УНИВЕРЗИТЕТ “Св. КИРИЛ И МЕТОДИЈ” - СКОПЈЕ

ФАКУЛТЕТ ЗА ЕЛЕКТРОТЕХНИКА И ИНФОРМАЦИСКИ ТЕХНОЛОГИИ

- ДИПЛОМСКА РАБОТА -

по предметот

РОБОТИКА 1

Тема

ФИЗИЧКА СИМУЛАЦИЈА НА РОБОТСКА РАКА

Ментор:
Доц. д-р Горјан Нацински

Изработил:
Васил Трендафилов, индекс бр. 139/2018
e-mail: vasetrendafilov@gmail.com

Скопје, декември 2022

Содржина

Листа на слики

Листа на табели

Апстракт

Симулацијата на роботска рака е моќна алатка која им овозможува на дизајнерите да ги тестираат и проценат перформансите на управувачките алгоритми и стратегиите за планирање на движење во виртуелна средина. Со симулирање на динамиката и физичките ограничувања на роботот, како и со инкорпорирање на модели на сензори и други фактори од реалниот свет, можно е точно да се предвиди однесувањето на роботот и да се идентификуваат сите потенцијални проблеми пред да се имплементира управувачки систем на физичката рака. Симулацијата обезбедена од pybullet, исто така, може да овозможи лесна визуелизација и анализа на однесувањето на роботот, што го прави вредна алатка за оптимизирање и дотерување на управувачките системи. Исто така, има многу модули кои помагаат при раката, како управување на додатокот, сликање, запишување на сите зглобови и цртање на траекторијата. Може исто така да креирате модуларна роботска рака од нула или да внесете постоечка прилагодена URDF рака. Генерално, симулацијата е суштинска алатка за развој и евалуација на роботски раце.

Клучни зборови: симулација, креирање на рака, управување на рака

Abstract

Robot arm simulation is a powerful tool that allows designers to test and evaluate the performance of control algorithms and motion planning strategies in a virtual environment. By simulating the dynamics and physical constraints of a robot, as well as incorporating sensor models and other real-world factors, it is possible to accurately predict the behavior of a robot and identify any potential issues before implementing the control system on a physical robot. The simulation provided by pybullet can also enable easy visualization and analysis of the robot's behavior, making it a valuable tool for optimizing and fine-tuning control systems.There are also plenty of modules that help with the arm like actuating the attachment, taking a picture, loging all the joints and drawing the trajectory. You can also create a modular robot arm from scratch or import an existing custom URDF arm. Overall the simulation is an essential tool for the development and evaluation of robot arms.

Keywords: simulation, creating an arm, controlling an arm

Вовед

Симулација на роботска рака е компјутерска програма што им овозможува на корисниците да дизајнираат и симулираат движење и функционирање на роботска рака. Овие симулации може да се користат за различни цели, како што се тестирање и развој на управувачки алгоритми, обука на оператори или демонстрирање на способностите на роботските раце. Симулацијата може да се користи за да се предвиди движењето и однесувањето на раката на роботот под различни услови, како што се различни тежини или средини. Исто така може да се користи во образованието и обуката, која е главната примена на овој проект, овозможувајќи им на студентите да учат за роботиката и управувачките системи во безбедна и контролирана средина.

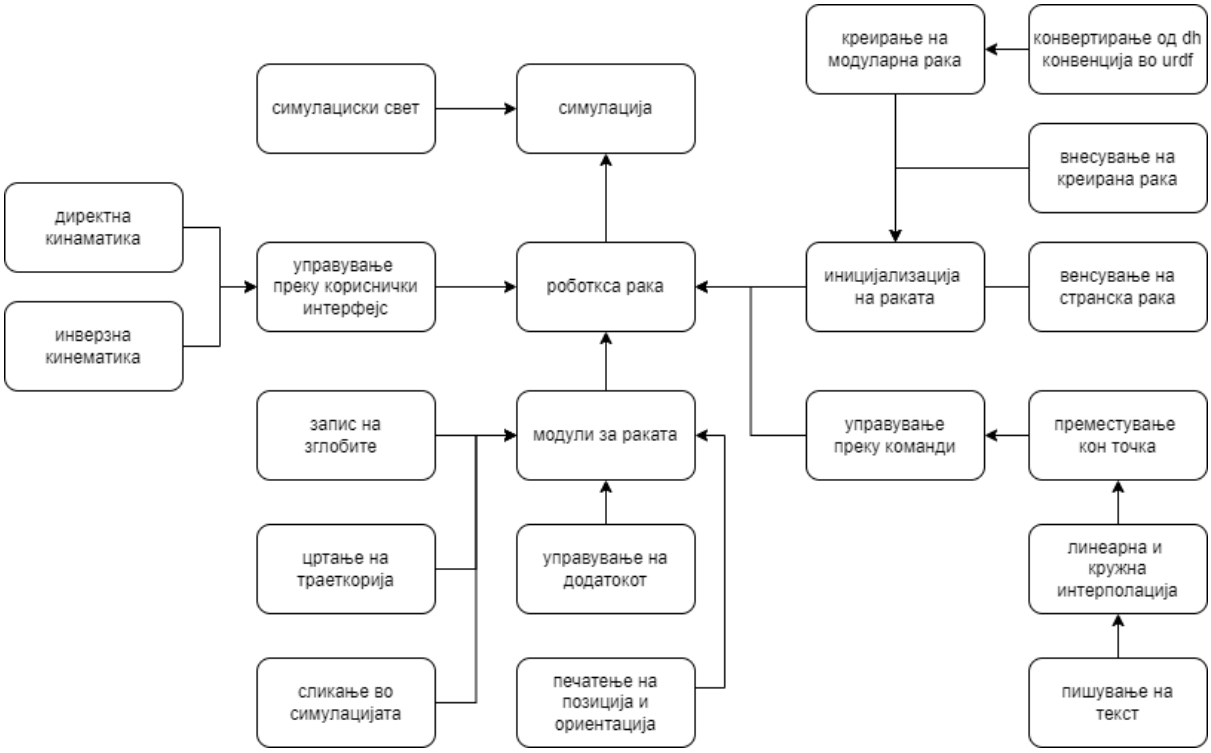
Во проектот ќе разгледаме како роботска рака може лесно да се креира според на DH (Denavit–Hartenberg) конвенцијата или вчита постоечка URDF (Unified Robot Description Format) датотека. Како да се управува со роботот преку графички кориснички интерфејс или преку команди во симулиран свет со генерирани пречки и предизвици. Исто така како е имплементирана раката mitsubishi RV-2F-Q каде се објаснува како да се воспостави конекција преку интернет и како да се генерира програма од симулацијата која ќе се изврши на реалната рака.

Со следниот дијаграм на Слика 1 ќе се објаснат сите функционалности на проектот во кратки црти и која е поврзаноста меѓу нив. Главниот дел од целиот проект е симулацијата која е обезбедена од pybullet и се разгранува на:

- симулацискиот свет кој се состои од голем број на објекти кои се внесуваат во симулацијата и служат за интеракција со раката,
- роботската рака која е главната имплементација на проектот и е поделена на четири групи на функционалности.

Постојат три начини за полнење на раката, креирање на модуларна рака со DH конвенцијата и внесување на веќе креирана или надворешна рака. Имаме две опции за управување на раката, преку интерактивен кориснички интерфејс каде имаме опција да ја управуваме преку директна или инверзна кинематика со поставени лизгачи или преку команди каде имаме хиерархија која започнува со придвижување на раката кон посакувана точка, па линеарно и кружно интерполирање помеѓу две точки и на крај цртање на текст во кој се користи интерполацијата. Последно имаме дополнителни модули кои се додаваат на раката и ја подобруваат функционалноста:

- цртање не траекторијата на извршниот елемент на раката кој служи за визуелна проверка при извршување на поместувањата,
- печатење на позицијата и ориентацијата на извршниот елемент на раката кој служи за прецизно визуелно следење на извршниот елемент,
- сликање на симулацијата од извршниот елемент на раката каде добиваме слика во боја, длабочина на пикеслот и сегментација на сликата,
- управување за додатокот на раката каде моменталната функционалност е активација на фаќач,
- запис на секој зглоб од раката кој служи за проверка на поместувањата и нивна конверзија во програма за управување на mitsubishi раката.



Слика 1 Општ дијаграм за опис на проектот

Во проектот ќе бидат изложени сите користени технологии и софтверски алатки со кои е креиран софтверскиот пакет. Прво ќе се објасни како да се инсталира софтверскиот пакет и потребните библиотеки и како да се користи кодот преку jupyter тетратки во детали. Дополнително ќе се објасни корисничкиот интерфејс на pybullet и како да се воспостави конекција со управувачот на mitsibishi раката. Следно се објаснува архитектурата, каде во кодирањето се однесува на целокупниот дизајн и структура на софтверскиот систем. Главниот дел од проектот е документацијата каде во детали ќе се објаснат сите класи со нивните функционалности. На крај има поставено потточка за надградба на проектот каде се објаснува како да се внесе нова рака, додаток и симулациски свет. Можни нови функционалности и голем број на примери за имплементација на кодот.

1 Користени технологии и алатки

Софтверскиот пакет содржи три меѓусебно поврзани библиотеки. Главната библиотека е pybullet [6] кој е брз и лесен за употреба софтверски пакет за симулација на роботи и машинско учење, со фокус на трансфер од симулацијата во реалноста. Обезбедува симулација на директна динамика, пресметување на инверзна динамика, директна и инверзна кинематика, детекција на судир и пресметка за пресек на зраци. Воглавно во проектот се користи за симулирање на раката, креирање на светот околу неа и запишување на сите зглобови.

Основата за создавање на роботската рака е направена со sympy [7] која е библиотека на пајтон за симболична математика. Се користи за пресметка на DH матрицата и јакобијанот на раката. Третата библиотека е numpy [8] која се користи за олеснување на операциите со матрици, линеарна интерполација помеѓу две точки, конверзија од радијани во степени и споредување на низи.

2 Инсталација на софтверскиот пакет

2.1 Инсталација на потребни апликации

Основни софтверски апликации кои ги употребува пакетот се python со соодветен инсталатер на библиотеки и веб пребарувач за користење на jupyter тетратките. Популарна алатка за инсталирање на библиотеките која се користи е mini conda кој е бесплатен минимален инсталатер за conda. Чекорите за инсталација на апликацијата може да се следат во следниот линк од нивната документација [3].

Освен веб пребарувач, jupyter тетратките може да се отворат и со софтверскиот пакет vscode [4] каде имаме вградена екстензија за нивно вчитување. Како последен софтверски пакет кој е опционален е гит [5] кој е бесплатен и дистрибуиран систем за контрола на верзии со отворен код. Може лесно да се клонира проектот и понатаму да се надградуваат неговите функционалности.

2.2 Инсталација на библиотеки и клонирање на проектот

Потребни библиотеки кои проектот ги користи се pybullet, sympy и numpy. Верзијата на python нема потреба да се одбира бидејќи пакетите за инсталирање на библиотеките автоматски ја одбира заедничката верзија за сите библиотеки. Со следните линии команди ќе се покажат чекорите потребни за клонирање на софтверскиот пакет, инсталирање на потребните библиотеки и стартување на jupyter-lab серверот:

1. Клонирање на софтверскиот пакет:
   git clone https://github.com/vasetrendafilov/robot-simulation
2. Креирање на околината и инсталирање на потребните библиотеки:
   conda create --name robot_simulation -c conda-forge pybullet jupyterlab sympy
3. Активација на околината
   conda activate robot_simulation
4. Стартување на jupyter-lab за отворање на jupyter тетратките
   jupyter lab

3 Користење на софтверскиот пакет

Проектот доаѓа со три jupyter тетратки [10] каде што се имплементирани сите функционалности на софтверскиот пакет. Во следните под точки ќе се објаснат сите тетратки во детали со сите чекори како да се креира или внесе роботска рака, управува со неа преку кориснички интерфејс или команди и објаснување на модулите на раката.

3.1 Модуларна роботкса рака

Се креираат модуларни роботски раце користејќи ја DH конвенцијата каде може да се креира нов транслациски, ротациски или фиксен зглоб. Примери за управување со нив и пресметка на DH матрица на раката и пресметка на линеарен и ротациски јакобијан на раката.

На почеток се внесуваат библиотеките sympy и numpy кои ќе се користат за креирање на симболички променливи и олеснување при работа со матриците. Следно се полнат класите RobotArm() за креирање на робтоска рака и PybulletSimulation() за креирање на светот околу раката.

```
import numpy as np
import sympy as sp
from packages.robot_arm import RobotArm
from packages.pybullet_sim import PybulletSimulation
```

3.1.1 Креирање и пресметување на DH матрица и јакобијан на раката

Со sympy се креираат симболички променливи за параметрите на зглобовите. Потоа се користи класата RobotArm() каде прво се иницијализира и се додаваат зглобови според DH конвенцијата. Освен четирите основни параметри (тета, d, a, алфа), во функциите дополнително може да се постават долни и горни граници, максималната брзина и забрзување на зглобот. На крај од блокот код се внесува раката во симулацијата со функцијата load_robot_arm().

```
d1, theta2, theta3, d4, l2, l3 = sp.symbols('d1, theta2, theta3, d4, l2, l3')
robot = RobotArm()
robot.add_prismatic_joint(0, d1, 0, 0)
robot.add_revolute_joint(theta2, 0, l2, 0)
robot.add_revolute_joint(theta3, 0, l3, sp.pi)
robot.add_prismatic_joint(0, d4, 0, 0)
robot.add_subs([(l2, 2), (l3, 2)])
robot.load_robot_arm()
```

Со функцијата get_dh_joint_to_joint() се пресметува DH матрицата за целата рака, каде на Слика 2 се преставува зависноста на извршниот елемент од позицијата на секој зглоб. Функцијата jacobian() кој резултат е преставен на Слика 3 го пресметува линеарниот и ротацискиот јакобијан на целата роботска рака.

```
robot.get_dh_matrix()
robot.jacobian
```

$$\begin{bmatrix} \cos(\theta_2+\theta_3) & \sin(\theta_2+\theta_3) & 0 & l_2\cos(\theta_2)+l_3\cos(\theta_2+\theta_3) \\ \sin(\theta_2+\theta_3) & -\cos(\theta_2+\theta_3) & 0 & l_2\sin(\theta_2)+l_3\sin(\theta_2+\theta_3) \\ 0 & 0 & -1 & d_1-d_4 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Слика 2 DH матрица на раката

$$\begin{bmatrix} 0 & -l_2\sin(\theta_2)-l_3\sin(\theta_2+\theta_3) & -l_3\sin(\theta_2+\theta_3) & 0 \\ 0 & l_2\cos(\theta_2)+l_3\cos(\theta_2+\theta_3) & l_3\cos(\theta_2+\theta_3) & 0 \\ 1 & 0 & 0 & -1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 \end{bmatrix}$$

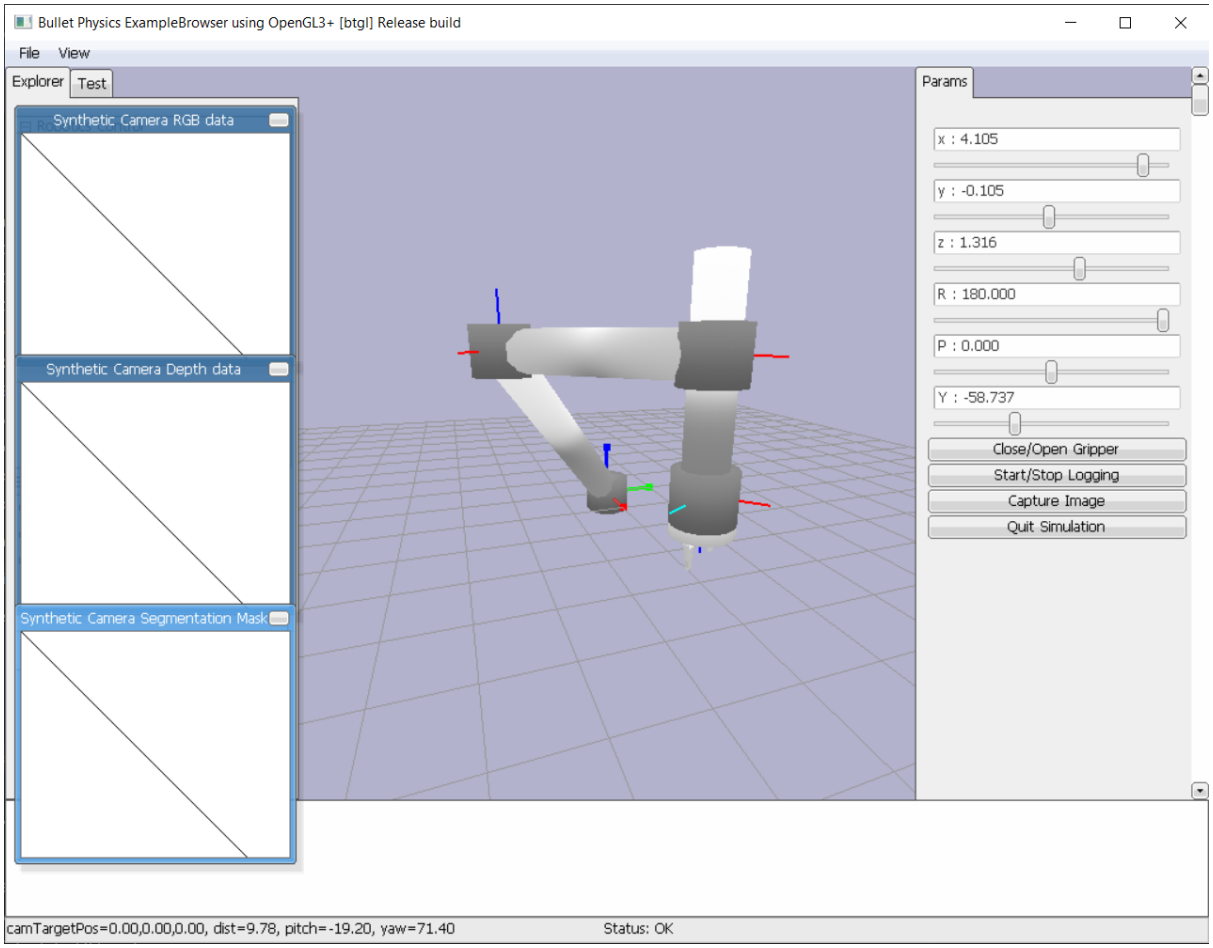
Слика 3 Јакобијан за раката

3.1.2 Инверзна кинематика на роботска рака со транслациски фаќач

Вториот блок код креира модуларна роботска рака на која се поставува транслациски фаќач и интерактивно се управува раката преку кориснички интерфејс. Се користат истите чекори за креирање на модуларна рака како првиот пример. За додавање на фаќачот се користи функцијата add_attachment() кој стандардно се додава транслацискиот фаќач. Финално се повикува главната функција interact() каде се креира интерактивен кориснички интерфејс во pybullet за управување на раката со инверзна кинематика каде стандардно се користи вградената нумеричка апроксимација обезбедена од pybullet.

```
theta1, theta2, theta4, d1, d2, d3, a1, a2 = sp.symbols('theta1, theta2,
theta4, d1, d2, d3, a1, a2')
robot = RobotArm()
robot.add_revolute_joint(theta1, d1, a1, 0)
robot.add_revolute_joint(theta2, 0, a2, -sp.pi)
robot.add_prismatic_joint(0, d3, 0, 0)
robot.add_revolute_joint(theta4, 0, 0, 0)
robot.add_subs([(d1, 3),(a1, 3),(a2, 3)])
robot.add_attachment()
robot.interact('inverse')
```

Корисничкиот интерфејс на pybullet преставен со Слика 4 се составува од главен прозорец каде се прикажува генерираниот свет, каде во моментов е само преставена рообтската рака. На десната страна се генерирани лизгачи и копчиња за управување на раката. Кога се користи инверзна кинематика се поставуваат три лизгачи за позиција на извршниот елемент по оските каде стандардно е од -5 до 5, и три лизгачи за ориентацијата по конвенција RPY каде имаме ротации по x, y и z оски. Исто така има и копчиња за затворање и отворање на фаќач, старт и стоп за запишување на зглобовите, сликање на симулацијата од извршниот елемент на раката и на крај копче за исклучување на симулацијата.



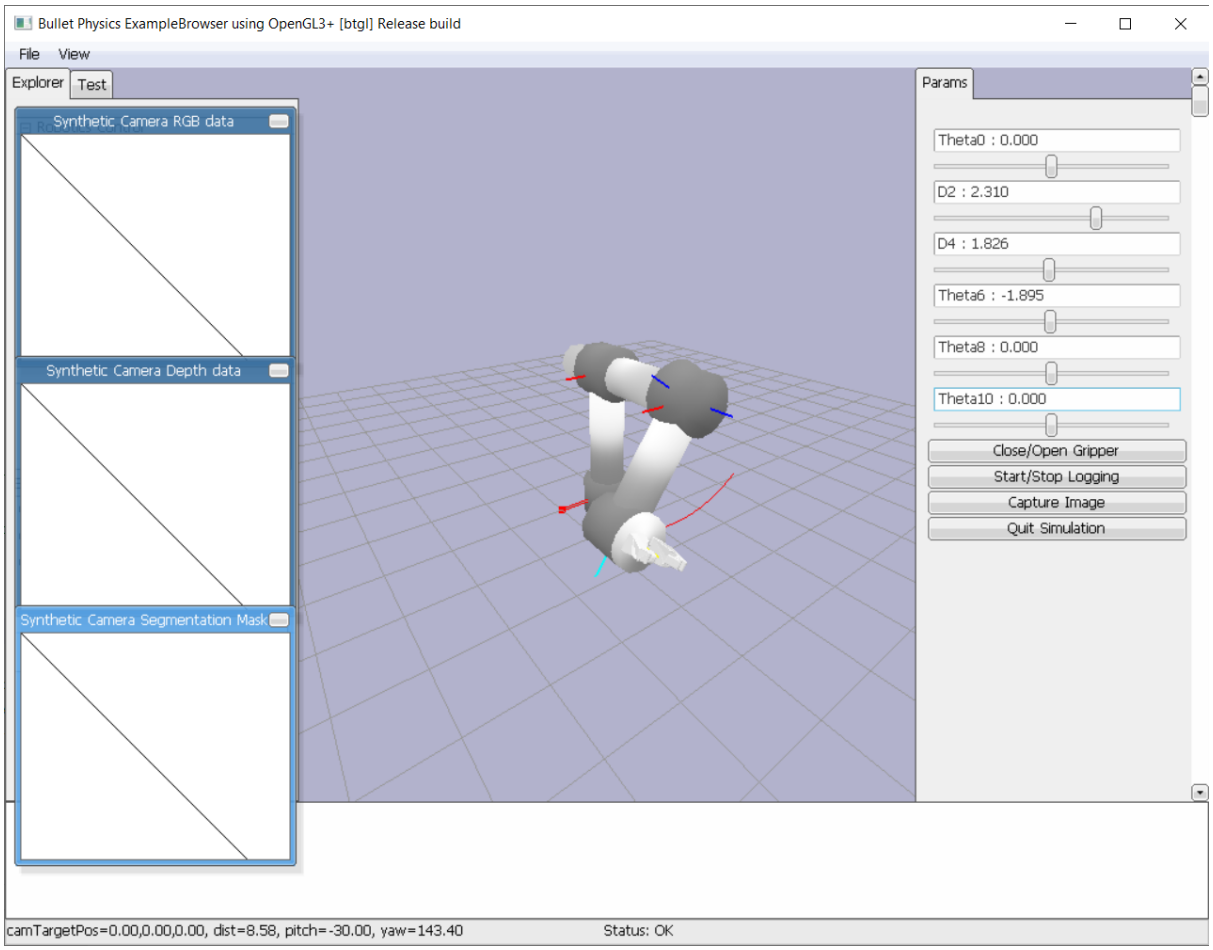
Слика 4 Кориснички интерфејс за управување со инверзна кинематика

3.1.3 Директна кинематика на роботска рака со ротациски фаќач

Следниот пример покажува управување на модуларна роботска рака со ротациски фаќач. Кога се додава нов додаток на раката треба да се постави офсетот и ориентацијата од извршниот елемент и да се постават позициите на отворање и затворање на фаќачот. Кога при управувањето се користи динамика имаме проблем каде има голема грешка во позицијата на ротациските зглобови, бидејќи во примерот имаме многу зглобови кои ротираат во место и симулацијата не може да го измоделира тоа. Едно од решенијата е да се управува раката без динамика, но во овој случај не може да се користи фаќачот.

```
theta1, d2, d3, theta4, theta5, theta6 = sp.symbols('theta1, d2, d3, theta4,
theta5, theta6')
robot = RobotArm(use_dynamics=True)
robot.add_revolute_joint(theta1, 0, 0, 0)
robot.add_prismatic_joint(0, d2, 0, -sp.pi/2)
robot.add_prismatic_joint(0, d3, 0, 0)
robot.add_revolute_joint(theta4, 0, 0, sp.pi/2)
robot.add_revolute_joint(theta5, 0, 0, -sp.pi/2)
robot.add_revolute_joint(theta6, 0, 2, 0,)
robot.add_attachment('revolute_gripper',orientation = (0,-90,0))
robot.set_attachment_targets((0.548,0.548),(0,0))
robot.interact()
```

За управување на роботската рака со директна кинематика, корисничкиот интерфејс е прикажан на Слика 5 каде има мали промени. Лизгачите за промена на позицијата и ориентацијата на извршниот елемент се заменети со лизгач за секој зглоб од раката кој е во поставените граници. Исто така на сликата се прикажува и траекторијата на извршниот елемент каде во овој случај се гледа како се однесува како нишало пробувајќи да ги постави зглобовите на точната позиција.

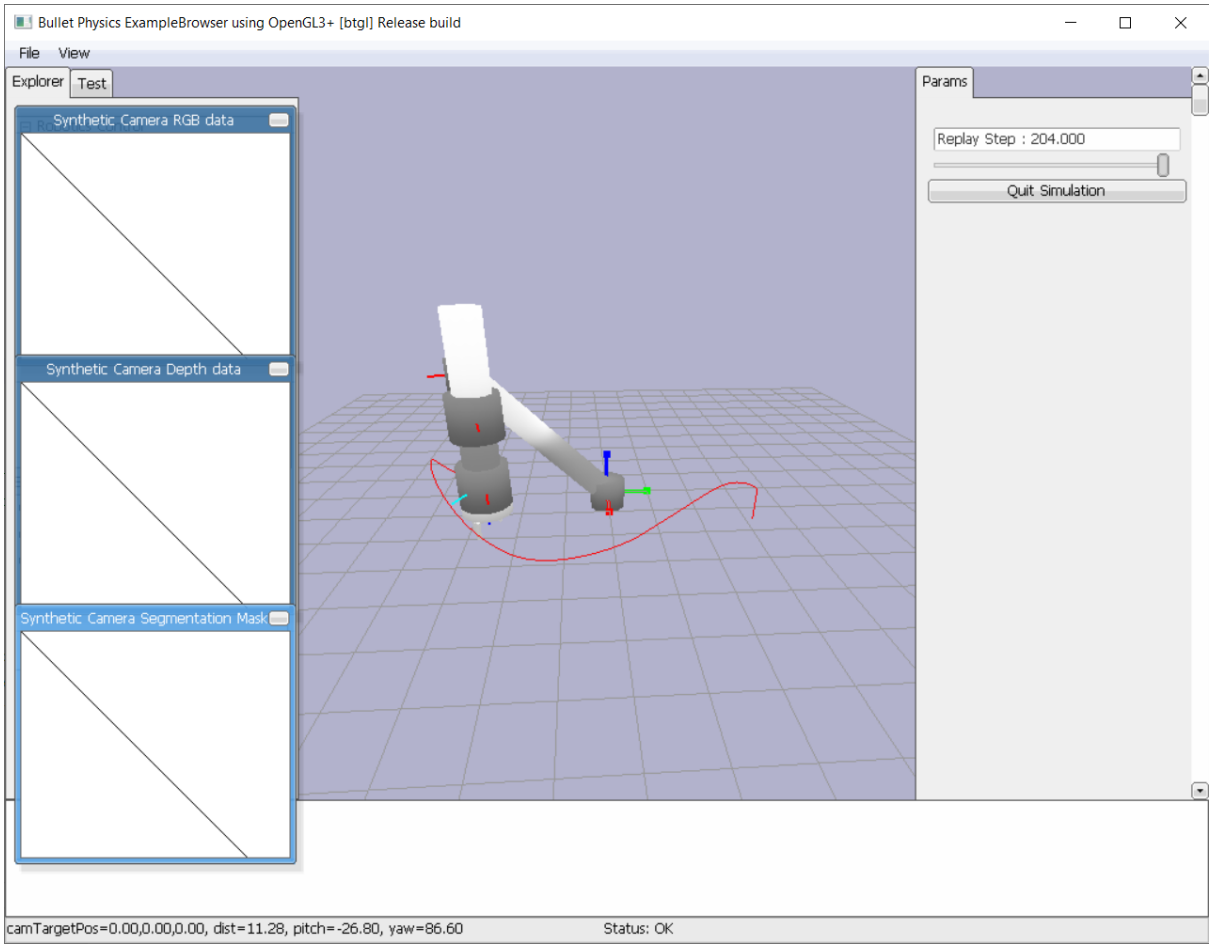


Слика 5 Кориснички интерфејс за управување со директна кинематика

3.1.4 Внесување на рака и репродуцирање на записите

Следниот пример се прикажува како да се внесе веќе креиран робот кој е направен од проектот и како да се репродуцираат зачуваните записи на раката. При репродуцирање на записите има опција да се оди низ секој чекор секвенцијално или да се одбере преку лизгач кој чекор да се изврши. Во Слика 6 е претставен корисничкиот интерфејс за репродуцирање на секој чекор од записот, каде на десната страна има лизгач кој може да помине низ сите чекори и копче за да се изгаси симулација.

```
robot = RobotArm()
robot.import_robot_arm('robot_arms/my_robot/my_robot.urdf')
robot.load_robot_arm()
robot.replay_logs('inverse',skim_trough=True)
```



Слика 6 Кориснички интерфејс за репродуцирање на записите

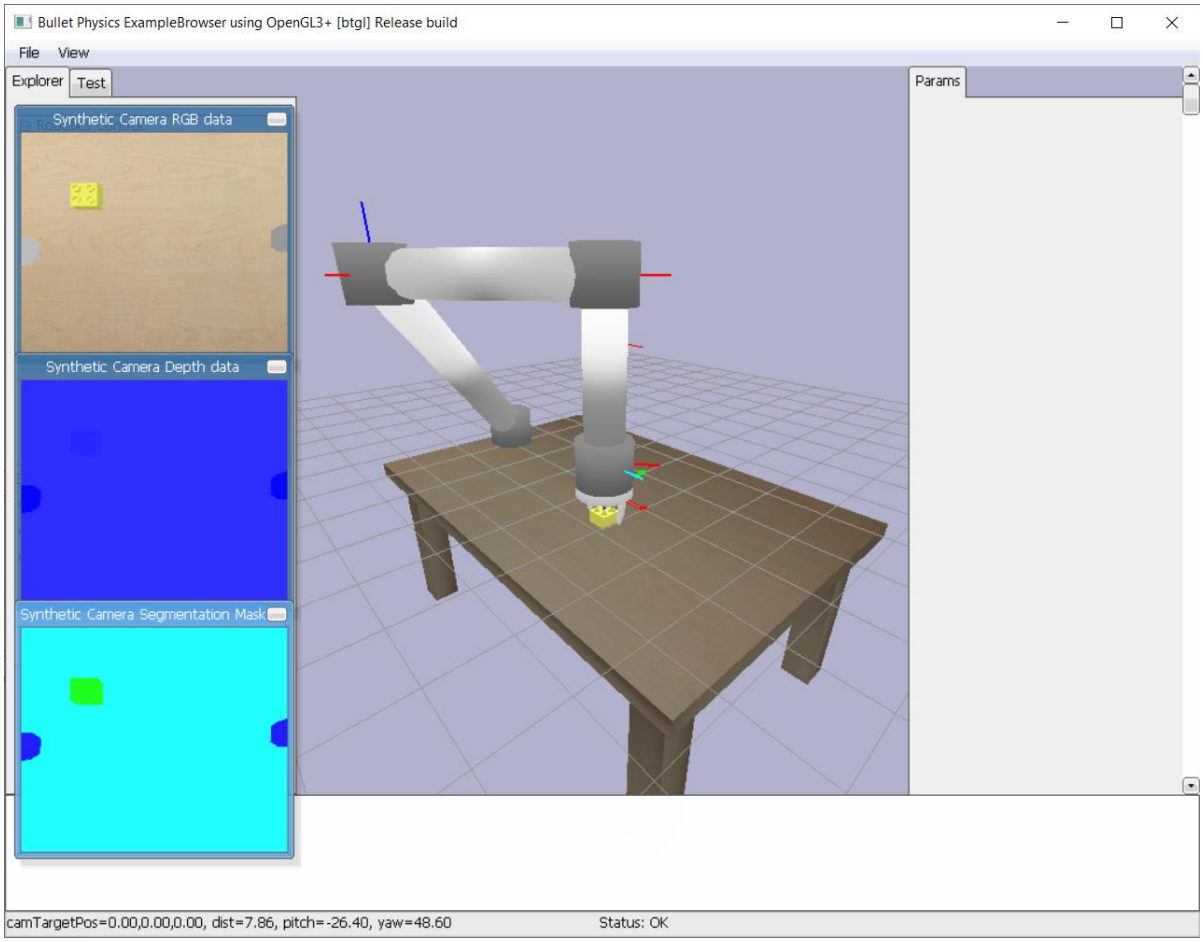
3.1.5 Пример за детекција и земање на објект со фаќач

Следниот пример е едноставна детекција на објект и земање на објектот со фаќачот на роботската рака. Прво се создава светот околу роботксата рака користејќи ја класата PybulletSimulation(), каде се стартува и конектира со pybullet. Се гради светот така што се поставува работна маса, и лего на случајна позиција на масата. Следно се внесува роботската рака и се поставува на едниот дел од масата. За да имаме прецизни движења силата на зглобовите се намалуваат на 400 и се поставуваат динамичките услови да грешката во зглобовите да е многу мала и да чека многу чекори за да запре ако не успее да стигне до саканата позиција.

```
sim = PybulletSimulation()
sim.connect()
sim.load_table()
sim.load_lego((np.random.uniform(-1.5,1.5), np.random.uniform(-
1.5,1.5),1),scaling=10)
robot = RobotArm((-3,0,0.4),joint_forces=400)
robot.import_robot_arm('robot_arms/my_robot/my_robot.urdf')
robot.set_dynamic_conditions(1000,0.001)
robot.load_robot_arm()
```

Со следниот блок код се објаснува како работи краткиот алгоритам да го препознае објектот и да го фати. Прво извршниот елемент на роботската рака се поставува на центарот на масата и на највисоката позиција која може раката да ја постигне. Потоа се зголемува видното поле на камерата и се зима слика од извршниот елемент на раката каде е претставено на Слика 7. Ориентацијата на извршниот елемент се поместува за 180 по x оска, бидејќи камерата слика од z оската нагоре. Со помош на numpy се зима позицијата на пикселите каде вредноста е 1 (поставеното лего) и се врши аритметичка средина по двете оски за да се најде центарот на објектот во сликата. Следно се заокружува позицијата на објектот до најблискиот пиксел и со помошната вградената функција rgbd_2_world() се конвертира позицијата од пиксели и длабочината во таа точка во координати на симулацијата. Посакуваната позиција се поместува по z оска за 0.75 за точно да се постави фаќачот. Потоа со двете команди за поместување, извршниот елемент се поставува точно над објектот и со actuate_attachment() се затвора фаќачот преку код. На крај се подига раката на почетната позиција.

```
robot.move2point((0,0,2.5),(180,0,0))
camera = robot.capture_image(fov=80)
h,w = np.mean((np.where(camera.seg == 1)), axis=1)
h,w = round(h),round(w)
target_pos = camera.rgbd_2_world(w,h,camera.depth[h,w]) + [0,0,0.75]
robot.move2point((target_pos[0],target_pos[1],2.5),(180,0,0))
robot.move2point(target_pos,(180,0,0))
robot.actuate_attachment(joint_targets = robot.attachment_close_targets)
robot.move2point((0,0,2),(180,0,0))
```



Слика 7 Престава на сликите земени од симулацијата

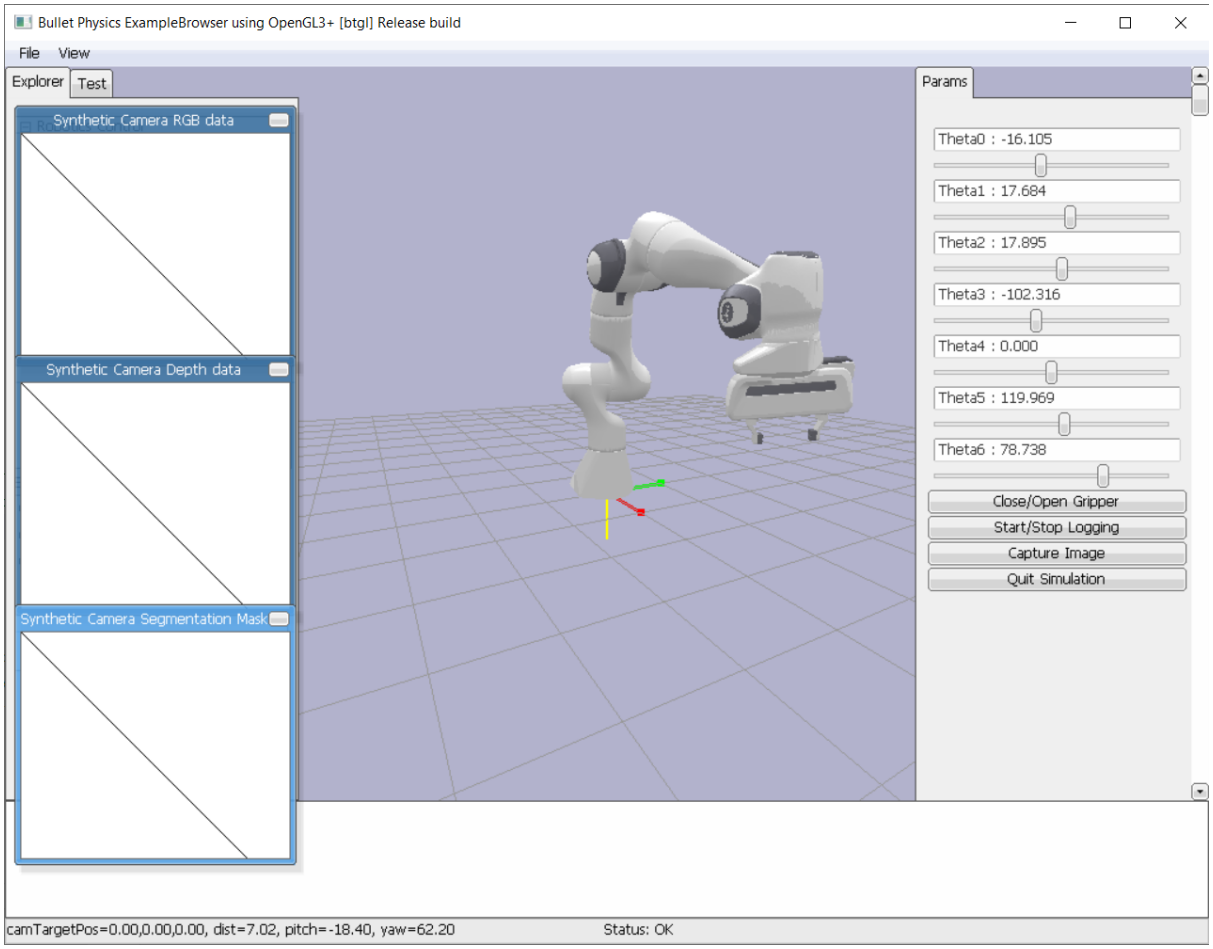
3.2 Популарни роботски раце

Со оваа јупитер тетратка ќе се разгледаат две примери од вистински роботски раце, како да се внесат во pybullet и да се покажат сите можности за нивно управување. Се полнат истите библиотеки и пакети како претходната тетратка.

3.2.1 Franka Emirka panda роботска рака

Прво е преставена раката Franka Emirka panda [11] управувана преку директна кинематика на Слика 8. Во овој случај се внесува преку import_foreign_robot() бидејќи е надворешна рака и дополнително може да се постави и скалирањето на раката. Исто така при внесување на надворешна рака треба да се променат позициите за отворање и затварање на фаќачот.

```
robot = RobotArm()
robot.import_foreign_robot('robot_arms/panda/panda.urdf')
robot.set_attachment_targets((0.2,0.2), (0,0))
robot.interact()
```

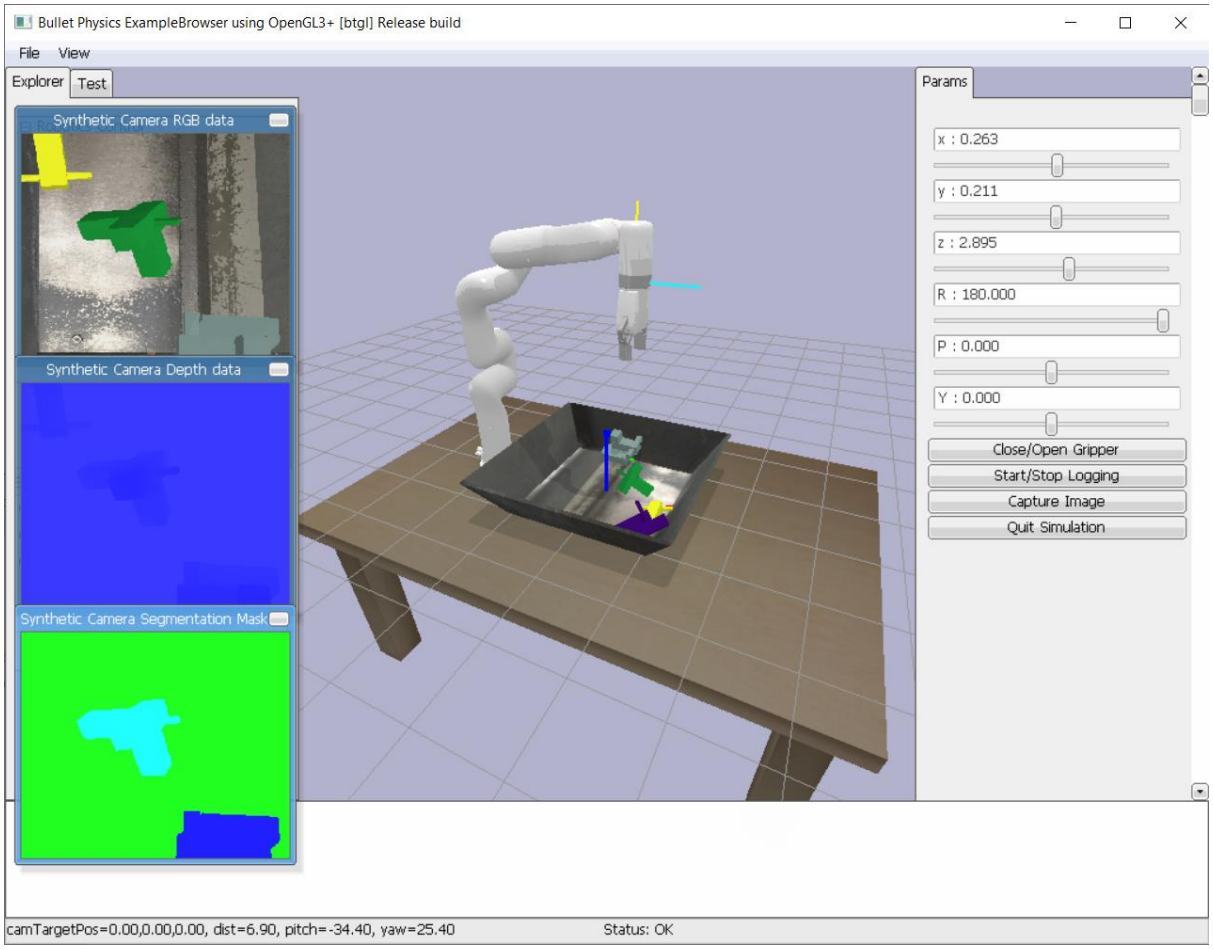


Слика 8 Управување на Franka Emirka panda со директна кинематика

3.2.2 xArm роботкса рака

Втората роботска рака е обезбедена од xArm [12] со која имаме повеќе конфигурации каде може да се видат во патеката robot_arms/xarm [10]. Прво се креира свет каде се полнат случајни објекти кои се наоѓаат во pybullet data. Се користи инверзна кинематика за управување на раката и дополнително се поставува и опсегот на лизгачите за позиција, во овој пример се поставува z оската од 0 до 5 за да не се удри во масата. Поставените промени на корисничкиот интерфејс и пример за сликање во симулацијата преку копче се преставени во Слика 9.

```
sim = PybulletSimulation()
sim.load_playground_2()
robot = RobotArm((-2,0,0))
robot.import_foreign_robot('robot_arms/xarm/xarm6_with_gripper.urdf')
robot.set_attachment_targets([0]*6, [0.6981]*6)
robot.interact('inverse',z_range=(0,5,2))
```



Слика 9 Управување на xArm со инверзна кинематика

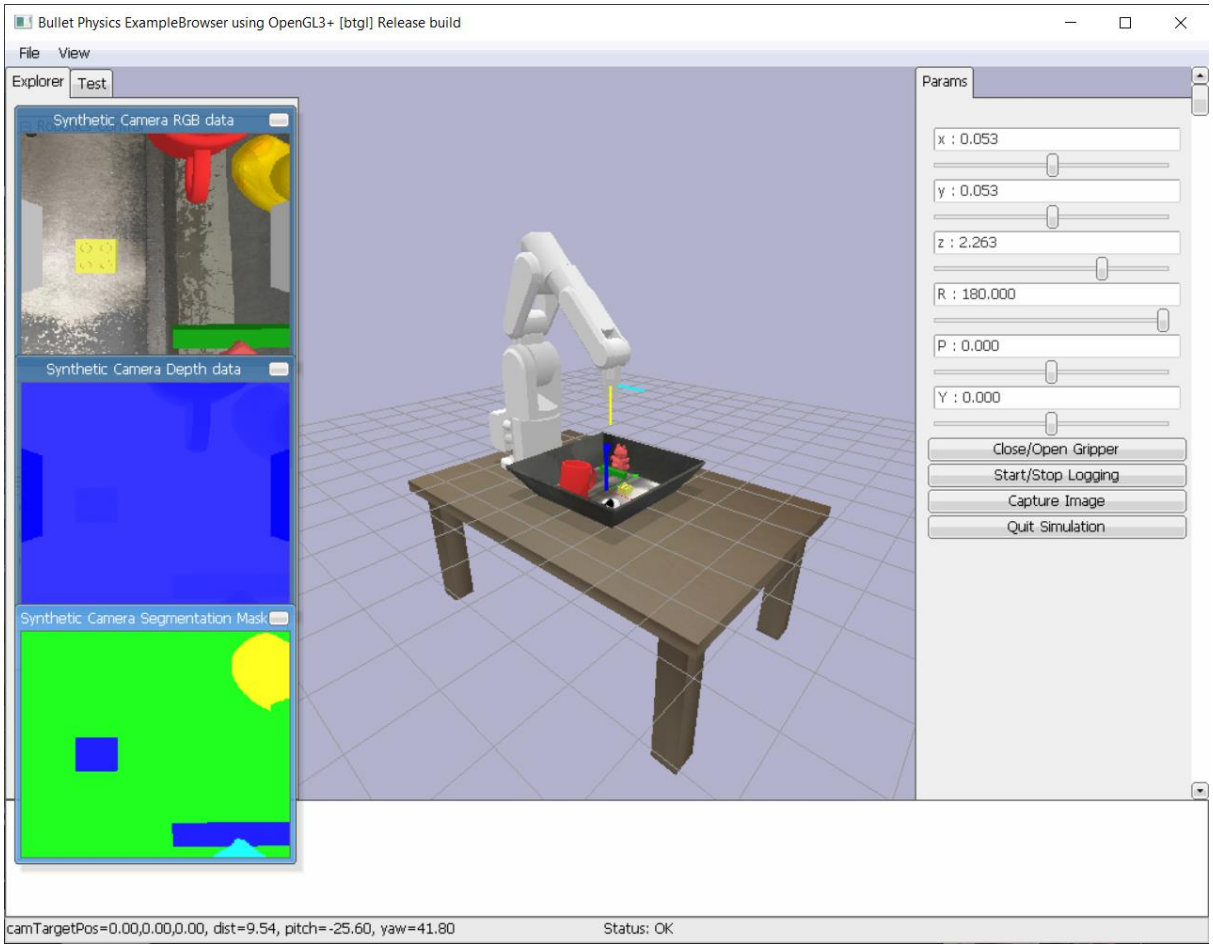
3.3 Mitsubishi роботска рака

Со оваа jupyter тетратка се симулира mitsubishi раката RV-2F-Q и се генерира програма од симулацијата која ќе се изврши на реалната рака. Креирани се две верзии на раката со и без фаќач, но засега правилно е имплементирана раката без фаќач бидејќи уште не е довршен хардверскиот дел.

3.3.1 Управување на mitsubishi роботската рака со фаќач

Креираме свет во кој се полни работната маса, подлога и сите објекти кои се обезбедени од pybullet. Дополнително се поставува и офсетот на камерата за точно да биде поставена на почетокот од фаќачот. Симулацискиот свет playground_1 и управувањето на раката може да се види во Слика 10.

```
sim = PybulletSimulation()
sim.load_playground_1()
robot = RobotArm((-2.2,0,0))
robot.camera_offset = (0,0,0.02)
robot.import_foreign_robot('robot_arms/mitsubishi/RV2FQG.urdf',8)
robot.interact('inverse')
```



Слика 10 Управување со mitsubishi роботска рака

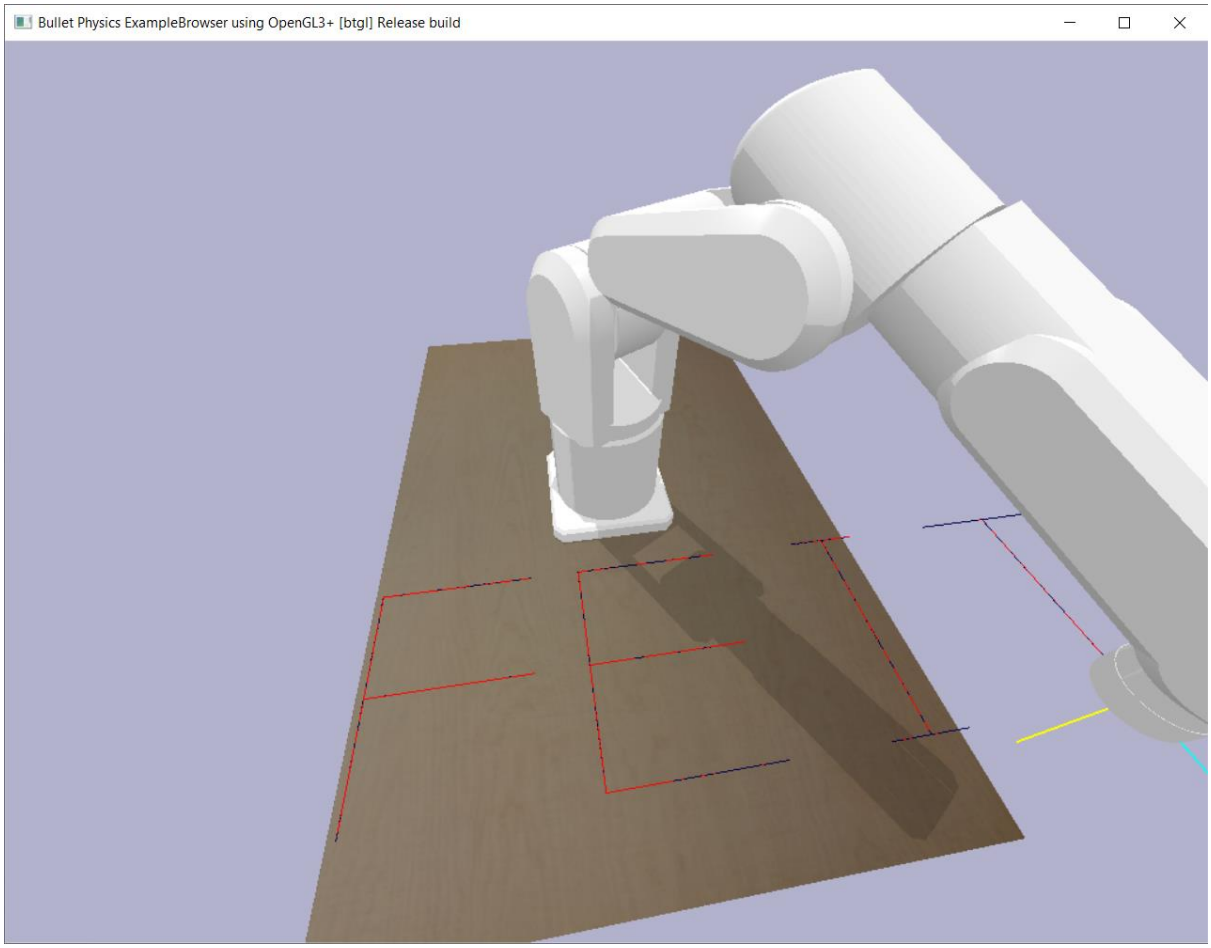
3.3.2 Поднесување на параметри на раката за управување преку команди

Креираме симулиран свет во кој точно се реплицира реалниот свет и не се користи динамика при управувањето бидејќи ќе се користи за креирање на програма. Секој чекор треба да биде прецизен и нема потреба да се чека раката да стигне до саканата позиција, тоа ќе се направи на физичката рака. Многу важно е да се постават и границите за до каде може извршниот елемент на раката да се придвижи, во овој случај да не удри во масата и самата рака. На крај се ресетираат зглобовите за да помогне при нумеричката инверзна кинематика бидејќи ако почне од нулта позиција, има тенденција да се приближи кон другото решение, но не може да конвергира поради границите на зглобовите. Опционално може да се постави и колку секунди линијата за траекторијата да биде прикажана.

```
sim = PybulletSimulation()
sim.load_table()
robot = RobotArm(use_dynamics=False)
robot.import_foreign_robot('robot_arms/mitsubishi/RV2FQ.urdf',8)
robot.set_position_limits(1.8,3.5,-3,3,0.3,4)
robot.load_robot_arm()
robot.reset_joints([0,0,2,0,0,0])
# robot.trajectory_lifetime = 30 # set the line life to 30 seconds
```

Следниот блок линии ги преставува сите опции како да се генерираат посакуваните точки. Најпросто движење е линеарното каде се поставува почетна и крајна точка и посакуваната ориентација во таа точка која исто така може да се интерполира. Исто така воведено е и кружно интерполирање помеѓу двете точки каде дополнително се поставуваат и четирите параметри. Првите две се а и б за како да изгледа елипсата, а другите две се бинарни вредности со кои првиот одбира кое решение да се одбере, а вториот од која старана да започне да црта. Исто така може да се да се одбере колку точки да се генерираат и да се ротира рамнината на кои се проектираат точките. Имаме дополнително и опција за започнување со запишување на зглобовите. Како апстракција на можните движења, воведена е функција за цртање текст кој резултат е преставен на Слика 11, но засега има мал број на букви што се вчитани.

```
robot.move((3,-2,3,180,0,0),(2,3,2,180,0,0))
robot.move((3.8,0,0,180,0,0),(4,0,0,180,0,0),
(0,-45,0),'circular',60,param=(1,1,1,0),log_move=True)
robot.write_text('FEIT',[2.3,-1.5,2],(180,0,0),log_text=False)
```



Слика 11 Цртање на текст со mitsubushi рака

Со последниот блок линии се проверуваат зачуваните записи со функцијата replay_logs() и со функцијата convert_logs_to_prg() се генерира програма која ќе се изврши на физичката рака. На крај симулацијата се исклучува преку командата quit_simulation().

```
robot.replay_logs('move_circular')
robot.convert_logs_to_prg('FEIT','feit')
robot.quit_simulation()
```

3.4 Кратки копчиња за управување со pybullet

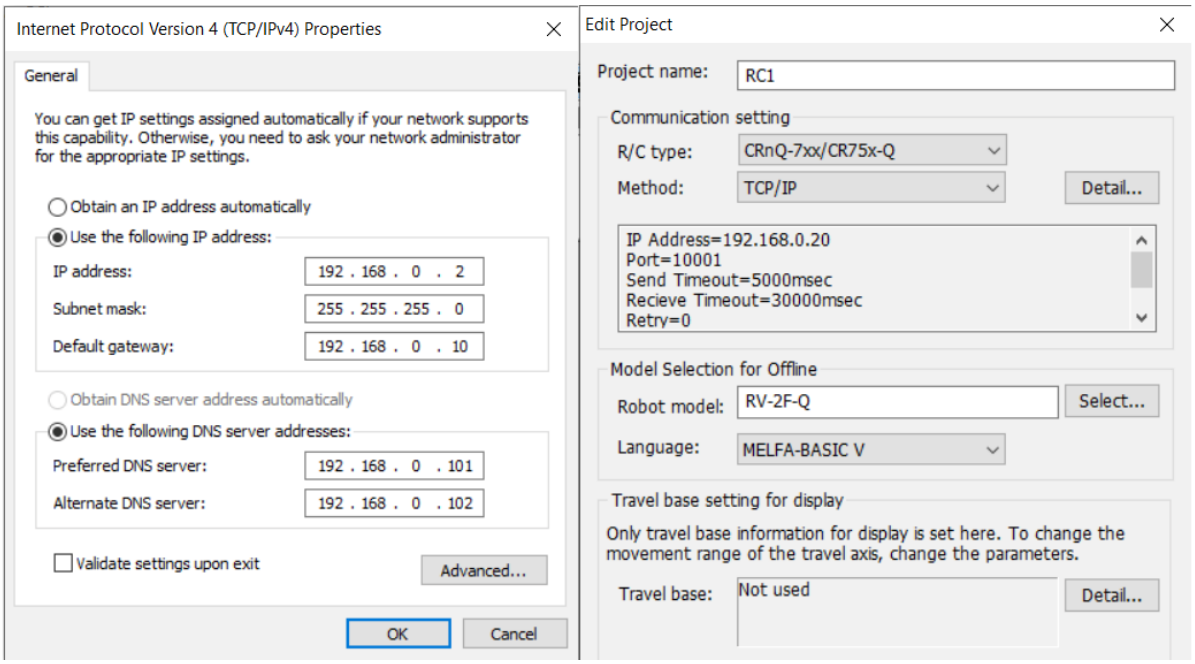
Кратенките на тастатурата се комбинации на копчиња кои овозможуваат брзо извршување на дејства на компјутерот. Во следната Табела 1 ќе се објаснат сите кратки копчиња за корисничкиот интерфејс на pybullet.

Акција	Операција	Инструкција
Отворете или затворете прозорци за пребарување, тестирање и параметри	g	Притиснете g за да ги префрлите сите менија
Паузирај ја симулацијата за физичка	i	Притиснете i за да ја паузирате симулацијата
Дампинг на профилот	p	Притиснете p за ги зачувате записите за користење на нишките
Вклучете ги светлата и сенките	s	Притиснете s за да ги вклучите/исклучите светлата и сенките
Вклучете ја визуелната геометрија	v	Притиснете v за да вклучите/исклучите визуелизацијата на геометриските лица
Вклучете ја жичаната рамка	w	Притиснете w за да го вклучите/исклучите режимот на рамка
Излезете од апликацијата	Ecs	Притиснете Esc за да излезете од симулаторот
Премести објект	click + drag	Кликнете на објект во рамките на симулацијата за да примените сила
Ротирај го приказот	Ctrl + drag	Кога го држите копчето Alt или Control, левото кликнување и влечење на глувчето ја ротира камерата
Преглед на превод	Ctrl + middle click	Кога го држите копчето Alt или Control, среден клик и влечење на глувчето ја поместува камерата
Зумирајте/ одзумирајте приказ	mouseweel	Користете го глувчето за да зумирајте и одзумирајте
Зумирајте/ одзумирајте приказ	Ctrl + right click	Кога го држите копчето Alt или Control, кликнете со десното копче на глувчето и повлечете ја камерата за зумирање и одзумирање
Вклучете го серверот за физика за зачувување слики	F1	Притиснете F1 за да го префрлите серверот за физика за фрлање слики (континуирано ги зачувува сликите од екранот на симулаторот)

Табела 1 Кратки копчиња за управување со pybullet

4 Конектирање на CR750-Q со RT Toolbox 2

Комуникацијата помеѓу компјутерот и управувачот CR750-Q се врши преку интернет кабел. Кабелот е поврзан со PLC интернет портата на CR750-Q и во интернет портата на компјутерот. Параметрите на управувачот на роботот за TCP/IP комуникација се веќе поставени, треба да се следат следните чекори за поставување на параметрите кај компјутерот. Прво во Windows треба да се отиде во Control Panel > Network and Internet > Network and Sharing Center > Change adapter settings > Select the Ethernet Properties > Choose the TCP/IPv4 properties и да се постават својствата на IPv4 како на левиот прозорец од Слика 12. Следно во програмата RT Toolbox 2 се креира нов проект и се копираат параметрите од десниот прозорец од Слика 12. Потоа од менито на RT Toolbox 2 се отвора Online, и како под опција се селектира пак Online со кој програмата воспоставува комуникација со управувачот. Се појавува нов прозорец кој не информира за статусот на врската и ако се е поднесено како што треба, статусот на линијата треба да покаже дека е конектирано и прозорецот да свети сина боја.



Слика 12 Параметри за конекција со управувачот на mitsubishi раката

5 Архитектура на софтверскиот пакет

Архитектурата во кодирањето се однесува на целокупниот дизајн и структура на софтверскиот систем. Ги опишува компонентите што го сочинуваат системот, односите помеѓу тие компоненти и начините на кои тие комуницираат. Добрата архитектура е од суштинско значење за создавање на скалабилен, одржуван и ефикасен проект. Тоа помага да се осигура дека системот е лесен за разбирање и работа со него, дури и кога расте и се развива со текот на времето. Ефективната архитектура, исто така, може да го олесни идентификувањето и поправањето грешки, како и додавањето нови функции и функционалности на системот. Во следните потточки ќе бидат објаснети комуникациите помеѓу класите и нивните функционалности.

5.1 Хиерархија помеѓу класите

Главната класа на проектот е RobotArm(), каде се извршуваат главните функционалности на проектот кој се поделени на:

- Функции за креирање на модуларна роботска рака
- Функции за управување на раката преку кориснички интерфејс
- Функции за управување на раката преку команди
- Модули за работксата рака во симулацијата

Комуникацијата со PybulletSimulation() е прилично мала бидејќи таа е застапена за креирање на светот околу роботот. Се повикува автоматски при иницијализација ако сакаме да креираме празен свет. DH2Urdf() се користи стриктно за конвертирање од DH конвенцијата при креирање на модуларниот робот во URDF датотека кој pybullet може да ја разбере. Посебниот пакет utils е превземен од отворениот проект по предметот роботика 1 [9] на факултетот за електротехника и информациски технологии во Скопје и е наменет за пресметување на DH матрицата каде формулите се превземени од книгата [1] и за пресметка на јакобијанот на раката. Дополнително се повикува и класата Camera() со која се слика во симулацијата.

5.2 Хиерархија на класата PybulletSimulation

Хиерархија на PybulletSimulation() е меѓусебно поврзана, каде што повеќето функции зависат помеѓу себе. При иницијализацијата се конфигурира гравитацијата, параметрите на главната камерата, брзината на симулацијата и патеката каде се сместени URDF датотеките. Има воведено повеќе основни функции каде се полни работна маса, послужавникот, сите објекти пронајдени во pybullet data и функција за полнење на случајни објекти. Како апстракција на основите функции има воведено две игралишта каде се повикуваат објектите заедно да се наполни симулацискиот свет со една функција.

5.3 Хиерархија на класата Camera

Класата Camera е апстракција на функцијата getCameraImage [13] со која се олеснува поставувањето на погледот и сите потребни параметри за камерата. При иницијализација се пресметуваат матриците за поглед и проекција на камерата со поставените параметри за широчина на видното поле, димензиите на сликата, филтер за опсег кој камерата може да го опфати. Исто така се пресметува и инверзната матрица за да се вратиме од пикселите во сликата во координати од симулираниот свет. Со главната функција shot() се слика симулацијата и се зачувува за понатамошна работа. Следнита функција rgbd_2_world() е за конвертирање од позицијата на пикелот од сликата во реалниот свет каде се користи инверзната матрица.

5.4 Хиерархија на класата DH2Urdf

Класата DH2Urdf() е наменета за конвертирање од DH параметри во URDF датотека која симулацијата може да ја прочита. При иницијализацијата се превземаат DH параметрите, поставените граници на зглобовите и додатокот ако е поставен. Има поставено четири основни функции кои генерираат соодветен текст според конвенцијата за запишување на URDF. Основната функција е write_link() со која се црта поврзаноста помеѓу зглобовите. Како под функции спагаат write_visual_shape() со кој се запишува визуелниот дел и write_inertial() со кој се внесуваат инерцијалните вредности на линкот кој засега не се користи. Со функцијата write_joint() се запишува блок кој ги спојува два линка. Главно се користи функцијата save_urdf() која ги спојува сите функции заедно и според димензиите на DH параметрите ги пресметува соодветните линкови и зглобови, кои пресметки се засновани од статијата [2]. Зглобот е преставен како темно сив краток

цилиндар, а линковите помеѓу нив се долги бели цилиндри кои ја преставуваат најкратката линија помеѓу нив. На крај генерираниот текст се зачувува во датотека на поставената патека.

5.5 Хиерархија за управување преку кориснички интерфејс

Управувањето преку интерактивниот кориснички интерфејс се врши со функцијата interact() каде имаме можност да користиме директна или инверзна кинематика каде дополнително може да се одбере дали да користи и ориентацијата на извршниот елемент при калкулациите. Редоследот на извршување е прво се користи класата DH2Urdf() за да се конвертира раката ако не е внесена и се полни во симулацијата на поставената позиција и ориентација. Потоа со помошните функции find_joint_ids() идентификуваат сите зглобови во раката, се додаваат фрејм линии по x и z оска на секој зглоб со add_joint_frame_lines() ако не е надворешна раката и се ресетира секој зглоб на почетната позиција. Зависно од тоа каква кинематика е одбрано се додаваат лизгачи за инверзна или директна кинематика со add_joint_sliders() или add_pose_sliders(). На крај се повикува функцијата step() каде се чита од лизгачи за посакуваната позиција и ориентација или позиција на секој зглоб и се управуваат соодветните зглобови со помош на функциите доделени од pybullet. При крај од функцијата имаме и дел со модули кои помагаат при управување, анализа и интеракција со роботската рака.

5.6 Хиерархија за управување преку команди

Управувањето преку код во класата RobotArm() е поделена на повеќе функции за подобра приспособливост. Основната функција е move2point() со која се врши најосновната работа да ја придвижи раката до посакуваната точка и ориентација. Се врши истото управување на зглобовите како погоре и дополнително имаме и дуален услов за запирање на циклусот. Ако се користи динамика се споредуваат сите зглобови дали грешката е поголема од поставената или се поминати одреден број на чекори во симулацијата бидејќи можно е инверзната кинематика да нема решение, а ако не користиме динамика се ресетираат зглобовите и се запира циклусот после првиот чекор. Како апстракција на основната функција е move() каде се врши линеарна или кружна интерполација. Дополнително се воведени и две функции write_text() и write_letter() со кои цртаат букви во кои се користи интерполација.

5.7 Хиерархија за креирање на модуларна роботска рака

Конвенцијата за креирање на нова модуларна рака во класата RobotArm() е превземена од fundamentals-of-robotics [9] каде се додадени дополнителни функционалности. За додавање на нов зглоб се наменети функциите add_revolute_joint() и add_prismatic_joint(), но исто така додаден е и фиксен зглоб add_fixed_joint(). Функциите дополнително имаат опција да се постават границите на зглобот, максималното забрзување и брзина. Функцијата add_subs() се користи ако се поставуваат дополнителни симболички променливи во зглобовите, бидејќи мора да имаат нумеричка вредност. Останатите функции за пресметување на DH матрицата и јакобијанот на раката се директно преземени со мали промени при додавање на фиксниот зглоб.

6 Документација за кодот на софтверскиот пакет

Пишувањето документација за код е важен дел од процесот на развој на софтвер. Тоа им помага на другите програмери (и самите себе) да се разбере како функционира кодот и користи кодот. Добрата документација треба да вклучува преглед на кодот на високо ниво, објаснувања за клучните концепти и алгоритми и јасни упатства за користење и менување на кодот. Во следните потточки ќе се објаснат сите класи во проектот со соодветните функции во детали.

6.1 DH2Urdf()

6.1.1 __init__(self, dh_params, constraints, attachment = None)

Со иницијализирањето се зачувуваат внесените параметри dh_params, constraints и attachment. Параметарот attachment е опционален бидејќи не секоја рака мора да има и додаток на неа. Дополнително има помошен речник со кој се мапираат зглобовите со идто дадено од pybullet. На крај се поставува xml текст на кој ќе се запишува раката.

```
self.joint_names = {0: 'revolute', 1: 'prismatic', 4: 'fixed'}
self.dh_params = dh_params
self.constraints = constraints
self.attachment = attachment
self.xml = ''
```

6.1.2 write_visual_shape(self, rpy, xyz, length, radius, material_name)

Функцијата е наменета за додавање на блок за визуелниот дел од роботот. Како влез прима ја ориентацијата и позицијата на визуелниот објект и се поставува должината и радиусот на цилиндарот. Дополнително се поставува и името на бојата на цилиндарот.

```
self.xml += "\t\t<visual>\n"
self.xml += "\t\t\t<origin rpy='{} {} {}' xyz='{} {} {}'/>\n".format(
    rpy[0], rpy[1], rpy[2], xyz[0], xyz[1], xyz[2], prec=5)
self.xml += "\t\t\t<geometry>\n"
self.xml += "\t\t\t\t<cylinder length='{}' radius='{}'/>\n".format(length,
radius)
self.xml += "\t\t\t</geometry>\n"
self.xml += "\t\t\t<material name='{}'/>\n".format(material_name)
self.xml += "\t\t</visual>\n"
```

6.1.3 write_inertial(self, rpy, xyz, mass, inertia_xxyyzz)

Оваа функција е наменета за запишување на инерцијалните вредности на линкот, каде се поставува ориентацијата и позицијата на центарот на маса на објектот, неговата маса и инерција по ixx, iyy и izz оските.

```
self.xml += "\t\t</inertial>\n"
self.xml += "\t\t\t<origin rpy='{} {} {}' xyz='{} {} {}'/>\n".format(
    rpy[0], rpy[1], rpy[2], xyz[0], xyz[1], xyz[2], prec=5)
self.xml += "\t\t<mass value='{}'/>\n".format(mass)
self.xml += "\t\t<inertia ixx='{}' ixy='0' ixz='0' iyy='{}' iyz='0' izz='{}'
/>\n".format(
    inertia_xxyyzz[0], inertia_xxyyzz[1], inertia_xxyyzz[2], prec=5)
self.xml += "\t\t</inertial>\n"
```

6.1.4 write_link(self, name, visual_shapes, save_visual = True)

Функцијата креира блок код со кој се формира линкот помеѓу два зглобови. Името на линкот треба да биде уникатно. Како влез прима листа од визуелни објекти каде мине низ секој објект и го запишува користејќи ја функцијата write_visual_shape(). Дополнително помеѓу блокот се додаваат и инерцијалните вредности, но за сега е коментирано бидејќи тој дел не е довршен. Секој линк има дополнително опција да се додаде и колизија, но засега со оваа конструкција за модуларниот робот не е возможно бидејќи има колизии помеѓу зглобот и линкот.

```
self.xml += "\t<link name='{}'>\n".format(name)
#self.write_inertial(self.xml)
if save_visual:
    for v in visual_shapes:
        self.write_visual_shape(v[0], v[1], v[2], v[3], v[4])
self.xml += "\t</link>\n"
```

6.1.5 write_joint(self, name, joint_type, parent, child, rpy, xyz, constraints = None)

Со овој шаблон се додава зглобот кој ги конектира двата линка. Како главни влезни параметри се името кое треба да биде уникатно и типот на зглоб кој може да биде транслаторен, ротациски или фиксен. Дополнително треба да се референцира кој е претходник и следбеник на зглобот и да се постави ориентацијата и позицијата на зглобот. Како за крај опционално се додаваат и границите на зглобот ако не е фиксен.

```
self.xml += "\t<joint name='{}' type='{}'>\n".format(name,
self.joint_names[int(joint_type)])
self.xml += "\t\t<parent link='{}'/>\n".format(parent)
self.xml += "\t\t<child link='{}'/>\n".format(child)
self.xml += "\t\t<axis xyz='0 0 1'/>\n"
self.xml += "\t\t<origin rpy='{} {} {}' xyz='{} {} {}'/>\n".format(
    rpy[0], rpy[1], rpy[2], xyz[0], xyz[1], xyz[2], prec=5)
if joint_type != 4: #fixed
    self.xml += "\t\t<limit effort='{}' lower='{}' upper='{}'
velocity='{}'/>\n".format(
        constraints[0], constraints[1], constraints[2], constraints[3],
prec=5)
self.xml += "\t</joint>\n"
```

6.1.6 save_urdf(self, file_path)

За крај ја имаме главната функција каде се користат сите останати и се конвертираат DH параметрите во URDF фајл кој се зачувува на дадената патека. Со првиот блок код се креира коментар каде се запишуваат сите вредности на DH параметрите и границите во табела кои служи за брз преглед како функционира роботската рака и понатаму се користат при внесување на раката во симулацијата. Се користи дополнителна листа за имињата на секоја колона од табелата и се креира формат за печатење сличен на листата на имиња. Следно се користи циклус кој мине низ сите DH параметри со соодветните граници на зглобовите. Сите броеви кои се реални се заокружуваат до трета децимала за подобар изглед на табелата.

```
self.xml = "<!-- DH Parameters and constraints\n"
names_list =
['Name','theta','d','a','alpha','effort','lower','upper','velocity','visual']
row_format = "{!s:>10s}" * (len(names_list))
```

```
self.xml += row_format.format(*names_list)+str('\n')
for DH_param, constraint in zip(self.dh_params, self.constraints):
    a = [self.joint_names[int(DH_param[0])]] + [round(float(i),4) for i in
DH_param[1:]] + [round(i,4) for i in constraint]
    self.xml += row_format.format(*a)+str('\n')
self.xml += "-->\n"
```

Дополнително на DH параметрите додаваме и фиксен зглоб кој помага при визуелниот дел и при пресметка на нумеричката инверзна кинематика. Се креира главниот таг robot и две бои кои се референцирани со имиња. Следно имаме циклус кој минува низ секој ред од DH параметрите.

```
self.dh_params.append((4,0,0,0,0))
self.constraints.append([True])
self.xml += "<robot name='robot_arm'>\n"
self.xml += "\t<material name='grey'>\n\t\t<color rgba='0.6 0.6 0.6
1'/>\n\t</material>\n"
self.xml += "\t<material name='white'>\n\t\t<color rgba='1 1 1
1'/>\n\t</material>\n"
```

При првиот чекор започнуваме со нулти вредности за поместувањата по x,z,row,yaw и не поставуваме зглоб помеѓу визуелниот дел на зглобот и линкот. При секој чекор со втората линија поставуваме линк кој преставува зглобот користејќи write_link().

```
y,z,x,r = (0,0,0,0) if i==0 else self.dh_params[i-1][1:] # transforms [i]
visual_shapes = [[(r,0,y), (x,0,z), 0.7, 0.43, 'grey']]
self.write_link(f'a{i}', visual_shapes, self.constraints[i][-1])
if (i != 0):
    self.write_joint(f'fix_a{i}_to_l{i-1}', 4, f'l{i-1}', f'a{i}', (0,0,0),
(0,0,0))
```

Со овој блок код се пресметува позицијата и ориентацијата на визуелниот дел од линкот кој ги поврзува двата визуелни зглобови. Првите неколку линии се наменети за пресметување на векторот на позиција од наредните DH параметри, кој се нормализира и се дели со два за да се најде центарот на векторот.

```
origins_vector = np.array([self.dh_params[i][3], 0,
                   self.dh_params[i][2]],dtype = float)
origins_vector_norm = np.linalg.norm(origins_vector)
cyor = origins_vector/2
rpy = (0, 0, 0)
```

Ако нормалниот вектор е различен од нула (имаме поместувања по x и z) следат одредени пресметки каде го пресметуваме аголот од нултиот вектор кон векторот генериран од DH параметрите. На крај ротациите по оските се добиваат со векторско множење на аглите со оските. Откако ќе се пресметаат позицијата и ориентацијата на линкот се гради визуелниот дел и се додава за линкот, додатно ако моменталниот зглоб е транслациски се додава и уште еден визуелен објект кој мине низ транслацискиот зглоб. Тој е поврзан со претходниот линк и има должина колку максималното издолжување на зглобот.

```
if (origins_vector_norm != 0.0):
    origins_vector_unit = origins_vector/origins_vector_norm
    axis = np.cross(origins_vector, np.array([0, 0, -1]))
    axis_norm = np.linalg.norm(axis)
    if (axis_norm != 0.0):
        axis = axis/np.linalg.norm(axis)
```

```
    angle = np.arccos(origins_vector_unit @ np.array([0, 0, 1]))
    rpy = angle*axis
visual_shapes = [[(rpy[0], rpy[1],
rpy[2]),(cyor[0],cyor[1],cyor[2]),origins_vector_norm,0.3,'white']]
if (self.dh_params[i][0] == 1):
    visual_shapes.append([(0,0,0), (0,0,-(self.constraints[i][2]/2-0.275)),
self.constraints[i][2], 0.3, 'white'])
```

Последно се запишува пресметаниот линк и се додава зглоб со поставените граници кој ги поврзува линкот и претходниот визуелен линк за зглобот.

```
self.write_link(f'l{i}',visual_shapes, self.constraints[i][-1])
self.write_joint(f'move_l{i}_from_a{i}',self.dh_params[i][0],f'a{i}',f'l{i}',
(r,0,y),(x,0,z),self.constraints[i])
```

Ако дополнително има поставено и додаток на крајот од роботската рака, се креира нов зглоб кој го поврзува извршниот елемент со додатокот, каде позицијата и ориентацијата се дадени. Според поставеното име на додатокот се отвара соодветната датотека и се копира се што има во неа. Оттука доаѓа зошто треба граничниот зглоб да биде наречен `attachment_joint` за правилно да се разделат зглобовите. Со последните линии код се затвора главниот таг и се зачувува текстот на поставената патека.

```
if self.attachment:
    name, orn, pos = self.attachment
    self.write_joint(f'attachment_joint',4,f'l{i}',f'base', orn, pos)
    file = open(f"attachments/{name}/{name}.urdf",'r')
    for line in file.readlines()[1:-1]:
        self.xml+= line
    file.close()
self.xml += "</robot>\n"
file = open(file_path, "w")
file.write(self.xml)
file.close()
```

6.2 Camera()

6.2.1 __init__(self,cam_target, distance, rpy, near, far, size, fov)

Со иницијализирањето се креира објект на камерата каде се зачувуваат внесените параметри како димензиите на сликата на камерата, опсег за блиската и далечната рамнина на пресек ,ориентацијата на камерата и видното поле.

```
self.width, self.height = size
self.near, self.far = near, far
row,pitch,yaw = rpy
self.fov = fov
```

Се пресметуваат матриците за преглед и проекција на камерата користејќи ги дадените агли, растојанија и димензии на приказот со дадените функции од pybullet. Матриците се користат за дефинирање на перспективата на камерата, а матрицата tran_pix_world е инверзна од производот на матриците за проекција и преглед, што се користи за претворање на координатите од пиксели во координати на симулацијата.

```
aspect = self.width / self.height
self.view_matrix =
p.computeViewMatrixFromYawPitchRoll(cam_target,distance,yaw,pitch,row,1)
self.projection_matrix = p.computeProjectionMatrixFOV(self.fov, aspect,
self.near, self.far)
_view_matrix = np.array(self.view_matrix).reshape((4, 4), order='F')
_projection_matrix = np.array(self.projection_matrix).reshape((4, 4),
order='F')
self.tran_pix_world = np.linalg.inv(_projection_matrix @ _view_matrix)
```

6.2.2 shot(self)

Функцијата е апстракција на функцијата getCameraImage [13] од pybullet која зима слика од симулацијата со поставени димензиите на сликата и пресметаните матрици за преглед и проекција. На крај добиената слика во боја, слика за длабочина и слика за сегментација на објектите се зачувуваат за понатамошно користење.

```
_, _, rgb, depth, seg = p.getCameraImage(self.width,
self.height,self.view_matrix, self.projection_matrix)
self.rgb, self.depth, self.seg = rgb, depth, seg
```

6.2.3 rgbd_2_world(self, w, h, d)

Со оваа функција се конвертира позицијата на пикселот во сликата и неговото растојание од камерата во координати на симулацијата користејќи ја инверзната матрица. Во функцијата rgbd_2_world_batch() ја имаме истата функционалност, но сега на влез е целата слика за длабочина и во овој случај се конвертираат сите точки.

```
x = (2 * w - self.width) / self.width
y = -(2 * h - self.height) / self.height
z = 2 * d - 1
pix_pos = np.array((x, y, z, 1))
position = self.tran_pix_world @ pix_pos
position /= position[3]
return position
```

6.3 PybulletSimulation()

6.3.1 __init__(self, connection_mode, fps, gravity, cam_param, cam_target)

При иницијализација се зачувуваат сите поставени параметри и се креира едена помошна променлива во кој е складирано знаме за користење на кеширани податоци во pybullet. Следно се конектираме со графичката симулација од pybullet и ако е успешно се конфигурира симулацијата со функцијата configure().

```
self.connection_mode = connection_mode
self.time_step= 1/fps
self.gravity = gravity
self.cam_param = cam_param
self.cam_target = cam_target
self.flags = p.URDF_ENABLE_CACHED_GRAPHICS_SHAPES
if p.connect(self.connection_mode) != -1:
    self.configure()
```

6.3.2 configure(self)

При конфигурацијата на симулацијата се поставува времето помеѓу секој симулиран чекор и гравитацијата. Дополнително со поставуваат и патеките за пребарување на URDF датотеките, се поставува моменталната патека и патека кон податоците обезбедени од pybullet. На крај се изменува позицијата на светлото и се ресетира ориентацијата и позицијата на камерата за дебагирање.

```
p.setTimeStep(self.time_step)
p.setGravity(self.gravity[0],self.gravity[1],self.gravity[2])
p.setAdditionalSearchPath(os.getcwd())
p.setAdditionalSearchPath(pd.getDataPath())
p.configureDebugVisualizer(lightPosition= (0,0,5))
p.resetDebugVisualizerCamera(self.cam_param[0], self.cam_param[1],
self.cam_param[2], self.cam_target)
```

6.3.3 load_any_object()

Останатите функции за полнење во класата се едноставни и преставуваат апстракција на функцијата loadURDF[13]. Имаме поставено функции за основни објекти како load_table(), load_tray(), load_lego() и функција load_common_objects() каде се полнат останатите објекти обезбедени од pybullet. Исто така на крај од класата има додатно и функции load_playground_x кои ги инкорпорираат сите основни функции погоре и се користат за полесно креирање на светот околу роботската рака.

6.3.4 load_random_objects(self, count_objects, position, orientation, scaling)

Имаме посебна функција каде се полнат 1000 различни објекти кои се дадени од pybullet кои имаат различна форма. Со помош на numpy креираме низа од случајни броеви од еден до илјада со големина од параметарот count_objects. Потоа минеме низ секој број и го полниме објектот на поставената позиција со случаен офсет.

```
for num in np.random.randint(1000, size=count_objects):
    rand_position = np.array(position) + np.random.uniform(-1,1,3)
    p.loadURDF(f"random_urdfs/{num:03}/{num:03}.urdf", rand_position,
orientation, flags=self.flags,globalScaling=scaling)
```

6.4 RobotArm()

6.4.1 __init__(self, params)

Иницијализацијата за роботската рака е прилично голема, каде се иницијализираат сите помошни променливи, и зачувуваат поставените параметри. Иницијалните параметри за роботската рака се:

- position, orientation: се поставува позицијата и ориентацијата на раката,
- name: името на раката кое понатаму служи за креирање на URDF фајлот,
- ik_function: се поставува функција во која се пресметува инверзна кинематика,
- fps, scaling: брзината на симулацијата и скалирање на роботксата рака,
- joint_forces: може да се постави за секој зглоб со една бројка или листа од сили за секој зглоб поединечно,
- use_dynamics: се одбира дали зглобовите на роботската рака да се управуваат динамички или со ресетирање на нивната позиција,
- use_draw_trajectory: се користи како знаме дали да се црта траекторијата на извршниот елемент на роботската рака,

- use_display_pos_and_orn: се користи како знаме дали да се печати моменталната позиција и ориентација на извршниот елемент на раката.

На крај проверуваме дали имаме воспоставено конекција со симулацијата на pybullet и ако нема креираме нов празен свет и го превземаме времето на чекор од симулацијата.

```
if not p.isConnected():
    sim = PybulletSimulation(fps=fps)
    self.time_step = sim.time_step
else:
    self.time_step = 1/fps
```

6.4.2 add_pose_sliders(self, x_range, y_range, z_range)

Со функцијата се креираат лизгачи за одбирање на позицијата и ориентацијата на извршниот елемент на раката. Лизгачите за позиција треба да се постават соодветните минимуми, максимуми и моментални позиции, а ориентацијата оди од -180 до 180 степени и се додаваат ако користиме ориентација при инверзната кинематика. Лизгач се додава со функцијата addUserDebugParameter [13] и се појавува на десната страна од корисничкиот интерфејс на симулацијата.

```
self.pose_sliders = []
self.pose_sliders.append(p.addUserDebugParameter(f"x", x_range[0], x_range[1],
x_range[2]))
self.pose_sliders.append(p.addUserDebugParameter(f"y", y_range[0], y_range[1],
y_range[2]))
self.pose_sliders.append(p.addUserDebugParameter(f"z", z_range[0], z_range[1],
z_range[2]))
if self.use_orientation_ik:
    self.pose_sliders.append(p.addUserDebugParameter(f"R", -180, 180, 0))
    self.pose_sliders.append(p.addUserDebugParameter(f"P", -180, 180, 0))
    self.pose_sliders.append(p.addUserDebugParameter(f"Y", -180, 180, 0))
```

6.4.3 add_joint_sliders(self)

Со функцијата се креира лизгач за секој зглоб на раката. Прво се проверува дали зглобот е транслациски или ротациски:

- Ако е транслациски се креира лизгач каде опсегот се границите на зглобот и почетна позиција е долната граница, дополнително се додава и знаме во листата за да се разликуваат зглобовите.
- Ако е ротациски се креира лизгач со истите опсези, но во овој случај се конвертираат од радијани во степени и моменталната позиција е нула.

```
self.joint_sliders = []
for i in self.joint_ids:
    joint_info = p.getJointInfo(self.robot_arm, i)
    if joint_info[2] == p.JOINT_PRISMATIC:
        self.joint_sliders.append((False, p.addUserDebugParameter(f"D{i}",
joint_info[8], joint_info[9], joint_info[8])))
    if joint_info[2] == p.JOINT_REVOLUTE:
        self.joint_sliders.append((True, p.addUserDebugParameter(f"Theta{i}",
np.rad2deg(joint_info[8]), np.rad2deg(joint_info[9]), 0)))
```

6.4.4 find_joint_ids(self)

Со оваа функција се мине низ секој зглоб на раката и се зачувува идто во листа. Дополнително се делат зглобовите на две листи кога ќе се наиде на зглоб наречен attachment_joint кој преставува границата помеѓу раката и додатокот. Зглобот се додава во листата ако е транслациски или ротациски. Ако немаме додаток на раката помошниот последен зглоб е и последниот на раката, инаку се зима зглобот пред граничниот.

```
self.joint_ids = []
add_attachment_joints = False
add_constraints_if_foreign = False
for i in range(p.getNumJoints(self.robot_arm)):
    joint_info = p.getJointInfo(self.robot_arm, i)
    # start adding attachment joints if joint name is attachment_joint
    if joint_info[1] == b'attachment_joint' or add_attachment_joints:
        if add_attachment_joints == False:
            self.last_joint_id = i-1
        add_attachment_joints = True
        if (joint_info[2] in [p.JOINT_PRISMATIC, p.JOINT_REVOLUTE]):
            self.attachment_joint_ids.append(i)
    else:
        if (joint_info[2] in [p.JOINT_PRISMATIC, p.JOINT_REVOLUTE]):
            self.joint_ids.append(i)
```

Имаме специјален случај ако раката не е направена од оваа класа, дополнително се зачувуваат границите на зглобот во листата constraints.

```
            if self.is_foreign and (len(self.constraints) == 0 or
add_constraints_if_foreign):
                add_constraints_if_foreign = True
                self.constraints.append([joint_info[10], joint_info[8],
                        joint_info[9], joint_info[11], True])
if add_attachment_joints is False:
    self.last_joint_id = i
```

6.4.5 reset_joints(self, joint_states=None)

Се користи за ресетирање на секој зглоб на раката и има дополнително и аргумент каде може да се ресетираат на посакуваните позиции. Ако зглобот е призматичен се поставува зглобот на долната граница, а ако е ротациски се поставува на нула.

```
for i, joint_id in enumerate(self.joint_ids):
    joint_info = p.getJointInfo(self.robot_arm, joint_id)
    if joint_states:
        p.resetJointState(self.robot_arm, joint_id, joint_states[i])
    else:
        p.resetJointState(self.robot_arm, joint_id,
            joint_info[8] if joint_info[2] == p.JOINT_PRISMATIC else 0)
```

6.4.6 load_robot_arm(self)

Главна функција со која се внесува роботксата рака во симулацијата. Ако роботската рака не е внесена преку датотека, се конвертираат DH параметрите во URDF користејќи ја класата DH2Urdf. Ако патеката не постои каде сакаме да ја зачуваме раката се креира со истото име на раката и на креираната патека се зачувува конвертираната рака.

```
if self.is_imported is False:
    subs = self.subs_joints + self.subs_additional
    DH_params = sp.Matrix(self.links).subs(subs).evalf()
    dh2urdf = DH2Urdf(DH_params.tolist(),self.constraints, self.attachment)
    if not os.path.exists(f'{self.file_head}/'):
        os.mkdir(f'{self.file_head}/')
    dh2urdf.save_urdf(f'{self.file_head}/{self.name}.urdf')
```

Се внесува роботксата рака со функцијата loadURDF на поставената позиција и ориентација. Дополнително се одбира основата на раката да се фиксирана и се поставува скалирањето на раката, имаме и посебно знаме за да се одобрат кешираните графички објекти за побрзо да се полни раката.

```
self.robot_arm = p.loadURDF(f'{self.file_head}/{self.name}.urdf',
self.position, self.orientation, useFixedBase=True, flags =
p.URDF_ENABLE_CACHED_GRAPHICS_SHAPES, globalScaling=self.scaling)
```

Откако се напони раката се пронајдуваат зглобовите, се додаваат фрејм линии на нив и се ресетираат на почетните позиции. Дополнително се конвертира променливата joint_forces во листа од сили за зглобовите ако е поставено само број како параметар. На крај ако имаме додаток на раката се поставува фаќачот да биде отворен.

```
self.find_joint_ids()
self.add_joint_frame_lines()
self.reset_joints()
self.joint_forces = self.joint_forces if type(self.joint_forces) in
(list,tuple) else [self.joint_forces]*len(self.joint_ids)
if self.attachment_joint_ids:
    self.actuate_attachment(joint_targets = self.attachment_open_targets)
```

6.4.7 interact(self, kinematics, use_orientation_ik, x_range, y_range, z_range)

Една од главните функции каде ни дава можност да ја управуваме раката преку интерактивниот кориснички интерфејс. Како влезни параметри имаме да обереме типот на кинематика и дали да се користи ориентацијата при калкулација на инверзна кинематика. Дополнително можат да се постават и опсезите на лизгачите за позиција каде по стандардно е од -5 до 5 и моментална позиција од еден. Следно се внесува раката во симулацијата и се одбира кој лизгачи да се додадат зависно од каква кинематика е одберена. На крај се повикува функцијата step() со која минеме низ симулацијата.

```
self.kinematics = kinematics
self.use_orientation_ik = use_orientation_ik
self.load_robot_arm()
if self.kinematics == 'forward':
    self.add_joint_sliders()
elif self.kinematics == 'inverse':
    self.add_pose_sliders(x_range, y_range, z_range)
self.step()
```

6.4.8 step(self)

Функцијата е во бесконечен циклус се додека имаме конекција со симулацијата на pybullet. Со функцијата stepSimulation, pybullet ги врши сите пресметки за следниот чекор во симулацијата. Зависно од каква кинематика се користи, се читаат вредностите на лизгачите со readUserDebugParameter [13] и се зачувуваат во joint_targets ако користиме директна кинематика или во position и orientation за инверзна кинематика.

Дополнително за ориентацијата се конвертира вредноста на лизгачот од степени во радијани и трите алги се конвертираат во кватернион кој се користи за нумеричката инверзна кинематика.

```
while (p.isConnected()):
    st = time.time()
    p.stepSimulation()
    if self.kinematics == 'forward':
        position = None # read the sliders for joint desired joints position
        joint_targets = [(np.deg2rad(p.readUserDebugParameter(parameter)) if
                          revolute else p.readUserDebugParameter(parameter))
                          for revolute, parameter in self.joint_sliders]
    elif self.kinematics == 'inverse':
        position = (p.readUserDebugParameter(self.pose_sliders[0]),
                    p.readUserDebugParameter(self.pose_sliders[1]),
                    p.readUserDebugParameter(self.pose_sliders[2]))
        if self.use_orientation_ik:
            orientation = p.getQuaternionFromEuler([
      np.deg2rad(p.readUserDebugParameter(self.pose_sliders[3])),
      np.deg2rad(p.readUserDebugParameter(self.pose_sliders[4])),
      np.deg2rad(p.readUserDebugParameter(self.pose_sliders[5]))])
```

Ако имаме функција за пресметка на инверзната кинематика се користи таа, а ако немаме се користи функцијата calculateInverseKinematics [13] со која се користи нумеричка апроксимација. Дополнително при пресметката на инверзната кинематика се лимитираат згобовите и саканата позиција.

```
if self.ik_function:
    joint_targets = self.ik_function(self.limit_pos_target(position),
                   orientation if self.use_orientation_ik else None)
else:
    joint_targets = p.calculateInverseKinematics(self.robot_arm,
self.last_joint_id, self.limit_pos_target(position), orientation if
self.use_orientation_ik else None, maxNumIterations=5)[:len(self.joint_ids)]
    joint_targets = self.limit_joint_targets(joint_targets)
```

Следно се управуваат зглобовите на раката зависно од какво управување е одбрано. Ако користиме динамика се користи функцијата setJointMotorControlArray [13] каде се користи управување на позицијата и се поставуваат идто на зглобовите, посакуваните позиции и максималната сила за секој зглоб. Ако не користиме динамика, минеме низ секој зглоб и го ресетираме на посакуваната позиција.

```
if self.use_dynamics:
    p.setJointMotorControlArray(self.robot_arm, self.joint_ids,
p.POSITION_CONTROL, joint_targets, forces=self.joint_forces)
else:
    for i, joint_id in enumerate(self.joint_ids):
        p.resetJointState(self.robot_arm, joint_id, joint_targets[i])
```

На крај се повикуваат сите модули на симулацијата кој помагаат при управување и мониторирање на раката. Поставуваме копчиња за отворање и затварање на фаќачот, запишување на зглобовите, земање на слика од извршниот елемент, и стопирање на симулацијата. Исто така се црта траекторијата на извршниот елемент на раката и ја печатиме моменталната позиција и ориентација ако се одобрени. На крајот од циклусот

поставуваме динамичко спиење на програмата, каде мериме колку време е поминато извршувајќи го кодот и се одзема од поставеното време за спиење.

```
if self.attachment_joint_ids:
    self.actuate_attachment()
self.state_logging(self.kinematics)
link_state = p.getLinkState(self.robot_arm, self.last_joint_id)
self.capture_image(link_state, self.camera_offset, use_gui=True)
if self.use_draw_trajectory:
    self.draw_trajectory(link_state[4], position, self.trajectory_lifetime)
if self.use_display_pos_and_orn:
    self.display_pos_and_orn(link_state[4], link_state[5], self.last_joint_id)
if self.quit_simulation(use_gui=True):
    break
 ex_time = self.time_step - (time.time() - st)
 time.sleep(ex_time if ex_time > 0 else 0)
```

6.4.9 display_pos_and_orn(self, position, orientation, link_id)

Во оваа функција се користи addUserDebugText [13] од pybullet за печатење на позицијата и ориентацијата во симулацијата до поставениот линк. Прво се разгледува позицијата и конвертира ориентацијата од кватернион во ојлер и следно во степени. При прво повикување на функцијата се врши иницијализација на текстот каде се креира објектот. После секое повикување се користи дополнителниот аргумент replaceItemUniqueId со кој само се заменува текстот во објектот и немаме пребришување и креирање на нов објект за прикажување на новата вредност.

```
x, y, z = position
R, P, Y = np.rad2deg(p.getEulerFromQuaternion(orientation))
if self.pose_text is None:
    self.pose_text = p.addUserDebugText(f"(x:{x:.2f} y:{y:.2f} z:{z:.2f})
(R:{R:.1f} P:{P:.1f} Y:{Y:.1f})", [0.5, 0, 0.5],textColorRGB=[0, 0, 0],
textSize=1, parentObjectUniqueId=self.robot_arm, parentLinkIndex=link_id)
p.addUserDebugText(f"(x:{x:.2f} y:{y:.2f} z:{z:.2f}) (R:{R:.1f} P:{P:.1f}
Y:{Y:.1f})", [0.5, 0, 0.5], textColorRGB=[0, 0, 0],
textSize=1,parentObjectUniqueId=self.robot_arm, parentLinkIndex=link_id,
replaceItemUniqueId=self.pose_text)
```

6.4.10 draw_trajectory(self, current, target=None, life_time=15, line_index=0)

Функцијата ја црта моменталната и посакуваната траекторија со додавање на линија помеѓу моменталната и претходната точка. Содржи три листи во кои се чуваат претходните точки во листите prev_pose_1 и prev_pose_2 и листа has_prev_pose каде се зачувува дали имаме претходна точка и исто така се користи за ресетирање на цртањето. Опционално може да се одбере идексот на линијата, бидејќи може да се цртаат многу линии истовремено со истата функција.

```
if (self.has_prev_pose[line_index] is True):
    if target:
        p.addUserDebugLine(self.prev_pose_1[line_index],
                           target, [0, 0, 0.3], 1.3, life_time)
    p.addUserDebugLine(self.prev_pose_2[line_index],
                           current, [1, 0, 0], 1.3, life_time)
self.prev_pose_1[line_index] = target
```

```
self.prev_pose_2[line_index] = current
self.has_prev_pose[line_index] = True
```

6.4.11 quit_simulation(self, use_gui = False)

Кратка функција со која се исклучува симулацијата, каде исто така имаме опционален аргумент за дали да се изгаси преку копче. Ако користиме кориснички интерфејс при првиот повик се додава копче во симулацијата. Следно условот за запирање на симулацијата е ако не користиме кориснички интерфејс или ако е притиснато копчето еднаш и ако се е успешно функцијата враќа потврден одговор.

```
if use_gui and self.quit_button is None:
    self.quit_button = p.addUserDebugParameter("Quit Simulation",1,0,1)
else:
    if not use_gui or p.readUserDebugParameter(self.quit_button) %2 == 0:
        p.disconnect()
        return True
    else:
        return False
```

6.4.12 actuate_attachment(self, joint_ids=None, joint_targets=None)

Функцијата е направена да биде модуларна, при додавање на нов додаток ќе треба да се смени функционалноста. Моментално направена е логиката за управување фаќач. Како влез се поставуваат кои зглобови сакаме да ги управуваме и нивните позиции. Ако не се постават зглобовите се користат сите пронајдени при полнење на раката. Ако не се постават посакуваните позиции на зглобовите влегуваме во мод за управување на додатокот преку копче каде при првиот повик се креира. При секој клик на копчето се зголемува вредноста за еден, со кое алтенираме помеѓу отворената и затворената позиција на фаќачот.

```
if joint_ids is None:
    joint_ids = self.attachment_joint_ids
if joint_targets is None:
    joint_targets = self.attachment_open_targets
    if self.attachment_button is None:
        self.attachment_button = p.addUserDebugParameter(
                                    "Close/Open Gripper",1,0,1)
    else:
        if p.readUserDebugParameter(self.attachment_button) %2 == 0:
            joint_targets = self.attachment_close_targets
        else:
            joint_targets = self.attachment_open_targets
            self.change_attachment_force = False
```

Ако е отворен фаќачот се поставува силата на 500 за да не се помести при помрднување на раката. Кога фаќачот ќе почне да се затвора и наиде на пречка ќе почне упорно да бутка и ќе се накачи до максималната сила од 500. Со функцијата getJointStates() се зима информација за секој зглоб колку е моменталната применета сила и се проверува дали е повеќе од 100. Ако секој зглоб ја има надминато горната граница поставуваме знаме дека зглобовите се запрени и треба да се смени максималната сила на 50. Сето ова се прави бидејќи ако го притискаме објектот со сила повеќе од 100 константно, започнуваат внатрешни колизии кои ќе го исфрлат објектот исто како во реалниот свет. Ако се постават посакувани позиции на влез на функцијата тогаш се

влегува во мод за управување на раката преку команди. Кодот тука е идентичен, единствена разлика е тоа што имаме циклус со 30 чекори за динамички да се затвори или отвори фаќачот.

```
max_force, joint_stopped = 500, True
for joint_state in p.getJointStates(self.robot_arm,self.attachment_joint_ids):
    if abs(joint_state[3]) < 100:
        joint_stopped = False
if (joint_stopped and joint_targets == self.attachment_close_targets) or
self.change_attachment_force:
    self.change_attachment_force = True
    max_force = 50
p.setJointMotorControlArray(self.robot_arm, joint_ids, p.POSITION_CONTROL,
joint_targets, forces=[max_force]*len(joint_ids))
```

6.4.13 capture_image(self, link_state, camera_offset, near, far, size, fov, use_gui)

Функцијата capture_image е апстракција на класата Camera каде камерата се прикачува на извршниот елемент на раката. Со аргументот link_state се одбира на кој линк од раката да се прикачи камерата, стандардно се прикачува на последниот линк. Дополнителен аргумент е офсетот од линкот за точно да се намести камерата бидејќи можно е да се најде внатре во раката. Следните параметри се користат за поставување на параметрите на камерата. Исто така функцијата има и две опции за користење, преку копче или директно преку повик на функцијата. Ако се користи преку кориснички интерфејс, дополнително се додава копче во симулацијата и при секој клик знаменцето use_gui се става невистинито за се влеземе во делот од кодот за сликање.

```
if link_state is None:
    link_state = p.getLinkState(self.robot_arm, self.last_joint_id)
if use_gui is True:
    if self.capture_image_button is None:
        self.capture_image_button = p.addUserDebugParameter(
                                    "Capture Image", 1, 0, 1)
    else:
        if p.readUserDebugParameter(self.capture_image_button) ==
                                    self.prev_button_state:
            self.prev_button_state += 1   use_gui = False
```

Прво се пресметува која е целната точка на камерата така што се почнува од позицијата на линкот и се офсетнува за поставениот аргумент camera_offset. За да има точен офсет по сите оски дополнително треба офсетот да го ротираме како ориентацијата на линкот. Ориентацијата за камерата се добива од ориентацијата на поставениот линк, каде дополнително треба да се претворат од радијани во степени. На крај се зима слика и се враќа објектот за понатамошна обработка.

```
if use_gui is False:
    rot = np.array(p.getMatrixFromQuaternion(link_state[5])).reshape(3, 3)
    offset = self.rotate_point(rot, camera_offset)
    target = (link_state[4][0]+offset[0], link_state[4][1]+offset[1],
link_state[4][2]+offset[2])
    orn = p.getEulerFromQuaternion(link_state[5])
```

```
    camera = Camera(target, 0.01, (np.rad2deg(orn[2]), np.rad2deg(-orn[0]),
np.rad2deg(orn[1])), near, far, size, fov)
    camera.shot()
    return camera
```

6.4.14 state_logging(self, log_name='', start_stop=None, object_ids=None)

Оваа функција е наменета за запишување на секој објект во симулацијата. Со листата object_ids се одбираат кои објекти да се запишуваат, стандардно се одбира само раката. Ако не се користи аргументот strat_stop, се креира копче кое со кое се започнува и завршува со запишувањето на раката. Дополнително ако не постои патеката logs/ се креира и сите датотеки се зачувуваат во таа патека, и на крајот од името на датотеката се додава број на запишување кое се користи за сортирање на датотеките.

```
if object_ids is None:
    object_ids = [self.robot_arm]

if start_stop is None:
    if self.record_button is None:
    self.record_button = p.addUserDebugParameter("Start/Stop Logging", 1,0,1)
    else:
        if p.readUserDebugParameter(self.record_button) % 2 == 0:
            if self.log is None:
                if not os.path.exists(f'{self.file_head}/logs/'):
                    os.mkdir(f'{self.file_head}/logs/')
                self.log = p.startStateLogging(p.STATE_LOGGING_GENERIC_ROBOT,
          f"{self.file_head}/logs/{log_name}_{self.rec_cnt}.txt", object_ids)
                self.rec_cnt += 1
        elif self.log is not None:
            p.stopStateLogging(self.log)
            self.log = None
else:
```

Се користи истиот код и за повик преку параметарот start_stop, единствена разлика е што кодот е поедноставен и нема потрепа од дополнителни услови за да не се започне со запишување пак следниот чекор при повик на функцијата.

```
    if start_stop == 'start':
        if not os.path.exists(f'{self.file_head}/logs/'):
            os.mkdir(f'{self.file_head}/logs/')
        self.log = p.startStateLogging(p.STATE_LOGGING_GENERIC_ROBOT,
          f"{self.file_head}/logs/{log_name}_{self.rec_cnt}.txt", object_ids)
        self.rec_cnt += 1
    else:
        p.stopStateLogging(self.log)
```

6.4.15 read_log_file(self, file_path)

Функција превземена од pybullet која служи за читање на датотеките за запис и зачувување за вредностите во листа која на крајот се враќа. Прво се читаат основните параметри од датотеката како големината, клучевите и нивниот број. Следно се оди низ сите парчиња поделени со текстот \xaa\xbb и се одпакуваат вредностите во листата.

```
f = open(file_path, 'rb')
keys = f.readline().decode('utf8').rstrip('\n').split(',')
fmt = f.readline().decode('utf8').rstrip('\n')
sz = struct.calcsize(fmt)
whole_file = f.read()
chunks = whole_file.split(b'\xaa\xbb')
log = []
for chunk in chunks:
    if len(chunk) == sz:
        values = struct.unpack(fmt, chunk)
        record = []
        for i in range(len(fmt)):
            record.append(values[i])
        log.append(record)
return log
```

6.4.16 autocomplete_logs(self, log_names)

Се користи за пронајдување на сите слични фајлови користејќи го glob. На влез се прима листа од имиња на датотеки за запис, но може и да биде едно име кој како исклучок се ковертира во листа со еден елемент. Следно се мине низ листата со имиња и се пронајдуваат сите датотеки со слични имиња и се сортираат по последниот индекс на запишување. Откако ќе се сортираат се чита пронајдената датотека и се додава на листата од записи кои на крајот се враќа.

```
def sort_key_func(s):
    return int(os.path.basename(s).split('_')[-1][:-4])
log = []
if type(log_names) != list:
    log_names = [log_names]
for log_name in log_names:
    for log_name_index in
sorted(glob.glob(f'{self.file_head}/logs/'+f"{log_name}*.txt"),
key=sort_key_func):
        log += self.read_log_file(log_name_index)
return log
```

6.4.17 replay_logs(self, log_names, skim_trough=True, object_ids=None)

Функција која ги користи сите горни функции за запис и се врши рекреација на секој извршен чекор на раката. Стандардно се користи само идито на раката и се генерира листа од пронајдените записи, кој како исклучок ако не е наполнета листата завршува функцијата. Имаме дополнителна опција за одбирање на чекорите преку лизгач или секвенцијално да се изврши секој чекор.

```
if object_ids is None:
    object_ids = [self.robot_arm]

logs = self.autocomplete_logs(log_names)
if not logs:
    return "No logs found"

recordNum, objectNum = len(logs), len(object_ids)
```

```
if skim_trough
    step_index_param = p.addUserDebugParameter("Replay Step", 0, recordNum /
objectNum - 1, 0)
else:
    step_index = -1
while (p.isConnected()):
    st = time.time()
```

Ако се користи лизгач секој циклус се чита од лизгачот кој чекор од листата да земе, инаку индексот се зголемува за еден. Следно се ресетира позицијата и ориентацијата на основата на раката и се креира циклус кој мине низ секој зглоб на раката. Се земаат сите информации за зглобот и ако не е фиксен се поставува на посакуваната позиција.

```
    p.stepSimulation()
    if skim_trough:
        step_index = int(p.readUserDebugParameter(step_index_param))
    else:
        step_index+=1
    for objectId in range(objectNum):
        record = logs[step_index * objectNum + objectId]
        Id = record[2]
        p.resetBasePositionAndOrientation(Id, [record[3], record[4],
record[5]],[record[6], record[7], record[8], record[9]])
        jf_index = 0
        for i in range(p.getNumJoints(Id)):
            joint_info = p.getJointInfo(Id, i)
            qIndex = joint_info[3]
            if joint_info[1] == b'attachment_joint':
                break # break if joint is from attachment
            if qIndex > -1:
                if self.use_dynamics:
                    p.setJointMotorControl2(Id, i, p.POSITION_CONTROL,
record[qIndex - 7 + 17], force=self.joint_forces[jf_index])
                    jf_index+=1
                else:
                    p.resetJointState(Id, i, record[qIndex - 7 + 17])
```

Дополнително се додаваат и модулите за цртање на траекторијата и печатење на позицијата и ориентацијата на извршниот елемент на раката. На крај имаме двоен услов каде ако го користиме лизгачи треба да се запре симулацијата преку копчето за исклучување или ако се користи секвенцијално ќе заврши циклусот кога ќе се стигне до последниот запис од листата.

```
    link_state = p.getLinkState(self.robot_arm,self.last_joint_id)
    if self.use_draw_trajectory:
        self.draw_trajectory(link_state[4], life_time =
self.trajectory_lifetime)
    if self.use_display_pos_and_orn:
        self.display_pos_and_orn(link_state[4], link_state[5],
self.last_joint_id)

    if step_index == recordNum / objectNum - 1 and skim_trough is False:
```

```
            break # break condition for the last log
        if skim_trough and self.quit_simulation(use_gui=True):
            break
        ex_time = self.time_step - (time.time() - st)
        time.sleep(ex_time if ex_time > 0 else 0)
```

6.4.18 convert_logs_to_prg(self, log_names, file_name)

Со оваа функција се конвертираат датотеките за запис на раката mitshubisi во програма за софтверот RT Toolbox 2. Прво се креира листа од записите и ако нема пронајдено функцијата запира. Потоа се поставуваат првите линии код со кои се вклучуваат сервата, се поставува брзината. Следно се креира циклус кој мине низ секој запис од низата и ги поместува зглобовите на посакуваните позиции. Низата се креира на крајот од програмата каде се користит конвенцијата jStep(x) = (j1,j2,j3,j4,j5,j6) и на крај се зачувува генерираниот текст во датотеката со поставеното име.

```
logs = self.autocomplete_logs(log_names)
if not logs:
    return "No logs found"
num_steps = len(logs)
pgr_string =  f"1 Servo on  \n2 Wait M_Svo=1 \n3 Base 0 \n4 Ovrd 10 \n5 Dim
jStep({num_steps})\n"
pgr_string += f"6 For m1 = 1 To {num_steps} \n7 Mov jStep(m1) \n8 Next m1 \n9
end \n"
for step in range(num_steps):
    pgr_string +=  f"jStep({step+1})=("
    for joint in self.joint_ids:
        pgr_string += f"{np.rad2deg(logs[step][joint + 17]):.3f}"
        pgr_string +=  ',' if joint != self.joint_ids[-1] else ')\n'
file = open(f"{self.file_head}/{file_name}.prg", "w")
file.write(pgr_string)
file.close()
```

6.4.19 import_robot_arm(self, file_path)

Функцијата се користи за внесување на креирана роботска рака од овој проект. Се поставува знамето дека робот е внесен и се сменува името и патеката до датотеката. Следно се читаат првите линии од датотеката каде се зачувани во коментар DH параметрите и границите на секој зглоб. DH параметрите се додаваат по истата конвенција како што се креира модуларен робот. Секоја вредност се добива со тоа што линијата се разделува во листа со терминатор од празна линија и се претвораат сите елементи во реален број.

```
self.is_imported = True
file_head, tail = os.path.split(file_path)
self.file_head = file_head
self.name = tail[:-5]
file = open(file_path, 'r')
lines = file.readlines()
for i, line in enumerate(lines[2:]):
    if line.strip() == '-->':
        break
    dh_list = line.strip().split()
```

```
    if dh_list[0] == 'revolute':
        self.add_revolute_joint(sp.symbols(f'theta{i+1}'),float(dh_list[2]),fl
oat(dh_list[3]),float(dh_list[4]),
            np.rad2deg(float(dh_list[6])),np.rad2deg(float(dh_list[7])),float(
dh_list[8]),float(dh_list[5]),int(dh_list[9]))
    elif dh_list[0] == 'prismatic':
        self.add_prismatic_joint(float(dh_list[1]),sp.symbols(f'd{i+1}'),float
(dh_list[3]),float(dh_list[4]),float(dh_list[6]),float(dh_list[7]),float(dh_li
st[8]),float(dh_list[5]),int(dh_list[9]))
    else:
        self.add_fixed_joint(float(dh_list[1]),float(dh_list[2]),float(dh_list
[3]),float(dh_list[4]),int(dh_list[9]))
```

6.4.20 import_foreign_robot_arm(self, file_path, scaling=5)

За внесување на надворешни роботски раце дополнително се поставува и знамето is_foreign кое се користи во останатиот дел од програмата. Дополнително се поставува и скалирањето бидејќи не секој робот е исто димензиониран. На крај се зачувуваат името и пакета кон датотеката.

```
self.is_imported = self.is_foreign = True
self.scaling = scaling
file_head, tail = os.path.split(file_path)
self.file_head = file_head
self.name = tail[:-5]
```

6.4.21 write_text(self, text, position, orientation, spacing, plane, scale, log_text)

Функцијата write_text е најголемата апстракција од управувањето на раката преку код каде со неа се црта текст во симулацијата, кој може дополнително да се запишува секој чекор. Со циклус се оди низ секоја буква и се користи функицјата write_letter() за цртање на буквата на поставената позиција и ориентација. Дополнително се одбира оддалечувањето помеѓу буквите и нивната големина.

```
if log_text:
    self.state_logging(text,'start')
for letter in text:
    self.write_letter(letter, position, orientation, plane, scale)
    position[1] += spacing
if log_text:
    self.state_logging(start_stop='stop')
```

6.4.22 write_letter(self, letter, position, orientation, plane=(0, 0, 0), s=0.5)

Се користи за цртање на буква користејќи ја функцијата move(). На почеток се разделуваат позицијата и ориентацијата и зависно од која буква е внесена се извршува кодот внатре во еден од условите. Подолу имаме пример за буквата т каде се креираат две линии и дополнително се поставува и променливата s која се користи за скалирање на буквата.

```
x,y,z = position
R,P,Y = orientation
if letter == 'T':
    self.move((x,y,z,R,P,Y),(x,y+1.5*s,z, R,P,Y),plane,'linear',30)
```

```
    self.move((x,y+0.75*s,z, R,P,Y),(x+2*s,y+0.75*s,z,
R,P,Y),plane,'linear',30)
```

6.4.23 move(self, start, end, plane, interpolation, steps, param, closed, log_move)

Функција со која се пресметува линеарна и кружна интерполација помеѓу две точки. При секој повик прво се ресетира цртањето на траекторијата и се разделуваат почетната и крајната точка за позиција и ориентација. Ако дополнително се запишува линијата, прво се поместува раката на посакуваната позиција, се ресетира цртањето на траекторијата и се започнува со запишувањето. Со помош на функцијата rotation_matrix_from_euler_angles се креира ротациската матрица која ќе се користи за проектирање на точките на неа.

```
self.has_prev_pose = [False]*5
x1, y1, z1, R1, P1, Y1 = start
x2, y2, z2, R2, P2, Y2 = end
if log_move:
    self.move2point((x1,y1,z1),(R1,P1,Y1))
    self.has_prev_pose = [False]*5
    self.state_logging(f"move_{interpolation}_{start}_{end}"+ ('' if param is
None else f"_{param}"),'start')
rot_mat = rotation_matrix_from_euler_angles(np.deg2rad(plane),'xyz')
```

Ако интерполацијата е линеарна се користи функцијата linspace од numpy која интерполира помеѓу точките со одреден број чекори. Потоа се креира циклус кој мине низ сите точки и ја помрднува раката кон нив со `move2point()`.

```
if interpolation == 'linear':
    for x,y,z,R,P,Y in
zip(np.linspace(x1,x2,steps),np.linspace(y1,y2,steps),np.linspace(z1,z2,steps)
,np.linspace(R1,R2,steps),np.linspace(P1,P2,steps),np.linspace(Y1,Y2,steps)):
        if not self.move2point((x,y,z),(R,P,Y)):
            return False
```

Ако интерполацијата е кружна дополнително треба да се додадат и параметрите за елипсата, кое решение да се одбере и од која страна да започне со цртање. Равенките се решаваат со помош на sympy и ако резултатот не е имагинарен се продолжува со пронајдување на почетниот и крајниот агол на елипсата да одговара со почетната и крајната точка.

```
elif interpolation == 'circular' and param != None:
    xs,ys = sp.symbols('xs,ys')
    a,b,res_num,side = param
    eq1 = sp.Eq((xs-x1)**2/a**2+(ys-y1)**2/b**2,1)
    eq2 = sp.Eq((xs-x2)**2/a**2+(ys-y2)**2/b**2,1)
    results = sp.solve([eq1,eq2],(xs,ys))
    if not all(x[0].is_real or x[1].is_real for x in results):
        print("No result for the parameters a and b")
        return False
    x_center,y_center = results[res_num]
    res_t = [-np.arccos(float((x1-x_center)/a)) + 2*np.pi,
np.arccos(float((x1-x_center)/a))]
    start_angle = [t for t in res_t if np.around(float(y_center +
b*np.sin(t)),3) == np.around(y1,3)][0]
```

```
    res_t = [-np.arccos(float((x2-x_center)/a)) + 2*np.pi,
np.arccos(float((x2-x_center)/a))]
    end_angle = [t for t in res_t if np.around(float(y_center +
b*np.sin(t)),3) == np.around(y2,3)][0]
```

Ако се постави дополнителното знаме closed да биде вистинито, се менуваат почетниот и крајниот агол да се нацрта целиот круг. Следно зависно од која страна сакаме да се нацрта елипсата се конвертираат почетниот и крајниот агол во интерполирана листа од агли помеѓу нив.

```
    if closed:
        start_angle, end_angle = (0,2*np.pi) if side else (2*np.pi,0)
    if side:
        t = np.linspace(start_angle, end_angle, steps)
    else:
        if start_angle < end_angle:
            t = np.hstack((np.linspace(start_angle,
0,int(steps*start_angle/(2*np.pi-end_angle+start_angle))),
np.linspace(2*np.pi, end_angle, int(steps*(2*np.pi-end_angle)/(2*np.pi-
end_angle+start_angle)) )))
        else:
            t = np.hstack((np.linspace(start_angle, 2*np.pi,
int(steps*(2*np.pi-start_angle)/(2*np.pi-start_angle+end_angle)) ),
            np.linspace(0, end_angle, int(steps*end_angle/(2*np.pi-
start_angle+end_angle)) )))
```

На крај се креира циклус кој мине низ сите генерирани точки и ја помрднува раката кон нив со move2point(). Во овој случај секогаш се црта кругот по x и y рамнината и се ротира со rot_mat матрицата. Кога ќе се заврши со сите генерирани точки се запира запишувањето ако е одобрено преку параметарот.

```
    for x,y,z,R,P,Y in zip(x_center + a*np.cos(t),y_center +
b*np.sin(t),np.linspace(z1, z2, steps),
 np.linspace(R1,R2,steps),np.linspace(P1,P2,steps),np.linspace(Y1,Y2,steps)):
        if not self.move2point(self.rotate_point(rot_mat,(x,y,z)),(R,P,Y)):
            return False
if log_move:
    self.state_logging(start_stop='stop')
```

6.4.24 move2point(self, position, orientation)

Функцијата е идентична како функцијата step(), во која се управува раката. Во овој случај се тргнати копчињата бидејќи се управува раката преку код и имаме дополнителен услов за запирање на циклусот кога се стигне до посакуваната позиција и ориентација извршниот елемент на рака. Имаме двоен услов, каде ако се користи динамика само се чека еден чекор на симулацијата за да се стопира. Ако се користи динамика за секој чекор се проверува со функцијата allclose од numpy колку се блиску позициите на сите зглобови до посакуваните со зададениот праг или ако се поминати одреден број на чекори затоа што можно е раката да остане во бесконечен циклус.

```
if self.use_dynamics and
(np.allclose([p.getJointState(self.robot_arm,joint_id)[0]
      for joint_id in self.joint_ids],joint_targets,self.ik_joint_error) or
      steps > self.frames_to_complete):
```

```
    break
elif not self.use_dynamics and steps > 1:
 break
steps+=1
```

6.4.25 limit_pos_target() and limit_joint_targets()

Кратки функции во кои се лимитираат позициите на зглобовите и позицијата во тридимензионалниот свет. Користејќи clip од numpy се лимитира секој зглоб со поставените горни и долни прагови, а позицијата се лимитира ако се поставени лимитите за оските кои се поставуваат со функцијата set_position_limits().

6.4.26 get and set params()

Има дополнителни функции кои помагаат при поставување на параметрите на раката. Првата е set_dynamic_conditions() каде се поставува максималната грешка во зглобовите и максималниот број на чекори на извршување. Следна е set_attachment_targets со која се поставуваат посакуваните позиции на фаќачот кога е отворен и затворен. Со set_position_limits се поставуваат минималната и максималната оскина позиција која раката може да ја достигне. На крај е функцијата get_joint_states која ги враќа информациите за секој зглоб во раката.

6.4.27 add joints()

Со функциите add_revolute_joint, add_prismatic_joint и add_fixed_joint се додава модуларно нов зглоб на раката. Се користи dh конвенција за креирање зглобовите и дополнително се поставуваат долната и горната граница на зглобот, максималната брзина и забрзување. Сите параметри се зачувуваат во соодветните листи, каде за фиксниот зглоб имаме исклучок во тоа што се креира случајна симболичка променлива и не се додаваат ограничувањата.

6.4.28 get_dh_joint_to_joint(self, start_joint, end_joint)

Се користи за пресметка на DH матрицата од поставениот стартен и краен зглоб според книгата [1]. Исто така има дополнителна функција get_dh_matrix која ја повикува да пресмета за сите зглобови во раката. Формулата за пресметка на матрицата е превземена од класата која припаѓа на fundamentals-of-robotics [9]. На крај користејќи sumpy се симплифицира матрицата и се враќа назад.

```
pose = hpose3()
for link in self.links[start_joint:end_joint]:
    _, theta, d, a, alpha = link
    pose = pose * dh_joint_to_joint(theta, d, a, alpha)
pose.simplify()
return pose
```

6.4.29 jacobian(self)

Користејќи sumpy се пресметува јакобијанот на роботската рака и се дели на линеарен и ротациски јакобијан. Линеарниот се добива преку пресметка на dh матрицата и пресметување јакобијан на матрицата со сите симболички променливи. Ротацискиот јакобијан се пресметува така што вертикално се надодава матрица со нули ако зглобот е транслаторен или се зимаат првите 3 елементи од третиот ред на dh матрицата.

Дополнително ако зглобот е фиксиран се прескокнува додавање на нов ред во матрицата на ротацискиот јакобијан.

```
linear_jacobian = self.get_dh_matrix()[:3, 3].jacobian(self.joint_variables)
pose = hpose3()
angular_jacobian = sp.Matrix([[], [], []])
for link in self.links:
    joint_type, theta, d, a, alpha = link
    z_i_m1 = sp.Matrix([0, 0, 0]) if joint_type == p.JOINT_PRISMATIC else
pose[:3, 2]
    if joint_type != p.JOINT_FIXED:
        angular_jacobian = sp.Matrix.hstack(angular_jacobian, z_i_m1)
    pose = pose * dh_joint_to_joint(theta, d, a, alpha)
```

7 Надградба на софтверскиот пакет

Проектот е направен од старт да биде модуларен каде може многу работи да се напреднат. Лесно може да се додадат нови додатоци и вистински роботски раце, исто така да се додадат и нови објекти во симулацијата. Имаме и многу функционалности што може да се наградат како и примери користејќи ја симулацијата. Во следните потточки ќе биде објаснето како да се имплементираат новите функционалности.

7.1 Креирање на нов додаток

За да се креира нов додаток што ќе се додаде на роботкстака рака, треба да се следи следната конвенција. Се креира нова папка со името на додатокот во папката /attachments каде се стават сите датотеки во неа. Главниот URDF треба да биде со исто име со папката и првата врска во датотеката треба да се нарекува base за да може да се рефренцира со крајниот зглоб на раката. Следно треба да се провери дали додатокот е компатибилен со функцијата actuate_attachment(), засега функцијата е кодирана за отворање и затворање на фаќач. За слични фаќачи, може само да се искористи функијата set_attachment_targets() со која се поставуваат саканите позиции на фаќачот.

7.2 Додавање на нова роботска рака

Кога се додава нова роботка рака треба да се провери дали патеката на визуелните датотеки се правилно референцирани во главната датотека. Важно е да се отстранат сите хасго макроа бидејќи не работат со pybullet, а има многу раце што се така конфигурирани за олеснување при работа со нив. Има два примери Franka Emirka panda [11] и xArm [12] во патеката robot_arms/ како примери за како точно треба да се имплементира новата рака. Ако раката на роботот има додаток, треба да се преименува зглобот што го поврзува со раката во attachment_joint за функцијата find_joint_ids() правилно да ги идентификува сите зглобови на роботската рака.

7.3 Креирање на нов свет во pybullet

Користејќи ја класата PybulletSimulation, може да се направи нова функција за додавање објекти во симулацијата. Засега само се користат URDF датотеките обезбедени од pybullet каде се направени основни објекти како работна маса, послужавник и сите вообичаени објекти пронајдени во патеката pybullet_data.

7.4 Нови функционалности на софтверскиот пакет

Проектот има многу добра основа и функционалности за креирање на комплициран свет, но секогаш ќе има подобрувања и нови функционалности во pybullet што ќе треба да се имплементираат и тука. Затоа целиот проект е отворен за сите за да може да се продолжи со неговиот развој. Имам поставено листа од можни имплементации:

- Опција за внесување на инерцијални вредности на линковите на раката
- Имплементација на сферични зглобови
- Други типови на интерполација
- Да се додадат и останатите букви во write_letter()
- Имплементација на инверзна динамика
- Нова класа за креирање и управување на мобилни роботи
- Нова класа за креирање и управување на квадропед робот
- Точна имплементација на фаќачот на Mitsubishi раката
- Директна комуникација со управувачот CR750-Q [16]

7.5 Нови примери користејќи го софтверскиот пакет

Со трите jupyter тетратки во главно се поминати сите основни функционалности на софтверскиот пакет, но има многу можности за креирање на многу покомплицирани сценарија. Поставена е листа на можни имплементации на нови примери:

- Покомплициран систем за детекција и земање на објекти
- Имплементација на мобилна роботкса рака [14]
- Имплементирање на рака за фрлање на објекти [15]
- Користење на машинско учење за управување на раката

8 Заклучок

Симулирањето на роботска рака може да обезбеди корисен и ефикасен начин за тестирање и оценување на перформансите на управувачките алгоритми и стратегиите за планирање на движењето на раката. Со користење на симулација, можно е прецизно да се моделира динамиката и физичките ограничувања на раката на роботот, како и да се вклучат модели на сензори и други фактори од реалниот свет. Ова може да помогне да се идентификуваат и да се отстранат проблемите пред да се имплементира управувачкиот систем на физичкиот робот.

Употребата на pybullet овозможува лесна визуелизација и анализа на симулираната роботска рака, што го прави корисна алатка за прототип и развој на управувачките системи. Освен роботски раце може да се надгради симулацијата со голем број на мобилни роботи како дронови, квадропеди. Да се направи комплициран симулиран свет кој може да се користи како многу добра алатка за проучување и истражување во роботиката.

Овој концепт добро се покажа при креирање на Mitsubishi RV-2F-Q роботската рака во симулаторот. Лесно може да се изврши анализа на раката и да се управува преку интерактивниот кориснички интерфејс или преку командите за управување. Сите примери во тетратката за планирање на траекторијата на раката како извршување на линеарна и кружна интерполација се конвертираат во програма за да се изврши на раката. Генерираните движења се испробуваат на физичката рака каде имаме прецизно репродуцирање на движењето направено во симулација.

9 Референци

[1] Елизабета Лазаревска, Вовед во роботиката, Универзитет „Св. Кирил и Методиј“, Скопје, 2021

[2] Asad Yousuf, Introducing Kinematics with Robot Operating System, https://peer.asee.org/introducing-kinematics-with-robot-operating-system-ros.pdf, 10.12.2022

[3] https://docs.conda.io/en/latest/miniconda.html

[4] https://code.visualstudio.com/

[5] https://git-scm.com/

[6] Erwin Coumans, Yunfei Bai, PyBullet, a Python module for physics simulation for games, robotics and machine learning, https://pybullet.org/wordpress/,10.12.2022

[7] https://www.sympy.org/en/index.html

[8] https://numpy.org/

[9] https://gitlab.com/feeit-freecourseware/fundamentals-of-robotics

[10] https://github.com/vasetrendafilov/robot-simulation

[11] https://www.franka.de/

[12] https://www.ufactory.cc/xarm-collaborative-robot

[13] Erwin Coumans, Yunfei Bai, PyBullet Quickstart Guide, https://docs.google.com/document/d/10sXEhzFRSnvFcl3XxNGhnD4N2SedqwdAvK3dsihxVUA/edit?usp=sharing, 10.12.2022

[14] https://github.com/bulletphysics/bullet3/blob/master/examples/pybullet/examples/inverse_kinematics_husky_kuka.py

[15] https://pybullet.org/wordpress/index.php/2019/03/30/tossingbot-learning-to-throw-arbitrary-objects-with-residual-physics/

[16] https://github.com/tork-a/melfa_robot