["LEG_LIMIT"][1], PARAM["LEG_LIMIT"
       ][1]]
13         self.t_3_lim = [PARAM["LEG_LIMIT"][2][0], PARAM["LEG_LIMIT
       "][2][1]]
14
15         # Position limits
16         self.x_lim = [-PARAM["TRANSLATION_LIMIT"][0], PARAM["
       TRANSLATION_LIMIT"][0]]
17         self.y_lim = [-PARAM["TRANSLATION_LIMIT"][1], PARAM["
       TRANSLATION_LIMIT"][1]]
18         self.z_lim = [PARAM["TRANSLATION_LIMIT"][2][0], PARAM["
       TRANSLATION_LIMIT"][2][1]]
19
20         # Current leg index
21         self.leg = leg
22
23         self.get_leg_angles()
```

8. kódrészlet. Leg osztály implementációja.

A következő metódusok vannak megvalósítva egy adott láb esetében:

```
def get_leg_angles(self):

    # Read the current motor values.
    # Format: [11, 12, 13, 21, 22, 23, 31, 32, 33, 41,42 ,43],
    # where 11... 43 are motor IDs.

    angles = self.robot.read_motor()

    if self.leg == "front_left":
        self.t_1 = angles[0]
        self.t_2 = angles[1]
        self.t_3 = angles[2]

    elif self.leg == "front_right":
        self.t_1 = angles[3]
        self.t_2 = angles[4]
        self.t_3 = angles[5]

    elif self.leg == "back_right":
        self.t_1 = angles[6]
        self.t_2 = angles[7]
        self.t_3 = angles[8]

    elif self.leg == "back_left":
        self.t_1 = angles[9]
        self.t_2 = angles[10]
        self.t_3 = angles[11]
```

9. kódrészlet. Motrok szögeinek frissítése

5.1.3. PathPlanner osztály

Ez az osztály felel egy adott láb pályatervezéséért illetve láb eseményszekvencia meghatározásáért adott paraméterek alapján. Szükséges ismerjük a robot objektumot amelyre ezt el akarjuk végezni, ugyanis az tárolja a robot paramétereit illetve lábait. Itt mondjuk meg a mintavételezési periódust, a lépés kezdeti idejét, a lépés végének idejét, a támasztó illetve transzfer faktort amelyből meg tudjuk határozni az aktív ciklusidőt.

A PathPlanner osztály felépítése a következő.

```
class PathPlanner:
    def __init__(self, robot):
        self.robot = robot

        # Path planning parameters
        self.sample_period = 0.034
        self.start_time = 0
        self.end_time = 4
        self.support_factor = 3
        self.transfer_factor = 4
        self.duty_factor = self.support_factor / self.
    transfer_factor

        # Event timings
        self.event_timings = {
            "front_left": np.array([]),
            "front_right": np.array([]),
            "back_left": np.array([]),
            "back_right": np.array([])
        }

        # Event phases
        self.event_phase = {
            "front_left": 0,
            "front_right": 0.5,
            "back_left": self.duty_factor,
            "back_right": fractional_function(self.duty_factor -
    0.5)
        }

        # Starting position
        self.robot.update_legs()
        self.leg_start = {
            "front_left": [...],
            "front_right": [...],
            "back_left": [...],
            "back_right": [...]
        }

        # Ending position
        self.Xg = 0.1
        self.Yg = 0.0
        self.Zm = -0.015

        self.leg_end = {
```

```
            "front_left": [...],  # Plus Xg, Yg and Zm
            "front_right": [...], # Plus Xg, Yg and Zm
            "back_left": [...],   # Plus Xg, Yg and Zm
            "back_right": [...]   # Plus Xg, Yg and Zm
        }

        # Trajectory
        self.leg_trajectory = {
            "front_left": [],
            "front_right": [],
            "back_left": [],
            "back_right": []
        }
```

10. kódrészlet. PathPlanner osztály implementációja.

Kétféle lehetőség van meghatározni a láb kiinduló pozícióját. Egyik megoldás az, hogy közvetlenül a robot indítása után beolvassuk egy adott láb csuklószögeit és megoldjuk a direkt geometriai feladatot. A második megoldás egyszerűbb és hatékonyabb, amely nem más mint a robot indítása után a `leg()` metódus segítségével nullpozícióba, vagy egy általunk meghatározott és ismert referencia pozícióba elhelyezzük és az így kapott pozíció lesz a kiindulópont.

Ha ez sikeresen megtörtént, akkor el tudjuk végezni a pályatervezést amely egy lépésnek a térbeli pontjait határozza meg. Egy három fázisú lépésfüggvényről (pályáról) beszélünk, szóval három darab jól elkülöníthető kódrészletre kell osztani az algoritmust. Fázisonként meghatározzuk a lábvég x, y illetve z koordinátáit a (20) - (22) képletek alapján. Ez után vagy parancsot küldünk a robotnak, hogy az adott lábon található egyenáramú motrokat forgassa el a kiszámolt pozícióba, vagy pedig eltároljuk a kapott pozíciót.

Az algoritmus pszeudo-kódja a következő. (Nincs konkrét kódrészlet ugyanis eléggé összetett.)

```
function GaitFunction:

    Read gait function parameters:
        Starting position: Xs, Ys, Zn
        End position:      Xg, Yg, Zm
        Step length:       Tm
        Sample time:       Ts

    start_time = getSimulationTime()
    current_time = getSimulationTime()
    elapsed_time = current_time - start_time

    while elapsed_time <= Tm

        if elapsed_time < Tm / 4
            Calculate: X, Y, Z
        else if elapsed time < Tm / 2
            Calculate: X, Y, Z
        else if elapsed_time < Tm
            Calculate: X, Y, Z
        end

        current_time = getSimulationTime()
        elapsed_time = current_time - start_time
    end

    Store X, Y and Z coordinates.
end
```

11. kódrészlet. Egy láb esetében a pályatervezés algoritmusa.

A pályatervezést megvalósító algoritmus folyamatábrája megtalálható a függelékek között (26. ábra).

Ha a pályatervezés sikeresen megtörtént, akkor a következő lépés a láb eseményszekvencia meghatározása. Ez a (15)-(19) képletek segítségével történik, ehhez azonban egy segéd függvényre van szükségünk amely megvalósítja a (19)-es képletet, azaz a törtfüggvényt. Ez a következőképpen lett implementálva.

```
import numpy as np

def fractional_function(x):
    frac = np.mod(np.abs(x), 1)

    if x >= 0:
        return frac
    else:
        return 1 - frac
```

12. kódrészlet. Törtfüggvény implementációja.

Megjegyzés: Ez a függvény egy különálló függvény a PathPlanner osztálytól.

Az eseményszekvencia kiszámolásának az elve a következő: Meghatározzuk egy láb ciklusát úgy, hogy csak azokra az értékekre nulla amikor a láb transzfer szerepet tölt be és a többi érték 1, ez lesz a referencia lábhoz tartozó ciklus. Ha ez megtörtént, akkor körkörösen el tudjuk tolni az adott láb fázisával balra, időben beszélve. Ha ezt mintavétel tartományban akarjuk megoldani akkor a láb fázisát szükséges megszorozni a mintavételek számával. Ez a következőképpen van implementálva.

```
def plan_event_timings(self):

    for t in self.time:
        self.event_timings["front_left"] = np.append(
            self.event_timings["front_left"],
            (t + self.event_phase["front_left"] * self.end_time) <
    (self.duty_factor * self.end_time))

    self.event_timings["front_right"] = np.roll(self.event_timings
    ["front_left"], int(self.event_phase["front_right"] * len(self.
    event_timings["front_left"])))

    self.event_timings["back_left"] = np.roll(self.event_timings["
    front_left"], int(self.event_phase["back_left"] * len(self.
    event_timings["front_left"])))

    self.event_timings["back_right"] = np.roll(self.event_timings[
    "front_left"], int(self.event_phase["back_right"] * len(self.
    event_timings["front_left"])))
```

13. kódrészlet. Lábszekvencia generálása.

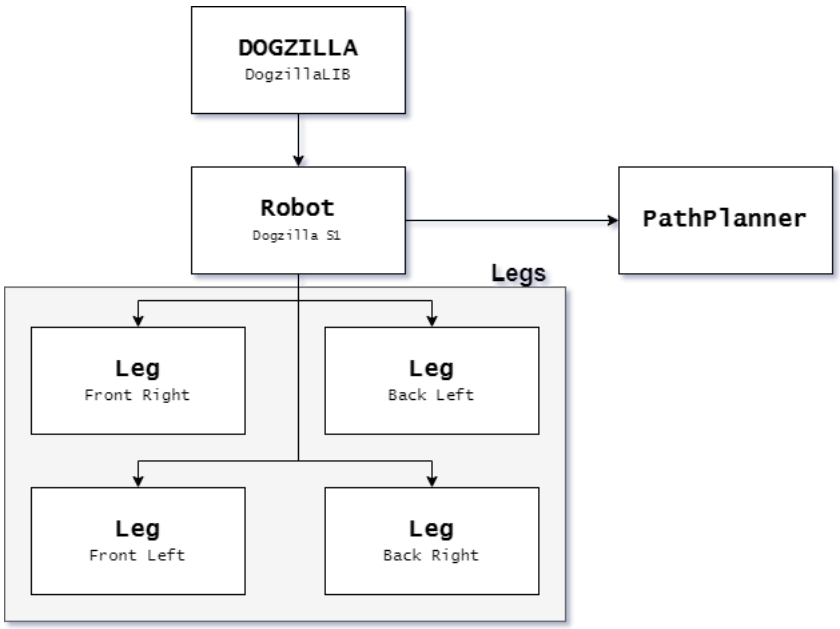
5.1.4. Szoftver UML diagramja

Az UML (Unified Modeling Language) egy szabványosított grafikus nyelv, amelyet szoftvertervezési és dokumentációs célokra használnak. Az UML diagramok segítségével lehetőség van a rendszerek, alkalmazások és folyamatok struktúrájának és viselkedésének modellezésére és kommunikálására.

Az UML diagramok különböző típusai különböző aspektusokat fednek le a tervezés során, például az osztályok struktúráját (osztálydiagram), a folyamatokat (állapotgép diagram), a viselkedést (szekvenciadiagram), az adatáramlást (tevékenységdiagram) stb.

Az UML diagramok gyakran használatosak a szoftvertervezési folyamatok során, hogy segítsenek a tervezőknek és fejlesztőknek megérteni és kommunikálni a tervezett rendszer vagy alkalmazás részleteit [9].

A megvalósított szoftver UML diagramja a következő:



23. ábra. A robot szoftver UML diagramja (hardveres implementáció)

5.2. Lépés összegezve

Mivel az osztályok tárgyalásra kerültek, a következő lépés egy olyan algoritmus felépítése amely ténylegesen elvégez egy lépést, követi a lábeseményszekvenciát valamint valós időben (amennyire megengedi a hardver) beolvassa a csuklópozíciókat, meghatározza a lábvég pozícióját valamint kiküldi a beavatkozó jelet a pályán található következő pont követésére.

1. Az első lépés a pályatervezés. A pályatervezést nem szükséges minden egyes ciklusban elvégezni, hanem elég ha csupán egyszer, az inicializálás során elvégezzük, utána pedig elmentjük a kiszámolt pályát a memóriába, vagy egy fájlba, ez esetben egy `.csv` állományba.

2. Lábeseményszekvencia generálása a megválasztott paraméterek alapján. Fontos, hogy úgy válasszuk meg, hogy stabil állapotba tudja helyezni magát a robot még ha rövid ideig instabilitás is lép fel.

3. Ezután indíthatunk egy ciklust, amely addig fut amíg el nem végzünk egy lépést egy láb esetében. Itt történik a lábvég pozíciójának meghatározása illetve a beavatkozó jel kibocsájtása. Azt, hogy jelenleg melyik lábat mozgassuk azt az eseményszekvencia határozza meg.

4. A 2-es pontban említett ciklust még egy nagyobb ciklusban el tudjuk helyezni, amely ismétli a lépésszekvenciát addig amig a mért idő el nem éri a meghatározott időt, vagy valamilyen jelzést nem kap, hogy álljon meg a robot.

A következő pszeudokód leírja az előbb említett algoritmust.

```
Initialization of the robot.

    robot = Robot()
    robot.initialize()

Initialization of the path planner.

    paht_planner = PathPlanner(robot)

Plan the trajectory and event timings.

    path = paht_planner.plan_trajectory()
    event_timings = paht_planner.plan_event_timings()

Start timer.

    current_time = Get current time
    end_time = current_time + path_planner.end_time

Main loop.

    step = 0

    while current_time <= end_time

        Read current motor position.

            motors = roobt.read_motors()

        Get next trajectory point.

            next = path[step]

        Move the corresponding leg to the next trajectory point.

            if path_planner.event_timings["front_left"] == 0
                Move first leg to next position.

            else if path_planner.event_timings["front_right"] == 0
                Move second leg to next position.
```

```
41              else if path_planner.event_timings["back_left"] == 0
42                  Move third leg to next position.
43
44              else if path_planner.event_timings["back_right"] == 0
45                  Move fourth leg to next position.
46
47              end
48
49          Update timer.
50
51              current_time = Get current time
52
53          Update step.
54
55              step = step + 1
56
57      end
```

14. kódrészlet. Lépés algoritmusa magasabb rendű függvényhívásokkal

Fontos megjegyezni, hogy magas szintű tervezés szempontjából van leírva a megvalósított szoftver pszeudokódja, azaz egy magasabb absztrakciós szintről beszélünk. Szükséges, hogy megírjuk a benne szereplő függvényeket valamint algoritmusokat, ilyen például egy láb mozgását megvalósító algoritmus.

Könnyebb átláthatóság érdekében vegyük figyelembe egy lépés algoritmusának a folyamatábráit. Ez megtalálható a függelékek között (27. ábra valamint 28 ábra).

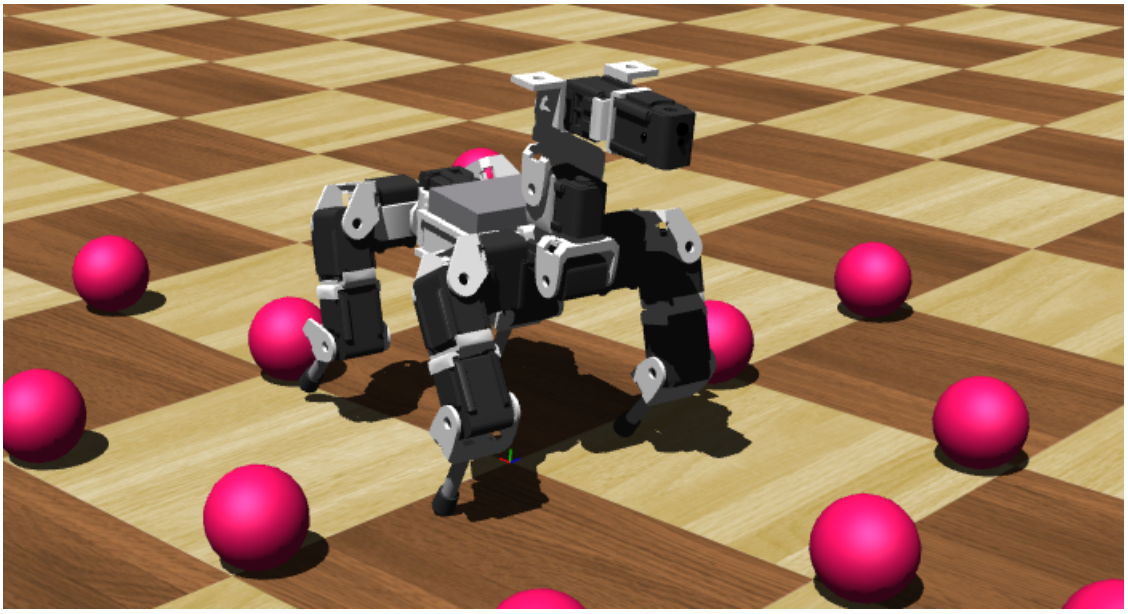
6. Implementáció (Szimuláció)

Már láthattuk, hogy hardver szinten hogyan történik az implementáció. Szoftveres szimuláció esetében nem nagy az eltérés a hardveres implementációtól, ugyanis ugyan azok az algoritmusok illetve tervezési módszerek vannak felhasználva, helyenként némi különbségekkel.

A környezet amelyben a szoftveres szimuláció megtörténik az a **Cyberbotics** által fejlesztett **Webots** szimulációs környezet. A Webots egy nyílt forráskódú és többplatformos asztali alkalmazás, amelyet robotok szimulálására használnak. Teljes fejlesztői környezetet biztosít a robotok modellezéséhez, programozásához és szimulálásához. Professzionális használatra tervezték, és széles körben használják az iparban, az oktatásban és a kutatásban [5].

Lehetőségünk van egy már létező robottal dolgozni, vagy pedig modellezni (felépíteni) is tudunk egy saját robot modellt ha úgy tartja a szükséglet. Egy ilyen robot modell komplexitásából kiindulva és a szükséges belefektetett időből amely egy modell elkészítését igényli, egy már létező robotmodell lesz felhasználva.

A felhasznált robotmodell nem más mint a **Robotis** által tervezett **Bioloid Dog** robotmodell lesz [13].



24. ábra. Bioloid Dog robotmodell Webots megvalósítása [13]

6.1. Különbségek a valós hardverrel szemben

Fontos megemlíteni a különbségeket a valós Dogzilla S1 robot hardvere illetve a Webots környezetben felépített Bioloid Dog modell között:

1. Megfigyelhetjük, hogy a Bioloid Dog modell rendelkezik egy fej résszel is amely hozzáad még 3 szabadságfokot a rendszerhez (a következő csuklók: `head`, `neck_1`, `neck_2`).

2. A Bioloid Dog modell még rendelkezik egy csuklóval amely egy olyan tengely elforgatását biztosítja, amelyen található a robot két hátsó lába (`back_left` valamint `back_right`). A csukló forgástengelye a robot testéhez képest kifele mutató irányban van. Ez növeli a teljes rendszer szabadságfokát 1-el.

3. Egy lábat nézve a válltól számított csuklótól, az első két csukló fel van cserélve a kinematikai láncban, azaz először történik forgatás a csuklótengelyre merőlegesen, utána a vele párhuzamos tengely körül.

Ezeket a különbségeket figyelembe véve, a Bioloid Dog modellnek összesen 16 szabadságfoka van (azaz 12 darab csukló), de ebből csak 12 kerül felhasználásra azért, hogy tudjuk úgy kezelni mint a Dogzilla S1 robotot.

6.2. Bioloid Dog megvalósítása Webots környezetben

A Webots szimulációs környezetben minden egyes `Robot` objektumhoz szükséges egy robotvezérlő (*robot controller*) fájlt/szkriptet csatolni. Ez esetben ennek a szkriptnek a megírása Python nyelvben történik. Általában minden beépített robotmodell mellé érkezik egy robotvezérlő kód is, ez esetben is ez igaz [15]. Az eredeti robotvezérlő kódja C++ nyelvben van írva, amelyet átírtam Python nyelvre a projekt egyszerűségét megtartva. Egy másik indok a Python nyelvre való áttérésre, hogy a későbbiekben említett továbbfejlesztési lehetőségek megvalósulhassanak.

A szimulációban megvalósított algoritmusok a már tárgyalt elveken alapszanak. A pályatervezés esetében a különbség az, hogy a négyszög alapú pálya helyett egy ellipszis-görbét követ a láb vége, azonban ez könnyen módosítható a már tárgyalt eljárások implementálásával.

Egy Webots robotmodellhez képesek vagyunk különböző eszközöket csatolni, akár csuklókat, kamerákat, tengelyeket, koordináta-transzformációkat, GPS illetve IMU szenzorokat, merev testeket, geometriát valamint a robot szimulációjához szükséges fizikai állandókat/törvényeket is meg tudjuk határozni.

A Bioloid Dog robotkontrollere a következő:

```
from controller import Robot
import numpy as np

class RobotController(Robot):

    # Important parameters
    MAX_MOTORS = 16
    SIMULATION_STEP_DURATION = 16

    # Link lengths
    L1 = 9
    L2 = 8.5

    # Motors (joints)
    MOTOR_NAMES = [...]
    PELVIS = 0
    FRONT_LEFT_1 = 1
    FRONT_RIGHT_1 = 2
    # etc...

    # Gait types
    GAIT_NAME = ["trot", "walk", ...]
    GAIT_PHASE_SHIFT = [...]

    def __init__(
        self,
        _controlStep=SIMULATION_STEP_DURATION,
        _stepCount=0
    ):
        self.controlStep = _controlStep
        self.stepCount = _stepCount
        self.motors = []
        self.position_sensors = []

        # Initialization of the attached devices
        for motor in self.MOTOR_NAMES:
            self.motors.append(self.getDevice(motor))
            self.position_sensors.append(self.motors[-1].
   getPositionSensor())
            self.position_sensors[-1].enable(_controlStep)

        self._gps = self.getDevice("gps")
        self._gps.enable(_controlStep)
```

15. kódrészlet. Bioloid Dog robotvezérlő osztálya

A motrok/csuklók helyzetének beállítása/olvasása

```
def set_motor_position(self, motor, position):
    self.motors[motor].setPosition(position)

def get_motor_position(self, motor):
    return self.position_sensors[motor].getValue()
```

16. kódrészlet. Motor pozíciójának beállítása/olvasása

A motrokhoz/csuklókhoz tartozó pozíció szenzor aktiválása

```
def get_motor_position(self, motor):
    return self.position_sensors[motor].getValue()
```

17. kódrészlet. Pozíció szenzor aktiválása

Egy bizonyos ideig való várakozás

```
def wait(self, x):
    num = x / (self.controlStep / 1000)

    for i in range(int(num)):
        self.step(self.controlStep)
```

18. kódrészlet. Várakozás bizonyos ideig

A robot alaphelyzetbe való állíása

```
def stand(self):
    self.set_motor_position(self.NECK_1, -np.pi / 2)
    self.set_motor_position(self.FRONT_LEFT_2, np.pi / 2)
    self.set_motor_position(self.FRONT_RIGHT_2, -np.pi / 2)
    self.set_motor_position(self.BACK_LEFT_2, np.pi / 2)
    self.set_motor_position(self.BACK_RIGHT_2, -np.pi / 2)
    self.wait(1)
```

19. kódrészlet. Robot négy lábra való állítása

Inverz kinematikai feladat megoldása (csak függvény deklaráció)

```
def compute_walking_position(
    self,
    t,                      # Length of the gait in seconds
    gait_frequency,         # Speed of the gait
    gait_type,              # Which gait type
    leg_id,                 # Which leg
    stride_length_factor,   # Duty cycle
    backward                # Backwards motion
)
```

20. kódrészlet. Inverz kinematika megoldása Bioloid Dog modellhez

7. Összegzés

A dolgozatom egy általános bevezető egy quadruped mobilis robot működésébe. Kezdetben a matematikai alapok kerülnek magyarázatra (egy pont leírása térben, rotáció, transzláció, koordináta transzformációs mátrixok, Denavit-Hartenberg standardok). A matematikai alapok szükségesek a robot direkt- illetve inverz kinematikájának illetve dinamikájának felírásához. Ha ismert a robot direkt- illetve inverz kinematikája, akkor vizsgálni tudjuk egy adott láb munkaterét, szinguláris konfigurációkat illetve a különböző járásmódokat.

Különböző járásmódokat figyelembe véve, vizsgálatra kerül két féle stabilitási margó vizsgálata (folytonos illetve szaggatott), valamint az aktív ciklusidő és a lépés magassága függvényében a járás sebessége valamint periódusa.

Ezek mellett tanulmányozva van egy láb esetében a pályatervezés különböző paraméterek alapján, amely szimmetrikusan alkalmazható mind a négy lábra, és a négy lábnak az ütemezése (eseményszekvencia) generálása, amely szoros kapcsolatban van a járásmóddal.

Két féle implementáció van megvalósítva. Az egyik implementáció hardver szintjén történik, azaz a valós roboton, amely nem más mint a **Dogzilal S1** nevű robot. Tárgyalva van a robot szerkezete, operációs rendszere, hogyan csatlakozunk rá, programozási eljárások valamint az algoritmusok pszeudokódja. A másik implementáció szimuláció szinten történik **Webots** környezetben ahol a **Bioloid Dog** robotmodell van felhasználva. A szimuláció esetében néhány lényeges különbség lép fel a valós robot illetve a szimulált robot között.

A quadruped robotok tervezése és fejlesztése egyre nagyobb jelentőséggel bír a jövőben az automatizálás és a robotika terén. A dolgozatom által elért eredmények és megközelítések hozzájárulhatnak ezeknek a rendszereknek a hatékonyságának és funkcionalitásának növeléséhez, elősegítve ezzel a quadruped robotok széles körű alkalmazását és elfogadását a mindennapi életben.

8. Továbbfejlesztési lehetőségek

A dolgozat jelenlegi állapota elégséges a tudományos következtetések levonására, de még akadnak továbbfejlesztési lehetőségek is amelyek a dolgozat tudományos jellegét bővítheti.

8.1. Megerősíéses tanulás a járás optimalizálására

Az egyik ilyen továbbfejlesztési lehetőség a **mesterséges inelligenciával** való ötvözése a projektnek. A mesterséges intelligencián belül a **gépi tanulás** egy jó lehetőséget ad **megerősítéses tanulás** módszereket felhasználva a pályagenerálás optimalizálása. A megerősítéses tanulás igényel egy **költségfüggvényt** amely egy epoch vagy tanítási ciklus végén kiértékelve megmondja, hogy az adott generáció mennyire fejlődik. Ha a költségfüggvény változása konvergál a számunkra elvárt értékhez, általában a nullához, akkor az **egyedek (ágensek)** egyre jobban fejlődnek. Egy lehetséges költségfüggvény, amely négyzetes hibát vesz figyelembe, felépítése a következő:

$$\mathcal{F}_{cost} = \delta^2_{velocity} + \delta^2_{stability} \tag{32}$$

Ahol:

$$\delta_{velocity} = V_{ref} - V_{average} \tag{33}$$

$$\delta_{stability} = S_{ref} - S_{average} \tag{34}$$

A V_{ref} egy referencia sebesség, az S_{ref} pedig egy általunk választott (folytonos vagy szaggatott esetben) referencia stabilitási margó értékét határozza meg (lásd 13. ábra). Az átlagolt értékek ($V_{average}$ illetve $S_{average}$) egy adott tanítási ciklusban, vagy epoch-ban, megmondja a mért sebesség illetve stabilitási margók átlagát. Több módszer létezik a költségfüggvény minimalizálására, ilyen például a **Q-Learning** vagy a **PPO (Proximal Policy Optimization)**. Ezek mind iteratív eljárások.

A paraméterek hangolása a pályatervezés során történik. Négyzetes pálya esetén a **térbeli határok** kerülnek hangolásra (Xs, Xg, Ys, Yg, Zn illetve Zm), valamint az esemény szekvencia esetében az **aktív ciklusidő** (β) kerül hangolásra. Ez a két paraméter együttese befolyásolja a robot egy irány menti sebességét valamint stabilitását.

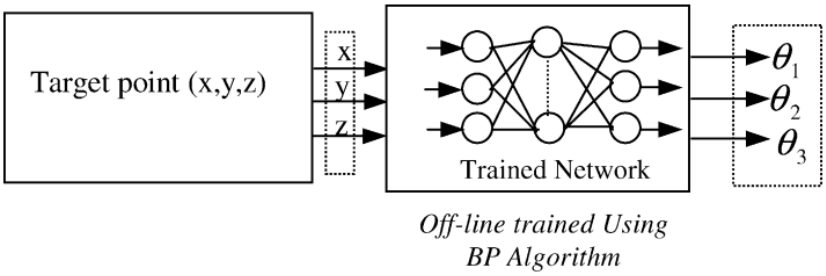
A megerősítéses tanulás megvalósítása Python környezetben történne a **Tensorflow** könyvtárat felhasználva amely egy nyílt forráskódú könyvtár a mesterséges intelligencia illetve gépi tanulásra kifejlesztve. Segítségre szolgál még az **OpenAI Gym** környezet is.

8.2. Inverz kinematika megközelítése neuronhálóval

Tudjuk, hogy az inverz kinematika megmondja egy referencia végpont (pozíció és orientáció) szerint a csuklópozíciókat. A dolgozatban az inverz kinematikai feladat megoldása numerikus módszerekkel történik amelyet minden egyes alkalommal megkell oldani ha a referencia pozíció változik, ez a választott optimumkereső algoritmustól függően nagyon számítás igényes lehet.

A probléma megoldására több fajta lehetőség is létezik, az egyik ilyen megoldás az inverz kinematika becslése/megközelítése neurális hálók segítségével. Egy egyszerű tanító algoritmus a hiba visszaterjesztés *(back propagation)* módszere [8] [6].

A quadruped robotok esetében kétféle képpen tudjuk megoldani a problémát. Az egyik megoldás egy neuronháló tervezése minden lábra és annak párhuzamosan való futattása, vagy pedig kibővítjük a 25. ábrán található neurális hálót úgy, hogy 4×3 bemenete legyen (lábanként 3 X, Y illetve Z koordináták) és 12 darab kimenete amely a csuklópozíciók lábanként.



25. ábra. Neurális hálón alapuló inverz kinematika egyszerűsített ábrája

Ábrák jegyzéke

Táblázatok jegyzéke

Kódrészletek jegyzéke

Hivatkozások

[1] Saurav Agarwal, Abhijit Mahapatra, and Shibendu Shekhar Roy. Dynamics and optimal feet force distributions of a realistic four-legged robot. *IAES International Journal of Robotics and Automation*, 1(4):223, 2012.

[2] Michel Aractingi, Pierre-Alexandre Léziart, Thomas Flayols, Julien Perez, Tomi Silander, and Philippe Souères. Controlling the solo12 quadruped robot with deep reinforcement learning. *scientific Reports*, 13(1):11945, 2023.

[3] Priyaranjan Biswal and Prases K. Mohanty. Development of quadruped walking robots: A review. *Ain Shams Engineering Journal*, 12(2):2017–2031, 2021.

[4] Codespring. `https://www.codespring.ro/`.

[5] Cyberbotics. `https://cyberbotics.com/`.

[6] Bassam Daya, Shadi Khawandi, Mohamed Akoum, et al. Applying neural network architecture for inverse kinematics problem in robotics. *Journal of Software Engineering and Applications*, 3(03):230, 2010.

[7] Pablo Gonzalez De Santos, Elena Garcia, and Joaquin Estremera. *Quadrupedal locomotion: an introduction to the control of four-legged robots*, volume 1. Springer, 2006.

[8] Raşit Köker, Cemil Öz, Tarık Çakar, and Hüseyin Ekiz. A study of neural network based inverse kinematics solution for a three-joint robot. *Robotics and autonomous systems*, 49(3-4):227–234, 2004.

[9] Miro. `https://miro.com/diagramming/what-is-a-uml-diagram/`.

[10] Fehér Áron Márton Lőrinc. Robotirányítások laboratóriumu Útmutató, 2023.

[11] Dilip Kumar Pratihar, Kalyanmoy Deb, and Amitabha Ghosh. Optimal path and gait generations simultaneously of a six-legged robot using a ga-fuzzy approach. *Robotics and Autonomous Systems*, 41(1):1–20, 2002.

[12] Carlos Queiroz, Nuno Goncalves, and Paulo Menezes. A study on static gaits for a four leg robot. In *Proc. Control-UK ACC Int. Conf. Control*, pages 1–6, 1999.

[13] Robotis. `https://www.cyberbotics.com/doc/guide/bioloid?version=cyberbotics:R2019a`.

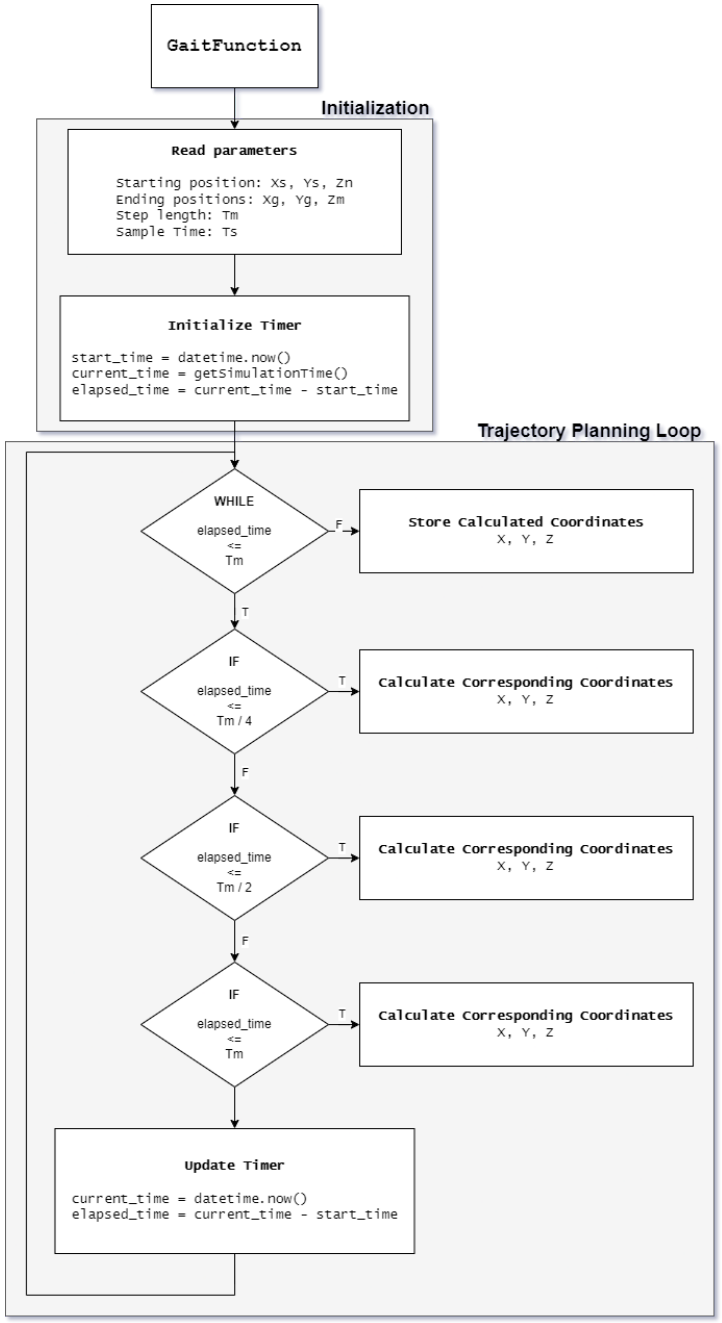
[14] Mahmoud Tarokh and Malrey Lee. Systematic method for kinematics modeling of legged robots on uneven terrain. *International Journal of Control and Automation*, 2(2):9–18, 2009.

[15] Webots. `https://github.com/cyberbotics/webots/tree/master/projects/robots/robotis/bioloid`.

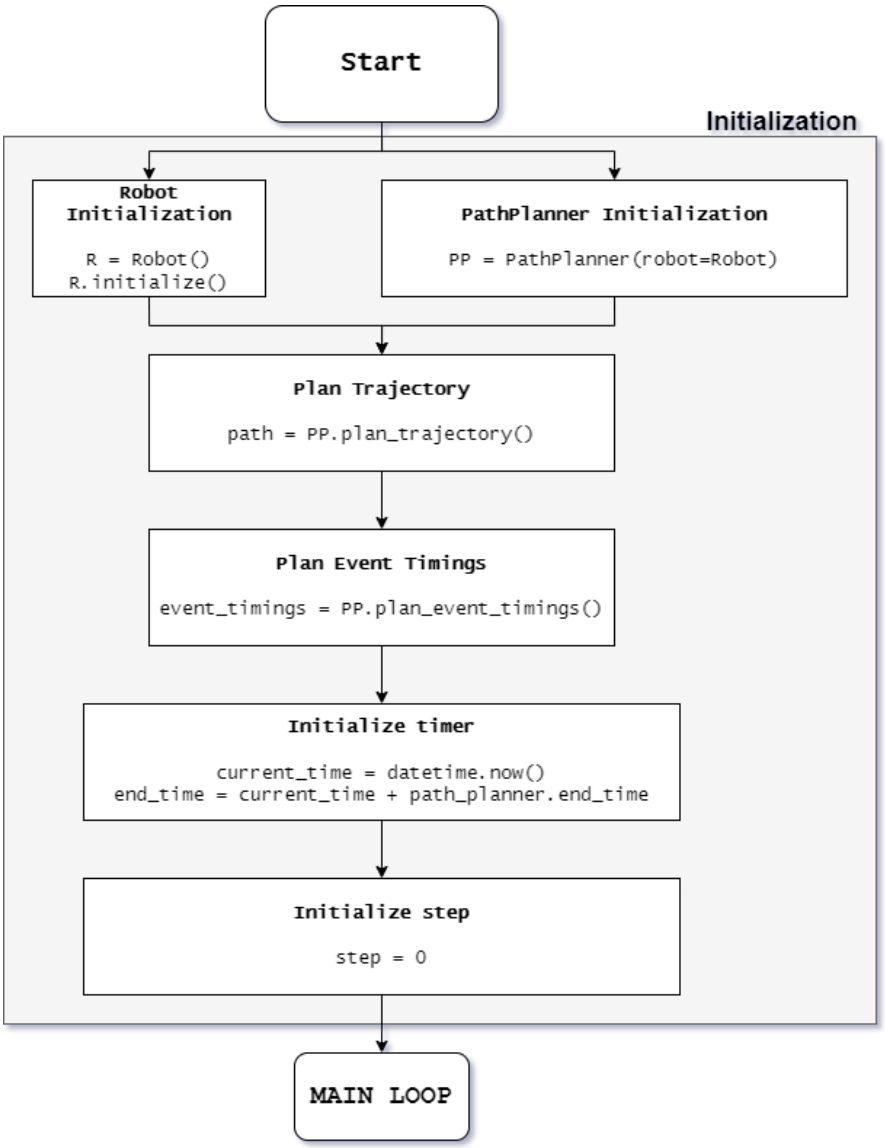
[16] YahboomTechnology. `https://github.com/YahboomTechnology/DOGZILLA-S1`.

[17] YahboomTechnology. `http://www.yahboom.net/study/DOGZILLA-S1`.

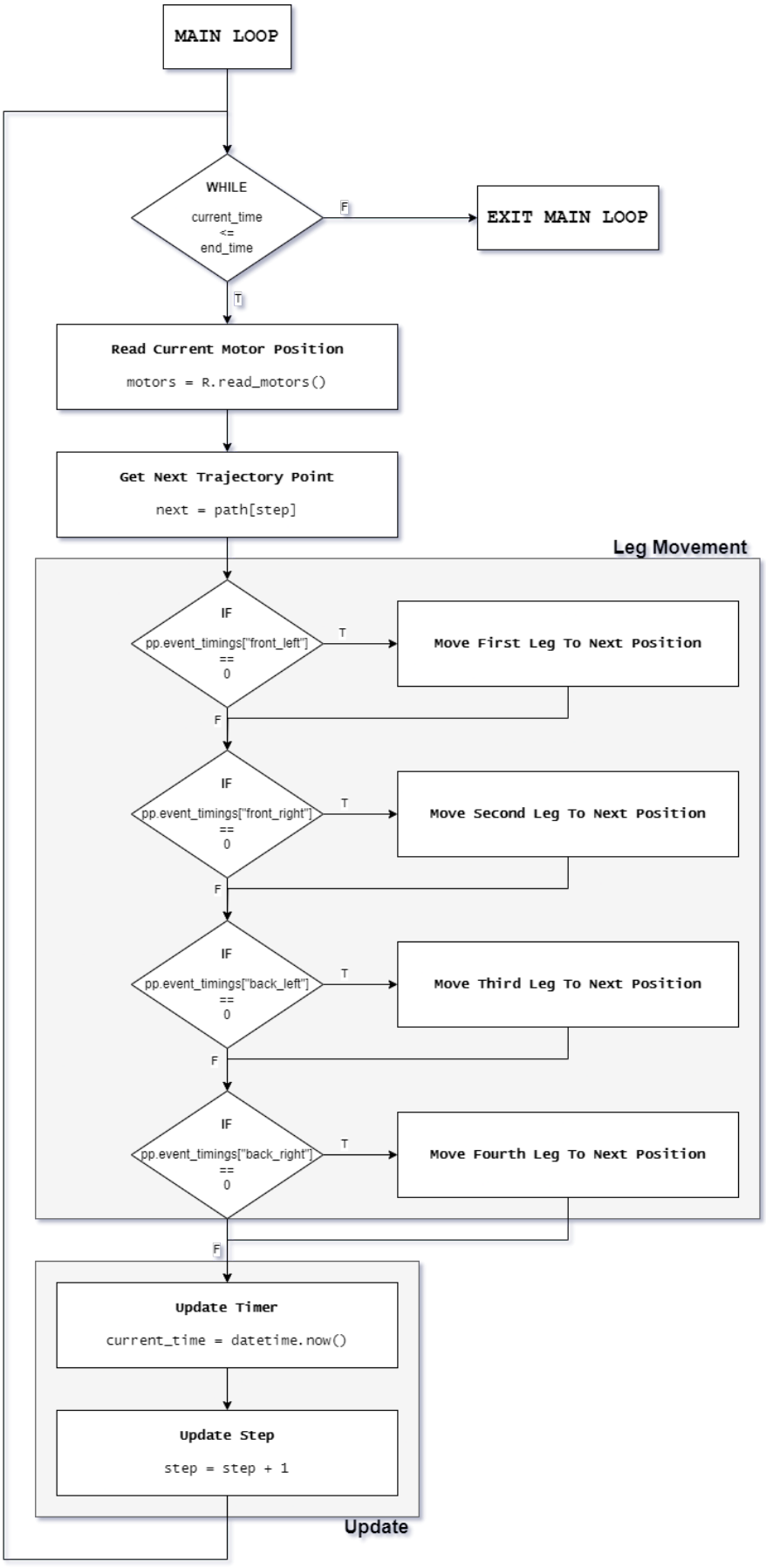
Függelékek



26. ábra. Pályatervező algoritmus folyamatábrája



27. ábra. Egy lépés folyamatábrája



28. ábra. A fő ciklus folyamatábrája