Biološki inspirirano kretanje mobilnog robota temeljeno na evolucijskom algoritmu

Ugrinić, Bartol

SVEUČILIŠTE U ZAGREBU
FAKULTET STROJARSTVA I BRODOGRADNJE

DIPLOMSKI RAD

Bartol Ugrinić

Zagreb, 2024

SVEUČILIŠTE U ZAGREBU
FAKULTET STROJARSTVA I BRODOGRADNJE

DIPLOMSKI RAD

Mentor: Student:

Izv. prof. dr. sc. Tomislav Stipančić, dipl. ing. Bartol Ugrinić

Zagreb, 2024.

Izjavljujem da sam ovaj diplomski rad izradio samostalno koristeći znanja stečena tijekom studija i navedenu literaturu.

Zahvaljujem se prof. Stipančiću na srdačnosti i prihvaćanju mentorstva diplomskog rada. Hvala prof. Ćurkoviću na brojnim savjetima, idejama i podršci tokom pisanja rada. Hvala prijateljima koji nikad nisu zaboravili na studenta u dalekom gradu. Veliko hvala mojoj obitelji na konstantnoj podršci tokom studiranja. I hvala Anđeli koja je bila uz mene kroz padove i uspone bez koje ovaj rad ne bi bio tako brzo završen.

Bartol Ugrinić

SVEUČILIŠTE U ZAGREBU
FAKULTET STROJARSTVA I BRODOGRADNJE
Središnje povjerenstvo za završne i diplomske ispite
Povjerenstvo za diplomske ispite studija strojarstva za smjerove:
Proizvodno inženjerstvo, inženjerstvo materijala, industrijsko inženjerstvo i menadžment, mehatronika i robotika, autonomni sustavi i računalna inteligencija

Sveučilište u Zagrebu Fakultet strojarstva i brodogradnje	
Datum	Prilog
Klasa: **602 - 04 / 24 - 06 / 1**	
Ur.broj: **15 - 24 -**	

DIPLOMSKI ZADATAK

Student: **Bartol Ugrinić** JMBAG: 0069082104

Naslov rada na hrvatskom jeziku: **Biološki inspirirano kretanje mobilnog robota temeljeno na evolucijskom algoritmu**

Naslov rada na engleskom jeziku: **Biologically inspired movement of a mobile robot based on an evolutionary algorithm**

Opis zadatka:

Roboti pokretani nogama motivirani su načinima kretanja uočenim u prirodi, poput kretanja kukaca, sisavaca i na kraju čovjeka. Priroda je dugotrajnim evolucijskim procesom razvila ovakva kretanja koja su efikasna, fleksibilna i robusna. U računalnom kontekstu, evolucijski algoritmi omogućuju implementaciju pojednostavljenog prirodnog evolucijskog procesa s ciljem pronalaženja tehničkog rješenja optimiranog prema skupini zadanih kriterija. Temeljem iznesenog, u radu je potrebno oblikovati evolucijski algoritam koji će omogućiti četveronožnom robotu da pronađe koordinirane pokrete nogu, s ciljem efikasnog kretanja po pravcu.

Potrebno je napraviti sljedeće:

- osmisliti skup evaluacijskih kriterija i oblikovati evolucijski algoritam za učenje kretanja mobilnog robota
- dostupne 3D modele robota transformirati u URDF oblik i ugraditi u fizički simulator Pybullet
- koristiti Pybullet za evaluaciju kretnji generiranih od strane evolucijskog algoritma povezujući fizički simulator s okolinom za učenje
- kritički se osvrnuti na parametre evolucijskog algoritma i njihov utjecaj na kvalitetu pronađenog rješenja.

U radu je potrebno navesti korištenu literaturu i eventualno dobivenu pomoć.

Zadatak zadan:	Datum predaje rada:	Predviđeni datumi obrane:
7. ožujka 2024.	9. svibnja 2024.	13. – 17. svibnja 2024.
Zadatak zadao:		Predsjednik Povjerenstva:
Izv. prof. dr. sc. Tomislav Stipančić		Prof. dr. sc. Ivica Garašić

SADRŽAJ

POPIS SLIKA

POPIS OZNAKA I KRATICA

Oznaka	Jedinica	Opis
A	-	Duljina genoma
B	-	Broj genoma u populaciji
CAD	-	eng. Computer Aided Design
EA	-	Evolucijski Algoritam
Fitnes	-	Mjera dobrote pojedinca
IDE	-	eng. Integrated Development Enviroment
I	kgm^2	Moment inercije oko određene osi ili parova osi
K	-	Nasumična vrijednost kojom se odlučuje o križanju
M	-	Nasumična vrijednost kojom se odlučuje o mutaciji
NPZ	-	Binarni format datoteke za spremanje matrica
P_k	-	Vjerojatnost križanja genoma
P_m	-	Vjerojatnost mutacije genoma
Random roditelj	-	Nasumična vrijednost kojom se odabiru roditelji
Relativni fitnes	-	Mjera dobrote pojedinca naspram svoje populacije
ROS	-	eng. Robot Operating System
SSG	-	Stupnjevi Slobode Gibanja
STL	-	eng. Stereolitography – format datoteke koji opisuje sirovu i nestrukturiranu trianguliranu površinu
Suma fitnesa	-	Ukupan zbroj svih dobrota pojedinaca u populaciji
URDF	-	eng. Unified Description Robot Format
X	[m]	Udaljenost koju je prešao robot po apsolutnoj *x*-osi okoline simulatora
XML	-	eng. eXtensible Markup Language
Y	[m]	Udaljenost koju je robot prošao po apsolutnoj *y*-osi okoline simulatora
Ψ	[°]	Kut između *x*-osi tijela robota i apsolutne *x*-osi okoline simulatora

SAŽETAK

U ovom diplomskom radu je navedena primjena evolucijskog algoritma u svrhu traženja optimalnog hoda četveronožnog robota. Naveden je postupak korištenja 3D CAD STL modela robota u URDF strukturi i korištenje URDF formata u Pybullet simulaciji. Objašnjene su osnove evolucijskih algoritama i primijenjen evolucijski algoritam u Python kodu. Izvedena je simulacija hoda i primjer rezultata evolucijskog algoritma, te njegova usporedba s ručno programiranim hodom robota. Na kraju su navedeni problemi Pybullet simulatora i evolucijskog algoritma koji su susretnuti tokom pisanja ovog rada i moguće nadogradnje, kako simulatora tako i evolucijskog algoritma.

Ključne riječi: Python, Pybullet, fizički simulator, URDF, evolucijski algoritam, robotski hod

SUMMARY

In this graduate thesis, the application of the evolutionary algorithms for the purpose of searching for the optimal gait of a four-legged robot is stated. The procedure for using the 3D CAD STL model of the robot in the URDF structure and the use of the URDF format in the Pybullet simulator are listed. The basics of evolutionary algorithms and the applied evolutionary algorithm in Python code are explained. A simulation of the gait and an example of the result of the evolutionary algorithm was performed, and its comparison with the manually programmed gait of the robot. At the end the problems of the Pybullet simulator and the evolutionary algorithm that were encountered during the writing of this thesis and the possible upgrade of both the simulator and the evolutionary algorithm are listed.

Key words: Python, Pybullet, physics simulator, URDF, evolutionary algorithm, robot gait

1. UVOD

Mobilni roboti pokretani nogama imaju velikog potencijala za primjenu, no većinom je za svakog robota potrebno posebno razvijanje specifičnog hoda. Velika većina vrsta takvih hodova je inspirirana iz prirode i reverzibilnim inženjerstvom prilagođena anatomiji robota. Postupak takve prilagodbe je uglavnom vremenski zahtjevan i kompliciran ako se radi o robotu sa više stupnjeva slobode gibanja. Također je iterativan, gdje je potrebno isprogramirati hod i testirati, ispraviti greške i dotjerati sekvence pokreta i testirati u krug. Fizički simulatori do veće mjere olakšaju postupak jer nije potrebno odmah graditi fizički model i na njemu testirati. Time je moguće uvidjeti neke probleme ili greške koje bi se predvidjele ako bi se odmah testiralo na fizičkom modelu i moguće je testirati robota u uvjetima gdje se robot neće oštetiti.

Uz simulator se javlja sljedeća ideja: dali bi računalo moglo samo pronaći optimalnu sekvencu pokreta za primjenu hoda robota? Ovakav zadatak treba uključivati neku vrstu umjetne inteligencije i velik potencijal imaju evolucijski algoritmi. Postavljanjem par uvjeta na hod robota i pravilnim odabirom parametara algoritma je moguće izvršiti zadatak. Tj. simuliranjem hoda robota pretražiti područje optimalnih pokreta robota i razviti ih u koristan i primjenjiv hod. Uz to smanjiti vrijeme i trud na proces programiranja hodanja robota. U daljnjem radu će se objasniti primjena evolucijskog algoritma na četveronožnom robotu s osam stupnjeva slobode gibanja.

2. KORIŠTENI PROGRAMSKI PAKETI I MODULI

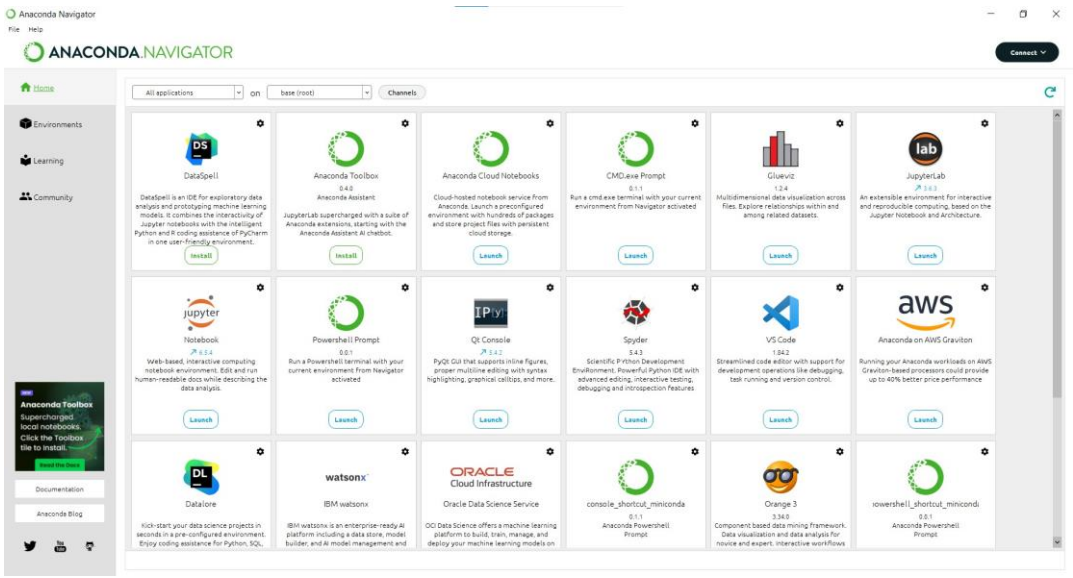
Za simuliranje hoda robota koriste se različiti programski programi i moduli, od kojega su svi otvorenog koda. Za izvršenje ovog zadatka i rada su se koristili Python u paketu Anaconda kao osnovna podloga programiranja, Pybullet kao simulator hoda i Visual Studio Code za sastavljanje strukture robota (URDF datoteke) u XML programskom kodu.

Također za potrebe funkcioniranja programa potrebni su python moduli:

- Numpy – modul za matematičke i numeričke operacije [1]
- Time – modul za vremenski povezane funkcije (mjerenje i pretvorba)[2]
- Threading – modul za konstruiranje dodatnih procesa koji se pokreću istovremeno s glavnim procesom [3]
- Matplotlib – modul za statičke, animirane ili interaktivne vizualizacije [4]
- Pybullet – modul fizikalnog simulatora [5]

2.1. Anaconda

Anaconda je distribucija Python i R programskih jezika. Otvorenog je koda i namijenjena je za znanstvenu primjenu u računalstvu. Cilj Anaconde je pojednostavniti potrebne pakete instalacije i pokretanja, te jednostavno korisničko sučelje za programiranje. Paket je prikladan za Windows, Linux i macOS.[6]

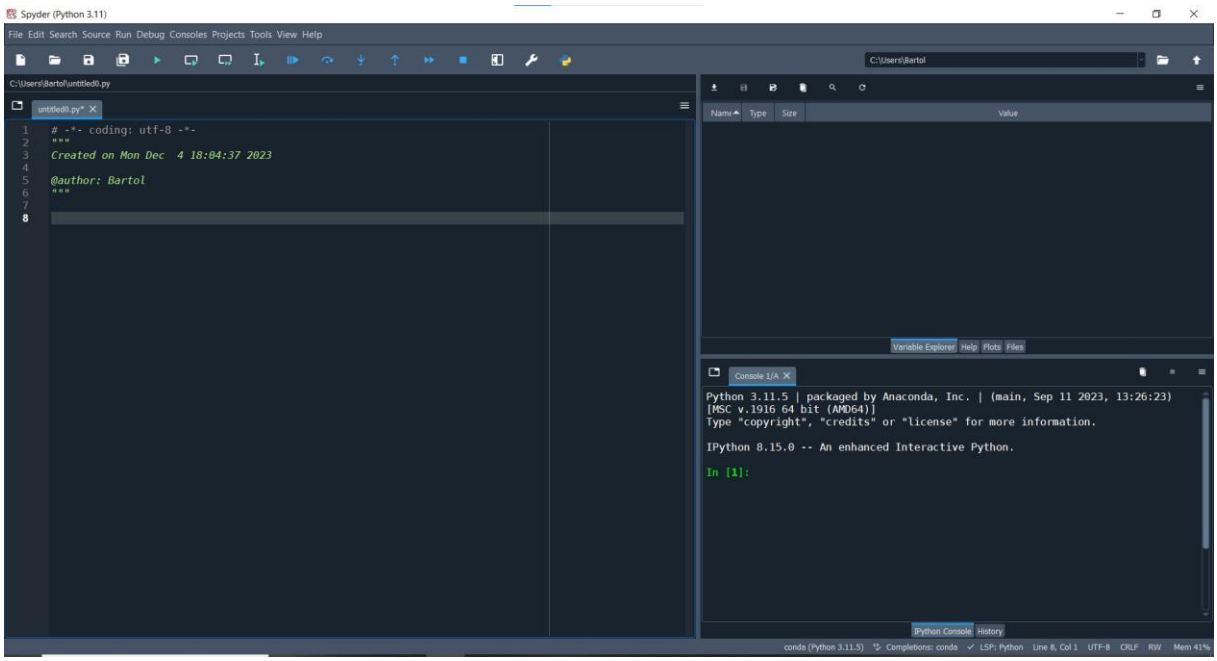


Slika 2.1. Sučelje Anaconde [6]

Anakondino glavno sučelje je Anaconda Navigator i u njemu se nalaze razne aplikacije vezane uz programiranje i vizualizaciju podataka. Za koristi ovoga rada biti će potrebna samo aplikacija Spyder.

2.1.1. Spyder

Spyder je korisničko grafičko sučelje za programiranje u Pythonu. Spyder (eng. The Scientific Python Development Enviroment) je IDE (eng. Integrated Develpoment Enviroment) softverska aplikacija koja pruža sveobuhvatne mogućnosti za razvoj softvera. Razvijen je za znanstvenike, inženjere i analitičare podataka, sastoji se od nekoliko komponenti kao urednik skripte koda (eng. editor), preglednik varijabli (eng. variable explorer), jezgra programa (eng. kernel), plot preglednik i pomoć pri kodiranju (eng. help). [7]

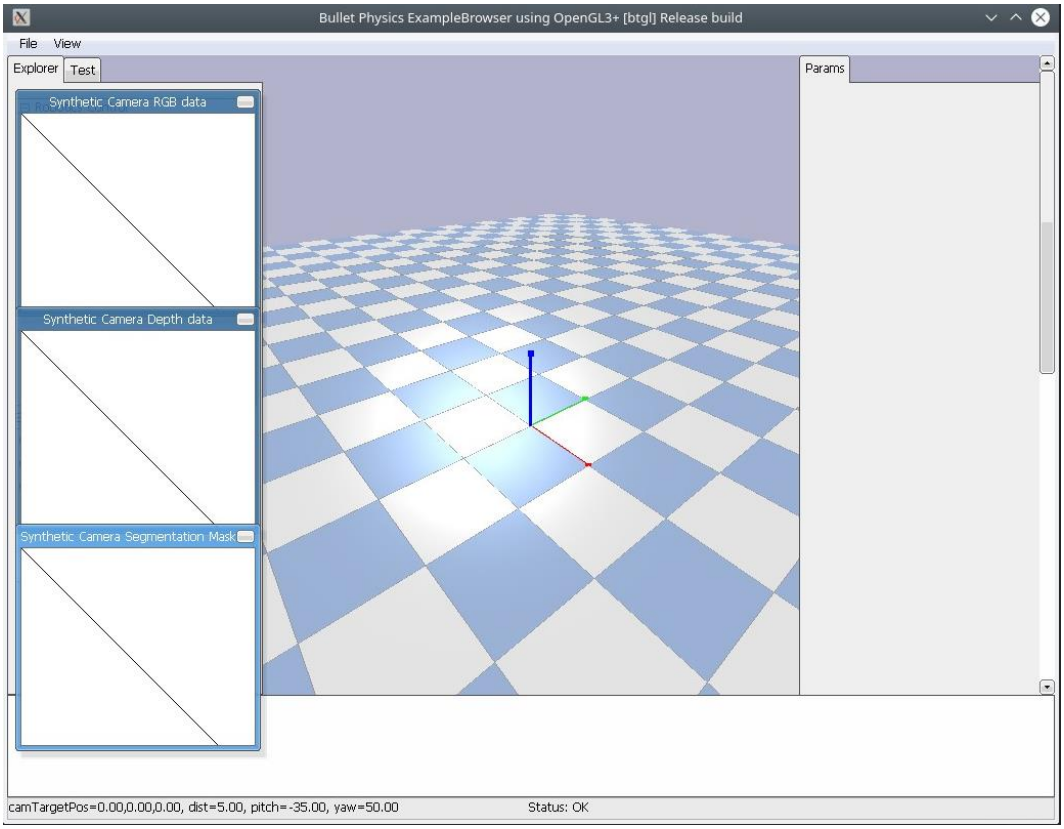


Slika 2.2. Sučelje Spydera [7]

U uredniku skripte se piše kod dok najveću pomoć nudi preglednik varijabli gdje se mogu vidjeti sve informacije vezane uz varijable programa. Kernel programa omogućuje da se mogu pokretati naredbe i vidjeti greške prilikom pokretanja. Kernel ne označuje samo liniju koje je krivo napisana već i daje opis greške koja se događa, kao tipfeler, množenje matrica krivih dimenzija, krivo korištenje određene naredbe itd.

2.2. Pybullet

Pybullet je fizički simulator otvorenog koda koji se temelji na Python programskom jeziku. Razvijen je od strane razvojnog programera Erwin Coumansa i koristi se kao fizikalni simulator u razvoju strojnog učenja u robotici, stvaranje igara ili realističnih vizualnih efekata. Pybullet modul omogućava analizu kolizija, direktne i inverzne kinematike i dinamike robota, učenje lokomocije neuronskim mrežama ili evolucijskim algoritmima, kao podrška u radu s virtualnom stvarnošću...[5] Fokus je na prijenosu iz simulacijskog u realno okruženje.



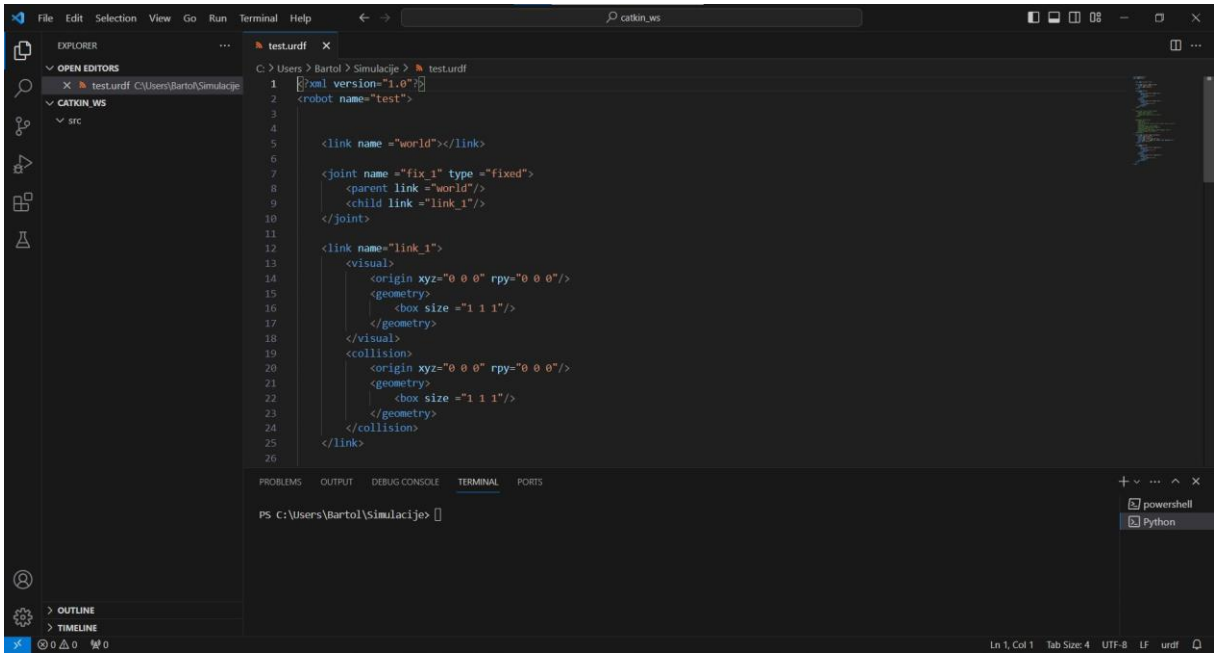
Slika 2.3. Sučelje Pybullet-a [5]

Sučelje ne nudi opcije za rad već se sve predprogramira u python skripti.

Za uvođenje strukture robota koristi se format datoteke .urdf (Unified Description Robot Format), .sdf (Structure-Data File) ili .mjcf (MuJoCo fomat). A za potrebe ovog rada će se koristiti .urdf format.

2.3. Visual Studio Code

Visual Studio Code je urednik koda, no nije prilagođen koliko i Spyder. Najveća prednost VS Code-a naprema Spyderu je ta da se može programirati u više jezika (Java, Python, XML...).[8] Te će se u svrhe ovog rada koristiti samo za uređivanje .urdf datoteke koja je napisana u XML kodu. XML označuje eng. Extensible Markup Language.



Slika 2.4. Sučelje Visual Studio Code-a [8]

3. EVOLUCIJSKI ALGORITMI

Evolucijski algoritmi (EA) spadaju pod granu znanosti o računalima, ne u biologiji. Neki nazivi i funkcionalnosti algoritma su inspirirane iz biologije. Evolucijski algoritmi spadaju u grupu algoritama umjetne inteligencije i sastoje se od raznih podgrupa, kao genetski algoritmi, genetsko programiranje, evolucijsko programiranje... EA su stohastički algoritmi, jer se rezultati temelje na radu s populacijama rješenja. To znači da ako već nemamo rješenje nećemo znati dali je algoritam našao točno rješenje ili je u nekom lokalnom optimumu. [9]

Temelji se na nekim Darwinovim teorijama[9]:

1. Preživljavanje najboljeg – okolina ima ograničene resurse i može opskrbiti ograničen broj jedinki rješenja. Jedinke imaju osnovne instinkte za reprodukciju i preživljavanje, te najuspješnije jedinke imaju veću šansu za preživljavanje i reprodukciju.
2. Raznovrsnost pokreće promjenu – ponašanja i/ili fizičke razlike koje predstavljaju reakciju na okolinu, djelomično nasljeđem i djelomično tijekom razvoja. Ako ta obilježja rezultiraju većom šansom jedinke za preživljavanje i reprodukciju, a mogu se naslijediti, tada će se obilježja učestalije pojavljivati u nadolazećim generacijama.

Motivacija za korištenje EA se sve češće nailazi pošto vrijeme potrebno za detaljnu analizu procesa se smanjuje i kompleksnost problema se povećava. Nije jednostavno naći točno rješenje ili optimum i potrebne su robusnije metode rješavanja problema.

Prije prelaska na glavne dijelove EA potrebno je znati par osnovnih pojmova korištenih u algoritmu:

- Pojedinac (genom) – jedno moguće rješenje problema. Obično je prikazan u algoritmu kao vektor brojeva (binarnih ili decimalnih) ili slova.
- Populacija – skup pojedinaca u određenoj generaciji. Ako je pojedinac vektor, tada je populacija matrica brojeva ili slova. Važno je da broj pojedinaca bude paran broj.
- Roditelji – skup pojedinaca koji su odabrani iz populacije za proces križanja i mutacije. Obično je iste veličine kao i populacija.
- Potomci – skup pojedinaca dobivenih iz križanih i mutiranih roditelja. Potencijalni novi pojedinci za novu generaciju populacije.
- Fitnes funkcija – svakom pojedincu se računa njegov fitnes (kvaliteta ili dobrota) rješenja. S obzirom na vrijednost fitnesa moguće je odrediti kolika je uspješnost rješavanja, vjerojatnost prelaska iz populacije u roditelje, opstanak u sljedeću

generaciju... Funkcija može poprimati razne oblike i ne postoji jedna robusna i fiksna za sve probleme. Ovisno o problemu koji se rješava fitnes treba minimizirati ili maksimizirati, a fitnes funkcija mora biti mjerodavna za sve pojedince. Tako npr. fitnes funkcija može biti suma prijeđenog puta ako tražimo minimalnu rutu odredišta ili zbroj kraljica koje se ne križaju na šahovskoj ploči u igri 8 kraljica itd.

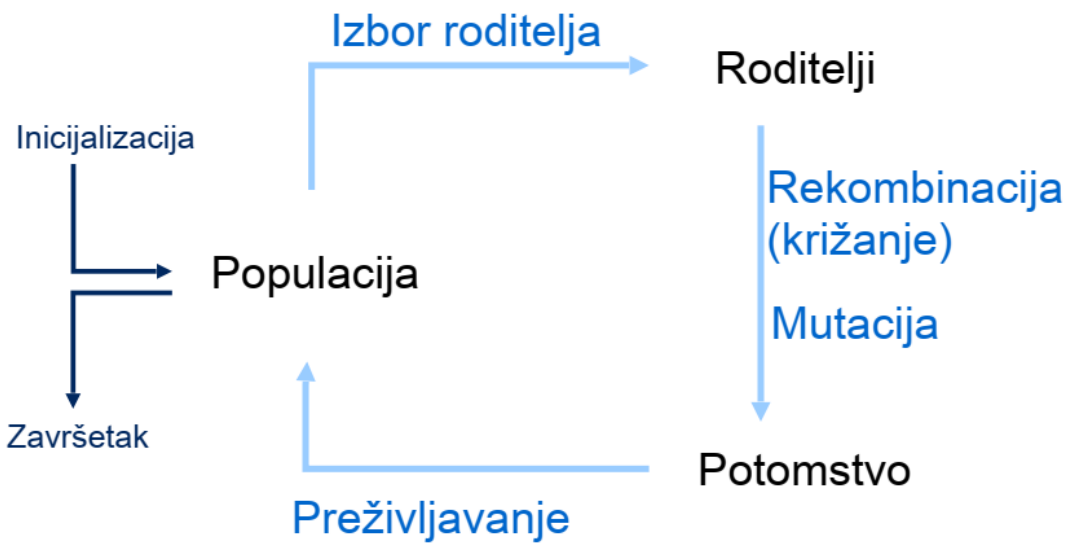
- Generacija – EA je iterativni postupak, započinje od nulte generacije i svaka iteracija rezultira s novom populacijom i novom generacijom.

3.1. Glavni dijelovi evolucijskog algoritma

Glavni djelovi EA se mogu podijeliti na [9]:

1. Inicijalizacija algoritma
2. Izbor roditelja
3. Križanje (Rekombinacija)
4. Mutacija
5. Evaluacija (preživljavanje)
6. Prekid algoritma

Shema osnovnog rada dijagrama je vidljiva na slici (3.1.).



Slika 3.1. Dijagram toka evolucijskog algoritma [9]

3.1.1. Inicijalizacija algoritma

Inicijalizacija algoritma se obično izvodi stohastički, da se osigura ravnomjeran raspored jedinki i miješanje elemenata gena jedinki u populaciji. Tj. za početak algoritma i prvu populaciju stvaramo s nasumičnim vrijednostima ograničenim samo granicama odabranog skupa brojeva.
Također možemo započeti s poznatim vrijednostima, no većinom se za početak uzimaju nasumične vrijednosti.

3.1.2. Izbor roditelja

Roditelji su pojedinci iz populacije koji su odabrani za križanje i mutaciju, te krajnju pretvorbu u potomstvo. Roditelje se bira uglavnom probabilistički. Mogu se birati ruletnim pravilom da im se dodjeljuje vrijednost različita od fitnesa i da usporedbom zadanih vrijednosti i izračunatih fitnesa biramo roditelje tako da pojedinci s boljim fitnesom imaju veću šansu da postanu roditelj. Ili drugim mehanizmima kao turnirska selekcija gdje se dvije jedinke natječu tko će biti roditelj prema fitnesu. Stohastička priroda izbora pomaže pri izbjegavanju lokalnog optimuma.

3.1.3. Križanje (Rekombinacija)

Izabrane roditelje križamo, zbog toga je imperativ da broj jedinki u populaciji bude paran broj. Susjedne jedinke tvore parove i s obzirom na vjerojatnost križanja, križamo pojedince. Tj. mijenjamo im dijelove s obzirom na nasumično mjesto križanja, primjer je ako imamo pojedince AAAA i BBBB, i njihovo mjesto križanja je na pola, rezultat je AABB i BBAA. Vjerojatnost križanja se zadaje unaprijed i za svaki par populacije provjeravamo dali se odabrani par križa ili ne.
Križanje spaja informacije od roditelja u potomke i nada je da će neki pojedinac biti bolji od prethodnih roditelja.

3.1.4. Mutacija

Nakon križanja slijedi mutacija, mala slučajna promjena elementa u pojedincu. Može garantirati povezanost prostora traženja, kod genetskih algoritama zadužena je za očuvanje raznovrsnosti, kod evolucijskog programiranja je jedini operator pretrage, a kod genetskog programiranja se gotovo i ne koristi. Vjerojatnost mutiranja se zadaje unaprijed.

3.1.5. Evaluacija (Preživljavanje)

Zbog fiksne veličine populacije u većini EA-a, potreban je mehanizam za prijelaz roditelja i potomaka u novu populaciju. Postupak je često deterministički, ovisi o postignutom fitnesu pojedinca, rangiraju se pojedinci iz potomaka i roditelja i odabiru najbolji. Ili temeljem starosti – generira se potomaka koliko i roditelja, te se zamjenu sva mjesta u novoj populaciji sa potomcima. Ponekad se i kombiniraju metode.
Evaluacija omogućava da fitnes stalno raste (kod maksimizacije, kod minimizacije mora padati) ili stagnira.

3.1.6. Prekid algoritma

Prekidanje algoritma se postiže ako smo dostigli neki uvjet ili prešli neki granicu. Uglavnom su razlozi prekida algoritma prijeđen zadan broj generacija, dosegnut dovoljan fitnes pojedinca, ako raznovrsnost populacije padne ispod određene razine ili ako određen broj generacija završi u stagnaciji fitnesa.

3.2. Prednosti evolucijskih algoritama

Prednosti EA:

- Implicitni paralelizam – evolucijski algoritam pretražuje prostor rješenja za sve parametre odjednom.
- Robusnost – algoritam je jednostavan za primjenu i bez većih promjena za druge probleme.
- Oponašanje postupaka iz prirode – evolucijski algoritmi su dobili inspiraciju iz evolucije biološkog života. Kako određena vrsta organizma promjeni svoju fiziologiju s obzirom na promjenu okoline, inspiracija je dali je moguće primijeniti isti ako ne i sličan proces za evoluciju potencijalnih rješenja za tehničke probleme.

- Jednostavno prilagodba i hibridizacija – jednostavno je prilagoditi algoritam s postojećim matematičkim modelima, stvarnim procesima i simulacijama. Te je jednostavno povezati s drugim postupcima optimiranja.
- Pronalazak globalnog ekstrema – ne ovisi o vrsti zadatka i nije potrebno proširivanje postupka, već samom iteracijom dolazimo do rješenja. Jedna je od temeljnih odlika EA i zaslužna je stohastička priroda algoritma.
- Dobra moguća rješenja uz prihvatljiv utrošak vremena – ponajviše za probleme koji nemaju točno rješenje ili je potreban utrošak puno vremena s deterministčkim metodama rješavanja.
- Široko područje primjene – moguće je primijeniti od oblikovanje topologije konstrukcija i robota, određivanja ruta u logistici do optimiranja efikasnih kretnji robota itd.

3.3. Nedostaci evolucijskih algoritama

Nedostaci EA:

- Vrijeme procesiranja – nekada vrijeme pretrage područja rješenja je predugo
- Preuranjena konvergencija – znači da je populacija ranije konvergirala pri nekom rješenju koje nije u globalnom optimum i ograničen je prostor pretrage.
- Problematičan izbor funkcije fitnesa – izbor funkcije ovisi o problemu kojeg se rješava i o načinu kako bi smo prikazali fitnes nekog pojedinca. Potrebno je malo "znanja" unaprijed kako bi smo znali koji pojedinci su dobri, a koji loši, kako bi smo ih pravilnim izborom funkcije fitnesa mogli razlikovati.
- Nedostatak teorijske podrške – evolucijski algoritmi ne garantiraju sigurnost rješavanja zadanog problema, zbog toga se i u industrijskim prostorima većinom preferiraju deterministčke metode.

4. MODEL ROBOTA

Pybullet simulator za model robota prihvaća .urdf format datoteke. Stoga je potrebno napraviti takvu vrstu datoteke. To je moguće pomoću ROS (Robot Operating System) gdje bi trebali instalirati ROS i u bilo kojem uređivaču teksta upisati XML kod modela robota. Nakon toga ubaciti datoteku u ROS parser da bi smo iz upisanog koda dobili URDF format datoteke.

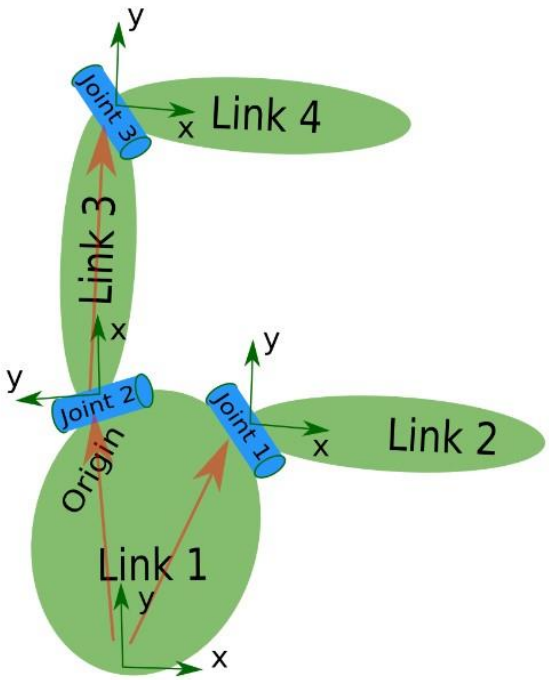
No postoji i drugi način, kako bi smo izbjegli instaliranje još dodatnih programa moguće je iskoristiti već stvoren primjer URDF datoteke i skinuti je na računalo. Izbrisati postojeći kod i upisati vlastit.

Na internetskoj stranici [10] je primjer URDF datoteke R2D2 robota poznatog iz Star Wars franšize. Skidamo datoteku i brišemo kod primjera nakon čega pišemo vlastit, potrebna nam je samo datoteka. XML kodom u URDF datoteci možemo programirati samo jednostavne 3D oblike pa je potrebno konstruirati model robota u nekom od 3D softveru. Možemo koristiti velik raspon 3D softvera, minimalni uvjet je da mogu sačuvati model u .stl (eng. Standard triangle Language) datoteci. Iznimka je softver Solidworks koji ima ugrađenu funkciju da spremi cijeli model robota u već sređenu URDF datoteku. U ovom slučaju su se STL datoteke dijelova robota, stvorene u Autodesk Fusion CAD programu, pozivale u funkciju iz URDF datoteke.

Za programiranje URDF modela u XML kodu potrebno je poznavanje funkcioniranja i grananja dijelova koda, koje je moguće pronaći na ROS-ovim internetskim stranicama. [11]

4.1. URDF struktura

Stuktura robota u URDF datoteci se može prikazati kao struktura drva. Gdje se iz debla (karike robota) granaju manje grane (noge i zglobovi robota). Na slici (4.1.) je prikazan primjer strukture robota.[11]

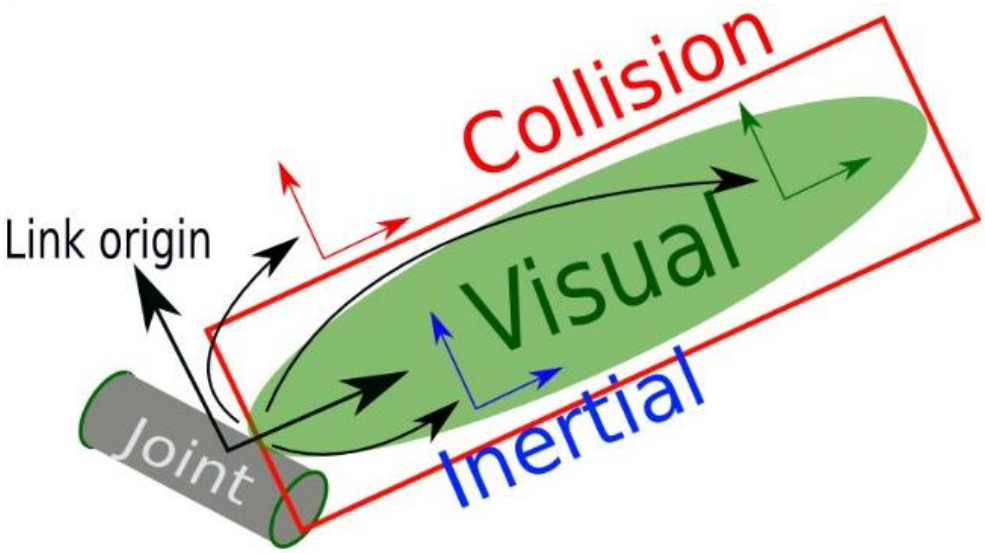


Slika 4.1. Grananje URDF strukture robota [11]

Karike (eng. links) su fizički dijelovi robota, dok su zglobovi (eng. joints) poveznice između karika i daju im mogućnosti gibanja. Pri povezivanju karika i zglobova potrebno je paziti na strukturu drva jer se povezivanje vrši kao na primjeru karika 1 – zglob 1 – karika 2 – zglob 2... Također jedna karika može se povezati na više zglobova i onda na više karika, pri čemu takvo povezivanje nazivamo otvorenim kinematskim lancem. Svaki puta kada povezujemo kariku u zglobu koji ih povezuje definiramo koja karika je roditelj (eng. parent), a koja karika je potomak (eng. child). To je važan detalj jer ako pokušamo spojiti potomke natrag na glavnu kariku tvorimo zatvoren kinematski lanac i on nije podržan URDF datotekom, te rad s takvom datotekom neće biti moguć. Iako nije moguće definirati zatvorene kinematske lance direktno u datoteci moguće je zaobići taj problem tako da se robot djelomično definira u datoteci s otvorenim kinematskim lancima u URDF datoteci, a u Pybullet-u postavi zaseban zglob koji će zatvoriti taj lanac. No u ovom radu će biti potrebni samo otvoreni kinematski lanci koji su prikazani primjerom na slici (4.1.).

4.1.1. Karike

Karike opisuju kruta tijela s vizualnim obličjima, kolizijskim obličjem i svojstvima, te inercijom. Karikama opisujemo fizičke dijelove robota kao tijelo i noge. Minimalno je potrebno definirati ime karike. No da bi smo prikazali CAD modelirane dijelove i postigli što točniju simulaciju potrebno je definirati i tri glavna dijela karike. A to su vizualna, kolizijska i inercijska svojstva kako je prikazano na slici (4.2.). Svaka karika ima tri koordinatna sustava koja odgovaraju pripadajućim svojstvima, a to znači da možemo imati ishodište inercije, kolizije i vizualne značajke na različitim mjestima. [12]



Slika 4.2. Prikaz karike i pripadajuća tri koordinatna sustava [12]

Razdvojeni koordinatni sustavi su dobra stvar ako želimo smanjiti napor procesuiranja računala tokom rada simulacije. U vizualni dio postavimo mesh CAD modela odgovarajućeg dijela. A u kolizijski dio možemo postaviti drukčiji i jednostavniji oblik u poziciju koja nam najviše odgovara. Tako računalo ne treba raditi izračune kolizije za komplicirane oblike već se koristimo približan i jednostavniji oblik s manje izračuna i zadovoljavajućom kolizijom.

Slično kao i kod strukture karika i zglobova, i struktura karike je razgranata te je na slici (4.3.) prikazan primjer koda gdje se nalaze glavni dijelovi karike.

```
<link name="my_link">
  <inertial>
    <origin xyz="0 0 0.5" rpy="0 0 0"/>
    <mass value="1"/>
    <inertia ixx="100"  ixy="0"  ixz="0" iyy="100" iyz="0" izz="100" />
  </inertial>

  <visual>
    <origin xyz="0 0 0" rpy="0 0 0" />
    <geometry>
      <box size="1 1 1" />
    </geometry>
    <material name="Cyan">
      <color rgba="0 1.0 1.0 1.0"/>
    </material>
  </visual>

  <collision>
    <origin xyz="0 0 0" rpy="0 0 0"/>
    <geometry>
      <cylinder radius="1" length="0.5"/>
    </geometry>
  </collision>
</link>
```

Slika 4.3. XML kod jedne karike [12]

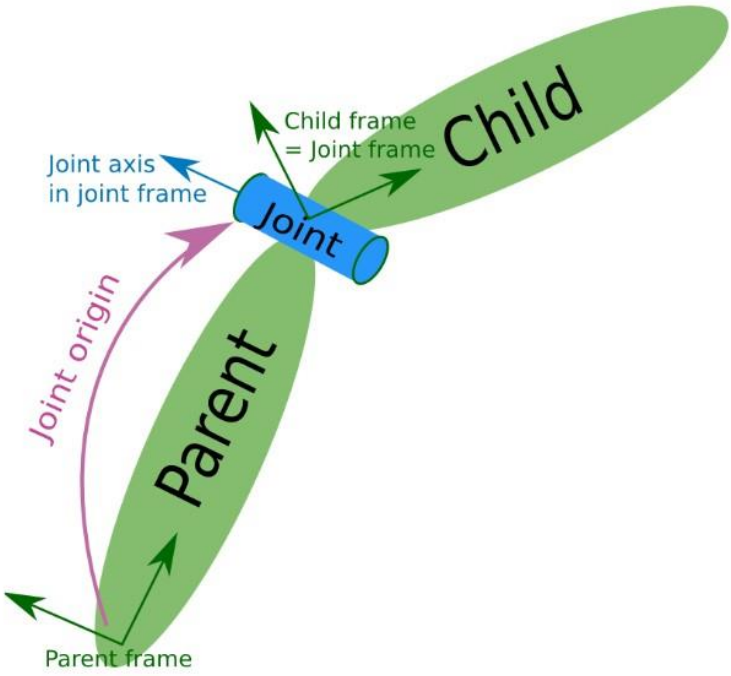
Pod inercijskom granom možemo upisat ishodište koordinatnog sustava (poziciju i orijentaciju), masu dijela (čija sila djeluje u ishodištu inercijskog koordinatnog sustava) i momente inercije: I_{xx}, I_{xy}, I_{xz}, I_{yy}, I_{yz} i I_{zz}. Momenti inercije imaju djelovanje u inercijskom koordinatnom ishodištu sustava.

Pod vizualnom granom možemo upisati ishodište koordinatnog sustava, geometriju dijela i materijal (boju karike). Pod geometrijom možemo dodati opće 3D oblike kao kvadar, cilindar i sferu. Iako je moguće, vrlo je vremenski i memorijski zahtjevno prikazivanje nekih kompliciranijih oblika upotrebom tijela primitiva. Zato uz njih postoji i funkcija mesh, s kojom definiramo lokaciju jedne STL datoteke koja sadrži već oblikovan dio robota. S mesh funkcijom jednostavno pozivamo predefinirane 3D oblik koji smo oblikovali u nekom CAD softveru. Geometrija pod vizualnim dijelom će biti vidljiva u simulaciji.

Pod kolizijskom granom ponovno definiramo poziciju i orijentaciju koordinatnog sustava i geometriju. S time da isto navedeno za vizualni dio o geometriji vrijedi i za kolizijski. Oblik kojeg definiramo neće biti vidljiv u simulaciji, no od velike je važnosti jer s definiranim oblikom smo u kontaktu s okolinom, ne s vizualnim.

4.1.2. Zglobovi

Zglobovi (eng. joints) opisuju kinematiku i dinamiku gibanja jedne karike naspram druge i granice gibanja zgloba. S zglobom naznačavamo kako i na kojem mjestu će se određena karika gibati, primjerom prikazano na slici (4.4.). Također zglobovi nisu vidljivi niti imaju mogućnosti biti u koliziji s drugim objektima i karikama u simulaciji. [13]



Slika 4.4. Prikaz zgloba i pripadajućeg koordinatnog sustava [13]

Primjer koda zgloba je prikazan na slici (4.5.).

```
<joint name="my_joint" type="floating">
   <origin xyz="0 0 1" rpy="0 0 3.1416"/>
   <parent link="link1"/>
   <child link="link2"/>

   <calibration rising="0.0"/>
   <dynamics damping="0.0" friction="0.0"/>
   <limit effort="30" velocity="1.0" lower="-2.2" upper="0.7" />
   <safety_controller k_velocity="10" k_position="15" soft_lower_limit="-2.0" soft_upper_
limit="0.5" />
</joint>
```

Slika 4.5. XML kod zgloba [13]

Zglobni element je također strukture drva i nadalje su opisani glavni elementi.

Za glavnu stavku u kodu zgloba potrebno je označiti ime (eng. name) i tip (eng. type) zgloba. Ime biramo sami i preporuča se malo početno slovo, te mora biti jedna riječ ili više riječi spojene podvučenom crtom, npr. base_link. Tip zgloba biramo po potrebi kretnji robota. Rotacijski (eng. revolute) je zglob koji se rotira oko označene osi i koristi se kada je potrebno točno okretno pozicioniranje. Jer naspram njega kontinuirani (eng. continuous) zglob se koristi kada nam je potrebno upravljati brzinom vrtnje. Linearni (eng. prismatic) zglob omogućuje linearne kretnje karike po odabranoj osi. Fiksni (eng. fixed) zglob se koristi za spajanje dvije karike bez mogućnosti pomaka. Plutajući (eng. floating) zglob omogućuje slobodno kretanje u svih šest stupnjeva slobode gibanja. Ravninski (eng. planar) zglob omogućuje slobodu gibanja okomito na odabranu os.

Zglobovi imaju samo jedan koordinatni sustav i isto kao i u slučaju karika definira se pozicija i orijentacija ishodišta.

Roditelj je oznaka koja govori koja karika je iznad u strukturi drva. Tj. ona karika koja prethodi vezivanju nove. U roditelju je potrebno navesti ime karike koja će biti roditelj.

Potomak je oznaka koja govori koja karika je ispod u strukturi drva, tj. označava koja karika se vezuje za prethodnu – roditelja. U potomku je potrebno navesti ime karike koja će postati potomak.

Os (eng. axis) označava vektor od tri broja koji predstavljaju osi x, y i z. Brojevi u vektoru mogu varirati od 0 do 1 čime variramo nagib željene osi s kojom ćemo gibati kariku potomka. Npr. vektor [0 1 0] kod rotacijskog i kontinuiranog zgloba predstavlja os oko koje će se karika okretati, a to je y os. Kod linearnog zgloba to je os po kojemu se vrši linearno gibanje, dok za ravninski zglob je normala na ravninu po kojoj je omogućeno kretanje. Fiksni i plutajući zglob ne koriste funkciju osi.

Granica (eng. limit) označava granice gibanja, brzine i sile zgloba. Sastoji se od donje (eng. lower) i gornje (eng. upper)granice gibanja, brzine (eng. velocity) i sile/momenta (eng. effort) s kojim je moguće pokretati zglob. Služi samo u rotacijskim i linearnim zglobovima.

Mimikrija (eng. mimic) je opcionalna funkcija i omogućuje da zglob imitira pokrete drugog zgloba.

Dinamika (eng. dynamics) je također opcionalna funkcija i omogućuje postavljanje nekih fizikalnih svojstava zgloba, kao prigušenja (eng. damping) i trenja (eng. friction) zgloba.

Kalibracija (eng. calibration) je opcionalna funkcija i služi za određivanje referentne pozicije zgloba kojom kalibriramo apolutnu poziciju zgloba.

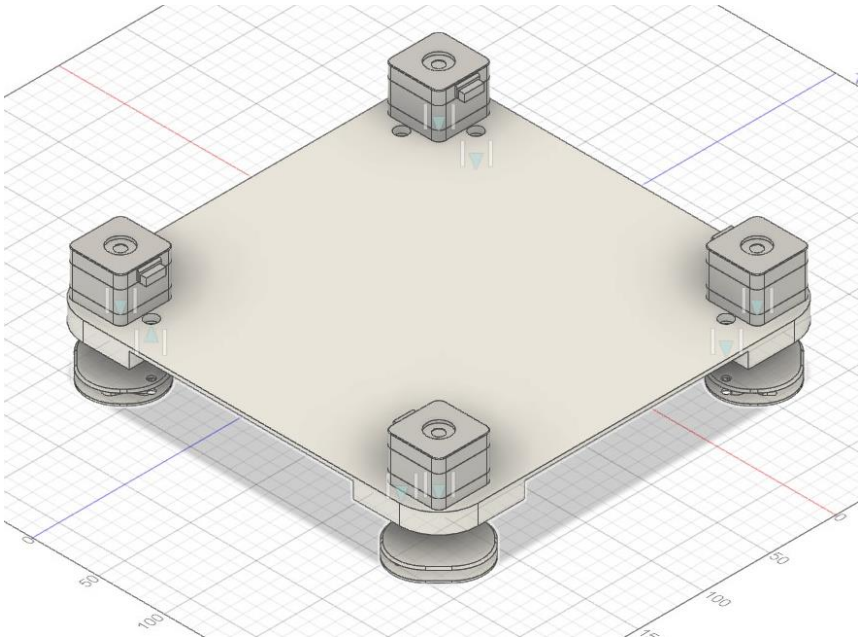
Sigurnosni kontroler (eng.safety controller) je opcionalna funkcija kojom možemo postaviti "meke" granice gibanja zgloba.

4.2. URDF struktura robota

Struktura robota se sastoji od tri CAD sklopa: tijelo, noga i prst robota. Sva tri sklopa su eksportana iz Fusion CAD programa u STL formatu i koriste se za karike pomoću funkcije mesh.

U sklop koji planiramo ekspotirati u STL format sklapamo sve dijelove koji su nepomični u sklopu prilikom korištenja. Jer URDF model uzima kariku kao jedno kruto tijelo i na njemu se ne mogu micati dijelovi. Inače bi trebali dodavati više karika i zglobova. Isti razlog tomu je što imamo tri STL formata jer ako bi smo pretvorili cijelog robota u jedan STL format naš robot se ne bi mogao micati jer bi bio jedno kruto tijelo bez naznačenih zglobova.

Na slici (4.6.) je prikazan CAD model sklopa tijela robota.



Slika 4.6. CAD model tijela robota

Na sklopu tijela se nalazi glavna konstrukcija tijela, četiri motora, donje pločice za nošenje noge i vijci.

Na slici (4.7.) je opisana karika tijela i glavni dijelovi karike kao vizualni, kolizijski i inercijalni dio karike.

```
<link name="base_link">
    <visual>
        <origin xyz = "0 0 0" rpy ="0 0 0"/>
        <geometry>

            <mesh filename="package://STL_djelovi/Tijelo.stl" scale="0.001 0.001 0.001"/>

        </geometry>
        <material name="sivo">
            <color rgba="0.5 0.5 0.5 1"/>
        </material>
    </visual>

    <collision>
        <origin xyz = "0 0 0" rpy ="0 0 0"/>
        <geometry>
            <box size="0.3 0.3 0.015"/>
        </geometry>
    </collision>

    <inertial>
        <mass value="2.46"/>
        <origin xyz="0 0 0.01" rpy="0 0 0"/>
        <inertia ixx="0.034" ixy="0" ixz="0" iyy="0.039" iyz="0" izz="0.067"/>
    </inertial>
</link>
```

Slika 4.7. XML kod karike tijela

Pod vizualnim potrebno je samo označiti ishodište (eng. origin) koordinatnog sustava vizualnog dijela tijela, pod geometrijom u *mesh* naredbi navesti lokaciju STL datoteke. Te ju je potrebno skalirati. Prilikom eksportiranja CAD modela u STL model potrebno je pripaziti u kojim se mjernim jedinicama konstruiralo, jer STL model pamti samo brojke ne i mjerne jedinice. Dok Pybullet pretpostavlja da su sve mjerne jedinice po SI sustavu mjernih jedinica (u ovom slučaju metri) i učitava brojke iz STL modela (konstruirane u milimetrima). Time bi smo predstavili milimetre kao metre i zato je potrebna pažnja i pretvorba mjernih jedinica skaliranjem modela naredbom *scale* u sve tri dimenzije. Pod vizualnim dijelom još je postaljena boja pod granom materijala naredbom *color*. naredbom za boju postavljamo s četiri broja koja predstavljaju redom udio: crvene, zelene, plave i alfa. Prva tri broja predstavljaju udio svoje boje od 0 do 255

do alfa predstavlja prozirnost u rasponu od 0 do 1. Razlog je prepoznavanja različitih karika i preklapanja tijekom programiranja samog URDF modela robota.

Za kolizijski dio također postavljamo ishodište i geometriju koja će biti u kontaktu s okolinom u simulaciji. Nažalost zbog tvrde granice mogućeg broja faceta u Pybulllet simulatoru potrebno je pojednostaviti STL model, tj. smanjiti broj faceta modela. Moguće je pojednostavniti model pomoću CAD programa s kojim su se i konstruirali modeli. Ako program sadrži alate za uređivanje *mesh* objekata, tada je moguće jednostavno homogeno pojednostavljenje broja faceta ili ručni odabir faceta koje treba otkloniti ili spojiti u jedno. No ako oblici mogu biti približno opisani primitivnim oblicima kao cilindri ili kvadri. Na slici (4.7.) je pod geometriju kolizijskog dijela postavljen kvadar (eng. box) s ishodištem u centru volumena dok se duljine označuju s tri broja po tri osi (po x, y i z).

Pod inercijom možemo upisati sve realne podatke bez pojednostavljenja. Iako je kolizijski model pojednostavljen, za sile inercije i masu se koriste podaci od originalnih volumena. Tako je u inercijskoj grani potrebno samo upisati masu karike (eng. mass), centar mase (eng. origin) i vrijednosti matrice inercije (eng. inertia). Vrijednosti matrice se mogu izračunati ručno, no pošto su kompliciraniji oblici tada su i kompliciraniji izračuni. Jednostavnije i sigurnije je izračunati ih pomoću CAD programa i samo upisati dobivene vrijednosti na potrebna mjesta (I_{xx}, I_{xy}, I_{xz} , I_{yy}...).

Na slici (4.8.) je prikazan XML kod zgloba, točnije prednjeg desnog ramena robota.

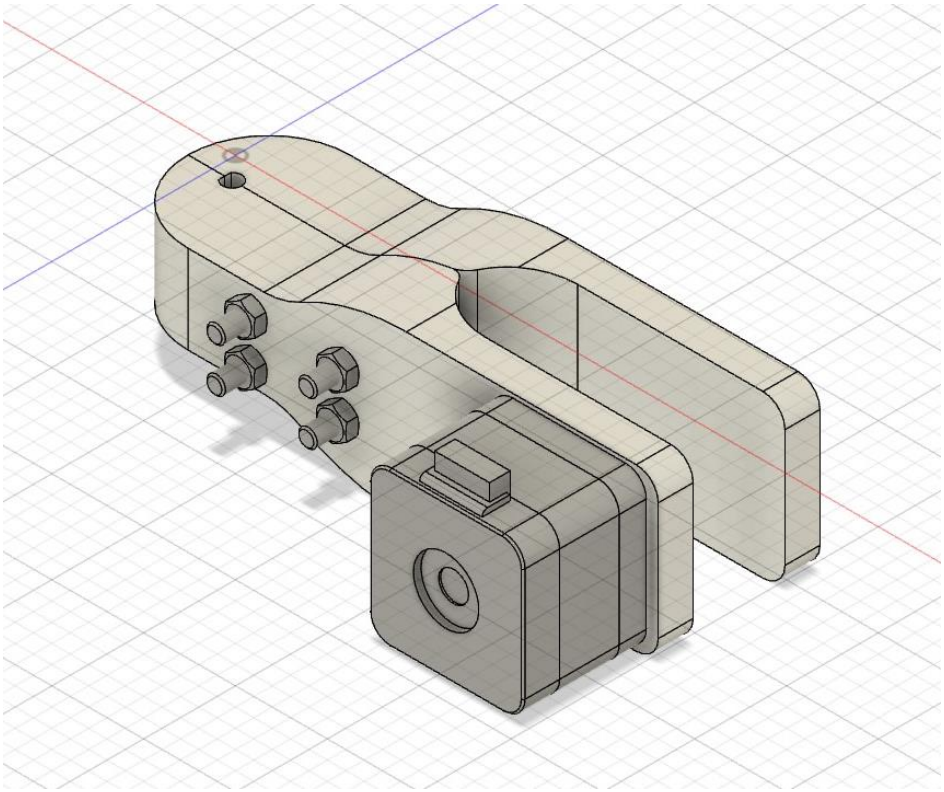
```
33    <joint name ="desno_rame_1" type ="revolute">
34        <origin xyz ="0.121 -0.121 -0.01134" rpy ="0 0 -0.7854"/>
35        <parent link ="base_link"/>
36        <child link ="desna_noga_1"/>
37        <axis xyz ="0 0 1"/>
38        <limit lower ="-0.768" upper ="0.768" effort ="10" velocity ="10"/>
39    </joint>
```

Slika 4.8. XML kod zgloba prednjeg desnog ramena robota

Zglob je rotacijski i mora sadržavati vlastito ime. Za rameni zglob potrebno je navesti centar (eng. origin) gibanja, označiti roditelja (eng. parent) i potomka (eng. child) u poveznici karika. U ovom slučaju je tijelo robota roditelj, a noga je potomak. Također se mora naznačiti oko koje osi se gibanje odvija, potrebno je upisati tri broja, po jedan za svaku os. Osim toga je potrebno

naznačiti granice zgloba (eng. limit). Donju granicu pomaka (eng. lower) i gornju granicu pomaka (eng. upper) u radijanima, maksimalni moment u zglobu (eng. effort) u njutn metrima i maksimalnu brzinu zgloba (eng. velocity) u radijanima po sekundi. Za ostale zglobove ramena se ponavlja kod osmi promjena imena, ishodišta kretanja i imena roditelja i potomaka povezani s zglobom.

Na slici (4.9.) je prikazan CAD model noge robota.



Slika 4.9. CAD model noge robota

U sklopu noge su dvije konstrukcije noge, motori za pokretanje prsta i vijci za stezanje nogu i pričvršćivanje motora na nogu.

Na slici (4.10.) je prikaza XML kod karike prednje desne noge.

```
<link name="desna_noga_1">
    <visual>
        <origin  xyz="0 0 0" rpy ="0 0 0"/>
        <geometry>
            <mesh filename="package://STL_djelovi/Noga.stl" scale="0.001 0.001 0.001"/>
        </geometry>
        <material name="crno">
            <color rgba="0 0 0 1"/>
        </material>
    </visual>

    <collision>
        <origin  xyz="0.096 0 0" rpy="0 0 0"/>
        <geometry>
            <box size="0.108 0.044 0.044"/>
        </geometry>
    </collision>

    <inertial>
        <mass value="0.67"/>
        <origin xyz="0.097 -0.003 -0.012" rpy="0 0 0"/>
        <inertia ixx="0" ixy="0" ixz="0" iyy="0.001" iyz="0" izz="0.001"/>
    </inertial>
</link>
```

Slika 4.10. XML kod karike noge robota

XML kod nogu nema velike razlike od koda tijela. Mijenja se ime karike, STL model, boja, približan kolizijski oblik i vrijednosti mase i inercije. Sve ostalo je po programirano po istoj shemi.

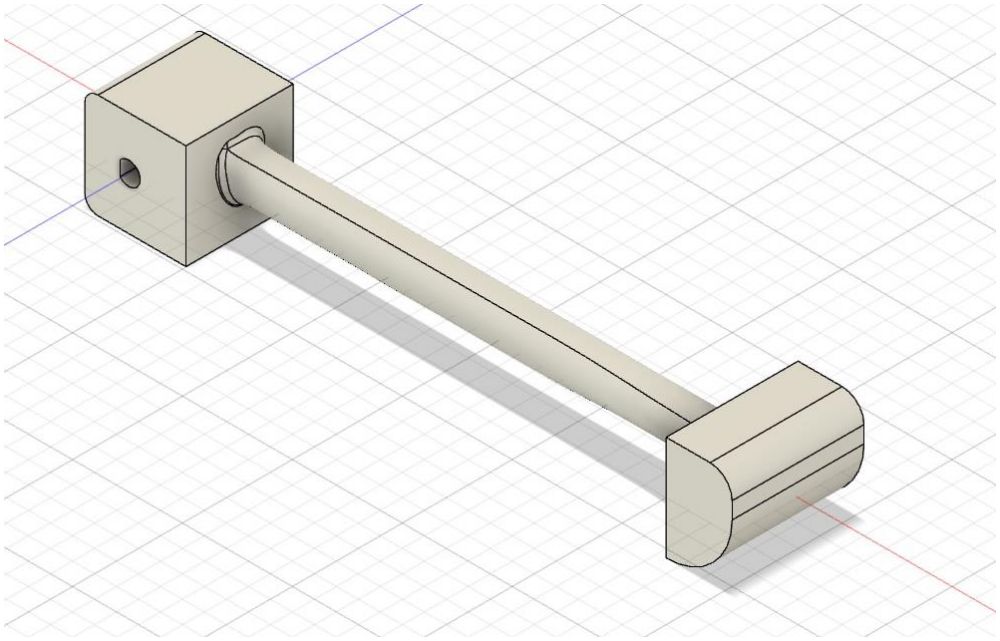
Na slici (4.11.) je prikazan XML kod zgloba koljena prednje desne noge.

```
<joint name ="desni_zglob_1" type ="revolute">
    <origin xyz ="0.114 0 -0.012" rpy ="0 -0.5 0"/>
    <parent link ="desna_noga_1"/>
    <child link ="desni_prst_1"/>
    <axis xyz ="0 1 0"/>
    <limit effort="10" velocity="10"/>
    <limit upper ="1.57" lower ="-1.57"/>
</joint>
```

Slika 4.11. XML kod zgloba koljena prednje desne noge

Zglobovi koljena su također rotacijski i samo je potrebno mijenjati podatke zgloba. Kao ishodište gibanja, karike koje se gibaju, os gibanja i granice pokreta.

Na slici (4.12.) je prikazan CAD model prsta.



Slika 4.12. CAD model prsta robota

Prst se sastoji od jednog komada, ali se prati isti postupak eksportiranja STL datoteke.

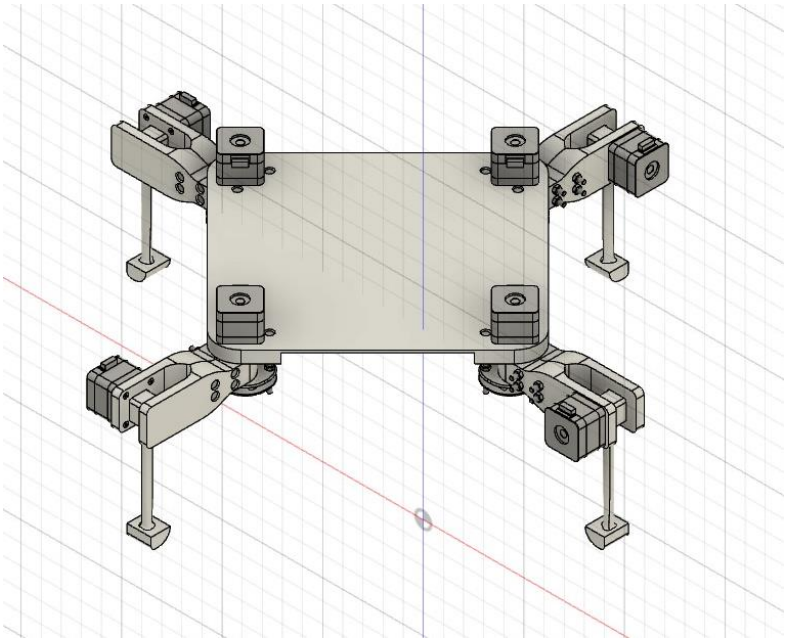
Na slici (4.13.) je XML kod karike prsta za prednju desno nogu.

```
175     <link name="desni_prst_1">
176         <visual>
177             <origin  xyz="0 0 0" rpy ="0 0 0"/>
178             <geometry>
179                 <mesh filename="package://STL_djelovi/Prst.stl" scale="0.001 0.001 0.001"/>
180             </geometry>
181             <material name="sivo">
182                 <color rgba="0.5 0.5 0.5 1"/>
183             </material>
184         </visual>
185 
186         <collision>
187             <origin  xyz="0 0 -0.125" rpy ="1.57 0 0"/>
188             <geometry>
189                 <cylinder radius="0.015" length="0.03"/>
190             </geometry>
191         </collision>
192 
193         <inertial>
194             <mass value="0.031"/>
195             <origin xyz="0 0 -0.059" rpy="0 0 0"/>
196             <inertia ixx="0" ixy="0" ixz="0" iyy="0" iyz="0" izz="0"/>
197         </inertial>
198     </link>
```

Slika 4.13. XML kod karike prsta robota

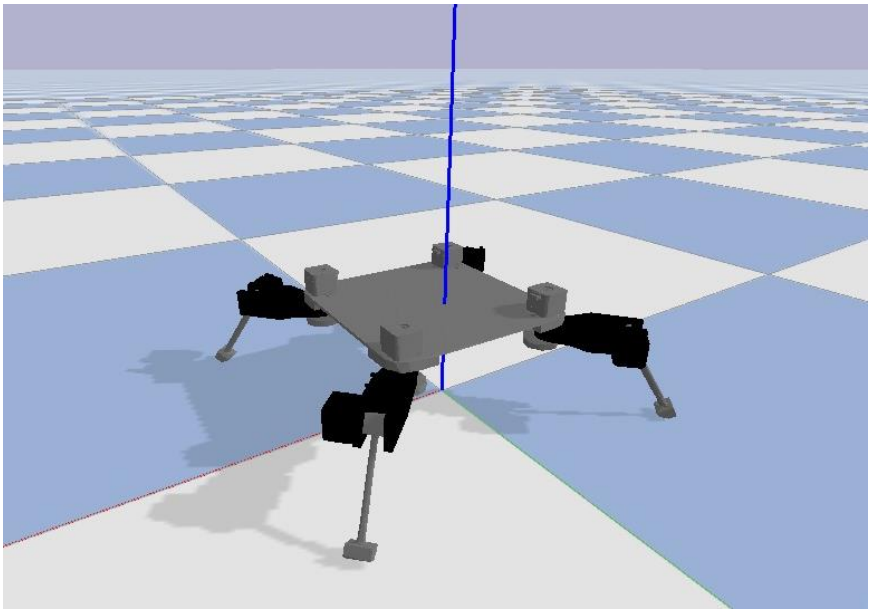
Isto kao i prethodne karike za prst učitavamo vizualni oblik iz STL datoteke, za kolizijski oblik uzimamo pojednostavljen oblik, a za inerciju i masu prenosimo podatke iz CAD softvera kojim je konstruiran oblik. No za kolizijski oblik radi jednostavnosti i oblika prsta postavljamo cilindar kao geometriju, koji će svojim plaštom biti u kontaktu s podlogom u svrhu hodanja.

Cjelokupna konstrukcija robota je vidljiva na slici (4.14.), gdje su spojeni svi dijelovi robota potrebni za simulaciju.



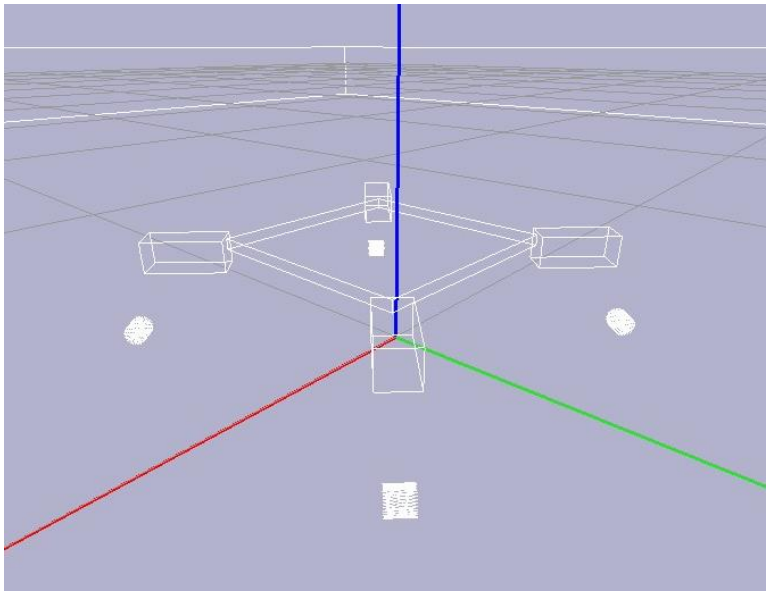
Slika 4.14. CAD model robota

Na slici (4.15.) je prikaz robota u Pybullet simulacijskom okruženju, prikazani su svi vidljivi dijelovi robota koji su specificirani pod vidljivom geometrijom u URDF datoteci.



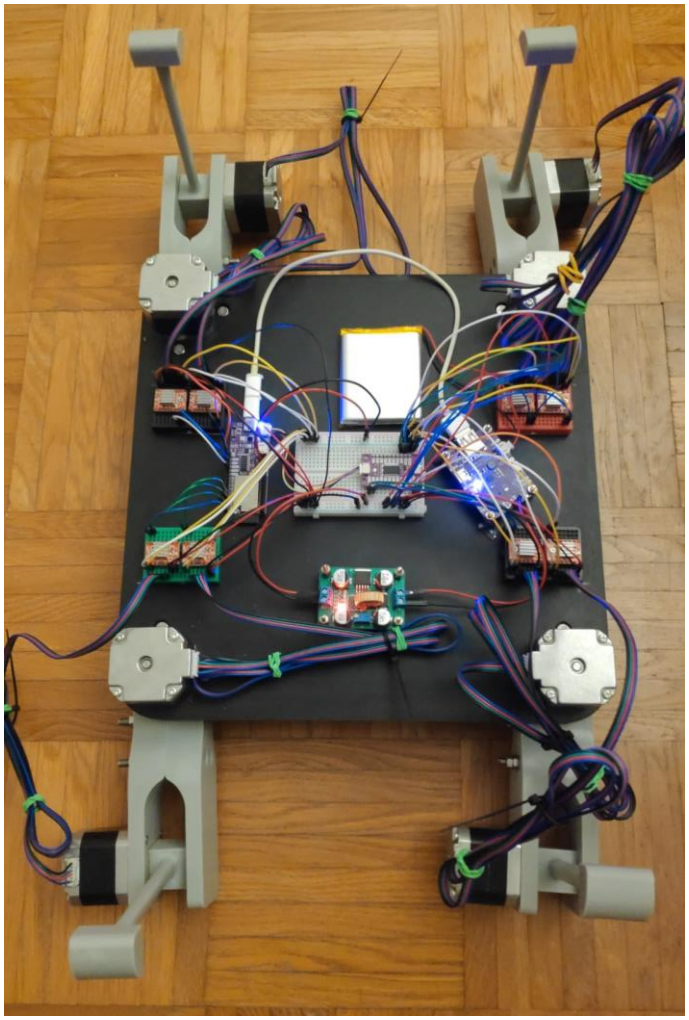
Slika 4.15. Vizualni prikaz robota u simulaciji

Na slici (4.16.) je prikaz geometrije kolizije robota. Iako se koriste primitivni oblici, pet kvadra i četiri cilindra, postižu se približno isti rezultati kao i sa originalnom geometrijom robota. Na ovaj način je ušteđeno vrijeme procesiranja kolizija oblika i omogućeno generiranje više robota u simulaciji bez da smo dosegli maksimalnu granicu broja faceta robota.



Slika 4.16. Kolizijska geometrija robota

Na slici (4.17.) je realni model robota čiji su se konstrukcijski dijelovi 3D printali.



Slika 4.17. Realni model robota

Pod prilogom [1] nalazi cjelokupni XML kod URDF modela robota.

5. SIMULACIJA I EVOLUCIJSKI ALGORITAM HODA ROBOTA

Za objašnjenje simulacije i primijenjenog evolucijskog algoritma biti će potrebno praćenje koda i znanje nekoliko naredbi vezanih uz Pybullet. Te će zbog toga kroz dalje gradivo uz slike koda biti objašnjene i glavne naredbe koje osiguravaju funkcioniranje simulacije.

5.1. Ideja evolucijskog algoritma u simulaciji hoda

Glavno pitanja vezano za učenje hoda robota može biti kako ocijeniti dobar hod? Kako prikazati hod robota s pojedincem u populaciji? Ili kako hod robota uopće izgleda?

Glavnu ideju hoda pronalazimo u prirodi, gdje se ljudi, životinje, insekti ili čak i bakterije kreću ponavljajući set pomaka zglobova. Npr. ako imamo čovjeka koji je dešnjak, za početak hoda (prema naprijed) će nagnuti tijelo van balansa i ispružiti desnu nogu naprijed, a lijevu natrag. Odgurujući se lijevom i dočekujući se na desnu nogu. Nakon toga će prebaciti težište tijela na desnu nogu i ljevom kročiti naprijed, a desnom odgurujući se natrag. Ponavljanjem ovih seta pomaka zglobova i prebacivanja težišta tijela sa noge na nogu, pojedinac će hodati. Princip hoda robota i sa više nogu je isti, robot će napraviti set pomaka rezultirajući korakom te će ponavljati te pokrete da hoda.

Iako robot ima osam SSG (stupnjeva slobode gibanja) ideja je ista, pojedinac u evolucijskom algoritmu će izgledati kao matrica s osam redaka i odabrani broj stupaca. Odabrani broj stupaca kako bi smo omogućili veću fleksibilnost robota za podešavanje zglobova. Broj redaka pojedinca predstavlja SSG robota, tj. zglobove robota. Broj stupaca predstavlja potreban broj pomaka zglobova kako bi robot ostvario korak. Stoga je pojedinac u populaciji prikazan kao 2D matrica pomaka zglobova robota. Na slici (5.1.) je prikazan primjer programiranog koraka pojedinca.

	0	1	2
0	-30	30	30
1	-30	30	30
2	30	-30	-30
3	30	-30	-30
4	-30	-30	0
5	0	0	0
6	0	0	0
7	-30	-30	0

Slika 5.1. Matrica hoda – programiran hod

Za vrijeme hoda izvršava se istovremeno redom cijeli vektorski stupac u matrici, tako da pokrećemo sve zglobova odjednom u označene pozicije. Program tad pričeka određeno vrijeme za pomak zglobova i nastavlja izvršavanje sljedećeg stupca naredbi za pomak zglobova. Kada program dođe do kraja matrice i izvrši zadnji stupčasti vektor pomaka, mjerimo poziciju i orijentaciju robota s referencom od početne pozicije. Ovisno o prijeđenom putu i devijaciji s puta i smjera ocjenjujemo pojedinca pomoću fitnes funkcije. Što je pojedinac (robot) prešao veći put prema naprijed i manje odstupio sa linije ishodišta, to mu je fitnes (ocjena) veća. S navedenim principom ideje ulazimo u konstruiranje simulacije i evolucijskog algoritma.

5.2. Simulacija

Za početak simulacije potrebno je postaviti glavne postavke simulatora. Potrebno je prvo uvesti sve potrebne module za funkcioniranje algoritma. Odabrati vrstu konekcije s Pybullet simulatorom i odabrati tok vremena izvođenja simulacije.

Na slici (5.2.) je prikazano koje sve Python module je potrebno koristiti kako bi radio algoritam. Osim modula za rad simulacije kao pybullet i pybullet_data potrebni su time, numpy i matplotlib. Time modul nam omogućava da se koristimo funkcijom vremena, zaustavimo vrijeme izvršenja nekih linija koda ili mjerimo prijeđeno vrijeme aktiviranog algoritma. Numpy je modul za matematičke i numeričke funkcije i osnova je pri stohastičnosti evolucijskog

algoritma pri generiranju slučajnih brojeva. Dok je matplotlib modul nam omogućava vizualni prikaz podataka i rezultata.

```
#   Library import-----------------------------------------------------------
import pybullet as p
import time
import pybullet_data
import numpy as np
import matplotlib.pyplot as plt
#-------------------------------------------------------------------------

#   Postavke simulacije------------------------------------------------------
if p.isConnected()==1:
    p.disconnect()
physicsClient = p.connect(p.GUI)
#physicsClient = p.connect(p.DIRECT)
Povezanost = p.getConnectionInfo(physicsClient)
p.setAdditionalSearchPath(pybullet_data.getDataPath())
p.setGravity(0,0,-9.81)

p.setRealTimeSimulation(1)

p.configureDebugVisualizer(p.COV_ENABLE_MOUSE_PICKING,0)
p.configureDebugVisualizer(p.COV_ENABLE_RGB_BUFFER_PREVIEW,0)
p.configureDebugVisualizer(p.COV_ENABLE_DEPTH_BUFFER_PREVIEW,0)
p.configureDebugVisualizer(p.COV_ENABLE_SEGMENTATION_MARK_PREVIEW,0)
#-------------------------------------------------------------------------
```

Slika 5.2. Moduli i postavke algoritma

Za početak rada koda potrebno je resetirati vezu s Pybullet simulatorom. Ako je već prije pokrenut kod i veza nije ispravno prekinuta naredbom *pybullet.disconnect()*, pri ponovnom pokretanju će biti greška prilikom pokretanja. Te se na početku postavlja uvjet i provjerava veza s funkcijom *pybullet.isConnected()* koja za vraća informaciju povezanosti sa simulatorom. Ako je povezan – 1, ako nije – 0. Ukoliko je još povezan, veza se prekida i izvršavanje koda se nastavlja. S naredbom *pybulet.connect()* ostvarujemo vezu s Pybullet simulatorom, a ovisno o načinu rada kojeg odaberemo ćemo imati i različite rezultate. Ako u funkciji odaberemo naredbu *p.GUI*, simulacija će se odvijati i prikazivati u novome grafičkome sučelju kako bi se vidjeli rezultati simulacije i učitanih objekata. Naredba *p.DIRECT* također pokreće simulaciju, no ne daje vizualni prikaz već se simulacija odvija u pozadini računala. Postoji još izbora pri pokretanju simulacije, no u svrhe ovog rada će se koristiti opcija s vizualnim prikazom. Naredba *p.setAdditionalSearchPath(pybullet_data.getDataPath())* je neobavezna naredba no ipak je vrijedno ju postaviti jer omogućuje uvoz primjernih objekata dobivenih instalacijom Pybullet modula. Naredba *pybullet.setGravity(x,y,z)* postavlja silu gravitacije u prostor simulacije. Mjerna jedinica je m/s^2 i za simulaciju rada iznosi 9,81 po suprotnom smjeru osi *z*. Vrijeme simulacije je moguće manipulirati no u kodu je s naredbom *pybullet.setRealTimeSimulation(1)* postavljeno na izvođenje u realnom vremenu. Jedinica u zagradama se odnosi na potvrdu

naredbe, nula bi označavala odabir drugog prolaska virtualnog vremena s drugom naredbom. Naredbe *pybullet.configureDebugVisualizer()* se odnose na grafičko sučelje simulacije. S njima je moguće isključiti mogućnost utjecanja dodira mišom na objekte u simulaciji i sakriti nepotrebne prozore kako bi se više ekrana posvetilo simulaciji. Nula na kraju predstavlja isključenje odabranih mogućnosti.

Na slici (5.3.) je kod parametara evolucijskog algoritma. Varijable *A, B,* P_k, P_m i *Iteracija* predstavljaju glavne varijable EA. *A* je duljina genoma, duljina matrice ili broj stupaca u matrici. *A* je broj pokreta motora u jednom koraku i izabran je iskustveno prema više pokretanja simulacije. Utvrđeno je da su najbolji rezultati dobiveni ako duljina broja pokreta koraka je izabrana između 4 i 12 koraka. Što je *A* manji, rezultati hoda su trzaviji i sporijeg povećanja fitnesa. Što je *A* veći, vrijeme izvođenje simulacije se eksponencijalno povećava, no rezultirani hodovi su finiji i prirodnijeg izgleda. Varijabla *B* predstavlja broj genoma u populaciji, treću dimenziju 3D matrice (dubinu ili broj matrica populacije). *B* mora biti paran broj (zbog križanja) i teži se da je veći jer predstavlja više različitih pojedinaca u jednoj populaciji. No opet i tu je tehnološko ograničenje računala, više pojedinaca u populaciji predstavlja više robota istovremeno u jednoj simulaciji. Što je više robota simulacija se sporije odvija i ima utjecaj na fitnes nekih pojedinaca. Dok se naredbe šalju istovremeno na svakog robota za pomicanje motora, ako je broj robota prevelik, prvi roboti čekaju dok se naredbe u zadnjim redovima ne izvrše. Razlog tome je što računala rade zadatak jedan po jedan i nije moguća savršena paralelizacija zadataka. Zbog tih razloga i razloga tvrdog broja mogućih faceta tijela u simulaciji, potrebno je broj B ograničiti do maksimalno 40. P_k je vjerojatnost križanja populacije, raspon varijable je od 0 do 1, što predstavlja 0% ili 100%. Pm je vjerojatnost mutacije populacije s istim rasponom kao i križanje, predstavlja stohastičku promjenu vrijednosti u matrici. Iskustveno i repetitivnim iskušavanjima algoritma je potrebno držati vrijednosti križanja u rasponu od 0.4 do 0.8, a vjerojatnost mutacije u rasponu od 0.5 do 0.9. Varijabla *Iteracija* predstavlja broj iteracija evolucijskog algoritma u petlji. Broj iteracija predstavlja broj generacija evolucijskog algoritma i vidljivi su rezultati općenito za 500 ili više generacija. *Pozicija_max* je varijabla mogućeg dosega električnih motora. Predstavlja pola maksimalnog kutnog pomaka za koji se motor može pomaknuti. Raspon gibanja motora je 80°, no kako je početna pozicija motora na polovici prostora gibanja, potrebno je prepoloviti raspon i postaviti da je pola vrijednosti mogućeg gibanja u pozitivnom smjeru i pola u negativnom.

```
#   Parametri evolucijskog algoritma
A = 8
B = 20
Pk = 0.3
Pm = 0.8
Iteracija = 200
Pozicija_max = 40

Nastavak = True

#Podaci iz prošlog pokretanja EA ako želimo nastaviti s prošlom populacijom
if Nastavak == True:
    Data = np.load('Gen_Fit_Pop.npz')
    Data_gen = Data['Generacija']
    Data_fit = Data['Fitnes']
    Data_sum = Data['Suma_fitnesa_plot']
    Data_elite = Data['Elitni_plot']
    Data_pop = Data['Populacija']
#-------------------------------------------------------------------------
```

Slika 5.3. Parametri evolucijskog algoritma i programa

Nastavak predstavlja opciju nastavka evolucijskog algoritma s generacijom, fitnesom i populacijom od zadnjeg pokretanja programa. Ukoliko je varijabla označena kao *True*, program nastavlja s zadnjim podacima. Ukoliko je False program počinje od nulte generacije s nasumično generiranom populacijom. Datoteka u koju program pohrani rezultate je .npz formata iz numpy modula dobivena naredbom *numpy.savez()*. Datoteka omogućuje spremanje više matrica u istu datoteku bez obzira na njihovu dimenziju.

Za simuliranje hoda robota je potrebno imati samo dvije stvari. Podloga ili prostor na kojem će se robot učiti hodati i samog robota. Na slici (5.4.) je prikazan dio koda zaslužan za generiranje podloge i robota u simulaciju. Generiranje objekata se vrši naredbom *pybullet.loadURDF()*, te se u zagradama specificira koji objekt želimo generirati. Prvo je potrebno generirati plohu iz primjera datoteka dobivenih instalacijom Pybullet modula pod nazivom plane.urdf. Nakon toga potrebno je generirati više inačica istog robota. Razlog je paralelizacija zadatka, umjesto da se na jednom robotu pokreću naredbe genoma jedan po jedan. Generira se onoliko robota koliko je velika populacija i svakom robotu se šalju naredbe pokreta iz vlastitog genoma. Stoga ako je veličina populacije (*B*) 20, generirati će se 20 robota, svaki s razmakom od 1m po y-osi. Učitava se jedna URDF datoteka (Crocko.urdf) i nadodaje na listu Crocko naredbom *append()*.

```
60  #   Učitavanje robota i objekata-----------------------------------------------
61  planeId = p.loadURDF('plane.urdf')
62  CrockoStartPos = [0,0,0.2]
63  CrockoStartOrientation = p.getQuaternionFromEuler([0,0,0])
64
65  Crocko = []
66  for i in range(B):
67      Crocko.append(p.loadURDF('Spidy.urdf',
68                               CrockoStartPos,
69                               CrockoStartOrientation))
70      CrockoStartPos += np.array([0,1,0])
71
72      p.addUserDebugText(text="Robot {}".format(i+1),
73                         textPosition=[0,i,0.2],
74                         textColorRGB=[0,0,0],
75                         textSize=1,
76                         lifeTime=0)
77      #pokazuje smjer kretanja
78      p.addUserDebugLine([0,i,0],[10,i,0],[1,0,0],lineWidth=3,lifeTime=0)
79
80  #--------------------------------------------------------------------------------
```

Slika 5.4. Generacija objekata i robota

Također u simulaciju se nadodaju vizualni pokazatelji kao redni broj robota i linija smjera njihovih gibanja.

Prije davanja naredbi za pokretanje nogu potrebno je izvući informacije o robotu iz URDF datoteke. Na slici (5.5.) je dio koda zaslužan za očitavanje zglobova i zadavanje njihovih imena u Pybulletu za lakše slanje naredbi. Pošto se generira više istih robota, za pregled informacija o zglobovima potrebno je pregledati samo jednog robota. Pregled broja zglobova dobivamo naredbom pybullet.getNumJoints() koja vraća broj zglobova. U petlji za svaki zglob naredbom *pybullet.getJointInfo()* dobivamo informacije o zglobu kao: indeks, ime, vrsta, granice... Za pokretanje robota nam je samo potrebna poveznica između imena zgloba i indeksa zgloba. Pošto naredbe za pokrete moraju sadržavati indeks zgloba kojeg pokrećemo, njih mijenjamo imenima zglobova radi lakšeg raspoznavanja. Dalje raspodijelimo sva imena zglobova u svoje zasebne varijable i u vektor *Motori*. Vektor *Motori* sadrži sve zglobove na robotu koje je moguće pokretati.

```
#   Informacije o zglobovima i link-ovima-----------------------------------
nJoints = p.getNumJoints(Crocko[0])
jointCrockoToId = {}
for i in range(nJoints):
  jointInfo = p.getJointInfo(Crocko[0], i)
  jointCrockoToId[jointInfo[1].decode('UTF-8')] = jointInfo[0]
  p.setJointMotorControl2(Crocko[0],i, p.VELOCITY_CONTROL,force=0)

desno_rame_1 = jointCrockoToId['desno_rame_1']
lijevo_rame_1 = jointCrockoToId['lijevo_rame_1']
desno_rame_2 = jointCrockoToId['desno_rame_2']
lijevo_rame_2 = jointCrockoToId['lijevo_rame_2']

desni_zglob_1 = jointCrockoToId['desni_zglob_1']
lijevi_zglob_1 = jointCrockoToId['lijevi_zglob_1']
desni_zglob_2 = jointCrockoToId['desni_zglob_2']
lijevi_zglob_2 = jointCrockoToId['lijevi_zglob_2']

Motori = np.array([desno_rame_1,
                   lijevo_rame_1,
                   desno_rame_2,
                   lijevo_rame_2,
                   desni_zglob_1,
                   lijevi_zglob_1,
                   desni_zglob_2,
                   lijevi_zglob_2])

#-------------------------------------------------------------------------------
```

Slika 5.5. Prikupljanje informacija o karikama i zglobovima robota

Nakon informacija o zglobovima potrebno je definirati pogled kamere dokuda će biti vidljiva simulacija, tj. scena simulacije. Za parametre se preddefinira skretanje, nagib i udaljenost između kamere i ciljane točke (*Yaw* [°], *Pitch* [°] i *Zoom* [m]). Vidljivo na slici (5.6.) naredbom *pybullet.resetDebugVisualizerCamera()* nam daje mogućnost da usmjerimo pogled scene. Osim unesenih parametara potrebno je i navesti ciljanu točku pogleda (*cameraTargetPosition*). ona je postavljena da bude na sredini početne linije na kojoj stoje roboti.

```
104  #   Varijable pomaka----------------------------------------------------------
105  velocity = 0.2 #m/s
106  force = 5 #N
107
108  #-------------------------------------------------------------------------------
109
110  #   Polozaj i orijentacija glavne kamere--------------------------------------
111  Yaw = 270 #početne vrijednosti
112  Pitch = -60
113  Zoom = 5
114
115  p.resetDebugVisualizerCamera(cameraDistance=Zoom,
116                               cameraYaw=Yaw,
117                               cameraPitch=Pitch,
118                               cameraTargetPosition= [0,B/2,0])
119
120  #-------------------------------------------------------------------------------
```

Slika 5.6. Varijable pomaka i postavke kamere

Pokretanje zglobova robota u simulaciji se postiže naredbama *pybullet.setJoint-MotorControl2()* za pomicanje pojedinog zgloba ili *pybullet.setJointMotorControlArray()* za pomicanje više zglobova odjednom. U daljnjem radu potonja naredba će biti korištena za davanje naredbi pomaka. Da bi se pojednostavnilo pomicanje više zglobova odjednom za redom u petlji, potrebno je definirati funkcije pomaka. Na slici (5.7.) su definirane funkcije. Funkcija *povratak_pozicija()* naređuje svim zglobovima svih robota da se vrate u početnu poziciju, nultu poziciju motora. Koristi se kada je završeno izvođenje jedne populacije pomaka i potrebno je vratiti robote u početne pozicije kako bi se započela simulacija sljedeće populacije potomaka. Funkcija *vektor_pomak()* definira pomake svih zglobova jednog robota. Za korištenje ove funkcije potrebno je navesti ime robota, imena zglobova koje treba pomicati i vektor pozicija svakog zgloba. Funkcija *relax()*, isto kao i funkcija *povratak_pozicija()*, se odnosi na sve robote i sve njihove zglobove. Šalje naredbu zglobovima da njihova brzina gibanja bude nula, i koristi se između generiranja robota u simulaciji i pokretanja hoda. Razlog tome je što se zna dogoditi da prilikom generiranja robota neki zglobovi imaju početnu brzinu i to dovodi do neočekivanog pomaka robota. Ovim načinom se zglobovi "opuštaju" prije slanja naredbi za pomicanje.

```
#   Definiranje pomaka motora-----------------------------------------------
def povratak_pozicija():
    for r in range(B):
        p.setJointMotorControlArray(bodyIndex = Crocko[r],
                                    jointIndices = [desno_rame_1,
                                                    lijevo_rame_1,
                                                    desno_rame_2,
                                                    lijevo_rame_2,
                                                    desni_zglob_1,
                                                    lijevi_zglob_1,
                                                    desni_zglob_2,
                                                    lijevi_zglob_2],
                                    controlMode = p.POSITION_CONTROL,
                                    targetPositions = 0*np.ones(8),
                                    forces = force*np.ones(8))

def vektor_pomak(robot,vektor_motor,target,force):
    p.setJointMotorControlArray(bodyIndex = robot,
                                jointIndices = vektor_motor,
                                controlMode = p.POSITION_CONTROL,
                                targetPositions = np.radians(target),
                                targetVelocities=0*np.ones((8,1)),
                                forces = 10*np.ones((8,1)))

def relax():
    for r in range(B):
        p.setJointMotorControlArray(bodyIndex=Crocko[r],
                                    jointIndices=[desno_rame_1,
                                                    lijevo_rame_1,
                                                    desno_rame_2,
                                                    lijevo_rame_2,
                                                    desni_zglob_1,
                                                    lijevi_zglob_1,
                                                    desni_zglob_2,
                                                    lijevi_zglob_2],
                                    controlMode=p.VELOCITY_CONTROL,
                                    targetVelocities=np.zeros((8)))

#--------------------------------------------------------------------------------
```

Slika 5.7. Funkcije pomaka motora

5.3. Evolucijski algoritam hoda

Algoritam započinjemo po standardnoj shemi, stvaramo inicijalnu populaciju i računamo fitnes, relativni fitnes i pokrećemo petlju u kojoj se nalaze križanje, mutacija i evolucija genoma. Petlja se ponavlja dok god se ne postigne određen uvjet prekida.

5.3.1. Definiranje varijabli i inicijalizacija

Za početak algoritma potrebno je definirati varijable koje će se koristiti u petlji kao na slici (5.8.). Varijable u pitanju su vektori i matrice koje se koriste u petljama koje je potrebno prethodno definirati veličine. Također osim definiranja pokrećemo funkciju *relax()* da se resetiraju zglobovi robota i počinjemo mjeriti vrijeme evolucijskog algoritma naredbom *time.time()*. Naredba *time()* nam samo zabilježava trenutno vrijeme. No ako upotrijebimo

naredbu ponovo na kraju algoritma i samo oduzmemo vrijednosti, ostatak će biti vrijeme rada algoritma. Još vrijedno spomenuti je vektor *Razlika*. Pošto se koristi više robota poslagani na početne pozicije po pozitivnoj *y*-osi simulacije. Rezultati naredbe *pybullet.getBasePositionAndOrientation()* daje apsolutne vrijednosti pozicije i orijentacije robota naspram glavnog koordinatnog sustava. Zbog toga je potrebno uračunati početnu poziciju robota i oduzeti potrebnu vrijednost radi ispravnog računanja fitnes vrijednosti.

```
163  #   Evolucijski algoritam-----------------------------------------------------
164  relax()
165  Tic = time.time()
166  #Vektori i matrice za petlju
167  Roditelji = np.zeros((8,A,B))
168  Potomci = np.zeros((8,A,B))
169  Fitnes = np.zeros((B,1))
170  Fitnes_potomaka = np.zeros((B,1))
171
172  XYZ = np.zeros((B,3))
173  Orijentacija = np.zeros((B,4))
174  Psi = np.zeros((B,1))
175  Razlika = np.arange(0,B,1)
```

Slika 5.8. Varijable algoritma

Inicijalizacija programa je moguća na dva načina. Da se pokrene s populacijom nasumičnih vrijednosti ili da se populacija uvede iz vanjske datoteke. Drugi način se koristi za nastavak evolucijskog algoritma, tj. koristi zadnju populaciju koju je algoritam spremio u vanjskoj datoteci. Prethodne podatke koji su spremljeni u .npz datoteku izvlačimo i s njima krećemo u petlju kao da nastavljamo evolucijski algoritam. Podaci kao generacija, fitnes pojedinaca u populaciji, ukupna suma fitnesa populacija, fitnes elitnog genoma i sama populacija. Ako se algoritam inicijalizira s nasumičnom populacijom, tada se za vrijednosti populacije generiraju nasumične vrijednosti od -40° do 40°. Populacija je 3D matrica dimenzije 8x*A*x*B*, za broj početne generacije se označava 0 i naredbama *pybullet.addDebugUserParameter()* i *pybullet.addUserDebugText()* nadodaje oznaka generacije u simulacijsko sučelje kako bi se lakše pratilo stanje simulacije. Inicijalizacija je prikazana na slici (5.9.).

```
176
177 if Nastavak == True:
178     Generacija = Data_gen
179     Fitnes = Data_fit
180     Suma_fitnesa_plot = Data_sum
181     Populacija = Data_pop
182
183     Suma_fitnesa = np.sum(Fitnes)
184     Elitni_plot = Data_elite
185
186     #Oznaka generacije
187     Slider = p.addUserDebugParameter("Generacija",0,Iteracija+Data_gen,Generacija)
188     Oznaka_gen = p.addUserDebugText(text="Generacija {}".format(Generacija),
189                                     textPosition=[0,B/2,0.7],
190                                     textColorRGB=[0,0,0],
191                                     textSize=2,
192                                     lifeTime=0)
193
194 else:
195     #Inicijalna populacija - random
196     target = 88/2
197     Populacija = np.random.randint(-target, target,size=(8, A, B))
198
199     #Oznaka generacije
200     Generacija = 0 #Početna generacija - nulta
201     Slider = p.addUserDebugParameter("Generacija",0,Iteracija,Generacija)
202     Oznaka_gen = p.addUserDebugText(text="Generacija {}".format(Generacija),
203                                     textPosition=[0,B/2,0.7],
204                                     textColorRGB=[0,0,0],
205                                     textSize=2,
206                                     lifeTime=0)
207
```

Slika 5.9. Inicijalizacija

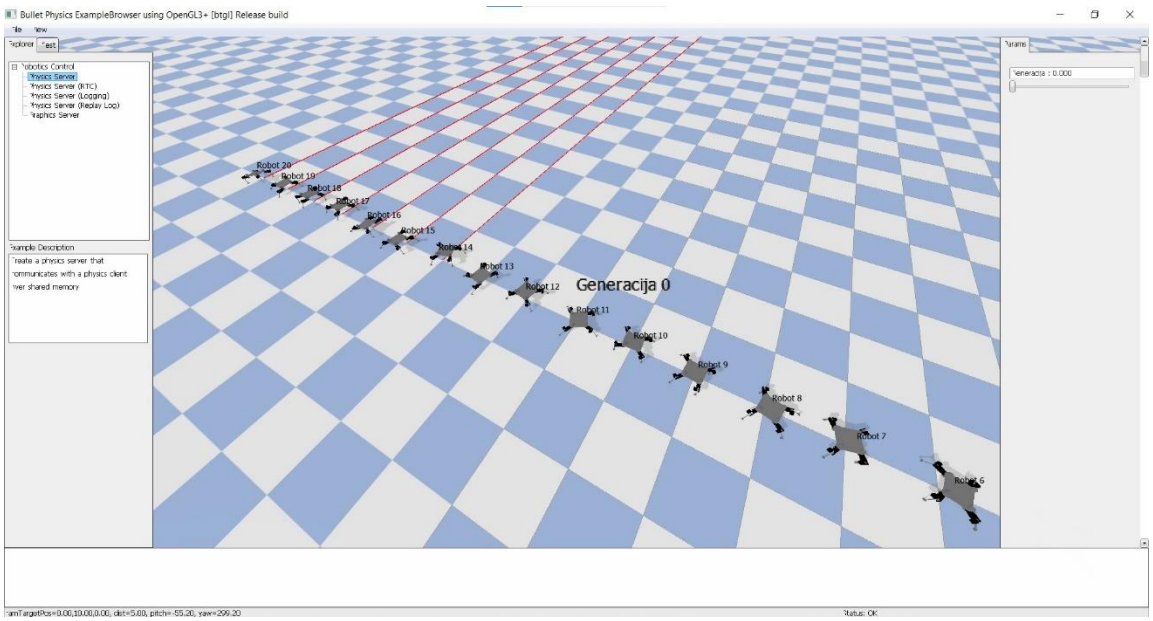
5.3.2. Fitnes funkcija

Fitnes funkcija se sastoji od ocjenjivanja puta koju je prevalio pojedinac. Prvo je potrebno robotima koji su na svojim početnim pozicijama dati naredbu pokretanja zglobova. Vidljivo iz slike (5.10.) pomaci se šalju iz populacije pomoću funkcije *vektor_pomak()*. Funkcija je postavljena u dvije petlje tako da prolazi kroz drugu i treću dimenziju matrice populacije. U njoj je definiran robot, zglobovi koji se pokreću, vektor iz populacije i maksimalni moment koji je potreban za pomak. Prva (unutarnja) petlja prolazi kroz treću dimenziju matrice (*B*) populacije i svakom pojedinom robotu zadaje pomake iz zadanih genoma. Druga (vanjska) petlja prolazi kroz stupce matrice populacije (*A*) s pauzom od 0,4 s mijenjajući vektor potrebnih pomaka.

```
207
208     for i in range(A):
209
210         time.sleep(0.3)
211         for j in range(B):
212             vektor_pomak(Crocko[j],Motori,Populacija[:,i,j],force)
213     time.sleep(0.1)
214     povratak_pozicija()
215
216     for i in range(B):
217         XYZ[i], Orijentacija[i] = p.getBasePositionAndOrientation(Crocko[i])
218         Psi[i] = p.getEulerFromQuaternion(Orijentacija[i,:])[0]
219         p.resetBasePositionAndOrientation(Crocko[i], [0,i,0.1], CrockoStartOrientation)
220
221
222     X = XYZ[:,0]
223     Y = XYZ[:,1]-Razlika
224
225     for i in range(B):
226         if X[i] <= 0: #ako se robot kreće prema nazad
227             X[i] = 0
228         Fitnes[i] = 500*X[i]**2/(0.001+25*abs(Y[i])+470*abs(Psi[i]))
229
230
231     Suma_fitnesa = np.sum(Fitnes)
232     Suma_fitnesa.astype(np.ndarray)
233     Suma_fitnesa_plot = Suma_fitnesa
234
235     Elitni_plot = np.max(Fitnes)
236     Elitni_plot.astype(np.ndarray)
237
```

Slika 5.10. Fitnes funkcija

Funkcije pomaka su vidljive u simulatoru, te ostali dio kod nema većeg utjecaja na simulaciju (ako ne ubrajamo početne postavke simulacije). Da bi smo dobili fitnes svakog pojedinca iz populacije moramo ih pokrenuti kako je vidljivo na slici (5.11.). Svi roboti djeluju istovremeno, populacija se testira odjednom ne jedan po jedan pojedinac.



Slika 5.11. Paralelna evolucija robotskog hoda

Nakon odrađenih seta pomaka na svim robotima potrebno je zapisati njihove položaje i orijentacije. Naredbom *pybullet.getBasePositionAndOrientation()* dobivamo dva tuple objekta. Jedan sadrži tri broja, predstavljaju poziciju koordinatnog sustava baze robota s obzirom na koordinatni sustav simulatora. Drugi tuple objekt sadrži četiri broja koji predstavljaju kvarterione za orijentaciju, no njih pretvaramo u lakše razumljive Eulerove kutove. Naredba *pybullet.getEulerFromQuarterion()* pretvara kvarterione u Eulerove kutove i važan nam je samo prvi kut. Tj. pomak *x*-osi baze robota naspram apsolutne *x*-osi simulatora. Nakon toga dovodimo svakog robota u početnu poziciju i orijentaciju, te računa se prijeđeni put po osima. Za svaki genom računamo njegov fitnes pomoću funkcije (5.1.)

$$Fitnes = \frac{500X^2}{0{,}001 + 25|Y| + 470|\Psi|} \tag{5.1.}$$

gdje je:

X - prijeđeni put koordinatnog sustava baze robota po *x*-osi koordinatnog sustava simulatora

Y - prijeđeni put koordinatnog sustava baze robota po *y*-osi koordinatnog sustava simulatora

Ψ - kut *x*-osi baze robota naspram *x*-osi apsolutnog koordinatnog sustava simulatora

Fitnes se pokušava maksimizirati te prema funkciji (5.1.) se vidi da je fitnes veći što je X veći, a Y i Ψ manji. Time pojedinac koji ima mala odstupanja po Y i Ψ, a veće vrijednosti po X ima i veći fitnes. Konstanta 0,001 predstavlja mjeru sigurnosti u slučaju da vrijednosti Y i Ψ budu nula.

5.3.3. Relativni fitnes

Relativni fitnes predstavlja ocjenu pojedinca relativnu na svoju populaciju. Da bi smo dobili relativni fitnes pojedinaca potrebno je prvo izračunati sumu fitnesa, što je zbroj svih fitnesa u populaciji. Računanje sume fitnesa je vidljivo u formuli (5.2.) ili u kodu na slici (5.10.).

$$Suma\ fitnes = \sum_{i}^{B} Fitnes_i \tag{5.2.}$$

gdje je:

$Fitnes_i$ – fitnes pojedinca

i – redni broj fitnesa

B – broj genoma u populaciji

Relativni fitnes je tada fitnes pojedinca podijeljen sa ukupnom sumom fitnesa, vidljivo u formuli (5.3.).

$$Relativni\ fitnes_i = \frac{Fitnes_i}{Suma\ fitnesa} \tag{5.3.}$$

Zbroj svih relativnih fitnesa mora uvijek rezultirati 1.

Također izračun relativnog fitnesa prikazano u kodu na slici (5.12.)

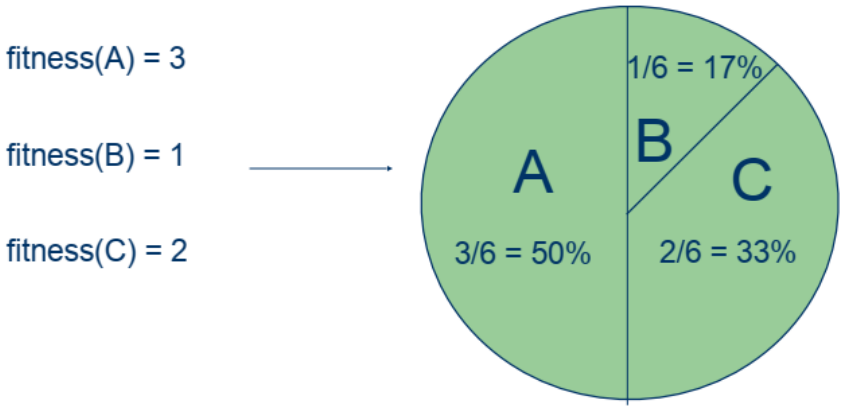
```
256     #Relativni fitnes
257     Relativni_fitnes = Fitnes/Suma_fitnesa
```

Slika 5.12. Relativni fitnes

Relativni fitnes je važan u procesu križanja za selekciju roditelja nove generacije opisano u sljedećem potpoglavlju .

5.3.4. Izbor roditelja

Roditelji koji stvaraju sljedeće generacije potomaka se odabiru pomoću ruletnog pravila. Ruletno pravilo je stohastička metoda selekcije pojedinaca i ovisi o fitnesu pojedinca. Što je veći fitnes pojedinca veća je i vjerojatnost da će taj pojedinac postati roditelj. Za odabir je važan relativni fitnes, po njemu se odabire koji pojedinac postaje roditelj. Te pojedinac može postati roditelj na dva načina: da ima veći relativni fitnes od nasumično odabrane vrijednosti od 0 do 1 ili kumulativno zaradi veći relativni fitnes. Na slici (5.13.) je primjer ruletnog odabira s 3 pojedinaca opisanih sa svojim fitnesima i relativnim fitnesima.



Slika 5.13. Ruletno pravilo [9]

Odabire se nasumična vrijednost od 0 do 1 koju nazivamo Random_roditelj.

$$Random\ roditelj\ \in \boldsymbol{R}, \qquad Random\ roditelj\ \in\ [0,1] \tag{5.4.}$$

Na primjer:

$$Random\ roditelj = 0{,}43 \tag{5.5.}$$

Pravilo za odabir roditelja, prikazano na formuli (5.6.), vrijedi da pojedinac postaje roditelj ako je njegov relativni fitnes veći ili jednak od nasumično odabrane vrijednosti.

$$Random\ roditelj \leq Relativni\ fitnes \tag{5.6.}$$

Što u ovom slučaju i vrijedi jer prvi pojedinac "A" ima veći relativni fitnes i postaje prvi roditelj. Dalje za drugog roditelja odabiremo ponovno nasumičnu vrijednost:

$$Random\ roditelj = 0{,}63 \tag{5.7.}$$

Sada opet se daje šansa prvom pojedincu no kako on ne zadovoljava uvjet (5.6.) njegov relativni fitnes se zbraja s pojedincem "B". Sada se vrednuje kumulativna vrijednost, tj. zbroj relativnih fitnesa "A" i "B". Pošto sada zbroj iznosi 0,67 uvjet u formuli (5.6.) je zadovoljen i drugi roditelj je drugi pojedinac u populaciji. Postupak se nastavlja dok god se ne popune sva mjesta roditelja.

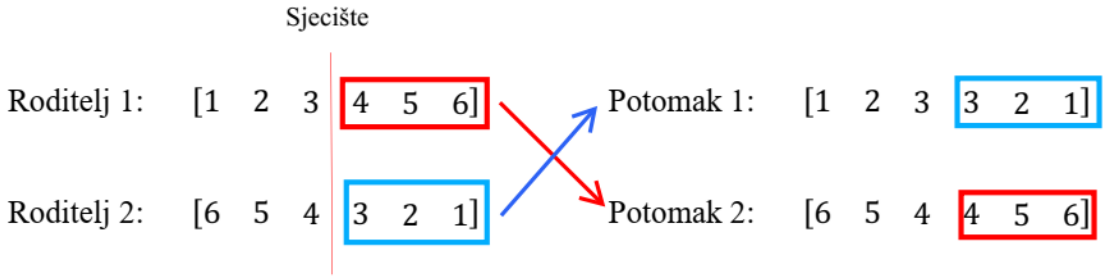
Isti postupak se primjenjuje i u kodu što je prikazano u programiranom obliku na slici (5.14.). U početku se odmah izvlače nasumične vrijednosti s naredbom numpy.random.rand(*B*,1) koja vraća vektor nasumičnih vrijednosti između 0 i 1 veličine [*B* x 1]. Vanjska petlja iterira za mjesto roditelja, a unutarnja za odabir roditelja iz populacije. Sumator ima početnu vrijednost prvog člana populacija i on se koristi za uvjet odabira. U unutarnjoj petlji se ispituj relativni fitnesi i ako je prvi relativni fitnes veći ili jednak nasumičnoj vrijednosti, prvi pojedinac postaje prvi roditelj i prekida se unutarnja petlja. Ako ne zadovoljava uvjet, ispituje se drugi uvjet, tj. dali je pojedinac zadnji u populaciji. Tada je njegova vrijednost Sumatora jednaka 1 i sigurno postaje roditelj. Ako nije zadnji, Sumatoru se zbraja sljedeći relativni fitnes od sljedećeg pojedinca i ispituje se sljedeći pojedinac za mjesto roditelja. Proces se ponavlja dok se ne odaberu svi roditelji, tj. dok se ne izvrti vanjska petlja.

```
258
259     #Izbor roditelja - ruletno pravilo
260     Random_roditelj = np.random.rand(B,1)
261
262     for i in range(B):
263         Sumator = Relativni_fitnes[0]
264         for j in range(B):
265             if (Sumator >= Random_roditelj[i]):
266                 Roditelji[:,:,i] = Populacija[:,:,j]
267                 break
268             elif (j == B-1):
269                 Roditelji[:,:,i] = Populacija[:,:,B-1]
270             else:
271                 Sumator = Sumator + Relativni_fitnes[j+1]
272
```

Slika 5.14. Izbor roditelja

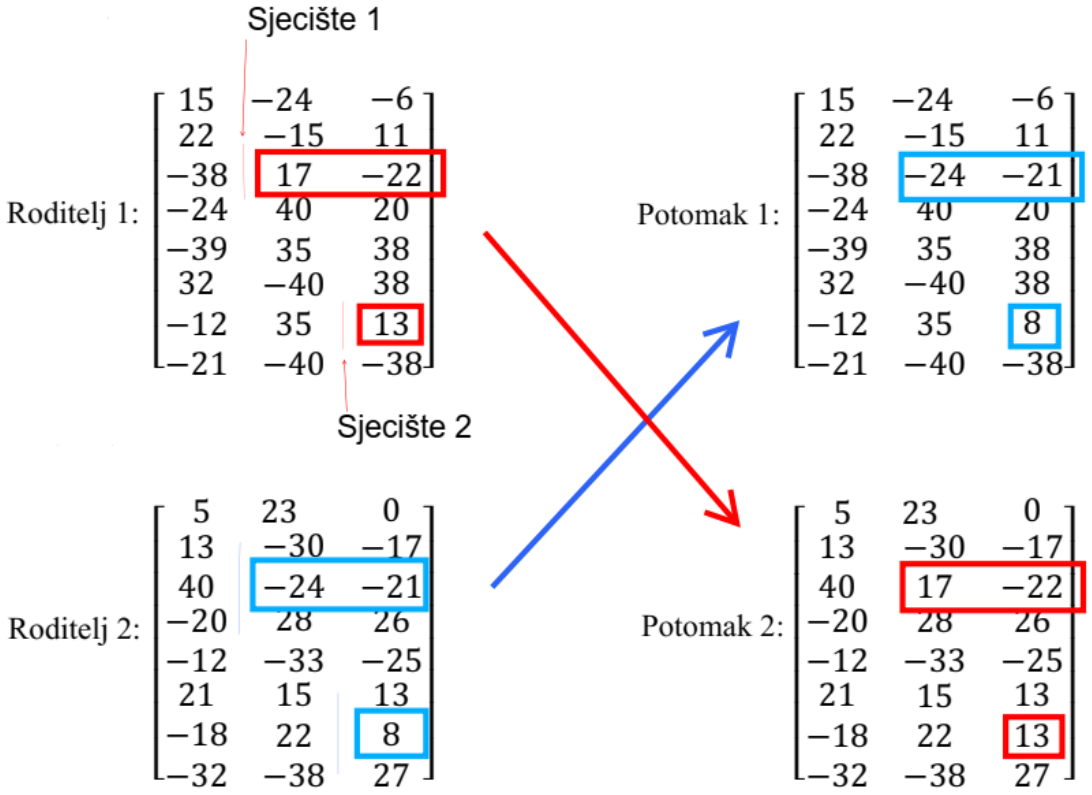
5.3.5. Križanje

Križanje ili rekombinacija je proces miješanja dvaju genoma (roditelja) u svrhu dobivanja dva nova genoma (potomka). Križanjem ne dobivamo nove vrijednosti u genomima, oni se samo miješaju na nasumično određenoj granici. Tako da vrijednosti u roditeljima i potomcima su ista, samo se nalaze u drugom genomu. Na slici (5.15.) je primjer jednog križanja vektorskih genoma. Obično se križanje vektorskih genoma odvija da se nasumično odabere sjecište genoma i vrijednosti na jednoj strani tih sjecišta zamjene mjesta u vektorima. Time se dobivaju novi genomi s istim vrijednostima.



Slika 5.15. Križanje vektorskih genoma

Ako su genomi matričnog oblika kao što je u ovom slučaju i na slici (5.16.), tada se izvodi slični proces. Ako bi smo koristili jedno sjecište, tada bi križanje bilo preveliko i previše podataka bi se prebacivalo iz genoma u susjedni genom. Također ako je genom samo jedan vektor dalje od boljeg fitnesa, nikako neće moći zamijeniti taj jedan vektor osim ako je zadnji u matrici. Zato se koristi više sjecišta i vektori se mijenjaju kao na slici (5.16.). Za križanje je važan parametar Pk jer on predstavlja vjerojatnost križanja ili udio genoma koji će se križati. Za svaka dva genoma u populaciji se odvija križanje i za svaki redak se izdvaja nasumična vrijednost između 0 i 1. Ako je nasumična vrijednost na određenome retku unutar vrijednosti Pk, tada se taj vektor križa i u prvom i drugom roditelju. Odabire se nasumično sjecište i jedan dio vektora prelazi u susjedni genom, a susjedni genom isti redak samo svoje vrijednosti predaju u prvi genom. Tako je u primjeru na slici (5.16.) prikazano da se križaju treći i sedmi stupac, dok je prvo sjecište na prvom mjestu retka, a drugo sjecište je na drugom mjestu retka.



Slika 5.16. Križanje matričnih genoma

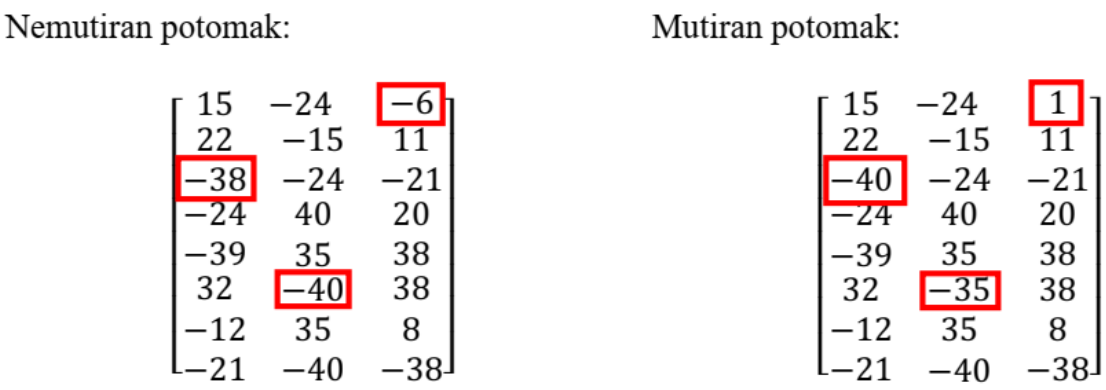
Isti proces se koristi i u kodu, te je na slici (5.17.) prikazan programski dio koda križanja matričnih genoma. Prvo se definiraju potomci da budu isti kao i roditelji, pri čemu je lakše samo promijeniti vektore koji se križaju. Dvije su petlje, prva vanjska prelazi po svakome drugome pojedincu u populaciji. Druga unutarnja petlja prelazi po recima i za svaki redak se zadaje nasumična vrijednost *K* koja se tad ispituje uvjetom dali je manja ili jednaka od P_k. U slučaju da je manja ili jednaka, odvija se križanje između dva pojedinca.

```
272
273     #   Križanje - križaju se dvije susjedne 2D matrice u 3D matrici
274     Potomci = Roditelji
275     for i in range(0,B,2):#po matrici
276         for j in range(8):#po retku (motoru)
277             K = np.random.rand()
278             if (K <= Pk): # ako je unutar Pk
279 
280                 Sjecište = np.random.randint(low=1,high=A)
281                 Potomci[j,Sjecište:,i] = Roditelji[j,Sjecište:,i+1]
282                 Potomci[j,Sjecište:,i+1] = Roditelji[j,Sjecište:,i]
283 
```

Slika 5.17. Križanje

5.3.6. Mutacija

Mutacija je mala slučajna promjena genoma i povezuje prostor pretrage mogućih rješenja. Iako se križanjem dobivaju potomci oni nisu potpuni ako ne mutiraju, tj. ako nemaju malu slučajnu promjenu u svojim vrijednostima. Proces je prilično jednostavan, potrebno je najprije definirati vjerojatnost mutacije P_m. Tada se za svaku vrijednost u populaciji izvlači jedna nasumična brojka između 0 i 1. Ako ta brojka je manja ili jednaka od vjerojatnosti mutiranja P_m, tada vrijednosti se mijenja. Mijenjati se može na mnoge načine, no najjednostavniji način je da se već postojećoj vrijednosti pridruži ili oduzme neka mala vrijednost. Na slici (5.18.) prikazan je primjer gdje su nakon križanja vrijednosti, u crvenom označene, mutirale.



Slika 5.18. Primjer mutacije genoma

Isti proces se izvodi i u programskom dijelu, na slici (5.19.), gdje se koriste tri petlje koje prelaze preko svake dimenzije 3D matrice populacije. Čime se za svaku vrijednost u matrici izdvaja nasumična vrijednost *M* koja varira između 0 i 1. Ako vrijednost *M* zadovoljava uvjet mutacije, tj. da je manja ili jednaka vjerojatnosti mutacije P_m, tada se izvodi mutacija na toj vrijednosti. Mutira se tako da se već postojećoj vrijednosti nadodaje broj nasumično izabran između -10 i 10. Nakon svega je još potrebno ograničiti vrijednosti u populaciji kako ne bi prelazili moguće vrijednosti pokreta motora robota. Naredbom *numpy.clip()* se sve vrijednosti veće od navedenih smanji na naveden maksimum, dok se sve manje povećaju na naveden minimum mogućeg pokreta.

```
284
285     #   Mutacija - svaki broj u 3D matrici ima 'Pm' vjerojatnost da mutira
286     for i in range(8):
287         for j in range(A):
288             for k in range(B):
289                 M = np.random.rand()
290                 if (M <= Pm):
291                     Potomci[i,j,k] = Potomci[i,j,k]+np.random.randint(low=-10,high=10)
292     Potomci = np.clip(Potomci,-Pozicija_max,Pozicija_max)
293     #ograniči se mutacija da ne varira van mogućih pomaka motora robota (+-44°)
294
```

Slika 5.19. Mutacija

5.3.7. Evaluacija

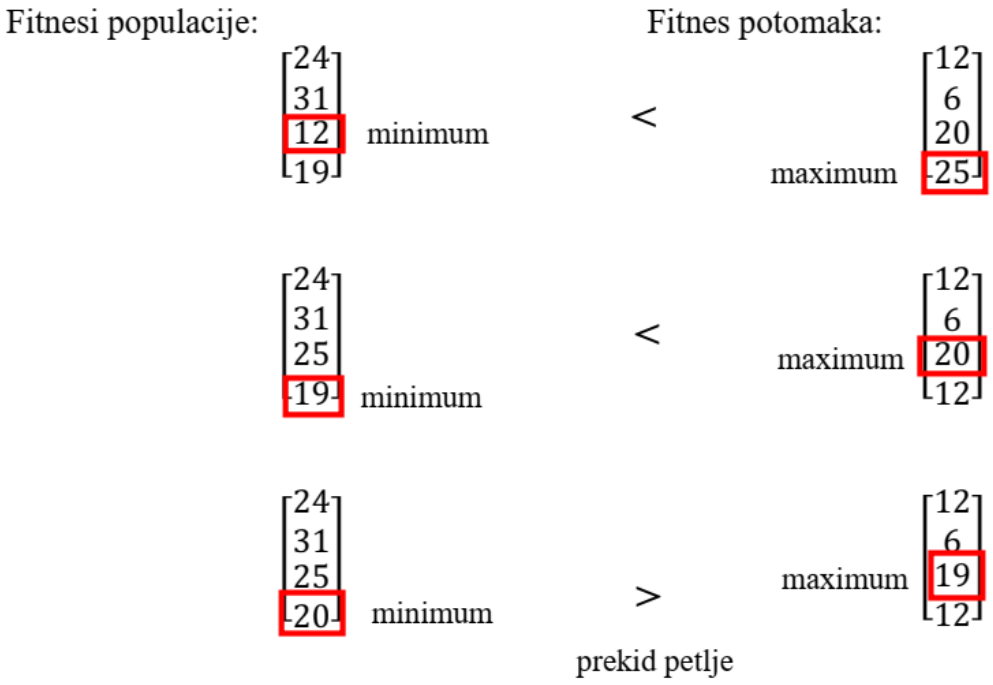
Potomci nakon provedenog križanja i mutiranja ne postaju odmah nova generacija populacije. Prvo je potrebno izračunati vrijednosti njihovih fitnesa i usporediti ih s prošlom generacijom. U slučaju da su potomci gori od starije generacije, moraju se odbaciti i zadržati bolji pojedinci. Evaluacija ili preživljavanje služi ako je neki dio potomaka bolji od starije generacije, da se pojedinci s većim fitnesima kombiniraju u staru populaciju za novu generaciju, a gori pojedinci odbacuju. Na slici (5.20.) je prikazan programski kod takve evaluacije. koristi se jedna petlja i provjerava se minimum u fitnesu populacije, a maksimum u fitnesu potomaka. Ako je minimum populacije manji od maksimuma potomaka, mijenjaju mjesta ta dva pojedinca. Proces se ponavlja dok god je neki pojedinac u potomcima bolji od pojedinca u populaciji. Ako više nema boljih pojedinaca u potomcima petlja se prekida.

```
321
322     for i in range(B): #evaluacija
323
324         Fitnes_min = np.min(Fitnes) #najgori član roditelja
325         Argmin = np.argmin(Fitnes)
326
327         Fitnes_potomaka_max = np.max(Fitnes_potomaka) #najbolji član potomaka
328         Argmax = np.argmax(Fitnes_potomaka)
329
330         if (Fitnes_min < Fitnes_potomaka_max):
331
332             Fitnes_potomaka[Argmax] = Fitnes_min #izmjena fitnesa
333             Fitnes[Argmin] = Fitnes_potomaka_max
334
335             Genom = Populacija[:,:,Argmin] #izmjena genoma
336             Populacija[:,:,Argmin] = Potomci[:,:,Argmax]
337             Potomci[:,:,Argmax] = Genom
338
339         else: #ako nema boljeg člana u potomcima prekini evaluaciju
340             break
341
```

Slika 5.20. Evaluacija

Proces je lakše vizualizirati na navedenom primjeru s četiri pojedinaca u populaciji na slici (5.21.). U populaciji se traži pojedinac s minimalnim fitnesom, dok se među potomcima traži

pojedinac s maksimalnim fitnesom. Ako je najbolji pojedinac (najveći fitnes) potomaka bolji od najgoreg pojedinca (najmanji fitnes) populacije, tada oni mijenjaju mjesta. Proces se ponavlja dok god vrijedi uvjet da ima boljih potomaka, kada taj uvjet nije ispunjen petlja se prekida. Kao rezultat u populaciji ostaju pojedinci s najboljim fitnesima i prenose se u novu generaciju. Dok u potomcima ostaju najgori pojedinci i odbacuju se. Time se održava da fitnes kroz generacije na pada, već samo raste ili stagnira.



Slika 5.21. Primjer evaluacije

5.3.8. Zapis podataka

Nakon prekida glavne petlje algoritam se privodi kraju i potrebno je memorirati rezultate. Glavni podatak je elitni genom ili najbolji pojedinac populacije u zadnjoj generaciji. Na slici (5.22.) je dio programa koji sprema podatke u dvije vrste datoteke. Jedna je tekstualna, dok je druga NPZ vrsta. Tekstualna datoteka je format .txt i moguće ju je otvoriti i pregledati sadržaj ili sačuvati podatke za učitavanjem u drugoj skripti. Druga datoteka je formata .npz od numpy modula i koristi GZIP kompresiju podataka za njihovo spremanje. NPZ datoteka omogućava da se u nju memoriziraju više matica različitih dimenzija. Što uglavnom nije izvedivo sa TXT formatom, jer bi morali nadoknađivati dimenzije matrica ako mislimo podatke ponovno učitati. Također direktnim otvaranjem NPZ datoteke se ne mogu vidjeti sadržaji jer su kompresirani i

namijenjeni raspakiranju u Python okruženju, naspram TXT datoteke koju je moguće otvoriti i pregledati sadržaj. Glavni podatak na kraju algoritma je elitni genom ili najbolji pojedinac populacije u zadnjoj generaciji. Elitni genom se izdvaja iz populacije tako da se traži njegovo mjesto u populaciji preko najvećeg fitnesa. Naredbom *numpy.argmax(Fitnes)* izdvaja se redni broj genoma s maksimalnim fitnesom u populaciji. Za spremanje elitnog genoma u tekstualnu datoteku prvo je potrebno otvoriti takvi datoteku i dati joj ime naredbom *open('ImeDatoteke.txt', w).* Pri čemu atribut *w* označuje pisanje u datoteku ili eng. write. U datoteku se piše naredbom *write* ili sprema vektor ili matrica naredbom *numpy.savetxt().* NPZ datoteke se koriste za dvije namjene: prva je da spremi sve podatke potrebne za nastavak evolucijskog algoritma. Kao broj generacije, fitnesi populacije, podatke potrebne za prikaz rezultata u grafikone i samu populaciju. Druga namjena je da spremi sve podatke potrebne za reprodukciju hoda elitnog genoma, kao matricu elitnog genoma i povijest fitnesa kroz evolucijski algoritam. Naredba *numpy.savez()* omogućava stvaranje takve datoteke i njenim otvaranjem i zatvaranjem se brišu prethodni podaci i upisuju novi.

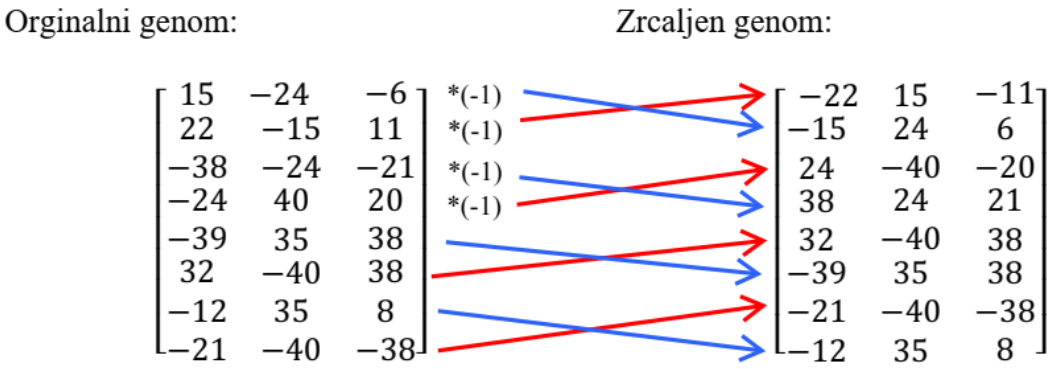
```
453
454  #Zapis konačnog elitnog genoma u txt datoteku
455  open('Elitni_gait.txt', 'w').close()#brisanje starih podataka iz datoteke
456  Argmax = np.argmax(Fitnes)
457  Elitni_gait = Populacija[:,:,Argmax]
458  f=open('Elitni_gait.txt','w')
459  z = repr(Generacija)
460  f.write('#Generacija:'+z+'\n')
461  z = repr(Argmax)
462  f.write('#Genom:'+z+'\n')
463  z = repr(float(Fitnes[Argmax]))
464  f.write('#Fitnes genoma:'+z+'\n')
465  np.savetxt(f, Elitni_gait, fmt='%1.5f',delimiter=',')
466  f.close()
467
468  #Spremanje podataka o populaciji ako želimo nastaviti EA
469  np.savez('Gen_Fit_Pop',
470           Generacija = Generacija,
471           Fitnes = Fitnes,
472           Suma_fitnesa_plot=Suma_fitnesa_plot,
473           Elitni_plot=Elitni_plot,
474           Populacija = Populacija)
475
476  #Spremanje podataka o elitnom genomu
477  np.savez('Elitni_gait',
478           Elitni_gait=Elitni_gait,
479           Fitnes=np.column_stack((Generacija_plot,Suma_fitnesa_plot,Elitni_plot)))
480
```

Slika 5.22. Spremanje podataka

Cijeli kod evolucijskog algoritma je priložen u prilogu pod [2].

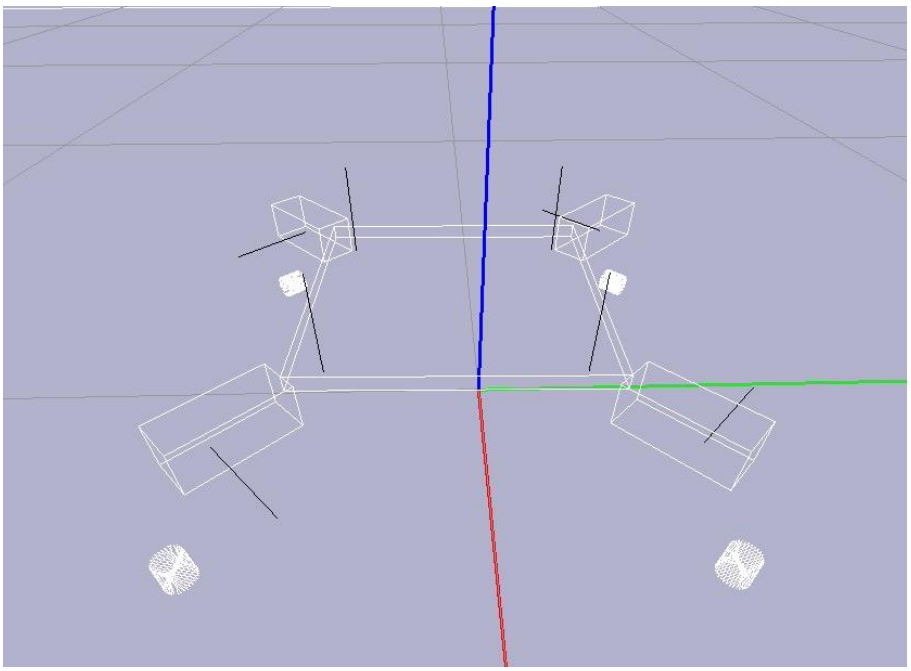
6. REZULTATI EVOLUCIJSKOG ALGORITMA

Evolucijski algoritam daje samo set pokreta jednog koraka, stoga da bi se vidjeli rezultati korak se mora ponavljati u petlji kako bi se razvio hod. Programski dio za reprodukciju hoda robota je dan u prilogu pod [3]. Osim ponavljanja koraka potrebno je kompenzirati hodanje tako da se svaki drugi korak 'zrcali' u matrici genoma. Proces zrcaljenja matrice je prikazan na slici (6.1.), i svodi se na zamjenu svakog drugog redka u matrici i promjene predznaka za sve motore ramena robota. Kako su motori određeni po retcima matrice tako da je prvi redak prednje desno rame, drugi redak je prednje lijevo rame, treći redak zadnje desno rame, četvrti redak zadnje lijevo rame i tako dalje istim poretkom i za koljena robota. Zamjenom svaka dva susjedna retka prebacujemo pokrete koji su originalno bili na desnoj strani na lijevu i obrnuto.



Slika 6.1. Zrcaljenje genoma

No to nije točno zrcaljenje hoda jer treba paziti i na orijentaciju zglobova, kako je prikazano na slici (6.2.) crnim linijama. Zglobovi koljena, zadnja četiri retka genoma, su jednako orijentirana, tj. izviru sa lijeve strane noge robota. To znači da ako je na desnoj strani vrijednost podizala nogu, zamjenama retka će se isto podizati i na lijevoj strani. No isto ne vrijedi i za ramena robota. Ako je vrijednost u originalnom genomu na desnoj strani pokretala nogu prema naprijed, zamjenom redaka sada će ista vrijednost pokretati lijevu nogu prema natrag. I zato je potrebno promijeniti predznak vrijednosti pokreta ramena robota kako bi mogli točno zrcaliti pokreti s obzirom na lijevu i desnu stranu robota. Zbog tog razlog, kako je na slici (6.1.), se prilikom promjene redaka sve vrijednosti ramena množe s -1.

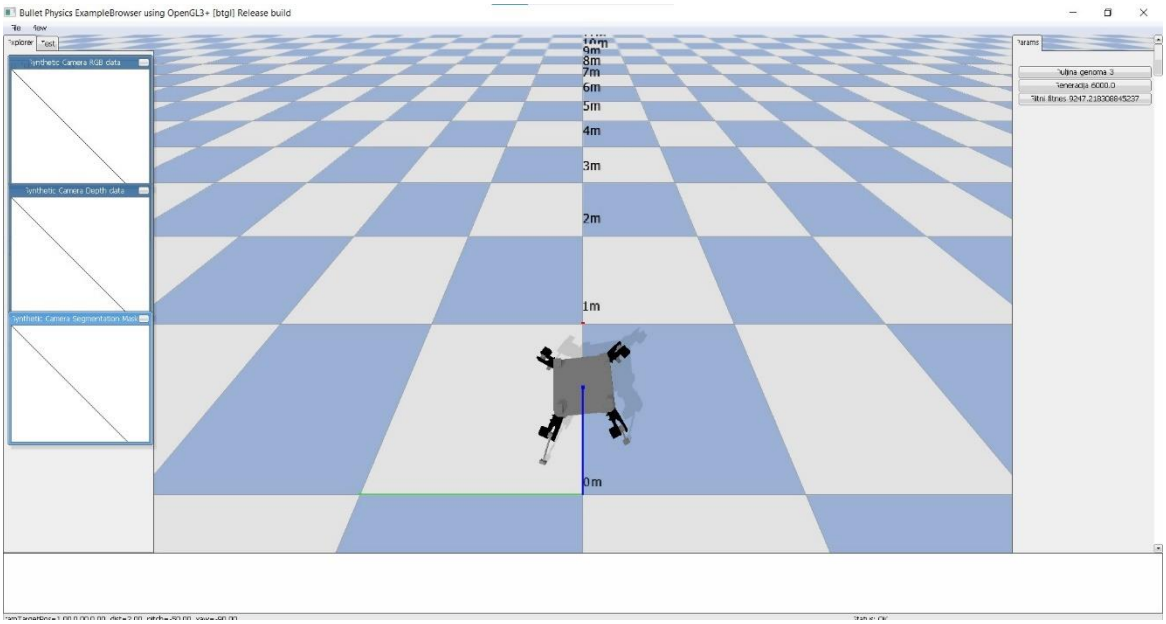


Slika 6.2. Osi zglobova robota

Program reprodukcije hoda tada pokreće dvije matrice u petlji. Jednu originalnu matricu genoma, zatim zrcaljenu, zatim originalnu, pa zrcaljenu... Dok ne obavi potreban broj iteracija. Rezultat je kompenziran hod u pravcu x-osi prikazan primjerom na slici (6.3.). Rezultat je postignut s konstantnim parametrima:

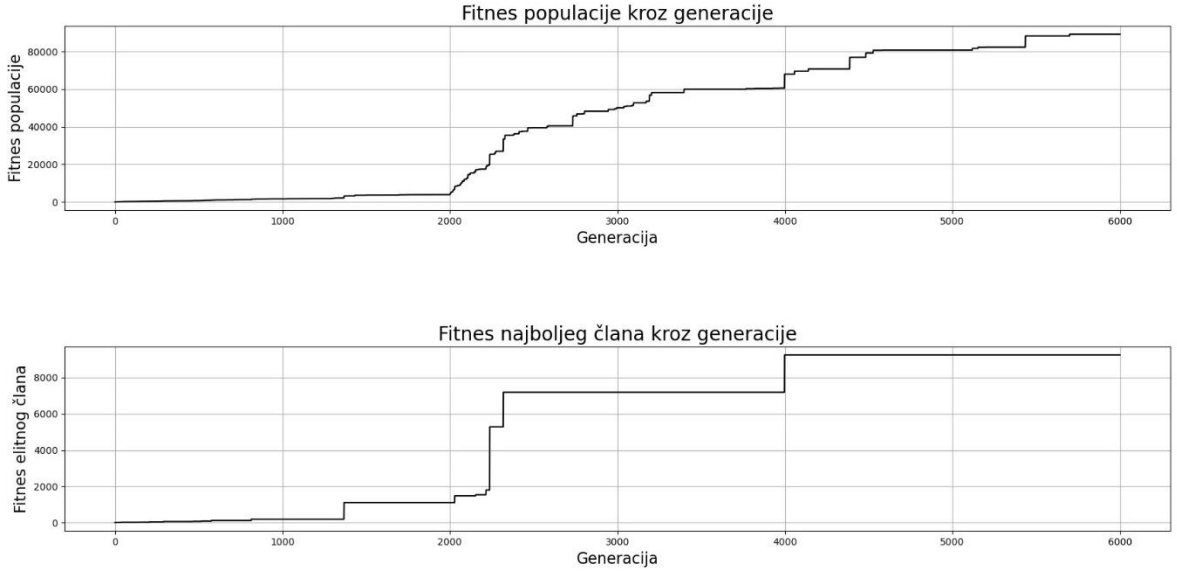
- veličina genoma $A = 3$
- vjerojatnost križanja $P_k = 0,4$
- vjerojatnost mutacije $P_m = 0,7$

Kroz 6000 generacija najbolji član je dosegao fitnes od 9247,22.



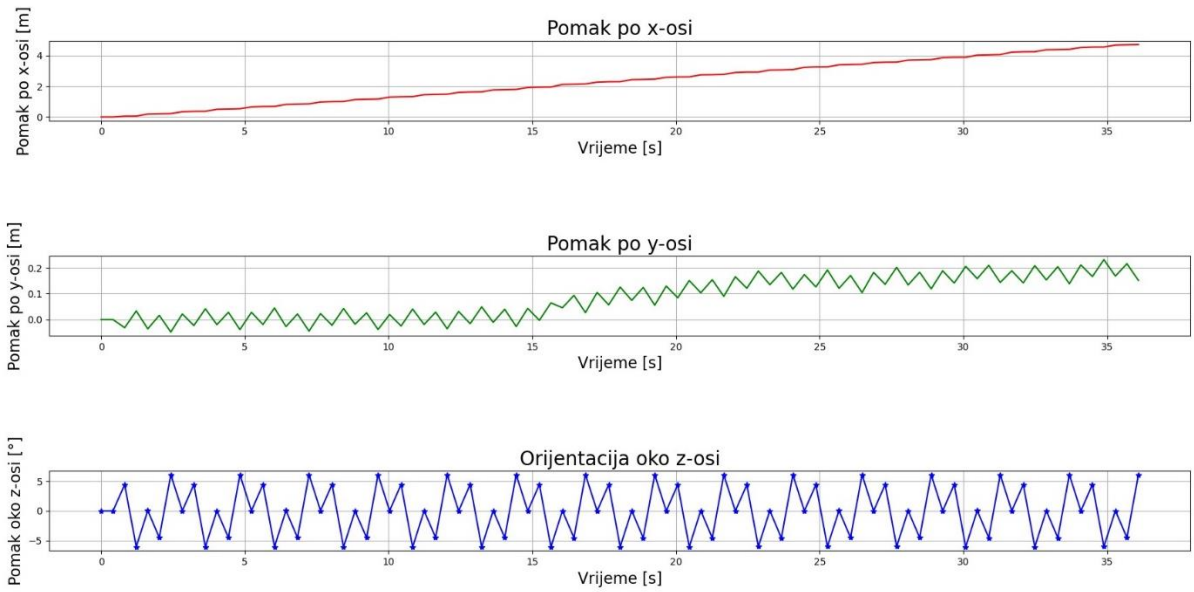
Slika 6.3. Primjer hoda

Nakon izvršenog hoda prikazuju se rezultati hoda i fitnes dobiven kroz generacije do trenutnog pojedinca. Slika (6.4.) prikazuje povijest fitnesa populacije i elitnog genoma.



Slika 6.4. Povijest fitnesa populacije i elitnog pojedinca

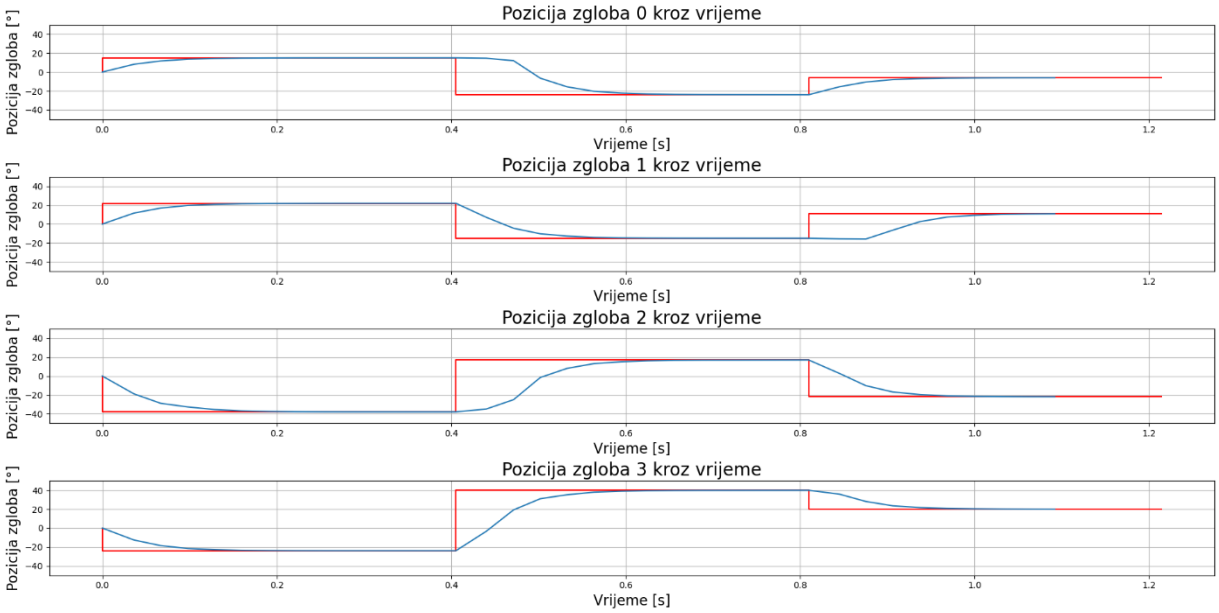
Od trenutka pokretanja do kraja hodanja se mjeri pozicija i orijentacija robota. Mjeri se prijeđen put po *x* i *y*-osi, te orijentacija oko *z*-osi tijela robota. Na slici (6.5.) je prikazan prijeđeni put s 30 koraka. Sa grafova je vidljivo da robot napreduje značajno po *x*-osi, no također odstupa po *y*-osi.



Slika 6.5. Pomak i orijentacija robota

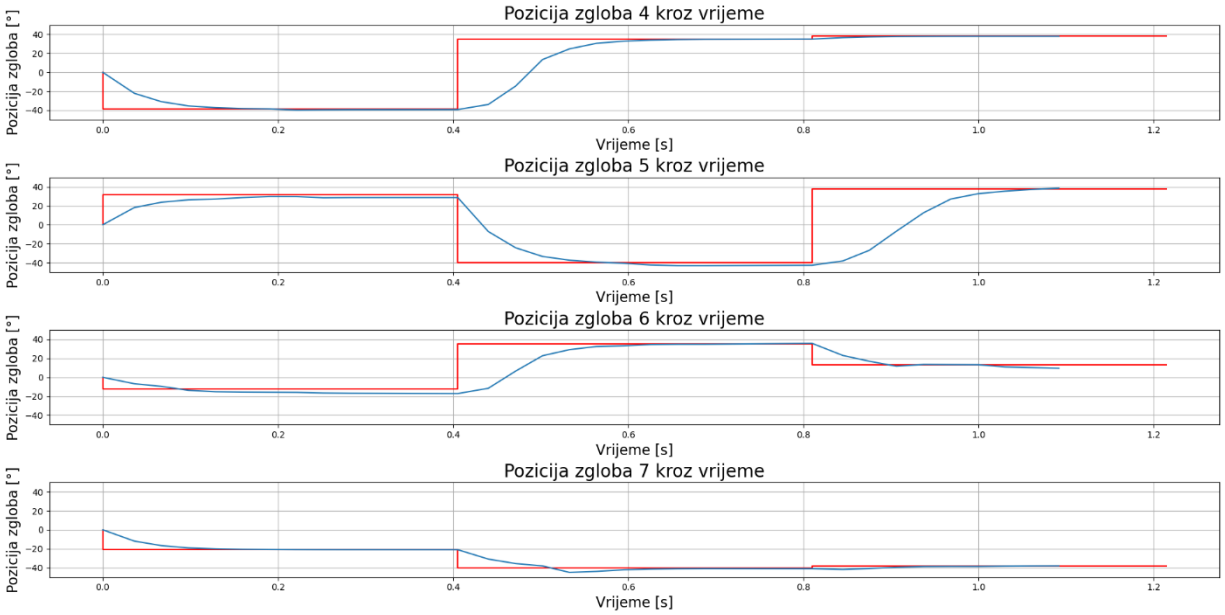
Pomaci, brzine i momenti zglobova se ne mjere cijelim putem jer nema smisla mjeriti istu stvar više puta. Već se mjere samo za jedan set korak, tj. za jedan genom.

Na slici (6.6.) su prikazani pomaci zglobova ramena robota. Crveno je označena referenca koju treba robot pratiti sa svojim zglobom, dok je plavo označena prava pozicija zgloba. Potrebno je neko vrijeme dok zglob dohvati referencu, kako zglob ubrzava i usporava pred poziciju i zato na početku svake nove reference ima nagli uspon/pad prema referenci.



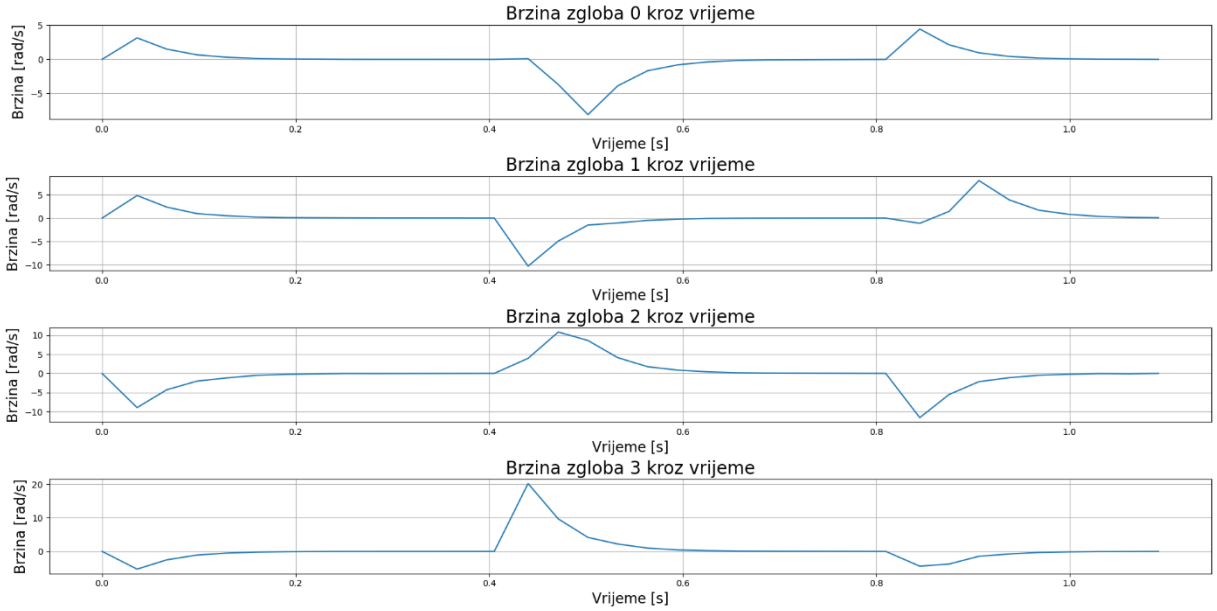
Slika 6.6. Pomaci zglobova ramena robota

Na slici (6.7.) su vidljivi pomaci koljena robota i kod njih je najznačajnije odstupanje od reference. To se uglavnom događa kada se cijeli robot osloni na jednu nogu i pod teretom se pomakne pozicija zgloba koljena. Tada je vidljivo odstupanje i regulator pokušava vratiti poziciju zgloba u zadanu vrijednost.

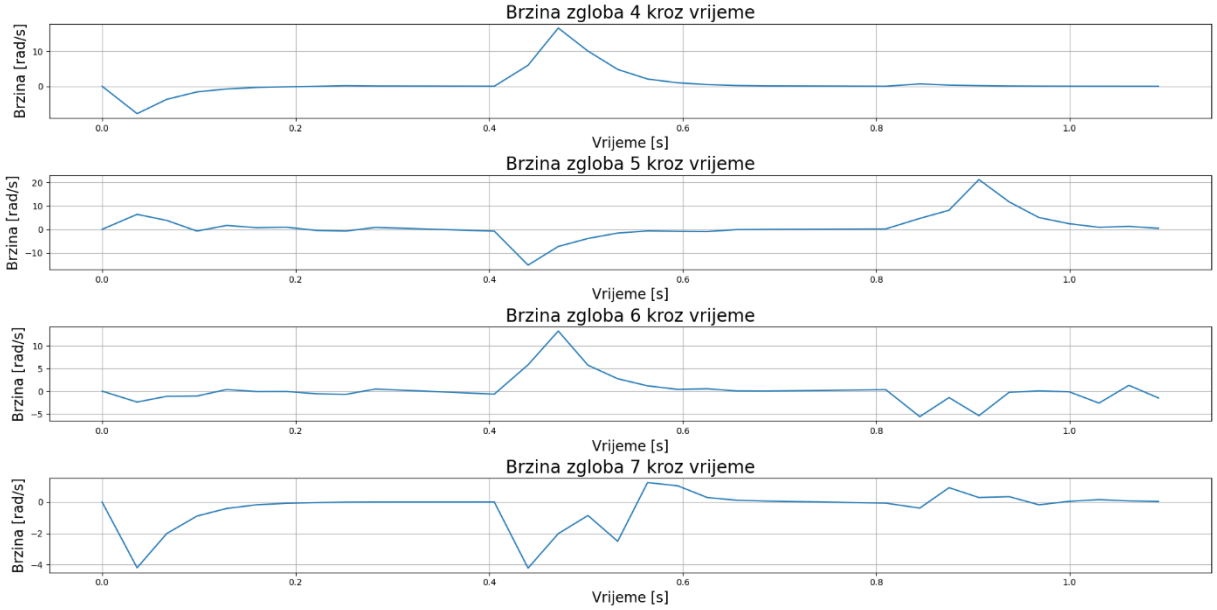


Slika 6.7. Pomaci zglobova koljena robota

Brzine zglobova su prikazane na slikama (6.8.) i (6.9.). Na grafovima je vidljivo da u trenutcima promjene pozicije iskaču vrijednosti brzine. Prilikom zaprimanja nove referentne pozicije zglob naglo ubrzava, te usporava kada je blizu reference, zbog toga su signali za brzine oštro ispupčeni.

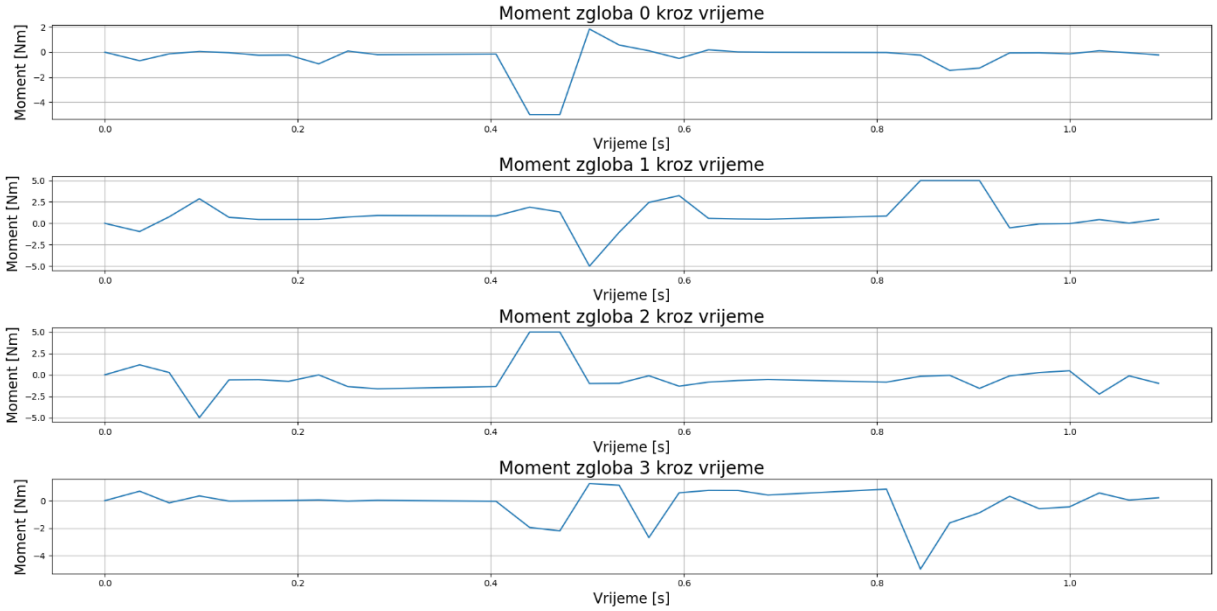


Slika 6.8. Brzine zglobova ramena robota

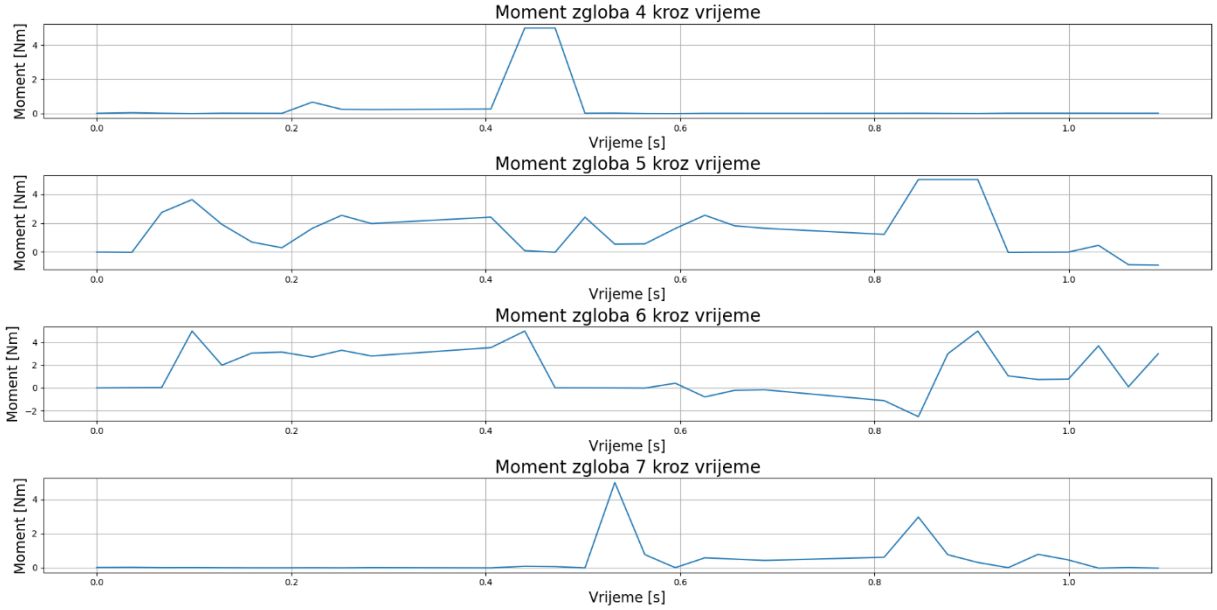


Slika 6.9. Brzine zglobova koljena robota

Na slikama (6.10.) i (6.11.) su prikazani grafovi momenata zglobova robota. Momenti su ograničeni u simulaciji na 5 Nm, no poželjno bi bilo da budu što manji. Jer se tada troši manje struje i energije na potrebne pokrete za fizički model robota i moguće je odabrati motore manjih snaga, a s time i manjih masa motora.



Slika 6.10. Momenti zglobova ramena robota



Slika 6.11. Momenti zglobova koljena robota

Kroz 30 koraka robot je prešao 4,66 m i odstupao od *x*-osi 0,195 m, dok su najveći zaokreti tijela od smjera kretanja bili 6,11° i -6,09°. Ekstremne vrijednosti brzina ramena ne prijelaze 20 rad/s, a momenti dosežu granice od 5 Nm. Ekstremi brzina koljena robota ne prijelaze 21 rad/s, a momenti isto kao i ramena dosežu 5 Nm. Robot sa većim prijeđenim putem sve više odstupa od *x*-osi, jedan razlog tome je nesavršenost početka hoda. Kada robot započinje svoj hod njemu su svi zglobovi postavljeni u nulti položaj i kada krene prvu sekvencu pomaka robot se tijelom malo, ali ipak primjetno rezultatima, zarotira oko svoje osi. Čime je u startu krenuo s nekom početnom greškom koja se samo povećava što je više koraka ostvario.

Ponovnim pokretanjem algoritma bilo pod istim parametrima ili drukčijim moguće je dobiti drukčije rezultate. Razlog je vrednovanje fitnesa, jer se fitness vrednuje po više varijabli: *X, Y* i *Ψ*. Iako se fitnes funkcija podesi da promjena po primarnoj varijabli donosi najveću promjenu u fitnesu, u ovom slučaju *X*. Dogodi pojedinci zapnu na nekoj sekundarnoj varijabli i iskazuju sličan fitnes uz drukčije kretanje i rezultate. Moguće je da se pojedinci više fokusiraju na minimiziranju kretnji po *Y* nego na kretanju prema naprijed po *X*.

Tako je novi pokušaj s parametrima:

$A = 2$

$B = 20$

$Pk = 0{,}4$

$Pm = 0{,}7$

je rezultirao fitnesom od 8324,03. Prijeđenim putem po *X*-u 1,57 m i po *Y*-u 0,16 m. Orijentacija *Ψ* je varirala između 3,63° i -3,58°.

Pokušaj s parametrima:

$A = 8$

$B = 20$

$Pk = 0{,}3$

$Pm = 0{,}7$

je rezultirao fitnesom od 6580,82. Prijeđenim putem po *X*-u 10,22 m i po *Y*-u 0,37 m. Orijentacija *Ψ* je varirala između 4,58° i -4,61°.

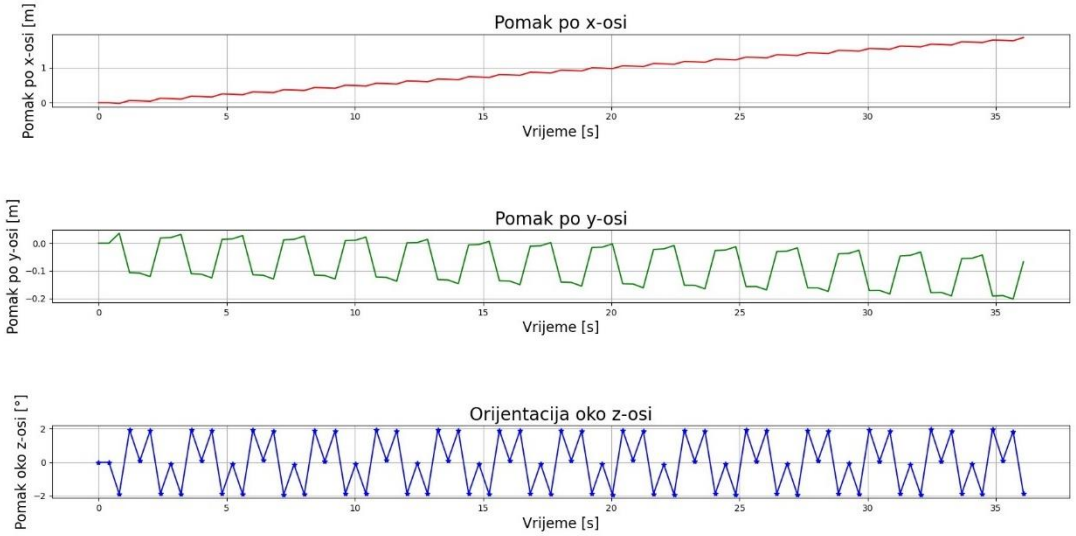
Neki populacije se više fokusiraju na točnost kretanja tako da manje odstupaju po poziciji i orijentaciji, kao drugi primjer pokušaja. Dok neke populacije se više fokusiraju na prijeđen put

i imaju veća odstupanja, kao treći primjer. Među više pokretanja evolucijskog algoritma potrebno je pronaći rješenje koje zadovoljava sve varijable, što je u ovom slučaju priložen prvi primjer.

7. USPOREDBA EVOLUIRANOG I PROGRAMIRANOG HODA

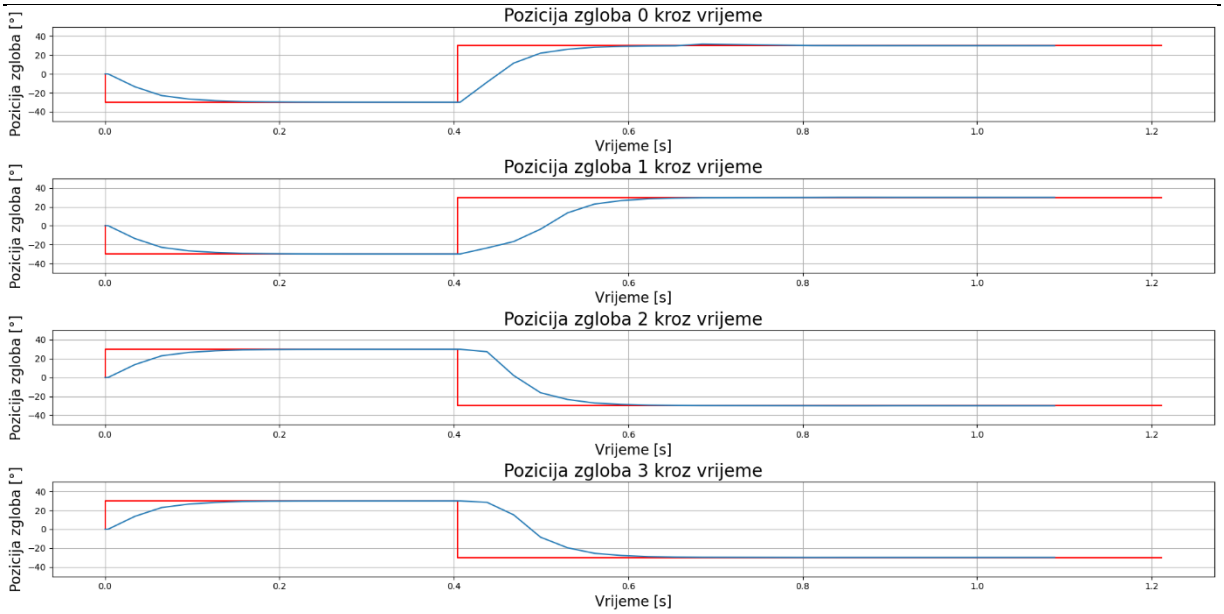
Hod robota nije ni potrebno tražiti evolucijskim algoritmima ako možemo bolje rezultate ručno napisati. No zato je potrebna usporedba, na slici (5.1.) je prikazana matrica pokreta koja je ručno napisana u kraće vremena nego što je bilo potrebno pisanje koda evolucijskog algoritma i njegovo izvršavanje. Usporedba rezultata je potrebna da se prikaže dali vrijedi utrošak više vremena u evolucijski algoritam radi dobivanja mogućeg kvalitetnijeg rezultata. Na slici (7.1.) je prikazan rezultat pomaka programiranog hoda.

Kroz 30 koraka robot je prešao put od 1,86 m po *x*-osi, dok je odstupao od *x*-osi 0,036 m. Zaokret tijela od smjera gibanja su bili 1,99° i -1,93°. Za jedan korak prijeđenog puta fitnes programiranog koraka iznosi 0,00216.

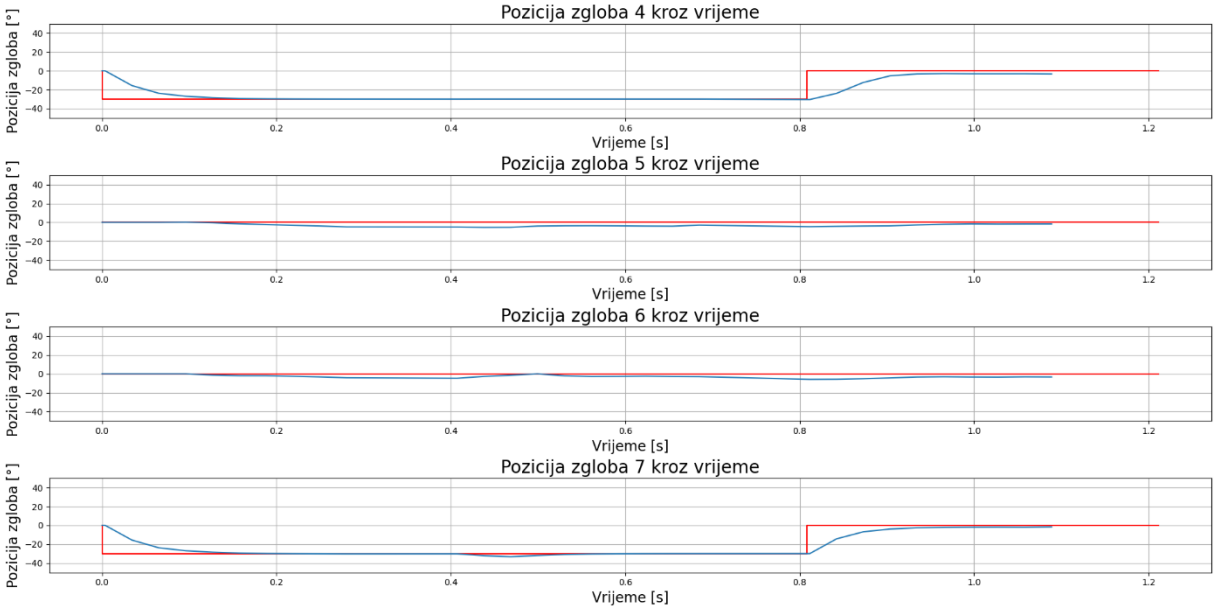


Slika 7.1. Pomak i orijentacija robota s programiranim hodom

Slika (7.2.) prikazuje pomake zglobova ramena robota, a slika (7.3.) pomake zglobova koljena robota. Također je vidljivo da najveća odstupanja su kod zglobova koljena kada su aktivirani pokreti zglobova ramena, tj., kada se terete zglobovi koljena.

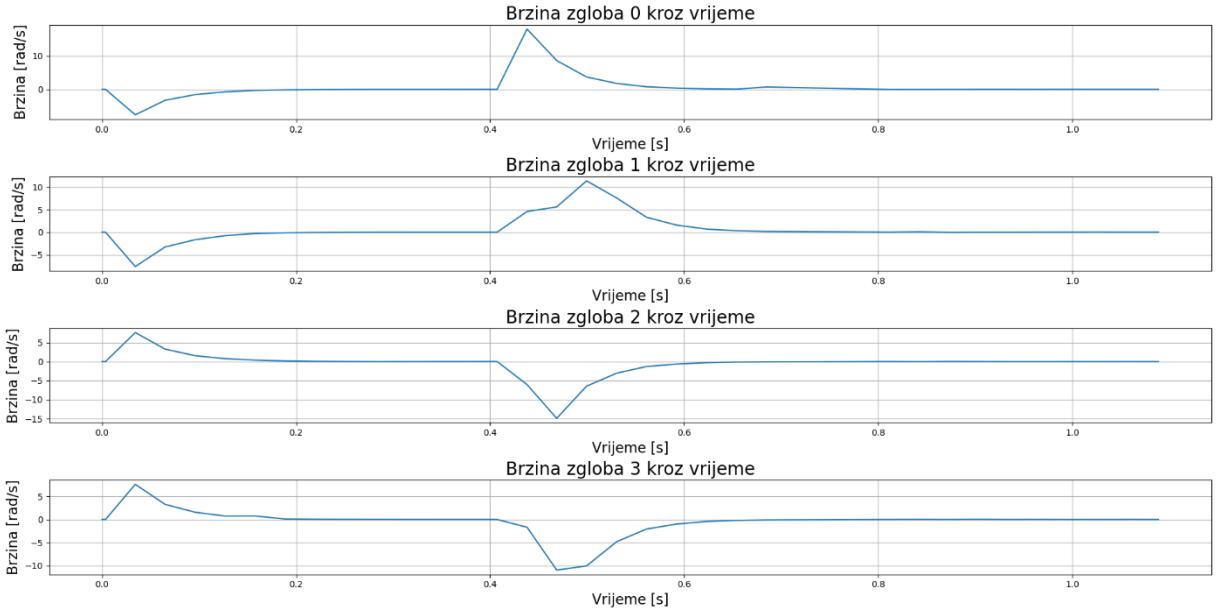


Slika 7.2. Pomaci ramena robota (programiran hod)

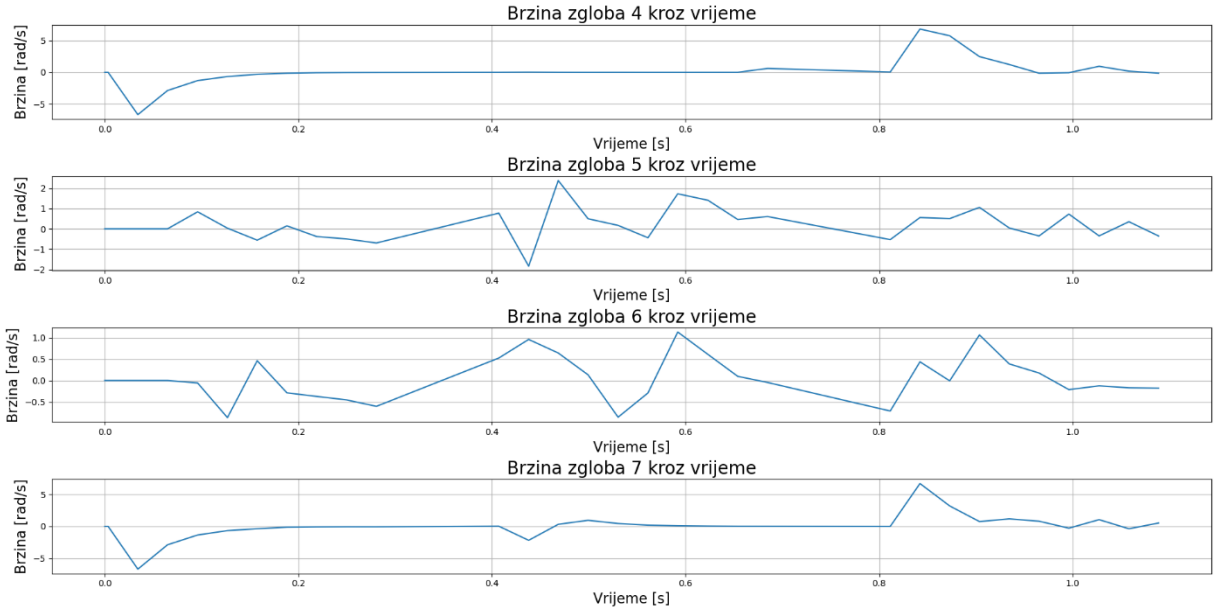


Slika 7.3. Pomaci koljena robota (programiran hod)

Brzine zglobova ramena dosežu ekstreme od 18,06 rad/s što je vidljivo na slici (7.4.). Brzine koljena imaju veće turbulencije, kako je vidljivo na slici (7.5.), te ekstremne vrijednosti mogu dosegnuti 11,37 rad/s.

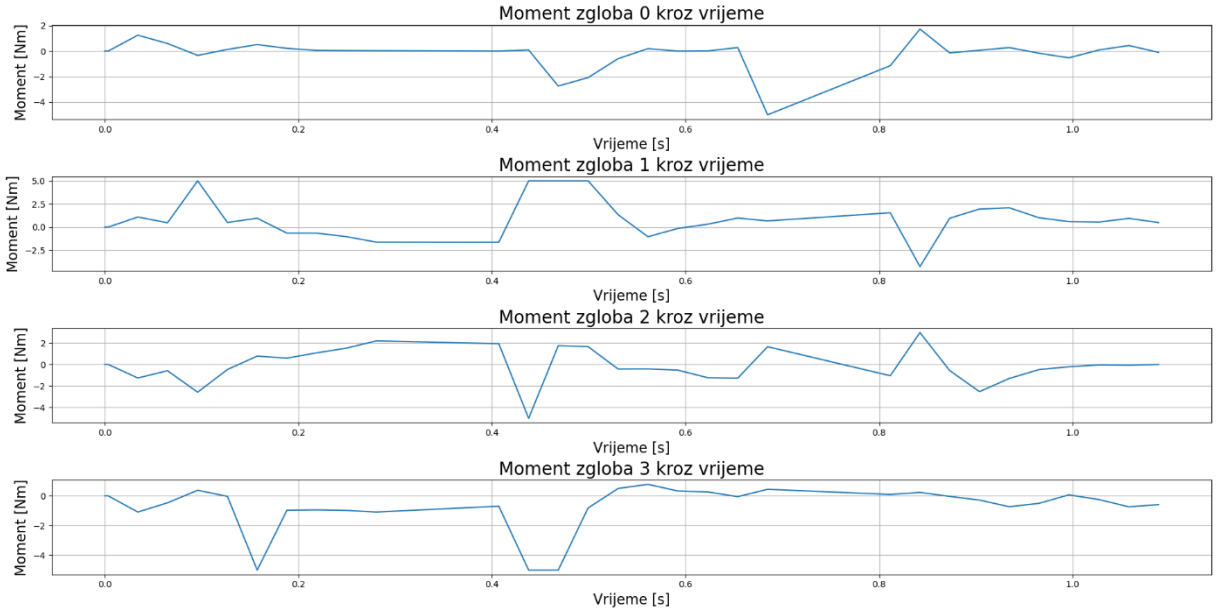


Slika 7.4. Brzine robota (programiran hod)

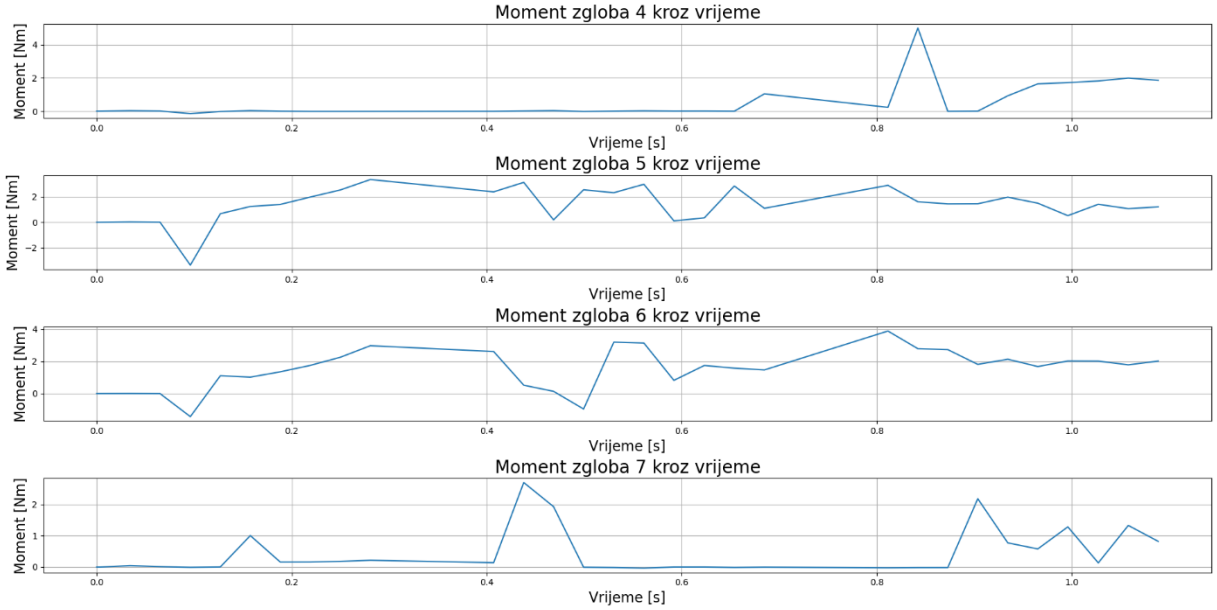


Slika 7.5. Brzine koljena (programiran hod)

Momenti zglobova ramena robota su prikazani na slici (7.6.), dok su momenti zglobova koljena prikazani na slici (7.7.). Ekstremi zglobova ramena dosežu granicu od 5 Nm, dok ekstremi zglobova koljena ne prelaze 0,77 Nm.



Slika 7.6. Momenti ramena robota (programiran hod)



Slika 7.7. Momenti koljena robota (programiran hod)

Usporedbom evoluiranog i programiranog hoda moguće je primijetiti da se evoluirani hod više ponaša impulsno, tj. koristi veće odskoke brzine da bi se pokretao. Prema slikama (6.5.) i (7.1.) je vidljivo da evoluirani hod više odstupa od x-osi, no programiran hod se više ljulja po stranama prilikom hodanja. Evoluirano hodanje se više zaokreće oko svoje osi, no manje se ljulja po stranama. Programirani hod je precizniji, no prešao je manji put nego evoluirani. Lako je naslutiti da će se s većim prijeđenim putem povećavati i odstupanje od smjera gibanja. Na kraju glavni faktor je prijeđen put robota. Evoluirani je prešao 4,66 m u potrebnom smjeru gibanja, dok je programiran hod prešao 1,86 m. Evoluirani hod koristi veće brzine pokreta zglobova, no zauzvrat rezultira kvalitetnijem hodu po ravnoj površini. Zaključno evoluirani hod prevali veći put uz malo veća odstupanja što je zanemarivo u usporedbi s programiranim hodom robota.

8. PROBLEMI SIMULATORA I EVOLUCIJSKOG ALGORITMA

Evolucijski algoritam dolazi do približnog rješenja hoda no i on ima nekoliko nedostataka. Kako je algoritmu ključan dio Pybullet simulator, sve nedostatke koje ima simulator su na kraju isprepletene u krajnji rezultat.

8.2. Ograničenja Pybullet simulatora

Pybullet modul je besplatan i fleksibilan iako dolazi s nekoliko ograničenja, prvo što je primjetno čak i u početku rada je nedostatak dokumentacije. Na internetu se nalazi manji dokument s pojašnjenjima osnovnih funkcija [5], no nije detaljno prorađen i zapravo zasad ne postoji kompletna dokumentacija programa. Odgovore na neka pitanja se uglavnom pronalaze po forumima i po razgovorima drugih korisnika koji imaju isti ili sličan problem. Ili problem moramo riješiti sami iterativno isprobavajući razna moguća rješenja, no to vrlo brzo postane dugotrajno i zamorno. Nedostatak detaljne dokumentacije ima veliki učinak na razvoj simulacije, iako je simulator dosta fleksibilan što se tiče mogućih primjena, čim je potrebno razraditi nešto u detalje potrebnih odgovora nema ili su skriveni na nekom forumu na internetu.

Simulator ima ograničen broj faceta i potrebno je paziti koje i koliko oblika želimo u jednoj simulaciji. Zbog velikog broja robota u jednoj simulaciji i kompliciranе 3D strukture tijela robota, bilo je potrebno pojednostavniti strukturu. No ako se ne pojednostavni struktura i pokrene simulacija i ako prelazimo dopušten broj faceta, simulacija neće stati ili javiti grešku već će nastaviti s programom i s manjkom ili nepotpunim robotima. Može se dogoditi da uopće ne učita tijela robota ili polovično učita i tako ostavi nekog robota bez noge ili dvije.

Mjerne jedinice također nisu nigdje naznačene i koristi se SI sustav mjernih jedinica. To znači da ako učitamo URDF model robota i ako smo ga definirali u mm, npr. ako je duljina stranice tijela robota 300 mm, simulator će učitati robota kao da mu je stranica duga 300 m. Zato je potrebno paziti na mjerne jedinice prilikom stvaranja modela i skalirati ih na potrebnu mjeru bilo u URDF modelu ili Python skripti. Moguće je raditi u simulatoru i bez skaliranja vrijednosti, no tada je potrebno podesiti ostale parametre koji se koriste, npr. silu gravitacije.

Kod mirovanja robota dolazi do čudnog slučaja "poskakivanja" ili vibriranja po plohi. Naime ako u koliziju dolaze dva tijela i jedan je jako manji od drugog, primjer tu je noga robota i ploha

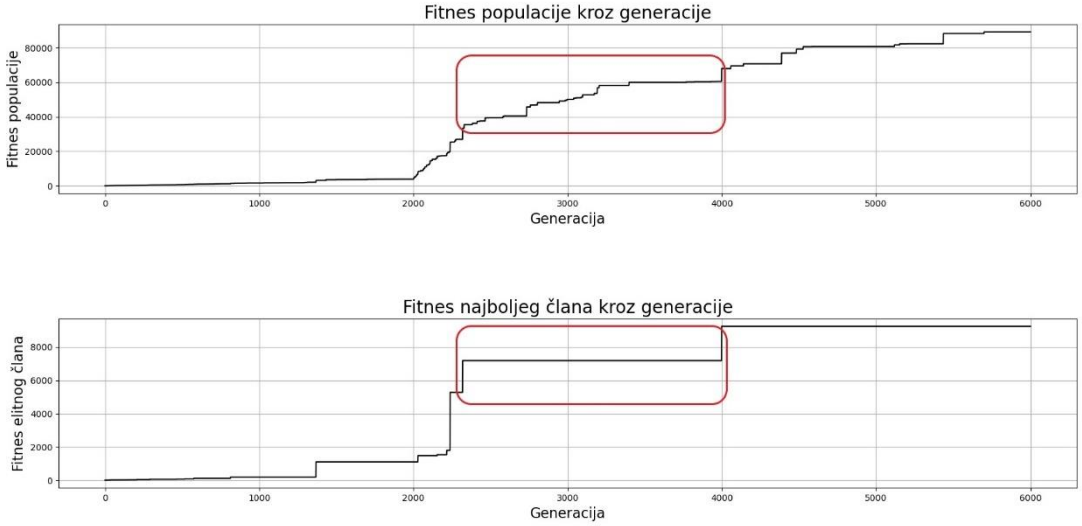
na kojoj se nalazi, manje tijelo zadire u veće i naglo se izbacuje zbog prepoznate sile kolizije. Ali zato što su to jako mali volumeni koji se preklapaju, teško je prepoznati koliziju i javljaju sile koje odbijaju dva tijela. Zbog toga se zna javiti pojava laganog vibriranja robota po plohi i robot zna klizati sa svoje početne pozicije i orijentacije. Iako su vrijednosti promjene male, znaju narušiti vrednovanje fitnesa robota ako predugo roboti miruju na početku svojeg koraka.

Trenje je također potrebno spomenuti, iako se roboti pokušavaju simulirati u što stvarnijim uvjetima, algoritam ne bi mogao funkcionirati bez daška idealnih uvjeta. Tako su koeficijenti trenja podešeni da su 100% u linearnim pokretima, a 0 ili neke manje vrijednosti u rotacijskim pokretima. Da nisu tako podešena javljalo bi se klizanje robota po plohi jer je ploha idealno ravna ili se robot ne bi mogao pomaknuti jer ne bi mogao klizeći rotirati svoju nogu po plohi. Za što stvarniji prikaz hoda robota potrebno je iterativno isprobavati različite vrijednosti, no to možda nije moguće ako kroz simulaciju uvijek imamo drukčiju sekvencu pokreta.

8.3. Ograničenja evolucijskog algoritma

Fitnes je mjera uspješnosti pojedinca i kroz generacije on mora rasti ili stagnirati, ne smije padati. To znači da moraju preživljavati pojedinci koji imaju bolji fitnes od ostalih u populaciji [14, 15]. No tu se i javlja manji problem ili ograničenje u algoritmu, specifičnije pojava stagnacije zbog nedostatka raznolikosti u populaciji [16]. Početne generacije su generirane nasumično i svaki pojedinac je različit od ostalih u populaciji. No kroz generacije preživljavaju samo najbolji pojedinci i imaju veće šanse da postanu roditelji. Kroz generacije polako se gasi raznolikost populacije tako da se zadržavaju pojedinci koji su slični najboljem, elitnom, pojedincu. Time je ograničeno upravljanje raznolikosti populacije s parametrom mutacije i križanja, no najviše mutacijom. Zbog toga je potrebno postaviti vjerojatnost mutacije na ili iznad 0,5.

Kako je prikazano na slici (8.1.), gdje je prikazano stagniranje fitnesa elitnog genoma no unatoč tome raste fitnes populacije. To pokazuje da u novu populaciju ulaze pojedinci koji su slični elitnome no imaju manji ili jednak fitnes kao i elitni genom. Na početku algoritma usponi fitnesa elitnog genoma su mnogobrojniji ali manjih iznosa. Kako se algoritam nastavlja na većim generacijama prorjeđuju se usponi fitnesa i učestalije je stagniranje, no zato su usponi fitnesa veći.



Slika 8.1. Stagnacija i gubitak raznovrsnosti

9. ZAKLJUČAK

Evolucijski algoritmi su moćan alat za pretragu rješenja u problemima koji nemaju točan odgovor. Od optimiranja oblika konstrukcija, planiranja minimalnog puta ili u ovom slučaju traženja optimalnog hoda robota. U ovom radu je pokazano da je moguće dobiti korisne rezultate hoda robota pomoću evolucijskog algoritma. Naspram programiranog hoda, evolucijski rezultat bio optimalniji uz isti broj mogućih pokreta. Unatoč tome postoji prostor za unaprijeđenje već napomenutih problema, kao gubitak raznovrsnosti populacije, korištenjem različitim metoda evaluacije ili odabira roditelja. Nada je da se ovim načinom može nastaviti razvijati i upotrebljavati evoluirani robotski hod, bilo evolucijom na različitim terenima ili dodatkom regulacije i upravljanja za navigaciju robota. Evolucijski algoritmi još nisu čvrsto zastupljeni u tvrtkama i često preko evolucijskih odabiru deterministička rješenja. No s obzirom na velike mogućnosti i potencijal evolucijskih algoritama za rješavanje problema, vjerujem da će se češće pojavljivati evolucijska rješenja u tehničkim i ostalim strukama. Pybullet je također dobar alat za razvoj robotike i svakim danom raste broj korisnika i ažuriranja kvalitete simulatora. Unatoč nekim nedostacima, Pybullet je odličan program za simuliranje hoda robota i već postojeći primjeri govore o mogućnosti primjene u bliskoj i daljnjoj budućnosti razvoja robotike.

LITERATURA

[1] Numpy, https://numpy.org/, 12.1.2024.

[2] Time, https://docs.python.org/3/library/time.html, 12.1. 2024.

[3] Threading, https://docs.python.org/3/library/threading.html, 16.1.2024.

[4] Matplotlib, https://matplotlib.org/, 20.1. 2024.

[5] Pybullet, https://pybullet.org/wordpress/, 11.10. 2023.

[6] Anaconda, https://www.anaconda.com/, 3.10. 2023.

[7] Spyder, https://www.spyder-ide.org/, 3.10. 2023.

[8] Visual Studio Code, https://code.visualstudio.com/, 5.11.2023.

[9] P. Ćurković, Predavanja iz kolegija Evolucijski algoritmi, 2023.

[10] r2d2.urdf , https://github.com/bulletphysics/bullet3/blob/master/data/r2d2.urdf, 16.10.2023.

[11] ROS create your own URDF file, http://wiki.ros.org/urdf/Tutorials/Create%20your%20own%20urdf%20file, 16.10.2023.

[12] ROS link element, http://wiki.ros.org/urdf/XML/link, 16.10.2023.

[13] ROS joint element, http://wiki.ros.org/urdf/XML/joint, 16.10.2023.

[14] Ćurković, Petar ; Čehulić, Lovro. Evolutionary Planner of 3D Trajectories. 2018. str. 400-405 doi: 10.2507/29th.daaam.proceedings.058

[15] D. Grundler, Evolucijski algoritmi: Pobude i načela, Zagreb, 2001.

[16] Ćurković, Petar ; Čehulić, Lovro. Diversity maintenance for efficient robot path planning // Applied sciences (Basel), 10 (2020), 5; 10051721, 15. doi: 10.3390/app10051721

PRILOZI

I. URDF kod robota

```
<?xml version="1.0"?>
<robot name="spidy">

    <!--TIJELO ROBOTA-->

  <link name="base_link">
    <visual>
                <origin xyz = "0 0 0" rpy ="0 0 0"/>
      <geometry>
                    <mesh filename="package://STL_djelovi/Tijelo.stl"
scale="0.001 0.001 0.001"/>
                </geometry>
      <material name="sivo">
        <color rgba="0.5 0.5 0.5 1"/>
      </material>
    </visual>

    <collision>
                <origin xyz = "0 0 0" rpy ="0 0 0"/>
      <geometry>
                    <box size="0.3 0.3 0.015"/>
      </geometry>
    </collision>

    <inertial>
      <mass value="2.46"/>
```

```
                    <origin xyz="0 0 0.01" rpy="0 0 0"/>
        <inertia ixx="0.034" ixy="0" ixz="0" iyy="0.039" iyz="0" izz="0.067"/>
    </inertial>
  </link>

    <!--NOGE I ZGLOBOVI NOGE ROBOTA-->

    <joint name ="desno_rame_1" type ="revolute">
            <origin xyz ="0.121 -0.121 -0.01134" rpy ="0 0 -0.7854"/>
            <parent link ="base_link"/>
            <child link ="desna_noga_1"/>
            <axis xyz ="0 0 1"/>
            <limit lower ="-0.768" upper ="0.768" effort ="10" velocity ="10"/>
    </joint>

    <link name="desna_noga_1">
    <visual>
                    <origin  xyz="0 0 0" rpy ="0 0 0"/>
      <geometry>
                        <mesh filename="package://STL_djelovi/Noga.stl" scale="0.001
0.001 0.001"/>
      </geometry>
      <material name="crno">
        <color rgba="0 0 0 1"/>
      </material>
    </visual>

    <collision>
                    <origin  xyz="0.096 0 0" rpy="0 0 0"/>
```

```
        <geometry>
                        <box size="0.108 0.044 0.044"/>
        </geometry>
      </collision>

      <inertial>
        <mass value="0.67"/>
        <origin xyz="0.097 -0.003 -0.012" rpy="0 0 0"/>
        <inertia ixx="0" ixy="0" ixz="0" iyy="0.001" iyz="0" izz="0.001"/>
      </inertial>
    </link>

        <joint name ="lijevo_rame_1" type ="revolute">
                <origin xyz ="0.121 0.121 -0.01134" rpy ="0 0 0.7854"/>
                <parent link ="base_link"/>
                <child link ="lijeva_noga_1"/>
                <axis xyz ="0 0 1"/>
                <limit lower ="-0.768" upper ="0.768" effort ="10" velocity ="10"/>
        </joint>

        <link name="lijeva_noga_1">
                <visual>
                        <origin  xyz="0 0 0" rpy ="0 0 0"/>
                        <geometry>
                                <mesh filename="package://STL_djelovi/Noga.stl" scale="0.001
0.001 0.001"/>
                        </geometry>
                        <material name="crno">
                                <color rgba="0 0 0 1"/>
```

```
                </material>
            </visual>

            <collision>
                <origin  xyz="0.096 0 0" rpy="0 0 0"/>
                <geometry>
                    <box size="0.108 0.044 0.044"/>
                </geometry>
            </collision>

            <inertial>
                <mass value="0.67"/>
                <origin xyz="0.097 -0.003 -0.012" rpy="0 0 0"/>
                <inertia ixx="0" ixy="0" ixz="0" iyy="0.001" iyz="0" izz="0.001"/>
            </inertial>
        </link>

        <joint name ="desno_rame_2" type ="revolute">
            <origin xyz ="-0.121 -0.121 -0.01134" rpy ="0 0 -2.356"/>
            <parent link ="base_link"/>
            <child link ="desna_noga_2"/>
            <axis xyz ="0 0 1"/>
            <limit lower ="-0.768" upper ="0.768" effort ="10" velocity ="10"/>
        </joint>

        <link name="desna_noga_2">
            <visual>
                <origin  xyz="0 0 0" rpy ="0 0 0"/>
                <geometry>
```

```
                    <mesh filename="package://STL_djelovi/Noga.stl" scale="0.001
0.001 0.001"/>
                </geometry>
                <material name="crno">
                    <color rgba="0 0 0 1"/>
                </material>
            </visual>

            <collision>
                <origin  xyz="0.096 0 0" rpy="0 0 0"/>
                <geometry>
                    <box size="0.108 0.044 0.044"/>
                </geometry>
            </collision>

            <inertial>
                <mass value="0.67"/>
                <origin xyz="0.097 -0.003 -0.012" rpy="0 0 0"/>
                <inertia ixx="0" ixy="0" ixz="0" iyy="0.001" iyz="0" izz="0.001"/>
            </inertial>
        </link>

        <joint name ="lijevo_rame_2" type ="revolute">
            <origin xyz ="-0.121 0.121 -0.01134" rpy ="0 0 2.356"/>
            <parent link ="base_link"/>
            <child link ="lijeva_noga_2"/>
            <axis xyz ="0 0 1"/>
            <limit lower ="-0.768" upper ="0.768" effort ="10" velocity ="10"/>
        </joint>
```

```
<link name="lijeva_noga_2">
        <visual>
                <origin  xyz="0 0 0" rpy ="0 0 0"/>
                <geometry>
                        <mesh filename="package://STL_djelovi/Noga.stl" scale="0.001
0.001 0.001"/>
                </geometry>
                <material name="crno">
                        <color rgba="0 0 0 1"/>
                </material>
        </visual>

        <collision>
                <origin  xyz="0.096 0 0" rpy="0 0 0"/>
                <geometry>
                        <box size="0.108 0.044 0.044"/>
                </geometry>
        </collision>

        <inertial>
                <mass value="0.67"/>
                <origin xyz="0.097 -0.003 -0.012" rpy="0 0 0"/>
                <inertia ixx="0" ixy="0" ixz="0" iyy="0.001" iyz="0" izz="0.001"/>
        </inertial>
</link>

<!--PRSTI I ZGLOBOVI PRSTIJU ROBOTA-->
<joint name ="desni_zglob_1" type ="revolute">
```

```
            <origin xyz ="0.114 0 -0.012" rpy ="0 -0.5 0"/>
            <parent link ="desna_noga_1"/>
            <child link ="desni_prst_1"/>
            <axis xyz ="0 1 0"/>
            <limit effort="10" velocity="10"/>
            <limit upper ="1.57" lower ="-1.57"/>
        </joint>

        <link name="desni_prst_1">
            <visual>
                <origin  xyz="0 0 0" rpy ="0 0 0"/>
                <geometry>
                    <mesh  filename="package://STL_djelovi/Prst.stl"  scale="0.001
0.001 0.001"/>
                </geometry>
                <material name="sivo">
                    <color rgba="0.5 0.5 0.5 1"/>
                </material>
            </visual>

            <collision>
                <origin  xyz="0 0 -0.125" rpy ="1.57 0 0"/>
                <geometry>
                    <cylinder radius="0.015" length="0.03"/>
                </geometry>
            </collision>

            <inertial>
                <mass value="0.031"/>
```

```
                <origin xyz="0 0 -0.059" rpy="0 0 0"/>
                <inertia ixx="0" ixy="0" ixz="0" iyy="0" iyz="0" izz="0"/>
            </inertial>
        </link>

        <joint name ="lijevi_zglob_1" type ="revolute">
            <origin xyz ="0.114 0 -0.012" rpy ="0 -0.5 0"/>
            <parent link ="lijeva_noga_1"/>
            <child link ="lijevi_prst_1"/>
            <axis xyz ="0 1 0"/>
            <limit effort="100" velocity="100"/>
            <limit upper ="1.57" lower ="-1.57"/>
        </joint>

        <link name="lijevi_prst_1">
            <visual>
                <origin  xyz="0 0 0" rpy ="0 0 0"/>
                <geometry>
                    <mesh  filename="package://STL_djelovi/Prst.stl"  scale="0.001
0.001 0.001"/>
                </geometry>
                <material name="sivo">
                    <color rgba="0.5 0.5 0.5 1"/>
                </material>
            </visual>

            <collision>
                <origin  xyz="0 0 -0.125" rpy ="1.57 0 0"/>
                <geometry>
```

```
                        <cylinder radius="0.015" length="0.03"/>
                    </geometry>
                </collision>

                <inertial>
                    <mass value="0.031"/>
                    <origin xyz="0 0 -0.059" rpy="0 0 0"/>
                    <inertia ixx="0" ixy="0" ixz="0" iyy="0" iyz="0" izz="0"/>
                </inertial>
            </link>

            <joint name ="desni_zglob_2" type ="revolute">
                <origin xyz ="0.114 0 -0.012" rpy ="0 -0.5 0"/>
                <parent link ="desna_noga_2"/>
                <child link ="desni_prst_2"/>
                <axis xyz ="0 1 0"/>
                <limit effort="10" velocity="10"/>
                <limit upper ="0.768" lower ="-0.768"/>
            </joint>

            <link name="desni_prst_2">
                <visual>
                    <origin  xyz="0 0 0" rpy ="0 0 0"/>
                    <geometry>
                        <mesh  filename="package://STL_djelovi/Prst.stl"  scale="0.001
0.001 0.001"/>
                    </geometry>
                    <material name="sivo">
                        <color rgba="0.5 0.5 0.5 1"/>
```

```
                </material>
            </visual>

            <collision>
                <origin  xyz="0 0 -0.125" rpy ="1.57 0 0"/>
                <geometry>
                    <cylinder radius="0.015" length="0.03"/>
                </geometry>
            </collision>

            <inertial>
                <mass value="0.031"/>
                <origin xyz="0 0 -0.059" rpy="0 0 0"/>
                <inertia ixx="0" ixy="0" ixz="0" iyy="0" iyz="0" izz="0"/>
            </inertial>
    </link>

    <joint name ="lijevi_zglob_2" type ="revolute">
            <origin xyz ="0.114 0 -0.012" rpy ="0 -0.5 0"/>
            <parent link ="lijeva_noga_2"/>
            <child link ="lijevi_prst_2"/>
            <axis xyz ="0 1 0"/>
            <limit effort="10" velocity="10"/>
            <limit upper ="0.768" lower ="-0.768"/>
    </joint>

    <link name="lijevi_prst_2">
            <visual>
                <origin  xyz="0 0 0" rpy ="0 0 0"/>
```

```
                    <geometry>
                        <mesh  filename="package://STL_djelovi/Prst.stl"  scale="0.001
0.001 0.001"/>
                    </geometry>
                    <material name="sivo">
                        <color rgba="0.5 0.5 0.5 1"/>
                    </material>
                </visual>

                <collision>
                    <origin  xyz="0 0 -0.125" rpy ="1.57 0 0"/>
                    <geometry>
                        <cylinder radius="0.015" length="0.03"/>
                    </geometry>
                </collision>

                <inertial>
                    <mass value="0.031"/>
                    <origin xyz="0 0 -0.059" rpy="0 0 0"/>
                    <inertia ixx="0" ixy="0" ixz="0" iyy="0" iyz="0" izz="0"/>
                </inertial>
            </link>

</robot>
```

II. Evolucijski algoritam hoda robota

```
#  Library import---------------------------------------------------------
import pybullet as p
import time
import pybullet_data
import numpy as np
import matplotlib.pyplot as plt
#-------------------------------------------------------------------------

#  Postavke simulacije----------------------------------------------------
if p.isConnected()==1:
   p.disconnect()
physicsClient = p.connect(p.GUI)
#physicsClient = p.connect(p.DIRECT)
Povezanost = p.getConnectionInfo(physicsClient)
p.setAdditionalSearchPath(pybullet_data.getDataPath())
p.setGravity(0,0,-9.81)

p.setRealTimeSimulation(1)

p.configureDebugVisualizer(p.COV_ENABLE_MOUSE_PICKING,0)
p.configureDebugVisualizer(p.COV_ENABLE_RGB_BUFFER_PREVIEW,0)
p.configureDebugVisualizer(p.COV_ENABLE_DEPTH_BUFFER_PREVIEW,0)
p.configureDebugVisualizer(p.COV_ENABLE_SEGMENTATION_MARK_PREVIEW,0)
#-------------------------------------------------------------------------

#  Parametri evolucijskog algoritma
A = 20
```

```
B = 20
Pk = 0.2
Pm = 0.8
Iteracija = 1000
Pozicija_max = 40

Nastavak = False

#Podaci iz prošlog pokretanja EA ako želimo nastaviti s prošlom populacijom
if Nastavak == True:
    Data = np.load('84_3_Gen_Fit_Pop.npz')
    Data_gen = Data['Generacija']
    Data_fit = Data['Fitnes']
    Data_sum = Data['Suma_fitnesa_plot']
    Data_elite = Data['Elitni_plot']
    Data_pop = Data['Populacija']
#------------------------------------------------------------------------

#   Učitavanje robota i objekata-----------------------------------------------
planeId = p.loadURDF('plane.urdf')
CrockoStartPos = [0,0,0.2]
CrockoStartOrientation = p.getQuaternionFromEuler([0,0,0])

Crocko = []
for i in range(B):
    Crocko.append(p.loadURDF('Crocko.urdf',
                             CrockoStartPos,
                             CrockoStartOrientation))
    CrockoStartPos += np.array([0,1,0])
```

```
    p.addUserDebugText(text="Robot {}".format(i+1),
                       textPosition=[0,i,0.2],
                       textColorRGB=[0,0,0],
                       textSize=1,
                       lifeTime=0)
    #pokazuje smjer kretanja
    p.addUserDebugLine([0,i,0],[100,i,0],[1,0,0],lineWidth=3,lifeTime=0)

#------------------------------------------------------------------------

#   Informacije o zglobovima i link-ovima-------------------------------------
nJoints = p.getNumJoints(Crocko[0])
jointCrockoToId = {}
for i in range(nJoints):
  jointInfo = p.getJointInfo(Crocko[0], i)
  jointCrockoToId[jointInfo[1].decode('UTF-8')] = jointInfo[0]
  p.setJointMotorControl2(Crocko[0],i, p.VELOCITY_CONTROL,force=0)

desno_rame_1 = jointCrockoToId['desno_rame_1']
lijevo_rame_1 = jointCrockoToId['lijevo_rame_1']
desno_rame_2 = jointCrockoToId['desno_rame_2']
lijevo_rame_2 = jointCrockoToId['lijevo_rame_2']

desni_zglob_1 = jointCrockoToId['desni_zglob_1']
lijevi_zglob_1 = jointCrockoToId['lijevi_zglob_1']
desni_zglob_2 = jointCrockoToId['desni_zglob_2']
lijevi_zglob_2 = jointCrockoToId['lijevi_zglob_2']
```

```
Motori = np.array([desno_rame_1,
                   lijevo_rame_1,
                   desno_rame_2,
                   lijevo_rame_2,
                   desni_zglob_1,
                   lijevi_zglob_1,
                   desni_zglob_2,
                   lijevi_zglob_2])

#------------------------------------------------------------------------------

#   Varijable pomaka-------------------------------------------------------------
velocity = 0.2 #m/s
force = 5 #N

#------------------------------------------------------------------------------

#   Polozaj i orijentacija glavne kamere-----------------------------------------
Yaw = 270 #početne vrijednosti
Pitch = -60
Zoom = 5

p.resetDebugVisualizerCamera(cameraDistance=Zoom,
                             cameraYaw=Yaw,
                             cameraPitch=Pitch,
                             cameraTargetPosition= [0,B/2,0])

#------------------------------------------------------------------------------
```

```
#  Definiranje pomaka motora-------------------------------------------------
def povratak_pozicija():
   for r in range(B):
      p.setJointMotorControlArray(bodyIndex = Crocko[r],
                                  jointIndices = [desno_rame_1,
                                                  lijevo_rame_1,
                                                  desno_rame_2,
                                                  lijevo_rame_2,
                                                  desni_zglob_1,
                                                  lijevi_zglob_1,
                                                  desni_zglob_2,
                                                  lijevi_zglob_2],
                                  controlMode = p.POSITION_CONTROL,
                                  targetPositions = 0*np.ones(8),
                                  forces = force*np.ones(8))


def vektor_pomak(robot,vektor_motor,target,force):
   p.setJointMotorControlArray(bodyIndex = robot,
                               jointIndices = vektor_motor,
                               controlMode = p.POSITION_CONTROL,
                               targetPositions = np.radians(target))


def relax():
   for r in range(B):
      p.setJointMotorControlArray(bodyIndex=Crocko[r],
                                  jointIndices=[desno_rame_1,
                                                lijevo_rame_1,
                                                desno_rame_2,
                                                lijevo_rame_2,
```

```
                                desni_zglob_1,
                                lijevi_zglob_1,
                                desni_zglob_2,
                                lijevi_zglob_2],
                        controlMode=p.VELOCITY_CONTROL,
                        targetVelocities=np.zeros((8)))

#------------------------------------------------------------------------

#   Evolucijski algoritam-----------------------------------------------------
relax()
Tic = time.time()
#Vektori i matrice za petlju
Roditelji = np.zeros((8,A,B))
Potomci = np.zeros((8,A,B))
Fitnes = np.zeros((B,1))
Fitnes_potomaka = np.zeros((B,1))

XYZ = np.zeros((B,3))
Orijentacija = np.zeros((B,4))
Psi = np.zeros((B,1))
Razlika = np.arange(0,B,1)

if Nastavak == True:
    Generacija = Data_gen
    Fitnes = Data_fit
    Suma_fitnesa_plot = Data_sum
    Populacija = Data_pop
```

```
    Suma_fitnesa = np.sum(Fitnes)
    Elitni_plot = Data_elite

    #Oznaka generacije
    Slider = p.addUserDebugParameter("Generacija",0,Iteracija+Data_gen,Generacija)
    Oznaka_gen = p.addUserDebugText(text="Generacija {}".format(Generacija),
                                    textPosition=[0,B/2,0.7],
                                    textColorRGB=[0,0,0],
                                    textSize=2,
                                    lifeTime=0)

else:
    #Inicijalna populacija - random
    target = 88/2
    Populacija = np.random.randint(-Pozicija_max, Pozicija_max,size=(8, A, B))

    #Oznaka generacije
    Generacija = 0 #Početna generacija - nulta
    Slider = p.addUserDebugParameter("Generacija",0,Iteracija,Generacija)
    Oznaka_gen = p.addUserDebugText(text="Generacija {}".format(Generacija),
                                    textPosition=[0,B/2,0.7],
                                    textColorRGB=[0,0,0],
                                    textSize=2,
                                    lifeTime=0)

    for i in range(A):
        time.sleep(0.4)
        for j in range(B):
            vektor_pomak(Crocko[j],Motori,Populacija[:,i,j],force)
```

```
time.sleep(0.4)
povratak_pozicija()

for i in range(B):
    XYZ[i], Orijentacija[i] = p.getBasePositionAndOrientation(Crocko[i])
    Psi[i] = p.getEulerFromQuaternion(Orijentacija[i,:])[0]
    p.resetBasePositionAndOrientation(Crocko[i], [0,i,0.1], CrockoStartOrientation)

X = XYZ[:,0]
Y = XYZ[:,1]-Razlika

for i in range(B):
    if X[i] <= 0: #ako se robot kreće prema nazad
        X[i] = 0
    Fitnes[i] = 500*X[i]**2/(0.001+25*abs(Y[i])+470*abs(Psi[i]))

Suma_fitnesa = np.sum(Fitnes)
Suma_fitnesa.astype(np.ndarray)
Suma_fitnesa_plot = Suma_fitnesa

Elitni_plot = np.max(Fitnes)
Elitni_plot.astype(np.ndarray)

open('Evoluirani_hodovi.txt', 'w').close()#brisanje starih podataka iz datoteke
Argmax = np.argmax(Fitnes)
Elitni_gait = Populacija[:,:,Argmax]
f = open('Evoluirani_hodovi.txt', 'a')
f.write('#Generacija: {}'.format(Generacija)+'\n')
```

```
    np.savetxt(f,Elitni_gait,fmt='%-4d',delimiter=',')
    f.close()

Button_fit = p.addUserDebugParameter('Elitni fitnes {}'.format(Elitni_plot),1,0,1)



for c in range(Iteracija): #Petlja

    #Provjera veze sa simulacijom
    Povezanost = p.getConnectionInfo(physicsClient)
    if Povezanost.get('isConnected') != 1:
        p.disconnect(physicsClient)
        print('Prekinuta veza')
        break

    #Relativni fitnes
    Relativni_fitnes = Fitnes/Suma_fitnesa



    #Izbor roditelja - ruletno pravilo
    Random_roditelj = np.random.rand(B,1)

    for i in range(B):
        Sumator = Relativni_fitnes[0]
        for j in range(B):
            if (Sumator >= Random_roditelj[i]):
                Roditelji[:,:,i] = Populacija[:,:,j]
                break
            elif (j == B-1):
```

```
            Roditelji[:,:,i] = Populacija[:,:,B-1]
        else:
            Sumator = Sumator + Relativni_fitnes[j+1]



# Križanje - križaju se dvije susjedne 2D matrice u 3D matrici
Potomci = Roditelji

for i in range(0,B,2):#po matrici
    for j in range(8):#po retku (motoru)
        K = np.random.rand()
        if (K <= Pk): # ako je unutar Pk

            Sjecište = np.random.randint(low=1,high=A)
            Potomci[j,Sjecište:,i] = Roditelji[j,Sjecište:,i+1]
            Potomci[j,Sjecište:,i+1] = Roditelji[j,Sjecište:,i]



# Mutacija - svaki broj u 3D matrici ima 'Pm' vjerojatnost da mutira
for i in range(8):
    for j in range(A):
        for k in range(B):
            M = np.random.rand()
            if (M <= Pm):
                Potomci[i,j,k] = Potomci[i,j,k]+np.random.randint(low=-2,high=2)
Potomci = np.clip(Potomci,-Pozicija_max,Pozicija_max)
#ograniči se mutacija da ne varira van mogućih pomaka motora robota

# Evaluacija
```

```
for i in range(A):
    time.sleep(0.4)
    for j in range(B):
        vektor_pomak(Crocko[j],Motori,Potomci[:,i,j],force)

time.sleep(0.4)
povratak_pozicija()

for i in range(B):
    XYZ[i], Orijentacija[i] = p.getBasePositionAndOrientation(Crocko[i])
    Psi[i] = p.getEulerFromQuaternion(Orijentacija[i,:])[0]
    p.resetBasePositionAndOrientation(Crocko[i], [0,i,0.1], CrockoStartOrientation)

X = XYZ[:,0]
Y = XYZ[:,1]-Razlika
#Z = XYZ[:,2]
for i in range(B):
    if X[i] <= 0: #ako se robot kreće prema nazad
        X[i] = 0
    Fitnes_potomaka[i] = 500*X[i]**2/(0.001+25*abs(Y[i])+470*abs(Psi[i]))

Suma_fitnesa_potomaka = np.sum(Fitnes_potomaka)



for i in range(B): #evaluacija

    Fitnes_min = np.min(Fitnes) #najgori član roditelja
    Argmin = np.argmin(Fitnes)
```

```
    Fitnes_potomaka_max = np.max(Fitnes_potomaka) #najbolji član potomaka
    Argmax = np.argmax(Fitnes_potomaka)

    if (Fitnes_min < Fitnes_potomaka_max):

        Fitnes_potomaka[Argmax] = Fitnes_min #izmjena fitnesa
        Fitnes[Argmin] = Fitnes_potomaka_max

        Genom = Populacija[:,:,Argmin] #izmjena genoma
        Populacija[:,:,Argmin] = Potomci[:,:,Argmax]
        Potomci[:,:,Argmax] = Genom

    else: #ako nema boljeg člana u potomcima prekini evaluaciju
        break

Suma_fitnesa = np.sum(Fitnes)

Suma_fitnesa_plot = np.append(Suma_fitnesa_plot,Suma_fitnesa)
Elitni_clan = np.max(Fitnes)
Elitni_plot = np.append(Elitni_plot,Elitni_clan)



#Oznaka generacije
if Nastavak == True:
    Generacija += 1
    p.removeAllUserParameters()
    Slider = p.addUserDebugParameter("Generacija",0,Iteracija+Data_gen,Generacija)
    p.removeUserDebugItem(Oznaka_gen)
```

```
    Oznaka_gen = p.addUserDebugText(text="Generacija {}".format(Generacija),
                                    textPosition=[0,B/2,0.7],
                                    textColorRGB=[0,0,0],
                                    textSize=2,
                                    lifeTime=0)
    Button_fit = p.addUserDebugParameter('Elitni fitnes {}'.format(Elitni_clan),1,0,1)
else:

    Generacija += 1
    p.removeAllUserParameters()
    Slider = p.addUserDebugParameter("Generacija",0,Iteracija,Generacija)
    p.removeUserDebugItem(Oznaka_gen)

    Oznaka_gen = p.addUserDebugText(text="Generacija {}".format(Generacija),
                                    textPosition=[0,B/2,0.7],
                                    textColorRGB=[0,0,0],
                                    textSize=2,
                                    lifeTime=0)
    Button_fit = p.addUserDebugParameter('Elitni fitnes {}'.format(Elitni_clan),1,0,1)

if (Generacija % 50 == 0):
    Argmax = np.argmax(Fitnes)
    Elitni_gait = Populacija[:,:,Argmax]
    f = open('Evoluirani_hodovi.txt', 'a')
    f.write('#Generacija: {}'.format(Generacija)+'\n')
    np.savetxt(f,Elitni_gait,fmt='%-4d',delimiter=',')
    f.close()
```

```
Parametri_simulacije = p.getPhysicsEngineParameters(physicsClient)
p.disconnect()

Toc = time.time() #kraj mjerenja vremena
Vrijeme = Toc-Tic # vrijeme simulacije

Generacija_plot = np.linspace(0,Generacija,Generacija+1)
Generacija_plot.astype(float)

print('Vrijeme simulacije: ',Vrijeme,'[s]')
print('Elitni genom: ',Elitni_clan)



#Zapis konačnog elitnog genoma u txt datoteku
open('Elitni_gait.txt', 'w').close()#brisanje starih podataka iz datoteke
Argmax = np.argmax(Fitnes)
Elitni_gait = Populacija[:,:,Argmax]
f=open('Elitni_gait.txt','w')
z = repr(Generacija)
f.write('#Generacija:'+z+'\n')
z = repr(Argmax)
f.write('#Genom:'+z+'\n')
z = repr(float(Fitnes[Argmax]))
f.write('#Fitnes genoma:'+z+'\n')
np.savetxt(f, Elitni_gait, fmt='%1.5f',delimiter=',')
f.close()

#Spremanje podataka o populaciji ako želimo nastaviti EA
np.savez('Gen_Fit_Pop',
```

```
        Generacija = Generacija,
        Fitnes = Fitnes,
        Suma_fitnesa_plot=Suma_fitnesa_plot,
        Elitni_plot=Elitni_plot,
        Populacija = Populacija)

#Spremanje podataka o elitnom genomu
np.savez('Elitni_gait',
        Elitni_gait=Elitni_gait,
        Fitnes=np.column_stack((Generacija_plot,Suma_fitnesa_plot,Elitni_plot)))



#   Plotanje-----------------------------------------------------------------
figure, axes = plt.subplots(2,1)

axes[0].set_xlabel('Generacija')
axes[0].set_ylabel('Fitnes populacije')
axes[0].set_title('Fitnes populacije kroz generacije')
axes[0].grid()

axes[1].set_xlabel('Generacija')
axes[1].set_ylabel('Fitnes elitnog člana')
axes[1].set_title('Fitnes najboljeg člana kroz generacije')
axes[1].grid()

plt.tight_layout()

axes[0].plot(Generacija_plot, Suma_fitnesa_plot, color='black')
```

```
axes[1].plot(Generacija_plot, Elitni_plot, color='black')



plt.savefig('Graf.jpg',format='jpg',bbox_inches='tight')
plt.show()

#------------------------------------------------------------------------
```

III. Program testiranja hoda

```
#  Library import-------------------------------------------------------
import pybullet as p
import time
import pybullet_data
import numpy as np
import threading
import matplotlib.pyplot as plt
from sys import exit

#------------------------------------------------------------------------
#  Uzima se matrica iz .txt dobivena evolucijskim algoritmom-----------------

a = int(input('\nOdaberi:\n     1)Programiran hod\n     2)Evolucijski naučen hod\n'))
if a == 1:
   #Programiran hod

   Elitni_gait = np.array(([-30, 30, 30],
                           [-30, 30, 30],
                           [ 30,-30,-30],
```

```
                    [ 30,-30,-30],
                    [-30,-30,  0],
                    [  0,  0,  0],
                    [  0,  0,  0],
                    [-30,-30,  0]))
    Generacija_plot = 0
    Suma_fitnesa_plot = 0
    Elitni_plot = 0

elif (a==2):

    Data = np.load('93_6_Elitni_gait.npz')
    Elitni_gait = Data['Elitni_gait']
    Generacija_plot = Data['Fitnes'][:,0]
    Suma_fitnesa_plot = Data['Fitnes'][:,1]
    Elitni_plot = Data['Fitnes'][:,2]

    Data_evo = np.loadtxt('86_Evoluirani_hodovi.txt',delimiter=',',comments='#')

else:
    print('Krivi odabir! Pokušajte pokrenuti ponovo (1 ili 2).')
    exit()



A = np.size(Elitni_gait[0,:])
Korak = 30
Vel_limit = 10
Evolucija = False
```

```
# Postavke simulacije----------------------------------------------------------
if p.isConnected()==1: #ako je povezan već da resetira spoj
   p.disconnect()

physicsClient = p.connect(p.GUI)
Povezanost = p.getConnectionInfo(physicsClient)
p.setAdditionalSearchPath(pybullet_data.getDataPath()) #optionally
p.setGravity(0,0,-9.81)

p.setRealTimeSimulation(1)

p.configureDebugVisualizer(p.COV_ENABLE_GUI,1)# Da se ne vidi ono sa strane na simulaciji
#--------------------------------------------------------------------------------

# Učitavanje robota i objekata i njihovo spajanje----------------------------
planeId = p.loadURDF('plane.urdf')
CrockoStartPos = [0,0,0.2]
CrockoStartOrientation = p.getQuaternionFromEuler([0,0,0])
Crocko = p.loadURDF('Crocko.urdf',
                    CrockoStartPos,
                    CrockoStartOrientation)
#pokazuje smjer kretanja
p.addUserDebugLine([0,0,0],[100,0,0],[1,0,0],lineWidth=5,lifeTime=0)

for i in range(100):
   p.addUserDebugText(text="{}m".format(i),
                      textPosition=[i,0,0.1],
                      textColorRGB=[0,0,0],
```

```
                textSize=1.5,
                lifeTime=0)

#--------------------------------------------------------------------------

#   Informacije o zglobovima i link-ovima-------------------------------------
nJoints = p.getNumJoints(Crocko)
jointSpidyToId = {}
for i in range(nJoints):
  jointInfo = p.getJointInfo(Crocko, i)
  jointSpidyToId[jointInfo[1].decode('UTF-8')] = jointInfo[0]
  p.setJointMotorControl2(Crocko,i, p.VELOCITY_CONTROL,force=0)

desno_rame_1 = jointSpidyToId['desno_rame_1']
lijevo_rame_1 = jointSpidyToId['lijevo_rame_1']
desno_rame_2 = jointSpidyToId['desno_rame_2']
lijevo_rame_2 = jointSpidyToId['lijevo_rame_2']

desni_zglob_1 = jointSpidyToId['desni_zglob_1']
lijevi_zglob_1 = jointSpidyToId['lijevi_zglob_1']
desni_zglob_2 = jointSpidyToId['desni_zglob_2']
lijevi_zglob_2 = jointSpidyToId['lijevi_zglob_2']

Motori = np.array([desno_rame_1,
                   lijevo_rame_1,
                   desno_rame_2,
                   lijevo_rame_2,
                   desni_zglob_1,
                   lijevi_zglob_1,
```

```
                desni_zglob_2,
                lijevi_zglob_2])

#--------------------------------------------------------------------------

#   Varijable pomaka-----------------------------------------------------------
velocity = 10 #rad/s
#--------------------------------------------------------------------------

#   Trenje na prstima
p.changeDynamics(Crocko,desni_zglob_1,lateralFriction = 1,spinningFriction = 0)
p.changeDynamics(Crocko,lijevi_zglob_1,lateralFriction = 1,spinningFriction = 0)
p.changeDynamics(Crocko,desni_zglob_2,lateralFriction = 1,spinningFriction = 0)
p.changeDynamics(Crocko,lijevi_zglob_2,lateralFriction = 1,spinningFriction = 0)
p.changeDynamics(planeId,-1,lateralFriction = 1,spinningFriction = 0)
#--------------------------------------------------------------------------

#   Definiranje pomaka motora--------------------------------------------------
def povratak_pozicija():
   p.setJointMotorControlArray(bodyIndex = Crocko,
                               jointIndices = [desno_rame_1,
                                               lijevo_rame_1,
                                               desno_rame_2,
                                               lijevo_rame_2,
                                               desni_zglob_1,
                                               lijevi_zglob_1,
                                               desni_zglob_2,
                                               lijevi_zglob_2],
                               controlMode = p.POSITION_CONTROL,
```

```
                    targetPositions = 0*np.ones(8),
                    forces = 5*np.ones(8))

def vektor_pomak(vektor_motor,target):
   p.setJointMotorControlArray(bodyIndex = Crocko,
                    jointIndices = vektor_motor,
                    controlMode = p.POSITION_CONTROL,
                    targetPositions = np.radians(target),
                    #velocityGains = 0.001*np.ones(8),
                    #targetVelocities = -1*np.ones(8),
                    forces = 5*np.ones(8))

def relax():
   p.setJointMotorControlArray(bodyIndex = Crocko,   # relakscija mišića
                   jointIndices = [desno_rame_1,
                           lijevo_rame_1,
                           desno_rame_2,
                           lijevo_rame_2,
                           desni_zglob_1,
                           lijevi_zglob_1,
                           desni_zglob_2,
                           lijevi_zglob_2],
                   controlMode = p.VELOCITY_CONTROL,
                   forces = np.zeros(8))

#------------------------------------------------------------------------

#   Polozaj i orijentacija glavne kamere---------------------------------------
```

```
Yaw = -90 #početne vrijednosti
Pitch = -50
Zoom = 2

basePos, baseOrn = p.getBasePositionAndOrientation(Crocko)
p.resetDebugVisualizerCamera(cameraDistance=Zoom,
                             cameraYaw=Yaw,
                             cameraPitch=Pitch,
                             cameraTargetPosition= [1,0,0])

p.resetBasePositionAndOrientation(Crocko, CrockoStartPos, CrockoStartOrientation)

#------------------------------------------------------------------------------

t = [0]
te = [0]
X_plot = np.array([0])
Y_plot = np.array([0])
Psi_plot = np.array([0])
E_plot = np.zeros((8,1))
E_plot = np.hstack((E_plot,Elitni_gait))

#   Thread za čitanje podataka zglobova----------------------------------------
t2 = 0
Pos_0 = 0
Pos_1 = 0
Pos_2 = 0
Pos_3 = 0
Pos_4 = 0
```

```
Pos_5 = 0
Pos_6 = 0
Pos_7 = 0
Vel_0 = 0
Vel_1 = 0
Vel_2 = 0
Vel_3 = 0
Vel_4 = 0
Vel_5 = 0
Vel_6 = 0
Vel_7 = 0
M_0 = 0
M_1 = 0
M_2 = 0
M_3 = 0
M_4 = 0
M_5 = 0
M_6 = 0
M_7 = 0

def jointState():
    global t2, Pos_0, Pos_1,Pos_2,Pos_2,Pos_3,Pos_4,Pos_5,Pos_6,Pos_7,jointStates
    global Vel_0,Vel_1,Vel_2,Vel_3,Vel_4,Vel_5,Vel_6,Vel_7
    global M_0,M_1,M_2,M_3,M_4,M_5,M_6,M_7,Crocko
    dt = 10
    print('Thread running')

    for n in range(dt):
```

```
#Očitanje stanja zglobova
jointStates = p.getJointStates(Crocko,Motori,physicsClient)

#Zapis očitanih pozicija
Pos_0 = np.append(Pos_0, np.rad2deg((jointStates[0])[0]))
Pos_1 = np.append(Pos_1, np.rad2deg((jointStates[1])[0]))
Pos_2 = np.append(Pos_2, np.rad2deg((jointStates[2])[0]))
Pos_3 = np.append(Pos_3, np.rad2deg((jointStates[3])[0]))
Pos_4 = np.append(Pos_4, np.rad2deg((jointStates[4])[0]))
Pos_5 = np.append(Pos_5, np.rad2deg((jointStates[5])[0]))
Pos_6 = np.append(Pos_6, np.rad2deg((jointStates[6])[0]))
Pos_7 = np.append(Pos_7, np.rad2deg((jointStates[7])[0]))

#Zapis očitanih brzina
Vel_0 = np.append(Vel_0, ((jointStates[0])[1]))
Vel_1 = np.append(Vel_1, ((jointStates[1])[1]))
Vel_2 = np.append(Vel_2, ((jointStates[2])[1]))
Vel_3 = np.append(Vel_3, ((jointStates[3])[1]))
Vel_4 = np.append(Vel_4, ((jointStates[4])[1]))
Vel_5 = np.append(Vel_5, ((jointStates[5])[1]))
Vel_6 = np.append(Vel_6, ((jointStates[6])[1]))
Vel_7 = np.append(Vel_7, ((jointStates[7])[1]))

#Zapis očitanih momenata u motorima
M_0 = np.append(M_0,(jointStates[0])[3])
M_1 = np.append(M_1,(jointStates[1])[3])
M_2 = np.append(M_2,(jointStates[2])[3])
M_3 = np.append(M_3,(jointStates[3])[3])
M_4 = np.append(M_4,(jointStates[4])[3])
```

```
        M_5 = np.append(M_5,(jointStates[5])[3])
        M_6 = np.append(M_6,(jointStates[6])[3])
        M_7 = np.append(M_7,(jointStates[7])[3])

        T2 = time.time()-T0
        t2 = np.append(t2,T2)

        time.sleep(0.3/dt)

d = 0
def kamera():
    global d
    while(d < Korak):
        basePos, baseOrn = p.getBasePositionAndOrientation(Crocko)
        p.resetDebugVisualizerCamera(cameraDistance=1,
                                     cameraYaw=0,
                                     cameraPitch=-60,
                                     cameraTargetPosition= [basePos[0],0,0])
        time.sleep(0.01)

#--------------------------------------------------------------------------

#   Hodanje

Nesto = np.zeros((8,A))

Nesto[0:4,0:A] = Elitni_gait[0:4,0:A]*(-1)
Nesto[4:,0:A] = Elitni_gait[4:,0:A]
```

```
Motori_kom = np.array([lijevo_rame_1,
                       desno_rame_1,
                       lijevo_rame_2,
                       desno_rame_2,
                       lijevi_zglob_1 ,
                       desni_zglob_1,
                       lijevi_zglob_2 ,
                       desni_zglob_2])


povratak_pozicija()


stalak = p.createConstraint(parentBodyUniqueId = planeId, #početno postolje
                            parentLinkIndex = -1,
                            childBodyUniqueId = Crocko,
                            childLinkIndex = -1,
                            jointType = p.JOINT_FIXED,
                            jointAxis = [0, 0, 0],
                            parentFramePosition = [0,0,0.2],
                            childFramePosition = [0,0,0])
time.sleep(3)
p.removeConstraint(stalak) #    uklanjanje postolja


Generacija = 0
if (a==2) and (Evolucija == True):
    b,a = np.shape(Data_evo)
    for i in range(0,b,8):


        Oznaka = p.addUserDebugParameter("Generacija",0,50*b/8,Generacija)
        for j in range(A):
```

```
            vektor_pomak(Motori,Data_evo[i:i+8,j])
            time.sleep(0.3)

        Generacija += 50
        p.removeAllUserParameters()
    p.disconnect()

else:
    if (a == 2):
        Button_A = p.addUserDebugParameter('Duljina genoma {}'.format(A),1,0,1)
        Button_gen = p.addUserDebugParameter('Generacija {}'.format(Generacija_plot[-
1]),1,0,1)
        Button_fit = p.addUserDebugParameter('Elitni fitnes {}'.format(Elitni_plot[-1]),1,0,1)

    T0 = time.time() #početak mjerenja vremena

    #(thread_kamere := threading.Thread(target=kamera,daemon=True)).start()

    for z in range(Korak):  #Broj koraka
        for i in range(A):

            if (z == 0):
                (thread := threading.Thread(target=jointState, daemon=True)).start()
                print('Thread started\n')

            if (z%2==0):
                vektor_pomak(Motori,Elitni_gait[:,i])

            else:
```

```
            vektor_pomak(Motori_kom,Nesto[:,i])

        XYZ, Orijentacija = p.getBasePositionAndOrientation(Crocko)

        time.sleep(0.4)
        T1 = time.time() - T0
        t = np.append(t,T1)
        X_plot = np.append(X_plot,XYZ[0])
        Y_plot = np.append(Y_plot,XYZ[1])
        Psi_plot = np.append(Psi_plot,np.rad2deg(Orijentacija[0]))
        if (z == 0):
            thread.join() #pričekaj da se izvrti subproces(thread)
            te = np.append(te,T1)
    d += 1

p.disconnect()
for thread in threading.enumerate():
    print(thread.name)

#   Plotanje----------------------------------------------------------------

#Plot za fitnes elitnog genoma
figure_fit, axes_fit = plt.subplots(2,1)

axes_fit[0].set_xlabel('Generacija',fontsize=16)
axes_fit[0].set_ylabel('Fitnes populacije',fontsize=16)
axes_fit[0].set_title('Fitnes populacije kroz generacije',fontsize=20)
axes_fit[0].grid()
```

```
axes_fit[1].set_xlabel('Generacija',fontsize=16)
axes_fit[1].set_ylabel('Fitnes elitnog člana',fontsize=16)
axes_fit[1].set_title('Fitnes najboljeg člana kroz generacije',fontsize=20)
axes_fit[1].grid()

plt.tight_layout()

axes_fit[0].plot(Generacija_plot, Suma_fitnesa_plot, color='black')
axes_fit[1].plot(Generacija_plot, Elitni_plot, color='black')

#Plot za poziciju i orijentaciju robota
figure_pos, ax_pos = plt.subplots(3,1)

ax_pos[0].set_xlabel('Vrijeme [s]',fontsize=16)
ax_pos[0].set_ylabel('Pomak po x-osi [m]',fontsize=16)
ax_pos[0].set_title('Pomak po x-osi',fontsize=20)
ax_pos[0].grid()
ax_pos[1].set_xlabel('Vrijeme [s]',fontsize=16)
ax_pos[1].set_ylabel('Pomak po y-osi [m]',fontsize=16)
ax_pos[1].set_title('Pomak po y-osi',fontsize=20)
ax_pos[1].grid()
ax_pos[2].set_xlabel('Vrijeme [s]',fontsize=16)
ax_pos[2].set_ylabel('Pomak oko z-osi [°]',fontsize=16)
ax_pos[2].set_title('Orijentacija oko z-osi',fontsize=20)
ax_pos[2].grid()

ax_pos[0].plot(t, X_plot, color='red')
ax_pos[1].plot(t, Y_plot, color='green')
ax_pos[2].plot(t,Psi_plot,color='blue',marker='*')
```

```
figure_pos.tight_layout()


figure_pos.savefig('Graf_XY.jpg',format='jpg',bbox_inches='tight')


#Plot za motore ramena
fig_shoulder_pos, ax_shoulder_pos = plt.subplots(4,layout='constrained')
#fig_shoulder_pos.suptitle('Pozicije motora ramena robota', fontsize=16)
i = 0
for z in [0,1,2,3]:
    ax_shoulder_pos[i].set_xlabel('Vrijeme [s]',fontsize=16)
    ax_shoulder_pos[i].set_ylabel('Pozicija zgloba [°]',fontsize=16)
    ax_shoulder_pos[i].set_ylim(bottom=-50,top=50)
    ax_shoulder_pos[i].set_title(f'Pozicija zgloba {z} kroz vrijeme',fontsize=20)
    ax_shoulder_pos[i].grid()
    i += 1

for z in range(4):
    ax_shoulder_pos[z].step(te,E_plot[z,:],where='pre',color='red')
ax_shoulder_pos[0].plot(t2,Pos_0)
ax_shoulder_pos[1].plot(t2,Pos_1)
ax_shoulder_pos[2].plot(t2,Pos_2)
ax_shoulder_pos[3].plot(t2,Pos_3)




#Plot za motore nogu
fig_leg_pos, ax_leg_pos = plt.subplots(4,layout='constrained')
#fig_leg_pos.suptitle('Pozicije motora koljena robota', fontsize=16)
i = 0
```

```
for z in [4,5,6,7]:
    ax_leg_pos[i].set_xlabel('Vrijeme [s]',fontsize=16)
    ax_leg_pos[i].set_ylabel('Pozicija zgloba [°]',fontsize=16)
    ax_leg_pos[i].set_ylim(bottom=-50,top=50)
    ax_leg_pos[i].set_title(f'Pozicija zgloba {z} kroz vrijeme',fontsize=20)
    ax_leg_pos[i].grid()
    i += 1

for z in range(4):
    ax_leg_pos[z].step(te,E_plot[z+4,:],where='pre',color='red')
ax_leg_pos[0].plot(t2,Pos_4)
ax_leg_pos[1].plot(t2,Pos_5)
ax_leg_pos[2].plot(t2,Pos_6)
ax_leg_pos[3].plot(t2,Pos_7)



#Plot brzina motora ramena
fig_shoulder_vel, ax_shoulder_vel = plt.subplots(4,layout='constrained')

#fig_shoulder_vel.suptitle('Brzine motora ramena robota', fontsize=16)
i = 0
for z in [0,1,2,3]:
    ax_shoulder_vel[i].set_xlabel('Vrijeme [s]',fontsize=16)
    ax_shoulder_vel[i].set_ylabel('Brzina [rad/s]',fontsize=16)
    ax_shoulder_vel[i].set_title(f'Brzina zgloba {z} kroz vrijeme',fontsize=20)
    ax_shoulder_vel[i].grid()
    i += 1

ax_shoulder_vel[0].plot(t2,Vel_0)
```

```
ax_shoulder_vel[1].plot(t2,Vel_1)
ax_shoulder_vel[2].plot(t2,Vel_2)
ax_shoulder_vel[3].plot(t2,Vel_3)



#Plot brzina motora nogu
fig_leg_vel, ax_leg_vel = plt.subplots(4,layout='constrained')
#fig_leg_vel.suptitle('Brzine motora koljena robota', fontsize=16)
i = 0
for z in [4,5,6,7]:
    ax_leg_vel[i].set_xlabel('Vrijeme [s]',fontsize=16)
    ax_leg_vel[i].set_ylabel('Brzina [rad/s]',fontsize=16)
    ax_leg_vel[i].set_title(f'Brzina zgloba {z} kroz vrijeme',fontsize=20)
    ax_leg_vel[i].grid()
    i += 1

ax_leg_vel[0].plot(t2,Vel_4)
ax_leg_vel[1].plot(t2,Vel_5)
ax_leg_vel[2].plot(t2,Vel_6)
ax_leg_vel[3].plot(t2,Vel_7)



#Plot momenata motora ramena
fig_shoulder_M, ax_shoulder_M = plt.subplots(4,layout='constrained')

#fig_shoulder_M.suptitle('Momenti motora ramena robota', fontsize=16)
i = 0
for z in [0,1,2,3]:
```

```
    ax_shoulder_M[i].set_xlabel('Vrijeme [s]',fontsize=16)
    ax_shoulder_M[i].set_ylabel('Moment [Nm]',fontsize=16)
    ax_shoulder_M[i].set_title(f'Moment zgloba {z} kroz vrijeme',fontsize=20)
    ax_shoulder_M[i].grid()
    i += 1

ax_shoulder_M[0].plot(t2,M_0)
ax_shoulder_M[1].plot(t2,M_1)
ax_shoulder_M[2].plot(t2,M_2)
ax_shoulder_M[3].plot(t2,M_3)



#Plot momenata motora koljena
fig_leg_M, ax_leg_M = plt.subplots(4,layout='constrained')

#fig_leg_M.suptitle('Momenti motora koljena robota', fontsize=16)
i = 0
for z in [4,5,6,7]:
    ax_leg_M[i].set_xlabel('Vrijeme [s]',fontsize=16)
    ax_leg_M[i].set_ylabel('Moment [Nm]',fontsize=16)
    ax_leg_M[i].set_title(f'Moment zgloba {z} kroz vrijeme',fontsize=20)
    ax_leg_M[i].grid()
    i += 1

ax_leg_M[0].plot(t2,M_4)
ax_leg_M[1].plot(t2,M_5)
ax_leg_M[2].plot(t2,M_6)
ax_leg_M[3].plot(t2,M_7)
```

```
    #Prikazati grafove
    fig_leg_M.show()
    fig_shoulder_M.show()
    fig_leg_vel.show()
    fig_shoulder_vel.show()
    fig_leg_pos.show()
    fig_shoulder_pos.show()
    figure_pos.show()
    figure_fit.show()

if (a==1):
    Fitnes = 500*X_plot[-1]**2/(0.001+25*abs(Y_plot[-1])+470*abs(Psi_plot[-1]))

if (a==2):
    Data.fid.close()
#------------------------------------------------------------------------
```