Vaja 3: Robotski manipulator

Vsako od vaj zagovarjate pri asistentu. Za uspešen zagovor morate rešiti vse naloge, razen tistih, ki so označene z ★. Če katera od nalog zahteva skiciranje ali ročno računanje, morate to pokazati na zagovoru. Programerske naloge morate rešiti samostojno in izvorno kodo ter rezultate predstaviti na zagovoru. Če kode, ki rešuje vaše naloge, ne boste poznali v detajle, lahko asistent vaš zagovor zavrne. V primeru zavrnitve zagovora, ga lahko ponovno opravljate z zamudo, a bo zato vaše končno število točk kaznovano s faktorjem **0.75**.

Za rešen obvezen del nalog lahko dobite največ **75** točk, dodatne naloge pa vam lahko prinesejo do **25** dodatnih točk. Za pomanjkljive rešitve ali slabo teoretično znanje vam asistent lahko odšteje točke.

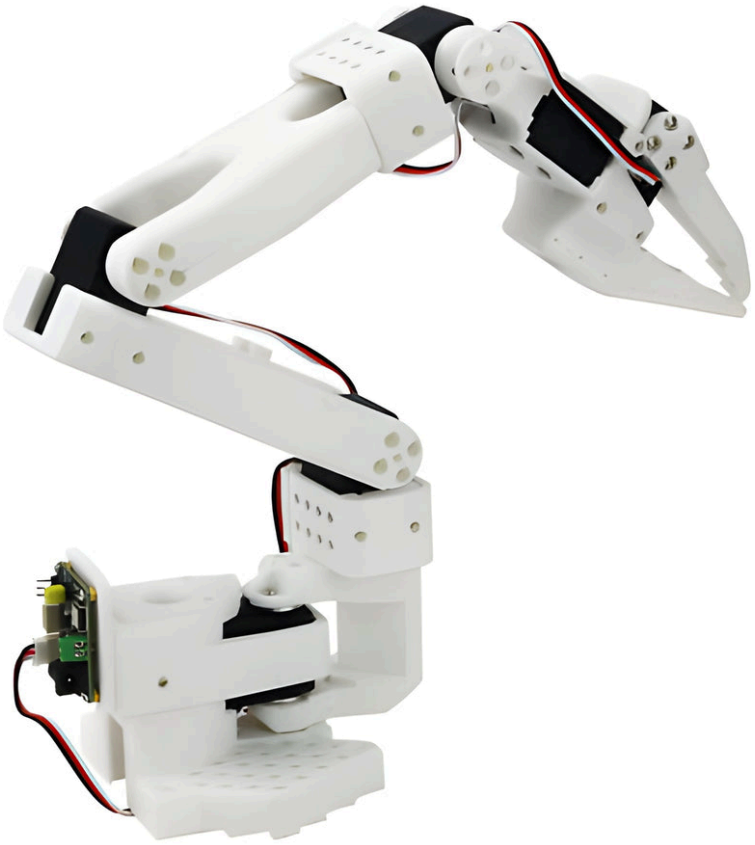
Pogoste napake in nasveti
- Zamenjava x in y osi
- Napačen podatkovni tip: `float`, `uint8`
- Napačno definicijsko območje podatkov: [0,255], [0,1]
- Rešitev razvijajte na enostavnih primerih (lahko so sintetični)

---

1. Robotski manipulator

V laboratorijski učilnici imate na voljo roke SO-101, ki jih boste uporabljali pri praktičnem delu te naloge. Roka SO-101 je sestavljena iz šestih rotacijskih sklepov, od tega jih prvih pet določa pozicijo aktuatorja, zadnji pa določa odprtost prijemala. Roka je zasnovana odprtokodno, vsi deli so 3D tiskani in njihovi modeli so javno dostopni, isto velja za kodo, ki se uporablja za nadzor roke. Koda je dostopna na povezavi.

Za delo z robotom boste potrebovali naslednje knjižnice:
- `lerobot`
  - Omogoča nastavitve motorjev in implementira komunikacijo z robotom.
- `ikpy`
  - To knjižnico bomo uporabili za izračun direktne in inverzne kinematike robota.



Branje vrednosti z robotske roke

Robotska roka se v linuxu mapira na napravo (v mapi `/dev/`), z uporabo ustreznih pravil pa lahko posamezne serijske številke preslikamo na poljubno ime naprave. V našem primeru se robotske roke preslikajo na imena `/dev/arm_f0` do `/dev/arm_f8`. Zaporedna številka roke je označena na roki in ustreznem napajalniku.

Branje pozicij sklepov lahko izvedemo z naslednjo kodo:

```
from lerobot.robots.so101_follower import SO101Follower,
SO101FollowerConfig
from pathlib import Path

arm_name = 'arm_f1'
port = f'/dev/{arm_name}'
calibration_dir='calibrations'
robot_config = SO101FollowerConfig(port=port, id=arm_name,
calibration_dir=Path(calibration_dir))
robot = SO101Follower(robot_config)

robot.connect() # povezava na robota
robot.bus.disable_torque() # ugasnemo motorje

while True:
  current_obs = robot.get_observation()
  print(f'{current_obs=}')

robot.disconnect()
```

a) Povežite se na eno od rok in preverite, kakšen je razpon vrednosti za vsakega od sklepov. Ker smo motorje ugasnili, lahko roko premikate ročno in hkrati opazujete vrednosti sklepov v terminalu. V kakšni poziciji je roka, če so vsi sklepi na poziciji 0?

b) Robotske roke SO-101 lahko funkcionirajo tudi v načinu leader-follower. Priključite obe roki v napajanje in USB hub, ter uporabite naslednji ukaz za zagon nadzora:

```
lerobot-teleoperate --robot.type=so101_follower --robot.port=/dev/
arm_f1 --robot.id=arm_f1 --teleop.type=so101_leader --
teleop.port=/dev/arm_l1 --teleop.id=arm_l1
```

V ukazu morate imeni rok zamenjati z imeni konkretnih rok, ki ju uporabljate. Preko teleoperacije poskusite prijeti kak predmet z delovne površine. Koliko natančno lahko kontrolirate pozicijo roke? Razmislite, kakšne so omejitve sklepov robota, ki ga uporabljate. Lahko dosežete poljubno točko v prostoru? Lahko s konico robota narišete poljubno obliko, ali vas konstrukcija roke omejuje?

2. Denavit-Hartenberg parametri

V sklopu naslednjih nalog bomo v praksi spoznali nekaj osnovnih pristopov k upravljanju robotskega manipulatorja. Za upravljanje manipulatorja moramo prvo spoznati njegove lastnosti, ki jih na kompakten način opišemo z Denavit-Hartenberg parametri. Ti parametri zajamejo nujne parametre geometrijskega modela robotskega manipulatorja. Kako uporabljamo te parametre si bomo pogledali v prvi nalogi.

Geometrijski model manipulatorja lahko sicer v splošnem določimo kot verigo transformacij. Bolj natančno gre za transformacije, ki nas iz izhodiščnega prostora manipulatorja preslikajo v prostor posameznega sklepa (torej lahko neposredno določimo,

kako daleč je določena točka od prijemala, lahko pa določimo tudi njeno relativno lego). Na ta način lahko seveda določimo tudi položaj zadnjega sklepa (prijemala), če s končno transformacijo te verige preslikamo kar izhodiščno točko $(0, 0, 0)$.

Transformacijo lahko definiramo s pomočjo homogenih koordinat, oziroma matrike, ki opisuje novi prostor:

$$\boldsymbol{T}(q) = \begin{bmatrix} n(x) & s(x) & a(x) & p(x) \\ n(y) & s(y) & a(y) & p(y) \\ n(z) & s(z) & a(z) & p(z) \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

matriko pa lahko interpretiramo tudi kot štiri vektorje n, s, a, in p, ki definirajo preslikani prostor:

- p - translacija med prostoroma
- a - approach (približevanje objektu) (z os)
- n - normala (x os)
- s - slide (y os)

V zgoraj opisanem primeru je matrika izražena kot funkcija parametra q. Ta je vektor spremenljivk, ki definirajo stanje sklepov.

Problem določanja položaja robotskega manipulatorja lahko poenostavimo, saj gre pri vsem skupaj le za nizanje omejenega števila bazičnih operacij, ki so odvisne zgolj od tipa sklepov ter njihovih položajev. Denavit-Hartenberg parametri nam omogočajo prav to - poenostavljeno računanje transformacije.

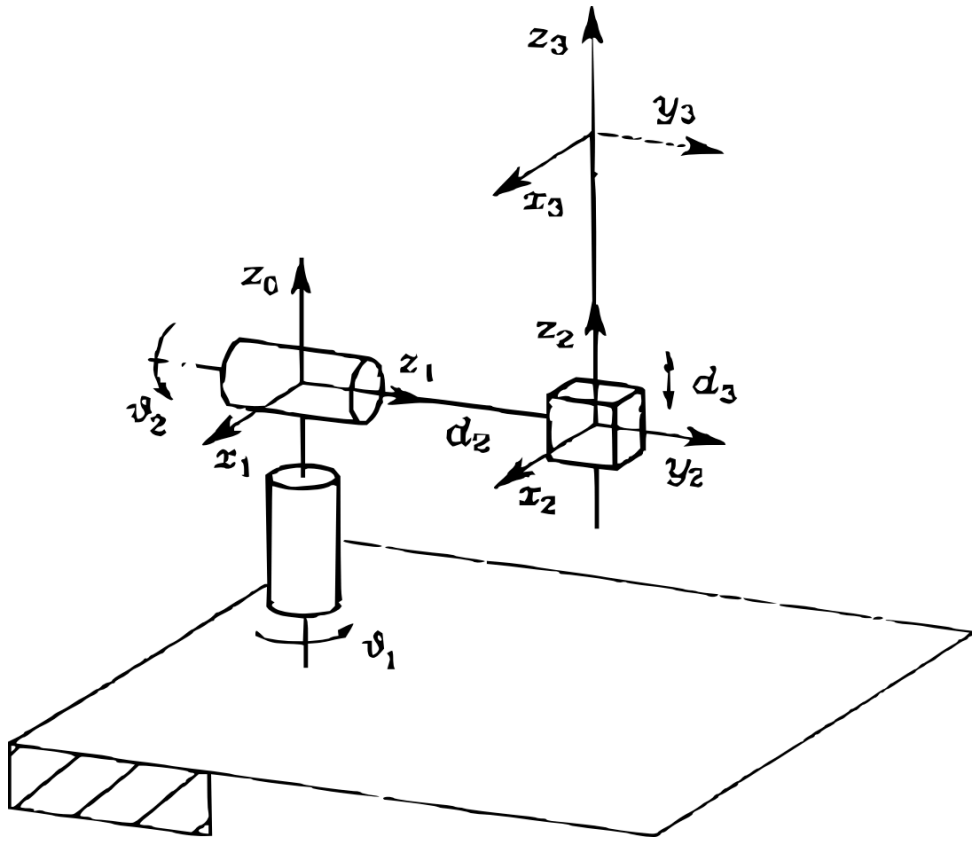
Vsak sklep opisujejo štirje parametri, trije statični ter eden, ki se spreminja (kateri je to, je odvisno od tipa sklepa):

- ϑ - rotacija okrog osi z (parameter v q, če je sklep rotacijski)
- d - premik v samem sklepu po z (parameter v q, če je sklep translacijski)
- a - razdalja med sklepoma po x osi
- α - rotacija okrog osi x osi

Izračun matrike za i-ti sklep izrazimo rekurzivno; z uporabo matrike za $(i-1)$-ti sklep računamo matriko za i-ti sklep.

$$\boldsymbol{T}_i = \boldsymbol{T}_{i-1} = \begin{bmatrix} \cos\vartheta_i & -\sin\vartheta_i & 0 & 0 \\ \sin\vartheta_i & \cos\vartheta_i & 0 & 0 \\ 0 & 0 & 1 & d_i \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & a_i \\ 0 & \cos\alpha_i & -\sin\alpha_i & 0 \\ 0 & \sin\alpha_i & \cos\alpha_i & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

a) Za Stanfordski model, predstavljen na spodnji sliki, določite tabelo Denavit-Hartenberg parametrov. Označite parametre, ki se lahko med delovanjem manipulatorja spreminjajo.



b) Napišite funkcijo `stanford_manipulator()`, ki za dani vektor parametrov vrne matrike transformacij iz izhodišča v vse tri sklepe. Dolžina prvih dveh sklepov naj bo 5 enot, zadnjega pa 2 enoti. Na podoben način določite tudi funkcijo `antropomorphic_manipulator()`, ki vrne isti rezultat za antropomorfni manipulator s predavanj. Dolžina vsakega od treh sklepov naj bo 3 enote. Pri reševanju naloge si pomagajte s spodaj priloženo funkcijo `dh_transform()` (nahaja se tudi v datoteki `utils.py`).

```
import numpy as np

def dh_transform(a, alpha, d, theta):
  ca, sa = np.cos(alpha), np.sin(alpha)
  ct, st = np.cos(theta), np.sin(theta)

  return np.array([
    [ct, -st * ca,  st * sa, a * ct],
    [st,  ct * ca, -ct * sa, a * st],
    [0.0,    sa,       ca,      d   ],
    [0.0,   0.0,      0.0,     1.0  ]
  ])
```

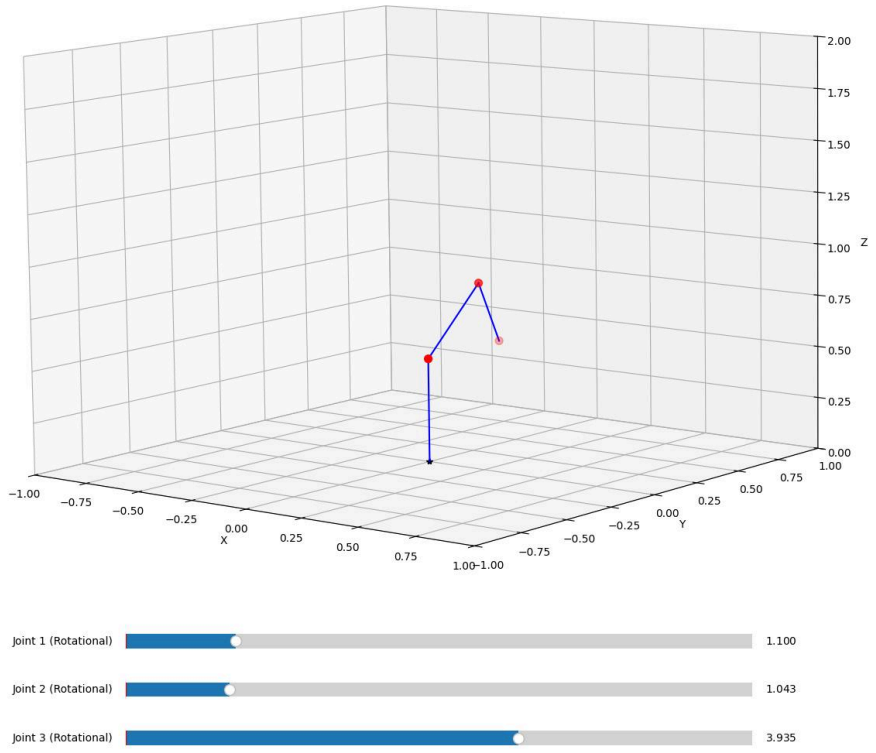
c) Vizualizirajte položaje posameznih sklepov robotskega manipulatorja za oba implementirana robota. Najprej izračunajte matrike za dano stanje manipulatorja, nato izhodiščno točko $(0, 0, 0)$ pomnožite z matriko vsakega sklepa posebej. Tako dobite položaje za posamezen sklep robota. Nato prikažite koordinatne sisteme za vse tri

sklepe s pomočjo priložene funkcije `show_coordinate_system()`. Za sam izris manipulatorja lahko uporabite matplotlib funkcije za risanje 3D grafov `plot()` ter `scatter()`. Izrisati morate torej pozicije sklepov in linije, ki sklepe povezujejo.

3. Direktna Kinematika

Direktna kinematika nam omogoča, da iz definicije robota in vrednosti parametrov posameznega sklepa določimo pozicijo posameznega sklepa v globalnem koordinatnem sistemu. To lahko izračunamo z zaporedno uporabo 4×4 transformacijskih matrik T_i, ki jih dobimo preko DH parametrov. Z direktno kinematiko lahko na primer analiziramo delovno območje robota in v simuliranem okolju določimo kritične pozicije robota (trk s podlago).

a) Izračunajte 3D pozicijo konice robotskih rok iz prejšnje naloge (Stanford in antropomorfna) za različne vrednosti sklepov. Priporočljivo je, da pozicije sklepov vežete na drsnike in pri premikih posodabljate izris.



Slika 3: Nastavljanje pozicije antropomorfnega manipulatorja z uporabo drsnikov.

4. Inverzna Kinematika

Za praktično uporabno vodenje robota pa potrebujemo še obraten proces, torej način, kako za želeno 3D točko P ugotovimo pozicije sklepov, ki postavijo konico robota v P. Formalno zapisano iščemo vektor parametrov $\boldsymbol{q}$, za katerega velja enačba $f(\boldsymbol{q}) = P$, kjer funkcija f

predstavlja direktno kinematiko našega robota. Procesu iskanja rešitev tega problema rečemo povratna oz. *inverzna* kinematika.

a) Zakaj je inverzna kinematika v splošnem težko rešljiv problem?

Namig: Koliko je lahko naborov parametrov, ki konico robotskega manipulatorja postavijo v določeno stanje?

b) V dodatnem materialu za nalogo lahko najdete implementacijo Newton-Raphson metode za inverzno kinematiko (funkcija `ik_newton_dh()` v datoteki `utils.py`). Več o njej si lahko preberete na povezavi. Metodo boste preizkusili tako, da boste poiskali rešitve za različne robote, ciljne točke in izhodiščne pozicije. Z uporabo funkcije `generate_figure_8()` lahko pridobite točke v obliki osmice, nato pa v zanki določite ustrezne parametre robota za vsako od njih. Preverite, kako se rešitve spremenijo, če za začetno pozicijo optimizacije vedno nastavite iste vrednosti, ali pa izhajate iz rešitve za prejšnjo točko.

c) Preizkusite skripto `SO-101_simulation.py`, ki naloži model roke SO-101 iz `.urdf` datoteke in pripravi drsnike, ki jih lahko uporabite za nastavitev pozicije. Definirajte točko $p = [0.1, -0.2, 0.05]$ in jo izrišite. Poleg tega izračunajte oddaljenost konice robota od ciljne točke. Izrišite tudi vektor, ki kaže od konice roke do ciljne točke. Nato z drsniki nastavite vrednosti sklepov robotske roke tako, da bo konica prijemala kar se da blizu točki p. Poskusite definirati algoritem, ki opisuje vaš pristop k optimizaciji pozicije robotske roke (v psevdokodi).

★ d) (15 točk) Implementirajte algoritem za inverzno kinematiko *cyclic coordinate descent* (CCD), ki je opisan na naslednji povezavi, opisan pa je tudi tukaj. V splošnem se algoritem v zanki sprehaja po zaporednih sklepih, in za vsakega določi vrednost parametra, ki minimizira razdaljo med tarčo in vrhom robota. Ker CCD ne obravnava medsebojnih odvisnosti sklepov, je lahko konvergenca za določene pozicije počasna, a za večino primerov bi moral algoritem delati dovolj dobro.

Preizkusite algoritem za različne robote, izhodiščne vrednosti in parametre. Poročajte število korakov, ki ga algoritem potrebuje za dosego dobre rešitve in končno napako v poziciji konice robota. Uporabite nabore točk, kot sta npr. `generate_square()` ali `generate_figure_8()` v datoteki `utils.py`. Seveda lahko nabore točk definirate tudi sami.

5. Nadzor robotskega manipulatorja

V zadnjem delu naloge boste z uporabo funkcij za inverzno kinematiko nadzorovali fizično robotsko roko.

a) V skripti `SO-101_IK.py` se nahaja prikaz uporabe knjižnice `ikpy` za izračun inverzne kinematike robotske roke SO-101. Knjižnica `ikpy` za sestavo modela robota uporabi datoteko `.urdf` (*Unified Robot Description Format*), ki je priložena. Inverzno kinematiko nato izračunamo s funkcijo `my_chain.inverse_kinematics()`, ki ji podamo ciljno točko. V dokumentaciji si poglejte še ostale možne argumente funkcije in jih preizkusite. Če želimo, lahko podamo tudi trenutno stanje robota, kar poenostavi

optimizacijo kinematike. Določimo lahko tudi končno orientacijo zadnjega sklepa, kar nam omogoča doseči bolj predvidljive pozicije prijemala. Premik fizičnega robota nato izvršimo s funkcijo `robot.send_action(action)`, kjer je `action` slovar, ki vsebuje imena sklepov in njihove ciljne pozicije. Orientacijo prijemala lahko v optimizacijo vključite z naslednjim parametrom:

```
target_orientation = np.eye(3)
target_orientation[2,2]=-1
ik = my_chain.inverse_kinematics(pt, target_orientation, 'all',
optimizer='scalar')
```

b) Napišite skripto, ki fizični robotski manipulator v zanki premika po vnaprej določenem zaporedju točk. Glede na doseg roke določite smiselno delovno območje robota, pri tem pa bodite pozorni, da so enote za ciljno točko izražene v metrih. Inverzno kinematiko lahko izračunate z ustreznimi funkcijami knjižnice `ikpy`. Preizkusite lahko enostavne like, kot sta kvadrat ali krog, ali pa kaj bolj kompleksnega. Orientacijo prijemala lahko fiksirate na konstantno pozicijo tako, da v izračun dodate parametra `target_orientation` in `orientation_mode="all"`. V tem primeru morate podati še 3×3 matriko, ki določa orientacijo prijemala, npr.

```
from ikpy.utils import geometry
target_orientation = geometry.rpy_matrix(0, np.deg2rad(180),0) #
prijemalo usmerjeno navzdol
```

Za izhodišče lahko uporabite kodo, ki se nahaja v skripti `SO-101_control.py`

Namig: Da se izognete potencialno nevarnim pozicijam robota, si točke in premike najprej izrišite v simuliranem okolju. Bodite pozorni predvsem na enote (koordinatni sistem robota je izražen v metrih) in na usmeritev koordinatnega sistema (os x je usmerjena naravnost naprej, os y kaže v levo, os z pa kaže navpično navzgor).

★ c) (10 točk) Z uporabo inverzne kinematike napišite program, ki bo na vnaprej določenem mestu na delovni površini pobral kocko ter jo prenesel na drugo vnaprej določeno mesto. Nato se bo vrnil po novo kocko in jo postavil na prvo kocko. Ta proces se naj ponavlja, vmes pa lahko čakate nekaj sekund za postavitev nove kocke ali pa na pritisk tipke. Za polno število točk morate eno na drugo zložiti vsaj tri kocke. Končni rezultat lahko posnamete in na zagovor prinesete posnetek in izvorno kodo.