|  |  |  |
| --- | --- | --- |
| **images** | **УНИВЕРЗИТЕТ “Св. КИРИЛ И МЕТОДИЈ” - СКОПЈЕ**  **ФАКУЛТЕТ ЗА ЕЛЕКТРОТЕХНИКА И ИНФОРМАЦИСКИ ТЕХНОЛОГИИ** | **index** |

- **ДИПЛОМСКА РАБОТА** -

по предметот

**РОБОТИКА 1**

**Тема**

**ФИЗИЧКА СИМУЛАЦИЈА НА РОБОТКСА РАКА**

|  |  |
| --- | --- |
| Ментор: | Изработил: |
| Доц. д-р Горјан Наџински | Васил Трендафилов, индекс бр. 139/2018  e-mail: vasetrendafilov@gmail.com |

*Скопје, декември 2022*

**Содржина**

[Апстракт 4](#_Toc122362461)

[Вовед 5](#_Toc122362465)

[1 Користени технологии и алатки 5](#_Toc122362466)

[2 Инсталација на софтверскиот пакет 5](#_Toc122362467)

[2.1 Инсталација на потребни апликации 5](#_Toc122362468)

[2.2 Инсталација на библиотеки и клонирање на проектот 5](#_Toc122362469)

[3 Користење на софтверскиот пакет 6](#_Toc122362470)

[3.1 Модуларна роботкса рака 6](#_Toc122362471)

[3.1.1 Креирање и пресметување на DH матрица и јакобијан на раката 6](#_Toc122362472)

[3.1.2 Инверзна кинематика на роботска рака со присматичен грипер 7](#_Toc122362473)

[3.1.3 Директна кинематика на роботска рака со револутен грипер 8](#_Toc122362474)

[3.1.4 Импортирање на рака и репродуцирање на логовите 9](#_Toc122362475)

[3.1.5 Пример за детекција и зимање на објект со грипер 10](#_Toc122362476)

[3.2 Популарни роботски раце 12](#_Toc122362477)

[3.2.1 Franka Emirka panda роботска рака 12](#_Toc122362478)

[3.2.2 xArm роботкса рака 13](#_Toc122362479)

[3.3 Mitsubishi роботска рака 14](#_Toc122362480)

[3.3.1 Контрола на Mitsubishi роботската рака со гриппер 14](#_Toc122362481)

[3.3.2 Поднесување на параметри на раката за контрола преку комадни 15](#_Toc122362482)

[3.4 Кратки копчиња за pybullet 17](#_Toc122362483)

[4 Конектирање на CR750-Q со RT Toolbox 2 18](#_Toc122362484)

[5 Архитектура на системот 18](#_Toc122362485)

[5.1 Хиерархија помеѓу класите 18](#_Toc122362486)

[5.2 Хиерархија на класата PybulletSimulation 19](#_Toc122362487)

[5.3 Хиерархија на класата Camera 20](#_Toc122362488)

[5.4 Хиерархија на класата DH2Urdf 21](#_Toc122362489)

[5.5 Хиерархија на функцијата interact 21](#_Toc122362490)

[5.6 Хиерархија на контрола преку команди 23](#_Toc122362491)

[5.7 Хиерархија за креирање на модуларна роботска рака 24](#_Toc122362492)

[6 Документација за кодот 24](#_Toc122362493)

[6.1 DH2Urdf() 24](#_Toc122362494)

[6.1.1 \_\_init\_\_(self, dh\_params, constraints, attachment = None) 25](#_Toc122362495)

[6.1.2 write\_visual\_shape(self, rpy, xyz, length, radius, material\_name) 25](#_Toc122362496)

[6.1.3 write\_inertial(self, rpy, xyz, mass, inertia\_xxyyzz) 25](#_Toc122362497)

[6.1.4 write\_link(self, name, visual\_shapes, save\_visual = True) 26](#_Toc122362498)

[6.1.5 write\_joint(self, name, joint\_type, parent, child, rpy, xyz, constraints = None) 26](#_Toc122362499)

[6.1.6 save\_urdf(self, file\_path) 26](#_Toc122362500)

[6.2 Camera() 29](#_Toc122362501)

[6.2.1 \_\_init\_\_(self,cam\_target, distance, rpy, near, far, size, fov) 29](#_Toc122362502)

[6.2.2 shot(self) 29](#_Toc122362503)

[6.2.3 rgbd\_2\_world(self, w, h, d) 29](#_Toc122362504)

[6.3 PybulletSimulation() 30](#_Toc122362505)

[6.3.1 \_\_init\_\_(self, connection\_mode, fps, gravity, cam\_param, cam\_target) 30](#_Toc122362506)

[6.3.2 configure(self) 30](#_Toc122362507)

[6.3.3 load\_any\_object() 30](#_Toc122362508)

[6.3.4 load\_random\_objects(self, count\_objects, position, orientation, scaling) 30](#_Toc122362509)

[6.4 RobotArm() 31](#_Toc122362510)

[6.4.1 \_\_init\_\_(self, params) 31](#_Toc122362511)

[6.4.2 add\_pose\_sliders(self, x\_range, y\_range, z\_range) 31](#_Toc122362512)

[6.4.3 add\_joint\_sliders(self) 32](#_Toc122362513)

[6.4.4 find\_joint\_ids(self) 32](#_Toc122362514)

[6.4.5 reset\_joints(self, joint\_states=None) 33](#_Toc122362515)

[6.4.6 load\_robot\_arm(self) 33](#_Toc122362516)

[6.4.7 interact(self, kinematics, use\_orientation\_ik, x\_range, y\_range, z\_range) 34](#_Toc122362517)

[6.4.8 step(self) 34](#_Toc122362518)

[6.4.9 display\_pos\_and\_orn(self, position, orientation, link\_id) 36](#_Toc122362519)

[6.4.10 draw\_trajectory(self, current, target=None, life\_time=15, line\_index=0) 36](#_Toc122362520)

[6.4.11 quit\_simulation(self, use\_gui = False) 36](#_Toc122362521)

[6.4.12 actuate\_attachment(self, joint\_ids=None, joint\_targets=None) 37](#_Toc122362522)

[6.4.13 capture\_image(self, link\_state, camera\_offset, near, far, size, fov, use\_gui) 38](#_Toc122362523)

[6.4.14 state\_logging(self, log\_name='', start\_stop=None, object\_ids=None) 39](#_Toc122362524)

[6.4.15 read\_log\_file(self, file\_path) 39](#_Toc122362525)

[6.4.16 autocomplete\_logs(self, log\_names) 40](#_Toc122362526)

[6.4.17 replay\_logs(self, log\_names, skim\_trough=True, object\_ids=None) 40](#_Toc122362527)

[6.4.18 convert\_logs\_to\_prg(self, log\_names, file\_name) 42](#_Toc122362528)

[6.4.19 import\_robot\_arm(self, file\_path) 42](#_Toc122362529)

[6.4.20 import\_foreign\_robot\_arm(self, file\_path, scaling=5) 43](#_Toc122362530)

[6.4.21 write\_text(self, text, position, orientation, spacing, plane, scale, log\_text) 43](#_Toc122362531)

[6.4.22 write\_letter(self, letter, position, orientation, plane=(0, 0, 0), s=0.5) 43](#_Toc122362532)

[6.4.23 move(self, start, end, plane, interpolation, steps, param, closed, log\_move) 44](#_Toc122362533)

[6.4.24 move2point(self, position, orientation) 45](#_Toc122362534)

[6.4.25 limit\_pos\_target() and limit\_joint\_targets() 46](#_Toc122362535)

[6.4.26 get and set params() 46](#_Toc122362536)

[6.4.27 add joints() 46](#_Toc122362537)

[6.4.28 get\_dh\_joint\_to\_joint(self, start\_joint, end\_joint) 46](#_Toc122362538)

[6.4.29 jacobian(self) 47](#_Toc122362539)

[7 Надградба на софтверскиот пакет 47](#_Toc122362540)

[7.1 Додавање на нова алатка 47](#_Toc122362541)

[7.2 Додавање на новa роботска рака 47](#_Toc122362542)

[7.3 Креирање на нов свет во pybullet 48](#_Toc122362543)

[7.4 Нови функционалности на софтверскиот пакет 48](#_Toc122362544)

[7.5 Нови примери користејки го софтверскиот пакет 48](#_Toc122362545)

[8 Заклучок 48](#_Toc122362546)

[9 Референци 48](#_Toc122362547)

**Листа на слики**

[Слика 1 дх матрица 6](#_Toc121595434)

**Листа на табели**

[Табела 1 Карактеристики на батерии кои може да се полнат 6](#_Toc222636361)

# Апстракт

*Автономните возила сега се неопходност во овој свет, што го прави апсолутно важно да се има звучен алгоритамски систем кој ќе врши откривање на различни карактеристики на патот, како што се маркери за лента, откривање возила итн. Во овој проект, лентите ќе бидат откриени со користење на canny и hough lines и алгоритам за филтрирање на трите ленти. Ќе ја имплементираме и невронската мрежа yolov5 за откривање возила на патот. Овој пајплајн во реално време ќе се имплементира на duckiebot за тестирање на перформансите на алгоритмот.*

# Клучни зборови: компјутерски вид, детекција на лента, автономно возење, yolov5

**Abstract**

# *Self driving cars are now a necessity in this world, which makes it absolutely important to have a sound algorithmic system that would perform detection of various features on the road, such as lane markers, detecting vehicles etc. In this project, lanes will be detected using canny and houg lines and an algorithm to filter out the three lanes. We will also implement the neural network yolov5 for detecting vehicles on the road. This real-time pipeline will be implemented on the duckiebot to test the performance of the algorithm.*

# Keywords: computer vision, lane detection, autonomous driving, yolov5

# Вовед

Главната цел на овој проект е да се создаде симулатор на роботска рака каде што можете лесно да се креира или вчита постоечки urdf и да управива со роботот во симулиран свет со генерирани пречки и предизвици. Симулаторот што се користи е pybullet кој доаѓа со сите потребни компоненти за изградба на обемен и комплициран симулиран свет. Основата за создавање на роботската рака е направена со sympy и сите алатки се извлечени од основите на роботиката. Исто така, постојат посветени пакети за градење на светот околу роботската рака и конвертирање на параметрите на Denavit–Hartenberg DH во urdf за да може да се увезат во симулацијата. Во делот robot\_arms/ има вистински раце како панда и ксарм. Исто така, тука е и раката Mitsubishi RV-2F-Q каде што програмата исто така може да ги конвертира дневниците во програмска датотека RT Toolbox и да ја изврши програмата на вистинската рака......

# Користени технологии и алатки

Софтверскиот пакет содржи три мегусебно поврзани библиотеки. Главната библиотека е pybullet кој е брз и лесен за употреба пакет за симулација на роботи и машинско учење, со фокус на трансфер од симулациијата во реланоста. Oбезбедува симулација на динамика нанапред, пресметување на инверзна динамика, напредна 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 логирање на сите зглобови.

Основата за создавање на роботската рака е направена со sympy која е библиотека на пајтон за симболична математика. Се користи за пресметка на DH матрицата и јакобијанот на раката. Третата бибилиотека е numpy која се користи за олеснување на операциите со матрици и низи во прокетот.

# Инсталација на софтверскиот пакет

## Инсталација на потребни апликации

Основни софтверски апликации кои ги употребува пакетот се пајтон со соодветен инсталер на бибилиотеки и веб пребарувач за користење на јупитер тетратките. Популарна алатка за инсталирање на бибилотеките која се користи е мини конда кој е бесплатен минимален инсталатер за конда. Чекорите за инсталација на апликацијата може да се следат во следниот линк од нивната документација [1].

Освен веб пребарувач, јупитер тетратките може да се отворат и со софтверскиот пакет Vscode каде имаме вградена екстензија за нивно вчитување. Како последен софтверски пакет кој е опционален е гит кој е бесплатен и дистрибуиран систем за контрола на верзии со отворен код. Со кој може лесно да се клонира проектот и понатаму да се надградува.

## Инсталација на библиотеки и клонирање на проектот

Потребни библиотеки кои проектот ги користи се pybulllet, sympy и numpy. Верзијата на пајтон нема потреба да се одбира бидејки пакетите за инсталирање на библиотеките автоматски ја одбира заедничката верзија за сите библиотеки. Со следните линии команди ќе се покажат чекорите потребни за колинирање на софтверскиот пакет, инсталирање на потребните библиотеки и стартување на јупитер лаб серверот:

1. Клонирање на рaкетот:

git clone <https://github.com/vasetrendafilov/robot-simulation>

1. Креирање на околината и инсталирање на портребните библиотеки:

conda create --name robot\_simulation -c conda-forge pybullet jupyterlab sympy

1. Активација на околината

conda activate robot\_simulation

1. Стартување на јупитер лаб за отварање на јупитер тетратките

jupyter lab

# Користење на софтверскиот пакет

Овој проект доаѓа со три јупитер тетратки каде што се имплементирани сите функционалности на софтверскиот пакет. Во следните подточки ќе се објаснат сите јупитер тетратки во детали со сите чекори како да се креира или импортира роботкса рака, контрола со неа и објаснување на сите модули на симулацијата.

## Модуларна роботкса рака

Може да се креираат модуларни роботски раце користејќи (DH - Denavit–Hartenberg) конвенцијата. Примери за контрола со нив и пресметка на DH матрица на роботската рака и линиарен и ангуларен јакобијан на роботсктата рака.

На почеток се полнат библиотеките sympy и numpy кои ќе се користат подоле за креирање на симболички променливи и олеснување при работа со матириците. Следно се импортираат 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

### 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 инијализира и се додаваат зглобови според DH конвецијата. Освен четирите основни параметри (тета, d, а, алфа), во функциите дополнително може да се сетираат долни и горни граници, максималната брзина и забрзување на згобот. Ако се додадат дополнителни симболички променливи, треба да се додаде на нив и нумеричка вредност со функцијата add\_subs(). На крај се полни 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)

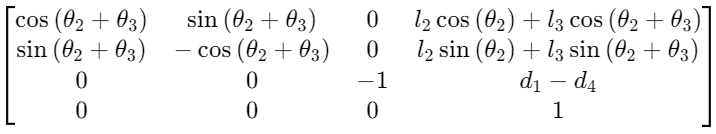
robot.add\_subs([(l2, 2), (l3, 2)])

robot.load\_robot\_arm()

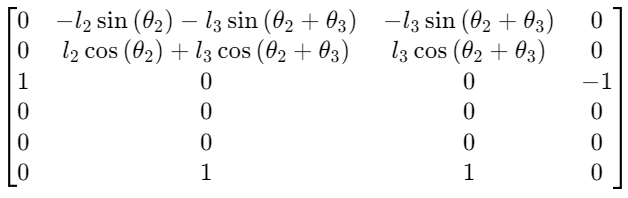
Со функцијата get\_dh\_matrix се пресметува дх матрцата за целата рака, со која се преставува зависноста на крајниот енд ефектор од позицијата на секој зглоб. Функцијата jacobian го пресметува линеарниот и англуарниот јакобијан за целата роботска рака.

robot.get\_dh\_**m**atrix()

robot.jacobian()



Слика 1 дх матрица



### Инверзна кинематика на роботска рака со присматичен грипер

Вториот пример креира модуларна роботка рака на која се поставува присматичен гриппер и интерактивно се контролира раката. Се користат истите чекори за креирање на модуларната рака како првиот пример. За додавање на гриперот се косити функцијата add\_attachment која по дефаулт се додава присматичниот гриппер. За крај се повикува главната функција interact() каде се креира интерактивно гуи во пубуллет за контрола на раката со инверзна кинематика каде по дефаулт се користи вградената нумерицка апроксимација.

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)

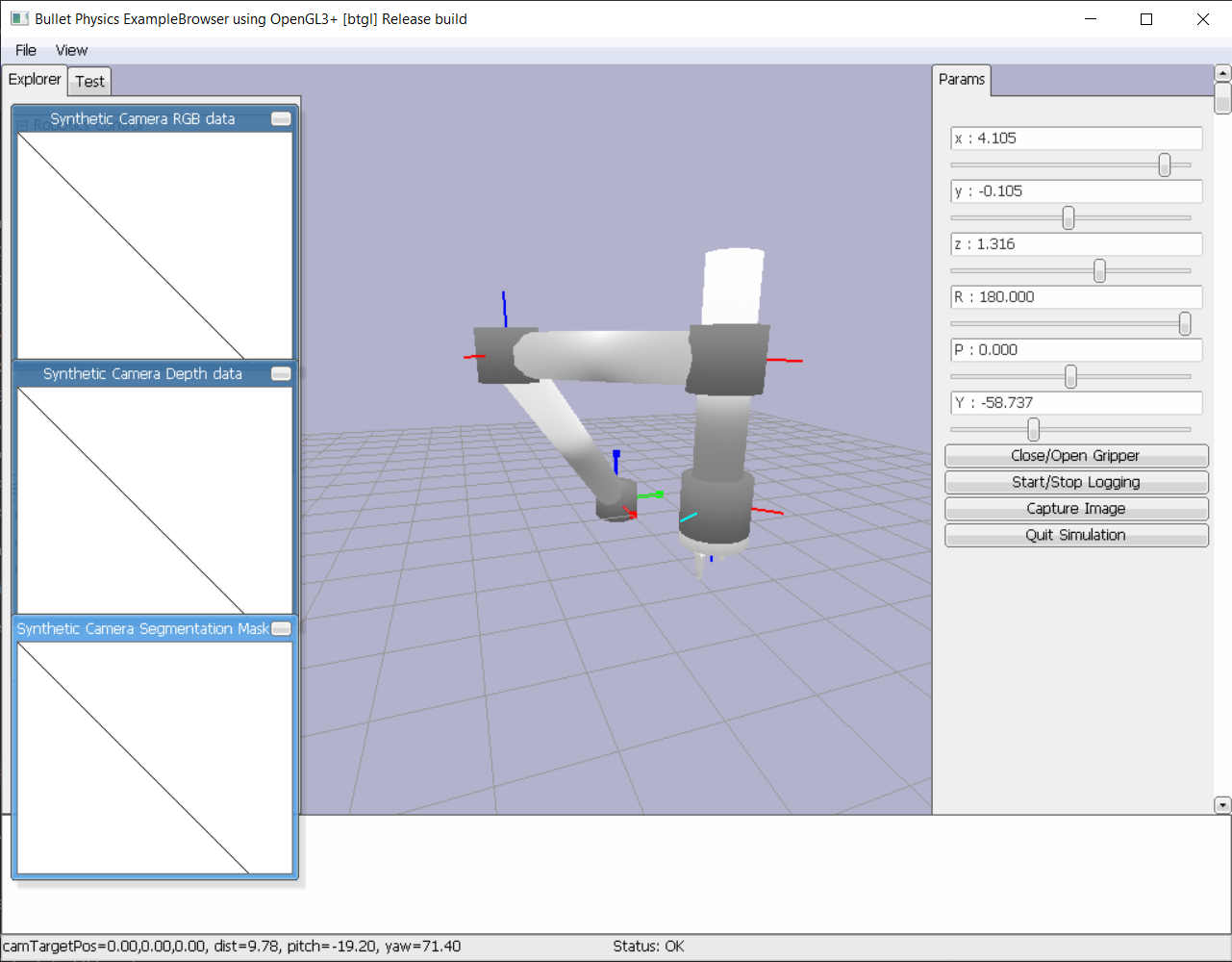
robot.add\_prismatic\_joint(0, d3, 0, 0)

robot.add\_revolute\_joint(theta4, 0, 0, 0)

robot.add\_subs([(d1, 3),(a1, 3),(a2, 3)])

robot.add\_attachment()

robot.interact('inverse')

Гуито на пѕбуллет се составува од главен прозорец каде се прикажува генерираниот свет, каде во моментов е само преставена рообтската рака. На десната страна се генерирани слајдери и копциња за контрола на раката. Кога се користи инверзна кинематика се поставуваат три слајдери за позиција на енд ефекторот по оските каде по дефаулт е од -5 до 5, и 3 слајдери за ориентацијата по конвенција RPY каде имаме ротации по x, y и z оски. Исто така имаме и копчиња за затворање и отворање на гриперот, да се запоцне и запре зачувањето на позициите на зглобовите, зимање на слика од енд ефекторот, и на крај исклучување на симулацијата.

Бројка 1

### Директна кинематика на роботска рака со револутен грипер

Следниот пример покажува контрола на модуларна роботска рака со револутен гриппер. Кога се додава нов додаток на раката треба да се постави оффсетот и ориентацијата од енд ефекторот и да се постават таргетите за точно колку да се отвора и затвора гриперот. Исто така имаме и случај при контрола на раката динамички имаме голема грешка во позицијата на зглобовите за ротарија. Тоа се дешава бидејки во примерот имаме многу зглобови кои ротираат во место и симулацијата неможе да го измоделира тоа. Има решениае и за тоа каде во римулацијата само ги ресетираме позициите на зглобовите и работи супер, но во вој мод неможе да се користи гриперот. На крај ја контролираме роботксата рака до директна кинематика кој е дефалтната опција.

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)

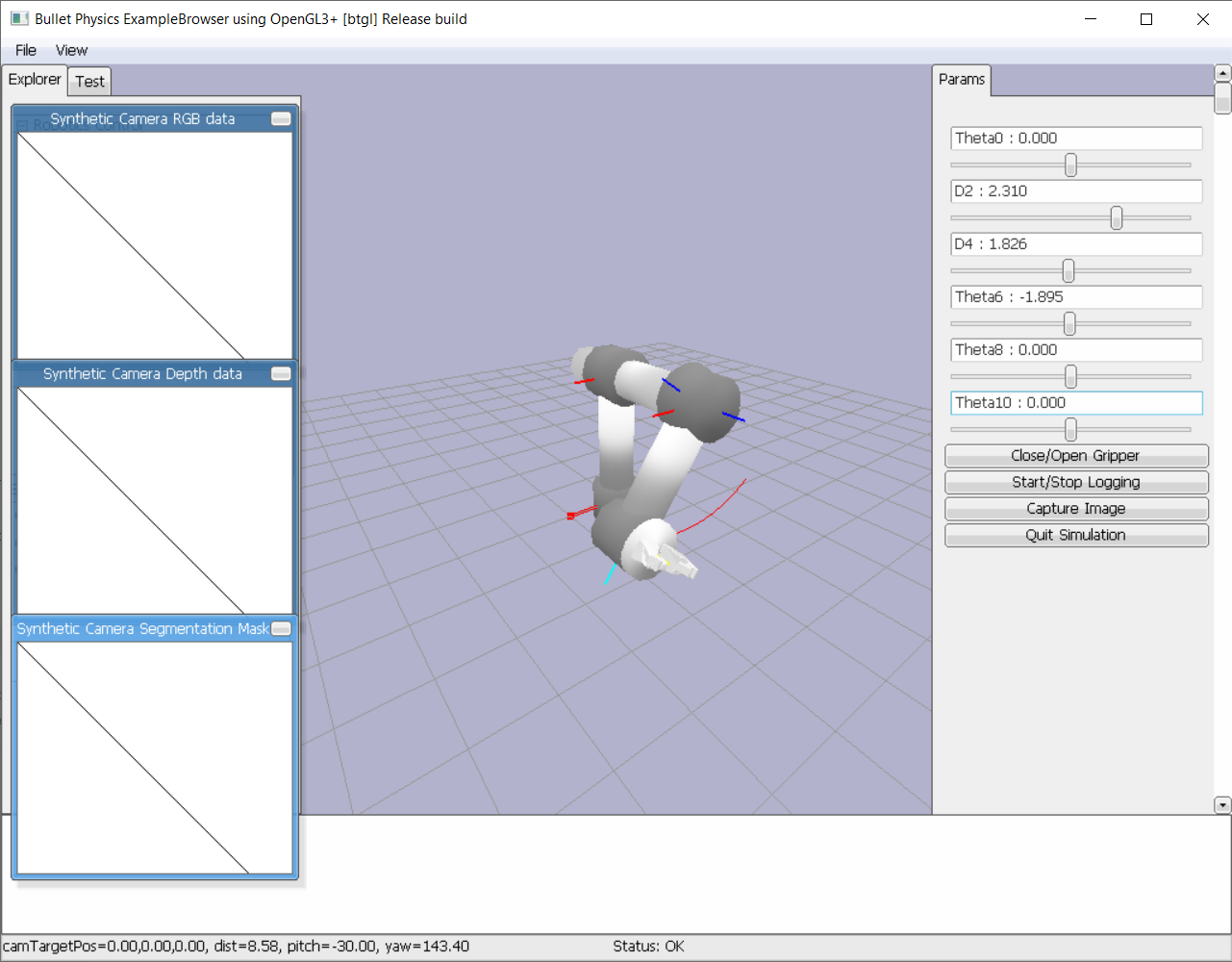
robot.add\_revolute\_joint(theta5, 0, 0, -sp.pi/2)

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 контрола на роботската рака со директна кинематика имаме мали промени, каде сајдерите за промена на посизијата и ориентација на ендефекрот се заменети со слајдер за секој зглоб во раката кој е во поставените гранизи на секој зглоб. Исто така на сликата се прикажува и траекторијата на енд ефецторот каде во вој случај се гледа како се однесува но нишало пробувајќи да ги постави згобовите на точната позиција.

### Импортирање на рака и репродуцирање на логовите

Следниот пример се прикажува како да се импортира веќе креиран робот кој е направен од проектот и како да се реплеј направените логови од раката. При реплејнување на логовите имаме опција да миниме низ секој чекор секвенцијално или да ообереме преку слајдер кој чекор да го помине.

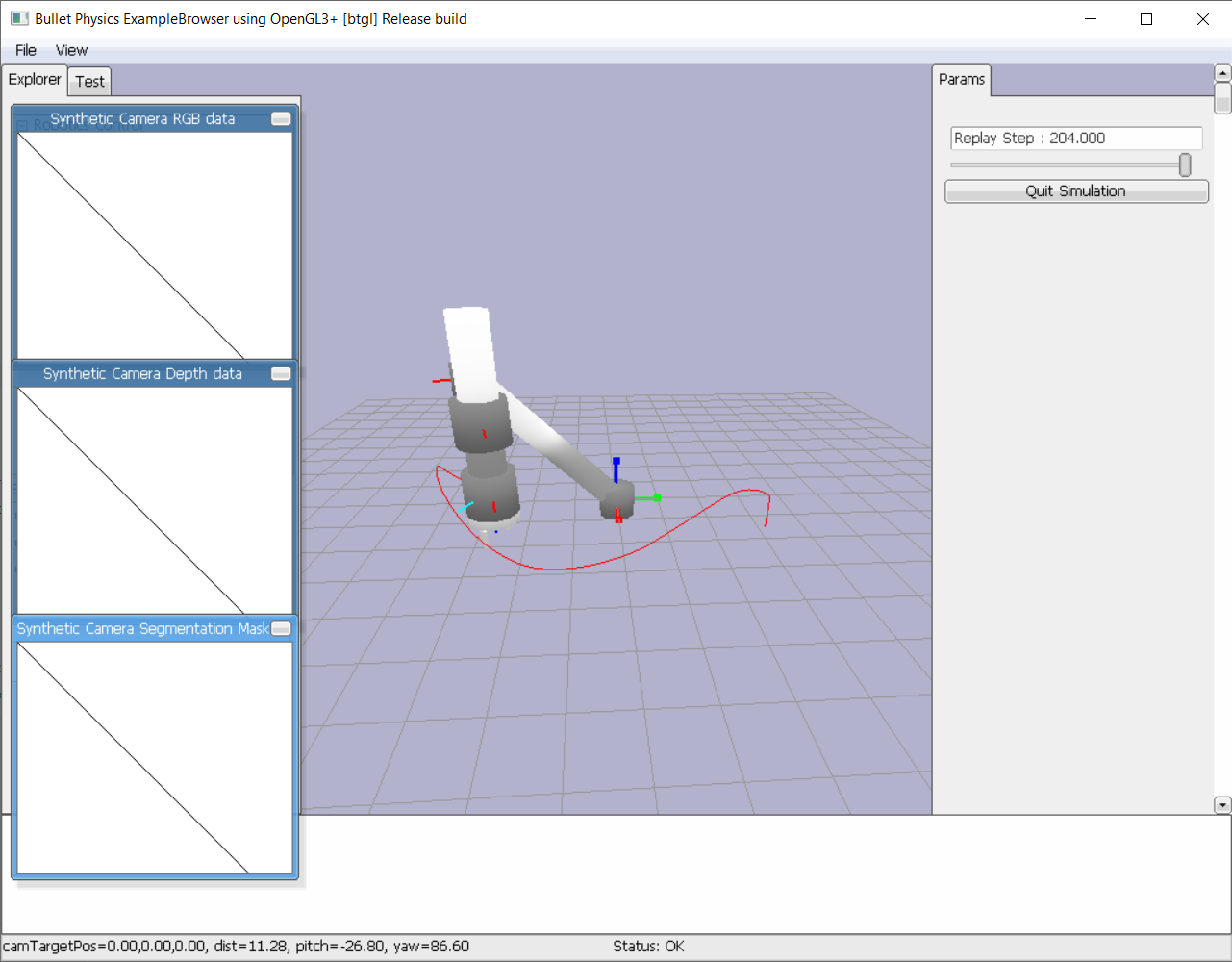
robot = RobotArm()

robot.import\_robot\_arm('robot\_arms/my\_robot/my\_robot.urdf')

robot.load\_robot\_arm()

robot.replay\_logs('inverse',skim\_trough=True)

Во сликата подоле е преставено гуито за реплејнување на секој чекор од логот.



### Пример за детекција и зимање на објект со грипер

Следниот пример е симпле пример за детекција на објект и зимање на објектот со грипперот на роботската рака. Прво се создава светот околу роботксата рака користејки ја каласата PybulletSimulation(), каде го стартуваме и конектираме со гуито на пѕбуллет. Го градиме светот така што поставуваме работна маса, и лего на случајан позиција на масата. Следно ја креираме роботската рака и ја поставуваме на едниот дел од масата. За да имаме прецизни двизења силата на зглобовите ја намалуваме на 400 и ги поставуваме динамичките цондиции да грешката во згобовите да е 10^-3 и да чека 1000 степс за да запре ако не успее да стигне до саканата позиција.

sim = PybulletSimulation()

sim.connect()

sim.load\_table()

# place a lego randomly on the 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')

# set the dynamic condition to have grater precision

robot.set\_dynamic\_conditions(1000,0.001)

robot.load\_robot\_arm()

Со следниот блок код се објаснува како работи краткиот алгоритам да го премознае објектот и да го фати. Прво енд ефекторот на роботската рака се поставува на центарот на масата и на највисоката позиција која може раката да ја постигне. Потоа се зима слика од енд ефекторот и се зголемува фовто за да имаме поширока слика. Ориентацијата на енд ефекотрот се помстува за 180 по џ оска бидејки цамерати зима слика од з оската нагоре. Пубиллет ни врака три типа од сликата, основната ргб слика, длабоцината на пикселите и слика со сегментација каде објектие се сегментирани посебно. За да проблемот биде олеснет се користи сегментацијата каде знаеме дека со вредност 1 е легото на сликата. Со помош на нумѕу се зима позицијата на пикселите касе вредноста е 1 и врзиме средниа по двете оски за да го најдеме центарот на објектот во сликата. Следно ја роунднуваме посизијата до најблискиот пиксел и со помос на вградената функција rgbd\_2\_world() ја конверираме позицијата од пиксели и длабочината во таа точка во кординати на симулацијата. Таргет поцизијата ја поместуваме по з за 0.75 за точно поставување на гриперот. Потоа енд ефекторот оди на објектот, се симиња по з над него и со actuate\_attachment го затвораме гриперот 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,capture\_now=True)

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]

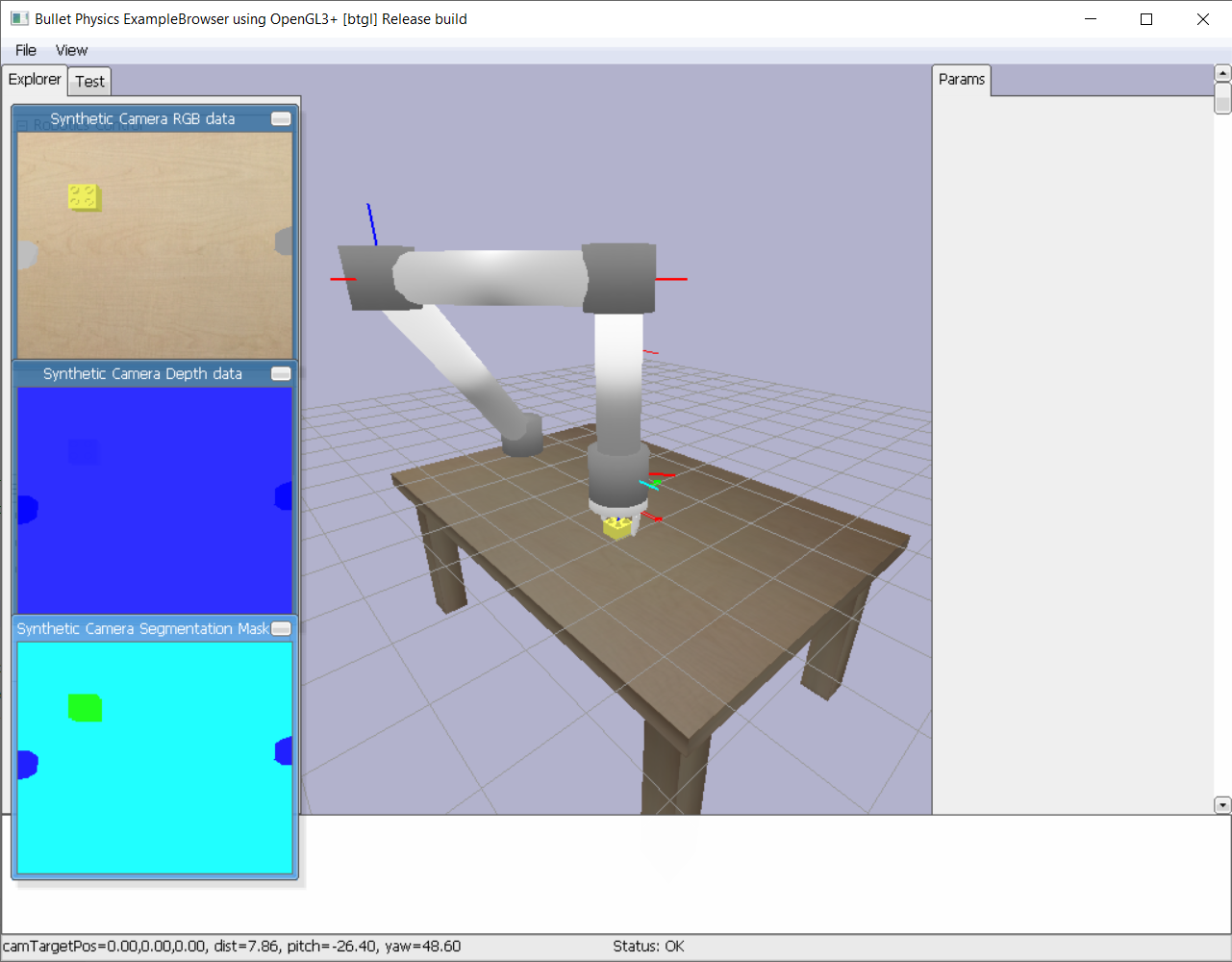
robot.move2point((target\_pos[0],target\_pos[1],2.5),(180,0,0))

robot.move2point(target\_pos,(180,0,0))

robot.actuate\_attachment(joint\_targets = robot.attachment\_close\_targets)

robot.move2point((0,0,2),(180,0,0))

Кога ке се земе слика од симулацијата од левата страна на гуи прозорецот се прекажуваат три прозорци со превју од земенеата слика.



## Популарни роботски раце

Со оваа јупитер тетратка ќе се разгледаат две примери од вистински роботски раце, како да се импортираат во пѕбуллет и контролираат. Се импортираат истите библиотеки и класи како прредходната тетратка.

### Franka Emirka panda роботска рака

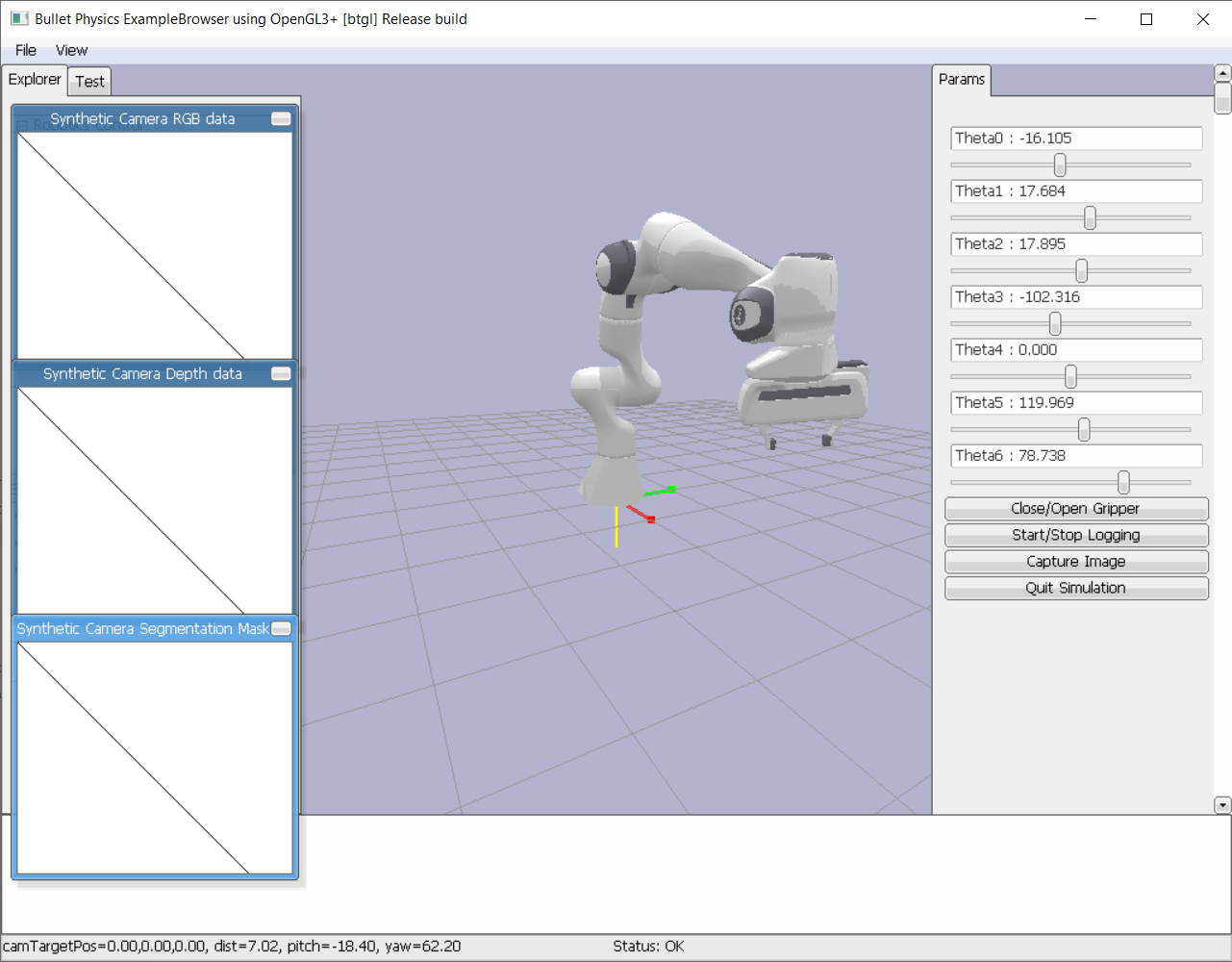
Прво е фрканка панда каде странските раце се импортираат преку import\_foreign\_robot() каде освен локацијата дополнително имаме и скалиранје за да се наместат димензии на раката. Исто така се поставуваат и таргетите за гриперот да се отвори и затвори.

robot = RobotArm()

robot.import\_foreign\_robot('robot\_arms/panda/panda.urdf')

robot.set\_attachment\_targets((0.2,0.2), (0,0))

robot.interact()



### xArm роботкса рака

Другата рака е од џарм и имамае повеке конфигурации, каде може да се видат во робот армс / џармс фолдерот. Прво креираме свет каде се полнат рандом објекти дадени од пѕбуллед и се наогаат во пѕбулеед дата . Потоа користиме инверзна кинематика и ја контролираме. Дополнително може да се постави и опсегот на слајдерите за позиција, во овој пример го поставуваме з да не оди во минус.

sim = PybulletSimulation()

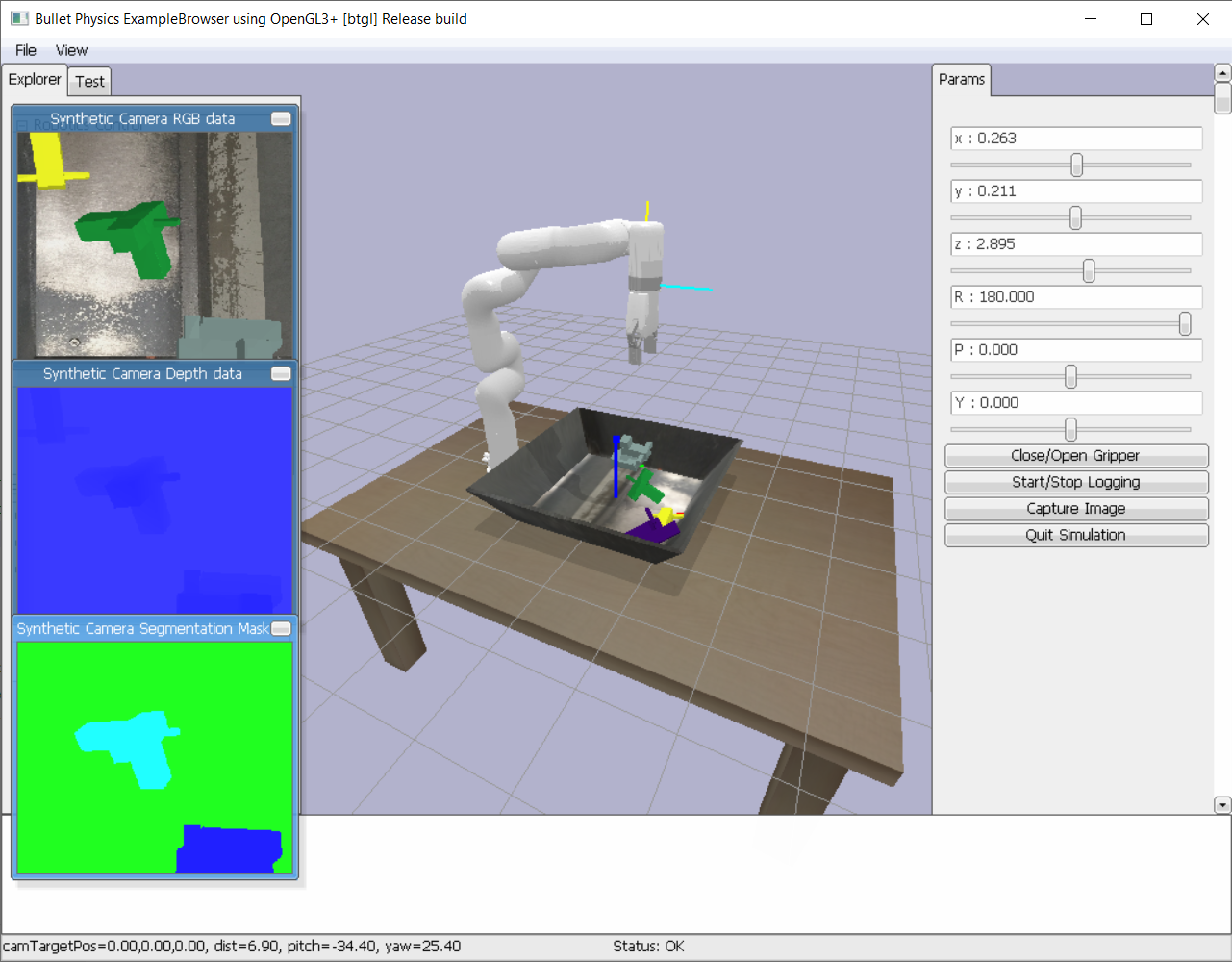
sim.load\_playground\_2()

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))



## Mitsubishi роботска рака

Со оваа јупитер тетратка ја симулираме раката RV-2F-Q каде плус креираке програма за контрола на вистинската рака. Креарани се две верзии на раката со и без гриперот, засега програмата разботи без грипер бидејки не е усте целосно имплементиран на раката.

### Контрола на Mitsubishi роботската рака со гриппер

Креираме свет во кој се полни маса траѕ и сите цоммон објекти дадени од пубуллет. Дополнително се поставува и оффсетот на камерата за да бите добро поставена од почетокот на грипперот.

sim = PybulletSimulation()

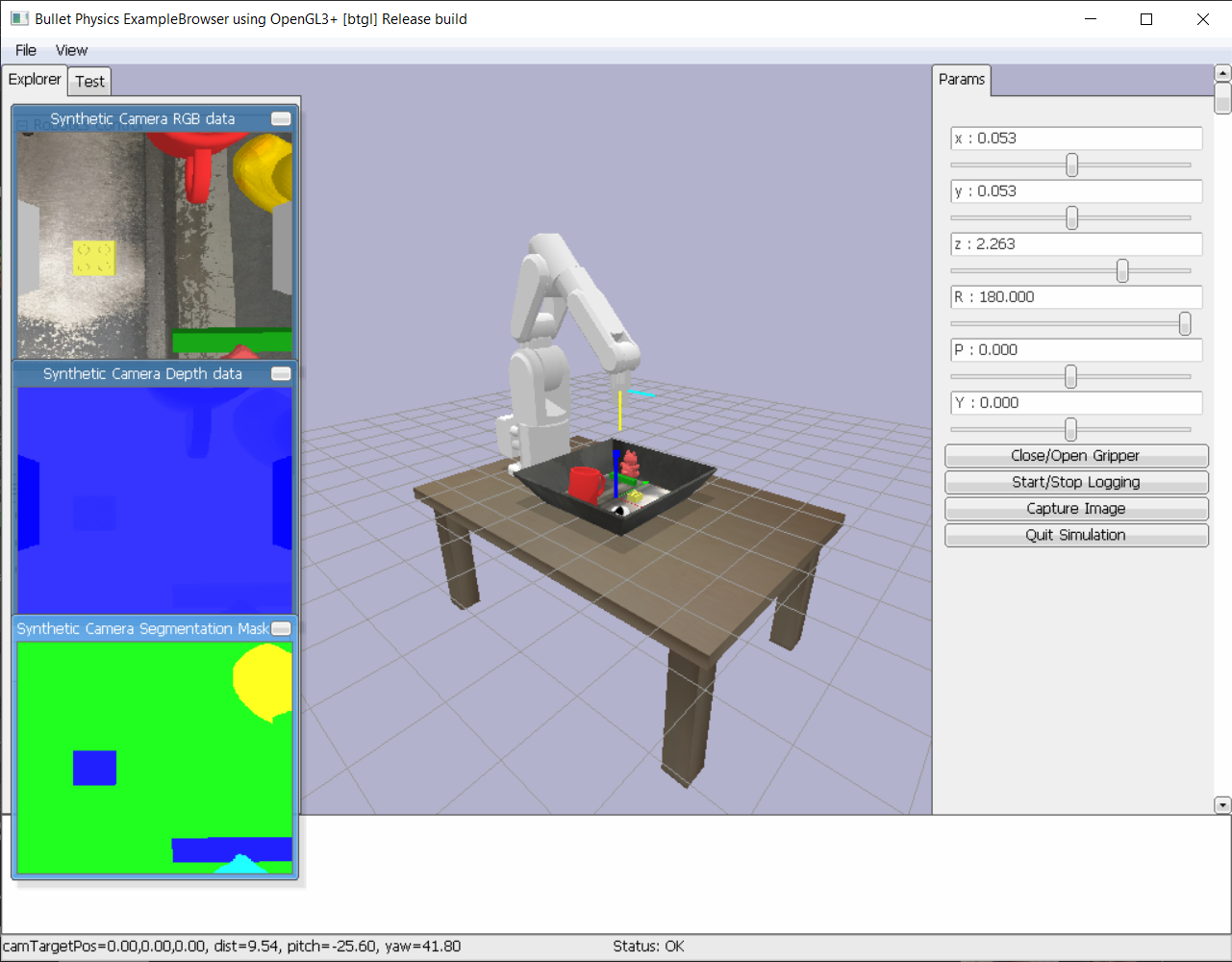
sim.load\_playground\_1()

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')



### Поднесување на параметри на раката за контрола преку комадни

Креираме свет кој точно го реплицира во реалноста, не користиме динамика бидејки ке го користиме креирање на програма и секој чекор ке биде прецизен нема да чекаме раката да стигне до саканата позиција. Многу важно е да се постават и лимитите за до каде може раката да оди, во вој слуцај да не удри во масата и самата себе. На крај се ресетираат збголобовите за да помогне при нумеричката инверзна кинематика бидејки ако почне од нулта позиција сака да дојде кон другото решение но неможе бидејки имаме лимити. Може и опционално да се сетира и колку линијата за траекторијата ке пиде 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

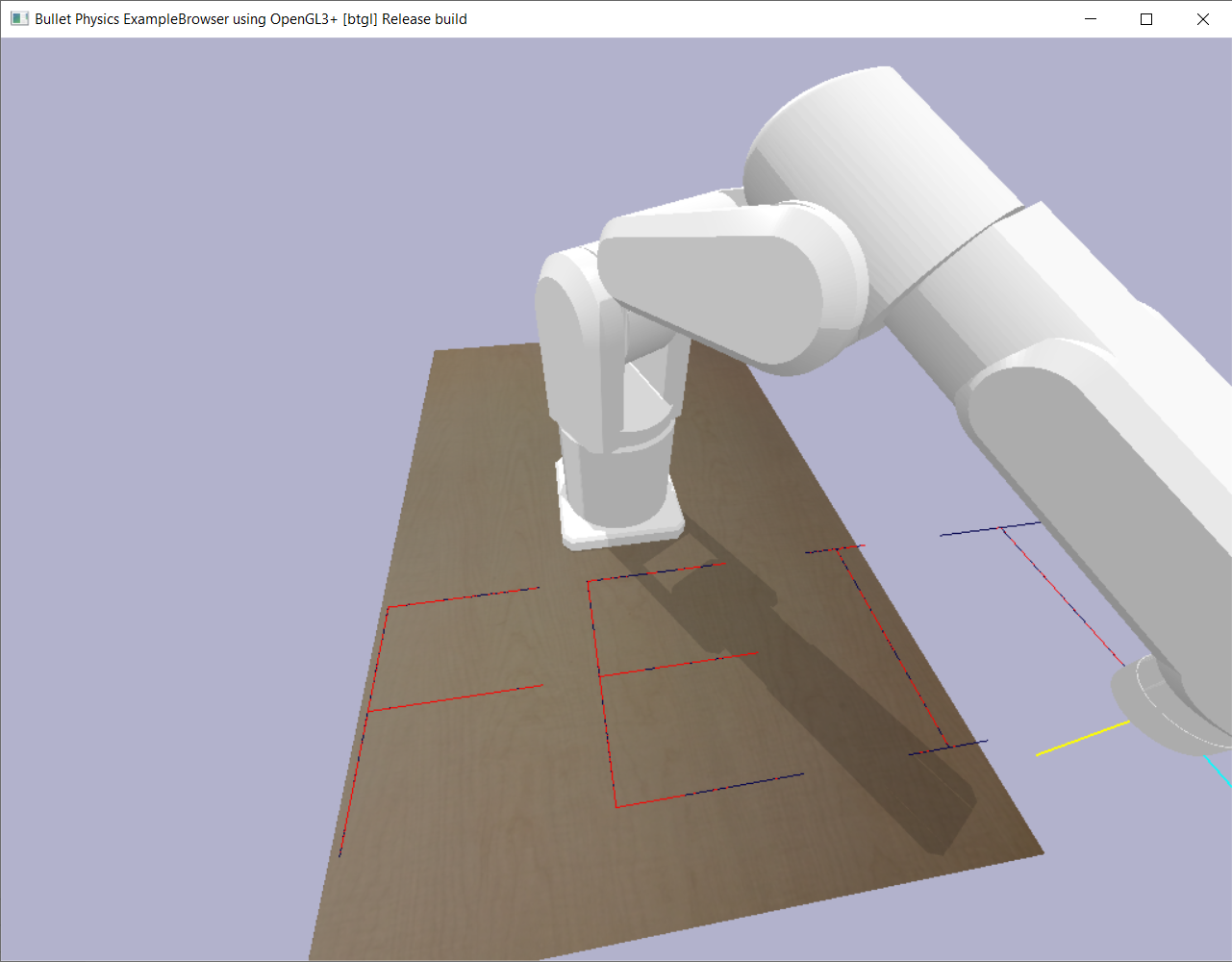
Следниот блок линии преставуваат сите опционалности како да се генерираат таргет точките. Најпросто движење е линеарното каде поставуваме почетна и крајна точка и посакуваната ориентациј која исто така може да се интерполира. Исто така воведено е и крижно интерполирање помеѓу двете точки каде дополнително 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, и од која старана да започне да црта. Исто така може да се да се сетираа и колку точки да се генерираат и рамнината на кои да се поректираат точките. На крај функцијата има дополнително и функција да започе со логирање на зглообовите. Како апстракција имам воведено функција да црта текст каде се користат движењата, засега има мал број на букви што се направени.

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)



За крај може да се проверат зачуваните логови и со функцијата convert\_logs\_to\_prg() да се генерира програма која потоа може да се изврши на раката.

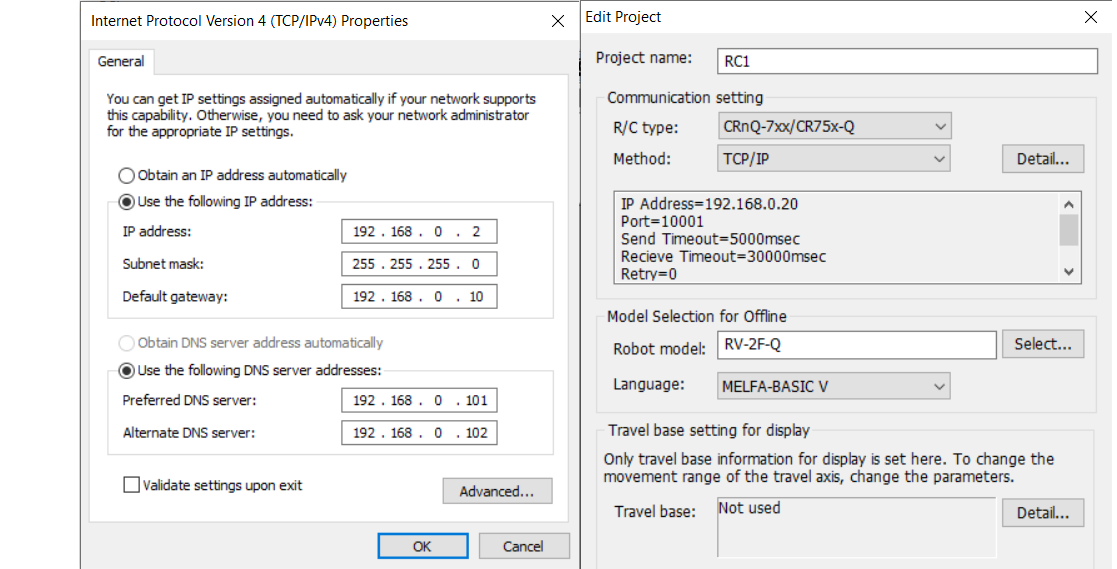
robot.replay\_logs('move\_circular')

robot.convert\_logs\_to\_prg('FEIT','feit')

## Кратки копчиња за pybullet

|  |  |  |
| --- | --- | --- |
| Акција | Операција | Инструкција |
| Отворете или затворете ги табовите 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 логовите 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 жичната 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/одзумирате приказ | mouseweel | Користете го глувчето за да зумирате и одзумирате |
| Зумирајте/одзумирате 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) |
| Зачувај „testFile.bullet“ | F3 | Притиснете го копчето F3 за да зачувате „testFile.bullet“ фајл од светот |

# Конектирање на CR750-Q со RT Toolbox 2

Комуникацијата помеѓу компјутерот и контролерот CR750-Q се врши преку етернет кабел. Кабелот е поврзан со plc интернет-портата на CR750-Q контролерот и во интернет портата на компјутерот. Параметрите на контролорот на роботот за TCP/IP комуникација се веќе поставени. 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 сликата 1неш. Во програмата RT Toolbox 2 се креира нов проект и се копираат параметрите од сликата подолу. Потоа од менито на RT Toolbox 2 се отвора Online, и како под опција се селектира пак Онлине со која програмата почиња да воспоставува комуникација со контролерот. Се појавува нов прозорец кој не информира за статусот на врската и ако се е подесено како треба, статусот на линијата треба да покаже конектирана и да свети сино.

# Архитектура на системот

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 подточки ке бидат објаснети комуникациите помегу класите и нивните функционалности.

## Хиерархија помеѓу класите

Главната класа на проектот е RobotArm, каде се извршуваат главните функционалности на проектот. Под функционалностите се поделени на:

* Функции за креирање на модуларна роботска рака
* Функции за контрола на раката преку интерактивно гуи
* Функции за контрола на раката преку команди
* Модули за работксата рака во симулацијата

Комунијацијата со PybulletSimulation е прилично мала бидејки таа е застапена за креирање на светот околку роботот. Се повикува автоматски при иницијализација ако сакаме да креираме празен свет. DH2Urdf се користи стрикно за конвертирање од DH конвенцијата при креирање на модуларниот робот во urdf фајл кој pybullet може да го прочита и да ја стави роботската рака во симулацијата. Посебниот пакет utils е превземен од друг проект е наменет за пресметување на DH матрицата и јакобијанот на раката. За крај се повикува и класата Camera со која се зима слика од симулацијата.

## 

## Хиерархија на класата PybulletSimulation

Хиерархија на PybulletSimulation е повеќе интерконектирана каде што повеќето функции завиcат помеѓу себе. При иницијализацијата се конфигурараат гравитацијата, параметрите на дегуг камерата, брзината на симулацијата и директориумот каде да ги бара урдф фајловите. Има воведено повеќе основни функции каде се полни работна маса, треј, сите обични објекти најдени во pybullet\_data и функција за полнење на рандом објекти. Како апстракција на на основите функции има воведено два прејдоундови каде полесно може да се повикаат повеже окбјекти заедно и да се направи светот.

## Хиерархија на класата Camera

Класата Camera е апстракција на функцијата getCameraImage во pybullet со која се олеснува поставувањето на погледот и сите параметри. При иницијализација се пресметуваат матриците за поглед и проекција на камерата со поставените параметри за широцина на видното поле, димензиите на сликата, флитер за опсег кој камерата може да го опфати. Исто така се пресметува и инверзната матрица за да се вратиме од пикселите во сликата во реални кординати во симулатниот свет. Со главната функција shot се зима слика од пресметаните матрици и се зачувува за понатомошна работа. Следните две функции се за конвертирање од пиксели во реалниот свет каде се користи инверзната матрица, дополнително треба да се постави и длабоцината од depth сликата за се конвертира точката во симулиранот свет.

## Хиерархија на класата DH2Urdf

Класата DH2Urdf е наменета за конвертирање од DH параметри во urdf фајл кој симулацијата може да го прочита. При иницијализацијата се превземаат DH парамтетрите, поставените лимити на зглобовите и алатот ако е поставен. Има поставено четири основни функции кои додаваат соодветен стринг 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 write\_joint се запушува блок код наречен јоинт 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. Зглобот преставува темно сив краток цилиндат, а линковите помегу нив се долги бели цилиндри кои се пресметени да бидат најкратката линија помегу нив. На крај генерираниот стринг се зачувува во фајл на поставениот директериум.

## Хиерархија на функцијата interact

Како што кажавме порано за класата RobotArm се дели на повеќе функционалности, тука ќе биде објаснето за контрола преку гуи на pybullet. Роот функција за оваа функционалност е interact со која имаме можност да користиме директна и инверзна кинематика каде дополнително може да се одбере дали да користи и ориентација. Првото што се прави при повик на функцијата е да се наполни роботската рака во симулацијата. Редоследот на извршување е прво се користи класата DH2URDF за да се конверита роботскат рака ако не е импортната и се полни во симулацијата. Потоа со помошните функции find\_joint\_ids идентификуваат сите згловови во раката, се додаваат фраме линии на секој зглоб со add\_joint\_frame\_lines ако не е надворешен и се ресетира секој зглоб на почетната позиција. Зависно од тоа каква кинематика е одбрано се додаваат слајдери за инверзна или директна кинематика во гуито со add\_joint\_sliders и add\_pose\_sliders. На крај се повикува функцијата step каде се чита од слајдерите за посакуваната позиција и ориентација или позиција на секој зглоб и се контролираат соодветните зглообви со помош на функциите доделени од pybullet. При крај од функцијата имаме и дел со модули кои помагаат при контрола и интеракција со роботската рака.

## Хиерархија на контрола преку команди

Контролата преку код во класата RobotArm е поделена на повеќе функции за подобра скалибилност. Основната функција е move2point со која се врши најосновната работа да ја придвижи раката до посакуваната точка и ориентација. Се врши истата контрола на зглобовите како погоре и се додаваат модулте за симулацијата. Во вој случај се полнат само draw\_trajectory и display\_pos\_and\_orn. Дополнително имаме и услов за запирање на лоопот. Ако се кодисти динамика се споредуваат сите зглобови дали грешката е поголема од поставената или се поминати одреден број на чекори во симулацијата бидејки можно е инверзната кинематика да нема решение, а ако не користиме динамика се ресетираат зглобовите и се брејкнува од лоопот после првиот чекор. Како апстракција на основната функција е move каде се врши интерполација и генерираните точки се извршуваат. Дополнително се воведени и две функции за цртање на букви каде се користи move. И двете функции write\_text и move имаат дополнителна опција за автопамтско логирање каде се користи функцијата state\_logging.

## Хиерархија за креирање на модуларна роботска рака

Конвенцијата за креирање на нова модуларна рака во класата RobotArm е превземена fundamentals-of-robotics каде се додадени доплнителни функционалности. За додавање на нов зглоб се наменети функциите add\_revolute\_joint и add\_prismatic\_joint , исто така додеден е и зглоб кој не мрда add\_fixed\_joint. Функциите дополнително имаат опција да се додатад и крајните опсези на позицијата на зглобот, максималното забрзување и максимална брзина. Функцијата add\_subs се користи ако поставуваме дополнителни симболички променливи во зглобовите, бидејки мора да има и нумеричка вредност. Останатите функции за пресметување на Дх матрицата get\_dh\_matrix и jacobian се директно превземени.

# Документација за кодот

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 подточки ќе се објаснат сите калси со соодветните функции во детали.

## DH2Urdf()

Еден начин да се претстави кинематиката на робот во формат URDF е прво да се одредат DH параметрите за роботот, а потоа да се користат овие параметри за да се специфицираат кинематичките врски помеѓу врските во URDF датотеката. Ова овозможува компактен и удобен начин да се дефинира кинематиката на робот на начин што е лесно разбирлив и може лесно да се обработи со софтвер за симулација на роботи.

### \_\_init\_\_(self, dh\_params, constraints, attachment = None)

Со иницијализирањето се зачувуаат внесените параметри dh\_params, constraints и attachment. Параметарот attachment е опционален бидејки не секоја рака мора да има и алат додадеон на неа. Дополнително имаме помошен dict со кој се мапираат зглобовите со идентифиционене број даден од pybullet. На крај го поставуваме xml стрингот на кој ке ја запишуваме раката.

self.joint\_names = {0: 'revolute', 1: 'prismatic', 4: 'fixed'}

self.dh\_params = dh\_params

self.constraints = constraints

self.attachment = attachment

self.xml = ''

### write\_visual\_shape(self, rpy, xyz, length, radius, material\_name)

Функцијата write\_visual\_shape е темплејт за доавање на блок за визуелниот дел од роботот. Како влез прима ориентацијата и позицијата на визуелниот објект, каде ориентацијата е во формат row, pitch и yaw. Понатаму се поставуваат должината и радиусот на цилидарот, и дополнително се поставува и името со која боја биде 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"

### write\_inertial(self, rpy, xyz, mass, inertia\_xxyyzz)

Оваа функција е наменета за запишување на инерцијалните вредноси на линкот, каде се поставува ориентацијата и позицијата на центарот на маса на објектот, неговата маса и инерзијата 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"

### write\_link(self, name, visual\_shapes, save\_visual = True)

Функцијата write\_link креира блок код со кој се формира линкот помегу два зголба.Името на линкот треба да биде уникатно. Како влез прима листа од визуелни објекти каде мине низ секој објект и го запшува користејки ја функцијата write\_visual\_shape. Допонително помеѓу блокот се додаваат и инерцијалните вредности, но да сега е комнтирано 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 возможо бидејќи ке има константно колизии помегу 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"

### write\_joint(self, name, joint\_type, parent, child, rpy, xyz, constraints = None)

Со овој темплејт се додава зглобот кој ги конектира два линка. Како главни влезни параметри се името кое треба да биде уникатно и типот на зглоб кој може да биде присматичен, револуционен или цврст. Дополнително треба да се рефренцираат кој е предходник и последник на зглобот и ориентацијата и позицијата на зглобот. Како за крај опционално дополнително се додава и таг за лимитите на зглобот ако се поставени како аргумент.

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"

### save\_urdf(self, file\_path)

За крај ја имаме главната функција каде се користат сите останати и се конвертират DH параметрите во urdf фајл кој се зачувува во дадениот дирекотрум. Со првиот блок код креираме коментар на парвите линии од фајлот каде се запишуваат сите вредности на DH параметрите и лимитите во табела кои служат за прз преглед како функционира роботската рака и понатаму се користат при импортирање на раката во симулацијата. Се корсити дополнителна листа за имињата на секоја колона од табелата и се креира формат за печатење сличен на листата на имиња. Потоа се користи лооп каде се мине низ сите DH парамтерти со соодветните лимити. 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 кои подоле ке бидат искористени. Следно имаме фор лооп 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" # robot arm colors

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 помеќи визуелниот дел на зглобот и линкот. При секој чекор со втората линија поставуваме линк кој преставува зглобот користејки 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 обој блок код се пресметува позицијата и ориентацијата на визуелниот дел од линкот кој ги поврзуа двата визуелни зглоба. Првите неколку линии се наметети за пресметување на векторот на позиција од надерните dh параметри кој се нормализира и се дели со два за да се најде центарот.

origins\_vector = np.array([self.dh\_params[i][3], 0 ,self.dh\_params[i][2]],dtype = float) # transform [i+1]

origins\_vector\_norm = np.linalg.norm(origins\_vector)

cyor = origins\_vector/2

rpy = (0, 0, 0)

Ако нормалнтион вектор е различен од 0 ( имаме поместувања по x и z) следат одредени пресметки каде го пресметуваме аголот од нултиот вектор кон векторот генериран од dj параметрите. На крај ротациите по оските се добиваат со векторско множење на аголоте со оските. Одкако се пресметаат позицијата и ориентацијата на линкот се гради визуелниот визуелнитот дел и се додава за линкот, додато ако моменталниот зглоб е присматичен се додава и уште еден визуелен објект кој мине низ присматичниот зглоб и е поврзан со предходниот 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 = R.from\_rotvec(angle \* axis).as\_euler('XYZ')

    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 зашињува пресметаниот линк со поставените лимити и зглоб кој ги поврзува линкот и предзодниот 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 алат на крајот од роботската рака, се креира нов зглоб кој го поврзува едн еффекторот со алатот, каде позицијата и ориентацијата се дадени. Според поставеното име на алатот го отвараме соодветниот фајл и копираме се што има во вој фајл. Одтука иде зошто треба граничниот зглоб да биде наречен attachment\_joint за правилно да се разделат зглобовите.

if self.attachment: # copy and join the attachment to the robot arm

    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()

Со последните линии код се затвора главниот таг и го зачувиваме стрингот во поставениот директориум.

self.xml += "</robot>\n"

# write the final string to the file

file = open(file\_path, "w")

file.write(self.xml)

file.close()

## Camera()

### \_\_init\_\_(self,cam\_target, distance, rpy, near, far, size, fov)

Со иницијализирањето се креира објект на камерата во тридимензиона околина каде се зачувуаат внесените параметри како димензиите на сликата на камерата, опсег за билската 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 во тридимензионалниот свет.

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)

### shot(self)

Функцијата е апстракција на функцијата getCameraImage обезбедена 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 растојание и слика за сегментација на објеките се зачувуваат за понатомошно 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

### rgbd\_2\_world(self, w, h, d)

Со оваа функција се конвертира позицијата на пикселот во сликата и неговото растојание од камерата во кординати на симулацијата користијки ја инверзната матрица. Во функијата rgbd\_2\_world\_batch имаме истата функционалност, но нега на влез ја имаме целата слика за длабочина и во овој случај ги конвертираме сите точки наеднаш.

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

## PybulletSimulation()

### \_\_init\_\_(self, connection\_mode, fps, gravity, cam\_param, cam\_target)

При иницализација ги зачувуваме сите поставени параметри, и едена помошна променлига во кој е складирано знаме за користење на кеширани податоци во pybullet. Потоа се конектираме со графичката симулација од pybullet и ако е успешно ја конфигурираме 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()

### configure(self)

При конфигурацијата на симулацијата го поставуваме 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 од каде сите објекти во класата се превземени. На крај се изменува позицијата на светлото и се ресетира ориентацијата и позицијата на целта 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)

### load\_any\_object()

Останатите функции за полнење во класата се едноставни и преставуваат апстракција на функцијата loadURDF од pybullet. 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 додаено и функции load\_playground\_x кои ги инкорперииат сите основни функции погоре и се користат за полесно креирање на светот околу роботската рака.

### 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)

## RobotArm()

### \_\_init\_\_(self, params)

Иницијализацијата за роботската рака е прилично голема, каде се иницијализираат сите помошни променливи, и зачувуваат поставените параметри. Иницијалните парамтерти 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 тректоријата на едн ефекторот роботската рака
* use\_display\_pos\_and\_orn: се користи како знаме дали да се пешати моменталната позиција и ориентација на енд ефекторот на раката

На крај проверуваме дали имаме воспоставено конекција со симулацијата на pybullet и ако нема креаираме 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

### add\_pose\_sliders(self, x\_range, y\_range, z\_range)

Со функцијата се креираат сајдери за одбирање на позицијата и ориентацијата на енд ефекторот на раката. Сајдерите 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. Слајдер се додава со функцијата addUserDebugParameter од pybullet и се појавува на десната страна од гуито 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))

### add\_joint\_sliders(self)

Со функцијата се креира слајдер за секој зглоб на раката. Прво се проверува дали зглобот е присматичен или револутен:

* Ако е присматичен се креира слајдер каде осегот е опсегот на зглобот и почетна позиција е долната граница на зглобот, дополнително се довава и знаме во листата за да се разликуваат зглобовите.
* Ако е револутен се креира слајдер 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)))

### find\_joint\_ids(self)

Со оваа функција се мине низ секој зглоб на раката и се зачувува идентификационот број 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 присматилен или револутен. 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 лимитите 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

### 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 револутен 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)

### load\_robot\_arm(self)

Главна функција со која се внесуа 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 ракта 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)

Одкако се напони раката се пронајдуваат зглобовите, се додаваат фрејм линии на нив и се ресетираат на почетните позиции. Дополнително се конверитра 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 гриперот 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)

### interact(self, kinematics, use\_orientation\_ik, x\_range, y\_range, z\_range)

Една од главните функции каде ни дава можнос да ја упвавуваме раката преку гуито. Како влезни параметри имаме да обереме типот на кинематика и дали да се корисити ориентацијта при калкулација на инверзната кинематика. Дополнително можат да се постават и опсезите на слајдерите 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 слајдери да се додатат 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()

### 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 корисити, се читаат вредностите на слајдерите со readUserDebugParameter и се зачувуваат во joint\_targets ако користиме директна кинематиа или во position и orientation за инверзна кинематика. Дополинтелно за ориентацијата се конвертира вредноста на слајдерот од сетепи во радијани и трите алги се конвертираат во кватренион 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 со која се корсити 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 каде користиме управување на позицијата и се поставуваат идентификувационите броеви на зглобовите, посакуваните позиции и максималната сила за секој зглоб. Ако не користиме динамика, минеме низ секој зглоб и го ресетираме на посакуванта 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 отварање и затварање на грипперот, логирање на зглобовите, зимање на слика од едн ефекторот, и стопирање на симулацијата. Исто така ја цртаме траекторијата на едн ефекторот на раката и принтаме моменталната позиција и ориентација ако се одобрени. На крајот од циклусот поставуваме динамичко спиење на програмата, мериме колку време е поминато извршувајки го кодот и го одзимаме 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)

### display\_pos\_and\_orn(self, position, orientation, link\_id)

Во оваа функција се користи addUserDebugText од pybullet за печатење на на позицијата и ориентацијата во текс во симулацијата до поставенито лик ид. Прво се разледува позицијата и конвертира ориентацијата од кватреноне во еулер и посота 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 препришување 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)

### draw\_trajectory(self, current, target=None, life\_time=15, line\_index=0)

Функцијата ја црта моменталната и посакуваната траекторија со додавање на линија помеѓу моменталтанта и предходната точка. Содржи три листи во кои се чуваат предходните точки во листите prev\_pose\_1 и prev\_pose\_2 и листа has\_prev\_pose каде се зачувува дали имаме предходна точка и исто така се корисит за ресетирање на цртањето. Има опционално да се додате и линија за посакуваната траекторија и да се одбере идексот на линијата, бидејки може да се цртаат многу линии истовремено.

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

### 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 копче при првиот повик се додава копче во симулацијата со име Quit Simulation. Следно условот за дисконектирање од симулацијата е ако не користиме гуи 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

### 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 грипер. Како влез се поставуваат кои зглобови дакаме 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 контрола преку копче каде при првиот повик се креира. При секој клик на копчето се зглолемува вредноста за еден кога ќе се прочита објектот и со тоа алтенираме помеѓу отворената и затворената позиција на гриперот.

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

Со овој блок код динамички се управува гриперот. Ако е отворен се поставува силата на 500 за да не се помести при помрднување на раката. Кога гриперот ке 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 од pybullet зимаме информација за секој зглоб колку е моменталната применета сила и проверуваме дали е повеќе од 100. Ако секој зглоб има надминато лимитот поставуваме знаме дека згобовите се запрени и треба да се смени максималната сила на 50. Сето воа е прави бидејќи ако го притискаме објектот со сила повеќе од 100 константно, почнуваат внатреши колизии кои ќе го исфрлат објектот исто како во реалниот свет. Ако се постават посакувани позиции на влез на функцијата тогаш влегуваме во мод за контрола на раката преку команди. Кодот тука е идентичен, едиснствена ралзика е тоа што имаме циклус со 30 чекори за динамички да зе затвори или отвори гриперот.

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))

### capture\_image(self, link\_state, camera\_offset, near, far, size, fov, use\_gui)

Функцијата capture\_image е аптстакција на класата Camera каде ја прикачува на енд ефекторот на раката. Со аргументот link\_state се одбира на кој линк од раката да се прикачи камерата, по дефаулт се прикачува на последниот линк. Доболнителен аргумент е и офсетот од линкот за точно да се намести камерата бидејчи можно е да биде и внатре во раката. Следните параметри се користат за поставување на параметриме на камерата. Исто така функцијата има и два модови за користење, преку копче или директно преку повик на функцијата. Ако се конттолира преку копче дополнително се додава копче во симулацијата и при секој клик знаменцето use\_gui се става невистинито за да велземе во делот од кодот за зимање на слика.

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 почнуваме од позицијата на линкот и го офсетнуваме за поставениот аргумент camera\_offset. За да имаме точен офсет по сите оски дополително треба офсетот да го ротираме на истата ориентација како линкот. Ориентацијата за камерата се добива од ориентацијата на последниот линк, каде дополнително треба да се претворат од радијани во степени. На крај се зима слика и се враќа објектот за понатамоњна 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

### state\_logging(self, log\_name='', start\_stop=None, object\_ids=None)

Оваа функција е наменета за логирање на секој објект во симулацијата. Со листата object\_ids се одбираат кои објекти да се логираат, стандардно се одбира само раката. Ако не се користи аргументот strat\_stop, се креира копче кое со кое се започнува и завршува со логирањето на раката. Дополнително ако не постои патеката /logs/ се креира и сеите датотеки се зачувуваат во таа патека, и на крајот од името на датокетака се додава број на логирање кое се корисити за понатомошно сортирање на фајловите.

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:

Истото се користи и за користиње на функцијата и преку код, едниствена разлика е што кодот е поедноставен и нема потрепа од дополнителни услови за да не започнеме со логирање 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)

### read\_log\_file(self, file\_path)

Функција превземена од pybullet која служи за читање на датотеките логирање и зачувување за вердностите во листа која на крајот се враќа.

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

### autocomplete\_logs(self, log\_names)

Се користи за пронајдување на сите слични фајлови користејки го glob. На влез се прима листа од имиња на датотеки за логирања, но може и да биде едено име кој како исклучок го ковертираме во листа со еден елемент. Следно миниме низ сите имиња и ги пронајдуваме сите фајлови со слични имиња кои се сортираат по последниот индекс на запишување. Откако се сортираат се чита пронајдениот фајл и се додава на листата од логови 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

### replay\_logs(self, log\_names, skim\_trough=True, object\_ids=None)

Функција која ги користи сите горни функции за логирање и се врши рекреација на секој извршен чекор на раката. Стандардно се користи само идито на раката и се генерира листа од пронајдените логови и ако немаме завршува функцијата. Имаме дополнителна опција за минеме низ чекорите со слајдер или секвензијално 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 користиме слајдер секој циклус читаме од слајдерот кој индекс од листата да земеме, инаку индексот се зголемува за еден. Следно ја ресетираме позицијата и ориентацијата на основата на раката и креираме циклус кој мине низ секој зглоб на раката. Се зимаат сите информации за зглобот и ако не е фиксен користејки динамика или само да се ресетира 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 енд ефекторот на раката. На крај имаме двоен услов каде ако го користиме слајдерот преба да ја запреме симулацијата преку копчето за исклучување или ако се користи секвенцијалната опција ќе заврзи кога ќе се стигне до последниот лог 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)

### convert\_logs\_to\_prg(self, log\_names, file\_name)

Со оваа функција се конвертираат логираните датотеки на рака mitshubisi во програма која софтверот RT Toolbox 2 може да го разбре. Прво се креира листа од логирања и ако нема пронајдено функцијата запира. Потоа се поставуваат првите линии код со кои се уклучуваат сервата ,се поставува брзината и се креира циклус кој мине низ секој лог од низата и ги поместува зглобовите на посакуваните позициии. Низата се креира на крајот од програмата каде се додаваат променливите во конвенција jStep(x) = (j1,j2,j3,j4,j5,j6) и на крај се зачувува генерираниот стринг 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()

### import\_robot\_arm(self, file\_path)

Функцијата се користи за внесување на креирана роботска рака од овој проект. Се поставува знамето дека робот е венесн, и се сменува името и патеката до дадотеката. Следно се читаат првите линии од датотеката каде се зачувани во коментар DH параметрите и лимитите 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 деминатор од празна линија и се претвораат сите елементи во реален број. Кога ќе се доиде на линијата со која завршува коментарот, циклусот завршува.

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]),float(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\_list[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]))

### import\_foreign\_robot\_arm(self, file\_path, scaling=5)

За внесување на надворешни роботски раце дополнително се поставува и знамето is\_foreign за да знаае останаиот дел од програмата дека внесуваме надворешен робот. Дополнително се поставува и скалирањето, и се зачувуваат името и пакетата 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]

### write\_text(self, text, position, orientation, spacing, plane, scale, log\_text)

Функцијата write\_text е најголемата апстракција од управувањето на раката преку код. Со неа се црта текс во симулацијата, кој може дополнително моментално да се логира 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 одалечувањето помеѓу буквите и нивната големиа.

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')

### 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 разледуваат позицијата и орентацијата и зависно од која буква е внесена се извршува кодот внатре во еден од условите. Подоле 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)

### move(self, start, end, plane, interpolation, steps, param, closed, log\_move)

Функција со која се пресметува линеарна и крижна интерполација помеѓу две точки. При секој повик прво се ресетира цртањето на траекторијата и се разледулваат почетната и крајната точка за позиција и орентација. Ако дополнително се логира линијата, прво се поместува раката на посакуваната позиција, се ресетира цртањето на траекторијата и се започнува со логирањето. Со помош на функцијата rotation\_matrix\_from\_euler\_angles се креира ротациска 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 ја користиме функцијата linspace од numpy која интерполира помеѓу точките со одреден број. 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 e кружна дополнително треба да се додатат 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 резлутато не е имагинарен се породолжува со наогање на почетниот и крајниот агло 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 мењаваат почетниот и крајниот агол да се нацрта целиот круг. Следно зависно од која страна сакаме да се нацрта елипсата ги конвертираме почетниот и крајниот алог во интерплирана 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 plane матрицата. Кога ќе заврси со сите генерирани точки се запира логирањето ако е одобрено.

    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')

### move2point(self, position, orientation)

Функијата е идентична како подогрната step, во која се управува раката. Во овој случај се тргнати копчињата бидејки се контролира раката преку код и имаме дополнителен услов за запирање на циклусот кога се стигне до посатуваната позиција и ориентација 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 провеува со фунцкијата allclose од numpy колку се блиску сите позициите на сите зглобови до посакуваните со зададен праг или ако се поминати одреден број на чекори оти 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

### limit\_pos\_target() and limit\_joint\_targets()

Кратки функции во кои се лимитираат позициите на зглобовите и позицијата во тридимензиналниот 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 функцијта set\_position\_limits.

### get and set params()

Има дополнителни функции кои помагаат при сетирање на параметрите на раката. Првата е set\_dynamic\_conditions каде се поставува максималната грешка во зглобовите и максималниот број на чекори на исзвршување. Следна е set\_attachment\_targets со која се поставуваат посакуваните позиции на гриперот кога е отворен и затворен. Со set\_position\_limits се поставуваат минималната и макималната оскина позиција која раката може да ја достигне. На крај е функцијата get\_joint\_states која ги враќа информациите за секој зглоб во раката.

### add joints()

Со фунцкиите 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 допонително се поставуваат и долната и горната граница на зглобот, максилната брзина и забрзување. Сите вие параметри се зачувуваат во соодветните листи, каде за фиксниот зглоб имаме исклучоци во тоа што се креира сличајна симполичка променлива и не се додаваат oграничувања.

### get\_dh\_joint\_to\_joint(self, start\_joint, end\_joint)

Се користи за пресметка на dh матрицата од поставениот стартен и краен зглоб. 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 дкка. 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

### jacobian(self)

Корискетјќи sumpy се пресметува јакобијанот на роботската рака и се дели на линеарен и ангуларен јакобиан. Линеарниот се добива преку пресметка на dh матрицата и пресметување јакобијан на матрицата со сите симболички променливи.

linear\_jacobian = self.get\_dh\_matrix()[:3, 3].jacobian(self.joint\_variables)

Ангуларниот јакобијан се пресметува така што вертикално се надодава матрица со нули ако зглобот е призматичен или се зимаат првите 3 елементи од третиот ред на dh матрицата. Дополнително ако зглобот е фиксиран се прескотнува додавање на нов ред во матрицата на ангуларниот јакобијан.

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)

# 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 алакти и вистински роботски раце, исто така да се додадат и нови објекни во симулацијата. Има и многу повеке функционалности што може да се наградат како и примери користејки ја симулацајата. Во следните подточки ќе биде објаснето во детали како да се имплементираат новите функционалности.

## Додавање на нова алатка

За да се креира нов алат што ќе се додаде на роботкстака рака, треба да се следи следната конвенција. Се креира нова папка со името на алатот во папката /attachments и се ставаат сите датотеки во неa. 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 алатот е компатибилен со функцијата actuate\_attachment(), засега функцијата е кодирана за отворање и затворање на грипер. За слични грипери, може само да се искористи функијата set\_attachment\_targets() со која се поставуваат саканите позииции на гриперот кога е отворен и затворен.

## Додавање на новa роботска рака

Кога се додава нова роботка рака треба да се провери дали патеката на визуелните датотеки е правилно венсени во главниот фајл. Важно е да се остранат и сите xacro макроа бидејки работат со pybullet, а има многу раце така конфигурарано онлајн за олеснување при работа со нив. Има два примери panda и xarm во директориумот 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 згобови на роботската рака.

## Креирање на нов свет во pybullet

Користејќи ја класата PybulletSimulation, може да се направи нова функции за додавање објекти во симулацијата. Засега само се користат urdf датотеките обезбедени од pybullet каде се направени основни објекти како работна маса, треј и сите вообичаени објекти пронајдени во директоријата pybullet\_data.

## 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 проек е и отворен за сите за да може да се продолжи со неговиот девелопмент и сите придонесѕвања се добредоени. Имаме поставено листа од можни имплементации што уште не се имплементирани:

* 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 контрола на мобилни роботи
* Нова класа за креирање и конторла на квадропед робот
* Директна комуникација со CR750-Q

## Нови примери користејки го софтверскиот пакет

Со трите јупитер тетратки воглавно се поминати сите основни функционалности на софтверскиот пакет, но има многу моножноски за креирање на многу покомплицирани сценарија. Подоле е поставено листа на можни имплементации на нови примери:

* Покомплициран систем за детекција и зимање на објекти
* Имплементација на мобилана роботкса рака
* Имплементирање на tossingbot
* Користење на машинско учење за контрола на раката

# Заклучок

# Референци

1. <https://docs.conda.io/en/latest/miniconda.html>