



УНИВЕРЗИТЕТ У НОВОМ  
САДУ

ФАКУЛТЕТ ТЕХНИЧКИХ  
НАУКА У НОВОМ САДУ




Немања Мајсторовић

**Имплементација  
Reinforcement learning  
алгоритма за  
балансирање hexapod  
робота**

Дипломски рад  
- Основне академске студије -

Нови Сад, 2024.



	УНИВЕРЗИТЕТ У НОВОМ САДУ <b>ФАКУЛТЕТ ТЕХНИЧКИХ НАУКА</b> 21000 НОВИ САД, Трг Доситеја Обрадовића 6	Датум:
	<b>ЗАДАТАК ЗА ИЗРАДУ ДИПЛОМСКОГ (BACHELOR) РАДА</b>	Лист: 1/1

(Податке уноси предметни наставник - ментор)

Врста студија:	Основне академске студије
Студијски програм:	Софтверско инжењерство и информационе технологије
Руководилац студијског програма:	проф. др Мирослав Зарић

Студент:	Немања Мајсторовић	Број индекса:	SV 10/2020
Област:	Инжењерство софтвера за Internet/Web of Things		
Ментор:	проф. др Милан Видаковић		

НА ОСНОВУ ПОДНЕТЕ ПРИЈАВЕ, ПРИЛОЖЕНЕ ДОКУМЕНТАЦИЈЕ И ОДРЕДБИ СТАТУТА ФАКУЛТЕТА ИЗДАЈЕ СЕ ЗАДАТАК ЗА ДИПЛОМСКИ РАД, СА СЛЕДЕЋИМ ЕЛЕМЕНТИМА:

- проблем – тема рада;
- начин решавања проблема и начин практичне провере резултата рада, ако је таква провера неопходна;
- литература

#### НАСЛОВ ДИПЛОМСКОГ (BACHELOR) РАДА:

**Имплементација Reinforcement learning алгоритма за балансирање хеxарод робота**

#### ТЕКСТ ЗАДАТКА:

Задатак представља развој система за учење хеxарод робота да из положаја спуштених ногу устане и стоји балансирајући се у том положају уз помоћ Reinforcement learning алгоритма. Велики део система је имплементирам помоћу ROS(Robot Operating System) алата док су скрипте коришћене за учење имплементирани у Python програмском језику. Апликација омогућава учитавање упрошћеног модела и његово тренирање у симулацији затим научене акције се преносе на реалног робота.

Руководилац студијског програма:	Ментор рада:

Примерак за: ☐ - Студента; ☐ - Ментора

# САДРЖАЈ

1. УВОД .....	7
2. ПРЕГЛЕД СЛИЧНИХ СИСТЕМА.....	9
3. КОРИШЋЕНЕ СОФТВЕРСКЕ ТЕХНОЛОГИЈЕ .....	11
3.1 ROS .....	11
3.1.1 URDF .....	11
3.1.2 Gazebo симулатор.....	12
3.1.3 Управљач уређаја .....	12
3.1.4 Rospy .....	12
3.1.5 OpenAI gym.....	13
3.2 Tensorflow и Pytorch.....	13
4. СПЕЦИФИКАЦИЈА .....	15
4.1 Спецификација захтева .....	15
4.1.1 Функционални захтеви .....	15
4.1.2 Нефункционални захтеви .....	15
4.2 Спецификација система .....	16
4.2.1 Модел података .....	16
4.2.2 Архитектура система .....	19
5. ИМПЛЕМЕНТАЦИЈА.....	21
5.1 ROS .....	21
5.1.1 URDF .....	22
5.1.2 Gazebo.....	26
5.1.3 Управљач контролера .....	27
5.1.4 Rospy .....	29
5.1.3 OpenAI gym.....	34
5.2 Алгоритми.....	35

5.2.1 Q-learning .....	36
5.2.3 Deep Q learning .....	39
5.2.3 PPO(Proximal Policy Optimization).....	41
5.2 Хардвер .....	46
6. ДЕМОНСТРАЦИЈА.....	49
7. ЗАКЉУЧАК.....	51
8. ЛИТЕРАТУРА.....	53
9. БИОГРАФИЈА.....	55
КЉУЧНА ДОКУМЕНТАЦИЈСКА ИНФОРМАЦИЈА.....	57
KEY WORDS DOCUMENTATION .....	59



## 1. УВОД

Задатак представља развој система за учење hexapod робота да из положаја спуштених ногу устане и што стабилније стоји уз помоћ Reinforcement learning алгоритма.

Инспирација за задатак је Crawler робот са курса за увод у вештачку интелигенцију универзитета Berkley[1]. Робот је уз помоћ Reinforcement learning-а учио да хода у симулацији те је идеја била да се то физички имплементира и отежа тиме што би робот уместо једне имао 6 ногу.

Велики део система је имплементирам помоћу ROS (Robot Operating System) алата док су скрипте коришћене за учење имплементираних у Python програмском језику. Кроз рад коришћени су различити алгоритми који су се примењивали у симулацији а потом и на реалну имплементацију робота..

Јединственост решења се огледа у секвенци примена различитих алгоритама у комбинацији са роботским системом и тиме истиче њихове предности и мане.

Прво поглавље је уводно, док се у другом налази опис изабраних технологија који су коришћени при изради. Треће поглавље представља детаљну спецификацију система. Четврто поглавље представља опис конкретне имплементације система, док је у петом поглављу изнет закључак.





## 2. ПРЕГЛЕД СЛИЧНИХ СИСТЕМА

Критеријуми за претрагу сличних решења били су у контексту:

- дизајна физичке имплементација робота
- симулације решења у окружењима сличних реалном и
- алгоритама вештачке интелигенције за учење сличних проблема.

Како имплементација многих робота захтева комплексне и скупе делове идеја је била да се нађе дизајн који се може обавити уз помоћ 3D штампања и коришћењем малих серво мотора. Нађено решење садржи потребне 3D моделе као и дизајн чијом је модификацијом и настало ово решење [1]. Предност ових решења су цена физичких имплементација док је мана квалитет компоненти (ломљивост компоненти, снага мотора).

Идеју за симулационо окружење и архитектуру система је пронађена у курсу за учење робота помоћу вештачке интелигенције са увидом на лакоћу преноса учења на физичку имплементацију [2]. Предност је модуларнија и организованија архитектура система са што приближнијом конфигурацијом симулације која одговара реалним условима.

Алгоритми вештачке интелигенције су се мењали у зависности од опажања резултата али идеје су преузете из курсева и научних радова [2] [3] [4]. Предности и мане ће бити приказане у објашњењу.



### 3. КОРИШЋЕНЕ СОФТВЕРСКЕ ТЕХНОЛОГИЈЕ

У овом поглављу ће бити описане технологије које су коришћене за имплементацију система. У поглављу 3.1 ROS описан је Robot Operating System (ROS), у поглављу 3.2 Tensorflow и Pytorch описана су Tensorflow и Pytorch Python библиотеке.

#### 3.1 ROS

ROS представља open-source колекцију алата који се често користе у изради роботских апликација и тиме убрзавају процес израде, као што су апстракција hardware-a, контрола уређаја на ниском нивоу, комуникација процеса и управљање пакетима. Покренути скупови процеса заснованих на ROS-у представљени су у архитектури графа где се обрада одвија у чворовима који могу да примају, шаљу и мултиплексирају податке за сензоре, контролу, стање, планирање, актуаторе и друге поруке путем publish/subscribe метода.

Најважнији подсистеми биће објашњени у следећи поглављима. У поглављу 3.1.1 URDF биће описан URDF (Unified Robot Description Format), у поглављу симулатор биће описан Gazebo симулатор, у поглављу 3.1.3 Управљач уређаја биће описан управљач контролера, у поглављу 3.1.4 Rospy биће описана rospy библиотека и у поглављу 3.1.5 OpenAI gym биће описан OpenAi gym библиотека која служи за развој и упоређивање софтвера за Reinforcement learning (RL).

##### 3.1.1 URDF

URDF садржи разне Extensible Markup Language (XML) анотације за моделе робота, сензоре, актуаторе, ограничења итд. Спецификација предвиђа да се робот састоји од карика (link) које су повезане зглобовима (joint). Главне компоненте спецификације су:

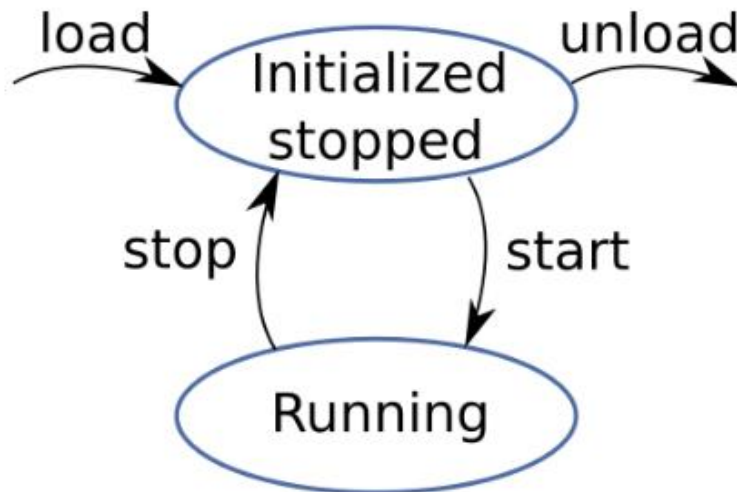
- кинематичка и динамичка својства робота,
- визуелна репрезентација робота и
- колизиона репрезентација робота

### 3.1.2 Gazebo симулатор

Gazebo је 2D/3D open source симулатор за роботiku. Има интегрисани engine за симулацију физике помоћу диференцијалних једначина, OpenGL рендеровање и подршку за коришћење сензора и актуатора. Има одличну интеграцију са ROS-ом због које се врло лако врши трансфер коришћења софтвера на реалну имплементацију робота.

### 3.1.3 Управљач уређаја

Управљач контролера (Controller manager) контролише петљу у реалном времену која контролише механизме робота који су репрезентовани помоћу хардверског интерфејса. Менаџер пружа инфраструктуру којом може да учитава, искључи, отпочне и заустави контролере као што је приказано на Слика 3.1.



Слика 3.1 рад контролера [5]

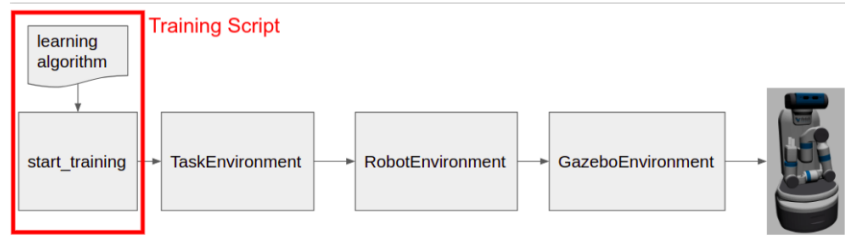
### 3.1.4 Rospy

Rospy је Python клијентска библиотека за ROS. Rospy клијент поседује програмски интерфејс за коришћење ROS-а уз помоћ Python-а. Дизајн rospy библиотеке фаворизује брзину имплементације пре него брзину

извршавања. Садржи разне апстракције за ROS чворове, њихову међусобну комуникацију као и коришћење самих ROS алата.

### 3.1.5 OpenAI gym

Gym је Python библиотека отвореног кода за развој и упоређивање алгоритама за учење (специфично RL алгоритама) уз помоћ пружања стандардног програмског интерфејса за комуникацију између алгоритама учења и окружења, као и стандардног скупа окружења усклађених са тим програмским интерфејсом. Интеграција са ROS-ом је омогућена кроз `openai_ros` библиотеку која дефинише ток комуникације оваквих система за учење као што је приказано на Слика 3.2.



Слика 3.2 Комуникација у `openai_ros` системима [6]

У зависности од овога се формира архитектура. Модуларност ове архитектуре се огледа у томе да су раздвојени стање робота, само окружење у коме се налази (у смислу конкретног дизајна робота као и у контексту симулације), задатак који треба да изврши и алгоритам који га учи задатку. На самом крају научене акције се преносе на физичку имплементацију робота.

Више о OpenAI gym на [7].

## 3.2 Tensorflow и Pytorch

Tensorflow је end-to-end платформа отвореног кода за машинско учење. Има свеобухватан, флексибилан екосистем алата, библиотека и ресурса заједнице који омогућава да се лако граде апликације које примењују машинско учење. Има подршку за Python. У оквиру tensorflow пројекта

налази се и Tensorboard. Tensorboard омогућава визуелизацију и алате потребне за експериментисање машинским учењем:

- праћење и визуелизација метрика као што су губитак и тачност,
- визуелизација графикана модела (операције и слојеви),
- прегледање хистограма тежина, пристрасности или других тензора како се мењају током времена,
- пројектовање уградње у простор ниже димензије,
- приказ слика, текста и аудио података и
- профилисање Tensorflow програма

Више о Tensorflow и Tensorboard на [7].

Pytorch је Python пакет који пружа две функције високог нивоа:

- тензорско израчунавање (као Numpy) са снажним GPU убрзањем и
- дубоке неуронске мреже изграђене на аутоградијентном систему заснованом на траци

Више о Pytorch на [8].

## 4. СПЕЦИФИКАЦИЈА

### 4.1 Спецификација захтева

Систем треба да прочита модел робота, постави га у симулацију и да га алгоритмом вештачке интелигенције научи да устане из задатог спуштеног положаја у положај стајања и одржавања у том стању.

#### 4.1.1 Функционални захтеви

Функционални захтеви се огледају у томе да робот треба да устане и одржава се у том положају у симулацији и да се то уз помоћ пропраћених акција пренесе и изврши на реалној имплементацији робота.

#### 4.1.2 Нефункционални захтеви

Систем јесте у реалном времену али не превише захтевном, одзив може бити до 0.5 секунди.

Алгоритми за учење се требају извршити у догледно време што је у овом случају не преко 10 сати.

Како се покретање свих компоненти користи помоћу терминала графички интерфејс није преко потребан. Међутим графички приказ кретања робота у симулацији би требао да се може укључити и искључити.

Апликација треба да је у складу са стандардима за креирање ROS апликација као и да је `openai_gos` ток комуникације кроз архитектуру задовољен.

Портабилност система је пожељно да буде у складу са Raspbian или Ubuntu оперативним системима како би се могла омогућити евентуална будућа интеграција са Raspberry Pi-ом.

Отпорност на грешке је од изузетне важности у контроли над реалним моторима због могућег физичког оштећења, уколико дође до оваквог сценарија апликација треба престати са радом.

## 4.2 Спецификација система

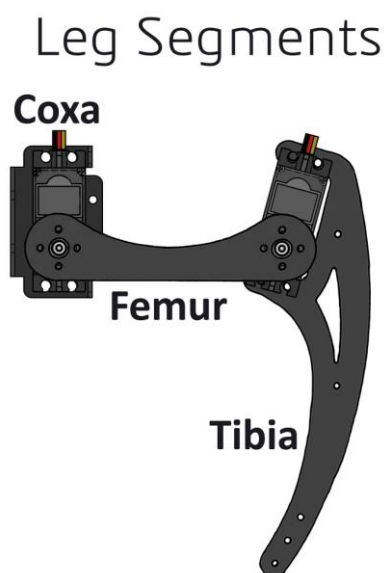
У поглављу 4.2.1 Модел података следи објашњење модела података који обухвата реалну имплементацију робота (4.2.1.1 Модел робота) и класни модел података у софтверу (4.2.1.2 Класни модел података). Поглавље 4.2.2 Архитектура система приказује архитектуру целог система.

### 4.2.1 Модел података

У овом поглављу објаснићемо како изгледа модел робота (поглавље 4.2.1.1 Модел робота) и класни модел података у пакету за учење (поглавље 4.2.1.2 Класни модел података).

#### 4.2.1.1 Модел робота

Модел робота је hexapod тј. само тело робота подсећа на тело појединих инсеката од којих је и добило назив (hexapoda). Тело се састоји од 6 ногу и средишњег дела тела (thorax). Свака од ногу има 3 зглоба који заједно за одређеним делом тела формирају део ноге, редом од thorax-а ка врховима, соха, femur и tibia као што се и види на Слика 4.1 Сегменти ноге .

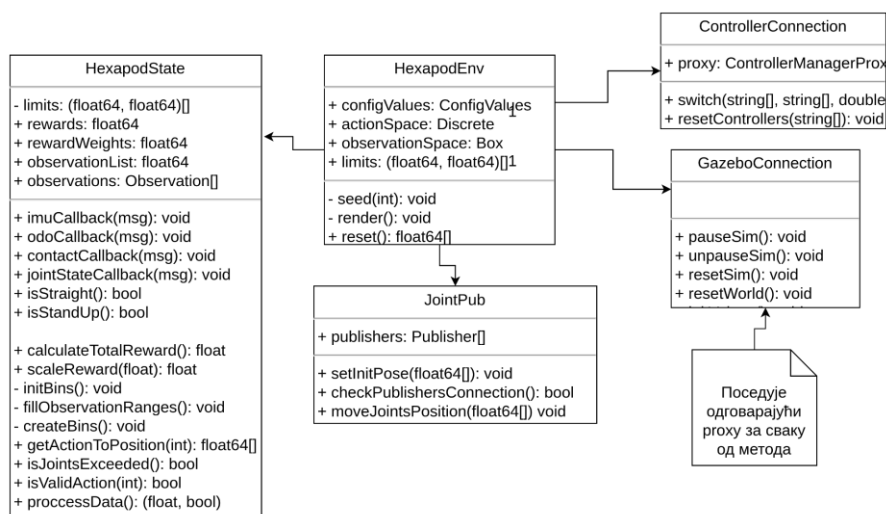


Слика 4.1 Сегменти ноге [6]



#### 4.2.1.2 Класни модел података

Класни модел има смисла правити једино у пакету за учење како нигде другде класа и нема. Како је дијаграм велики поделићемо га на секције. Како постоји велики број атрибута неки од њих су семантички повезани у класе на дијаграму иако заправо нису у коду.



Слика 4.2 Све класе осим класе за алгоритам за учење

GazeboConnection класа служи за комуникацију са Gazebo симулацијом, паузирање симулације, ресетовање симулације, ресетовања окружења.

ControllerConnection служи за комуникацију са управљачем контролера.

JointPub служи да пошаље поруку којом командује зглобовима.

HexapodState апстрахује тренутно стање робота.

HexapodEnv обухвата тренутно стање робота са контекстом окружења.

Класни модел (Слика 4.2 Све класе осим класе за алгоритам за учење) има приложене и напомене (Слика 4.3 генерална напомена (лево) и напомена за HexapodState (десно) и Слика 4.4 класа алгоритма за учење)

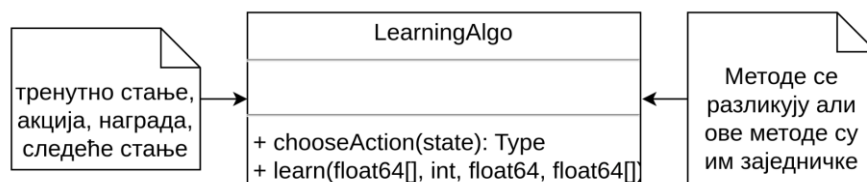
Set и Get методе  
се подразумевају  
и све везе су 1-1

Метода calculateTotalReward  
обрачунава све награде и  
сумира их а то су:

- награда за не губљење
- награда за позиције зглобова
- награда за мањак утрошеног  
труда
- награда за избегавање  
контакта thorax-а са подлогом
- награда за оријентацију
- награда за јачину контакта са  
земљом
- награда за удаљеност од  
жељене тачке
- награда за синхронизацију  
покрета
- награда за додир tibia делова  
са подлогом

Свака од ових метода има  
одговарајућу тежину коју  
множи са обрачунатом  
наградом

Слика 4.3 генерална напомена (лево) и напомена за HexapodState  
(десно)

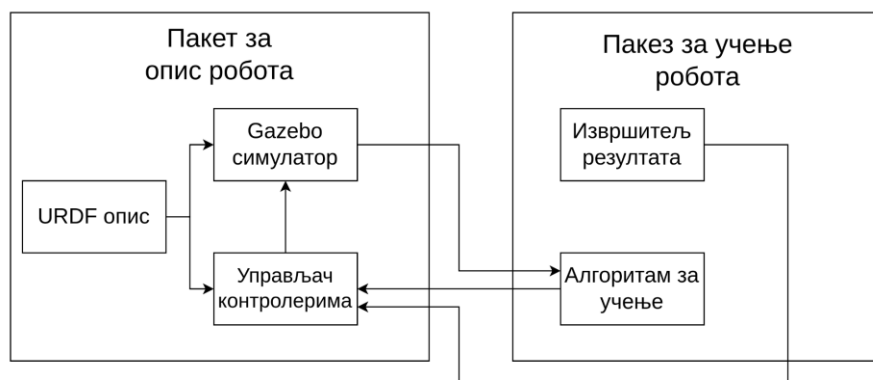


Слика 4.4 класа алгорита за учење

Класе инхерентно поседују ове методе иако се можда не зову тачно тако у коду (Слика 4.2 Све класе осим класе за алгоритам за учење).

## 4.2.2 Архитектура система

У овом поглављу ће бити представљена архитектура на апстрахованом ROS нивоу. ROS архитектура представља да постоји један радни простор где се налазе сви ROS пројекти, тај простор се састоји од пакета. Пакети се састоје од разних описа робота, скрипти за интегрисање компоненти, скрипти за покретање компоненти, програмских скрипти, конфигурација итд. Апстрахована је јер није потребно залазити и појединачне фајлове приказивати већ су неке компоненте семантички интегрисане.



Слика 4.5 Архитектура на високом ROS нивоу

На Слика 4.5 увиђамо да URDF опис бива потхрањен симулатору како би знао како се компоненте визуелно и колизионо манифестују као и која физичка својства (моменти силе, трење, гравитација...) робот поседује. Опис такође доставља податке о трансмисији односно податке који служе да опишу зглобове самог робота док управљач дефинише како њима треба управљати што и шаље симулатору који то извршава.

Алгоритам за учење задаје команде за управљање зглобовима симулатору посредством управљача. Алгоритам за учење добија повратне информације од Gazebo окружења какво је стање робота у симулацији (стања сензора, позиције, брзине, обртни моменат зглобова...) и на основу тога учи како треба да реагује у зависности од стања.

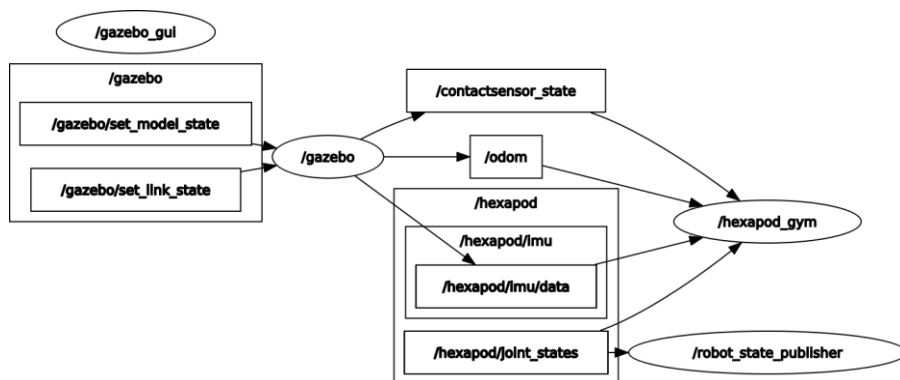
Извршитељ резултата извршава добијене резултате ради њихове визуелизације.

## 5. ИМПЛЕМЕНТАЦИЈА

У овом поглављу је представљена имплементација софтвера као и хардвера за проблем учења балансирању hexapod робота. Поглавље 5.1 ROS биће ослоњен на део који се односи на ROS док ће 5.2 Алгоритми бити ослоњен на алгоритме RL-а и њихову имплементацију.

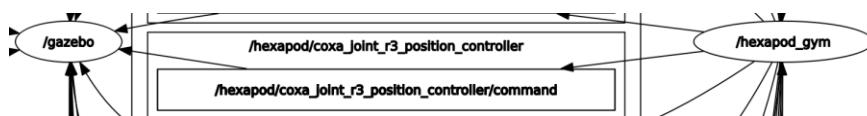
### 5.1 ROS

Коришћен је ROS Noetic 1.16. ROS систем се састоји од чворова који међусобно комуницирају помоћу publish/subscribe метода. Њихова организација је приказана на Слика 5.1.



Слика 5.1 организација чворова изгенерисана rqt\_graph алатом

Слика 5.1 приказује како Gazebo симулација путем сензора шаље податке чвору који је задужен за учење, сензори за контакт tibia делова са подлогом су изостављени како би слика била непрегледна, аналогно раде као и сензори за контакт thorax дела.



Слика 5.2 Контролери у rqt\_graph

Слика 5.1 изоставља компоненте и канале везане за контролере јер их има превише (36 укупно) међутим издвојени део на Слика 5.2

аналогно се односи на све остале зглобове. Софтвер за учење командује над зглобовима посредством контролера који преносе команде за управљање Gazebo симулацији.

### 5.1.1 URDF

Што се тиче URDF-а, коришћен је URDF са макро опцијама тј. Хасро (XML macros). Како симулација захтева доста ресурса у пракси се моделују прости објекти уместо компликованих 3D модела те се робот састоји само од квадрара. Како постоји доста репетитивног кода везаног за моделовање ногу (свака има соха, femur и tibia) са том разликом да су делови лево и десно односе као да су окренути у огледалу. У следећем делу следи приказ најважнијих делова URDF кода.

```
<xacro:property name="norm" value="100" />
```

Листинг 5.1 нормативна вредност

Листинг 5.1 приказује број са којим се деле све дужинске вредности обзиром да су изражене у центиметрима док формат захтева метре.

Следећа 2 листинга приказују макрое коришћене за дефинисање својстава везаних за моменат инерције као што су маса и инерциона матрица.

$$\mathbf{I} = \begin{bmatrix} I_{xx} & -I_{xy} & -I_{xz} \\ -I_{xy} & I_{yy} & -I_{yz} \\ -I_{xz} & -I_{yz} & I_{zz} \end{bmatrix}$$

Једначина 5.1 инерциона матрица

```
<!-- Define box macros for inertia -->
<xacro:macro name="box_inertia" params="m w h d
origin_xyz rpy_x rpy_y rpy_z">
  <inertial>
```

```

        <origin xyz="{origin_xyz}" rpy="{rpy_x}
${rpy_y} ${rpy_z}"/>
        <mass value="{m}" />
        <inertia ixx="{(m/12) * (h*h +
d*d)}" ixy="0.0" ixz="0.0" iyy="{(m/12) * (w*w +
d*d)}"
                                iyz="0.0" izz="{(m/12) * (w*w
+ h*h)}" />
    </inertial>
</xacro:macro>

```

Листинг 5.2 Макро за дефинисање инерције квадрa са центром у геометријској средини

```

<!-- Parallel axis theorem -->
<xacro:macro name="box_inertia_parallel_w"
params="m w h d origin_xyz rpy_x rpy_y rpy_z">
    <inertial>
        <origin xyz="{origin_xyz}"
rpy="{rpy_x} ${rpy_y} ${rpy_z}" />
        <mass value="{m}" />
        <inertia ixx="{(m/12) * (h*h + d*d
+ w*w/4)}" ixy="0.0" ixz="0.0" iyy="{(m/12) * (w*w
+ d*d + w*w/4)}"
                                iyz="0.0" izz="{(m/12) * (w*w
+ h*h + w*w/4)}" />
    </inertial>
</xacro:macro>

```

Листинг 5.3 Макро за дефинисање квадрa са центром у ивици квадрa

Како је симулација приказивала чудно понашање ногу робота дошло се до закључка да су инерционе матрице лоше подешене. Листинг 5.3 приказује модификовану инерциону матрицу квадрa која има померену референтну тачку осе ротације на ивици квадрa што је потребно за зглобове који се налазе приближно на ивицама карика. За ово је потребно применити Штајнерову теорему тј. теорему паралелних оса што приказује једначина Једначина 5.2.

$$[d] = \begin{bmatrix} 0 & -d_z & d_y \\ d_z & 0 & -d_x \\ -d_y & d_x & 0 \end{bmatrix}$$

$$[d]^2 = \begin{bmatrix} -d_y^2 - d_z^2 & d_x d_y & d_x d_z \\ d_x d_y & -d_x^2 - d_z^2 & d_y d_z \\ d_x d_z & d_y d_z & -d_x^2 - d_y^2 \end{bmatrix}$$

$$[I_S] = [I_R] - M[d]^2,$$

Једначина 5.2 [6]

Ово је и урађено у коду (Листинг 5.3), оса померена за  $w/2$  те се унутар заграда додаје се тај квадрирани термин тј.  $w*w/4$ .

```
<transmission name="coxa_tran_${side}${num}">

    <type>transmission_interface/SimpleTransmission<
/type>

        <joint
name="coxa_joint_${side}${num}">

            <hardwareInterface>hardware_interface/EffortJointInterface</hardwareInterface>
            </joint>
            <actuator
name="coxa_motor_${side}${num}">

                <hardwareInterface>hardware_interface/EffortJointInterface</hardwareInterface>

                <mechanicalReduction>1</mechanicalReduction>
            </actuator>
        </transmission>
```

Листинг 5.4 трансмисиони атрибути

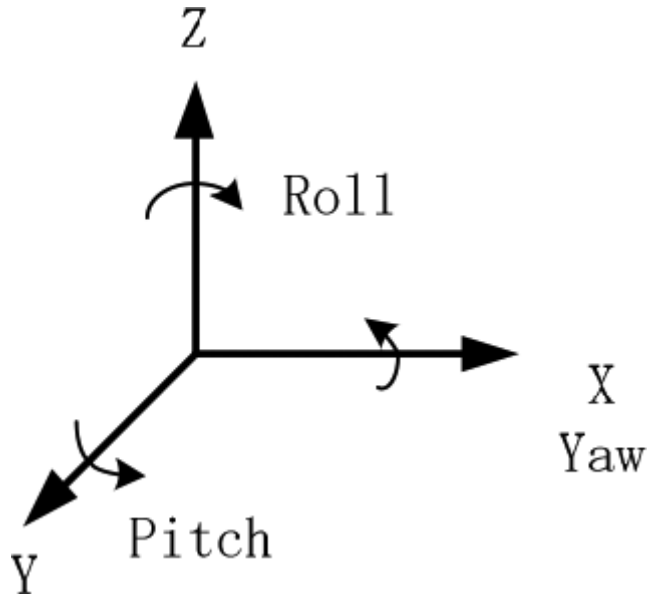


Трансмисиони атрибути приказани у Листинг 5.4 приказују како управљач контролера има увид у то којим зглобом и како управља путем назива зглоба. Користимо `EffortJointInterface` који нам омогућава да задамо `effort` односно момент силе и брзину самих мотора који имају та номинална својства.

```
<!-- Contact sensor -->
  <sensor name="contactsensor_sensor"
type="contact">
...
<!-- ODOM sensor -->
  <plugin name="p3d_base_controller"
filename="libgazebo_ros_p3d.so">
...
<!-- IMU sensor -->
  <plugin name="gazebo_ros_imu_controller"
filename="libgazebo_ros_imu.so">
```

Листинг 5.5 сензори

Листинг 5.5 приказује сензоре који нам дају податке који се сврставају у део опсервације које су потребне за учење робота. Контакт сензори приказују да ли се десио контакт између `thorax`-а и подлоге (такође и силу којом се контакт десио али нам то није потребно) и `tibia` делова и подлоге. Одометријски сензор нам показује позицију робота. Inertial Measurement Unit (IMU) сензор нам приказује оријентацију робота помоћу Ојлерових углова које означавамо са RPY - Rotation Pitch Yaw. Свака од 3 компоненте RPY дефинише ротацију око једне осе као што је приказано на Слика 5.3.



Слика 5.3 RPY

### 5.1.2 Gazebo

```
<!-- Compile xacro and load urdf -->
<param name="robot_description" command="$(find
xacro)/xacro $(find
urdf_demo)/urdf/crab_model.xacro" />

<!-- SInclude Gazebo -->
<include file="$(find
gazebo_ros)/launch/empty_world.launch">
  <arg name="world_name"
value="worlds/empty_world.world"/>
  <arg name="paused" value="false"/>
  <arg name="use_sim_time" value="true"/>
  <arg name="gui" value="true"/>
  <arg name="recording" value="false"/>
  <arg name="debug" value="false"/>
```

```

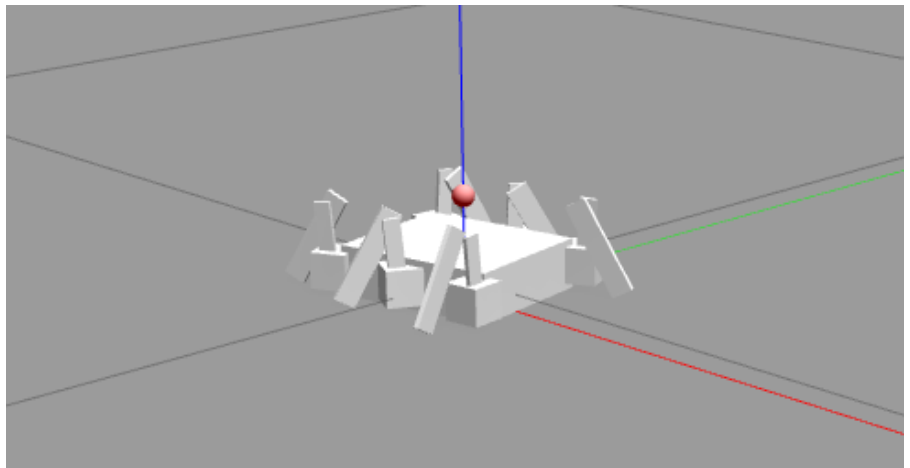
</include>

<!-- Spawn a robot into Gazebo -->
<node name="spawn_urdf" pkg="gazebo_ros"
type="spawn_model" args="-z 0.07 -param
robot_description -urdf -model hexapod" />

```

Листинг 5.6 део launch фајла задужен за Gazebo

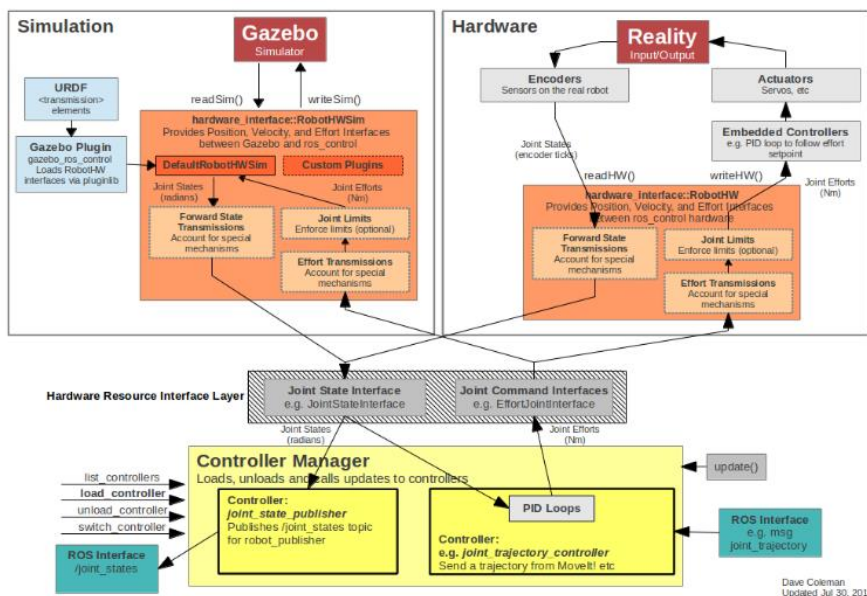
Листинг 5.6 представља учитавање URDF-а као параметра, затим покретање Gazebo симулације и учитавање робота у Gazebo окружење. Аналогно овоме се учитава и маркер који нам олакшава визуелизацију тога да ли је циљ задовољен. Визуелни приказ овога приказан је на Слика 5.4 изглед робота и маркера у Gazebo симулатору.



Слика 5.4 изглед робота и маркера у Gazebo симулатору

### 5.1.3 Управљач контролера

Управљач контролера је део `ros_control` пакета. Сваком од мотора додељен је контролер који њиме управља. Контролери су од изузетне користи јер се смањује потреба за променама при промени са симулације на реалног робота као што је приказано на Слика 5.5.



Слика 5.5 Управљач уређаја са реалним роботом и Gazebo симулацијом [7]

```

<launch>
<!-- Get controller config -->
<rosparam file="$(find
urdf_demo)/config/controllers.yaml"
command="load"/>
  <!-- load the controllers -->
  <node name="controller_spawner"
pkg="controller_manager" ns="hexapod"
type="spawner" respawn="false"
output="screen" args="--timeout 120
joint_state_controller
coxa_joint_l1_position_controller
coxa_joint_l2_position_controller
...
</launch>

```

Листинг 5.7 launch код за покретање управљача контролера

Листинг 5.7 приказује како се учитава конфигурација контролера учитава и похрањује у управљачу који даље почиње PID (Proportional-integral-derivative) петљу. PID петља тежи ка томе да управља актуатором тако што рачуна грешку између жељене и тренутне тачке и смањује је у жељеном маниру у зависности од подешених параметара (P, I и D). P дефинише удаљеност од жељене вредности, I дефинише конзистентност функције и D дефинише брзину приближавања жељеној тачки.

Како је PI популарнији од PID [7], експериментисањем је донет закључак да су задовољавајуће вредности  $P=1.0$ ,  $I=0.01$  и  $D=0.0$ . Подешена фреквенција за ажурирање је 50Hz. За више о PID погледати [8].

### 5.1.4 Rospy

У свим RL алгоритмима уводе се апстракције које дефинишу сам проблем а то су:

- агент - ентитет који уз помоћ полисе максимизује очекивану награду кретањем кроз окружење,
- опсервација - скуп својстава који представља зависну променљиву, од чега зависи исход, како моделујемо јединственост стања у коме се агент налази,
- акција - механизам којим агент прелази из једног у друго стање,
- награда - нумеричка скаларна вредност која дефинише вредност преласка из једног у друго стање посредством акције, најчешће што је већа награда то се бољи исход предвиђа што је и овде случај,
- тежине награда - коефицијенти који се множе са компонентама награде,
- критеријум завршетка - постоје стања која не могу прећи у друга стања, терминална стања, најчешће означавају позитиван или негативан завршни исход и
- полиса - пробабилистичка функција која мапира стања на акције

#### 5.1.4.1 Агент

Агент у систему јесте hexapod робот.

### 5.1.4.2 Опсервација

Опсервација у овом систему је једнодимензионални вектор који се састоји из компоненти у Листинг 5.8.

```
- distance_from_desired_point
- base_z
- base_roll
- base_pitch
# - base_yaw
#- base_angular_vel_x
#- base_angular_vel_y
#- base_angular_vel_z
#- base_linear_acceleration_x
#- base_linear_acceleration_y
#- base_linear_acceleration_z

# - contact_force
- joint_states
# - joint_effort
- touching_ground
- tibia_touching
```

Листинг 5.8 Компоненте опсервације

Компоненте опсервације односно стања:

- дистанца од жељене тачке - колико је агент удаљен од тачке која је потребно да се досегне обзиром да је потребно да робот устане, жељена тачка у координатном систему је  $(x, y, z) = (0.0, 0.0, 0.8)$ ,
- висина *thorax*-а робота - висина на којој се робот налази дефинише у којој фази устајања је робот,
- *roll* компонента Ојлеровог система - ротација око *x* осе дефинише да ли је робот накренут налево или надесно,
- *pitch* компонента Ојлеровог система - ротација око *y* осе дефинише да ли је робот накренут напред или назад,
- стања зглобова - углови свих зглобова - најважнија компонента која дефинише који је угао на ком се налази

сваки од актуатора, једнодимензионални вектор од 18 компоненти и

- додир подлоге - да ли thorax додирује подлогу, компонента која помаже агенту да превазиђе почетни проблем устајања приморавајући га да устане што пре и
- додир подлоге и tibia дела - да ли се ноге налазе на подлози или не, пожељно је да се све ноге налазе на земљи, ова се компонента није увек користила

Напомене у вези са закоментарисаним компонентама, уaw компонента није релевантна обзиром да није битно како је робот окренут, компоненте брзине и убрзања нису потребне у контексту балансирања (биле би релевантније да учимо робота да хода), сила падања није потребна јер да робот хода тако би се знало да ли је пао овде постоји компонента која дефинише да ли додирује подлогу стога би био вишак података чувати компоненте силе, коришћени моменат силе би било корисно чувати међутим значајно увећава димензионалност опсервације (за 18).

#### 5.1.4.3 Акције

Акције у почетку представљају дискретан простор од 36 елемената. Енкодиране су тако да парне вредности повећавају одговарајући угао зглоба док непарни смањују за вредност дефинисану у конфигурацији који је конкретно 0.2 rad што је и приказано у листингу.

```

if action == 0: #Increment coxa_l1
    rospy.logdebug("Action Decided:Increment
coxa_l1_joint>>>")
    self.current_joint_pose[0] +=
self._joint_increment_value
    elif action == 1: #Decrement coxa_l1
    rospy.logdebug("Action Decided:Decrement
coxa_l1_joint>>>")
    self.current_joint_pose[0] -=
self._joint_increment_value

```

Листинг 5.9 утицај акција на стање

Касније акције су у континуалном простору, тј. акција се енкодира као вектор од 18 елемената чија свака компонента репрезентује зглоб и његову угаону позицију и налази се у опсегу  $[-1.5, 1.5]$ , овиме се постиже да је позиција једнака акцији уколико се та акција изврши за разлику од дискретног где се повећава/смањује.

#### 5.1.4.3 Награда и тежине награда

```

weight_r1: 3.0 # Weight for joint positions ( joints in
the zero is perfect )
weight_r2: 0.0 # Weight for joint efforts ( no efforts is
perfect )
weight_r3: 0.0 # Weight for contact force similar to
desired ( weight of hexapod )
weight_r4: 1000.0 # Weight for orientation ( vertical is
perfect )
weight_r5: 10000.0 # Weight for distance from desired
point ( on the point is perfect )
weight_r6: 20.0 # Weight for not touching floor
weight_r7: 0.0 # Weight for not synchronization
weight_r8: 10.0 # Weight for not tibia touching floor

```

Листинг 5.10 тежине награда

Листинг 5.10 приказује које се тежине користе за које врсте награда у зависности од њихове важности. Награда се формира кумулативном сумом свих компоненти награда помноженим са



њиховом одговарајућом тежином и у зависности са опсегом њихових вредности. Компоненте награде су следеће:

- награда за позиције зглобова - када су зглобови позиционирани на угао 0, то је положај робота у стајаћем, жељеном, положају међутим како је то финално стање робот се пре тога може понашати како год може стога му не придајемо превелики значај,
- награда за моменат силе - што је мањи напор то је већа награда јер се смањује напор на актуаторима и тиме спречава њихово могуће физичко оштећење, ово је такође од врло малог значаја јер нам је примарни циљ да устане и одржава се, ово потпада у нефункционални захтев,
- награда за силу контакта - ирелевантност овога је објашњена у 5.1.4.2 Опсервација те му је додељена тежина 0,
- награда за оријентацију - такође објашњено у 5.1.4.2 робот не сме пасти, не сме се прекренути то је супротно задатом циљу,
- награда за дистанцу од жељене тачке - такође напоменуто у 5.1.4.2 Опсервација ово је такође и компонента циља те је изузетно важно,
- награда за додир подлоге - такође појашњено у 5.1.4.2 Опсервација, агент не сме додиривати подлогу јер се то противи услову циља, уколико додирује подлогу све награде осим награде за опстанак се потиру,
- награда за синхронизацију - награђујемо агента синхронизовање покрета и избегавамо хаотично кретање зглобова у истој категорији, при чему категорију дефинишу зглобови исте врсте са исте стране међутим агент се некад ради балансирања мора кретати несинхронизовано те је ово касније занемарено,
- награда за додир подлоге и tibia дела – пожељно је да све ноге буду у контакту са подлогом како агент не би нарушио баланс позиције хаотичним кретањима и
- награда за опстанак - нису сва лоша терминална стања иста, што дуже робот опстане то је боље и то их разликује

#### 5.1.4.4 Критеријум завршетка

Уколико је агент на удаљености мањој од задатог параметра удаљености, уколико је pitch компонента Ојлеровог система мања од задатог параметра накренутости сматра се да је агент достигао жељено стање, уколико агент то учини задати број пута сматрамо успешно завршио задатак и даје му се максимална награда и завршавамо епизоду.

Уколико је било који зглоб досегао угао већи од  $1.5\text{rad}$  или мањи од  $-1.5\text{rad}$  сматрамо да је неуспешно извршио задатак и дајемо му минималну награду и прекидамо ту епизоду.

### 5.1.3 OpenAI gym

Класа `HexapodState` представља апстракцију стања агента односно једну опсервацију агента као и операције над компонентама опсервације.

Класа `HexapodEnv` представља апстракцију стања у контексту окружења. Класа наслеђује класу `OpenAI gym-a gym.Env` те стога мора да дефинише атрибуте:

- `action_space` - дефинише простор акција, у овом случају дискретан простор од 36 елемената или континуалан простор од 18 елемената и
- `observation_space` - дефинише простор опсервација чије вредности нису изван опсега  $[-50, 50]$  и који је вектор дужине 23 или 29 и састоји се од `float` вредности што је и приказано у Листинг 5.11.

```
self.action_space = spaces.Discrete(36)
# or
self.action_space = spaces.Box(low=-1.5, high=1.5,
                                shape=(18,), dtype=np.float64)

self.observation_space = spaces.Box(low=-50,
                                     high=50, shape=(len(state),), dtype=np.float64)
```

Листинг 5.11 креирање простора акција и опсервација

као и да имплементира неколико метода и то

- `reset` - ставља окружење у почетно стање и враћа то стање, важно је истакнути како услед неретких случајева заглављивања зглобова само ресетовање симулације није довољно јер робот остаје у том положају и не престаје да губи чим се појави, те стога пратимо задњих неколико тренутака када се `reset` десио и уколико се то деси

систем уради hard reset који поново учитава робота и контролере што се и види на

- 
- Листинг 5.12. превентивно се hard reset ради са вероватноћом 0.1. Не ради се у свакој епизоди јер му је потребно значајно више времена да се изврши,

```
# Resets the state of the environment and returns
an initial observation.
def reset(self):
    now = time.time()
    self.timestamps.append(now)
    self.reset_counter += 1
    if self.reset_counter > 10:
        past_time = self.timestamps[0]
        diff = now - past_time
        rospy.loginfo("DIFF>>>" + str(diff))
        if diff < 4 or random.random() < 0.1:
            rospy.loginfo("HARD RESETTING")
            return self.hard_reset()
        return self.soft_reset()
```

Листинг 5.12 reset функција

- step - прослеђује се акција која се потом и изврши и поставља агента у ново стање, повратне вредности су ново стање, награда за извршену акцију и да ли је ново стање терминално,
- get\_state - добављање тренутног стања,
- \_seed - иницијализација насумичног генератора и
- action\_masks - функција која генерише вектор чија позиција одговара једној акцији и мапира је на bool вредност у зависности од тога да ли је могуће извршити ту акцију (ова функција је неопходна само за једну имплементацију)

## 5.2 Алгоритми

Кроз рад примењени су различити алгоритми у тежњи за што бољим резултатима те ће бити хронолошки приказани у следећим поглављима.

### 5.2.1 Q-learning

Инспирација за сам рад пронађена је у претходно поменутом курсу универзитета Berkley [1] први покушај је ишао у истом смеру у смислу RL-а, коришћењем Q-learning алгоритма. Коришћењем Белманове једначине, Марковљевих ланаца и Temporal Difference (TD) агент формира полису из искуства стања кроз која пролази у току учења. Епизоду дефинишемо као комплетну секвенцу стања, акција и награда чије је последње стање терминално. Q вредност дефинишемо као процену очекиване вредности коју ће агент имати уколико из тренутног стања одговарајућом акцијом пређе у ново стање. Q табелу дефинишемо као функцију која мапира пар стање-акција на Q вредности. Ажурирање Q вредности дефинишемо као:

$$\begin{aligned} \text{sample} &= r + \gamma \max_{a'} Q(s', a') \\ Q(s, a) &\leftarrow (1 - \alpha) \cdot Q(s, a) + \alpha \cdot \text{sample} \\ Q(s, a) &\leftarrow Q(s, a) + \alpha \left[ r + \gamma \max_{a'} Q(s', a') - Q(s, a) \right] \end{aligned}$$

Једначина 5.3 ажурирање Q вредности

Једначина 5.3 нам показује како се ажурира Q вредност. sample представља вредност којој  $Q(s, a)$  треба да тежи. Q вредност, уколико се из стања  $s$  пређе у стање  $s'$ , је једнака тренутној Q вредности сабраном са коефицијентом  $\alpha$ , који дефинише колико се алгоритам узда у нове информације у односу на старе, помноженим са наградом коју је агент добио преласком из стања  $s$  акцијом  $a$  сабраном са коефицијентом  $\gamma$ , који дефинише колико нам је битна тренутна награда у односу на будућу награду, помножену са максималном могућом Q вредношћу које стање  $s'$  може узети својим акцијама, од тога се затим одузима  $Q(s, a)$ , више о овоме се налази на [1].

Дилема истраживање-експлоатација је фундаментални концепт у доношењу одлука. Приказана као балансирање између две супротстављене стратегије. Експлоатација укључује избор најбоље опције на основу тренутног знања о систему (које може бити непотпуно или погрешно), док истраживање укључује испробавање нових опција које могу довести до бољих исхода у будућности на рачун могућности експлоатације. Имплементација је приказана у

## Листинг 5.13.

Постојало је неколико проблема са овим решењем:

- обзиром да се агент налази у континуалном простору неопходна дискретизација компонената опсервација како би се уопште користила, робот ће се изузетно тешко наћи у истом стању које је прошао уколико не дискретизујемо опсервације као што је приказано на
- Листинг 5.13 и
- Листинг 5.14,

```
self._bins = numpy.zeros((number_of_observations,
parts_we_discretize))
for counter in range(number_of_observations):
    obs_name = self._list_of_observations[counter]
    min_value = self._obs_range_dict[obs_name][0]
    max_value = self._obs_range_dict[obs_name][1]
    self._bins[counter] = numpy.linspace(min_value,
max_value, parts_we_discretize)
```

Листинг 5.13 креирање опсега за дискретизацију

```
state_discrete =
numpy.zeros(len(self._list_of_observations),
dtype=numpy.int32)
for i in range(len(self._list_of_observations)):
    state_discrete[i] =
int(numpy.digitize(observation[i], self._bins[i],
right=True))
```

Листинг 5.14 додела дискретне вредности

- опсервација и награда за додир са подлогом успешно уче робота да се одвоји од подлоге међутим ово најчешће доводи до пада робота чиме он често заврши са минималном наградом. Због овога се агент подстицао на експлорацију кроз  $\epsilon$  параметар и смањивањем  $\epsilon$  decay параметра, као и додавањем шума у одлуке одабира акције [9] међутим безуспешно, агент увиђа након неког времена да је боље да се мало издигне и тиме зарађује награде без даљег прогреса, проблем са падањем се погоршао, увиђањем [9] доноси се закључак да је проблем категоризован као *hard-exploring* проблем, тј. да агент јако често губи и као такав тешко долази до открића позитивног исхода и
- Q табеле заузимају доста меморије у зависности од дискретизације

```

def learnQ(self, state, action, reward, value):
    oldv = self.q.get((state, action), None)
    if oldv is None:
        self.q[(state, action)] = reward
    else:
        self.q[(state, action)] = oldv +
self.alpha * (value - oldv)

def chooseAction(self, state, return_q=False):
    q = [self.getQ(state, a) for a in
self.actions]
    maxQ = max(q)

    if random.random() < self.epsilon:
        minQ = min(q); mag = max(abs(minQ),
abs(maxQ))
        # add random values to all the actions,
recalculate maxQ
        q = [q[i] + random.random() * mag - .5
* mag for i in range(len(self.actions))]
        maxQ = max(q)

    count = q.count(maxQ)
    # In case there're several state-action max
values
    # we select a random one among them
    if count > 1:
        best = [i for i in
range(len(self.actions)) if q[i] == maxQ]
        i = random.choice(best)
    else:
        i = q.index(maxQ)

    action = self.actions[i]
    if return_q:
        return action, q
    return action

def learn(self, statel, action1, reward, state2):

```

```

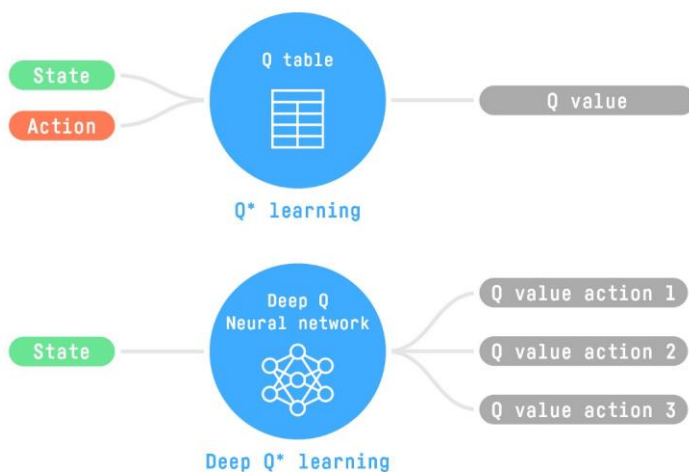
        maxqnew = max([self.getQ(state2, a) for a
in self.actions])
        self.learnQ(state1, action1, reward, reward
+ self.gamma*maxqnew)

```

Листинг 5.15 имплементација Q learn алгоритма

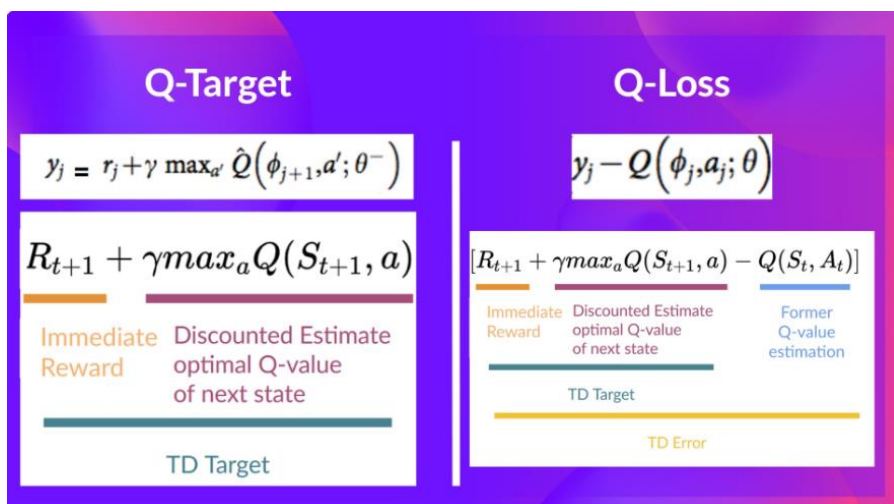
### 5.2.3 Deep Q learning

Deep Q learning је алгоритам сродан Q learning алгоритму са неким побољшањима. Уместо табеле која представља функцију уместо ње постоји дубока неуронска мрежа која апроксимира табелу као што је приказано на Слика 5.6.



Слика 5.6 разлика између Q learning и Deep Q learning [10]

Формирају се 2 неуронске мреже главна (main) и циљана (target), ово се ради јер главна мрежа се мења у сваком кораку те се континуално прилагођава и циљана вредност постаје нестабилна, уколико би се иста мрежа користила за генерисање циљаних Q вредности дошло би до осцилација или дивергенције. Аналогно Q learning-у као што је ту sample овде је target као што се и види на Слика 5.7.



Слика 5.7 Q target и дефиниција функције грешке [4]

Кораци су следећи:

- узорковање и
  1. одабира се акција, насумична са вероватноћом  $\epsilon$  или она која води ка највећој Q вредности и
  2. изврши се акција и добије се ново стање и награда потом се стање, акција, награда и ново стање се чувају у такозвани experience бафер
- тренирање
  1. проласком кроз узорак узет из бафера естимирамо Q вредност помоћу награде из бафера и target Q вредности и функцију грешке формирамо као MSE (Mean Squared Error) тј. квадрираним разликом естимоване и Q вредности главне мреже,
  2. урадити градијентни спуст и
  3. сваких C корака копирати главну у target мрежу

За више о Deep Q learning погледати [4].

Поређење у односу на Q learning:

- простор акција и даље мора да је дискретан јер резултат мреже су Q вредности за сваку акцију те опет морамо користити дискретизацију,



- проблем меморије је решен обзиром да неуронска мрежа има фиксан број параметара који не расте у зависности од кретања агента,
- hard exploring проблем није нестао агент и даље често пада и
- као и код Q learning алгоритма агент научи да се издигне и тиме избегне казну контакта са подлогом али ван тога нема напретка

Одговарајући код имплементације може се видети на репозиторијуму [5].

### 5.2.3 PPO (Proximal Policy Optimization)

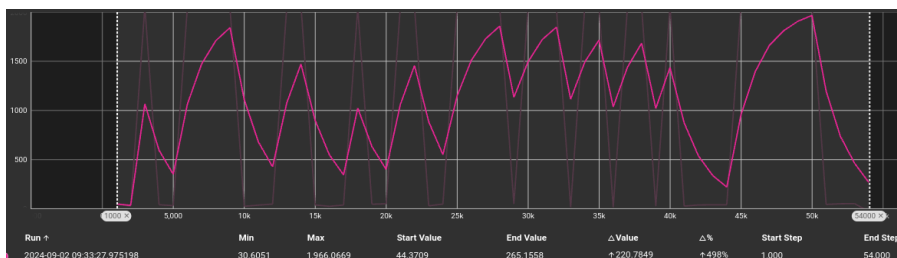
Даљим истраживањем тражило се како да се спрече невалидне акције, само избегавање њих би пореметило тежине неуронских мрежа док њихово извршавање доводи до честих губитака епизода и времена те се често користи техника маскирања невалидних акција. До сада агент је морао да изврши лошу акцију и да буде у том стању како би знао да је лоше. Ово се може спречити сугерисањем агенту да је акција лоша пре него што је изврши што је и назначено у овом раду [12]. У претходним методама бирала се акција која је аргумент максималне Q вредности која се може добити у том стању и ове методе су вредносно засноване док се код метода заснованим на полисама бира акција на основу пробалистичког модела (нпр. Softmax функција), PPO је такав алгоритам. PPO је изузетно популаран у hard exploring RL проблемима као и проблемима везаним за роботiku, такође може се користити и са континуалним акционим просторима.

Прва имплементација у овом раду са овим алгоритмом користила је маскирање акција. Како алгоритми засновани на полисама враћају вероватноће на излазу стратегија је да се невалидним акцијама пре улаза у слој вероватноће замени јако малим скаларом те ће вероватноћа на излазу бити приближна 0 [13]. Имплементација овога постоји у библиотеци Stable-Baselines3 и за сваку коресподентну акцију треба да јој се проследи коресподентна bool вредност у зависности од тога да ли је акција валидна или не, Листинг 5.16. Стратегија овим значајно убрзава конвергенцију [14].

```
def action_masks(self):
    mask = np.zeros(self.action_space.n,
dtype=bool)
    for action in range(self.action_space.n):
        mask[action] =
self.is_valid_action(action)
...
model = MaskablePPO("MlpPolicy", env, gamma=0.99,
seed=32, verbose=1, learning_rate=0.0003)
model.learn(total_timesteps=250000,
callback=tensorboard_callback)
```

Листинг 5.16 маскирање акција

Након овога број епизода у којима робот пада је значајно смањен чиме се ефикасније истражује простор стања. Зглобови се и даље могу пресавити у нежељену позицију услед недостатка снаге мотора али не у циљу намерног истраживања агента. Агент препознаје различите ситуације и повремено устане међутим резултати осцилирају Листинг 5.9.



Слика 5.8 један од резултата средње вредности награда у зависности од броја корака симулације

Мењањем параметара `learning rate`, `gamma`, `batch size`, `n_steps` исходovalo је сличним резултатима.

Следећи корак је био прелазак акција на континуални простор, сада акције нису више повећавање/смањивање одређених зглобова већ сама позиција зглоба у опсегу  $[-1.5, 1.5]$  што нам такође уклања потребу за маскама. Не може се више користити категоријска Softmax функција јер нам треба континуална функција те је

искоришћена Гаусова нормална дистрибуција. Свака компонента акције сада има своју медијану и стандардну девијацију.

PPO је Actor/Critic архитектура, Critic процењује функцију вредности, док Actor ажурира дистрибуцију политике у правцу који је предложио Critic (као што је са градијентима политике). Имплементација се види на

Листинг 5.17.

```
class Agent(nn.Module):
    def __init__(self, envs):
        super(Agent, self).__init__()
        self.critic = nn.Sequential(
            layer_init(nn.Linear(np.array(envs.single_observation_space.shape).prod(), 64)),
            nn.Tanh(),
            layer_init(nn.Linear(64, 64)),
            nn.Tanh(),
            layer_init(nn.Linear(64, 1), std=1.0),
        )
        self.actor_mean = nn.Sequential(
            layer_init(nn.Linear(np.array(envs.single_observation_space.shape).prod(), 64)),
            nn.Tanh(),
            layer_init(nn.Linear(64, 64)),
            nn.Tanh(),
            layer_init(nn.Linear(64,
np.prod(envs.single_action_space.shape)),
std=0.01),
        )
        self.actor_logstd =
nn.Parameter(torch.zeros(1,
np.prod(envs.single_action_space.shape)))

    def get_value(self, x):
        return self.critic(x)

    def get_action_and_value(self, x, action=None):
        action_mean = self.actor_mean(x)
        action_logstd =
self.actor_logstd.expand_as(action_mean)
        action_std = torch.exp(action_logstd)
```

```

probs = Normal(action_mean, action_std)
if action is None:
    action = probs.sample()
    return action,
probs.log_prob(action).sum(1),
probs.entropy().sum(1), self.critic(x)

```

Листинг 5.17 Actor/Critic мреже [15]

Помоћ при конвергенцији и осцилацијама доноси и нормализација награда и опсервација, опсервације су нормализоване стандардизацијом, док су награде нормализоване дељењем са стандардном девијацијом, такође додељен им је максимални и минимални опсег (clipping) у коме могу да се налазе, gym библиотека има одговарајуће wrapper класе, Листинг 5.18. Clipping је коришћен и у самом PPO, више о PPO на [15].

```

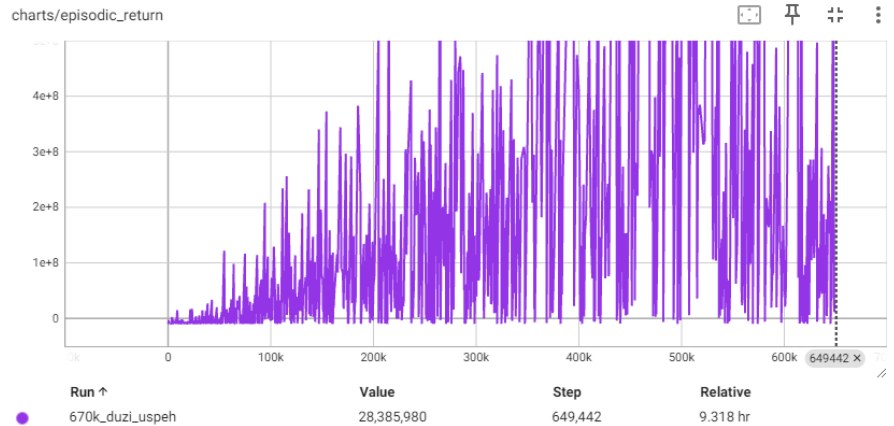
env = gym.wrappers.NormalizeObservation(env)
env =
gym.wrappers.TransformObservation(env, lambda obs:
np.clip(obs, -10, 10))
env = gym.wrappers.NormalizeReward(env)
env = gym.wrappers.TransformReward(env,
lambda reward: np.clip(reward, -10, 10))

```

Листинг 5.18 wrapper класе за нормализацију и clipping

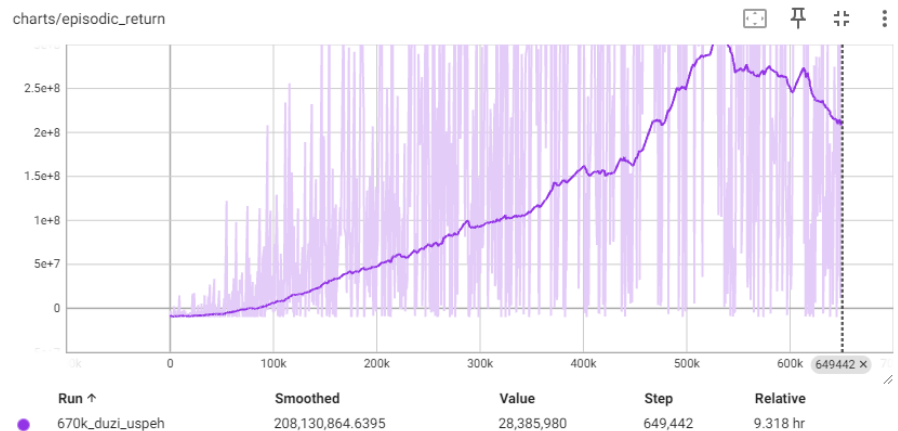
Параметри су подешени у сагласности са MuJoCo препоруком за сличне проблеме нађеном у [15]. Такође и имплементација је модификовани код са [15].

Након ове имплементације резултати су задовољавајући. Графикон на Слика 5.9 приказује вредности награда епизода.



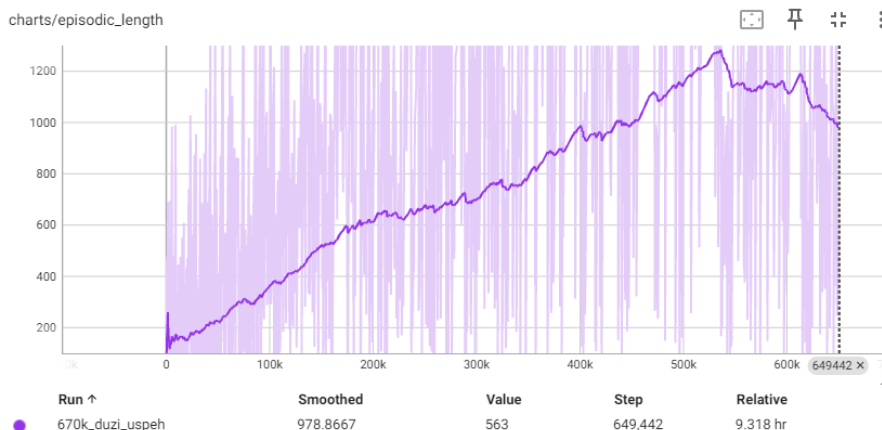
Слика 5.9 награде по епизодама

Како је велика разлика између доброг и лошег исхода са графикона делује да успех није задовољавајући међутим уколико применимо smoothing алат у Tensorboard-у видимо пораст.



Слика 5.10 награде по епизодама уз помоћ smoothing алата

Како су лоше награде негативне графикон мало осцилира али обзиром да се овај покушај састоји од приближно 650 000 корака. Због моделовања функције награда не треба посматрати саме вредности већ сам тренд.



Слика 5.11 дужина епизода у зависности од глобалног корака

Са Слика 5.11 дужина епизода у зависности од глобалног корака се види да тренд награда скоро линеарно прати тренд дужине епизода што је и жељени исход. Тренирање овог модела трајало је приближно 9.3 сата. Евалуацијом се увидело да се робот може одржати и барем 10 минута, стога је направљено ограничење са горње стране од 1000 корака. Евалуацијом над 10 узорака 6/10 је задовољило услов од 1000 корака.

## 5.2 Хардвер

Хардвер је одрађен по узору на [2], тело је конструисано помоћу 3D штампе, унутар thorax-а се налази Lynxmotion SSC-32U плоча која служи за управљање серво моторима. Ради олакшања оптерећења за разлику од [2] робот не носи батерије, не садржи џојстик као управљач и нема микроконтролер. Управљање се врши са рачунара помоћу FT232R USB UART серијске комуникације [16] као што се види на Листинг 5.19.

```
ssc32 = serial.Serial(
    port='/dev/ttyUSB',
    baudrate=9600
)

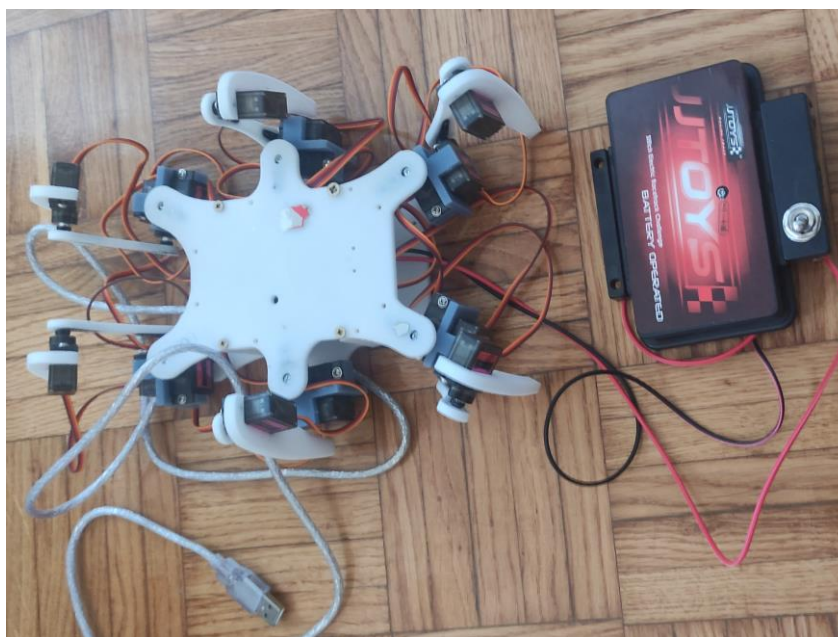
servo_pins = [31, 27, 23, 15, 11, 7, 30, 26, 22,
14, 10, 6, 29, 25, 21, 13, 9, 5]
```

```
is_mirrored = [False, False, False, True, True,
True, False, False, False, True, True, False,
False, False, True, True, True]
```

```
def set_joint_states(joints_states):
    command = ""
    for i in range(18):
        if i != 0:
            command += " "
            command += "#{0}
P{1}".format(servo_pins[i],
angle_rad_to_pwm(joints_states[i], is_mirrored[i]))
        command += "\r"
    command_bytes = str.encode(command)
    rospy.loginfo(command)
    try:
        ssc32.write(command_bytes)
    except Exception as e:
        rospy.logerr("Failure>>" + str(e))
    exit()
```

Листинг 5.19 слање команди роботу

Изглед хардвера види се на Слика 5.12.



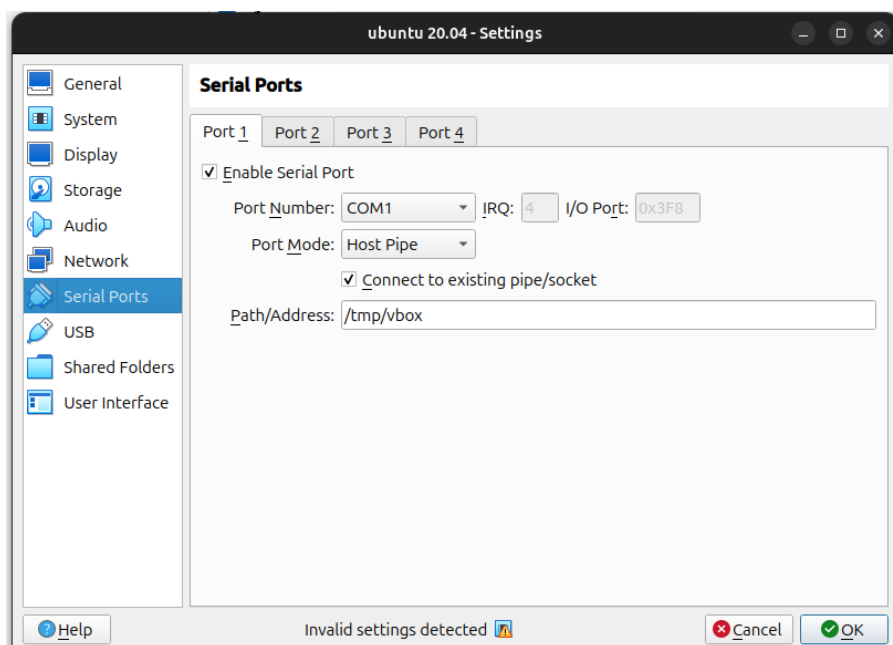
Слика 5.12 изглед хардвера



## 6. ДЕМОНСТРАЦИЈА

Пре свега неопходно је напоменути да уколико се софтвер не налази у виртуелној машини може се повезати без следећих корака, тј. може се користити комуникација преко USB-а помоћу COM протокола у Windows-у и ttyUSB протокола у Unix системима. Уколико се софтвер налази у виртуелној машини подешавање је другачије. Како подешавање Host Device у VirtualBox софтверу за виртуелизацију није успео да буде подешен, искоришћено је друго решење и односи се на Unix host системе:

- помоћу алата socat креира се socket пре покретања виртуелне машине и повезати га тако да шаље све податке на ttyUSB, Листинг 6.1 команде за покретање
- у подешавањима guest оперативног система подешавања треба поставити према Слика 6.1 подешавања Serial порта и
- затим покренути одговарајуће команде за дозволе приступ, Листинг 6.1 команде за покретање



Слика 6.1 подешавања Serial порта

```

socat -d -d UNIX-LISTEN:/tmp/vbox, fork
/dev/ttyUSB0,raw
chmod 777 /dev/ttyUSB0
chmod 777 /tmp/vbox

# set serial port in vbox as COM1, Host Pipe, check
Connect to existing, path is /tmp/vbox
#run vm
#in guest run chmod 777 /dev/ttyS0
#run the software

```

#### Листинг 6.1 команде за покретање

Покретање софтвера се врши из неколико корака:

- учитавање колекција ROS алата који између осталог и омогућавају комуникацију ROS чворова,
- учитавање модела и покретање Gazebo симулације што приказује,
- учитавање управљача контролера и креирање контролера и
- сада се избор своди на то да ли се покреће алгоритам за уење или извршавање

Сви кораци су приказани у истом редоследу на Листинг 6.22.

```

roscore
roslaunch urdf_demo main.launch
roslaunch urdf_demo controllers.launch
roslaunch hexapod_training main.launch
# or
roslaunch hexapod_training
neural_executioner.launch

```

#### Листинг 6.2 команде за покретање

## 7. ЗАКЉУЧАК

Како имплементација многих робота захтева комплексне и скупе делове идеја је била да се нађе дизајн који се може обавити уз помоћ 3D штампања и коришћењем малих серво мотора.

Инспирација за задатак је Crawler робот са курса за увод у вештачку интелигенцију универзитета Berkley. Робот је уз помоћ Reinforcement learning-а учио да хода у симулацији те је идеја била да се то физички имплементира и отежа тиме што би робот уместо једне имао 6 ногу.

Робот је научен да стоји и балансира се у том положају у симулационом окружењу помоћу РРО алгоритма, сачувана неуронска мрежа се учитава заједно са симулацијом и роботу се шаљу стања која треба да изврши.

Слични системи су пронађени само у виду писаних радова, ретко кода.

Могућа проширења обухватају transfer learning [17] тј. да робот учи што више може у симулацији а да потом то искористи за учење у реалном окружењу које треба да се адаптира. Ово би подухватало додавање разних сензора. Потенцијални следећи корак би био и учење ходању као и постављање микроконтролера или Raspberry Pi како робот не би био позиционо везан кабловима те се може слободно кретати независно од тога. Извршавање на рачунарима са више ресурса је такође једно од могућим усмерење и мала модификација у циљу паралелизације.



## 8. ЛИТЕРАТУРА

- [1] <https://youtu.be/yNeSFbE1jdY>
- [2] <https://www.instructables.com/3D-Printed-18DOF-Hexapod/>
- [3] <https://www.youtube.com/watch?v=HrapVFNB64&pp=ygUNcHBvIGV4cGxhaW5lZA%3D%3D>.
- [4] <https://huggingface.co/learn/deep-rl-course/en/unit3/deep-q-algorithm>.
- [5] <https://www.youtube.com/watch?v=HrapVFNB64&pp=ygUNcHBvIGV4cGxhaW5lZA%3D%3D>.
- [6] <https://wiki.ros.org/>.
- [7] <https://github.com/openai/gym>.
- [8] <https://www.tensorflow.org/tensorboard>.
- [9] <https://github.com/pytorch/pytorch>.
- [10] <https://www.instructables.com/Capers-II-a-Hexapod-Robot/>.
- [11] Dqwd
- [12] [https://classic.gazebosim.org/tutorials?tut=ros\\_control](https://classic.gazebosim.org/tutorials?tut=ros_control).
- [13] <https://apmonitor.com/pdc/index.php/Main/ProportionalIntegralControl>.
- [14] Kane, Thomas R., and David A. Levinson. *Dynamics, theory and applications*. McGraw Hill, 1985.
- [15] <https://lilianweng.github.io/posts/2020-06-07-exploration-drl/>.
- [16] <https://huggingface.co/learn/deep-rl-course/en/unit3/from-q-to-dqn>.
- [17] <https://github.com/Nemanja3214/crawler-robot>.
- [18] Seurin, Mathieu, Philippe Preux, and Olivier Pietquin. "I'm Sorry Dave, I'm Afraid I Can't Do That" Deep Q-Learning from Forbidden Actions." *2020 International Joint Conference on Neural Networks (IJCNN)*. IEEE, 2020.
- [19] Huang, Shengyi, and Santiago Ontañón. "A closer look at invalid action masking in policy gradient algorithms." *arXiv preprint arXiv:2006.14171* (2020).
- [20] [https://sb3-contrib.readthedocs.io/en/master/modules/ppo\\_mask.html](https://sb3-contrib.readthedocs.io/en/master/modules/ppo_mask.html).

- [21] <https://iclr-blog-track.github.io/2022/03/25/ppo-implementation-details/>.
- [22] <http://arxiv.org/abs/1707.06347>.
- [23] [https://ftdichip.com/wp-content/uploads/2020/08/DS\\_FT232R.pdf](https://ftdichip.com/wp-content/uploads/2020/08/DS_FT232R.pdf).
- [24] Pan, Sinno Jialin, and Qiang Yang. "A survey on transfer learning." *IEEE Transactions on knowledge and data engineering* 22.10 (2009): 1345-1359.
- [25] [https://www.researchgate.net/figure/Schematic-diagram-of-Euler-angle-RPY-angle\\_fig5\\_347296205](https://www.researchgate.net/figure/Schematic-diagram-of-Euler-angle-RPY-angle_fig5_347296205).
- [26] <https://boring-guy.sh/posts/masking-rl/>.

## **9. БИОГРАФИЈА**

Немања Мајсторовић рођен је 10.10.2001. године у Краљеву, где је стекао основно и средње образовање. Школске 2020/2021 године се уписује на Факултет техничких наука на студијски програм Софтверско инжењерство и информационе технологије. Положио је све испите предвиђене планом и програмом и стекао услов за одбрану завршног рада.





## КЉУЧНА ДОКУМЕНТАЦИЈСКА ИНФОРМАЦИЈА

Редни број, <b>РБР:</b>	
Идентификациони број, <b>ИБР:</b>	
Тип документације, <b>ТД:</b>	монографска публикација
Тип записа, <b>ТЗ:</b>	текстуални штампани документ
Врста рада, <b>ВР:</b>	дипломски рад
Аутор, <b>АУ:</b>	Немања Мајсторовић
Ментор, <b>МН:</b>	др Милан Видаковић, редовни професор
Наслов рада, <b>НР:</b>	Имплементација Reinforcement learning алгоритма за балансирање hexapod робота
Језик публикације, <b>ЈП:</b>	српски
Језик извода, <b>ЈИ:</b>	српски / енглески
Земља публикација, <b>ЗП:</b>	Србија
Уже географско подручје, <b>УГП:</b>	Војводина
Година, <b>ГО:</b>	2024
Издавач, <b>ИЗ:</b>	ауторски репринт
Место и адреса, <b>МА:</b>	Нови Сад, Факултет техничких наука, Трг Доситеја Обрадовића 6
Физички опис рада, <b>ФО:</b>	9 / 64 / 26 / 0 / 20 / 4 / 0
Научна област, <b>НО:</b>	Инжењерство софтвера за Internet/Web of things
Научна дисциплина, <b>НД:</b>	Софтверско инжењерство
Предметна одредница / кључне речи, <b>ПО:</b>	hexapod, вештачка интелигенција, PPO, RL
<b>УДК</b>	
Чува се, <b>ЧУ:</b>	Библиотека Факултета техничких наука, Трг Доситеја Обрадовића 6, Нови Сад
Важна напомена, <b>ВН:</b>	
Извод, <b>ИЗ:</b>	Задатак представља развој система за учење hexapod робота да из положаја спуштених ногу устане и стоји балансирајући се у том положају уз помоћ Reinforcement learning алгоритма. Велики део система је имплементирам помоћу ROS(Robot Operating System) алата док су скрипте коришћене за учење имплементирани у Python програмском језику. Апликација омогућава учитавање упрошћеног модела и његово тренирање у симулацији затим научене акције се преносе на реалног робота.
Датум прихватања теме, <b>ДП:</b>	
Датум одбране, <b>ДО:</b>	

Чланови комисије, КО:	
председник	др Милан Сегединац, ванредни професор
члан	др Марко Марковић, ванредни професор
ментор	др Милан Видаковић, редовни професор
Потпис ментора	

## KEY WORDS DOCUMENTATION

Accession number, <b>ANO</b> :	
Identification number, <b>INO</b> :	
Document type, <b>DT</b> :	monographic publication
Type of record, <b>TR</b> :	textual material
Contents code, <b>CC</b> :	bachelor thesis
Author, <b>AU</b> :	Nemanja Majstorović
Mentor, <b>MN</b> :	Milan Vidaković, full professor, PhD
Title, <b>TI</b> :	Implementation of Reinforcement learning algorithm for balancing hexapod robot
Language of text, <b>LT</b> :	Serbian
Language of abstract, <b>LA</b> :	Serbian / English
Country of publication, <b>CP</b> :	Serbia
Locality of publication, <b>LP</b> :	Vojvodina
Publication year, <b>PY</b> :	2024
Publisher, <b>PB</b> :	author's reprint
Publication place, <b>PP</b> :	Novi Sad, Faculty of Technical Sciences, Trg Dositeja Obradovića 6
Physical description, <b>PD</b> :	9 / 64 / 26 / 0 / 20 / 4 / 0
Scientific field, <b>SF</b> :	Software engineering for Internet/Web of things
Scientific discipline, <b>SD</b> :	Software Engineering
Subject / Keywords, <b>S/KW</b> :	Hexapod, artificial intelligence, PPO, RL
<b>UDC</b>	
Holding data, <b>HD</b> :	Library of the Faculty of Technical Sciences, Trg Dositeja Obradovića 6, Novi Sad
Note, <b>N</b> :	
Abstract, <b>AB</b> :	This thesis represents the development of a system for teaching a hexapod robot to stand up from the position of its legs down and stand while balancing in that position with the help of the Reinforcement learning algorithm. A large part of the system is implemented using the ROS (Robot Operating System) tool, while the scripts used for learning are implemented in the Python programming language. The application allows loading a simplified model and its training in a simulation, then the learned actions are transferred to a real robot.
Accepted by sci. Board on, <b>ASB</b> :	
Defended on, <b>DE</b> :	
Defense board, <b>DB</b> :	
president	Milan Segedinac, associate professor, PhD

member	Marko Marković, associate professor, PhD
mentor	Milan Vidaković, full professor, PhD
Mentor's signature	