Budapesti Műszaki és Gazdaságtudományi Egyetem
Villamosmérnöki és Informatikai Kar
Irányítástechnika és Informatika Tanszék

Pálya- és trajektóriatervezési algoritmus robotkarokhoz

TDK dolgozat

Készítette:

Szabó Dániel

Konzulens:

Gincsainé Szádeczky-Kardoss Emese

2019

Tartalomjegyzék

1. fejezet

Bevezetés

Napjainkban az automatizálás egyre nagyobb szerepet tölt be az életünkben. Az ipari automatizálásnak már eddig is fontos részét képezték az ipari robotkarok, melyekről általánosságban elmondható, hogy bizonyos feladatokat gyorsabban, pontosabban el tudnak végezni, mint az emberi operátorok.

A jelenlegi alkalmazásokra jellemző, hogy a robotok egy statikus vagy előre ismert módon változó térben végzik a munkájukat. Pontos információkkal rendelkeznek a környezetükben lévő tárgyakról, így a robotokat programozó mérnökök ütközésmentes útvonalat tudnak tervezni az manipulátor egyes munkaállomásai között. Tehát a legtöbb mai alkalmazásban nem csupán a robotkar végállomásait kell definiálnunk, hanem a teljes útvonal kézi megtervezésére is szükség van.

Ezen tervezési szakasz gyorsítása végett egyre inkább szükség van olyan megoldásokra, melyek önállóan képesek a robotkarok pálya- és trajektóriatervezésére az állomások között úgy, hogy akár az emberi jelenlétet vagy előre nem ismert mozgású dinamikus akadályokat is figyelembe veszik.

Pályatervezésnek nevezzük azt a feladatot, mely során egy kezdeti állapotból kiindulva olyan konfiguráció-sorozatot találunk, melyen végighaladva ütközésmentesen jutunk el egy kívánt célállapotba. A trajektóriatervezés során ezen konfiguráció-sorozatból kívánunk megvalósítható időfüggvényeket generálni.

Alapvetően a környezet időbeli változása alapján kétfajta pálya- és trajektóriatervezési feladatot különböztetünk meg: offline és online feladatot. Abban az esetben, ha a robot környezete állandó vagy előre ismert módon változik, a robotkar által leírandó trajektóriát már előre meghatározhatjuk, ezt nevezzük offline tervezési feladatnak. Amennyiben a térben dinamikusan, előre nem ismert módon mozgó akadályok vannak, akkor ebben a térben valós időben kell meghatároznunk a robotkar útját, ez nevezzük online mozgástervezési feladatnak. A feladatom során újszerű, hatékony megoldást kerestem mind a statikus, mind a dinamikus térben történő pálya- és trajektóriatervezésre.

Az offline pályatervezés során több módszert is számításba vettem, mint például a hagyományos grid-alapú módszereket (A* és D* algoritmusok), melyek képesek a mintavételezési háló felbontásához képesti optimális megoldást találni [32]. A hátránya ezen algoritmusoknak, hogy a keresési tér növekedésével exponenciálisan nő a számítási-, illetve a memóriaigény (ezt a hatást hívjuk a ”Dimenzió átkának”), így nem alkalmazható jó hatásfokkal nagy dimenziójú terek esetén.

Szintén megoldás jelenthetnek a mesterséges potenciáltereken alapú módszerek [15], mellyel egyszerűen és gyorsan meg tudjuk határozni a robot mozgását. Ezen módszerek hátránya, hogy egy-egy lokális optimum megakadályozhatja a sikeres pályatervezést, ezért ezt a módszert csak az online mozgástervezés során használtam, ahol a váratlan eseményekre történő gyors reakció prioritást élvez.

Napjainkban előszeretettel alkalmazottak a Rapidly-exploring Random Tree (RRT) családjába tartozó algoritmusok. Ezek inkrementális, véletlenszerű mintavételezésen alapuló algoritmusok, melyek nagy hatásfokkal képesek magas dimenziós terek esetén is ütközésmentes pályát találni. A hagyományos RRT algoritmus hátránya, hogy nem garantálja a megtalált út optimalitását [24].

Az előbb felsorolt problémák kiküszöbölésére a Transition-based RRT (T-RRT) algoritmus mellett döntöttem, mely ötvözi a grid-alapú és az RRT típusú pályatervezési algoritmusok előnyeit [17]. Azaz a T-RRT képes egy költségfüggvénnyel definiált jó minőségű út hatékony keresésére akár magas dimenziójú keresési terek esetén is.

A költségfüggvény definiálása mindig valamilyen heurisztika alapján történik, például egy humán-kollaboratív robotkarnál biztonsági okokból kifolyólag az operátorok körüli konfigurációk esetén (különös tekintettel a fej körüli térre) magas értékre választhatjuk [41]. Az én esetemben a költségfüggvényt úgy definiáltam, hogy az akadályok körüli konfigurációkban magas legyen, majd egyre távolodva tőlük csökkenjen az értéke. A kis számítási igény érdekében egy fuzzy függvény-approximáción alapuló költségfüggvény számító módszert dolgoztam ki.

Az offline pályatervezési feladat megoldásában kapott eredményeimet a 2019-es International Conference on Methods and Models in Automation and Robotics (MMAR) konferencián bemutattam [6].

Offline trajektóriatervezési feladat megoldására a szakirodalomban több módszert is találtam. Egyes módszerek az robotkar időoptimális irányítását tűzik ki célul, kezdetben még a rendszer dinamikáját nem vették figyelembe ezen tervezéseknél [18, 3], míg később már figyelembe véve azt, olyan trajektóriák tervezése is lehetővé vált, melyekkel a robotkar egyes csuklóin kifejtendő nyomatékok is limitálhatóvá váltak [21]. Én egy olyan algoritmust választottam erre a célra, mely biztosítja a robotkar csuklóinak folytonosan változó gyorsulását, így elértem, hogy nem szükséges végtelenül nagy beavatkozójel kiadása, azaz valós rendszeren is megvalósíthatóvá válik az algoritmus [2].

Azon megoldások nagy része, melyek dinamikusan változó környezetben működnek, csupán a környezettel való ütközés detektálását tudják elvégezni, majd ez alapján változtatják a megtervezett trajektória időbeli skálázását (lassítják, megállítják vagy akár visszafelé is mozgathatják a robotkart).

Általában ezt erő-, gyorsulás-, illetve nyomatékméréssel teszik meg [7]. Ha szeretnénk elkerülni az ehhez szükséges szenzorok többletköltségeit vagy az adott robotkar nem rendelkezik ilyen visszajelzésekkel, akkor használhatunk olyan módszert, mely csupán a pozíció- és sebességmérések alapján képes meghatározni az ütközés jelenlétét, illetve annak irányát [8]. Ez a biztonság szempontjából egy elengedhetetlen funkció, viszont ha szeretnénk még robusztusabbá tenni a rendszerünket, akkor az ütközések elkerülésére is érdemes figyelmet fordítanunk.

Vannak olyan mozgástervező módszerek is, melyekkel ismeretlen környezetben is lehetséges ütközésmentes mozgás tervezése. Tipikusan autonóm mobilis robotoknál vagy önvezető autóknál alkalmaznak ilyen algoritmusokat. Célom az volt a feladat megoldása során, hogy robotkarok esetén akár valós időben is használható algoritmust fejlesszek.

A szakirodalomban találtam olyan alkalmazásokat, melyekben a manipulátor környezetét mélységi kamerákkal figyelik meg [7], így meghatározható az akadályok a robotkar egyes pontjaihoz képesti távolsága [13], majd ezen távolságok alapján egy reaktív mozgástervezési metódussal valós időben módosíthatjuk az offline tervezés során kapott trajektóriát. A folyamatot gyorsíthatjuk, ha a számítás egyes részeit videokártyán (GPU-n) végezzük el [5]. A megoldásomban én is ezt a módszert vettem alapul.

Munkám során a fenti módszerek kombinálásával oldottam meg az online tervezési feladatot

Elsőként ismertetem a robotkarok kinematikai leírásának lehetőségeit, illetve esetleges nehézségeit (2. fejezet), majd a 3. fejezetben bemutatom a későbbiekben ismertetett mozgástervezési algoritmusok implementálása és tesztelése szempontjából fontos technológiákat. A 4. fejezetben ismertetem a statikus környezetben történő mozgástervezés lépéseit, kezdve a pályatervezéshez használt T-RRT algoritmus bemutatásával, majd az ütközésdetektálási módszerre térek ki. Ezt használom mind a pályatervezés, mind pedig a 4.3. fejezetben bemutatásra kerülő fuzzy függvényapproximáción alapuló költségfüggvény becslés során. Majd a 4.4. fejezetben kitérek az offline trajektóriatervezésnél használt módszer működésére. Az 5. fejezet az online mozgástervezési feladat megoldását tartalmazza. Bemutatom, hogyan lehet az OpenGL nevű grafikus függvénykönyvtár segítségével virtuális mélységképet generálni. Ezt követően ismertetem a kamera külső paramétereinek genetikus algoritmussal történő meghatározásának menetét. Majd pedig kitérek a mélységtérben történő távolságszámításra és a reaktív mozgástervezés menetére. Végül értékelem az eredményeket és kitérek az esetleges továbbfejlesztési lehetőségekre.

2. fejezet

Robotkar kinematikai leírása

Ebben a fejezetben röviden ismertetem a robotkarok kinematikai leírásának a feladat megoldásához szükséges részleteit.

Kinematikai láncnak nevezzük azon mechanikai rendszereket, melyekben a mechanizmusokat alkotó merev testek között valamilyen kényszerkapcsolat lép fel. Az itt fellépő kényszerkapcsolatok száma megadja a mechanizmus szabadságfokainak a számát (Degrees of Freedom (DoF)) [2].

Beszélhetünk zárt, illetve nyílt kinematikai láncokról. Az előbbi esetén a testek közötti kapcsolati gráf kört tartalmaz, míg nyílt kinematikai lánc esetén ez a tulajdonság nincs meg, a kapcsolati gráfot egy fa segítségével leírhatjuk.

A robotkarok nyílt kinematikai láncként modellezhetőek, a manipulátor egyes szegmensei közötti kapcsolat biztosításáért pedig a csuklók felelősek. Leggyakrabban rotációs, illetve transzlációs csuklókat alkalmazunk. Az előbbi relatív rotációs elmozdulást, míg az utóbbi transzlációs mozgást tesz lehetővé a szomszédos szegmensek között.

Az egyes szegmensek közötti relatív elmozdulást a csuklóváltozóval jellemezhetjük, melyet az i-dik csukló esetén q_i-vel jelölünk. Rotációs csukló esetén a relatív szögelfordulást, míg transzlációs csukló esetén a relatív lineáris elmozdulást adja meg.

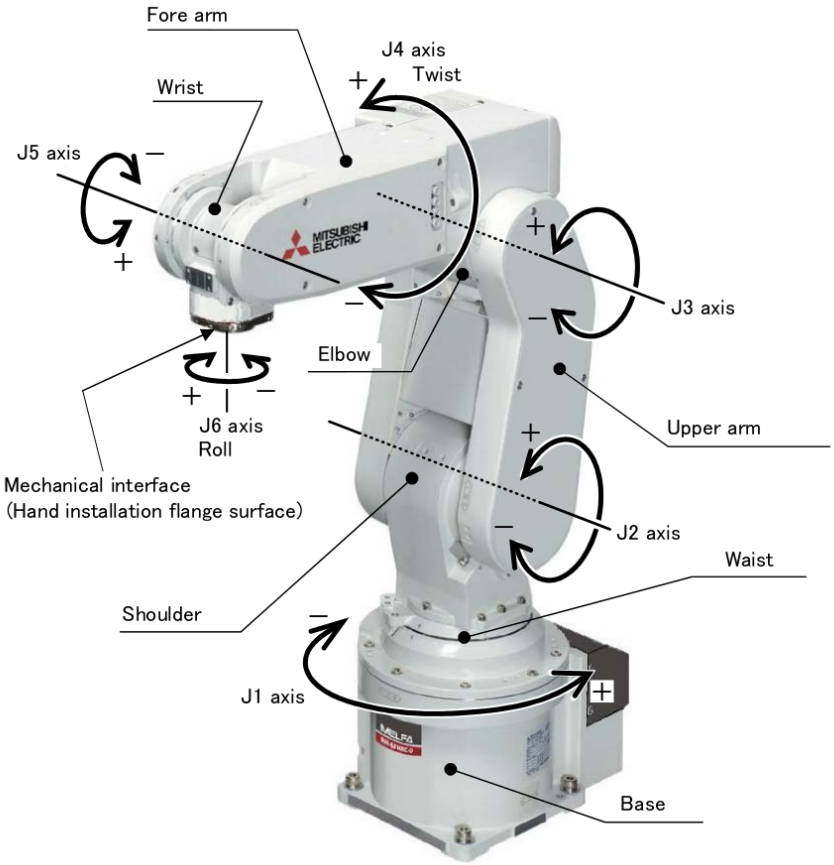
A szimulációk során a Mitsubishi RV-2F-Q típusú 6-DoF robotkarral dolgoztam, melynek felépítése a 2.1. ábrán látható.

2.1. Direkt geometriai feladat

A robotkar végeffektorának pozíciója és orientációja meghatározható a világ koordináta-rendszerben a robot bázisának pozíciójának és orientációjának, illetve a csuklóváltozók aktuális értékének ismeretében. Ezt nevezzük a robot direkt geometriai feladatának.

A manipulátor bázis koordináta-rendszeréből eljuthatunk a végeffektor keretébe úgy, hogy definiálunk minden egyes csuklóhoz egy-egy homogén transzformációs mátrixot, melyek szorzata megadja a végeffektor bázishoz viszonyított helyzetét. Ezen mátrixok egyértelműen megadják az egyes szegmensekhez tartozó koordináta-rendszereket, a 0. keret definiálást követően.

A homogén transzformációs mátrixokat a manipulátorhoz tartozó Denavit-Hartenberg leírás segítségével definiálhatjuk [31]. Minden egyes csuklóhoz rendelhetünk egy koordináta-rendszert, melynek z tengelye konvenció szerint az adott csukló forgástengelyével egyezik meg. A bázis koordináta-rendszer megválasztása úgy történik, hogy a z_0 tengely egybeessen az első csukló forgástengelyével, az x_0 tengely pedig merőleges legyen rá. Az n-edik, azaz az utolsó keret esetén z_n iránya tetszőleges, míg x_n merőleges a z_{n-1} és z_n tengelyekre. Az y koordinátatengelyeket az adott keret másik két tengelye egyértelműen meghatározza, mivel ezek összessége ortogonális rendszert alkot.



2.1. ábra. Az ábrán a szimulációk során használt hat szabadságfokú Mitsubishi RV-2F-Q manipulátor modellje látható [27]

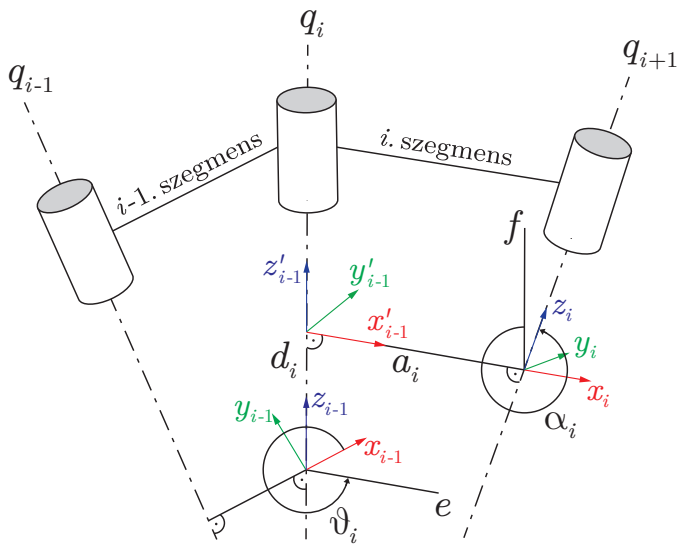
A Denavit-Hartenberg paraméterek szomszédos két csukló közötti transzformációt négy paraméter segítségével írják le:

- ϑ_i: z_{i-1} tengely körüli rotáció
- d_i: z_{i-1} tengely menti transzláció
- α_i: x'_{i-1} tengely körüli rotáció
- a_i: x'_{i-1} tengely menti transzláció

Ezen paraméterek hatása a szomszédos csuklók koordináta-rendszereire a 2.2. ábrán látható.

A trigonometriai összefüggések könnyebb áttekinthetősége érdekében vezessük be az alábbi jelöléseket:

$$S_\alpha := \sin(\alpha), \quad C_\alpha := \cos(\alpha) \tag{2.1}$$



2.2. ábra. A Denavit-Hartenberg konvenció tetszőleges robotkar geometriájának leírására használható. Az ábrán a leírásban található paraméterek jelentésének szemléletes ábrázolása látható. Az e segédegyenes az x_i tengellyel, míg az f a z_{i-1} tengellyel párhuzamos.

Homogén transzformációs mátrixok segítségével mind a rotáció, mind a transzláció egy adott tengely mentén egyszerűen definiálható. A z tengely mentén például az alábbi módon számíthatóak ezen mátrixok:

$$\mathbf{Rot}_z(\alpha) = \begin{bmatrix} C_\alpha & -S_\alpha & 0 & 0 \\ S_\alpha & C_\alpha & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad \mathbf{Trans}_z(a) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & a \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{2.2}$$

Azon $\mathbf{T}_{i-1,i}$ transzformáció, mely a $(i-1)$. és az i. szegmens közti kapcsolatot írja le, a Denavit-Hartenberg paraméterek által definiált rotációk és transzlációk eredőjeként definiálható:

$$\begin{aligned} \mathbf{T}_{i-1,i} &= \mathbf{Rot}_z(\vartheta_i) \cdot \mathbf{Trans}_z(d_i) \cdot \mathbf{Trans}_x(a_i) \cdot \mathbf{Rot}_x(\alpha_i) \\ &= \begin{bmatrix} C_{\vartheta_i} & -S_{\vartheta_i} & 0 & 0 \\ S_{\vartheta_i} & C_{\vartheta_i} & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & 0 & a_i \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & d_i \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & C_{\alpha_i} & -S_{\alpha_i} & 0 \\ 0 & S_{\alpha_i} & C_{\alpha_i} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \\ &= \begin{bmatrix} C_{\vartheta_i} & -S_{\vartheta_i} C_{\alpha_i} & S_{\vartheta_i} S_{\alpha_i} & a_i C_{\vartheta_i} \\ S_{\vartheta_i} & C_{\vartheta_i} C_{\alpha_i} & -C_{\vartheta_i} S_{\alpha_i} & a_i S_{\vartheta_i} \\ 0 & S_{\alpha_i} & C_{\alpha_i} & d_i \\ 0 & 0 & 0 & 1 \end{bmatrix} \end{aligned} \tag{2.3}$$

Ezek alapján a robot bázis koordináta-rendszeréből a végeffektor keretébe juttató transzformációt leíró $\mathbf{T}_{B,E}$ mátrix leírható homogén transzformációs mátrixok szorzataként:

$$\mathbf{T}_{B,E} = \mathbf{T}_{B,0} \cdot \mathbf{T}_{0,1} \cdot \ldots \cdot \mathbf{T}_{i-1,i} \cdot \ldots \cdot \mathbf{T}_{n-1,n} \cdot \mathbf{T}_{n,E}, \tag{2.4}$$

ahol n a robotkar csuklóinak a száma, $\mathbf{T}_{B,0}$ a manipulátor bázisából az első csukló koordináta-rendszerébe juttató mátrix, míg $\mathbf{T}_{n,E}$ a utolsó csukló és a végeffektor relatív elhelyezkedését hivatott leírni.

Meg kell jegyeznünk, hogy minden egyes csuklónál a ϑ_i (rotációs csukló esetén) vagy a d_i (transzlációs csukló esetén) változót csuklóváltozóként is használjuk, így a Denavit-Hartenberg leírásban megadott paraméterek értékét minden pillanatban az aktuális konfigurációval módosítani kell. Így kaptunk egy, a robot aktuális konfigurációjától függő leírást a bázis, illetve végeffektor keret között, amely tehát megoldása a direkt geometriai problémának.

2.2. Inverz geometriai feladat

A robotkar inverz geometriai feladatának megoldása során ismert végberendezés pozíció és orientáció segítségével határozzuk meg a robot lehetséges csuklóváltozóit.

Ez ideálisnak tűnhet, hiszen ezzel a módszerrel akár az euklideszi térben is elvégezhetjük a pályatervezést a végeffektorra, majd az így kapott trajektória minden egyes pontjában megoldva az inverz geometriai feladatot megkaphatjuk a csuklóváltozók kívánt állapotait. Ez valóban működőképes megoldás abban az esetben, amennyiben ismert a robotkar inverz geometriai feladatának megoldása, viszont ennek az analitikus alakja nem minden esetben meghatározható.

Például egy $n = 7$ szabadságfokú robotkar a pozícionálási és orientálási feladatot tekintve redundánsnak tekinthető, azaz létezik a munkaterének egy olyan részhalmaza, melyben a végeffektor egy adott pozíció és orientációját több konfigurációban is el tudja érni.

További probléma még, hogy ezen megoldások sokszor bonyolult trigonometriai összefüggésekhez vezetnek, melyek kiértékelése sok időt vesz igénybe már a pályatervezés során is.

Létezik a robotkaroknak egy részhalmaza, melyekre viszonylag egyszerűen megadható az inverz geometriai feladat megoldása. Amennyiben a robot utolsó három tengelye rotációs és egy pontban metszik egymást, abban az esetben a feladat felbontható egy pozícionáló és egy orientáló részproblémára, melyek egymástól függetlenül megoldhatóak.

Mivel én általános módszer elkészítésére törekedtem, ezért nem alkalmaztam olyan megoldásokat, melyek igényelték volna az inverz feladat megoldását.

2.3. Robotkar differenciális mozgása

Abban az esetben, ha az inverz geometria nem számítható analitikusan, más eszközhöz kell folyamodnunk. Ahelyett, hogy a végeffektor pozícióját és orientációját írnánk elő, annak sebességét és szögsebességét adjuk meg, majd meghatározzuk azon csuklósebességeket, melyek ezen mozgást lehetővé teszik [28].

A robot végeffektora és a csuklóváltozók között nemlineáris kapcsolat van, amely egy adott pontban linearizálható. Azaz egy adott konfigurációban az i. csukló változó értéke q_i, majd ezt δq_i-vel megváltoztatva úgy, hogy minden más csukló értéke változatlan marad, megkaphatjuk az adott pontban az i. csukló hatását a végeffektor sebességére és szögsebességére.

Ekkor definiálhatjuk az i. csukló által kifejtett parciális ${}^n\beta_{i-1}$ sebességeket és ${}^n\gamma_{i-1}$ szögsebességeket:

$$\begin{aligned} \text{Rotációs csukló esetén:} \quad & {}^n\boldsymbol{\beta}_{i-1} = \mathbf{A}_{i-1,n}^{-1} \cdot (\mathbf{z}_{i-1} \times \mathbf{p}_{i-1,n}) \\ & {}^n\boldsymbol{\gamma}_{i-1} = \mathbf{A}_{i-1,n}^{-1} \cdot \mathbf{z}_{i-1} \\ \text{Transzlációs csukló esetén:} \quad & {}^n\boldsymbol{\beta}_{i-1} = \mathbf{A}_{i-1,n}^{-1} \cdot \mathbf{z}_{i-1} \\ & {}^n\boldsymbol{\gamma}_{i-1} = 0 \end{aligned} \tag{2.5}$$

ahol ${}^n\boldsymbol{\beta}_{i-1}$ és ${}^n\boldsymbol{\gamma}_{i-1}$ felső indexe jelöli azt, hogy az n. szegmens koordináta-rendszerében adjuk meg a változók értékeit, $\mathbf{z}_{i-1}$ az i. szegmenshez tartozó keret z tengelyének, azaz az adott csukló forgástengelyének irányába mutató $\begin{bmatrix}0 & 0 & 1\end{bmatrix}^\top$ egységvektor, $\mathbf{p}_{i-1,n}$ pedig a végeffektor keretébe mutató vektor az i. csuklóhoz tartozó koordináta-rendszerből tekintve. Az $\mathbf{A}_{i-1,n}$ mátrix az $(i-1)$. és az n. koordináta-rendszer közötti rotációt írja le, ennek inverzével balról szorozva transzformálhatunk vektorokat az $(i-1)$. koordináta-rendszerből a végeffektor keretébe.

Ez utóbbit egyszerűen megkaphatjuk a $\mathbf{T}_{i-1,n}$ homogén transzformáció segítségével:

$$\mathbf{T}_{i-1,n} = \left[\begin{array}{ccc|c} & \mathbf{A}_{i-1,n} & & \mathbf{p}_{i-1,n} \\ \hline 0 & 0 & 0 & 1 \end{array}\right] \tag{2.6}$$

Ekkor a végeffektor eredő sebessége és szögsebessége meghatározható az előbbiekben kiszámított paraméterek és az aktuális csuklósebességek ismeretében:

$$\begin{aligned} {}^n\mathbf{v}_n &= \sum_{i=1}^{n} {}^n\boldsymbol{\beta}_{i-1}\dot{q}_i \\ {}^n\boldsymbol{\omega}_n &= \sum_{i=1}^{n} {}^n\boldsymbol{\gamma}_{i-1}\dot{q}_i \end{aligned} \tag{2.7}$$

Ezt követően bevezethetjük a robot Jakobi-mátrixát, mely kapcsolatot teremt a csuklóváltozók sebessége, illetve a végeffektor sebessége és szögsebessége között:

$$\begin{bmatrix} {}^n\mathbf{v}_n \\ {}^n\boldsymbol{\omega}_n \end{bmatrix} = \begin{bmatrix} {}^n\boldsymbol{\beta}_0 & {}^n\boldsymbol{\beta}_1 & \cdots & {}^n\boldsymbol{\beta}_{n-1} \\ {}^n\boldsymbol{\gamma}_0 & {}^n\boldsymbol{\gamma}_1 & \cdots & {}^n\boldsymbol{\gamma}_{n-1} \end{bmatrix} \begin{bmatrix} \dot{q}_1 \\ \dot{q}_2 \\ \vdots \\ \dot{q}_n \end{bmatrix} = {}^n\mathbf{J}_n \cdot \dot{\mathbf{q}} \tag{2.8}$$

Ezt a mátrixot áttranszformálhatjuk a bázis keretbe, így a bázis koordináta-rendszerben megadott sebességek és szögsebességek a csuklóváltozók által számíthatóvá válnak.

$${}^0\mathbf{J}_n = \begin{bmatrix} \mathbf{A}_{0,n} & \mathbf{0} \\ \mathbf{0} & \mathbf{A}_{0,n} \end{bmatrix} {}^n\mathbf{J}_n, \qquad \begin{bmatrix} {}^0\mathbf{v}_n \\ {}^0\boldsymbol{\omega}_n \end{bmatrix} = {}^0\mathbf{J}_n \cdot \dot{\mathbf{q}} \tag{2.9}$$

Ezzel a módszerrel, ha az inverz geometria nem számítható analitikusan, akkor megfogó sebességét és szögsebességét előírva a ${}^0\mathbf{J}_n$ Jakobi-mátrix pszeudoinverzének segítségével számítható $\dot{\mathbf{q}}$ vektor, majd ennek numerikus integrálásával meghatározható a csuklóváltozók $\mathbf{q}(t)$ időfüggvénye.

Az adott feladat szempontjából redundáns robotkar esetén végtelen sok megoldás közül kell választanunk, célszerű ezt úgy megtenni, hogy minimalizáljuk a $\|\dot{\mathbf{q}}\|$ normát, így a lehető legrövidebb tranzienshez juthatunk [2].

$$\dot{\mathbf{q}} = {}^0\mathbf{J}_n^{\#} \begin{bmatrix} {}^0\mathbf{v}_n \\ {}^0\boldsymbol{\omega}_n \end{bmatrix} = {}^0\mathbf{J}_n^{\top} ({}^0\mathbf{J}_n \, {}^0\mathbf{J}_n^{\top})^{-1} \begin{bmatrix} {}^0\mathbf{v}_n \\ {}^0\boldsymbol{\omega}_n \end{bmatrix} \tag{2.10}$$

ahol ${}^0\mathbf{J}_n^{\#}$ a Jakobi-mátrix pszeudoinverzét jelöli.

Pozicionálási és orientálási feladat során egy $n = 6$ szabadságfokú robotkar esetén a robot konfigurációs terének létezik olyan részhalmaza, melyben minden egyes pozíciót bármely orientációban el tud érni, ekkor a mátrix pszeudoinverze megegyezik a hagyományos mátrix inverzzel. Ezen a halmazon kívül a manipulátor Jakobi-mátrixa szinguláris lesz, azaz ekkor bizonyos irányok mentén vagy körül nem tud mozogni a robot.

Fontos megjegyezni, hogy nem csupán a végeffektorra definiálhatunk Jakobi-mátrixot, hanem a robotkar tetszőleges C kontrollpontja esetére. Ekkor a fenti módszerrel analóg módon meghatározható az adott ponthoz tartozó ${}^0\mathbf{J}_C$ Jakobi-mátrix.

3. fejezet

Használt technológia ismertetése

A feladat megoldása során többféle hardver és szoftver eszközt alkalmaztam, melyek rövid ismertetését tartalmazza ez a fejezet.

3.1. Szimuláció készítése Simulink környezetben

Az algoritmust a Mitsubishi RV-2F-Q típusú hatszabadságfokú robotkaron teszteltem, melynek főbb fizikai paramétereit az F.1. táblázat tartalmazza. A statikus pályatervezés tesztelésére létrehozott szimulációt a MATLAB Simulink Simscape Multibody toolboxa segítségével készítettem el, mely képes CAD tervezőszoftverben elkészített 3D-s modell ábrázolására, illetve dinamikus szimulációjára.

3.1.1. Simscape Multibody toolbox

A MATLAB Simulink Simscape toolbox-a hatékonyan képes komplex rendszerek modellezésére, legyenek azok elektronikai, hidraulikai, termikus, mechanikai, stb. problémák, vagy akár ezek kombinációi. Ezen a toolboxon belül található a Multibody könyvtár, amely merev testek dinamikai modellezését teszi lehetővé [26].

A ”World frame” vagy világ koordináta-rendszer segítségével definiálhatjuk a szimuláció bázis koordináta-rendszerét. Ez a keret időben állandó, homogén transzformációk segítségével adhatjuk meg más objektumok helyzetét a térben ehhez a koordináta-rendszerhez képest. Ezt a blokkot egy Simulink modellben csak egyszer helyezhetjük el.

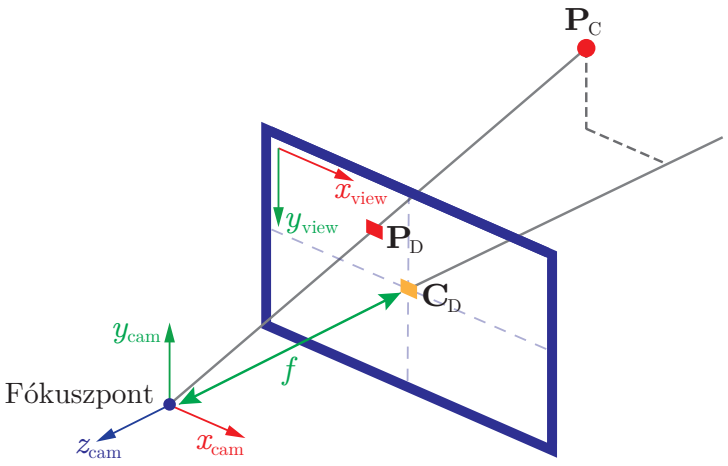
A ”Rigid Transform” blokk segítségével egy időben állandó homogén transzformációt definiálhatunk. A bemenetén egy koordináta-rendszert vár, melynek a transzformáltját adja ki a kimenetén, így akár homogén transzformációk sorozata is megvalósítható ezen blokk kaszkádosításával.

Ezzel a blokkal definiálhatjuk az egyes csuklók közötti konstans transzformációkat, melyeket a Denavit-Hartenberg alakból (2.1. fejezet) kaphatunk meg.

A ”Transformation Sensor” blokk segítségével mérhetjük két tetszőleges keret közötti eredő transzformációs mátrixot. Hasznos eszköz volt a hibakeresési folyamatok során.

A ”Solid” blokk segítségével merev testeket modellezhetünk. Megadható paraméterként a test geometriája. Ez választható előre definiált alakzatként (például téglatestnek, gömbnek) vagy megadható CAD szoftverből exportálható fájlként is. Definiálhatóak ezen kívül a test dinamikus paraméterei, mint a tömeg, tömegközéppont, tehetetlenségi nyomatékok stb. Ezek a paraméterek a CAD szoftver segítségével szintén mérhetőek.

A ”Revolute Joint” blokk rotációs csukló definiálására szolgál. Segítségével dinamikus transzformációt definiálhatunk különböző keretek között. Választható, hogy mi alapján szeretnénk vezérelni a mozgást: lehetséges a kiadott nyomaték szabályozása, illetve lehetőség van a konkrét trajektória érvényre juttatására, a csuklóváltozók, illetve első- és



3.1. ábra. Az ábrán látható a képtérre történő projekció menete, illetve a képtér és a kamera valós koordináta-rendszere is ábrázolva van

második deriváltjainak megadásával. Kimeneten szolgáltatják mind az adott csuklóváltozó tényleges értékét, mind első- és második deriváltjait, illetve a csuklókon ható nyomatékok megfigyelésére is lehetőségünk van.

Ezen eszközökkel megvalósítottam egy Simulink alrendszert, melynek bemenete az adott pillanatbeli kívánt csuklógyorsulások, illetve a kezdeti konfiguráció. A modell kimenete pedig a csuklók pozíciója, sebessége, gyorsulása, illetve a rajtuk fellépő nyomaték. A megvalósított modell az F.1. ábrán látható.

3.2. RGB-D kamera

Feladatom során a robot körüli környezet detektálásához RGB-D kamerát használtam, ezen belül is a Microsoft Kinect 1.0-s eszközt. Ezen kamerák jellegzetessége, hogy a hagyományos színtérben történő leképzés mellett képesek az úgynevezett mélységkép előállítására is.

A ”mélységtér” egy nemhomogén tér, a valós tér egy leképzése, melyben az első két koordináta megadja a valós tér egy pontjának képsíkra vett projekcióját, míg az adott pixel értéke pedig a pontnak a kamerától vett távolságát jelenti [14].

3.2.1. Mélységtér leképzése

Az adott kamerára jellemző leképzést két mátrix segítségével adhatjuk meg.

A kamera belső (instrinsic) paramétereit a $\mathcal{K}$ mátrixszal modellezhetjük. Ez leírja a valós tér egy pontjának a képsíkra történő projekcióját. A külső (extrinsic) paramétereket egy ε homogén transzformációs mátrixba foglalhatjuk, mely a kamera pozícióját és orientációját mondja meg egy választott referenciakerethez képest.

$$\mathcal{K} = \begin{pmatrix} \frac{f}{s_x} & 0 & c_x \\ 0 & \frac{f}{s_y} & c_y \\ 0 & 0 & 1 \end{pmatrix}, \quad \varepsilon = (\mathbf{R} \mid \mathbf{t}), \tag{3.1}$$

3.2. ábra. Microsoft Kinect 1.0 RGB-D kamera [39]

ahol f a kamera fókusztávolsága, s_x és s_y egy pixel méretét adja meg, míg c_x és c_y a kép középpontjának pixelkoordinátákban megadott pozícióját jelöli. $\mathbf{R}$ és $\mathbf{t}$ a kamera és a referencia koordináta-rendszer közötti rotációt és transzlációt jelenti.

Tehát ha vesszük a tér egy tetszőleges $\mathbf{P}_R = \begin{pmatrix} x_R & y_R & z_R \end{pmatrix}^\top$ pontját, akkor ennek a pontnak a kamera keretében vett koordinátáit megkaphatjuk az alábbi kifejezés segítségével:

$$\mathbf{P}_C = \begin{pmatrix} x_C & y_C & z_C \end{pmatrix}^\top = \mathbf{R}\mathbf{P}_R + \mathbf{t} \tag{3.2}$$

Majd a projekcióval megkaphatjuk a pont mélységtérben elfoglalt $\mathbf{P}_D = \begin{pmatrix} p_x & p_y & d_p \end{pmatrix}^\top$ helyét. A projekció menete látható a 3.1. ábrán. A szakirodalomban talált képletek helytelenek voltak, így ezeket magam vezettem le [14].

$$\begin{aligned} p_x &= c_x - \frac{x_C f}{z_C s_x} \\ p_y &= c_y + \frac{y_C f}{z_C s_y} \\ d_p &= -z_C, \end{aligned} \tag{3.3}$$

ahol p_x és p_y a pont pixelben megadott koordinátái, míg d_p a pont távolsága a kamerától. (3.3)-ban látható, hogy a projekció nincs hatással a harmadik koordináta értékét csupán megnegálja.

3.2.2. Microsoft Kinect kamera

Feladatom során a Microsoft Kinect 1.0-s típusú kamerával dolgoztam (ld. 3.2. ábra). Az eszközön található egy RGB kamera, egy infravörös jeladó és egy vevő.

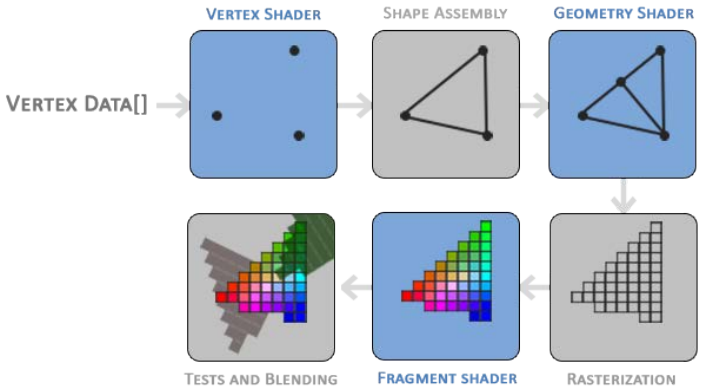
A mélységkép előállítása többféleképpen is történhet. A Kinect 1.0-s változata az infravörös jeladóval egy hálót vetít ki a mérni kívánt területre, majd a vevővel méri ennek a deformációját, ebből számítható minden egyes pontnak a kamerától vett távolsága. A Kinect 2.0 típusú eszköz már az úgynevezett Time-of-Flight (ToF) alapon működik. Ez azt jelenti, hogy a kamera fényt bocsát ki, majd méri azt az időt, ami ahhoz kell, hogy az útjába kerülő objektumokról a fény visszaérkezzen a kamerához.

Későbbiekben érdemes lehet áttérni a Kinect 2.0-ás verziójára. Ahogy a [38]-as cikkben is látható, ez az eszköz már jobb minőségű mélységképet tud szolgáltatni. További hátránya az 1.0-ás verziónak, hogy a működéséből adódóan a mélységkép mérési zaja a távolsággal exponenciálisan nő, míg a Kinect 2.0-ás verzióban ez a zaj konstans a teljes mérési tartományon.

A 3.1. táblázatban a Microsoft Kinect 1.0 és a Microsoft Kinect 2.0 kamerák képalkotásra vonatkozó főbb adatai láthatóak.

	Microsoft Kinect 1.0	Microsoft Kinect 2.0
RGB kép felbontása	640×480	1920×1080
Mélységkép felbontása	640×480	512×424
Mintavételi frekvencia	30 Hz	30 Hz
Mérési tartomány	$0.4 - 4.5m$	$0.5 - 4.5m$
Vízszintes látószög	57°	70°
Függőleges látószög	43°	60°

3.1. táblázat. Microsoft Kinect 1.0 és 2.0 főbb adatai [4, 38]



3.3. ábra. Az OpenGL által használt grafikus pipeline vázlata [9]

3.3. OpenGL

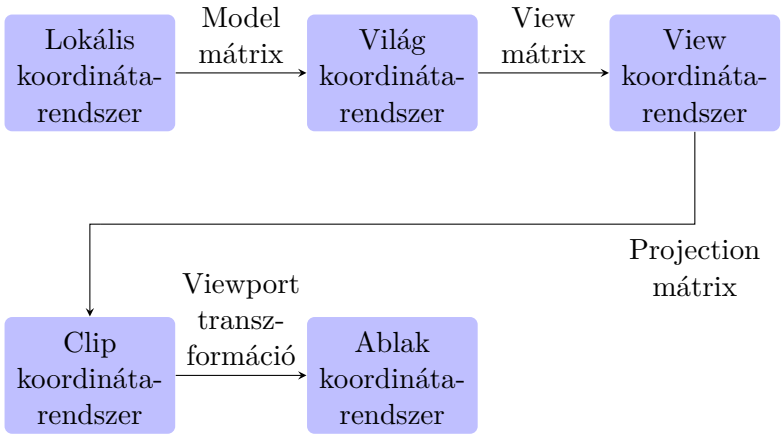
Az Open Graphics Library (OpenGL) a Silicon Graphics nevű amerikai cég által kifejlesztett függvénykönyvtár, mely lehetővé teszi háromdimenziós grafikai objektumok létrehozását, azok manipulációját, illetve megjelenítését. Főleg CAD rendszerek, szimulátorok használják és bár játékfejlesztésre is alkalmas, arra más API-k terjedtek el (például: Direct3D). Az általa biztosított több száz függvény segítségével könnyedén kihasználhatjuk a grafikus kártyák nyújtotta lehetőségeket.

A grafikus kártyák (GPU-k) a CPU-kon használt egyszálas számítási architektúrával ellentétben az adatfolyam elvű programozást teszik lehetővé. A mai GPU-kban akár több ezer műveletvégző egység található, melyek bár egyesével nem rendelkeznek olyan teljesítménnyel, mint egy CPU mag, viszont együttesen támogatják a Single Instruction Multiple Data (SIMD) típusú parancsokat, azaz egy adott utasítást nagy mennyiségű adaton, párhuzamosan képesek kiértékelni [30].

Az OpenGL alapvetően olyan egyszerű primitíveket tud kirajzolni, mint pontok, szakaszok, illetve poligonok (általában háromszögek). Ezek leírására a vertex-eket használjuk, melyek mindegyike a tér egy-egy pontját írja le homogén koordinátákkal.

3.3.1. Grafikai csővezeték modell

A grafikai megjelenítésre szolgáló függvénykönyvtárak és szoftverek előszeretettel használják az úgynevezett grafikai csővezeték (graphics pipeline) modellt, mely három dimenziós koordinátákból képes előállítani a megjelenítendő színes pixelekből álló képet, nincs ez másképp az OpenGL esetén sem. Az OpenGL által használt folyamat a 3.3. ábrán lát-



3.4. ábra. A kép előállítása során elvégzendő koordinátatranszformációk

ható. A kékkel jelölt szakaszok olyan műveleteket jelölnek, melyeket felül tudunk írni az általunk megírt shaderrel[1].

A folyamat bemenete a vertex data, ami a megjelenítendő primitíveket leíró csúcsok koordinátáit tartalmazó halmaz.

Ezek az adatok a vertex shaderhez kerülnek, melynek feladata feldolgozni a kapott adatokat. Például ha egy kamerát szeretnénk modellezni, akkor ebben a rétegben végezzük el a 3.3.2. fejezetben is leírt transzformációkat, illetve az adott kamerára jellemző projekciót. Fontos megjegyezni, hogy a párhuzamos végrehajtás miatt egy shader mindig csak egyetlen pixellel dolgozik.

A shape assembly szakaszban a vertex shader által szolgáltatott csúcsokból előállítja a kívánt primitíveket, melyeket a geometry shader elemi primitívekre bont, például háromszögekre.

Ezt követően az előállított primitívek raszterizálása következik, mely során kiválasztja a végső kép azon pixeleit, melyek az adott primitívekhez tartoznak.

A fragment shader szakaszban történik a pixelek színének meghatározása. Ebben a rétegben vannak az olyan külső tényezők figyelembe véve, mint a megvilágítás erőssége és színe, illetve az árnyékok hatásai.

Végül az objektum átlátszósági vizsgálata (alpha test) következik, illetve itt található a mélység értékek számítása, melyből el lehet dönteni, hogy egymást takaró objektumok közül melyik az, amelyiket meg kell jelenítenünk, és melyiket hagyhatjuk figyelmen kívül. A keretrendszer lehetőséget nyújt nekünk arra, hogy az itt meghatározott mélységpuffert, azaz minden pixelre vonatkozó legközelebbi pont távolságát az OpenGL-től lekérjük. Ezt a funkciót a későbbiekben még ki fogjuk használni.

3.3.2. Koordináta-rendszerek

Fontos még szót ejteni a vertex shaderben megvalósítandó koordinátatranszformációkról, melyek az objektumok saját koordináta-rendszereiből a megjelenített kép koordináta-rendszerébe visz. Ez a folyamat a 3.4. ábrán látható.

A kiindulási koordináta-rendszer minden egyes objektumnak a saját kerete. Ha a megjelenítendő alakzat egy CAD rendszerben megtervezett 3D-s objektum, akkor ez a keret a tervezés során definiált koordináta-rendszer.

[1]Shader: A GPU egyes magjain futó elemi program

Első lépésként az objektum saját koordinátáit át kell transzformálni a világ koordináta-rendszerbe, ez a minden egyes objektumra definiált Model mátrix segítségével tehető meg. Ez a homogén transzformáció definiálja azt a rotációt és transzlációt, mely segítségével egy közös koordináta-rendszerben meghatározható az objektum pozíciója és orientációja.

Ezt követően át kell térnünk az adott kamera szemszögébe. Először is a View mátrixszal megadjuk az objektumok helyzetét a kamerához rögzített koordináta-rendszerből tekintve. Ez a View mátrix megegyezik a 3.2.1. fejezetben ismertetett külső kamera paramétereket leíró ε mátrixszal.

Ezután a kamerára jellemző perspektív projekció elvégzése következik. Ez a kamera belső paraméterei által meghatározott Projection mátrix segítségével tehető meg. Ez a 3.2.1. fejezetben látható $\mathcal{K}$ mátrixtól annyiban tér el, hogy az OpenGL a Clip koordinátarendszer a kamera látóterébe eső részét a $[-1.0, 1.0]$ intervallumba skálázza.

Végül a Viewport transzformáció segítségével eljutunk az ablak koordinátarendszerbe, mely a $[-1.0, 1.0]$ intervallum skálázása a megjelenített ablak méretének megfelelő arányban. Így a folyamat eredményeként pixelkoordinátákban kaptuk meg az objektumok pozícióját a képen.

3.4. V-REP

Az online mozgástervezés során nem az előbbiekben bemutatott MATLAB Simulink környezetben működő szimulációt használtam, mivel a Virtual Robot Experimentation Platform (V-REP) szimulációs környezet több olyan tulajdonsággal is rendelkezik, ami miatt jobban megfelel az általam fejlesztett algoritmus tesztelésére (például a gyors szimuláció, Kinect kamera szimulációjának lehetősége).

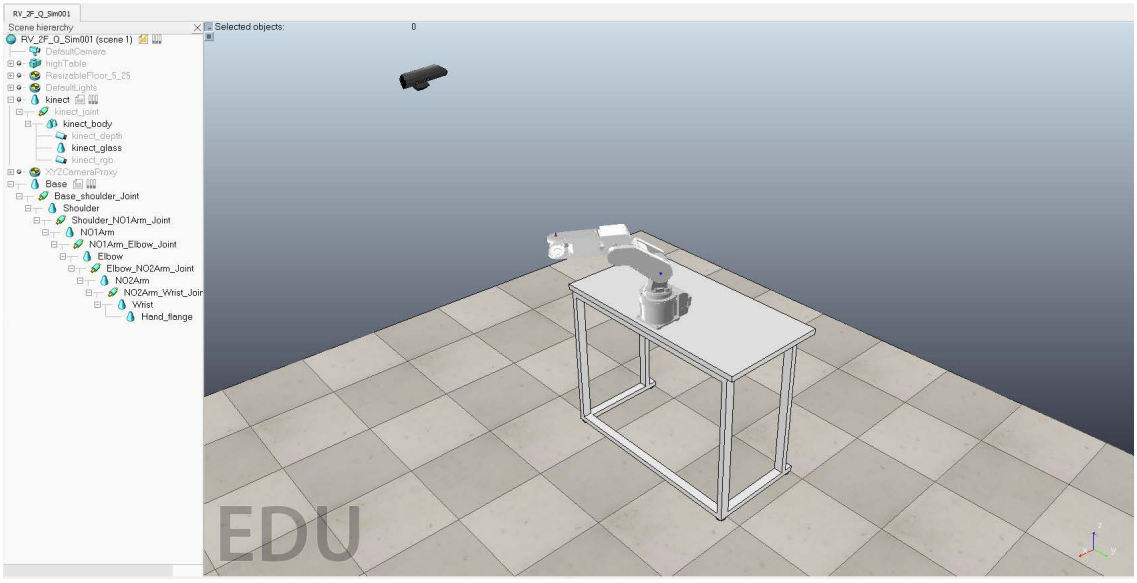
Algoritmusok fejlesztése során kiemelkedően fontos, hogy ne csak az éles rendszeren tudjunk tesztelni, hanem egy virtuális környezetben is képesek legyünk megbizonyosodni a megvalósított program helyes működéséről. Nincs ez másképp a robotok esetén sem, éppen ezért több szimulátorral is találkozhatunk, melyek képesek a robotokhoz kapcsolódó tesztek támogatására.

Ezek közül az egyik legelterjedtebb alkalmazás a V-REP. Ez egy flexibilis, platform független szimulátor, mely egy könnyen használható felületet ad mechanikai rendszerek akár dinamikus szimulációjára.

A V-REP-nek nem adható meg fő funkciója, sokkal inkább több, egymástól független funkciót valósít meg, melyek engedélyezhetőek, illetve tilthatóak az adott alkalmazás igényeinek megfelelően [29].

Az alábbi fő modulok találhatóak meg a keretrendszerben:

- **Kinematikai modul:** kinematikai számításokat tesz lehetővé, beleértve ebbe a robotkarokhoz tartozó direkt- és inverz geometriai számítások elvégzését.
- **Dinamikai modul:** lehetővé teszi az objektumok mozgásának dinamikai modellezését, mint a testek tehetetlensége, illetve ütközések, megfogás esetén történő interakciók.
- **Ütközés detektáló modul:** segítséget nyújt az objektumok közötti ütközések észlelésében.
- **Távolság számító modul:** gyorsan képes meghatározni bármely két alakzat közötti legkisebb távolságot.



3.5. ábra. A V-REP-ben elkészített szimulációs környezet

- **Pályatervezési modul:** mind holonóm, mind anholonóm rendszerek esetén képes elvégezni a pályatervezési feladatot (akár kinematikai láncok esetén is). A pályatervezéshez egy RRT típusú pályatervezési algoritmust használ.

3.4.1. Scene objektumok

A szimuláció elkészítéséhez úgynevezett scene objektumokat kell definiálnunk, melyek felelősek az egyes alakzatok, funkciók modellezéséért.

A mostani alkalmazásban fontosabb szerepet betöltő objektumok típusai az alábbiak:

Csuklók (Joints)

A csuklók kettő vagy több objektum közötti kötéseket, illetve mozgási típusokat definiálására szolgálnak.

Ezek típusa lehet transzlációs az egyenes vonal mentén történő relatív elmozdulás definiálásra, rotációs, a relatív szögelfordulás modellezésére.

Ezen kívül a csukló lehet screw típusú is, mely egy egy-szabadságfokú mozgást tesz lehetővé, a transzlációs és rotációs csuklók egy speciális kombinációja, ami a csavarok mozgásához hasonló mozgási mintákat tud modellezni.

Illetve lehetséges a gömbcsukló definiálása, mely egy három szabadságfokú forgatást tesz lehetővé, működése ekvivalens három darab rotációs csuklóval, melyek irányát megadó vektorai lineárisan független vektorrendszert alkotnak.

A csuklókat többféle üzemmódban is használhatjuk. Erő/nyomaték módban az általuk kifejtett erőt/nyomatékot írjuk elő. Lehetséges kívánt pozíció/szög megadása is, ekkor ha engedélyezve vannak a dinamikai számítások, akkor csukló egy motorként viselkedik, mely rendelkezik maximális nyomatékkal, a pozíciószabályzásra pedig egy általunk megadott paraméterű PID szabályzót használhatunk.

Ezen kívül létezik még az úgynevezett inverz kinematikai mód, amikor csuklók egy csoportját képes úgy vezérelni a szimulációs környezet, hogy a csuklókkal mozgatható viszonyítási pont (például robotkar esetén a végeffektor) megadott konfigurációba kerüljön.

Alakzatok (Shapes)

Az alakzatok olyan háromszög alapú hálók (mesh-ek), melyeket testek megjelenítésére, illetve dinamikai modellezésére használhatunk .

A CAD szoftverek mindegyike képes az elkészített 3D-s modell *.stl* formátumba való exportálására, mely szintén ilyen háromszög alapú mesh-ek leírására szolgáló fájlformátum. A V-REP pedig képes ezen fájlok importálására, így akár saját eszközt is tudunk szimulálni a környezet segítségével, nem szükséges gyárilag biztosított könyvtárra szorítkozni.

Vision szenzorok

Vision szenzorok segítségével komplex információkat szerezhetünk a térről (színek, méretek, mélység információ). A 3.5. ábrán látható a Kinect kamera modellje, mely tartalmaz egy RGB és egy mélység szenzort, ahogy az a bal oldali hierarchián látható. A szenzorok kiértékelését GPU segítségével gyorsítja, erre pedig a 3.3. fejezetben már ismertetett OpenGL-t használja.

Erő és nyomaték szenzorok

Két test között fellépő erő és nyomaték mérésére szolgál, illetve képes jelet adni, ha a mért érték egy előre beállított határértéket túllépett.

3.4.2. Remote API

A V-REP nem csupán a szimulációs környezet számára teszi elérhetővé a benne található objektumokat, hanem egy API segítségével lehetőséget ad külső programból a szimuláció monitorozására, illetve manipulációjára.

Lehetőség van olyan alapvető funkciók megvalósítására, mint egy szimuláció betöltése, elindítása, leállítása, de az API segítéségével szimuláció közben lekérhetjük az egyes szenzorok mért értékeit, illetve akár beavatkozójeleket is kiadhatunk az egyes csuklókra.

4. fejezet

Pálya- és trajektóriatervezés statikus környezetben

Ebben a fejezetben bemutatom azt az általam fejlesztett algoritmus-sorozatot, mely segítségével megvalósítottam statikus térben a robotkar ütközésmentes pályatervezését, majd az így kapott konfiguráció-sorozatból megvalósítható időfüggvények generálását.

A pályatervezési algoritmusok feladata egy olyan konfiguráció-sorozat generálása, melyen végighaladva ütközésmentesen eljuthatunk egy kezdeti konfigurációból egy kívánt célállapotba.

Robotkarok esetén általában a manipulátor végeffektorának egy kívánt állapotába (pozícióba és orientációba) szeretnénk eljuttatni a rendszert. Ezt a célt definiálhatjuk az Euklideszi-térben, viszont a robotkar pályájának tervezését már az úgynevezett konfigurációs térben végeztem el. Ebben a térben a robot csuklóváltozóit tekintjük a koordinátáknak, így a tér egy adott pontja a manipulátor egy adott konfigurációját jelöli ki. A konfigurációs tér minden egyes pontjában a végeffektor pozíciója meghatározható a direkt geometriai feladat megoldásaként, melyet a 2.1. fejezetben ismertettem.

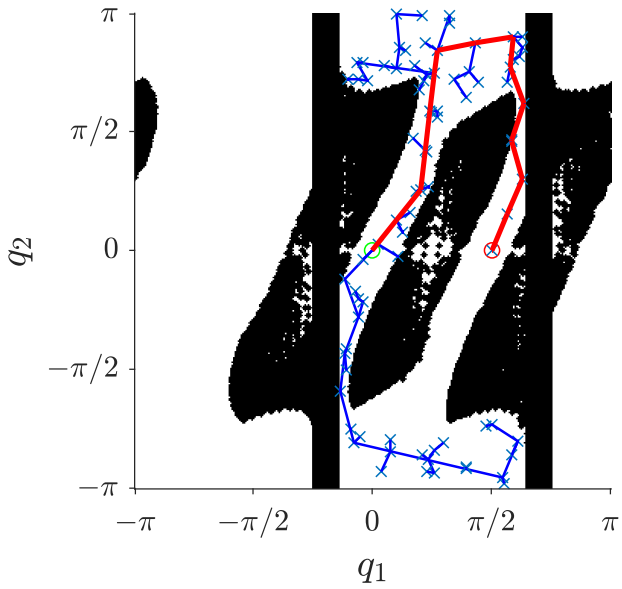
A trajektória generálás során a pályatervezés kimeneteként megkapott konfigurációsorozatot alakítjuk át olyan időfüggvényekké, melyek az adott rendszerben található szabályzási algoritmus alapjeléül szolgálhatnak. Az általam bemutatott algoritmusban előírom a robotkar csuklóváltozóinak, illetve azok első- és második deriváltjainak időfüggvényeit is.

4.1. Pályatervezési algoritmus

A bevezetésben bemutatott szempontok alapján az RRT módszert választottam a robotkar pályatervezési problémájának megoldására. Először bemutatom a hagyományos RRT algoritmust, majd ennek egy továbbfejlesztett verzióját.

4.1.1. RRT algoritmus

A hagyományos RRT algoritmus egy inkrementális, véletlenszerű mintavételezésen alapuló keresési módszer. Alapvetően egy keresési fa növelésével érjük el azt, hogy egy kiindulási q_{init} konfigurációból eljussunk a robotkar kívánt q_{goal} konfigurációjába. Az RRT egy sztochasztikus mintavételezésen alapuló eljárás, azaz az algoritmus egy adott állapotában a keresési fába újonnan beszúrandó pont a konfigurációs tér egy véletlenszerűen kiválasztott pontja [24].



4.1. ábra. Az RRT algoritmus megvalósítása egy kétszabadságfokú robotkar konfigurációs terében. Látható, hogy a pirossal jelölt megoldás nem a lehető legrövidebb utat szolgáltatja a kezdeti és a cél pont között.

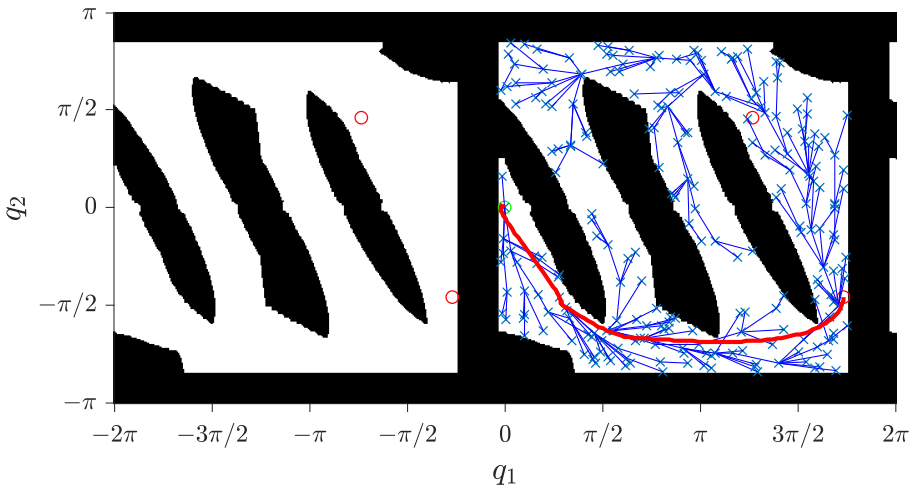
Manapság az RRT-szerű pályatervezési algoritmusok nagy népszerűségnek örvendenek, köszönhetően azon tulajdonságuknak, hogy képesek a keresési tér gyors feltérképezésére.

A tradicionális RRT működését az 1. algoritmusban, az eredményként kapott keresési fát, illetve a megtalált utat pedig a 4.1. ábrán láthatjuk. Az ábrán egy kétszabadságú robotkar esetére mutatok be egy példát az RRT módszer által szolgáltatott pályára. A feketével jelölt konfigurációkban ütközés történne, a kék gráf ütközésmentes mozgásokat ír le, a piros út pedig a zöld kiindulási- és a piros célkonfiguráció közötti pályatervezési feladat megoldását jelöli. A robotkar valós térben való mozgását az alábbi videó szemlélteti: `https://youtu.be/kY8sc2BG5dc`.

Algorithm 1: Rapidly-exploring Random Tree (RRT)

Input: the configuration space $\mathcal{C}$
the root q_{init} and the goal q_{goal}
Output: the tree $\mathcal{T}$

$\mathcal{T} \leftarrow$ InitTree(q_{init})
while not StopCondition($\mathcal{T}, q_{goal}$) **do**
  $q_{rand} \leftarrow$ SampleConf($\mathcal{C}$)
  $q_{near} \leftarrow$ NearestNeighbor($q_{rand}, \mathcal{T}$)
  $q_{new} \leftarrow$ Extend($\mathcal{T}, q_{rand}, q_{near}$)
  if NoCollision(q_{new}) **then**
    AddNewNode($\mathcal{T}, q_{new}$)
    AddNewEdge($\mathcal{T}, q_{near}, q_{new}$)
  end if
end while
return $\mathcal{T}$



4.2. ábra. Az RRT* algoritmus megvalósítása, az algoritmus egy kétszabadságfokú robotkar konfigurációs terében értékeltem ki, az ábrán pirossal jelöltem a megtalált utat. Az ábrán látható elrendezés során többirányú RRT*-ot valósítottam meg, azaz mind a kezdeti pozícióból, mind a lehetséges célállapotokból indítottam egy-egy keresési fát. Az ábrán csak a kezdeti állapotból indított fa látható.

Az algoritmus bemenetei a konfigurációs tér, ahol a pályatervezést elvégezzük, illetve a pályatervezés kezdeti- és egy vég konfigurációja. A metódus kimenete az algoritmus során létrehozott keresési fa. Első lépésként ezt a fát a bemeneten kapott kezdeti konfigurációval inicializáljuk, majd egy ciklusban folyamatosan növelve próbáljuk megközelíteni a célpontot.

A SampleConf függvényben egyenletes eloszlást követve kiválasztjuk a konfigurációs tér egy véletlenszerű q_{rand} pontját.

Ezt használva a NearestNeighbor függvényben megkeressük az aktuális keresési fában az ehhez a ponthoz legközelebbi q_{near} konfigurációt. Ennél a lépésnél léteznek olyan megoldások, melyek nem csak a gráf csúcsai között keresik a legkisebb távolságra elhelyezkedőt, hanem a pontokat összekötő éleket is vizsgálják ebből a szempontból. Én ezt a módszert az algoritmus implementálása során a nagyobb számítási költsége miatt nem alkalmaztam.

Az Extend függvényben generálunk egy q_{new} konfigurációt, amely q_{near}-hez képest ugyanolyan irányban található, mint q_{rand}, viszont a q_{new} és q_{near} konfigurációk távolságát egy δ paraméterrel maximalizáljuk. Bizonyos implementációk ebben a függvényben vizsgálják a q_{new} konfiguráció ütközésmentességét, a későbbiekben bemutatott algoritmus során látni fogjuk, hogy előnyösebb az ütközésvizsgálatot az utolsó lépésre hagyni.

A NoCollision függvény igaz értékkel tér vissza, ha a q_{new} konfiguráció ütközésmentes állapotban van. Ebben az esetben egy-egy új csúccsal és éllel bővítjük a keresési fát.

Mindaddig folytatjuk ezt a ciklust, amíg elég közel nem kerülünk a q_{goal} célkonfigurációhoz vagy elértünk egy maximális iterációs számot. Ezek a feltételeket a StopCondition függvény vizsgálja.

Több változata is létezik ennek az algoritmusnak, melyek egy adott probléma esetén hatékonyabb, illetve jobb minőségű megoldást adhatnak, mint a tradicionális algoritmus.

A hagyományos RRT algoritmus hátránya a grid-alapú pályatervezési (például A* és D*) algoritmusokkal szemben, hogy a talált út koránt sem lesz optimális, a pályatervezés során nem vesszük figyelembe az esetleges akadályok minél nagyobb ívben történő elkerülését, illetve a megtalált út nem a lehető legrövidebb út lesz.

Az utóbbi problémára kínál megoldást az RRT* algoritmus, mely annyiban különbözik az eredeti módszertől, hogy minden egyes iterációs ciklusban megvizsgáljuk, hogy lehetséges-e az újonnan beszúrt pont környezetében optimalizálni a keresési fa struktúráján [19]. Ennek a módszernek az eredménye látható a 4.2. ábrán. A pályatervezés során a zölddel jelölt kezdeti konfigurációból kell eljutnia a robotnak valamelyik pirossal jelölt célkonfigurációba. Ez utóbbiból azért nem csak egy van, mivel egy valós térbeli pozíciót egy kétszabadságfokú robot több konfigurációban is el tud érni, gondoljunk például arra, hogy az első csukló 360°-os elforgatásával ugyanabba a pozícióba jutunk. A robotkar valós térben lévő mozgását a módszerrel talált útvonal mentén az alábbi videó illusztrálja: `https://youtu.be/2siB_wANDBU`. A végső megoldásomban nem ezt a módszert használtam, mivel a folyamatos optimalizálással jelentősen megnő a algoritmus számítási igénye.

Az RRT$^+$ algoritmus szintén egy továbbfejlesztése az eredeti metódusnak, ez kifejezetten magas dimenziójú keresési terek esetén alkalmazható nagy hatásfokkal, ilyen például a hiperredundáns robotkarok konfigurációs tere [40], melyek szabadságfokainak száma akár tízes-százas nagyságrendbe is eshet. Például az emberi testnek körülbelül 244 szabadsági foka van [20], ha egy ilyen komplexitású rendszernek szeretnénk pályát tervezni, arra alkalmazhatnánk ezt a módszert.

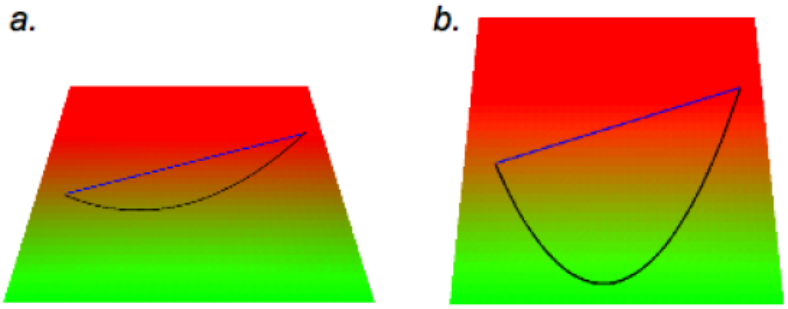
A feladatom során viszont hagyományos manipulátorok pályatervezésére kerestem megoldást, így nem volt szükség a hiperredundanciából fakadó extrém dimenzionövekedés okozta problémák kiküszöbölésére. Ennek ellenére elég nagy dimenzió térben dolgoztam ahhoz, hogy a hagyományos grid-alapú pályatervezési metódusok ne jöjjenek számításba.

Végül a T-RRT algoritmus mellett döntöttem, mely képes magas dimenziójú terek esetén is viszonylag kis számításigénnyel valamely általunk meghatározott költségdefiníció szerinti jó minőségű utat találni.

Algorithm 2: Transition-based RRT

Input: the configuration space $\mathcal{C}$
the cost function $c : \mathcal{C} \mapsto \mathbb{R}^*_+$
the root q_{init} and the goal q_{goal}
Output: the tree $\mathcal{T}$

1: $\mathcal{T} \leftarrow$ InitTree(q_{init})
2: **while not** StopCondition($\mathcal{T}, q_{goal}$) **do**
3: $q_{rand} \leftarrow$ SampleConf($\mathcal{C}$)
4: $q_{near} \leftarrow$ NearestNeighbor($q_{rand}, \mathcal{T}$)
5: $q_{new} \leftarrow$ Extend($\mathcal{T}, q_{rand}, q_{near}$)
6: **if** $q_{new} \neq NULL$ **then**
7: $d_{near-new} \leftarrow$ Distance(q_{near}, q_{new})
8: **if** TransitionTest($c(q_{near}), c(q_{new}), d_{near-new}$) **and** MinExpandControl($\mathcal{T}, q_{near}, q_{rand}$) **and** NoCollision($q_{new}$) **then**
9: AddNewNode($\mathcal{T}, q_{new}$)
10: AddNewEdge($\mathcal{T}, q_{near}, q_{new}$)
11: **end if**
12: **end if**
13: **end while**
14: **return** $\mathcal{T}$



4.3. ábra. Konstans meredekséggel rendelkező költségfüggvények, a második ábrán nagyobb a meredekség. A kékkel jelölt utak a mechanikai munka, míg feketével jelölt utak a út menti teljes költség optimalizálása során kapott eredmény. [17]

4.1.2. Transition-based RRT

A T-RRT algoritmus alapvetése, hogy próbálja egyesíteni mind a sztochasztikus keresési metódusok, mind a rácsozás alapú hagyományos pályatervezési algoritmusok előnyeit. Azaz a T-RRT képes magas dimenziójú terek esetén is jó minőségű utat találni, ahol az út jóságát a konfigurációs téren értelmezett költségfüggvénnyel mérhetjük [17].

A módszer működését a 2. pszeudokód írja le. Látható, hogy az algoritmus legnagyobb része megegyezik a 4.1.1. alfejezetben leírtakkal, az eltéréseket pirossal jelöltem. A fő különbség az, hogy ez az algoritmus bemenetként vár egy költségfüggvényt, illetve két új feltétellel bővült, hogy egy új csúcsot és élt adjunk hozzá a keresési gráfhoz. Ezen feltételek a TransitionTest és MinExpandControl függvények, melyek bővebb leírása a 4.1.3. és a 4.1.4. alfejezetekben található.

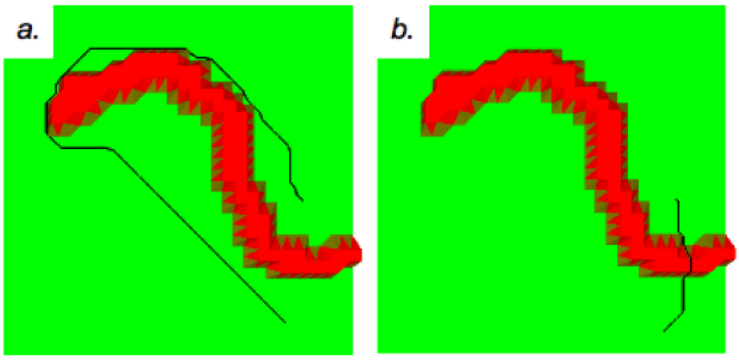
Mivel az algoritmus biztosítja a megtalált út minőségét az adott költségfüggvény szerint, ezért mindenképpen beszélnünk kell arról is, hogy mit is jelent egy adott út költsége. Több módszer is létezik a költség számítására, ilyenek például az út menti maximális-, átlag- vagy teljes költség. Ezek közül a maximális költség használható a legkevésbé, hiszen az csak az út egyetlen pontját veszi figyelembe a költségszámításkor.

Az átlagköltség szintén nem egy kielégítő megoldás, hiszen nem veszi figyelembe a pálya hosszát. Így előfordulhat olyan eset, hogy van egy elkerülhetetlen, magasabb költséggel rendelkező területe a költségfüggvénynek, akkor ezzel a módszerrel egy hosszabb út tévesen alacsonyabb költséget kaphat, ha sok időt tölt a függvény alacsonyabb költségű helyein, mint az az út, amely egyenesen áthalad a nagyobb költséggel rendelkező szakaszon.

Tehát első meggondolásként a költségfüggvény út menti integrálja tűnhet az ideális kritériumnak.

Viszont létezik egy másik megoldás erre a problémára, ahol a költségfüggvény mentén való mozgást valódi mozgásnak tekintjük, azaz a költség pozitív irányú megváltozását egy erőként értelmezzük, amely akadályozza a mozgást, így mechanikai munkát végez. Ekkor a pálya bejárásakor elhasznált energiát használhatjuk a megtalált út minőségének mérésére. Az implementációmban a negatív irányú költségváltozás nem okoz energiavesztést.

Ennek a módszernek a létjogosultságának igazolása céljából vizsgáljuk meg az alábbi két esetet. Az első esetben a költségfüggvény egy konstans meredekséggel rendelkező lejtő (4.3. ábrán látható). Ekkor a mechanikai munkát optimalizálva a triviális egyenes utat



4.4. ábra. Sík költségfüggvény, nagy költségű akadállyal. A bal oldali ábrán látható mechanikai munka, míg a jobb oldali ábrán költség út menti integrálja minimalizálásával kapott eredmény. [17]

kapjuk, míg az költségfüggvény út menti integrálját minimalizálva a kapott pálya függ a lejtő meredekségétől.

A második esetben egy sík költségfüggvényt vizsgálunk egy magas költségű akadállyal (4.4. ábrán látható). A minimális munkával járó pályának azt a lehető legrövidebb utat kapjuk, amely megkerüli ezt az akadályt, míg a teljes költség optimalizálásakor a kapott pálya keresztezi azt, amely nem minden esetben engedhető meg.

4.1.3. Transition-test függvény

Ahogy azt már az előzőekben tárgyaltuk, a tradicionális RRT két feltétellel lett kibővítve, amelyeknek teljesülniük kell, hogy egy új csúcs beszúrásra kerüljön a keresési fába. A q_{near} és q_{new} pontok közötti átmenet minőségét a TransitionTest függvényben vizsgáljuk. Ennek menete a 3. algoritmusban látható.

A függvény bemenetei c_i, c_j, melyek rendre a költségfüggvény q_{near} és q_{new} konfigurációban felvett értékei, illetve d_{ij}, mely a két pont euklideszi távolsága a konfigurációs térben. Kimenetként egy logikai értéket adunk vissza, amely jelzi az átmenet minőségének megfelelőségét.

Első lépésként a GetCurrentNFail függvény segítségével lekérdezzük, hogy hányszor volt sikertelen az előzőekben az átmenet elfogadása.

Ezt követően maximalizáljuk az újonnan hozzáadható konfigurációban a költségfüggvény lehetséges értékét, azaz ha a c_j bemenet értéke magasabb egy c_{max} paraméter értékénél, akkor automatikusan hamis értékkel térünk vissza. Így elkerülhetjük azokat a konfigurációkat, amelyek túl nagy költséggel járnának. A költség negatív irányú megváltozása esetén az adott átmenet automatikusan el lesz fogadva.

Ha viszont a költség megváltozása pozitív, akkor az átmenet minőségének tesztje egy, a Metropolis-kritériumon [33] alapuló valószínűségi változó szerint lesz elfogadva. Ezt a valószínűséget az alábbi módon definiálhatjuk:

$$p_{ij} = \exp(-\frac{\Delta c_{ij}}{K \cdot T}), \tag{4.1}$$

ahol $\Delta c_{ij} = \frac{c_j - c_i}{d_{ij}}$ a költségfüggvény meredeksége, K egy normalizáló konstans, a mintavételezett konfigurációk átlagköltségét adtam értékül neki az implementáció során. A T paramétert hőmérsékletnek nevezzük, ami az adott átmenet sikerességének nehézségét jellemzi. Ha a hőmérséklet magas, akkor nagyobb eséllyel lesz egy pozitív költségnövekedéssel

járó átmenet elfogadva a TransitionTest függvényben. T-t egy nagyon alacsony értékkel kell inicializálni (körülbelül $10^{-4} - 10^{-3}$-os nagyságrendbe) azért, hogy az algoritmus futása elején csak nagyon kicsi pozitív költségnövekedést engedjünk meg a pálya mentén. Ez a módszer ugyanazon az elven működik, mint a Szimulált lehűtés algoritmusa [33].

A Rand$(0, 1)$ függvény egy egyenletes eloszlás szerinti véletlen számot ad 0 és 1 között.

Az $\alpha > 1$ paraméter segítségével adaptívan hangolhatjuk a hőmérsékletet. Ha az adott átmenet sikeres volt, akkor α-val osztva csökkentjük T értékét. Abban az esetben, ha $nFail_{max}$-szor nem sikerült az átmenet, akkor növeljük a hőmérséklet értékét. Ez azt jelenti, ha az $nFail_{max}$ paraméter értékét nagyra állítjuk, abban esetben a keresési fa nagyobb eséllyel fog a költségfüggvény "völgyeiben" haladni, viszont jóval nehezebb lesz útvonalat találni egy magasabb költséggel rendelkező "dombon" keresztül.

Algorithm 3: TransitionTest(c_i, c_j, d_{ij})

```
1: nFail = GetCurrentNFail();
2: if c_j > c_max then
3:    return False
4: end if
5: if c_j < c_i then
6:    return True
7: end if
8: p = exp(-(c_j - c_i)/d_ij / (K·T))
9: if Rand(0, 1) < p then
10:   T = T/α
11:   nFail = 0
12:   return True
13: else
14:   if nFail > nFail_max then
15:      T = T · α
16:      nFail = 0
17:   else
18:      nFail = nFail + 1
19:   end if
20:   return False
21: end if
```

4.1.4. Minimális feltérképezés

A TransitionTest függvénynek van egy lehetséges mellékhatása, mely szerint képes lelassítani a keresési tér magasabb költséggel rendelkező területeinek felfedezését. Ekkor az újonnan hozzáadott pontok csak a már ismert terület információit finomítják.

Ez a hatás elkerülhető, ha előírunk egy minimális felderítési rátát. Azaz egy új csúcsot, amely közel helyezkedik el a fában lévő más pontokhoz, csak akkor szúrunk be a keresési gráfba, ha azon pontok aránya, melyek az ismert terület finomítását szolgálják, kisebb, mint egy adott ρ paraméter. Azon pontok, melyek új területek felfedezését teszik lehetővé automatikusan elfogadásra kerülnek.

Ez a módszer a 4. algoritmusban látható. Az UpdateNbNodeTree függvény segítségével számon tartjuk, hogy eddig hány csúcs lett sikeresen elfogadva a T-RRT algoritmus két feltétele által, míg az UpdateNbRefineNodeTree függvény egy olyan változó értékét növeli, mely számon tartja, hogy ezen pontok közül mennyi volt a már ismert tér finomítására szolgáló csúcs.

Egy adott q_{new} konfiguráció típusát a q_{rand} q_{near} ponttól való euklideszi távolsága határozza meg. Tehát nem maga a q_{new} konfigurációt vesszük alapul, mivel annak q_{near}-től való távolságát az EXTEND függvényben maximalizáltuk. Ha az előbb definiált távolság nagyobb, mint a δ lépésköz, abban az esetben úgy tekintjük, hogy az adott csúcs elősegíti az új területek feltérképezését, ellenkező esetben csak finomítja a már ismert területeket.

Ezzel a kiegészítéssel hatékonyan fogjuk felderíteni a teljes konfigurációs teret úgy, hogy végig a költségfüggvény völgyeiben haladunk, a TRANSITIONTEST függvénynek köszönhetően.

Algorithm 4: MinExpandControl($\mathcal{T}, q_{near}, q_{rand}$)

```
1: if Distance(q_near, q_rand) > δ then
2:   UpdateNbNodeTree(T)
3:   return True
4: else
5:   if NbRefineNodeTree(T+1) / NbNodeTree(T+1) > ρ then
6:     return False
7:   else
8:     UpdateNbRefineNodeTree(T)
9:     UpdateNbNodeTree(T)
10:    return True
11:  end if
12: end if
```

4.2. Ütközésdetektálás

Az ütközésdetektálási algoritmus, amelyre az 1. algoritmus 8. lépésében, illetve a 2. algoritmus 6. lépésében volt szükség, kritikus fontosságú az algoritmus megfelelő működésének szempontjából. Nehéz feladat egy adott konfigurációról egzaktul eldönteni, hogy az abban való tartózkodás ütközéssel jár-e. Illetve nem csak a környező objektumokkal történő ütközéseket kell elkerülnünk, hanem a robotkar önmagával való ütközését is meg kell akadályoznunk.

Szakirodalomból ismert olyan módszer [22], mely képes megtalálni azt a trajektóriát, amely minimalizálja a mozgáshoz szükséges energiafelhasználást. Ez a módszer magas számítási költséggel rendelkezik, a bonyolult optimalizálási folyamat miatt. Ennek ellenére, az abban alkalmazott ütközésdetektálási algoritmust fel tudtam használni a feladatom során.

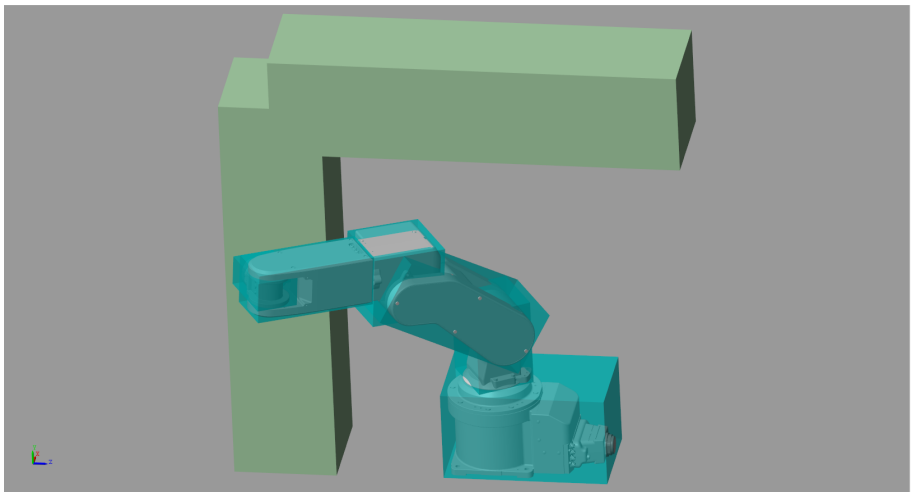
4.2.1. Ütközésdetektálás poliéderek segítségével

Az ütközésmentes konfigurációk meghatározásához egy közelítéssel élhetünk, mely szerint a robotkart és az azt körülvevő akadályokat modellezhetjük az őket körülhatároló konvex poliéderek segítségével. A 4.5. ábrán látható, ahogy ezt a Mitsubishi RV-2F-Q típusú robotkar esetén megtettem.

Egy adott féltér egy egyenlőtlenség és négy paraméter segítségével leírható:

$$a_x x + a_y y + a_z z \leq b \tag{4.2}$$

Egy adott $\mathbf{y} = [x, y, z]^\top$ pont eleme a féltérnek, ha teljesül rá a fenti egyenlőtlenség. Az $\mathbf{a} = [a_x, a_y, a_z]^\top$ és a b paraméterek a félteret leíró paraméterek, melyek megegyeznek a félteret határoló sík egyenletében szereplő tényezőkkel.



4.5. ábra. Az ábrán látható, ahogyan a Mitsubishi RV-2F-Q robotkar egyes szegmenseit, illetve a manipulátort körülvevő akadályokat az őket körülhatároló konvex poliéderek segítségével modelleztem

Egy adott poliéder definiálható félterek metszeteként. Az $\mathbf{y}$ térbeli pont eleme a poliédernek, ha kielégíti az azt leíró félterek egyenlőtlenségeinek uniójából képzett egyenlőtlenség-rendszert.

Legyen P a robotkart leíró konvex poliéderek uniója:

$$P = \bigcup_{i=1}^{n_P} P^{(i)} = \{\mathbf{y} \in \mathbb{R}^3 | \mathbf{A}^{(i)}\mathbf{y} \leq \mathbf{b}^{(i)}\} \tag{4.3}$$

ahol n_P a P-ben található poliéderek száma. $\mathbf{A}^{(i)}$ és $\mathbf{b}^{(i)}$ a $P^{(i)}$ poliédert körülhatároló síkok egyenleteinek paraméterei. Legyen p_i a $P^{(i)}$-ben lévő síkok száma, ekkor $\mathbf{A}^{(i)} \in \mathbb{R}^{p_i \times 3}$ és $\mathbf{b}^{(i)} \in \mathbb{R}^{p_i}$, $\forall\ i = 1, \ldots, n_P$.

Hasonlóan definiálhatjuk az akadályok unióját, melyet a továbbiakban Q-val jelölünk:

$$Q = \bigcup_{j=1}^{n_Q} Q^{(j)} = \{\mathbf{y} \in \mathbb{R}^3 | \mathbf{C}^{(j)}\mathbf{y} \leq \mathbf{d}^{(j)}\} \tag{4.4}$$

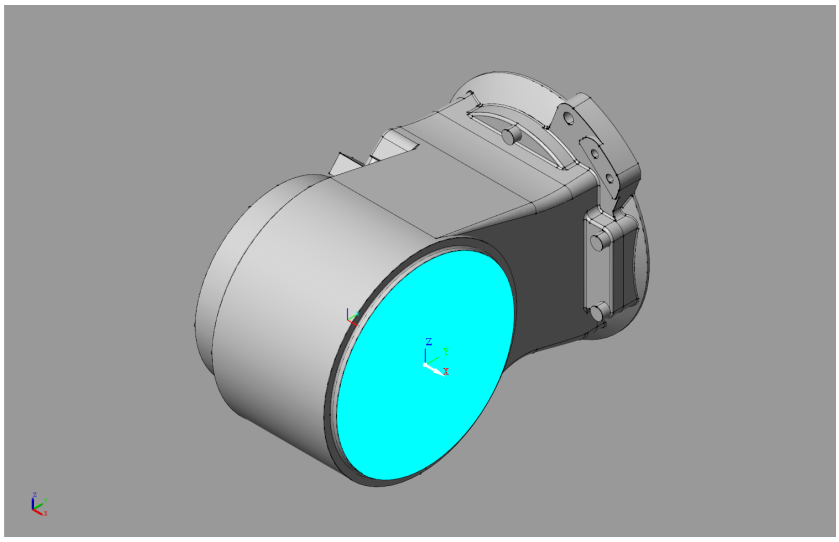
ahol n_Q a Q-ban található poliéderek száma. Amennyiben q_j a $Q^{(j)}$ poliédert leíró síkok száma, abban az esetben $\mathbf{C}^{(j)} \in \mathbb{R}^{q_j \times 3}$ és $\mathbf{d}^{(j)} \in \mathbb{R}^{q_j}$, $\forall\ j = 1, \ldots, n_Q$.

Legyen I olyan (k, l), $k \neq l$ indexpárok halmaza, ahol k és l a robotkar egy-egy poliéderét jelölik ki, amelyeket egymáshoz képest meg szeretnénk figyelni az önütközés vizsgálatakor. Belátható, hogy I nem tartalmazza az összes lehetséges (k, l) párt, mivel vannak a manipulátornak olyan szegmensei, amelyek fizikailag nem tudnak egymással ütközni (például szomszédos szegmensek esetén), így az ütközésvizsgálat ebben az esetben felesleges.

Tegyük fel, hogy Q adott, P-t pedig meghatároztuk egy adott konfigurációban. Ekkor az adott konfiguráció ütközésmentes, ha teljesül rájuk az alábbi két feltétel:

$$P^{(i)} \cap Q^{(j)} = \emptyset \quad \wedge \quad P^{(k)} \cap P^{(l)} = \emptyset \tag{4.5}$$

$\forall i = 1, \ldots, n_P$, $\forall j = 1, \ldots, n_Q$ és $\forall (k, l) \in I$ esetén. Azaz a P-ben és a Q-ban található poliédereknek a metszete üres halmaz. Ennek a feltételnek teljesülnie kell P azon poliédereire is, melyeket az I halmaz által az előzőekben kijelöltünk.



4.6. ábra. A robotkar második szegmenséhez tartozó referencia-koordiánatrendszer kiválasztása

Adott $P^{(i)}$ és $Q^{(j)}$ poliéderek nem ütköznek (hasonló módon az önütközés vizsgálatakor) abban az esetben, ha az általuk definiált egyenlőtlenség-rendszerek uniójának nincs megoldása. Azaz $\nexists \mathbf{y}^{(i,j)} \in \mathbb{R}^3$, hogy:

$$\begin{pmatrix} \mathbf{A}^{(i)} \\ \mathbf{C}^{(j)} \end{pmatrix} \mathbf{y}^{(i,j)} \leq \begin{pmatrix} \mathbf{b}^{(i)} \\ \mathbf{d}^{(j)} \end{pmatrix} \tag{4.6}$$

Egy lineáris egyenlőtlenség-rendszer megoldhatóságát a Farkas-lemma segítségével vizsgálhatjuk, mely kimondja, hogy egy lineáris rendszernek akkor és csakis akkor nem létezik megoldása, ha létezik $\mathbf{w}^{(i,j)} \in \mathbb{R}^{(p_i+q_j)}$ vektor, ami kielégíti az alábbi feltételeket:

$$\begin{aligned} \mathbf{w}^{(i,j)} \geq 0 \quad \text{és} \quad \begin{pmatrix} \mathbf{A}^{(i)} \\ \mathbf{C}^{(j)} \end{pmatrix}^{\top} \mathbf{w}^{(i,j)} = 0 \quad \text{és} \\ \begin{pmatrix} \mathbf{b}^{(i)} \\ \mathbf{d}^{(j)} \end{pmatrix}^{\top} \mathbf{w}^{(i,j)} < 0 \end{aligned} \tag{4.7}$$

4.2.2. Poliéderek paramétereinek meghatározása

A robotkarhoz tartozó poliéderek $\mathbf{A}^{(i)}$ és $\mathbf{b}^{(i)}$ paraméterei függnek a robot aktuális $\mathbf{q}$ konfigurációjától, ezért ezeket minden esetben újra meg kell határoznunk.

Definiáljuk az adott $P^{(i)}$ poliéderhez tartozó paramétereket egy adott szegmenshez rögzített koordináta-rendszerben. Ezzel a megoldással időben állandó paramétereket kapunk egy adott referenciakeretből tekintve, melyeket át kell transzformálnunk a bázis koordináta-rendszerbe minden egyes konfiguráció esetén. Jelöljük ezeket a paramétereket $\mathbf{A}_r^{(i)}$-vel és $\mathbf{b}_r^{(i)}$-vel.

A poliéderek kiválasztása esetén arra törekedtem, hogy a robotkar egyes szegmenseit egyetlen poliéderrel írjam le, mégpedig az azokat körülhatároló téglatestekkel. A referenciakeretet úgy választottam meg, hogy mindegyik koordinátatengely merőleges legyen a definiálni kívánt téglatest valamely oldalára. Mindezt úgy kell megtenni, hogy a keret origójától tekintve a befoglaló téglatest paraméterei könnyen mérhetőek legyenek a robotkar CAD modellje segítségével. Egy ilyen keret definiálása látható a 4.6. ábrán, a ro-

botkar második szegmense esetére. Ezzel a módszerrel elértem azt, hogy az összes téglatest $\mathbf{A}_r^{(i)}$ paramétermátrixa ugyanúgy néz ki, csupán a méréseket kell végeznünk a $\mathbf{b}_r^{(i)}$ vektor meghatározásához. A méréseket az Autodesk Inventor Professional 2017 környezetben végeztem el.

$$\mathbf{A}_r^{(i)} = \begin{bmatrix} 1 & 0 & 0 \\ -1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & -1 \end{bmatrix}, \quad \mathbf{b}_r^{(i)} = \begin{bmatrix} b_{x_+} \\ b_{x_-} \\ b_{y_+} \\ b_{y_-} \\ b_{z_+} \\ b_{z_-} \end{bmatrix} \tag{4.8}$$

ahol b_{x_+} a téglatest x tengelyre merőleges, a referenciakeret origójától pozitív x irányba elhelyezkedő oldalának távolsága, míg b_{x_-} a negatív irányba elhelyezkedő oldal távolsága, a többi paraméter ezzel analóg módon adható meg a másik két koordinátatengelyre.

Ezt követően az így kapott paramétereket át kell transzformálnunk a bázis koordináta-rendszerbe. Legyen $\mathbf{T}^{(i)}(\mathbf{q})$ a bázis és a $P^{(i)}$-hez tartozó referencia koordináta-rendszer közötti eredő transzformációs mátrix egy adott $\mathbf{q}$ konfigurációban.

Ekkor a bázis koordináta-rendszerben értelmezett paraméterek az alábbi módon számíthatók:

$$\begin{bmatrix} \mathbf{A}^{(i)} & \mathbf{b}^{(i)} \end{bmatrix} = \begin{bmatrix} \mathbf{A}_r^{(i)} & \mathbf{b}_r^{(i)} \end{bmatrix} \begin{bmatrix} \mathbf{T}^{(i)}(\mathbf{q}) \end{bmatrix}^{-1} \tag{4.9}$$

Az előzőekben ismertetett ütközésdetektálási algoritmust ezekkel a paraméterekkel kell kiértékelni.

4.3. Költségfüggvény becslése

A 4.1. fejezetben ismertetett módszerhez szükséges költségfüggvény számításakor a költség jelentésének definiálása fontos feladat, mely általában valamilyen heurisztikára épül. Például egy humán-kollaboratív robot esetén biztonsági meggondolások alapján úgy választhatjuk meg a költségfüggvényt, hogy az az ember közelében (különös tekintettel a fejre) magas értéket vegyen fel [41].

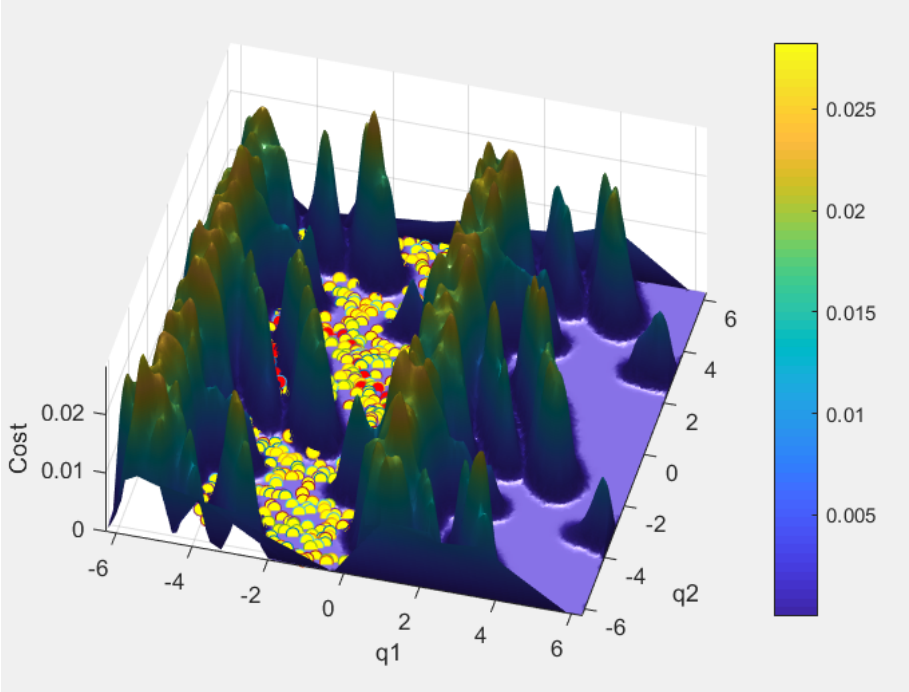
Egy másik megközelítés lehet, az idő, illetve trajektória mentén végzett munka minimalizálása a költség definiálásakor. Ehhez a módszerhez pontos információkkal kell rendelkeznünk a rendszer dinamikájáról.

Az én megoldásomban a költségfüggvény definiálásának célja az volt, hogy mozgás közben minél nagyobb távolságot tartson a robotkar a környező akadályoktól. Ennek a függvénynek az egzakt leírása nem lehetséges, ezért valamilyen approximációs módszert kell alkalmaznunk a költség becsléséhez.

4.3.1. Költségfüggvény becslése Gauss-függvényekkel

Első megoldásként azt az ötletet valósítottam meg, hogy elegendő mintát veszek a konfigurációs térből, majd minden egyes pontban a költségfüggvény értékét Gauss-függvények összege fogja adni. Ezen Gauss-függvények középpontjai azon mintavételi pontok lesznek, melyek ütközéssel járó konfigurációk voltak. Az ütközéssel járó konfiguráció megállapítására a 4.2. fejezetben már bemutatott módszert használtam.

Ennek a módszernek több hátránya is van. Ahogy a 4.7. ábrán is látható, a kapott függvény nagyon hullámos, továbbá a költségfüggvény maximális értékét nem lehet előre definiálni, így a 3. pszeudokódban leírt TransitionTest függvényben lévő c_{max} paramétert nehéz előre meghatározni. További skálázással ezt a problémát meg lehetne oldani, viszont ez növelné az algoritmus számítási igényét.



4.7. ábra. Költségfüggvény approximációja egy kétszabadságfokú robotkar esetén Gauss-függvények segítségével. Az ábrán látható a T-RRT algoritmus által talált keresési fa is.

Ezen kívül a költségfüggvényt minden egyes mintavételi pontban el kell tárolnunk, így az algoritmus memóriaigénye is jelentős, illetve bármely más pontban a közeli mintapontokból számított interpoláció lassú lenne.

4.3.2. A Fuzzy-függvényapproximáció alapjai

Egy alternatív megoldás lehet a fuzzy approximátor-rendszerek használata.

A fuzzy következtető-rendszerek különböző területen alkalmazhatóak, mint például az irányításelmélet [36], rendszer identifikáció [1], döntéshozás [16], stb... Ezenkívül szintén jól használhatóak nemlineáris függvények becslésére, amennyiben elegendő $y_j = f(x_j)$ tanítási ponttal rendelkezünk, ahol f a becsülni kívánt nemlineáris függvény, y_j az f függvény értéke az x_j konfigurációban, x_j pedig a j-dik tanítási pont helye a konfigurációs térben.

A fuzzy rendszerek előnye az approximációs feladatokban, hogy a fuzzy becslőt leíró szabályok könnyen hangolhatóak. A számítási kapacitás szintén töredéke a előzőekben bemutatott Gauss-függvények összegéből kapott megoldásnak.

A leginkább használt módszerek a fuzzy következtető rendszerekben az alábbiak [35]: Singleton fuzzyfikációt alkalmazunk, azaz a bemeneti jelek éles értékét használjuk az egyes szabályok adott bemeneti értékre adott tüzelési értékének meghatározására. Ezt követően szorzást használunk, mint fuzzy következtetés művelete, azaz az egyes bemeneti értékekre adott tüzelési értékek szorzata lesz az adott szabályra kapott eredő tüzelési érték. Továbbá előszeretettel alkalmazott modell a nulladrendű Sugeno-modell alapú fuzzy rendszer Gauss tagfüggvényekkel, amelynek jellemzője, hogy az egyes szabályok kimenetei konstansok, azaz éles értékek. Ezen kívül súlypont alapú defuzzyfikáció alkalmaztam, ami azt jelenti, hogy a rendszer kimenete a Sugeno-rendszer éles kimeneteinek a tüzelési értékekkel súlyozott átlaga lesz. Az így kapott következtetési folyamat látható a 4.8. ábrán.

4.8. ábra. A MATLAB Fuzzy Logic toolboxának segítségével ábrázolható a fuzzy következtetőrendszer működése

Ezzel a módszerrel az $f(x)$ nemlineáris függvény approximációja az alábbi módon számítható:

$$\hat{f}(x) = \frac{\sum_{l=0}^{M} \bar{y}^l \prod_{i=1}^{n} \exp(-(\frac{x_i - \bar{x}_i}{\sigma_i^l})^2)}{\sum_{l=0}^{M} \prod_{i=1}^{n} \exp(-(\frac{x_i - \bar{x}_i}{\sigma_i^l})^2)} \tag{4.10}$$

ahol n a konfigurációs tér dimenziója, $\bar{y}^l$ az éles kimeneti értéke a Sugeno-rendszer l-dik relációjának, $\bar{x}_i$ a Gauss tagsági függvény középpontja, illetve σ_i^l segítségével módosíthatjuk a tagsági függvény szélességét. Az $\bar{y}^l$, $\bar{x}_i$ és σ_i^l paraméterek adaptívan hangolhatóak.

Az univerzális approximáció tétele szerint [35], bármely kompakt $U \in \mathbb{R}^n$ halmazon értelmezett valós értékű folytonos f függvény és $\epsilon > 0$ esetén létezik $\hat{f}$ a (4.10)-ben leírt formában, hogy:

$$\sup_{x \in U} |f(x) - \hat{f}(x)| < \epsilon \tag{4.11}$$

azaz létezik approximáció, mellyel végtelenül meg tudjuk közelíteni az eredeti nemlineáris függvényt.

A továbbiakban bemutatok két algoritmust, melyekkel adaptívan hangolhatjuk a fuzzy következtető paramétereit. Ezen módszerekben feltesszük, hogy a tanítópontok be- és kimeneti értékeinek értelmezési tartománya véges.

4.3.3. Fuzzy-szabályrendszer generálása tanítópéldák alapján

Elsőként egy olyan algoritmussal szerettem volna megoldani az adaptív tanítást, amelyben egy előre meghatározott fuzzy szabálybázisból választjuk ki azokat a szabályokat a tanítópéldák alapján, melyek a lehető legjobban leírják a becsülni kívánt függvényt [37].

Az algoritmus első lépéseként véges számú intervallumra osztjuk a bemenetek és a kimenet értelmezési tartományát (ezt megtehetjük, mivel feltettük, hogy ezek véges tartományba esnek). Ezt követően minden egyes intervallumban definiálunk egy-egy Gauss tagfüggvényt, egymástól egyenlő távolságra elhelyezve. Ebben az algoritmusban a kimeneti tagsági függvényeket is Gauss függvényeknek értelmezzük.

A szabályokat az alábbi általános formában írhatjuk le a k-dik tanítási lépésben:

$$R^k : \mathsf{IF}\ x_1\ \mathsf{IS}\ F_1^k\ \mathsf{AND}\ x_2\ \mathsf{IS}\ F_2^k\ \mathsf{AND}\ \ldots\ \mathsf{AND}\ x_n\ \mathsf{IS}\ F_n^k\ \mathsf{THEN}\ y\ \mathsf{IS}\ G^k$$

ahol F_i^k az i-dik bemeneti változóhoz tartozó valamely tagsági függvény, G^k pedig a kimeneti változó egyik tagfüggvénye.

A bemeneti tagsági függvények száma egyértelműen definiálja a szabályok számát, mégpedig az egyes bemeneti változókon definiált tagfüggvények számának szorzataként.

Minden egyes tanítópontra ki kell választanunk azt a szabályt, amelynek a tüzelési értéke a legnagyobb az adott pont esetén, illetve azt a kimeneti tagfüggvényt kell választanunk, mely a legjobban leírja az adott tanítópár várt kimenetét.

A k-dik lépésben azt a szabályt kell kiválasztanunk, mely az adott (x^k, y^k) tanítópár esetén maximalizálja az alábbi összefüggést:

$$D(R^k) = \prod_{i=1}^{n} (\mu_{F_i^k}(x_i^k)) \cdot \mu_{G^k}(y^k) \tag{4.12}$$

ahol μ az adott tagfüggvény tüzelési értéke az aktuális tanítási pont esetén.

Akkor merülhet fel probléma, ha egy tanítópont esetén egy olyan szabály lesz az optimális, melyet már egyszer felvettünk, viszont más kimeneti tagfüggénnyel. Ekkor két megoldás közül választhatunk, az egyik szerint egyszerűen azt a szabályt tartjuk meg, melyre (4.12) nagyobb értéket ad. Ezenkívül vehetjük a két kimeneti tagfüggvény súlyozott átlagát is eredő tagsági függvényként.

Az esetemben ennek a megoldásnak van egy hátránya, mégpedig az, hogy (4.12)-t $N = N_1 \cdot N_2 \cdot \ldots \cdot N_n$-szor ki kell értékelnünk (ahol N_j a j-dik bemeneti változó tagfüggvények száma, n pedig a konfigurációs tér dimenziója). Azaz a bemenetek számának növekedésével a számítási idő exponenciálisan növekszik. Így ez a módszer már egy hat-szabadságfokú robotkar esetén is nehezen alkalmazható.

4.3.4. Legközelebbi szomszéd alapú klaszterezés

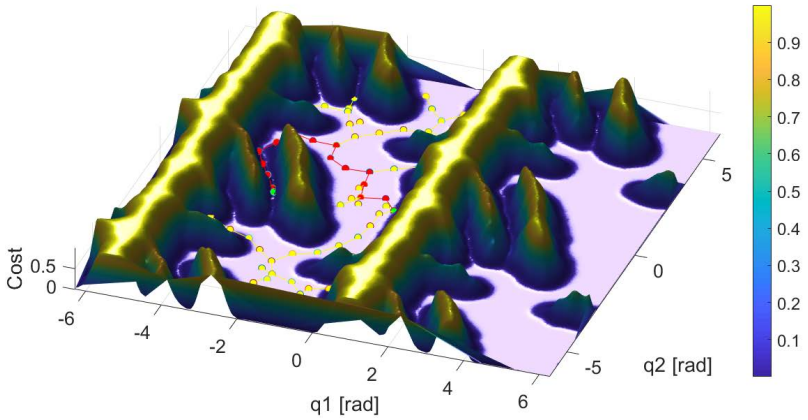
A fuzzy következtető hangolására egy másik elterjedt módszer a legközelebbi szomszéd alapú klaszterezés.

Ebben az algoritmusban a hangoláshoz használt tanítópontokat a térben elfoglalt pozíciójuk alapján klaszterekbe soroljuk. Egy adott osztályt három paraméterrel írhatunk le: x_0^l az l-dik klaszter középpontja, B^l az ebben az osztályban lévő pontok száma, illetve A^l ezen pontok értékének összege [34].

A tanítás folyamatát az 5. pszeudokód írja le. A klaszterek halmazát jelöljük $\mathcal{C}$-vel, melyet üres halmazként inicializálunk a tanítás legelején. $(\mathbf{X}_{train}, \mathbf{y}_{train})$ jelöli a tanítópárok halmazát. Egy (x, y) tanítópárt $(y = f(x))$ beszúrunk a hozzá legközelebbi klaszterbe, ha a klaszter x_0^l középpontjától vett távolsága kisebb egy adott r sugárnál. Ekkor az adott osztályhoz tartozó A^l és B^l értékeket frissítjük. Ellenkező esetben, ha a pont távolabb van minden eddig ismert osztálytól, akkor egy újat definiálunk az alábbi paraméterekkel: $\{x_0^l = x;\ A^l = y;\ B^l = 1\}$.

Algorithm 5: Legközelebbi szomszéd alapú klaszterezés

```
1:  C ← {}
2:  for each (x, y) ∈ (X_train, y_train) do
3:      l = GetClosestCluster(C, x)
4:      if C ≠ {} and Distance(x, x_0^l) < r then
5:          A^l = A^l + y
6:          B^l = B^l + 1
7:      else
8:          C ← AddNewCluster(x_0 ← x,  A ← y,  B ← 1)
9:      end if
10: end for
```



4.9. ábra. A Transition-based RRT algoritmus képes egy megadott költségfüggvény alapján alacsony költséggel rendelkező útvonal megtalálásra. Az ábrán látható függvény egy kétszabadságfokú robotkarra lett meghatározva a legközelebbi szomszéd alapú klaszterezés módszerével. A vízszintes sík a manipulátor konfigurációs terét ábrázolja, míg a függőleges tengelyen a költség alakulása látható.

A tanítást követően már tetszőleges x konfiguráció esetén meg tudjuk határozni a függvény becsült értékét:

$$\hat{f}(x) = \frac{\sum_{l=0}^{M} A^l \exp(-(\frac{x-\bar{x}_0^l}{\sigma^l})^2)}{\sum_{l=0}^{M} B^l \exp(-(\frac{x-\bar{x}_0^l}{\sigma^l})^2)} \tag{4.13}$$

ahol M a tanítás során létrehozott klaszterek száma. σ^l-nek az implementáció során konstans értéket adtam.

4.3.5. Költségfüggvény kiértékelése

Végül a legközelebbi szomszéd alapú klaszterezés algoritmusát használtam a T-RRT algoritmus költségfüggvényének meghatározására.

A magas dimenziójú konfigurációs térnek köszönhetően nem használhattam rácsot a tanítási pontok kijelöléséhez, hanem egyenletes eloszlást követve véletlenszerűen mintavételeztem a teret. Ezekben a pontokban értékeltem ki a 4.2. fejezetben ismertetett ütközésdetektálási algoritmust. Ha az x_j konfiguráció ütközéssel jár, abban az esetben 1 értéket adtam az x_j pontban a költségfüggvény y_j értékének, ellenkező esetben 0-t.

Ezt követően meghatároztam a függvény-approximátor paramétereit a legközelebbi szomszéd alapú klaszterezéssel. A 4.9. ábrán látható egy kétszabadságokú robotkarra elvégzett költségfüggvénybecslés, illetve pályatervezés példája (ennél magasabb dimenziójú költségfüggvények ábrázolása már nehezen megoldható).

A 4.1. táblázatban látható a Gauss-függvényekkel történő becslés, illetve a legközelebbi szomszéd alapú klaszterezés összehasonlítása. Az algoritmusokat egy Intel Core i7-7700HQ processzorral rendelkező gépen teszteltem a Mitsubishi RV-2F-Q robotkar esetére. A kiértékelési idő egyetlen ismeretlen konfigurációra történő becslés futási idejét jelenti. Látható, hogy mind a tanítási ideje, mind pedig a kiértékelési ideje a Gauss-függvényekkel történő approximációnak nagyon gyorsan növekszik a tanítópontok számának növelésével. A fuzzy becslés esetén ezek az értékek jóval lassabb ütemben növekednek. A klaszterek

Tanító pontok száma	Legközelebbi szomszéd alapú klaszterezés			Gauss	
	Tanítási idő	Klaszterek száma	Kiértékelési idő	Tanítási idő	Kiértékelési idő
1000	$0.0481s$	714	$22\mu s$	$0.0402s$	$7.8\mu s$
5000	$0.207s$	1649	$45\mu s$	$0.3027s$	$28.8\mu s$
10000	$0.4554s$	1974	$52\mu s$	$1.4151s$	$58.2\mu s$
50000	$3.0962s$	2431	$65\mu s$	$22.88s$	$137.9\mu s$
100000	$7.045s$	2521	$76\mu s$	$155.26s$	$1632\mu s$

4.1. táblázat. A fuzzy függvényapproximáció és a Gauss függvényekkel történő becslés összehasonlítása

használatával a kiértékelési idő nem növekszik számottevően, ami ideális a pályatervezési algoritmusban történő alkalmazáshoz.

4.4. Trajektória generálás

Az eddig bemutatott módszerek együtt képesek egy olyan pontsorozat megtalálására, melyen végighaladva eljutunk a robotkar kezdeti állapotából a célállapotba úgy, hogy elkerüljük az esetleges ütközéseket.

A konfigurációk közötti átmeneteknek a megvalósításához meg kell terveznünk a robotkar trajektóriáját is, azaz a csuklóváltozók (illetve azok első- és második deriváltjainak) kívánt időfüggvényeit is.

4.4.1. Trajektória generálás skalár változókra

A most bemutatott algoritmusban y jelöli a $\mathbf{q}$ konfiguráció egyetlen koordinátáját, azaz a robot egyetlen csuklóváltozóját.

Legyen $\{y_k\}$ a pályatervezési algoritmus által megtalált pontok halmaza, $\{t_k\}$ pedig ezen pontok elérésének kívánt abszolút ideje [2].

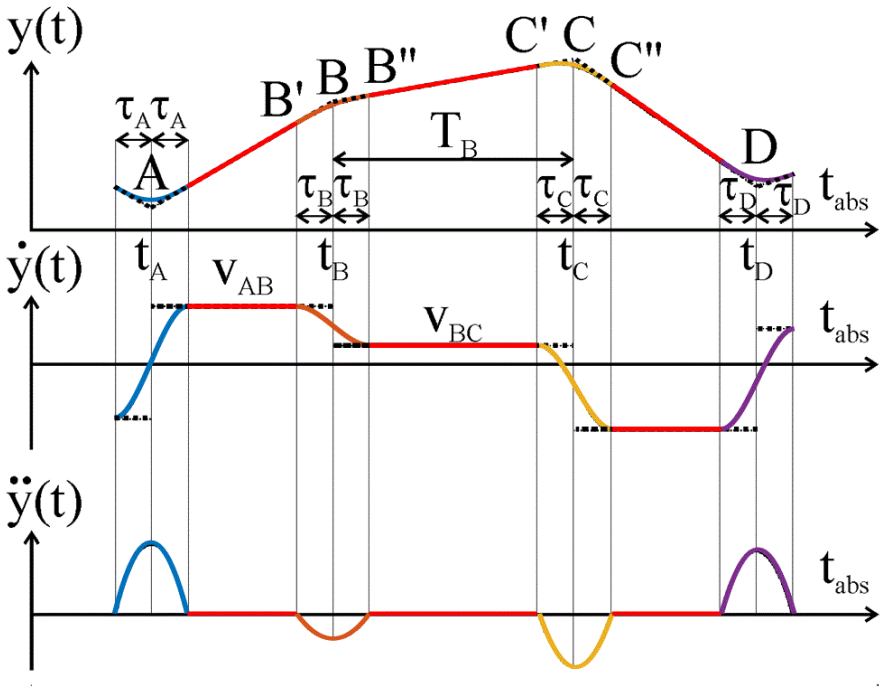
$$\begin{aligned} \{y_k\} &= \{\dots, A, B, C, D, \dots\}, \\ \{t_k\} &= \{\dots, t_A, t_B, t_C, t_D, \dots\} \end{aligned} \tag{4.14}$$

Mivel csuklókra kiadható nyomatékok csak véges tartományon belül mozoghatnak, ezért definiálnunk kell az adott csukló maximális gyorsulását: $|\ddot{y}|_{max}$.

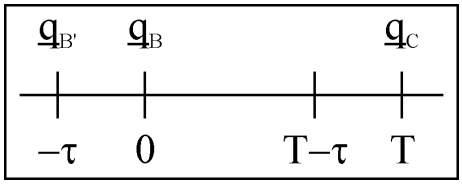
Első megoldásként legyen az $y(t)$ lineáris függvények sorozata, amelyek összekötik a $\{y_k\}$-ban található szomszédos pontokat. Ebben az esetben az első derivált $\dot{y}(t)$ konstans függvény lesz az egyes pontok között, majd ugrása lesz mindenhol ahol megváltozik $y(t)$ meredeksége. Ezeket a pontokat kivéve az $\ddot{y}(t)$ második derivált függvény értéke zérus lesz, míg a váltási pontokban Dirac-delta függvényeket fog tartalmazni, ami a valós rendszer esetén nem megvalósítható, mivel végtelen nagyságú beavatkozójelet igényelne. Ahhoz, hogy ezt a problémát eliminájuk, az $\ddot{y}(t)$ függvényt folytonosnak kell megválasztani.

A pontok közötti trajektória meghatározásához két fázist vezetünk be: egy utazó és egy gyorsító fázist. Az előbbit konstans sebesség és zérus gyorsulás jellemzi, míg az utóbbiban egy folytonosan változó gyorsulási függvényt alkalmazunk, hogy elérjük a következő utazó fázisban megkívánt sebesség értékét. A folytonosság másodfokú polinom segítségével biztosítható, így azt választjuk a második derivált alakjának (4.10. ábra).

Legyen B és C két szomszédos pont, míg B' és C' a kezdete, B'' és C'' pedig a vége a hozzájuk tartozó gyorsítási fázisnak. Ekkor bevezethetünk egy t relatív időt, melyet az



4.10. ábra. A gyorsulás folytonosságának biztosítása végett négyzetes függvényt választunk a gyorsulási fázisban. Az ábrán látható az így kapott kapott trajektória alakja.



4.11. ábra. Relatív időt vezetünk be, melyben a zérus időpillanat a $\mathbf{q}_B$ konfigurációban van. Ez a nullpont T értékkel eltolásra kerül mindig, amikor elértük a $T-\tau$ időpillanatot.

alábbi módon számolhatunk:

$$t = t_{abs} - t_B \in [-\tau_B, T_B], \tag{4.15}$$

ahol t_B a B pont elérésének abszolút ideje, τ_B a B-hez tartozó gyorsítási szakasz időintervallumának a fele. T_B pedig azt az időt jelenti, amennyi idő szükséges a C pont eléréséhez a B pontból. Ez látható a 4.10., illetve a 4.11. ábrán.

A másodfokú polinom integrálásával megkaphatjuk az $\dot{y}(t)$-hez, illetve az $y(t)$-hez tartozó függvények alakjait:

$$\begin{aligned} B'B'': \quad \ddot{y}(t) &= a_0 t^2 + a_1 t + a_2 \\ \dot{y}(t) &= a_0 \frac{t^3}{3} + a_1 \frac{t^2}{2} + a_2 t + a_3 \\ y(t) &= a_0 \frac{t^4}{12} + a_1 \frac{t^3}{6} + a_2 \frac{t^2}{2} + a_3 t + a_4 \end{aligned} \tag{4.16}$$

Látható, hogy öt paraméter ($a_0 \dots a_4$) meghatározásával megkaphatjuk a kívánt trajektóriát. Ehhez öt független feltételt kell definiálnunk:

$$\begin{aligned} \ddot{y}(-\tau_B) &= 0 \\ \ddot{y}(\tau_B) &= 0 \\ \dot{y}(-\tau_B) &= v_{B'B} = \frac{B - B'}{\tau_B} \\ \dot{y}(\tau_B) &= v_{BC} = \frac{C - B}{T_B} \\ y(\tau_B) &= B + v_{BC} \cdot \tau_B \end{aligned} \tag{4.17}$$

Így a következő lineáris egyenletrendszert írhatjuk fel:

$$\begin{bmatrix} \ddot{y}(-\tau_B) \\ \ddot{y}(\tau_B) \\ \dot{y}(-\tau_B) \\ \dot{y}(\tau_B) \\ y(\tau_B) \end{bmatrix} = \begin{bmatrix} \tau_B^2 & -\tau_B & 1 & 0 & 0 \\ \tau_B^2 & \tau_B & 1 & 0 & 0 \\ -\frac{\tau_B^3}{3} & \frac{\tau_B^2}{2} & -\tau_B & 1 & 0 \\ \frac{\tau_B^3}{3} & \frac{\tau_B^2}{2} & \tau_B & 1 & 0 \\ \frac{\tau_B^4}{12} & \frac{\tau_B^3}{6} & \frac{\tau_B^2}{2} & \tau_B & 1 \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \\ a_4 \\ a_5 \end{bmatrix} \tag{4.18}$$

$$\mathbf{y} = \mathbf{C} \cdot \mathbf{a}$$

Melynek a megoldása az alábbi módon számítható:

$$\mathbf{a} = \mathbf{C}^{-1} \cdot \mathbf{y} \tag{4.19}$$

Az utazó fázisban ($t \in [\tau_B, T_B - \tau_C)$) a skalár változó értéke lineáris függvény szerint változik:

$$y(t) = B + v_{BC} \cdot t \tag{4.20}$$

4.4.2. Trajektória generálása csuklóváltozókra

A skalár változókra bemutatott trajektória generálási módszer alkalmazható minden egyes csuklóváltozóra külön-külön, figyelembe véve azt a tényt, hogy a sebességek ($|\dot{\mathbf{q}}_i|_{max}$) és a gyorsulások ($|\ddot{\mathbf{q}}_i|_{max}$) határértékei az egyes csuklók esetén eltérhetnek.

Ezenkívül a $\mathbf{q}_{Ci}$ konfiguráció $\mathbf{q}_{Bi}$-ből való elérésének ideje is csuklóváltozónként eltérhet, ezért csak azt a csuklót kell maximális sebességgel mozgatnunk, melynek a legtöbb idő szükséges eljutni a következő állapotba. Így energiát takarítunk meg a mozgatás során ahhoz képest, hogyha minden egyes motort maximális sebességgel járatnánk.

Minden egyes csuklóváltozóra ugyanaz a τ érték használható, mely az alábbi módon számítható [2]:

$$\tau = \max_i \frac{3}{2} \frac{|\dot{\mathbf{q}}_i|_{max}}{|\ddot{\mathbf{q}}_i|_{max}} \tag{4.21}$$

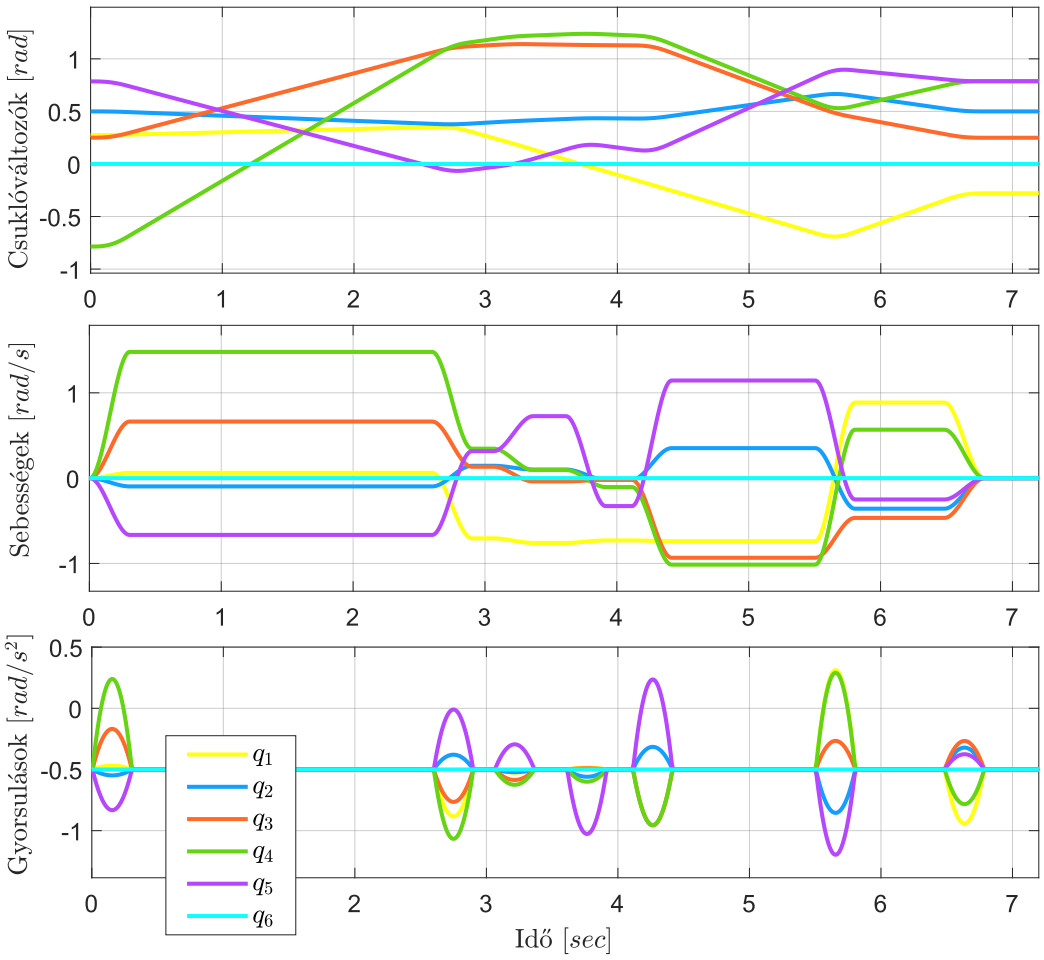
A trajektóriagenerálási algoritmust a 6. pszeudokód írja le.

A függvény bemenete Q a konfigurációk sorozata, mely a pályatervezés során talált utat tartalmazza.

A GetFirstConfiguration függvény a sor első elemét adja vissza, majd ez az elem törlésre is kerül sorból.

Induláskor a T változót a (4.21) összefüggésben számított τ értékkel inicializáltam, így az első iterációban a kezdeti állapot lesz a $\mathbf{q}_{B'}$ változó értéke.

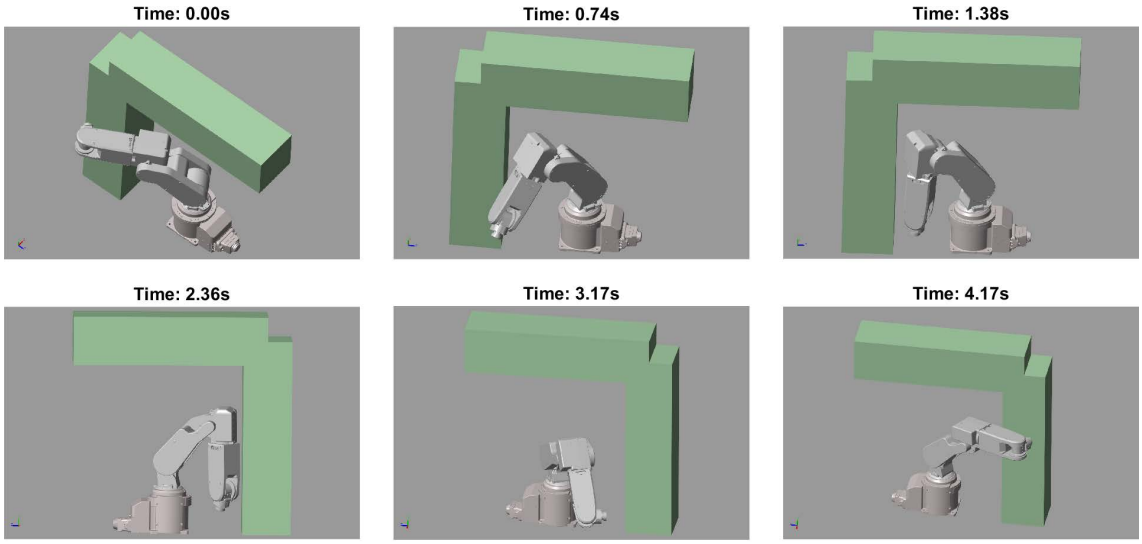
A 11. sorban található $\mathbf{C}$ mátrix, illetve $\mathbf{y}$ vektor meghatározásához a (4.17) és a (4.18) egyenletekben látható értékeket használtam.



4.12. ábra. A csuklóváltozókhoz tartozó trajektóriák

Algorithm 6: TrajectoryGeneration(Q, τ)

$\mathbf{q}_C$ =GetFirstConfiguration(Q)
$T = \tau$
while Size$(Q) > 0$ **do**
  $\mathbf{q}_{B'} = \mathbf{q}(T - \tau)$
  $\mathbf{q}_B = \mathbf{q}_C$
  $\mathbf{q}_C$ = GetFirstConfiguration(Q)
  $T_i = \frac{|\mathbf{q}_{Ci} - \mathbf{q}_{Bi}|}{|\dot{\mathbf{q}}_i|_{max}}$
  $T = \max\{\max_i\{T_i\}, 2\tau\}$
  $\mathbf{v}_{B'B} = \frac{\mathbf{q}_B - \mathbf{q}_{B'}}{\tau}$
  $\mathbf{v}_{BC} = \frac{\mathbf{q}_C - \mathbf{q}_B}{T}$
  $\mathbf{a} = \mathbf{C}^{-1} \cdot \mathbf{y}$
  $t = -\tau$
  while $t < T - \tau$ **do**
    $\mathbf{q}(t)$ = CalculateTrajectory$(\mathbf{a}, t)$
    $t = t + \Delta$
  end while
end while
return $\mathbf{q}(t)$



4.13. ábra. Az ábrán a manipulátor látható a szimuláció különböző időpontjaiban

A CalculateTrajectory függvényben a (4.16) egyenleteket értékeltem ki minden egyes csuklóváltozóra, tehát a $\mathbf{q}(t)$ változó dimenziója megegyezik a konfigurációs tér dimenziójával.

Az így kapott trajektória alakja a 4.12. ábrán látható.

4.5. Szimulációs eredmények

A 4.13. ábrán bemutatott példában a robotkar megkerül egy két poliéderrel modellezett akadályt. Az algoritmus egy Intel Core i7-7700HQ processzoron volt tesztelve.

Az alább található videón látható példában a legközelebbi szomszéd alapú klaszterezés algoritmusát használtam, mely $n = 20000$ tanítópontra lett kiértékelve, $r = 1\ rad$ klasztertávolsággal és $\sigma = 0.5$ értékkel a Gauss függvények szórás paraméterére. A tanítási idő ezekkel a paraméterekkel körülbelül $t \approx 120s$ volt, ami elfogadható, hiszen ezt csak egyszer kell kiértékelni egy adott akadálykonfiguráció esetén.

A T-RRT algoritmust az alábbi paraméterekkel futtattam: $T_{init} = 0.01$, $\alpha = 5$, $nFail_{Max} = 10$, $\rho = 0.1$, $c_{max} = 0.8$, $\delta = 0.4$. Így az átlagos futási idő $t \approx 7.3s$ volt.

Az alábbi videón látható a robotkar mozgása: `https://youtu.be/t6GO8LKG5js`.

Belátható, hogy egy újabb akadály hozzáadásával legrosszabb esetben n_p-szeresére nő a pályatervezés számítási ideje (ahol n_p a robotkart leíró poliéderek száma), mivel az ütközésdetektáló algoritmus minden egyes kiértékelése során ennyivel több poliéder-párra kell megvizsgálni az eredő egyenlőtlenség-rendszerük megoldhatóságát. Viszont a T-RRT algoritmus előnye, hogy az ütközésdetektálást csupán akkor végezzük el, ha az adott konfiguráció már minden előző feltételnek eleget tett. Ezzel a módszerrel a pályatervezés során nagyon sok pontot már előre kiszűrünk, melyek nem felelnek meg a minőségi elvárásoknak, így a legtöbb konfiguráció esetén nem végezzük el a legnagyobb számítási igénnyel rendelkező fázist, az ütközésdetektálást.

5. fejezet

Dinamikus objektumelkerülés

Dinamikusan változó környezet esetén fontos, hogy a robotkar közelében található objektumokról folyamatosan információt gyűjtsünk. Mindezt úgy kell megtenni, hogy az információ feldolgozása után legyen időnk valós időben beavatkozni a robotkar mozgásába, így kerülve ki az esetlegesen a robot útjába kerülő akadályokat.

Ebben a fejezetben egy olyan módszert mutatok be, mely először detektálja a robot környezetében található objektumokat, majd azok elkerülését virtuális taszító erők segítségével valósítja meg.

5.1. Virtuális mélységkép előállítása

Fontos feladat a robotkar körülötti objektumok, akadályok pozíciójának meghatározása, amelyet Kinect kamera (3.2.2. fejezet) mélységképének felhasználásával tettem meg. A kamerát úgy kell a térben elhelyezni, hogy a robot teljes munkaterét lefedje.

Az algoritmus során minden időpillanatban két darab mélységképpel dolgoztam. Az egyik a valós vagy szimulációban létrehozott kamera által szolgáltatott kép, mely tartalmazza mind a robotkart, mind az azt körülvevő objektumokat, míg a másik egy általunk generált virtuális mélységkép, melyen csak a robotkar látható [5]. Ezen képek az 5.1. ábrán láthatóak.

Az akadályok pozíciójának meghatározásnak az az alapvetése, hogy ha pontosan ismerjük a kamera belső és külső paramétereit (3.2. fejezet), illetve rendelkezünk a robotkart leíró 3D-s modellel, akkor ha a virtuális mélységképet kivonjuk a valós mélységképből, csak a látható akadályok maradnak.

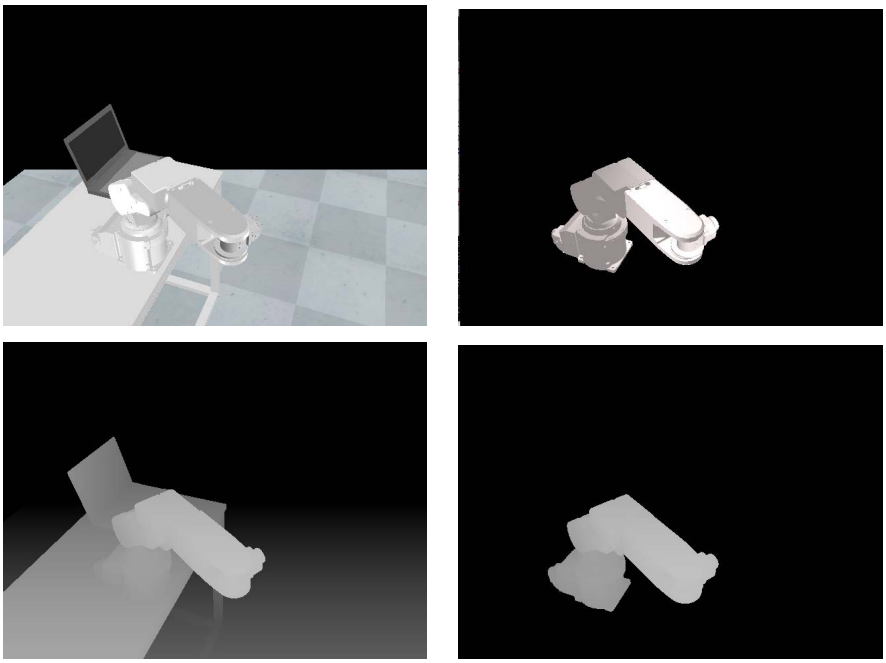
Első feladatként elő kell állítanunk a virtuális mélységképet. A robotkar egyes részegységeinek STL állományai interneten elérhetőek voltak. A teljes robotmodellt Autodesk Inventor Professional 2017-es fejlesztőkörnyezetben állítottam össze, mely modell az 5.2. ábrán látható.

A program lehetőséget nyújt az egyes szegmensek közti kapcsolat definiálására. Erre több lehetőség is van, számunkra releváns az úgynevezett „Rotational Joint" alkalmazása, mely a testek közötti rotációs mozgást teszik lehetővé. Megadhatjuk az egyes szegmensek kapcsolódási pontjait, illetve az egymáshoz képesti elfordulásuk tengelyét.

Az elkészült modellt STL formátumba exportáltam ki, amit OpenGL segítségével már programból tudtam manipulálni (3.3. fejezet). A kiexportált modell minden egyes tengelyhez tartalmazott egy-egy különálló háromszögekből álló hálót (mesh-t).

A mesh-ben található háromszögek csúcspontjainak koordinátái az Autodesk-ben definiált világ-koordináta-rendszerben vannak megadva.

Vegyük a k. szegmenst ($1 < k \leq n$) leíró hálót, legyen $\mathbf{T}_{k-1,k}(\mathbf{q})$ a $(k-1)$. és a k. szegmens koordináta rendszere közötti homogén transzformációs mátrix, illetve $\mathbf{x}_R(\mathbf{q})$ a



5.1. ábra. Felső két képen látható a V-REP által szimulált RGB kép, illetve az általam az OpenGL segítségével generált virtuális RGB kép. Az alsó két képen ugyanezen ábrák mélységképi megfelelője látható.

mesh egy pontjának világ koordináta rendszerben lévő koordinátáit leíró vektor a $\mathbf{q}$ konfiguráció esetén. A modell betöltésekor ismert a teljes robotkart leíró mesh-ek minden egyes $\mathbf{x}_R(\mathbf{0})$ pontja. Melyekből meghatározható minden egyes pontra az adott $\mathbf{q}$ konfigurációban felvett értéke:

$$\mathbf{x}_R(\mathbf{q}) = \mathbf{M}_k(\mathbf{q}) \cdot \mathbf{x}_R(\mathbf{0}), \tag{5.1}$$

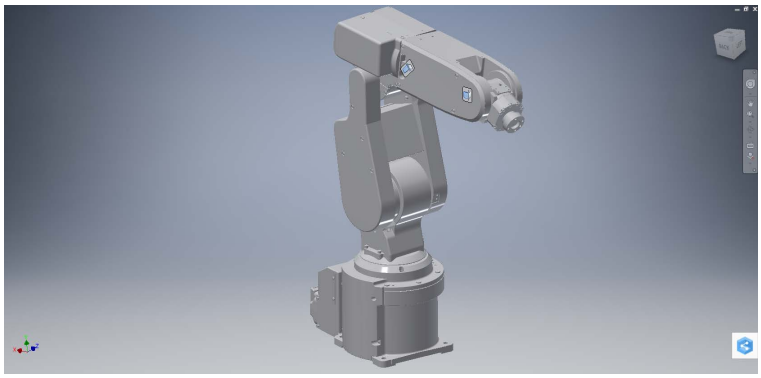
ahol $\mathbf{M}_k$-t a k. szegmenshez tartozó modell mátrixnak nevezzük (amit már a 3.4. ábrán láthattunk) és az alábbi módon számolható:

$$\begin{aligned}\mathbf{M}_k(\mathbf{q}) =& \mathbf{T}_{0,1}(\mathbf{q}) \cdot \mathbf{R}_1(\mathbf{q}) \cdot \mathbf{T}_{1,2}(\mathbf{q}) \cdot \mathbf{R}_2(\mathbf{q}) \cdot \ldots \cdot \mathbf{T}_{k-1,k}(\mathbf{q}) \cdot \mathbf{R}_k(\mathbf{q}) \cdot \\ & \cdot \mathbf{T}^{-1}_{k-1,k}(\mathbf{q}) \cdot \ldots \cdot \mathbf{R}_2^{-1}(\mathbf{q}) \cdot \mathbf{T}^{-1}_{1,2}(\mathbf{q}) \cdot \mathbf{R}_1^{-1}(\mathbf{q}) \cdot \mathbf{T}^{-1}_{0,1}(\mathbf{q}),\end{aligned} \tag{5.2}$$

ahol $\mathbf{R}_i(\mathbf{q})$ jelenti az i. szegmenshez tartozó z tengely körüli forgatásnak a transzformációs mátrixát:

$$\mathbf{R}_i(\mathbf{q}) = \begin{bmatrix} \cos(q_i) & \sin(q_i) & 0 & 0 \\ -\sin(q_i) & \cos(q_i) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{5.3}$$

(5.2) azt jelenti, hogy minden egyes szegmensen egészen a k. szegmensig végigmegyünk úgy, hogy egyesével átvisszük minden egyes csukló középpontját a világ koordináta rendszer középpontjába úgy, hogy annak z tengelye egybeessen a forgástengellyel. Elvégezzük az adott csuklóra vonatkozó forgatást, majd haladunk tovább. Miután már a k. szegmenst is elértük, visszafele megyünk végig az egyes transzformációk inverzein, így kapva meg a mesh összes $\mathbf{x}_R(\mathbf{q})$ pontját.



5.2. ábra. 3D-s modell

A 3.4. ábrán látható transzformációk közül hátra maradtak még a View (ε) és a Projection ($\mathcal{K}$) mátrixok, melyeket a kamera belső és külső paramétereiből tudunk meghatározni, a 3.2.1. fejezetben leírt módon.

Fontos megjegyezni, hogy az OpenGL nem kameraként kezeli a megjelenített ablakot, hanem a kamera hatásának eléréséhez annak mozgásakor a tér összes objektumát kell mozgatnunk a kamera mozgásával ellentétes irányba.

Így a vertex shaderben (3.3.1. fejezet) megvalósítandó transzformáció az alábbi:

$$\mathbf{x}_s(\mathbf{q}) = \mathcal{K} \cdot \varepsilon \cdot \mathbf{M}(\mathbf{q}) \cdot \mathbf{x}_R(0), \tag{5.4}$$

ahol $\mathbf{x}_s(\mathbf{q})$ az $\mathbf{x}_R(0)$ pont a megjelenítésre szolgáló ablakon lévő helye ($[-1, 1]$ intervallumra skálázva), a robot $\mathbf{q}$ konfigurációja esetén.

Miután minden egyes szegmensre elvégeztem az előző metódust, majd egyesével kirajzoltam azokat, már csak az OpenGL által generált mélységpuffer lekérése maradt hátra.

Az így kapott képet virtuális mélységképnek hívom a továbbiakban, melynek generálása átlagosan 5 *ms*-ot vett igénybe, egy Nvidia Geforce GTX 1050ti videokártyán kiértékelve.

5.2. Kamerakalibráció genetikus algoritmus segítségével

Az 5.1. fejezetben bemutatott módszer hátránya, hogy nehezen lehet pontosan meghatározni a kamera paramétereit, ezért el kell végeznünk a kamera kalibrációját.

Az volt a célom, hogy ismert robotkar állás esetén próbáljam meg azt az optimális kameraállást megtalálni, mellyel a lehető legjobban illeszkedik a generált virtuális mélységkép a valós kamera képére, feltéve, hogy a kamera belső paraméterei ismertek.

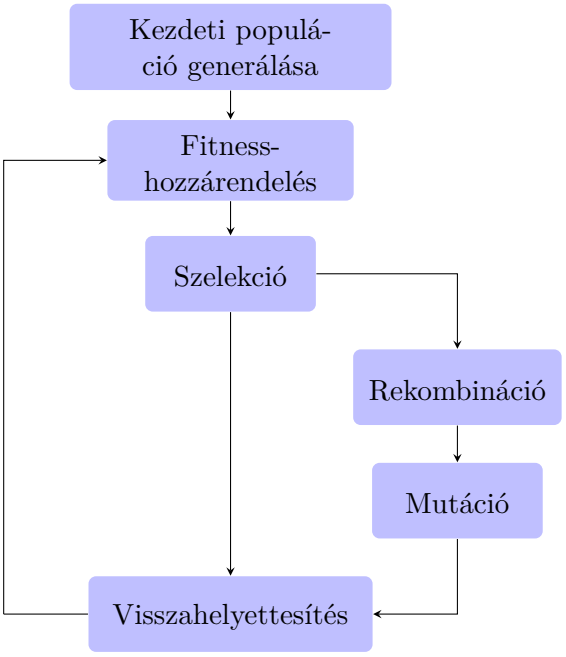
A kamera külső paramétereit genetikus algoritmus segítségével határoztam meg.

5.2.1. Genetikus algoritmus működése

A genetikus algoritmusok előnye, hogy a gradiens alapú optimumkeresési módszerekkel ellentétben akkor is képes megtalálni egy költségfüggvény globális minimumhelyét, ha az több lokális minimumhellyel is rendelkezik [23]. Az 5.3. ábrán látható az egypopulációs genetikus algoritmus általános leírása.

Mint minden lágy számítási módszer, ez is a természetből vette az alapötletet, mégpedig az evolúció folyamatát próbálja lemodellezni.

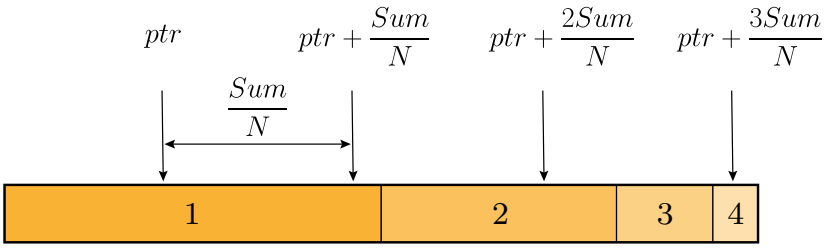
Az alábbi biológiai alapfogalmak szükségesek az algoritmus működésének megértéséhez:



5.3. ábra. Genetikus algoritmust leíró folyamatábra

- **Gén:** Genetikai információt hordozó öröklődési anyag, a szervezet bizonyos tulajdonságait határozza meg.
- **Kromoszóma:** A gének hordozója, egyedek egy új generációjának keletkezése során a szülők kromoszómái egymáshoz illeszkednek, majd egy- vagy többszörös átkereszteződéssel új kromoszóma jön létre.
- **Allél:** A kromoszómában található gének egy funkcionális módosulata.
- **Genotípus:** A génekben tárolt genetikai információk összessége, az egyed alléljainak felsorolása.
- **Fenotípus:** Az adott egyed megfigyelhető jellemzője, melyet a genotípus és a környezeti hatások szabnak meg.
- **Egyed:** Élő szervezet, mely rendelkezik bizonyos tulajdonságokkal, illetve az ezeket befolyásoló genotípussal.
- **Populáció:** Egy fajhoz tartozó egyedek egy csoportja, mely egyedei között fennáll a párosodás lehetősége.
- **Szelekció:** Bizonyos egyedek életben maradását, illetve szaporodását gátló természetes folyamat. Ez az egyedek bizonyos tulajdonságai alapján történik meg.
- **Rekombináció:** Az a folyamat, mely során a szülő egyedek kromoszómái kereszteződnek és egy új genotípust hoznak létre.
- **Mutáció:** Az öröklési anyagokban bekövetkező véletlenszerű változás.
- **Fitness (alkalmassági) érték:** Az adott egyed esélye az életben maradásra és szaporodásra.

A feladat során egy N_{var} változós skalárértékű f költségfüggvény globális minimumhelyét szeretnénk megtalálni. Minden egyes x_i változóra korlátokat kell előírnunk, azaz



5.4. ábra. Sztochasztikus univerzális mintavételezés

csak véges értelmezési tartományon keressük a függvény globális optimumát $L_i \leq x_i \leq U_i$, ahol L_i az értelmezési tartomány alsó, míg U_i a felső határértékét jelöli.

Fitness-hozzárendelés

Az $F(x)$ fitness-függvény segítségével az adott populáció minden egyes egyedéhez egy-egy életképességi mérőszámot rendel hozzá, az $f(x)$ költségfüggvény értéke alapján. A fitness-függvény tulajdonsága, hogy értéke pozitív, és megtartja a rendezettséget. Általában a jobb tulajdonságú egyedhez rendeljük nagyobb fitness értéket.

Szelekció

A szelekció során megadjuk, hogy az egyes egyedek mekkora valószínűséggel örökítsék tovább a génjeiket. Az egyedek fitness értéke alapján eldöntjük, hogy egy adott egyed mekkora eséllyel vesz részt a rekombinációban, majd pedig ez alapján véletlenszerűen kiválasztjuk az $N = GGAP \cdot N_{ind}$ darab szülőt.

Az alkalmazott módszer az úgynevezett sztochasztikus univerzális mintavételezés, mely hatékonyság szempontjából az egyik legjobbnak mondható ($\sigma(N)$ komplexitású), ezen felül zérus torzítással és minimális szórással rendelkezik.

A módszer lényege, hogy az egyes egyedeket levetítjük a fitness értékük alapján a $[0, sum]$ intervallumra, ahol sum jelenti a teljes N méretű populáció fitness értékeinek összegét. Ekkor véletlenszerűen válasszunk ki egy ptr mutatót a $[0, sum/N]$ intervallumon, majd jelöljünk ettől ekvidisztánsan $N-1$ darab újabb mutatót, egymástól $\frac{sum}{N}$ távolságra.

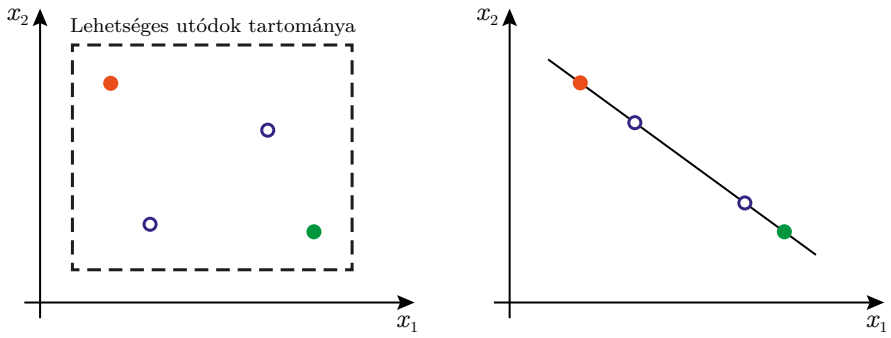
$$\{ptr, ptr + \frac{sum}{N}, \ldots, ptr + (N-1)\frac{sum}{N}\} \tag{5.5}$$

Ezen mutatók mindegyike a populáció egy adott egyedére fog mutatni, és mivel a $[0, sum]$ intervallum a fitness értékekkel súlyozva van felosztva, ezért a nagyobb fitness értékű egyedet nagyobb valószínűséggel választjuk ki a rekombinációra. A mutatók kiválasztásának menete az 5.4. ábrán látható.

Rekombináció

Valós populáció esetén történő rekombinációból két fajtát különbözetünk meg: a közbenső rekombinációt, illetve az egyenes mentén történő rekombináció (5.5. ábra).

A közbenső rekombináció során az $\mathbf{O}_1$ és $\mathbf{O}_2$ egyedek fenotípusait a $\mathbf{P}_1$ és $\mathbf{P}_2$ fenotípusa segítségével meghatározható hiperkockából választjuk ki. Ehhez generálnunk kell egy $\mathbf{a}_i$ vektort ($\dim \mathbf{a}_i = N_{var}$), melynek minden komponense a $[-0.25, 1.25]$ intervallum



5.5. ábra. Valós populáció esetén történő rekombináció, zölddel, illetve pirossal vannak jelölve a rekombinációban résztvevő egyedek, míg kékkel jelöltem az utódokat. A bal oldali ábrán látható a közbenső rekombináció, míg a jobb oldalon az egyenes mentén történő rekombinációt szemléltetem.

eleme.

$$\mathbf{O}_i = \mathbf{P}_1 + \mathbf{a}_i .* (\mathbf{P}_2 - \mathbf{P}_1), \tag{5.6}$$

ahol a $.*$ operátor a két vektor elemenként történő összeszorzását jelenti.

Egyenes mentén történő rekombináció esetén csupán utódonként egyetlen α_i véletlen skalár számot választunk a $[-0.25, 1.25]$ intervallumról.

$$\mathbf{O}_i = \mathbf{P}_1 + \alpha_i (\mathbf{P}_2 - \mathbf{P}_1), \tag{5.7}$$

Az elkészített algoritmusban a közbenső rekombináció algoritmusát használtam.

Mutáció

Valós populáció esetén, az egyedek mutációja során a kromoszóma értékének kis mértékű perturbációját végezzük el, az alábbi képlet segítségével:

$$x_{mut} = x + mutmask \cdot \frac{U - L}{2} \cdot mutshrink \cdot \Delta, \tag{5.8}$$

ahol x_{mut} az eredeti x véletlenszerűen kiválasztott kromoszóma mutálódott értéke. A $mutmask$ paraméterrel megadhatjuk, hogy az adott kromoszóma kiválasztásra kerüljön-e a mutáció folyamatában, illetve ha igen, milyen irányba mozduljon el az értéke ($mutmask \in \{-1, 0, +1\}$). Az U és L segítségével megadhatjuk a perturbáció abszolút értékének korlátait. A $mutshrink$ paraméter az úgynevezett összehúzási érték. Segítségével az algoritmus futása során csökkenthetjük a mutáció mértékét. A Δ paramétert pedig az alábbi módon kaphatjuk meg:

$$\Delta = \sum_{i=1}^{m-1} \alpha_i 2^{-i}, \tag{5.9}$$

ahol m a számábrázolás pontossága, bitszáma.

Visszahelyettesítés

A visszahelyettesítés során kiválasztjuk a régi populációnak azt az $rrate \cdot N_{ind}$ darab elemét, melyeket a következő generációban helyettesíteni szeretnénk az utódok valamely tagjaival.

5.6. ábra. A kamerakalibráció után számított különbségkép eredménye

Ezen kiválasztás történhet véletlenszerűen, ekkor egyenletes visszahelyettesítésről beszélünk, vagy történhet a fitness értékek alapján. Ez utóbbit elitista stratégiának is hívjuk, ugyanis ekkor a legjobb egyedek minden esetben bekerülnek a következő generáció populációjába is.

5.2.2. Kamerakalibráció menete

A kamerakalibrációhoz először is definiálnunk kell az egyedek kromoszómáinak felépítését. Ez az én esetemben öt darab „double" típusú változót választottam, melyek reprezentálják a kamera külső paramétereit. Ezek nem mások, mint a kamera $\begin{bmatrix} X & Y & Z \end{bmatrix}$ pozíciója, illetve a kamera koordináta rendszerében értelmezett „*yaw*", illetve „*pitch*" szögek. Feltételezhetjük, hogy a kamera síkjára merőleges z tengely mentén nem történik forgatás.

A kamera pozícióváltozóit egy előre kimért $\begin{bmatrix} X_0 & Y_0 & Z_0 \end{bmatrix}$ nominális érték köré írt $\pm 15cm$-es kockán belül, míg „yaw" szöget a $[-180°, 180°]$, a „pitch" szöget pedig a $[-90°, 90°]$ tartományon belül kerestem.

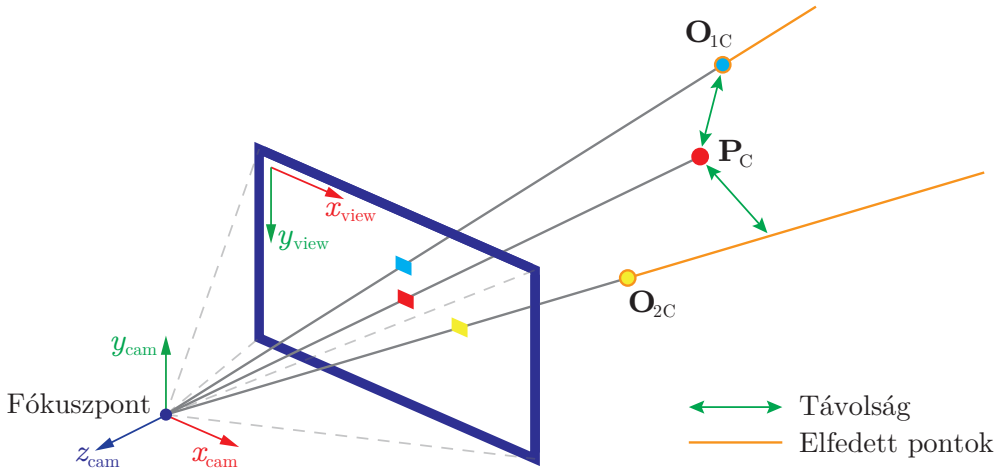
Ezen kívül még a költségfüggvény meghatározása van hátra, amihez használtam az 5.1. fejezetben megvalósított virtuális mélységképet.

Vettem a valós, illetve a virtuális mélységkép különbségét, majd minden olyan pontban lévő értéket megtartottam, mely pixelekben az virtuális mélységképen eredetileg megtalálható volt a robotkar. Majd vettem az így megmaradt pixelértékek négyzetének az átlagát és ezt definiáltam költségként. Így egy átlagos négyzetes eltérést a minimálisra optimalizáló genetikus algoritmust kapunk.

Azért volt szükség a négyzetek átlagát venni, mivel előfordulhat olyan eset, amikor csupán azzal, hogy a kamerát távolabb definiáljuk a robotkarhoz képest, a kamera konstans felbontásából adódóan kevesebb számú pontot vizsgálunk az összegzésnél, így akár egy rosszabb tulajdonságú egyed esetén is kisebb lehet ez az összeg.

Abban az esetben, ha nem a teljes robotkar volt látható eredetileg a virtuális képen, akkor egy kiugróan nagy költséget adtam a függvénynek, így küszöbölve ki a biztosan rossz megoldásokat.

A kalibráció után kapott különbségkép látható az 5.6. ábrán. Az esetlegesen fennmaradó kisebb hibákat a morfológiai nyitás műveletével el tudtam távolítani, így a kapott különbségképen megkaptam a robotkar körüli akadályok pozícióját a térben.



5.7. ábra. A mélységtérben történő távolságszámítást szemlélteti az ábra. Látható, hogy a kamerától a $\mathbf{P}_C$ ponthoz képest távolabb lévő $\mathbf{O}_{1C}$ ponttól közvetlenül mérjük a távolságot, míg ellenkező esetben az $\mathbf{O}_{2C}$ pont esetén figyelembe vesszük, hogy az elfedi a kamerától a pont mögött lévő teret.

5.3. Minimális távolság meghatározása

A robotkar mozgási trajektóriájának dinamikus módosításához szükségünk van a robotkar és az őt körülvevő akadályok minimális távolságára, valamint ennek irányára [14].

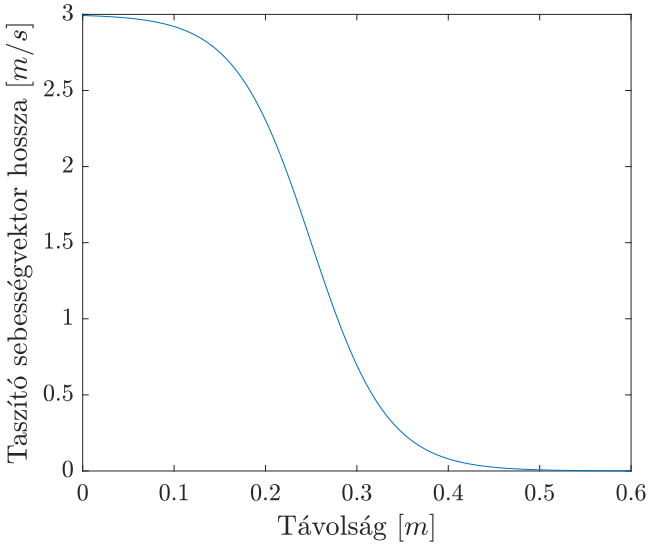
Legyen $\mathbf{O}$ az akadály egy pontja, melynek a mélységképen lévő koordinátái $\mathbf{O}_D = \begin{bmatrix} o_x & o_y & d_o \end{bmatrix}^\top$. Legyen $\mathbf{P}$ a robotkar azon pontja, melynek $\mathbf{O}$-tól való távolságát keressük, és $\mathbf{P}_D = \begin{bmatrix} p_x & p_y & d_p \end{bmatrix}^\top$ ennek mélységtérbeli megfelelője.

A távolság számításához két lehetséges esetet kell vizsgálnunk, elsőként ha $\mathbf{O}$ távolabb helyezkedik el a kamerától, mint $\mathbf{P}$. Ekkor biztos, hogy $\mathbf{O}$ közelebb lesz $\mathbf{P}$-hez, mint az objektum azon pontjai, melyeket $\mathbf{O}$ kitakar (ld. $\mathbf{O}_{1C}$ pont az 5.7. ábrán). (3.3) alapján levezethető a két pont távolságának képlete:

$$\begin{aligned} v_x &= \frac{(o_x - c_x)s_x d_o - (p_x - c_x)s_x d_p}{f} \\ v_y &= \frac{(c_y - o_y)s_y d_o - (c_y - p_y)s_y d_p}{f} \\ v_z &= d_p - d_o \\ \|\mathbf{D}(\mathbf{P},\mathbf{O})\| &= \sqrt{v_x^2 + v_y^2 + v_z^2}, \end{aligned} \tag{5.10}$$

ahol $\mathbf{D}(\mathbf{P},\mathbf{O}) = \begin{bmatrix} v_x & v_y & v_z \end{bmatrix}^\top$, a $\mathbf{P}$-ből $\mathbf{O}$-ba mutató irányvektor.

Amennyiben $d_o < d_p$, azaz $\mathbf{O}$ közelebb helyezkedik el a kamerához, abban az esetben nem tehetjük meg, hogy az ismert $\mathbf{O}$ ponthoz vesszük a $\mathbf{P}$ távolságát, mivel ez nem biztos, hogy a lehető legrövidebb távolság lesz, hiszen nincsen információnk az objektum alakjáról. Ilyen esetben azt kell feltételeznünk, hogy az objektum folytatódik mindaddig



5.8. ábra. Taszító sebességvektor hossza a távolság függvényében (paraméterek: $V_{max} = 3\frac{m}{s}$, $\rho = 0.5m$, $\alpha = 6$)

a pontig, amíg $d_o = d_p$ lesz, majd erre a pontra végezzük el (5.10) egyenletekben foglalt távolságszámítási metódust. Ezt szemlélteti az 5.7. ábra $\mathbf{O}_{2C}$ pontja.

5.4. Taszító sebességvektor számítása

A legtöbb dinamikus objektumelkerülést megvalósító algoritmus a robot körüli akadályokhoz képesti távolságot használja a trajektória meghatározásához, mely értékének meghatározását már az 5.3. fejezetben ismertettem.

$\mathbf{D}(\mathbf{P}, \mathbf{O})$ vektor, illetve annak nagysága segítségével előállíthatunk egy olyan vektort, mellyel később meghatározhatjuk a trajektória szükséges módosításának mértékét:

$$\mathbf{V}_C(\mathbf{P}, \mathbf{O}) = v(\mathbf{P}, \mathbf{O}) \frac{\mathbf{D}(\mathbf{P}, \mathbf{O})}{\|\mathbf{D}(\mathbf{P}, \mathbf{O})\|} \tag{5.11}$$

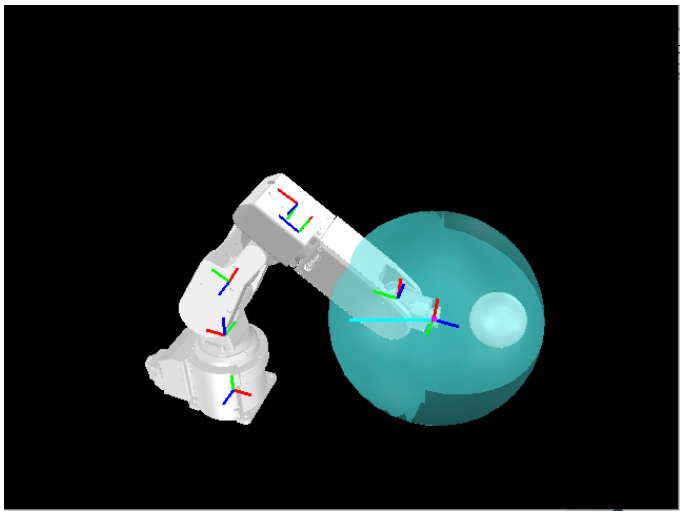
A vektor iránya megegyezik a $\mathbf{D}(\mathbf{P}, \mathbf{O})$ vektor irányával, a hossza pedig:

$$v(\mathbf{P}, \mathbf{O}) = \frac{V_{max}}{1 + e^{(\|\mathbf{D}(\mathbf{P},\mathbf{O})\|(2/\rho)-1)\alpha}}, \tag{5.12}$$

ahol V_{max} a hossz maximális értéke, amit $\|\mathbf{D}(\mathbf{P}, \mathbf{O})\| = 0$ esetén érhetünk el. Az α értékkel az exponenciális lecsengés alakját adhatjuk meg, míg ρ az a távolság, ahol a függvény eléri a nulla értéket. Az így kapott függvény alakja az 5.8. ábrán látható.

Első megfontolásként tekinthetnénk eredő taszító sebességvektornak a legkisebb távolságú objektumponthoz ($\mathbf{O}_{min}$) tartozó $\mathbf{V}_C(\mathbf{P}, \mathbf{O}_{min})$ vektort. Viszont, ahogy a [14] cikkben is látható, ez nem feltétlen vezet a legjobb megoldásra. Egy ésszerű választás lehet, ha a robotkar közvetlen $\mathbf{S}$ környezetében lévő összes objektum segítségével számítjuk ki a taszító vektor irányát, míg a vektor nagyságát származtatjuk csupán a legközelebb lévő pont segítségével.

$$\begin{aligned} \mathbf{V}_{C_T}(\mathbf{P}) &= \sum_{\mathbf{O} \in \mathbf{S}} \mathbf{V}_C(\mathbf{P}, \mathbf{O}) \\ \mathbf{V}_{C_{all}}(\mathbf{P}) &= v(\mathbf{P}, \mathbf{O}_{min}) \frac{\mathbf{V}_{C_T}(\mathbf{P})}{\|\mathbf{V}_{C_T}(\mathbf{P})\|} \end{aligned} \tag{5.13}$$



5.9. ábra. A végeffektorra számított taszító vektor ábrázolása

Ezt követően a meghatározott vektort át kell transzformálnunk a kamera koordináta rendszeréből a referencia koordináta rendszerbe:

$$\mathbf{V}_{R_{all}}(\mathbf{P}) = \mathbf{R}^T \mathbf{V}_{C_{all}}(\mathbf{P}) \tag{5.14}$$

Az így kapott vektort szemlélteti az 5.9. ábra, melyen a robotkar látható egy akadály jelenlétében (fehér gömb). Kék, 15 *cm* sugarú gömbbel jelöltem a végeffektor azon **S** környezetét, melyen belül figyelembe vesszük az objektumok pontjait a taszító vektor számításához. Jelöltem a bázis, illetve minden egyes csuklóban található koordináta-rendszert, illetve világoskék egyenessel végeffektorból kiinduló taszító sebességvektort.

5.5. Mozgástervezés

A robotkarnak mozgása során egy előírt taszkot, feladatot kell teljesíteni. Ez egy offline pályatervezési művelet eredménye, mely előírja a robotkar csuklókoordinátáinak időbeli változását, az előre ismert objektumok elkerülése mellett (ld. 4 fejezet). Az online trajektóriatervezés során ezt az előírt pályát kell módosítanunk úgy, hogy az elkerülje az időközben a manipulátor útjába kerülő akadályokat.

A trajektória tervezéséhez kontroll pontokat jelölünk ki a robotkar teljes hosszában, a végeffektorral bezárólag. Ezen pontok fogják alkotni azon P pontok halmazát, amelyekhez kiszámoljuk az 5.4. fejezetben ismertetett taszító sebességvektorokat.

A mozgástervezést két részre bontjuk, elsőként a végeffektoron található kontrollpontra, majd az egyéb kontrollpontokra vonatkozóan.

5.5.1. Trajektóriatervezés a végeffektorra

Feltehetjük, hogy a robotkar által teljesíteni kívánt taszk a végeffektor valós térben lévő $\dot{\mathbf{x}}_d$ kívánt sebességével van definiálva. Legyen $\mathbf{P}_{EE}$ a végeffektorhoz rögzített kontrollpont.

Ekkor a módosított trajektória megkapható az alábbi módon:

$$\dot{\mathbf{x}}_v = \dot{\mathbf{x}}_d + \mathbf{V}_{R_{all}}(\mathbf{P}_{EE}) \tag{5.15}$$

Majd az így kapott korrigált végeffektor sebességből számítható a csuklókoordináták sebessége az adott robotkarra jellemző Jacobi-mátrix pszeudoinverzével (ld. 2.3 fejezet):

$$\dot{\mathbf{q}} = {}^0\mathbf{J}_n^{\#}(\mathbf{q})\dot{\mathbf{x}}_v \tag{5.16}$$

Ez a megoldás a mesterséges potenciálterek módszerével egyezik meg, ennek hátránya, hogy léteznek lokális minimumhelyek, melyekből lehetséges, hogy nem tud kikerülni a robot.

5.5.2. Teljes robotkarra vonatozó trajektóriatervezés

A robotkaron található többi kontrollpont által kifejtett hatást egy másik módszer segítségével lehet figyelembe venni [12]. Az algoritmus során az a cél, hogy amennyiben nem szükséges ne változtassuk meg a végeffektor által leírandó trajektóriát. Ezt úgy lehet megtenni, ha kihasználjuk a robotkar esetleges redundanciáját.

A robot redundanciáját mindig az adott taszk m szabadságfokához képest nézzük. Ha a Mitsubishi RV-2F-Q típusú $n = 6$ szabadságfokú robotkart tekintjük, akkor ennek robotnak létezik olyan munkaterülete, hogy a manipulátor az ebben található minden egyes pozíciót bármely orientációban el tudja érni. Tehát ha az $m = 6$ szabadságfokú pozicionálás és orientálási feladatot vesszük, akkor ez a robot nem tekinthető redundánsnak, viszont ha csak pozicionálást végzünk ($m = 3$ szabadságfokkal), akkor a fent említett munkatérben a redundancia foka $n - m = 3$.

A robotkar redundanciáját kihasználhatjuk, ha nem a konkrét sebességeket írjuk elő az egyes csuklóváltozókra, hanem a trajektória generálása során kényszereket írunk fel a csuklóváltozók változásának sebességére.

Legyen $\mathbf{C}$ egy a kontrollpontok közül, és ${}^0\mathbf{J}_C$ az ehhez tartozó parciális Jakobi-mátrix. A pont és az objektumok közötti minimális $D_{min}(\mathbf{C}) = \|\mathbf{D}(\mathbf{C}, \mathbf{O}_{min})\|$ távolság az előzőekben bemutatottak szerint számítható.

Ekkor definiálhatunk egy függvényt, ami az ütközés kockázatát írja le:

$$f(D_{min}(\mathbf{C})) = \frac{1}{1 + e^{(D_{min}(\mathbf{C})(2/\rho)-1)\alpha}}, \tag{5.17}$$

ahol ρ és α az (5.12)-ben láthatóan definiálhatóak.

Ebből a legkisebb távolság irányába mutató vektor segítségével vektort képezve, majd ezt áttranszformálva a konfigurációs térbe, megkapjuk az $\mathbf{s}$ vektort, amelynek minden egyes s_i koordinátája megmondja, hogy az adott valós térbeli kényszer mennyire van hatással az i. csuklóra ($i = 1 \dots n$).

$$\mathbf{s} = {}^0\mathbf{J}_C^T \frac{\mathbf{D}(\mathbf{C}, \mathbf{O}_{min})}{\|\mathbf{D}(\mathbf{C}, \mathbf{O}_{min})\|} f(D_{min}(\mathbf{C})) \tag{5.18}$$

Az $\mathbf{s}$ vektor segítségével definiálhatjuk az egyes csuklóváltozók sebességére vonatkozó határértékeket:

$$\begin{array}{lll} \text{if} & (s_i \geq 0) & \dot{q}_{max,i} = V_{max,i}(1 - f(D_{min}(\mathbf{C}))) \\ \text{else} & & \dot{q}_{min,i} = -V_{max,i}(1 - f(D_{min}(\mathbf{C}))), \end{array} \tag{5.19}$$

ahol $V_{max,i}$ az i. csuklóváltozó sebességének eredeti korlátozása.

Ezzel a módszerrel elértük, hogy az ütközéselkerülés mindig magasabb prioritású feladat legyen, mint a kijelölt taszk végrehajtása, ami a biztonságkritikus rendszerek esetén elengedhetetlen szempont.

5.6. Szimulációs eredmények

Több esetben is vizsgáltam az algoritmus működését, melyeket videókon szemléltetek.

A videókon a képernyő jobb oldalán látható a V-REP szimulációs környezet, bal oldalon pedig az OpenGL-el generált virtuális robotkar mozgása, illetve az általam és a V-REP által generált mélységképek különbsége. A különbségképen ábrázoltam a bázishoz, illetve az egyes csuklókhoz tartozó koordináta-rendszereket, világoskék vektorral jelöltem az aktuális taszító sebességvektort, illetve sárga vonal jelzi a végeffektor által befutott pályát.

Az első esetben a robotkar végeffektorának kívánt mozgását egy egyenes mentén történő mozgásnak írtam elő, majd akadályok véletlenszerű mozgását szimuláltam. Az így kapott trajektória az alábbi videón látható: `https://youtu.be/ttIlK14tADY`.

Egy ehhez hasonló esetet mutat be a következő videó, melyen a robotkar végeffektorának egy háromszög mentén kell haladnia mindaddig, míg valamilyen dinamikusan mozgó akadály el nem téríti ettől a feladattól: `https://youtu.be/p4ydCNg-6Kk`.

A módszer alkalmazhatóságának vannak korlátai, például abban az esetben, ha az akadály közvetlenül a manipulátor mögött vagy előtt helyezkedik el, abban az esetben nem tudjuk egyértelműen meghatározni, hogy az adott konfiguráció ütközéssel jár-e.

Egy másik probléma, hogy a kamerához közeli akadályok befolyásolni tudják a robotkar mozgását, mivel a robotkar és az akadály távolságát ekkor a kontrollpont és kamerából az akadály pontjába induló egyenes távolságaként definiáltuk (5.3. fejezet).

Ezen korlátokat szemlélteti az alábbi videó: `https://youtu.be/gO0a2afhvSo`

Mindkét problémára megoldást jelent, ha egyszerre több megfelelően elhelyezett kamerával vizsgáljuk a robotkar környezetét [11]. Ekkor több probléma is felmerülhet, egyrészt meg kell feleltetni egymásnak az egyes kamerák által vett pontokat, illetve valós környezetben való alkalmazáskor figyelembe kell venni azt is, hogy a bemutatott mélységi kamerák működésükből adódóan interferenciába lépnek egymással [25].

Végül pedig teszteltem offline és az online mozgástervezés együttes használhatóságát. Az alábbi példán az ismertetett offline pályatervezési módszert által kapott utat követi a robotkar, miközben elkerüli az eközben útba kerülő dinamikus akadályokat: `https://youtu.be/I9321avg3do`.

Azt tapasztaltam, hogy nehezen megoldható olyan statikus akadályok megközelítése, melyek a robotkar és a kamera között találhatóak. Erre a problémára szintén megoldást jelent, ha több kamerával figyeljük meg a teret.

6. fejezet

Eredmények, továbbfejlesztési lehetőségek

Ebben a fejezetben bemutatom és értékelem a kapott eredményeket mind az offline, mind az online mozgástervezési feladat esetére, illetve kitérek az esetleges továbbfejlesztési lehetőségekre.

6.1. Offline mozgástervezés

A bemutatott offline mozgástervezési megoldás képes ütközésmentes pályát találni a robotkar két konfigurációja között, majd ennek megfelelő trajektóriát generálni.

A pályatervezéshez használt T-RRT algoritmus hatékony módszernek bizonyult arra, hogy a költségfüggvény alapján jó minőségű ütközésmentes pályát találjunk.

Több módszer közül végül a legközelebbi szomszéd alapú klaszterezés algoritmust választottam, mellyel hatékonyan tudtam megbecsülni a pályatervezéshez használt költségfüggvény.

A robotkar egyes szegmenseit befoglaló konvex poliéderek által meghatározott egyenlőtlenség-rendszerek megoldhatóságának vizsgálatával egyértelműen meg tudtam határozni egy adott konfigurációról, hogyha az adott konfiguráció ütközésmentes állapotot jelent.

Fontos megjegyezni, hogy a bemutatott módszer teljesen általános, tetszőleges robotkar esetén alkalmazható, az egyetlen megkötés az volt, hogy konvex poliéderekkel modelleztem a robotkart, de ezt bármely manipulátor esetén meg tudjuk tenni.

Az algoritmus továbbfejlesztéseként a T-RRT algoritmus helyett a bidirekcionális T-RRT algoritmust fogom kipróbálni, azaz mind a kiinduló, mind a célkonfigurációból indítunk egy-egy keresési fát, mely a szakirodalom szerint jóval gyorsabb futási időt eredményez, illetve komplexebb költségfüggvények esetén is effektív marad [10]. Fontos jövőbeli feladat még a módszer valós rendszeren történő működésének vizsgálata.

6.2. Online mozgástervezés

Az online mozgástervezési feladat megoldása során a robotkar körüli akadályokat mélységkamerával figyeltem meg, mellyel ki lehet számítani az általa szolgáltatott kép egyes pixeleinek a valós térben elfoglalt helyét.

Ahhoz, hogy a robotkart elkülönítsem a kamerán látható akadályoktól, létrehoztam egy virtuális teret, melyben a robotkar modelljét mozgatva legenerálható egy olyan mélységkép, melyen csak a manipulátor látható. A valódi és a virtuális mélységkép különbségéből pedig megkaptam a robotkar körüli akadályok mélységtérben elfoglalt helyét.

Ezt követően az akadályok pontjai és a robotkar egyes kontrollpontjai közötti távolságszámításával taszító sebességvektorokat számítottam. Majd a robotkar differenciális mozgásának leírását kihasználva meghatároztam, hogy a csuklóváltozók mely sebességei eredményezik a kontrollpontok előírt mozgását.

Az algoritmus továbbfejlesztési lehetőségei közé tartozik a valós rendszeren történő megvalósítás, illetve hogy több kamera által szolgáltatott mélységképet is figyelembe vegyek a mozgástervezésnél. Ezenkívül érdemes lenne adaptívan változtatni az adott akadály típusától függően azt a távolságot, amennyire még megközelíthetjük azt.

Továbbá sok számítást, melyek a mélységkép feldolgozását végezték, lehetne gyorsítani azzal, hogyha ezeket párhuzamosan egy videokártyán végeznénk el.

Köszönetnyilvánítás

Köszönettel tartozom konzulensemnek, Gincsainé Dr. Szádeczky-Kardoss Emesének, a rám áldozott idejéért, illetve a tanácsaiért, melyek segítettek, hogy elérjem a megvalósítani kívánt céljaimat.

Továbbá köszönöm menyasszonyomnak a kitartást és a sok segítséget, illetve szüleimnek a támogatást.

Irodalomjegyzék

[1] Robert Babuška – Henk Verbruggen: Neuro-fuzzy methods for nonlinear system identification. *Annual reviews in control*, 27. évf. (2003) 1. sz., 73–85. p.

[2] Lantos Béla: *Robotok irányítása (Robot Control)*. 2002, Akadémiai K.

[3] Michael Brady – John M Hollerbach – Timothy L Johnson – Tomás Lozano-Pérez – Matthew T Mason – Daniel G Bobrow – Patrick Henry Winston – Randall Davis: *Robot motion: Planning and control*. 1982, MIT press.

[4] Victor Castaneda – Nassir Navab: Time-of-flight and kinect imaging. *Kinect Programming for Computer Vision*, 2011.

[5] Massimo Cefalo – Emanuele Magrini – Giuseppe Oriolo: Parallel collision check for sensor based real-time motion planning. In *2017 IEEE International Conference on Robotics and Automation (ICRA)* (konferenciaanyag). 2017, IEEE, 1936–1943. p.

[6] Emese Gincsainé Szádeczky-Kardoss Dániel Szabó: Robotic manipulator path-planning: Cost-function approximation with fuzzy inference system. In *Proceedings of 2019 24th International Conference on Methods and Models in Automation and Robotics (MMAR)* (konferenciaanyag). 2019. augusztus, 259–264. p.

[7] Alessandro De Luca – Fabrizio Flacco: Integrated control for phri: Collision avoidance, detection, reaction and collaboration. In *2012 4th IEEE RAS & EMBS International Conference on Biomedical Robotics and Biomechatronics (BioRob)* (konferenciaanyag). 2012, IEEE, 288–295. p.

[8] Alessandro De Luca – Raffaella Mattone: Sensorless robot collision detection and hybrid force/motion control. In *Proceedings of the 2005 IEEE international conference on robotics and automation* (konferenciaanyag). 2005, IEEE, 999–1004. p.

[9] Joey de Vries: Learnopengl. `https://learnopengl.com/Getting-started/Hello-Triangle`. Accessed: 2019-05-11.

[10] Didier Devaurs – Thierry Siméon – Juan Cortés: Enhancing the transition-based rrt to deal with complex cost spaces. In *2013 IEEE International Conference on Robotics and Automation* (konferenciaanyag). 2013, IEEE, 4120–4125. p.

[11] Fabrizio Flacco – Alessandro De Luca: Multiple depth/presence sensors: Integration and optimal placement for human/robot coexistence. In *2010 IEEE International Conference on Robotics and Automation* (konferenciaanyag). 2010, IEEE, 3916–3923. p.

[12] Fabrizio Flacco – Alessandro De Luca – Oussama Khatib: Motion control of redundant robots under joint constraints: Saturation in the null space. In *2012 IEEE International Conference on Robotics and Automation* (konferenciaanyag). 2012, IEEE, 285–292. p.

[13] Fabrizio Flacco – Torsten Kroeger – Alessandro De Luca – Oussama Khatib: A depth space approach for evaluating distance to objects. *Journal of Intelligent & Robotic Systems*, 80. évf. (2015) 1. sz., 7–22. p.

[14] Fabrizio Flacco – Torsten Kröger – Alessandro De Luca – Oussama Khatib: A depth space approach to human-robot collision avoidance. In *2012 IEEE International Conference on Robotics and Automation* (konferenciaanyag). 2012, IEEE, 338–345. p.

[15] Sami Haddadin – Rico Belder – Alin Albu-Schäffer: Dynamic motion planning for robots in partially unknown environments. *IFAC Proceedings Volumes*, 44. évf. (2011) 1. sz., 6842–6850. p.

[16] Dug Hun Hong – Chang-Hwan Choi: Multicriteria fuzzy decision-making problems based on vague set theory. *Fuzzy sets and systems*, 114. évf. (2000) 1. sz., 103–113. p.

[17] Léonard Jaillet – Juan Cortés – Thierry Siméon: Sampling-based path planning on configuration-space costmaps. *IEEE Transactions on Robotics*, 26. évf. (2010) 4. sz., 635–646. p.

[18] Michael Edwin Kahn – B Roth: The near-minimum-time control of open-loop articulated kinematic chains. *Journal of Dynamic Systems, Measurement, and Control*, 93. évf. (1971) 3. sz., 164–172. p.

[19] Sertac Karaman – Emilio Frazzoli: Sampling-based algorithms for optimal motion planning. *The international journal of robotics research*, 30. évf. (2011) 7. sz., 846–894. p.

[20] Arthur D Kuo: A mechanical analysis of force distribution between redundant, multiple degree-of-freedom actuators in the human: Implications for the central nervous system. *Human movement science*, 13. évf. (1994) 5. sz., 635–663. p.

[21] Kostas J Kyriakopoulos – George N Saridis: Minimum jerk path generation. In *Proceedings. 1988 IEEE International Conference on Robotics and Automation* (konferenciaanyag). 1988, IEEE, 364–369. p.

[22] Chantal Landry – Matthias Gerdts – René Henrion – Dietmar Hömberg: Path-planning with collision avoidance in automotive industry. In *IFIP Conference on System Modeling and Optimization* (konferenciaanyag). 2011, Springer, 102–111. p.

[23] Béla Lantos: Fuzzy systems and genetic algorithms. *Műegyetemi Kiadó, Budapest*, 2002.

[24] Steven M LaValle: *Planning algorithms*. 2006, Cambridge University Press.

[25] Andrew Maimone – Henry Fuchs: Reducing interference between multiple structured light depth sensors using motion. In *2012 IEEE Virtual Reality Workshops (VRW)* (konferenciaanyag). 2012, IEEE, 51–54. p.

[26] MathWorks: Simscape. `https://www.mathworks.com/products/simscape.html`. Accessed: 2019-10-24.

[27] Mitsubishi Electic. *RV-2F-Q Series Standard Specifications Manual*. 2018. 6. BFP-A8902-AA.

[28] David E Orin – William W Schrader: Efficient computation of the jacobian for robot manipulators. *The International Journal of Robotics Research*, 3. évf. (1984) 4. sz., 66–75. p.

[29] Eric Rohmer – Surya PN Singh – Marc Freese: V-rep: A versatile and scalable robot simulation framework. In *2013 IEEE/RSJ International Conference on Intelligent Robots and Systems* (konferenciaanyag). 2013, IEEE, 1321–1326. p.

[30] Mark Segal – Kurt Akeley: *The OpenGL Graphics System: A Specification (Version 4.5 (Core Profile))*. 2017.

[31] Mark W Spong – Seth Hutchinson – Mathukumalli Vidyasagar és mások: *Robot modeling and control*. 2006, John Wiley & Sons, Inc.

[32] Anthony Stentz: Optimal and efficient path planning for partially known environments. In *Intelligent Unmanned Ground Vehicles*. 1997, Springer, 203–220. p.

[33] Peter JM Van Laarhoven – Emile HL Aarts: Simulated annealing. In *Simulated annealing: Theory and applications*. 1987, Springer, 7–15. p.

[34] L. . Wang: Training of fuzzy logic systems using nearest neighborhood clustering. In *[Proceedings 1993] Second IEEE International Conference on Fuzzy Systems* (konferenciaanyag). 1993. March, 13–17 vol.1. p.

[35] L-X Wang: Fuzzy systems are universal approximators. In *[1992 Proceedings] IEEE International Conference on Fuzzy Systems* (konferenciaanyag). 1992, IEEE, 1163–1170. p.

[36] L-X Wang: Stable adaptive fuzzy control of nonlinear systems. *IEEE Transactions on fuzzy systems*, 1. évf. (1993) 2. sz., 146–155. p.

[37] L-X Wang – Jerry M Mendel: Generating fuzzy rules by learning from examples. *IEEE Transactions on systems, man, and cybernetics*, 22. évf. (1992) 6. sz., 1414–1427. p.

[38] Oliver Wasenmüller – Didier Stricker: Comparison of kinect v1 and v2 depth images in terms of accuracy and precision. In *Asian Conference on Computer Vision* (konferenciaanyag). 2016, Springer, 34–45. p.

[39] Wikipedia: Kinect. `https://en.wikipedia.org/wiki/Kinect`. Accessed: 2019-10-18.

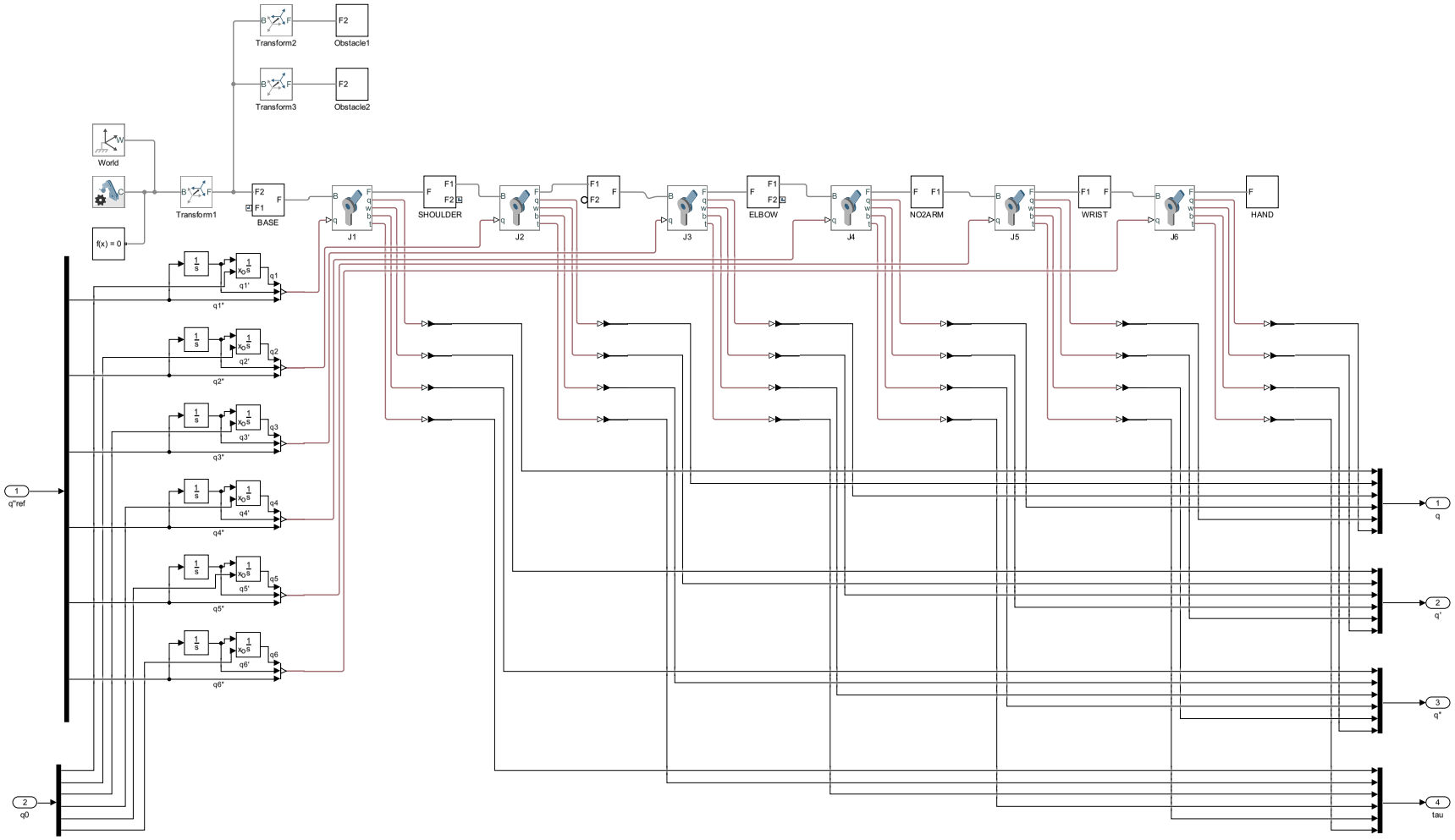
[40] Marios Xanthidis – Ioannis Rekleitis – Jason M O'Kane: Rrt+: Fast planning for high-dimensional configuration spaces. *arXiv preprint arXiv:1612.07333*, 2016.

[41] Ran Zhao: *Trajectory planning and control for robot manipulations*. PhD értekezés (Université Paul Sabatier-Toulouse III). 2015.

Függelék

Kategória	Robot komponens	Specifikáció
Szegmens hosszúság	Felkar	230 mm
	Alkar	270 mm
Csuklóváltozók határértékei	Első csukló - q_1	$[-240°, 240°]$
	Második csukló - q_2	$[-120°, 120°]$
	Harmadik csukló - q_3	$[0°, 160°]$
	Negyedik csukló - q_4	$[-200°, 200°]$
	Ötödik csukló - q_5	$[-120°, 120°]$
	Hatodik csukló - q_6	$[-360°, 360°]$
Csuklóváltozók sebességének határértékei	Első csukló - $\dot{q}_1$	$300°/s$
	Második csukló - $\dot{q}_2$	$150°/s$
	Harmadik csukló - $\dot{q}_3$	$300°/s$
	Negyedik csukló - $\dot{q}_4$	$450°/s$
	Ötödik csukló - $\dot{q}_5$	$450°/s$
	Hatodik csukló - $\dot{q}_6$	$720°/s$
Mozgás megismételhetősége	Végeffektor	± 0.02 mm
Maximálisan elérhető eredő sebesség		4.95 m/s

F.1. táblázat. Mitsubishi RV-2F-Q hat szabadságfokú manipulátor fizikai paraméterek [27]



F.1. ábra. A robotkar és az akadályok szimulációját megvalósító Simulink modell