# Instrukcja programowania robota humanoidalnego Unitree G1 EDU

dr inż. Mateusz Pomianek
Wersja: 1.8

Dokument dla zajęć laboratoryjnych i projektów studenckich. Obejmuje pracę z realnym robotem, konfigurację sieci i środowiska, pierwsze programy diagnostyczne, ROS 2, percepcję, planowanie, symulację w Isaac Lab oraz ścieżkę prowadzącą do bezpiecznego transferu sim-to-real.

**Skrót dla prowadzącego**

Jeżeli zespół pierwszy raz pracuje z realnym G1, zatrzymaj go na etapach 1-6. Do ROS2 i własnego sterowania przechodź dopiero wtedy, gdy zespół umie: ustawić sieć, sprawdzić stan pilota, odczytać stan robota, zalogować dane i bezpiecznie zakończyć sesję.

## Mapa dokumentu

| Sekcja | Po co istnieje |
|---|---|
| 1. Cel, zakres i sposób pracy | Ustala dla kogo jest instrukcja, czego nie robić na starcie i jak przechodzić między etapami. |
| 2. Platforma G1 EDU w pigułce | Zbiera najważniejsze parametry robota, sensory i ograniczenia mechaniczne. |
| 3. Bezpieczeństwo, stanowisko i bateria | Definiuje bezpieczne otoczenie, ładowanie, transport i wyłączanie. |
| 4. Pilot i pierwsze uruchomienie | Opisuje tryby pracy i scenariusz pierwszych testów bez własnego kodu. |
| 5. Architektura systemu i sieć | Porządkuje rolę komputerów, adresację IP i warstwy oprogramowania. |
| 6. Środowisko programistyczne | Podaje referencyjny zestaw narzędzi oraz zasadę zamrażania wersji. |
| 7–10. Pierwsze programy i ROS 2 | Buduje ścieżkę od odczytu stanu do pojedynczego sterowania stawem. |
| 11–14. Percepcja, logika i symulacja | Łączy sensory, FSM/MoveIt 2, symulację i trening RL/IL. |
| 15–16. Testy i ćwiczenia | Wprowadza standard odbioru, debugowania i dalszego rozwoju projektu. |
| Załączniki | Repozytorium, checklisty oraz lista materiałów źródłowych. |

## 1. Cel, zakres i sposób pracy

Ta instrukcja ma prowadzić zespół od pierwszego kontaktu z robotem do własnych eksperymentów w symulacji i na realnym sprzęcie. Nie jest to dokument dla sterowania wszystkimi 29 DOF naraz w pierwszym dniu pracy. Najpierw poznajesz platformę, potem komunikację, następnie diagnostykę, a dopiero później sterowanie, percepcję, planowanie i uczenie.

- Jeżeli pracujesz z realnym robotem po raz pierwszy, wykonuj wyłącznie etapy od bezpieczeństwa do odczytu stanu robota.

- Jeżeli masz już stabilne środowisko ROS 2 i wiesz, jak przejąć kontrolę pilotem, możesz przejść do warstwy węzłów, percepcji i logiki zadania.
- Uczenie ze wzmocnieniem lub imitacja mają sens dopiero wtedy, gdy zdefiniowano interfejs obserwacji, akcji i kryterium sukcesu zadania.

**Błąd początkujących**

Najdroższy błąd w pracy z humanoidem to zbyt szybkie przejście do sterowania niskiego poziomu. Bez stabilnej obserwacji stanu, procedury awaryjnej i ograniczenia zakresów ruchu test staje się niepowtarzalny i niebezpieczny.

## 2. Platforma G1 EDU w pigułce

Unitree G1 EDU to cywilna platforma humanoidalna wyposażona w 29 stopni swobody, kamerę głębi Intel RealSense D435i, lidar Livox MID360 i matrycę mikrofonową. Robot ma dwa komputery: niskopoziomowy kontroler lokomocji oraz komputer deweloperski, na którym uruchamiasz własne oprogramowanie.

| Element | Najważniejsze dane | Znaczenie dla projektu |
| --- | --- | --- |
| Kinematyka | 29 DOF; 6 stawów w każdej nodze, 3 w talii, 7 w każdym ramieniu; kolano do ok. 90 Nm. | Duża swoboda ruchu oznacza też większe ryzyko konfliktów między kontrolerami i błędów kinematyki. |
| Percepcja | RealSense D435i, Livox MID360, 4 mikrofony. | Możesz budować pipeline od obrazu RGB/głębi po chmurę punktów i lokalizację obiektów. |
| Napęd i równowaga | Niskopoziomowy kontroler lokomocji zarządza stabilnością i napędami. | Własny kod nie powinien obchodzić tej warstwy; komunikuj się przez oficjalne API i/lub ROS 2 bridge. |
| Moduły dłoni | Ręka protetyczna albo dłoń Dex3-1 / Inspire w zależności od wersji. | Zmiana końcówki wpływa na bezpieczeństwo ruchu i na planowanie trajektorii ramienia. |

**Ograniczenie mechaniczne ważne dla laboratorium**

Po zamontowaniu dłoni zręcznych Dex3-1 nie wykonuj ruchów typu przysiad i pozycja leżąca, jeżeli aktualna konfiguracja mechaniczna producenta ich nie dopuszcza. Przed testem sprawdź, czy przewody dłoni nie są dociśnięte, a trasa kabla omija radiatory, wlot wentylatora i elementy nośne.

## 3. Bezpieczeństwo, stanowisko i bateria

To jest sekcja obowiązkowa. Robot, pilot, bateria i stanowisko pracy muszą być przygotowane zanim uruchomisz własny kod.

### 3.1. Checklista stanowiska przed startem

| ☐ | Podłoga jest płaska, twarda i sucha, a wokół robota pozostawiono kilka metrów wolnej przestrzeni. |
| --- | --- |
| ☐ | Przewód Ethernet i zasilanie komputera laboratoryjnego są poprowadzone tak, aby nikt o nie nie zahaczył. |
| ☐ | Pilot jest naładowany, sparowany i leży w ręce operatora, a nie na biurku obok klawiatury. |
| ☐ | Jedna osoba odpowiada wyłącznie za bezpieczeństwo i pilot; druga obserwuje logi oraz uruchamia skrypty. |

| ☐ | Zespół uzgodnił procedurę awaryjną: kto przejmuje pilota, kto odłącza zasilanie, kto dokumentuje problem. |
|---|---|

### 3.2. Bateria: zasady, których nie wolno pomijać

- Przed każdym użyciem upewnij się, że bateria ma wystarczający poziom energii; przy poziomie poniżej 10% zakończ pracę tak szybko, jak to bezpieczne i doładuj lub wymień pakiet.
- Nie podłączaj i nie odłączaj baterii, gdy jej zasilanie jest włączone.
- Nie ładuj baterii natychmiast po intensywnej pracy robota. Pozwól jej ostygnąć do temperatury pokojowej.
- Ładuj tylko oficjalną ładowarką producenta, w dobrze wentylowanym miejscu, z dala od materiałów palnych.
- Do ładowania używaj otoczenia zgodnego z instrukcją producenta; unikaj skrajnych temperatur i nie zostawiaj procesu bez nadzoru.
- Nie zanurzaj baterii w cieczy, nie przebijaj jej, nie zgniataj i nie używaj pakietu z uszkodzoną obudową.
- Na dłuższe przechowywanie utrzymuj poziom energii około 70%; jeżeli pakiet spadnie poniżej 30%, doładuj go przed odłożeniem.
- Do transportu długodystansowego rozładuj pakiet do około 65% i nie przewoź go razem z metalowymi przedmiotami.

| Parametr | Wartość orientacyjna |
|---|---|
| Napięcie nominalne baterii | DC 46.8 V |
| Pojemność | 9000 mAh, 421.2 Wh |
| Ładowarka | 54.6 V, 5.5 A, 300 W |
| Typowy czas ładowania | około 1.5 h |
| Przechowywanie | najlepiej w suchym miejscu, ok. 22–28°C |

### 3.3. Checklista zamknięcia sesji

| ☐ | Zatrzymaj robot w stabilnej pozycji, przywróć kontrolę pilotem i potwierdź brak aktywnego ruchu. |
|---|---|
| ☐ | Zapisz logi: komendy, odczyty stawów, błędy aplikacji, parametry eksperymentu oraz seed. |
| ☐ | Wyłącz warstwę użytkownika, zakończ sesje ROS 2 / DDS i dopiero potem odłącz komponenty dodatkowe. |
| ☐ | Jeżeli kończysz dzień pracy, wyjmij baterię, obejrzyj złącza i odłóż pakiet do bezpiecznego miejsca przechowywania. |
| ☐ | Jeżeli test dotyczył dłoni dexterous, obejrzyj trasę przewodów i sprawdź, czy nie ma śladów dociśnięcia kabla. |

## 4. Pilot i pierwsze uruchomienie

Pierwsze testy robota wykonuj wyłącznie pilotem. Celem nie jest jeszcze sterowanie własnym kodem, ale zrozumienie postawy, reakcji na drążki oraz trybów awaryjnych.

| Tryb / funkcja | Kombinacja | Do czego używać |
|---|---|---|
| Debug mode | L2 + R2 | Przejście do trybu przydatnego przy programowaniu przez SDK i podczas testów integracyjnych. |

| | | |
|---|---|---|
| Zero torque | L2 + Y | Silniki nie stawiają aktywnego oporu; używaj tylko świadomie i w stabilnych warunkach. |
| Damping mode | L2 + B | Silniki są wygaszone, ale zachowują tłumienie; najlepszy tryb do pierwszego „czucia” robota. |
| Ready / posture switch | L2 + góra lub zgodnie z aktualnym profilem pilota | Pozycja przygotowawcza przed przejściem do ruchu lub innych postaw. |
| Squat / posture action | L2 + A / inne kombinacje zależne od mapy pilota | Testuj tylko wtedy, gdy konfiguracja dłoni i otoczenie są zgodne z instrukcją producenta. |
| Interakcje / demonstracje | SELECT + przyciski funkcyjne | Tryby pokazowe; nie używaj ich podczas pierwszej diagnostyki lub w ciasnym otoczeniu. |

- Włączanie pilota: krótkie naciśnięcie i następnie dłuższe przytrzymanie przycisku zasilania.
- Lewy drążek odpowiada za ruch w przód / tył, a prawy za obrót w lewo / prawo.
- Poziom naładowania pilota jest pokazywany przez diody LED; jeżeli zespół testuje kod dłużej niż krótki eksperyment, doładuj pilot przed zajęciami.

**Scenariusz pierwszych 5 minut**

1) przygotuj stanowisko; 2) uruchom pilota; 3) przejdź do damping; 4) ustaw robota do pozycji gotowości; 5) wykonaj tylko niewielkie wychylenia drążka w przód i tył; 6) sprawdź, czy cały zespół rozumie, kto w każdej chwili może zatrzymać test.

## 5. Architektura systemu i sieć

W projekcie przyjmij prosty model: percepcja → decyzja → sterowanie. Nie mieszaj tych warstw w jednym skrypcie, bo wtedy trudno odtworzyć stan eksperymentu i znaleźć źródło błędu.

| Warstwa | Rola | Przykłady |
|---|---|---|
| Pilot / operator | Bezpieczeństwo i ręczne przełączanie trybów. | Pilot G1 R3, procedura awaryjna. |
| SDK / DDS | Oficjalna komunikacja z robotem. | unitree_sdk2, unitree_sdk2_python. |
| ROS 2 | Łączenie modułów aplikacji. | g1_driver, joint_state_bridge, task_manager. |
| Percepcja | Obraz, głębia, lidar, ekstrakcja cech. | OpenCV, RealSense, Livox, cv_bridge. |
| Planowanie i logika | FSM, trajektorie, wybór akcji. | MoveIt 2, task manager, policy PPO/BC. |

| Urządzenie | Adres IP / uwaga |
|---|---|
| Komputer lokomocji | 192.168.123.161 |
| Komputer deweloperski | 192.168.123.164; domyślny login unitree / hasło 123 (zmień po pierwszym logowaniu) |
| Lidar Livox | 192.168.123.20 |
| Komputer laboratoryjny | Ustaw statyczny adres w podsieci 192.168.123.0/24, np. 192.168.123.199. |

Porty RJ45 oznaczone przez producenta jako porty zewnętrzne służą do podłączenia komputera laboratoryjnego. Robot nie udostępnia DHCP, więc konfiguracja statyczna jest obowiązkowa.

```
# przykład konfiguracji statycznego IP na komputerze laboratoryjnym
sudo ip addr flush dev eth0
sudo ip addr add 192.168.123.199/24 dev eth0
sudo ip link set eth0 up
```

```
ping 192.168.123.161
ping 192.168.123.164
```

**Ważna zasada**

Jeżeli robot jest w ruchu, nie eksperymentuj równocześnie z wieloma ścieżkami sterowania. Sterowanie lokomocji i własny regulator pozycji w tej samej chwili mogą doprowadzić do zachowania nieprzewidywalnego.

## 6. Środowisko programistyczne i zamrożenie wersji

Dla zajęć laboratoryjnych ważniejsza od „najnowszej możliwej” wersji jest wersja powtarzalna. Ustal jedną referencyjną konfigurację i zapisuj ją w repozytorium razem z nazwą sterownika, Pythona, ROS 2 i commitami.

| Składnik | Wersja referencyjna / uwaga |
|---|---|
| System operacyjny | Ubuntu 22.04 LTS |
| ROS 2 | Humble |
| Python dla symulacji | Conda, osobne środowisko np. env_isaaclab |
| SDK Unitree | unitree_sdk2 lub unitree_sdk2_python — wersję zapisz w README / lockfile |
| Isaac Sim / Isaac Lab | Stosuj jeden uzgodniony zestaw wersji dla całego kursu; w tej instrukcji zachowujemy tor zgodny z materiałem wyjściowym projektu. |
| RL / IL | rsl-rl-lib, skrl, robomimic / BC tylko po ustaleniu kompatybilności z wybraną wersją Isaac Lab |

```
sudo apt update && sudo apt upgrade -y
sudo apt install -y build-essential git curl wget unzip python3-pip \
  software-properties-common net-tools openssh-client

# ROS 2 Humble (schemat)
sudo apt install -y ros-humble-desktop python3-colcon-common-extensions python3-rosdep
sudo rosdep init
rosdep update
```

```
# oddzielne środowisko dla symulacji
wget https://repo.anaconda.com/miniconda/Miniconda3-latest-Linux-x86 64.sh -O
~/miniconda.sh
bash ~/miniconda.sh -b -p ~/miniconda3
~/miniconda3/bin/conda init bash
conda create -n env isaaclab python=3.11 -c conda-forge -y
conda activate env_isaaclab
```

**Co koniecznie zapisać w repozytorium**

Nazwę GPU, wersję sterownika, wynik nvidia-smi, wersje SDK i bibliotek, commit repozytorium symulacji, seed eksperymentu oraz polecenie uruchomienia. Bez tego wynik treningu nie jest odtwarzalny.

## 7. Kamienie milowe projektu

| Etap | Cel | Kryterium wyjścia |
|---|---|---|
| M0: bezpieczeństwo | Zespół umie przejąć kontrolę pilotem i zamknąć sesję. | Każda osoba zna procedurę awaryjną i check-listę start/stop. |

| M1: sieć i komunikacja | Robot odpowiada na ping, a komputer deweloperski jest osiągalny. | Stabilne połączenie z prawidłową adresacją. |
|---|---|---|
| M2: diagnostyka | Działa program odczytujący stan stawów i IMU. | Odczyty są stabilne, bez NaN, z przewidywalną częstotliwością. |
| M3: pojedyncze sterowanie | Jednym stawem można bezpiecznie wykonać minimalny ruch. | Komenda wraca w sprzężeniu zwrotnym, bez drgań i konfliktów. |
| M4: ROS 2 i percepcja | Pakiety są rozdzielone, a obraz z kamery trafia do węzła. | Workspace buduje się powtarzalnie, a logi pokazują poprawny przepływ danych. |
| M5: zadanie użytkowe | Działa FSM / planowanie dla prostego zadania pick-and-place. | Zespół potrafi wskazać stan, przejścia i warunki błędu. |
| M6: symulacja i uczenie | Zdefiniowano obserwacje, akcje, nagrodę i warunki końca epizodu. | W treningu widać postęp; model przechodzi walidację w symulacji zanim dotknie robota. |

## 8. Program 1: odczyt stanu robota

Pierwszy własny program nie powinien sterować mechaniką. Ma tylko pokazać, że potrafisz nawiązać połączenie, odczytać stan i zapisać go do logu. To daje największy zysk diagnostyczny przy minimalnym ryzyku.

- Odczytuj tylko to, co potrafisz później zweryfikować: nazwy stawów, pozycje, prędkości, momenty, orientację IMU, stan pilota.
- Na tym etapie zapisuj wszystko do CSV albo do uporządkowanego logu z timestampem.
- Nie ufaj samemu printowi w terminalu; sprawdź częstotliwość i kompletność danych.

```
import csv
import time
from dataclasses import dataclass

@dataclass
class JointStateSnapshot:
    name: str
    position: float
    velocity: float
    torque: float


def read_robot_state() -> list[JointStateSnapshot]:
    # tutaj podstaw właściwy odczyt z unitree_sdk2_python / ROS 2 bridge
    return [
        JointStateSnapshot("left_shoulder_pitch", 0.10, 0.00, 0.02),
        JointStateSnapshot("left_elbow", 0.42, 0.00, 0.01),
        JointStateSnapshot("waist_yaw", 0.00, 0.00, 0.00),
    ]


def main() -> None:
    with open("joint_debug.csv", "w", newline="") as f:
        writer = csv.writer(f)
        writer.writerow(["t", "joint", "position", "velocity", "torque"])
        for step in range(50):
            snapshots = read_robot_state()
            now = time.time()
            for s in snapshots:
                writer.writerow([now, s.name, s.position, s.velocity, s.torque])
            time.sleep(0.1)

if __name__ == "__main__":
    main()
```

### 8.1. Co sprawdzić po uruchomieniu

| ☐ | Czy wszystkie oczekiwane stawy pojawiają się w odczycie i czy nazwy są spójne z modelem robota. |
|---|---|
| ☐ | Czy częstotliwość odczytu jest zbliżona do założonej i nie „pływa” losowo. |
| ☐ | Czy w danych nie ma NaN, zer wygenerowanych przez błąd połączenia ani dziwnych skoków pozycjonowania. |
| ☐ | Czy log da się później złączyć z komendami sterującymi po czasie. |

## 9. Program nr 2: bezpieczne sterowanie jednym stawem

Dopiero po M2 przejdź do minimalnego sterowania. Najlepiej użyć pojedynczego stawu w ramieniu, z małą amplitudą, niską prędkością i krótkim oknem czasu.

> **Zasada**
>
> Nie wysyłaj trajektorii do wielu stawów naraz. W pierwszych testach mierz opóźnienie, zgodność kierunku ruchu i odpowiedź zwrotną tylko dla jednego stawu.

```
import rclpy
from rclpy.node import Node
from sensor_msgs.msg import JointState

class SingleJointCommander(Node):
    def __init__(self) -> None:
        super().__init__("single_joint_commander")
        self.pub = self.create_publisher(JointState, "/unitree/joint_command", 10)
        self.timer = self.create_timer(0.05, self.publish_command)
        self.phase = 0

    def publish_command(self) -> None:
        msg = JointState()
        msg.name = ["left_shoulder_pitch"]
        target = 0.20 if self.phase < 20 else 0.05
        msg.position = [target]
        msg.velocity = [0.0]
        msg.effort = [0.0]
        self.pub.publish(msg)
        self.phase = (self.phase + 1) % 40


def main() -> None:
    rclpy.init()
    node = SingleJointCommander()
    rclpy.spin(node)
    node.destroy_node()
    rclpy.shutdown()

if __name__ == "__main__":
    main()
```

- Najpierw potwierdź, że robot jest w poprawnym trybie pracy i że temat komend jest rzeczywiście odbierany przez właściwy moduł.
- Ogranicz zakres pozycji i czas trwania testu. Jeżeli pojawiają się drgania, zmniejsz amplitudę i częstotliwość.
- Po każdym teście porównaj komendę z raportowanym stanem stawu.

## 10. ROS 2: struktura workspace i podział odpowiedzialności

ROS 2 porządkuje projekt. Zamiast jednej dużej aplikacji budujesz kilka współpracujących węzłów o jasnych rolach. To krytyczne, jeżeli projekt ma przeżyć dłużej niż jedno laboratorium.

| Pakiet / węzeł | Rola |
|---|---|
| g1_driver_node | Komunikacja z robotem przez SDK / DDS. |
| joint_state_bridge_node | Publikacja sensor_msgs/JointState oraz danych IMU. |
| arm_controller_node | Sterowanie ramionami i dłonią. |
| locomotion_command_node | Komendy wysokiego poziomu dla ruchu bazy i kroków. |
| perception_node | Obsługa obrazu, głębi, lidaru, detekcji i filtrów. |
| task_manager_node | Logika zadania: szukaj, podejdź, złap, przenieś, odłóż. |

```
g1_student_ws/
└── src/
    ├── g1_driver/
    ├── g1_control/
```

```
    ├── g1 perception/
    └── g1 bringup/
```

```
from setuptools import setup
package name = "g1 control"
setup(
    name=package name,
    version="0.1.0",
    packages=[package name],
    install requires=["setuptools"],
    zip safe=True,
    maintainer="lab",
    description="Student control examples for Unitree G1",
    entry points={
        "console scripts": [
            "single joint commander = g1 control.single joint commander:main",
        ],
    },
)
```

> **Reguła projektowa**
>
> Jeden pakiet = jedna odpowiedzialność. Nie mieszaj kodu percepcji, sterowania i treningu RL w jednym katalogu. Jeżeli coś wymaga innego cyklu rozwoju lub innych zależności, zrób z tego osobny moduł.

## 11. Percepcja: kamera, lidar i walidacja danych wejściowych

Robot musi widzieć otoczenie zanim zacznie działać inteligentnie. Potok percepcji warto dzielić na trzy poziomy: detekcję, ekstrakcję cech i decyzję.

1. **Detekcja** — znajdź obiekt, człowieka albo obszar zainteresowania na obrazie lub w chmurze punktów.
2. **Ekstrakcja cech** — oblicz położenie 2D/3D, orientację, rozmiar i deskryptory potrzebne dalszej logice.
3. **Klasyfikacja lub wybór chwytu** — zadecyduj, co robić dalej: podejść, dopasować trajektorię, uruchomić chwyt.

```
import cv2

cap = cv2.VideoCapture(0)
if not cap.isOpened():
    raise RuntimeError("Camera not available")

ok, frame = cap.read()
if not ok:
    raise RuntimeError("Cannot read frame")

cv2.imwrite("frame debug.png", frame)
print("Saved frame debug.png")
cap.release()
```

```
import rclpy
from rclpy.node import Node
from sensor msgs.msg import Image
from cv bridge import CvBridge

class ImageDebugNode(Node):
    def   init  (self) -> None:
        super().  init  ("image debug node")
        self.bridge = CvBridge()
        self.subscription = self.create subscription(
            Image, "/camera/color/image raw", self.callback, 10
        )

    def callback(self, msg: Image) -> None:
        frame = self.bridge.imgmsg to cv2(msg, desired encoding="bgr8")
        self.get logger().info(f"Frame: {frame.shape}")
```

### 11.1. Minimalna walidacja sensoryki

| ☐ | Czy obraz nie jest odwrócony, prześwietlony lub losowo pusty. |
|---|---|

| ☐ | Czy timestamp wiadomości jest stabilny i pozwala później synchronizować obraz ze stanem stawów. |
|---|---|
| ☐ | Czy lidar publikuje chmurę punktów w oczekiwanej ramce odniesienia. |
| ☐ | Czy detektor działa na surowej klatce debugowej zanim zostanie wpięty do całego pipeline'u. |

## 12. Planowanie ruchu i logika aplikacji

Kiedy robot umie już obserwować stan oraz wykonywać krótkie komendy, kolejnym krokiem jest logika zadaniowa. Na zajęciach najlepiej zacząć od maszyny stanów, bo wprost pokazuje, gdzie kończy się percepcja, a zaczyna decyzja.

```
from enum import Enum, auto

class TaskState(Enum):
    SEARCH = auto()
    APPROACH = auto()
    ALIGN = auto()
    GRASP = auto()
    LIFT = auto()
    PLACE = auto()

current_state = TaskState.SEARCH
```

- FSM jest dobrą metodą dydaktyczną: możesz prześledzić przejścia, warunki błędu i zasady bezpieczeństwa bez wprowadzania modelu uczącego.
- MoveIt 2 wprowadź dopiero wtedy, gdy potrzebujesz planowania trajektorii ramienia z kolizjami, limitami stawów i ograniczeniami kinematycznymi.
- Logika zadaniowa nie powinna być zaszyta w węźle percepcji; przechowuj stan zadania i decyzje w osobnym module.

## 13. Symulacja, RL i opcjonalna ścieżka imitacyjna

Uczenie ma sens dopiero wtedy, gdy potrafisz nazwać zadanie, obserwacje, akcje i warunek sukcesu. Dla kursu najrozsądniejszym przykładem jest chwytanie i przenoszenie obiektu, a nie „uczenie wszystkiego naraz".

1. Skonfiguruj GPU, sterownik i osobne środowisko Conda.
2. Uruchom prosty świat w symulacji i potwierdź, że model robota ładuje się bez błędów.
3. Zdefiniuj obserwacje, akcje, nagrodę i warunki końca epizodu.
4. Zacznij od małej liczby środowisk równoległych i krótkiego treningu.
5. Zweryfikuj rollout oraz skale sygnałów zanim zaczniesz skalować eksperyment.

```
# szkic funkcji nagrody

def compute_reward(distance_to_object: float,
                   grasp_success: bool,
                   object_height: float,
                   action_penalty: float) -> float:
    reward = 0.0
    reward += 1.5 * max(0.0, 1.0 - distance_to_object)
    reward += 5.0 if grasp_success else 0.0
    reward += 3.0 * max(0.0, object_height)
    reward -= 0.01 * action_penalty
    return reward
```

- Nagroda powinna wspierać postęp etapami; sama nagroda końcowa za pełny chwyt zwykle jest za rzadka.
- Zanim model dotknie realnego robota, musi przejść walidację w symulacji na wielu seedach i przy lekkiej losowości parametrów.
- Wszystkie konfiguracje eksperymentów zapisuj w repozytorium, nie tylko w historii terminala.

### 13.1. Ścieżka opcjonalna: uczenie z demonstracji i teleoperacja

Jeżeli zespół ma do dyspozycji teleoperację, można zbudować tor data collection → HDF5 → BC/robomimic. Dobra demonstracja jest gładka, powtarzalna i ma czytelnie rozdzielone podzadania. Jakość demonstracji oraz anotacji silnie wpływa na wyniki treningu.

- Dane demonstracyjne można nagrywać z użyciem klawiatury, SpaceMouse lub hand-trackingu, zależnie od środowiska.
- Zanim wygenerujesz duży zbiór, obejrzyj mały dataset testowy i sprawdź, czy etykiety podzadań są sensowne.
- Przy ocenie policy nie polegaj tylko na ostatnim checkpointcie; testuj kilka epok, bo model pośredni często jest lepszy od finalnego.

**Sim-to-real**

Policy przenoszone z symulacji na robota powinny mieć dodatkowe bezpieczniki: ograniczenia amplitudy akcji, filtrację komend, jawne limity czasu i check-listę przejęcia kontroli pilotem. Najpierw uruchamiaj je w małych oknach testowych i przy minimalnym zakresie ruchu.

# 14. Logowanie, testy i debugowanie

W robotyce większość czasu zabiera integracja. Dobry eksperyment to taki, który można odtworzyć i zrozumieć po tygodniu, a nie tylko „uruchomić teraz”.

| Objaw | Najbardziej prawdopodobna przyczyna | Co sprawdzić najpierw |
|---|---|---|
| Brak połączenia z robotem | Błędna adresacja IP lub routing. | ping, ip addr, kabel, właściwa podsieć 192.168.123.x. |
| Program działa, robot się nie rusza | Zły tryb pracy albo konflikt kontrolerów. | Czy robot jest w trybie debug / gotowości, czy temat komend jest poprawny. |
| Drgania stawu | Za duża amplituda, zbyt wysoka częstotliwość lub konflikt komend. | Zmniejsz zakres ruchu i testuj jeden staw. |
| CUDA unavailable | Sterownik lub PyTorch niezgodny z GPU. | nvidia-smi, wersja CUDA, wheel PyTorch / Isaac. |
| Model RL nie uczy się | Źle zdefiniowana nagroda lub obserwacje. | Zwizualizuj rollout, sprawdź skale sygnałów i rozkład nagrody. |

## 14.1. Minimalny standard logowania

- timestamp, seed, commit repozytorium, nazwa środowiska, wersje bibliotek, identyfikator GPU, temat komend i temat odczytu stanu;
- log CSV lub ROS bag z komendami oraz stanem stawów dla każdego testu na realnym robocie;
- dla treningu: konfiguracja reward, num_envs, liczba iteracji, nazwa checkpointu i metryki walidacyjne.

## 14.2. Lista przed testem na realnym robocie

| | |
|---|---|
| ☐ | Czy działa pilot i czy procedura awaryjna została powtórzona ustnie przez zespół? |
| ☐ | Czy ten sam algorytm przeszedł przynajmniej podstawowy test w symulacji? |
| ☐ | Czy zakres ruchu jest ograniczony do minimum potrzebnego w teście? |
| ☐ | Czy logujesz komendy, stan stawów i błędy aplikacji? |
| ☐ | Czy jedna osoba obserwuje tylko bezpieczeństwo, a nie terminal? |

## 15. Ćwiczenia laboratoryjne i ścieżka rozwoju

1. Napisz program odczytujący i logujący stan trzech wybranych stawów oraz IMU.
2. Uruchom bezpieczny ruch jednego stawu i zmierz opóźnienie odpowiedzi między komendą, a stanem zwrotnym.
3. Zbuduj węzeł ROS 2, który odbiera obraz i zapisuje klatkę debugową z kamery.
4. Zaimplementuj prostą maszynę stanów do zadania „szukaj i podejdź”.
5. W symulacji zdefiniuj funkcję nagrody dla chwytania obiektu i porównaj dwa warianty shaping reward.
6. Porównaj rozwiązanie regułowe z polityką PPO albo BC dla tego samego zadania i opisz kompromisy.

Dalszy rozwój projektu może obejmować integrację MoveIt 2, sterowanie dłonią Inspire lub Dex3-1, zastosowanie modeli percepcyjnych z PyTorch, teleoperację do zbierania demonstracji oraz transfer sim-to-real z jawnym protokołem bezpieczeństwa.

## Załącznik A. Minimalna struktura repozytorium projektu

```
g1 project/
├── README.md
├── docs/
├── ros2 ws/
│   └── src/
│       ├── g1 driver/
│       ├── g1 control/
│       ├── g1 perception/
│       └── g1 bringup/
├── sim/
│   └── IsaacLab/
├── experiments/
│   ├── logs/
│   └── checkpoints/
└── scripts/
    ├── bootstrap.sh
    ├── run sim.sh
    └── run robot.sh
```

## Załącznik B. Dobre praktyki kodowania i eksperymentów

- Kod sterowania i konfiguracje zapisuj w repozytorium, nie tylko w terminalu.
- Każdy eksperyment oznaczaj nazwą, seedem i wersją środowiska.
- Węzły ROS 2 uruchamiaj przez launch files, a nie ręcznie z wielu terminali bez dokumentacji.
- Każdą zmianę na realnym robocie poprzedzaj testem jednostkowym, testem w symulacji albo testem HIL/SIL, jeśli zespół ma taki pipeline.
- Nazwy tematów, klas i parametrów trzymaj po angielsku i konsekwentnie zapisuj w README.

# Ćwiczenie 2: Percepcja, planowanie i komunikacja ROS 2

**Uwaga**

**Skrót dla prowadzącego:** Przed dopuszczeniem zespołu do tej instrukcji upewnij się, że każdy uczestnik: (1) zna procedurę awaryjną i obsługę pilota, (2) samodzielnie odczytał stan stawów i IMU, (3) przeprowadził bezpieczny ruch jednego stawu i zalogował odpowiedź sprzężenia zwrotnego.

## Mapa dokumentu

| Sekcja | Po co istnieje |
|---|---|
| 1. Cel i zakres ćwiczenia | Określa, co student ma osiągnąć i jakich błędów unikać. |
| 2. Przypomnienie: sieć i środowisko | Szybka weryfikacja, czy infrastruktura z Ćwiczenia 1 działa poprawnie. |
| 3. ROS 2 — workspace i węzły | Porządkuje strukturę pakietów i podział odpowiedzialności. |
| 4. Percepcja: kamera i lidar | Buduje potok od obrazu do ekstrakcji cech. |
| 5. Fuzja sensoryczna i transformacje TF2 | Łączy dane z różnych sensorów we wspólnej ramce. |
| 6. Planowanie ruchu i maszyna stanów | Definiuje logikę zadania i bezpieczne przejścia. |
| 7. MoveIt 2 i trajektorie ramienia | Wprowadza planowanie kinematyczne z limitami stawów. |
| 8. Logowanie, testy i debugowanie | Standard odbioru i lista diagnostyczna. |
| 9. Ćwiczenia i ścieżka rozwoju | Konkretne zadania i kierunki dalszego projektu. |
| Załączniki | Checklisty, struktura repozytorium, materiały dodatkowe. |

## 1. Cel i zakres ćwiczenia

Ćwiczenie 2 zakłada, że masz sprawne środowisko ROS 2, wiesz jak bezpiecznie uruchomić robota i potrafisz odczytać stan stawów. Celem tego etapu jest zbudowanie **warstwy percepcji i logiki zadaniowej** — tak, aby robot nie tylko wysyłał komendy, ale reagował na środowisko.

**Czego NIE robić na starcie tego ćwiczenia:**

- Nie przechodź od razu do MoveIt 2 bez sprawdzonego pipeline percepcyjnego.
- Nie uruchamiaj wszystkich węzłów równocześnie bez pliku launch.
- Nie testuj na realnym robocie algorytmu, który nie przeszedł testu w symulacji lub przynajmniej z danymi z pliku bag.

### Kamienie milowe ćwiczenia 2

| Etap | Kryterium wyjścia |
|---|---|

| | |
|---|---|
| M2.1: Workspace ROS 2 | Pakiety budują się bezawaryjnie; launch file uruchamia wszystkie węzły. |
| M2.2: Obraz z kamery w węźle | Klatkę debugową można zapisać z /camera/color/image_raw. |
| M2.3: Chmura punktów lidaru | Lidar publikuje PointCloud2 w spójnej ramce; dane są wizualizowane w RViz. |
| M2.4: Detekcja i lokalizacja obiektu | Węzeł percepcji zwraca PoseStamped celu z wiarygodnym timestampem. |
| M2.5: FSM i podejście | Robot podchodzi do obiektu przez maszynę stanów; każde przejście jest logowane. |
| M2.6: Trajektoria ramienia | MoveIt 2 planuje i wykonuje trajektorię jednego ramienia bez kolizji. |

## 2. Weryfikacja infrastruktury z Ćwiczenia 1

Zanim zaczniesz nowe węzły, potwierdź, że fundamenty są sprawne. Ta sekcja zajmuje maksymalnie 10 minut — jeżeli trwa dłużej, wróć do checklisty sieciowej z Ćwiczenia 1.

### 2.1. Checklista szybkiej weryfikacji

- Robot odpowiada na ping 192.168.123.161 i 192.168.123.164.
- Komputer deweloperski jest osiągalny przez SSH (login zmieniony po Ćwiczeniu 1).
- Zmienne ROS_DOMAIN_ID i RMW_IMPLEMENTATION są ustawione tak samo na wszystkich maszynach.
- ros2 topic list pokazuje tematy g1_driver po uruchomieniu bringup.
- Program diagnostyczny z Ćwiczenia 1 zapisuje dane do CSV bez błędów.

```
# Weryfikacja sieci
ping 192.168.123.161 -c 4
ping 192.168.123.164 -c 4

# Weryfikacja ROS 2
source /opt/ros/humble/setup.bash
source ~/g1 student ws/install/setup.bash
ros2 topic list
ros2 topic hz /unitree/joint states   # oczekiwane: ~50 Hz
```

**Ostrzeżenie**

Jeżeli ros2 topic hz zgłasza NaN lub nieregularną częstotliwość, najpierw napraw źródło danych. Dalsze węzły będą debugować się samo: postaw na stabilność podstawy.

## 3. ROS 2: workspace i podział odpowiedzialności

ROS 2 wymusza modularność: jeden pakiet, jedna odpowiedzialność. Właściwa struktura oszczędza czas przy każdej kolejnej iteracji.

## 3.1. Referencyjny układ workspace

```
g1 student ws/
└── src/
    ├── g1 bringup/        # launch files łączące wszystkie węzły
    ├── g1 driver/         # komunikacja SDK / DDS → ROS 2
    ├── g1 control/        # komendy stawów, FSM, logika sterowania
    ├── g1 perception/     # obraz, głębia, lidar, detekcja
    └── g1 interfaces/     # własne msg i srv, jeżeli potrzebne
```

## 3.2. Tabela węzłów i tematów

| Węzeł | Tematy / odpowiedzialność |
| --- | --- |
| g1_driver_node | Subskrybuje /unitree/joint_command; publikuje /unitree/joint_states, /imu |
| perception_node | Subskrybuje /camera/color/image_raw, /lidar/points; publikuje /detected_object/pose |
| arm_controller_node | Subskrybuje /detected_object/pose; wysyła komendy przez MoveIt 2 action |
| locomotion_command_node | Publikuje komendy ruchu bazy do g1_driver_node |
| task_manager_node | Zarządza FSM; subskrybuje stan percepcji i stawów; inicjuje akcje |

## 3.3. Minimalny plik launch

```
# g1 bringup/launch/full system.launch.py
from launch import LaunchDescription
from launch ros.actions import Node

def generate launch description():
    return LaunchDescription([
        Node(package='g1 driver',     executable='g1 driver node',
name='driver'),
        Node(package='g1 perception', executable='perception node',
name='perception'),
        Node(package='g1 control',    executable='task manager node',
name='task manager'),
    ])
```

**Wskazówka**

Uruchamiaj cały system przez launch file, a nie ręcznie z wielu terminali. Zapewnia to powtarzalność, czytelne logi i łatwe zamknięcie wszystkiego jednym Ctrl+C.

# 4. Percepcja: kamera i lidar

Warstwa percepcji zamienia surowe dane sensoryczne na informacje użyteczne dla logiki zadaniowej. Buduj ją poziomami: najpierw odbierz dane, potem waliduj, dopiero wtedy wyciągaj cechy.

## 4.1. Kamera Intel RealSense D435i

RealSense D435i dostarcza obraz RGB (topic `/camera/color/image_raw`) i mapę głębokości (`/camera/depth/image_rect_raw`). Oba tematy muszą mieć zgodny timestamp, żeby fuzja przestrzenna działała poprawnie.

```
import rclpy
from rclpy.node import Node
from sensor_msgs.msg import Image
from cv_bridge import CvBridge
import cv2

class PerceptionNode(Node):
    def __init__(self) -> None:
        super().__init__('perception_node')
        self.bridge = CvBridge()
        self.sub_rgb = self.create_subscription(
            Image, '/camera/color/image_raw', self.rgb_callback, 10
        )
        self.sub_depth = self.create_subscription(
            Image, '/camera/depth/image_rect_raw', self.depth_callback, 10
        )
        self.get_logger().info('PerceptionNode gotowy')

    def rgb_callback(self, msg: Image) -> None:
        frame = self.bridge.imgmsg_to_cv2(msg, desired_encoding='bgr8')
        self.get_logger().info(f'RGB frame: {frame.shape},
t={msg.header.stamp.sec}')

    def depth_callback(self, msg: Image) -> None:
        depth = self.bridge.imgmsg_to_cv2(msg, desired_encoding='passthrough')
        self.get_logger().info(f'Depth frame: {depth.shape}')
```

## 4.2. Minimalna walidacja kamery

- Obraz nie jest odwrócony, prześwietlony ani losowo pusty.
- Timestamps z RGB i depth różnią się o mniej niż 20 ms.
- Klatka debugowa zapisana do PNG wygląda tak, jak spodziewane otoczenie.
- ros2 topic hz /camera/color/image_raw wskazuje stabilne ~30 Hz.

## 4.3. Lidar Livox MID360

Lidar publikuje chmurę punktów na temacie `/lidar/points` w formacie `sensor_msgs/PointCloud2`. Zanim wykorzystasz dane do lokalizacji, potwierdź ramkę odniesienia i sprawdź zasięg w RViz.

```
import rclpy
from rclpy.node import Node
from sensor_msgs.msg import PointCloud2
import sensor_msgs_py.point_cloud2 as pc2

class LidarDebugNode(Node):
    def __init__(self) -> None:
        super().__init__('lidar_debug_node')
        self.sub = self.create_subscription(
            PointCloud2, '/lidar/points', self.callback, 5
        )

    def callback(self, msg: PointCloud2) -> None:
```

```
        points = list(pc2.read points(msg, field names=('x','y','z'),
skip nans=True))
        self.get logger().info(f'Punktów: {len(points)}, ramka:
{msg.header.frame id}')
```

**Wskazówka**

Uruchom RViz2 i dodaj wizualizację PointCloud2. Sprawdź, czy chmura punktów pokrywa się z rzeczywistym otoczeniem robota i czy nie ma artefaktów (płaszczyzna podłoża powinna być widoczna jako gęsty klaster punktów).

### 4.4. Detekcja obiektu w obrazie RGB

Dla pierwszych testów wystarczy detekcja koloru lub prostego kształtu. Nie zaczynam od modeli głębokiego uczenia — zaczynam od pewności, że mam poprawne dane.

```
import cv2
import numpy as np

def detect red object(frame bgr: np.ndarray) -> tuple[bool, tuple]:
    """Zwraca (found, (cx, cy, radius)) lub (False, ())."""
    hsv = cv2.cvtColor(frame bgr, cv2.COLOR BGR2HSV)
    lower = np.array([0, 120, 70])
    upper = np.array([10, 255, 255])
    mask = cv2.inRange(hsv, lower, upper)
    # zamknięcie morfologiczne eliminuje szum
    kernel = cv2.getStructuringElement(cv2.MORPH ELLIPSE, (7, 7))
    mask = cv2.morphologyEx(mask, cv2.MORPH CLOSE, kernel)
    contours,   = cv2.findContours(mask, cv2.RETR EXTERNAL,
cv2.CHAIN APPROX SIMPLE)
    if not contours:
        return False, ()
    c = max(contours, key=cv2.contourArea)
    if cv2.contourArea(c) < 500:  # filtr obszarowy
        return False, ()
    (cx, cy), r = cv2.minEnclosingCircle(c)
    return True, (int(cx), int(cy), int(r))
```

**Ostrzeżenie**

Detekcja koloru jest wrażliwa na oświetlenie. Przed testami na realnym robocie sprawdź maskę na kilku klatkach w różnych warunkach oświetleniowych. Fałszywe detekcje prowadzą do niepożądanego ruchu.

## 5. Fuzja sensoryczna i transformacje TF2

Detekcja w pikselach obrazu 2D musi zostać przetransformowana do przestrzeni 3D, a następnie do wspólnej ramki robota, zanim węzeł sterowania ją wykorzysta. Do tego służy **TF2** — biblioteka transformacji ROS 2.

### 5.1. Architektura transformacji

| Ramka (frame) | Znaczenie |
|---|---|
| world | Globalna ramka stała; używana do localization i mapowania. |
| base_link | Środek ciężkości robota; korzeń drzewa TF. |
| camera_link | Ramka montażu kamery; wyznacza kalibrację extrinsic. |
| lidar_link | Ramka lidaru; zwykle przesunięta względem camera_link. |

```
import rclpy
from rclpy.node import Node
from tf2_ros import Buffer, TransformListener
from geometry_msgs.msg import PoseStamped

class TFBridgeNode(Node):
    def __init__(self) -> None:
        super().__init__('tf_bridge_node')
        self.tf_buffer = Buffer()
        self.tf_listener = TransformListener(self.tf_buffer, self)

    def transform_to_base(self, pose_cam: PoseStamped) -> PoseStamped | None:
        try:
            return self.tf_buffer.transform(pose_cam, 'base_link')
        except Exception as e:
            self.get_logger().warn(f'TF error: {e}')
            return None
```

**Wskazówka**

Użyj ros2 run tf2_tools view_frames, żeby wygenerować diagram drzewa transformacji jako PDF. Brak połączenia między camera_link a base_link to najczęstszy powód, przez który obiekt "ginie" po transformacji.

### 5.2. Wyznaczanie głębokości obiektu

Mając piksel (cx, cy) z detekcji RGB i obraz głębokości, możesz wyliczyć pozycję 3D przez model pinhole kamery. Parametry intrinsic znajdziesz w temacie `/camera/color/camera_info`.

```
import numpy as np

def pixel_to_3d(cx: int, cy: int, depth_m: float,
                fx: float, fy: float, ppx: float, ppy: float
                ) -> np.ndarray:
    """Zwraca wektor [X, Y, Z] w ramce kamery (metry)."""
    x = (cx - ppx) * depth_m / fx
    y = (cy - ppy) * depth_m / fy
    return np.array([x, y, depth_m])

# Przykładowe parametry — pobierz z /camera/color/camera_info
# fx, fy ~ 615.0, ppx ~ 320.0, ppy ~ 240.0  (640x480)
```

## 6. Planowanie ruchu i maszyna stanów (FSM)

Maszyna stanów oddziela logikę zadaniową od percepcji i sterowania. Dzięki temu każdy etap zadania jest testowalny niezależnie i można wskazać dokładnie, w którym stanie robot się zatrzymał.

## 6.1. FSM dla zadania "podejdź i chwyć"

```
from enum import Enum, auto
from dataclasses import dataclass, field
from typing import Optional
import time

class TaskState(Enum):
    SEARCH   = auto()  # szukaj obiektu w FOV kamery
    APPROACH = auto()  # idź w kierunku obiektu
    ALIGN    = auto()  # ustaw ramię i orientację
    GRASP    = auto()  # wykonaj chwyt
    LIFT     = auto()  # unieś obiekt
    PLACE    = auto()  # odłóż w docelowym miejscu
    DONE     = auto()  # zadanie zakończone
    ERROR    = auto()  # stan awaryjny

@dataclass
class TaskContext:
    state: TaskState = TaskState.SEARCH
    object_pose: Optional[object] = None
    retries: int = 0
    state_entered_at: float = field(default_factory=time.time)

    def transition_to(self, new_state: TaskState) -> None:
        print(f'[FSM] {self.state.name} → {new_state.name}')
        self.state = new_state
        self.state_entered_at = time.time()

    def time_in_state(self) -> float:
        return time.time() - self.state_entered_at
```

## 6.2. Pętla FSM w węźle ROS 2

```
import rclpy
from rclpy.node import Node
from geometry_msgs.msg import PoseStamped

TIMEOUT_SEARCH   = 30.0  # s — po tym czasie przejdź do ERROR
TIMEOUT_APPROACH = 20.0
DISTANCE_GRASP   = 0.35  # m — odległość do uruchomienia chwytu

class TaskManagerNode(Node):
    def __init__(self) -> None:
        super().__init__('task_manager_node')
        self.ctx = TaskContext()
        self.sub_pose = self.create_subscription(
            PoseStamped, '/detected_object/pose', self.pose_callback, 10
        )
        self.timer = self.create_timer(0.1, self.step)

    def pose_callback(self, msg: PoseStamped) -> None:
        self.ctx.object_pose = msg

    def step(self) -> None:
        match self.ctx.state:
```

```
            case TaskState.SEARCH:
                if self.ctx.object pose is not None:
                    self.ctx.transition to(TaskState.APPROACH)
                elif self.ctx.time in state() > TIMEOUT SEARCH:
                    self.ctx.transition to(TaskState.ERROR)
            case TaskState.APPROACH:
                dist = self. distance to object()
                if dist < DISTANCE GRASP:
                    self.ctx.transition to(TaskState.ALIGN)
                elif self.ctx.time in state() > TIMEOUT APPROACH:
                    self.ctx.transition to(TaskState.ERROR)
            # kolejne stany implementowane analogicznie
            case TaskState.ERROR:
                self.get logger().error('FSM: stan awaryjny — przejmij kontrolę 
pilotem!')

    def  distance to object(self) -> float:
        if self.ctx.object pose is None:
            return float('inf')
        p = self.ctx.object pose.pose.position
        return (p.x**2 + p.y**2) ** 0.5
```

**Wskazówka**

Każde przejście między stanami loguj z timestampem i nazwami stanów. Bez tego rekonstrukcja przebiegu eksperymentu po fakcie jest niemożliwa.

### 6.3. Warunki błędu i tranzycje awaryjne

| Warunek błędu | Zalecana tranzycja |
|---|---|
| Brak detekcji obiektu przez > 30 s | SEARCH → ERROR; zażądaj przejęcia pilotem |
| Odległość nie maleje przez > 20 s | APPROACH → ERROR; sprawdź lokalizację obiektu |
| MoveIt 2 zgłasza brak rozwiązania IK | ALIGN → ERROR; zmień pozycję bazową i powtórz |
| Moment stawu przekracza próg bezpieczeństwa | Dowolny → ERROR; natychmiastowy stop |
| Strata połączenia z g1_driver | Dowolny → ERROR; pilot przejmuje kontrolę |

## 7. MoveIt 2 i trajektorie ramienia

MoveIt 2 dostarcza planowanie trajektorii z uwzględnieniem kolizji, limitów stawów i ograniczeń kinematycznych. Na etapie M2.6 używasz go do zaplanowania ruchu jednego ramienia G1 do wyliczonej pozycji celu.

**Ostrzeżenie**

MoveIt 2 wprowadzaj dopiero po potwierdzeniu M2.5. Każde niepowodzenie planera powinno prowadzić do stanu ERROR w FSM, nie do ponownej próby w nieskończonej pętli.

## 7.1. Konfiguracja MoveGroup

```python
import rclpy
from rclpy.node import Node
from moveit.planning import MoveItPy
from geometry_msgs.msg import Pose

class ArmControllerNode(Node):
    def __init__(self) -> None:
        super().__init__('arm_controller_node')
        self.moveit = MoveItPy(node_name='arm_moveit')
        self.arm_group = self.moveit.get_planning_component('right_arm')

    def move_to_pose(self, target: Pose) -> bool:
        self.arm_group.set_start_state_to_current_state()
        self.arm_group.set_goal_state(pose_stamped_msg=target,
                                      pose_link='right_hand_palm')
        plan = self.arm_group.plan()
        if plan:
            self.moveit.execute(plan, controllers=[])
            return True
        self.get_logger().warn('Planowanie trajektorii nie powiodło się.')
        return False
```

## 7.2. Bezpieczne granice ruchu ramienia

| Parametr | Wartość testowa (start) |
|---|---|
| Prędkość skalująca (velocity_scaling) | 0.1 — 10% maks. prędkości |
| Przyspieszenie skalujące (acceleration_scaling) | 0.1 |
| Czas planowania (planning_time) | 5 s |
| Liczba prób planowania (num_planning_attempts) | 5 |
| Cel chwytaka (end_effector_link) | right_hand_palm |

**Wskazówka**

Zacznij od velocity_scaling = 0.05. Zwiększaj dopiero gdy trajektoria jest powtarzalna i zgodna z wizualizacją w RViz. Nigdy nie zmieniaj skali prędkości w połowie zadania.

## 7.3. Wizualizacja w RViz2

```bash
# Uruchom RViz2 z konfiguracją MoveIt
ros2 launch moveit_ros_visualization moveit_rviz.launch.py
# Lub użyj konfiguracji z bringup:
ros2 launch g1_bringup rviz.launch.py

# Elementy do dodania w RViz:
#   MotionPlanning plugin (MoveIt 2)
#   PointCloud2          → /lidar/points
#   Image                → /camera/color/image_raw
#   TF                   → wszystkie ramki
#   MarkerArray          → /detected_object/markers
```

# 8. Logowanie, testy i debugowanie

Dobry eksperyment to taki, który można odtworzyć i zrozumieć po tygodniu, a nie tylko uruchomić teraz. Standard logowania nie zmienia się między ćwiczeniami.

## 8.1. Tabela diagnostyczna

| Objaw | Co sprawdzić najpierw |
|---|---|
| Brak obrazu z kamery | ros2 topic list, kabel USB, uprawnienia /dev/video*, realsense-viewer |
| Pusta chmura punktów | Adres IP lidaru 192.168.123.20, temat /lidar/points, ramka TF lidar_link |
| FSM nie przechodzi ze SEARCH | Czy detekcja publikuje /detected_object/pose? Sprawdź ros2 topic echo. |
| MoveIt 2: brak rozwiązania IK | Sprawdź, czy cel jest w zasięgu kinematycznym; zmniejsz odległość lub zmień postawę bazy. |
| Drgania ramienia | Obniż velocity_scaling; sprawdź czy g1_driver nie nadpisuje komend. |
| TF: błąd 'no transform' | view_frames, sprawdź czy wszystkie linki są publikowane przez driver. |

## 8.2. Nagrywanie i odtwarzanie bagów

```
# Nagrywanie wybranych tematów
ros2 bag record -o lab2 session \
    /camera/color/image raw \
    /camera/depth/image rect raw \
    /lidar/points \
    /unitree/joint states \
    /detected object/pose \
    /tf /tf static

# Odtwarzanie (debug bez robota)
ros2 bag play lab2 session --rate 0.5
```

**Wskazówka**

Debuguj algorytm percepcji najpierw na nagraniu bag, a nie na żywo z robotem. Oszczędza to czas i baterie.

## 8.3. Checklista przed testem na realnym robocie

- ☐ Pilot naładowany, operator gotowy, procedura awaryjna powtórzona ustnie przez zespół.
- ☐ Ten sam algorytm przeszedł test na nagraniu bag lub w symulacji.
- ☐ velocity_scaling ustawiony na 0.05 lub niżej dla pierwszego uruchomienia.
- ☐ FSM ma jawny stan ERROR i log każdego przejścia.
- ☐ Logujesz bag ze wszystkimi tematami wymienionymi w sekcji 8.2.
- ☐ Jedna osoba obserwuje wyłącznie bezpieczeństwo i nie patrzy na terminal.

## 9. Ćwiczenia laboratoryjne i ścieżka rozwoju

1. Uruchom węzeł percepcji i zapisz do PNG 5 klatek debugowych w różnych warunkach oświetleniowych. Oceń, jak zmienia się jakość detekcji.
2. Zaimplementuj transformację piksel → 3D i opublikuj PoseStamped wykrytego obiektu w ramce camera_link. Sprawdź wynik w RViz.
3. Dodaj transformację z camera_link do base_link przez TF2 i potwierdź, że pozycja obiektu w ramce base_link jest stabilna podczas drobnych ruchów głowy.
4. Zbuduj maszynę stanów FSM dla sekwencji SEARCH → APPROACH → ALIGN. Każde przejście loguj z timestampem i spowodowanym warunkiem.
5. Zaplanuj trajektorię ramienia do pozycji stałej przez MoveIt 2 z velocity_scaling = 0.05. Zmierz czas planowania i wykonania.
6. Nagraj bag z całą sesją i odtwórz go, uruchamiając tylko węzeł percepcji. Porównaj jakość detekcji online i offline.

### Kierunki dalszego rozwoju

- Integracja dłoni Inspire / Dex3-1 — sterowanie chwytaczem przez ROS 2 action.
- Modele percepcyjne z PyTorch — detekcja obiektów przez YOLO lub DepthAnything.
- Lokalizacja 3D obiektu z lidaru — segmentacja płaszczyzny i klastrowanie chmury punktów.
- Transfer do symulacji — przetestowanie pełnej ścieżki SEARCH→PLACE w Isaac Lab.
- Teleoperacja do zbierania demonstracji — pipeline do uczenia z imitacji (BC/robomimic).

# Załącznik A. Minimalna struktura repozytorium po ćwiczeniu 2

```
g1 project/
├── README.md             # wersje SDK, ROS 2, CUDA, commit, seed
├── docs/
|   └── tf frames.pdf     # diagram z view frames
├── ros2 ws/
|   └── src/
|       ├── g1 bringup/     # launch files
|       ├── g1 driver/
|       ├── g1 control/     # FSM, arm controller
|       ├── g1 perception/ # kamera, lidar, detekcja
|       └── g1 interfaces/ # własne msg/srv
├── bags/
|   └── lab2 session/     # nagrania ROS bag
├── experiments/
|   └── logs/             # CSV komendy, stany stawów
└── scripts/
    ├── bootstrap.sh
    └── run lab2.sh
```

## Załącznik B. Dobre praktyki

- Każdy węzeł ma jeden subskrybowany typ danych wejściowych i jeden publikowany temat wyjściowy. Nie mieszaj percepcji z logiką sterowania w jednym pliku.

- Przed każdą zmianą na realnym robocie uruchom 30-sekundowy test na nagraniu bag.
- Warunki błędu i timeouty FSM definiuj jako stałe na górze pliku, nie jako magic numbers w kodzie.
- Zawsze uruchamiaj system przez launch file z pełnym logowaniem (--log-level INFO).
- TF tree sprawdzaj view_frames na początku każdej sesji — drzewo może się zmienić po aktualizacji drivera.
- Nazwy tematów i parametrów trzymaj konsekwentnie po angielsku i dokumentuj w README.

## Załącznik C. Materiały źródłowe

- **Unitree SDK2:** github.com/unitreerobotics/unitree_sdk2_python
- **ROS 2 Humble:** docs.ros.org/en/humble
- **MoveIt 2:** moveit.ros.org
- **TF2 tutorials:** docs.ros.org/en/humble/Tutorials/Intermediate/Tf2
- **Intel RealSense ROS2:** github.com/IntelRealSense/realsense-ros
- **Livox ROS2 driver:** github.com/Livox-SDK/livox_ros_driver2
- **Instrukcja Ćwiczenia 1:** Instrukcja_Unitree_G1.pdf — niniejszy zestaw dokumentów

# Ćwiczenie 3: Symulacja, uczenie i transfer sim-to-real

**Uwaga**

**Skrót dla prowadzącego:** Dopuść zespół do tej instrukcji dopiero gdy: (1) samodzielnie przeprowadził pełną sesję diagnostyczną na realnym robocie, (2) zbudował działającą FSM z percepcją w ROS 2, (3) zaplanował i wykonał co najmniej jedną trajektorię ramienia przez MoveIt 2. Uczenie w symulacji ma sens tylko przy dobrze zdefiniowanym interfejsie obserwacji i akcji.

## Mapa dokumentu


| Sekcja | Po co istnieje |
|---|---|
| 1. Cel i zakres ćwiczenia | Określa co student osiągnie i jakich pułapek unikać. |
| 2. Środowisko symulacyjne Isaac Lab | Instalacja, weryfikacja GPU i pierwsze uruchomienie świata. |
| 3. Model robota i interfejs środowiska | URDF/USD, przestrzeń obserwacji, akcji i warunki końca. |
| 4. Shaping nagrody i curriculum | Projektowanie funkcji nagrody etapami; analiza krzywych uczenia. |
| 5. Uczenie ze wzmocnieniem (RL) | PPO, rsl-rl, konfiguracja treningu i walidacja w symulacji. |
| 6. Uczenie z imitacji (IL/BC) | Teleoperacja, zapis demonstracji HDF5, trening BC/robomimic. |
| 7. Domain randomization i robustness | Randomizacja fizyki, sensorów i geometrii; ocena odporności. |
| 8. Transfer sim-to-real | Protokół bezpiecznego przenoszenia polityki na realny robot. |
| 9. Logowanie, testy i debugowanie | Standard odbioru, checklisty i diagnoza częstych błędów. |
| 10. Ćwiczenia i ścieżka rozwoju | Konkretne zadania i kierunki projektu badawczego. |
| Załączniki | Struktura repozytorium, checklisty i literatura. |


## 1. Cel i zakres ćwiczenia

Ćwiczenie 3 zamyka pętlę projektu: od definicji zadania przez uczenie w symulacji aż po bezpieczny transfer polityki na platformę G1 EDU. Zakłada, że masz **sprawne środowisko ROS 2, działającą percepcję i przetestowaną FSM** z Ćwiczenia 2. Celem nie jest uruchomienie treningu jak najszybciej — celem jest wyuczenie polityki, która **przejdzie walidację na wielu seedach** i zachowa się bezpiecznie na realnym robocie.

**Czego NIE robić na tym etapie:**

- Nie uruchamiaj transferu sim-to-real zanim polityka nie przejdzie testu na co najmniej 5 różnych seedach w symulacji.
- Nie trenuj z nagrodą końcową jako jedynym sygnałem. Dla zadań manipulacyjnych nagroda jest zbyt rzadka.
- Nie ignoruj domain randomization: model wytrenowany bez losowości fizyki prawie zawsze jest kruchy na realnym sprzęcie.

- Nie przechodź do IL/BC bez obejrzenia przynajmniej 10 demonstracji — jakość danych determinuje jakość polityki.

### Kamienie milowe ćwiczenia 3

| Etap | Kryterium wyjścia |
| --- | --- |
| M3.1: Środowisko symulacyjne | Isaac Lab ładuje scenę z modelem G1; nvidia-smi potwierdza aktywne GPU. |
| M3.2: Interfejs środowiska | Własna klasa środowiska zwraca poprawne obserwacje i przyjmuje akcje bez błędów kształtu tensora. |
| M3.3: Funkcja nagrody | Rollout 100 epizodów pokazuje niezerową nagrodę i rosnący trend — nawet bez treningu, przez random policy. |
| M3.4: Trening RL (PPO) | Krzywa episodic reward rośnie monotonicznie; model z checkpoint_final przechodzi walidację na 5 seedach. |
| M3.5: Demonstracje (IL) | Zbiór ≥ 20 demonstracji zapisany w HDF5; obejrzany i zaakceptowany przez prowadzącego. |
| M3.6: Trening BC | Polityka BC osiąga success rate > 70% w symulacji na zestawie ewaluacyjnym. |
| M3.7: Domain randomization | Success rate spada < 10 pp przy ±20% randomizacji masy i tarcia — jeśli więcej, wzmocnij DR. |
| M3.8: Transfer sim-to-real | Polityka wykonuje zadanie na realnym robocie w 3/5 próbach przy velocity_scaling ≤ 0.15. |

## 2. Środowisko symulacyjne Isaac Lab

Isaac Lab (dawniej Isaac Gym / OmniIsaac) to platforma NVIDIA do równoległej symulacji robotycznej na GPU. Zapewnia **tysiące równoległych środowisk** w jednym procesie, co drastycznie skraca czas treningu. Na zajęciach korzystamy z wersji uzgodnionej dla całego kursu. Nie aktualizuj samodzielnie bez zgody prowadzącego.

### 2.1. Wymagania sprzętowe i weryfikacja GPU

| Komponent | Minimalne wymaganie |
| --- | --- |
| GPU | NVIDIA RTX 3080 / A4000 lub wyższy; VRAM ≥ 10 GB |
| CUDA | 12.1 lub wyższy (zgodny z zainstalowanym sterownikiem) |
| RAM | 32 GB; zalecane 64 GB przy num_envs > 1024 |
| Dysk | SSD NVMe; min. 50 GB wolnego miejsca na USD assets |
| System | Ubuntu 22.04 LTS; sterownik NVIDIA ≥ 535 |

```
# Weryfikacja GPU i CUDA
nvidia-smi
nvcc --version

# Aktywacja środowiska Isaac Lab
conda activate env isaaclab
```

```
python -c "import torch; print(torch.cuda.is_available(), torch.version.cuda)"

# Uruchomienie prostego testu środowiska (headless na serwerze)
python scripts/run_env.py --task G1-Reach-v0 --num_envs 64 --headless
```

**Błąd / Diagnostyka**

**CUDA unavailable:** Sprawdź nvidia-smi, wersję sterownika i wheel PyTorch. Niekompatybilność CUDA ↔ PyTorch to najczęstszy powód niedziałającego Isaac Lab. Zainstaluj wheel pasujący do wersji CUDA z nvidia-smi, nie z nvcc.

### 2.2. Struktura katalogów Isaac Lab

```
IsaacLab/
├── source/
|   ├── isaaclab/              # rdzień platformy (nie modyfikuj)
|   ├── isaaclab_tasks/        # gotowe środowiska referencyjne
|   └── isaaclab_assets/       # modele USD robotów i obiektów
├── scripts/
|   ├── run_env.py             # szybki test środowiska
|   └── train.py               # główny skrypt treningu
└── logs/                      # checkpointy i TensorBoard

# Własne środowisko G1 umieszczasz w:
source/isaaclab_tasks/isaaclab_tasks/g1_tasks/
```

### 2.3. Pierwsze uruchomienie sceny G1

```
conda activate env_isaaclab
cd ~/IsaacLab

# Tryb wizualny (lokalny komputer z GPU i monitorem)
python scripts/run_env.py --task G1-Reach-v0 --num_envs 16

# Tryb headless (serwer / zdalne połączenie)
python scripts/run_env.py --task G1-Reach-v0 --num_envs 256 --headless

# Streaming do przeglądarki przez LiveStream (jeśli dostępne)
python scripts/run_env.py --task G1-Reach-v0 --num_envs 16 --livestream 2
```

**Wskazówka**

Przy pierwszym uruchomieniu Isaac Lab kompiluje shadery i ładuje USD assets — może to trwać kilka minut. Kolejne uruchomienia są znacznie szybsze dzięki cache. Nie przerywaj pierwszego uruchomienia.

## 3. Model robota i interfejs środowiska

Własne środowisko dla G1 dziedziczy z klasy `DirectRLEnv` lub `ManagerBasedRLEnv`. Musisz zdefiniować cztery elementy: **obserwacje, akcje, nagrodę i warunki końca epizodu**. Od precyzji tych definicji zależy, czy polityka będzie działać na realnym robocie.

## 3.1. Przestrzeń obserwacji

Obserwacje to wejście do sieci polityki. Muszą być możliwe do odtworzenia na realnym robocie — nie używaj danych, których nie możesz uzyskać z sensorów G1.

| Obserwacja | Źródło na realnym G1 |
|---|---|
| Pozycje stawów (29 DOF) | unitree_sdk2: LowState.motor_state[i].q |
| Prędkości stawów (29 DOF) | unitree_sdk2: LowState.motor_state[i].dq |
| Orientacja IMU (roll, pitch, yaw) | LowState.imu_state.rpy |
| Prędkość kątowa IMU (3 DOF) | LowState.imu_state.gyroscope |
| Pozycja docelowa w base_link (3D) | TF2 transform z /detected_object/pose |
| Historia akcji (opcjonalnie) | Bufor wewnętrzny polityki |

**Ostrzeżenie**

Nie dodawaj do obserwacji danych niedostępnych na realnym sprzęcie (np. ground-truth pozycji z symulatora). Polityka nauczona na takich danych nie przeniesie się na rzeczywistość.

```
import torch
from isaaclab.envs import DirectRLEnv, DirectRLEnvCfg
from isaaclab.utils import configclass

@configclass
class G1ReachEnvCfg(DirectRLEnvCfg):
    # Środowisko
    num envs: int = 512
    episode length s: float = 8.0
    decimation: int = 4           # krok polityki co 4 kroki fizyki
    # Przestrzenie
    observation space: int = 29 + 29 + 3 + 3 + 3  # q, dq, rpy, gyro, goal
    action space:       int = 12  # tylko stawy ramion i talii
    state space:        int = 0

class G1ReachEnv(DirectRLEnv):
    cfg: G1ReachEnvCfg

    def  get observations(self) -> dict:
        q   = self.robot.data.joint pos[:, self.arm joint ids]
        dq  = self.robot.data.joint vel[:, self.arm joint ids]
        rpy = self.robot.data.root ang vel w          # przybliżenie IMU
        obs = torch.cat([q, dq, rpy, self.goal pos b], dim=-1)
        return {"policy": obs}

    def  apply action(self) -> None:
        # Akcje jako docelowe pozycje stawów (position control)
        target q = self.actions * self.cfg.action scale
        self.robot.set joint position target(target q,
joint ids=self.arm joint ids)
```

## 3.2. Warunki końca epizodu

| Warunek | Zalecana obsługa |
|---|---|
| Sukces: efector w odległości < 3 cm od celu | terminated=True; bonus nagrody +10 |
| Timeout: przekroczono episode_length_s | truncated=True; brak kary |
| Upadek: roll lub pitch > 60° | terminated=True; kara -5 |
| Kolizja: sila kontaktu > próg | terminated=True; kara -2 |
| NaN w akcji lub obserwacji | terminated=True; log błędu; reset |

## 4. Projektowanie funkcji nagrody

Funkcja nagrody to **najważniejszy element projektu RL**. Zła nagroda jest częstszym powodem niepowodzenia niż zły algorytm. Projektuj ją iteracyjnie: zacznij od prostego sygnału postępu, sprawdź rollout z losową polityką, dopiero potem dodawaj człony korekcyjne.

### 4.1. Anatomia funkcji nagrody

```
import torch

def compute_reward(
    ee_pos: torch.Tensor,        # pozycja efektora [N, 3]
    goal_pos: torch.Tensor,      # pozycja celu     [N, 3]
    joint_vel: torch.Tensor,     # prędkości stawów [N, DOF]
    contact_forces: torch.Tensor,  # siły kontaktu [N]
    success_mask: torch.Tensor,    # bool [N]
) -> torch.Tensor:
    """Zwraca reward [N] dla każdego środowiska."""

    dist = torch.norm(ee_pos - goal_pos, dim=-1)    # [N]

    # 1. Postęp: malejąca funkcja odległości (główny sygnał)
    r_progress = torch.exp(-4.0 * dist)             # 0→1 gdy dist→0

    # 2. Bonus sukcesu (rzadki, ale silny)
    r_success  = 10.0 * success_mask.float()

    # 3. Kara za prędkość stawów (gładkość ruchu)
    r_smooth   = -0.002 * torch.sum(joint_vel ** 2, dim=-1)

    # 4. Kara za siły kontaktowe (bezpieczeństwo)
    r_contact  = -0.05 * torch.clamp(contact_forces - 5.0, min=0.0)

    return r_progress + r_success + r_smooth + r_contact
```

### 4.2. Zasady projektowania nagrody

| Zasada | Uzasadnienie |
|---|---|
| Sygnał postępu powinien być niezerowy od pierwszego kroku | Nagroda końcowa dla trudnych zadań jest zbyt rzadka; agent nie uczy się kierunku. |

| Używaj torch.exp(-k * dist) zamiast -dist | Wykładniczy kształt daje silniejszy gradient blisko celu. |
| --- | --- |
| Kary mnoż przez małe współczynniki (0.001–0.05) | Zbyt duże kary powodują, że agent unika jakiegokolwiek ruchu. |
| Sprawdzaj skalę nagród: mean(reward) powinno być w [-1, 5] | Skrajne wartości destabilizują trening PPO (clipping staje się nieefektywny). |
| Nie nagradzaj za coś, czego nie możesz zmierzyć na robocie | Symulator może udostępniać ground-truth niedostępny w rzeczywistości. |

### 4.3. Curriculum learning

Curriculum to stopniowe zwiększanie trudności zadania w trakcie treningu. Dla zadań chwytania najskuteczniejszy jest curriculum odległości: zacznij od celów blisko efektora, przesuwaj je dalej w miarę poprawy agenta.

```
class CurriculumManager:
    def __init__(self, start_dist=0.10, max_dist=0.50, threshold=0.7):
        self.max_dist  = start_dist
        self.target_max = max_dist
        self.threshold  = threshold   # success rate do awansu

    def step(self, success_rate: float) -> None:
        """Wywołuj po każdej epoce treningu."""
        if success_rate >= self.threshold:
            self.max_dist = min(self.max_dist * 1.15, self.target_max)
            print(f'[Curriculum] max_dist → {self.max_dist:.3f} m')

    def sample_goal(self, n: int) -> torch.Tensor:
        r   = torch.rand(n)  * self.max_dist
        phi = torch.rand(n)  * 2 * 3.14159
        return torch.stack([r * torch.cos(phi), r * torch.sin(phi),
                            torch.zeros(n)], dim=-1)
```

**Kluczowa zasada**

Naucz się czytać krzywe TensorBoard zanim zaczniesz modyfikować nagrodę. Nagroda rosnąca przez pierwsze 500 iteracji, a następnie plateau, wskazuje na pułapkę lokalną — zmień curriculum, nie architekturę sieci.

## 5. Uczenie ze wzmocnieniem - PPO z rsl-rl

Na zajęciach używamy biblioteki **rsl-rl** (RSL Reinforcement Learning), zaprojektowanej do treningu polityk dla robotów humanoidalnych. Jest zoptymalizowana pod kątem Isaac Lab i domyślnie wspiera rollout na GPU.

### 5.1. Konfiguracja PPO

```
# configs/ppo_g1_reach.yaml
seed: 42
device: cuda:0
```

```yaml
runner:
  num steps per env: 24        # horyzont rollout na środowisko
  max iterations:    1500      # epoki treningu
  save interval:     100       # co ile iteracji zapisać checkpoint
  experiment name:   g1 reach v1
  run name: "ppo seed42"

policy:
  class name: ActorCritic
  actor hidden dims:  [512, 256, 128]
  critic hidden dims: [512, 256, 128]
  activation: elu
  init noise std: 1.0

algorithm:
  class name: PPO
  value loss coef:   1.0
  use clipping:      true
  clip param:        0.2
  entropy coef:      0.01
  num learning epochs: 5
  num mini batches:  4
  learning rate:     1.0e-3
  gamma:             0.99
  lam:               0.95
  desired kl:        0.01
  max grad norm:     1.0
```

## 5.2. Uruchomienie treningu

```bash
conda activate env isaaclab
cd ~/IsaacLab

# Trening (headless, logi do TensorBoard)
python scripts/train.py \
    --task G1-Reach-v0 \
    --num envs 512 \
    --headless \
    --seed 42 \
    --cfg configs/ppo g1 reach.yaml

# Monitoring w TensorBoard (osobny terminal)
tensorboard --logdir logs/g1 reach v1 --port 6006

# Ewaluacja checkpointu
python scripts/play.py \
    --task G1-Reach-v0 \
    --num envs 32 \
    --checkpoint logs/g1 reach v1/checkpoint final.pt
```

## 5.3. Interpretacja krzywych TensorBoard

| Metryka | Oczekiwane zachowanie | Sygnał problemu |
|---|---|---|
| Episode Reward Mean | Rośnie monotonicznie, plateau po konwergencji | Oscyluje lub spada po 200 it. → sprawdź nagrodę |

| Episode Length Mean | Rośnie (agent żyje dłużej) lub maleje (osiąga cel szybciej) | Natychmiast 0 → robot pada w pierwszym kroku |
|---|---|---|
| Value Loss | Maleje i stabilizuje się | Rosnąca lub eksplodująca → zmniejsz lr |
| Surrogate Loss | Blisko 0, małe wahania | Duże wahania → zmniejsz clip_param |
| KL Divergence | Blisko desired_kl (0.01) | Duże → lr za duże; małe → lr za małe |
| Success Rate (własna) | Rośnie, osiąga ≥ 70% | Plateau < 30% po 500 it. → curriculum lub nagroda |

**Wskazówka**

Trenuj zawsze na co najmniej 3 różnych seedach (np. 42, 123, 777). Jeśli jeden seed konwerguje a inne nie, wynik może być dziełem przypadku, a nie skuteczności algorytmu.

## 6. Uczenie z imitacji (IL) i teleoperacja

Uczenie z imitacji jest alternatywą lub uzupełnieniem RL. Zamiast definiować nagrodę, **pokazujesz robotowi jak wykonać zadanie** przez teleoperację i uczysz politykę imitować te demonstracje (Behavioral Cloning, BC).

### 6.1. Zbieranie demonstracji

Demonstracje zbierasz w symulacji przez teleoperację klawiaturą, SpaceMouse lub kontrolerem. Każda demonstracja musi być: gładka, powtarzalna i podzielona na czytelne podzadania.

```
# Teleoperacja klawiaturą w Isaac Lab
python scripts/teleop.py \
    --task G1-Reach-Teleop-v0 \
    --num envs 1 \
    --device keyboard \
    --output dir datasets/g1 reach demos/

# Konwersja surowych danych do HDF5 (format robomimic)
python scripts/convert to hdf5.py \
    --input dir datasets/g1 reach demos/ \
    --output    datasets/g1 reach.hdf5
```

### 6.2. Wymagania jakościowe demonstracji

- Każda demonstracja kończy się sukcesem — odrzucaj nieudane próby.
- Ruch jest gładki: brak gwałtownych szarpnięć widocznych w krzywej pozycji stawów.
- Liczba demonstracji: minimum 20 dla prostych zadań, 100+ dla złożonych.
- Zbiór obejrzany w całości przed treningiem — błędna demonstracja zatruwa cały dataset.
- Podzadania oznaczone przez subtask_id w metadanych HDF5.

### 6.3. Trening Behavioral Cloning

```
# Trening BC przez robomimic
conda activate env isaaclab

python robomimic/scripts/train.py \
    --config configs/bc g1 reach.json \
    --dataset datasets/g1 reach.hdf5 \
    --output dir logs/bc g1 reach v1/

# Fragment configs/bc g1 reach.json:
# {
#   "algo": "bc",
#   "train": { "batch size": 100, "num epochs": 600 },
#   "observation": {
#     "modalities": { "obs": { "low dim": ["joint pos", "joint vel", "goal pos"]
} }
#   }
# }
```

**Uwaga**

Przy ocenie polityki BC nie polegaj wyłącznie na ostatnim checkpointcie — przetestuj kilka epok, bo model z epoki 400 często jest lepszy od finalnego (overfitting na demonstracjach).

### 6.4. Porównanie RL vs IL

| Kryterium | RL (PPO) | IL (BC) |
| --- | --- | --- |
| Wymagane dane wejściowe | Funkcja nagrody | Demonstracje eksperta |
| Czas przygotowania | Projekt nagrody (trudny) | Zbieranie danych (pracochłonne) |
| Czas treningu | Długi (tysiące iteracji) | Krótki (setki epok) |
| Generalizacja | Dobra przy DR | Ograniczona do zakresu demo |
| Zachowanie poza dystrybucją | Chaotyczne, ale adaptuje się | Cicho zawodzi (distribution shift) |
| Zalecane użycie na G1 | Lokomocja, złożone manipulacje | Proste sekwencje chwytania |

## 7. Domain Randomization i ocena odporności

Domain Randomization (DR) to technika, która zmniejsza przepaść między symulacją, a rzeczywistością przez **losowanie parametrów fizyki i sensorów** podczas treningu. Polityka nauczona na szerszej przestrzeni parametrów jest bardziej odporna na niedokładności modelu rzeczywistego robota.

### 7.1. Parametry do randomizacji dla G1

| Parametr DR | Zakres (±% wartości nominalnej) |
| --- | --- |
| Masa segmentów ciała | ±15% |

| Środek masy segmentów | ±5% długości segmentu |
|---|---|
| Tarcie stawów (damping) | ±20% |
| Sztywność stawów (stiffness) | ±10% |
| Opóźnienie akcji (action latency) | 0–20 ms |
| Szum obserwacji (pozycja stawów) | Gaussa σ=0.01 rad |
| Szum obserwacji (prędkość stawów) | Gaussa σ=0.05 rad/s |
| Tarcie podłoża | ±30% |
| Skala grawitacji | ±2% |

```
# Fragment konfiguracji DR w Isaac Lab
from isaaclab.utils.noise import GaussianNoiseCfg, UniformNoiseCfg

randomization_cfg = {
    # Szum dodawany do obserwacji na każdym kroku
    "joint_pos_noise": GaussianNoiseCfg(mean=0.0, std=0.01),
    "joint_vel_noise": GaussianNoiseCfg(mean=0.0, std=0.05),

    # Perturbacje fizyki na początku każdego epizodu
    "mass_scale":    UniformNoiseCfg(min=0.85, max=1.15),
    "friction_scale": UniformNoiseCfg(min=0.70, max=1.30),
    "action_delay_steps": 2,   # stałe 2 kroki = 20 ms przy 100 Hz
}
```

## 7.2. Protokół oceny odporności

Po treningu z DR oceń politykę systematycznie, zmieniając jeden parametr na raz.

```
# Ewaluacja przy różnych poziomach perturbacji masy
for mass_scale in [0.7, 0.85, 1.0, 1.15, 1.3]:
    success = evaluate_policy(
        checkpoint='checkpoint_final.pt',
        num_episodes=100,
        mass_scale=mass_scale
    )
    print(f'mass_scale={mass_scale:.2f}  success_rate={success:.1%}')

# Oczekiwany wynik (przykład):
# mass_scale=0.70  success_rate=68.0%
# mass_scale=0.85  success_rate=79.0%
# mass_scale=1.00  success_rate=85.0%
# mass_scale=1.15  success_rate=81.0%
# mass_scale=1.30  success_rate=72.0%
```

**Kluczowa zasada**

Jeśli success rate spada o więcej niż 20 pp przy ±20% perturbacji masy, polityka jest zbyt krucha na transfer sim-to-real. Wróć do treningu z szerszym zakresem DR lub dodaj adaptację online.

# 8. Transfer sim-to-real i protokół bezpieczeństwa

Transfer sim-to-real jest **najbardziej ryzykownym etapem projektu.** Polityka nauczona w symulacji nigdy nie widziała prawdziwego robota. Każde uruchomienie na G1 EDU wymaga przestrzegania poniższego protokołu — bez wyjątku.

## 8.1. Checklisty przed pierwszym uruchomieniem polityki na G1

### Warunki konieczne (blokujące)

- Polityka przeszła walidację na ≥ 5 seedach i ≥ 3 poziomach DR masy i tarcia.
- Maksymalna akcja w zbiorze ewaluacyjnym mieści się w dopuszczalnym zakresie stawów.
- velocity_scaling w MoveIt 2 (lub ekwiwalent w SDK) ustawiony na ≤ 0.10.
- Akcje polityki są filtrowane: zastosowano filtr dolnoprzepustowy lub ograniczenie Δq/krok.
- Jawny limit czasu wykonania: ≤ 10 s na jeden epizod testowy.
- Pilot w ręce operatora; procedura awaryjna powtórzona ustnie przez cały zespół.
- Robot na wieszaku lub w stabilnej pozycji przy pierwszym uruchomieniu (jeśli dotyczy lokomocji).

### Sekwencja uruchomienia

- Uruchom robota w damping mode; sprawdź stan stawów programem diagnostycznym z Ćwiczenia 1.
- Wczytaj checkpoint polityki i wykonaj 10 kroków bez ruchu (dry run na danych ze stanu robota).
- Uruchom politykę na 1 epizod z velocity_scaling = 0.05. Obserwuj każdy staw i sił momentu.
- Jeśli ruch jest poprawny, zwiększ velocity_scaling do 0.10 i powtórz 3 epizody.
- Dopiero po 5 udanych epizodach pod ścisłym nadzorem rozważ velocity_scaling = 0.15.
- Loguj każdy epizod: komendy, stan stawów, nagrodę online i wynik.

## 8.2. Filtrowanie akcji i ograniczenia bezpieczeństwa

```
import numpy as np

class SafetyFilter:
    def   init  (
        self,
        joint limits: np.ndarray,    # shape [DOF, 2]: [[min, max], ...]
        max delta q: float = 0.05,  # maks. zmiana pozycji na krok [rad]
        alpha: float = 0.8,         # współczynnik filtru EMA
    ) -> None:
        self.limits    = joint limits
        self.max dq    = max delta q
        self.alpha     = alpha
        self. prev q   = None

    def   call  (self, raw action: np.ndarray,
                 current q: np.ndarray) -> np.ndarray:
        # 1. Filtr EMA (wygładzenie)
        if self. prev q is None:
            self. prev q = current q.copy()
        action = self.alpha * raw action + (1 - self.alpha) * self. prev q

        # 2. Ograniczenie kroku
        delta = np.clip(action - current q, -self.max dq, self.max dq)
        action = current q + delta

```

```
        # 3. Ograniczenie zakresów stawów
        action = np.clip(action, self.limits[:, 0], self.limits[:, 1])

        self. prev q = action.copy()
        return action
```

> **Ostrzeżenie**
>
> Nigdy nie uruchamiaj polityki bez filtru bezpieczeństwa na realnym robocie. Skok akcji o kilka radianów w jednym kroku może mechanicznie uszkodzić staw lub zrzucić robota.

### 8.3. Analiza luki sim-to-real

Po każdej sesji na realnym robocie porównaj krzywe pozycji stawów z symulatora i z rzeczywistości dla identycznych komend wejściowych.

| Objaw | Prawdopodobna przyczyna i działanie |
|---|---|
| Robot porusza się poprawnie, ale wolniej niż w symulacji | Tarcie lub ineryjność niedoszacowane → poszerz DR tarcia |
| Ruch jest poprawny, ale drgający | Brak filtru EMA lub zbyt duży max_delta_q → zmniejsz krok |
| Robot ignoruje komendę (staw nie reaguje) | Błędne mapowanie indeksów stawów sim ↔ real → sprawdź joint_ids |
| Staw trafia w ogranicznik mechaniczny | Zakresy stawów w konfiguracji USD różnią się od hardware → synchronizuj URDF |
| Robot pada przy pierwszym kroku | Polityka nie widziała perturbacji postawy → dodaj DR orientacji bazowej |

## 9. Logowanie, testy i debugowanie

### 9.1. Minimalny standard logowania - ćwiczenie 3

| Co logować | Gdzie przechowywać |
|---|---|
| Konfiguracja YAML treningu (pełna) | logs/{experiment}/config.yaml |
| Seed, commit repozytorium, wersje bibliotek | logs/{experiment}/env_info.txt |
| TensorBoard: reward, loss, KL, success rate | logs/{experiment}/tb/ |
| Checkpointy: co 100 it. + checkpoint_final | logs/{experiment}/checkpoints/ |
| Wyniki ewaluacji DR (tabela) | logs/{experiment}/eval_dr.csv |
| Logi sesji sim-to-real: komendy + stan stawów | logs/{experiment}/realrobot/{data}.bag |
| Demonstracje HDF5 (jeśli IL) | datasets/{task}_{version}.hdf5 |

### 9.2. Najczęstsze błędy i diagnostyka

**Błąd / Diagnostyka**

**Trening nie konwerguje po 500 it.:** Sprawdź skalę nagrody (tensorboard: reward mean). Jeśli < 0.01 lub > 100, przeskaluj. Sprawdź rollout wizualnie — czy agent w ogóle próbuje wykonać zadanie?

**Błąd / Diagnostyka**

**NaN w loss lub gradientach:** Zmniejsz learning_rate o rząd wielkości. Sprawdź czy obserwacje są znormalizowane. Dodaj torch.nn.utils.clip_grad_norm_ z max_norm=1.0.

**Błąd / Diagnostyka**

**BC: success rate nie rośnie powyżej 30%:** Obejrzyj demonstracje — czy są spójne? Sprawdź czy obserwacje w datasecie pasują do obserwacji środowiska ewaluacyjnego (te same pola, ta sama normalizacja).

**Błąd / Diagnostyka**

**Sim-to-real: robot drgania przy każdym kroku:** Zmniejsz max_delta_q do 0.02 rad. Zwiększ alpha filtru EMA do 0.9. Sprawdź czy decimation w symulacji pasuje do częstotliwości sterowania na robocie.

### 9.3. Checklista zamknięcia sesji RL

- Zapisano checkpoint_final i plik konfiguracyjny YAML do repozytorium.
- TensorBoard zawiera metryki przez cały czas treningu (bez przerw w logach).
- Wyniki ewaluacji DR zapisane do CSV z datą, seedem i nazwą eksperymentu.
- Jeśli były sesje na realnym robocie: bag nagrany i opisany w README eksperymentu.
- Conda environment wyeksportowany: conda env export > environment.yml.

## 10. Ćwiczenia laboratoryjne i ścieżka rozwoju

- Uruchom środowisko G1-Reach-v0 i zmodyfikuj funkcję nagrody: dodaj człon za gładkość prędkości stawów. Porównaj krzywe uczenia dla wersji z i bez tego członu.
- Zaimplementuj CurriculumManager i uruchom trening. Zaobserwuj, w którym momencie curriculum przesuwa zakres celów i jak wpływa to na success rate.
- Trenuj PPO na 3 różnych seedach z tymi samymi hiperparametrami. Narysuj krzywe reward mean z przedziałem ufności. Opisz wariancję między seedami.
- Zbierz 20 demonstracji teleoperacją. Obejrzyj je w wizualizatorze, odrzuć złe próby. Wytrenuj BC i porównaj success rate z wynikiem PPO.
    - Przeprowadź ablację DR: wytrenuj politykę bez DR, z DR tylko masy, z pełnym DR. Oceń success rate przy ±20% perturbacji masy dla każdego wariantu.
    - Przenieś najlepszą politykę na realnego G1 według protokołu z sekcji 8. Zarejestruj co najmniej 5 prób i oblicz success rate. Opisz zaobserwowaną lukę sim-to-real.

### Kierunki dalszego rozwoju projektu

- **Hierarchiczne RL:** wysoki poziom wybiera podzadanie (FSM), niski poziom wykonuje przez nauczoną politykę.
- **Adaptacja online:** RMA (Rapid Motor Adaptation) — enkoder stanu ukrytego dla szybkiej adaptacji do nowych parametrów fizyki.
- **Percepcja w pętli:** polityka przyjmuje embedding z kamery jako część obserwacji; trening end-to-end lub z zamrożonym enkoderem.
- **Dłoń dexterous:** rozszerzenie środowiska o sterowanie dłonią Dex3-1; redefiniowanie przestrzeni akcji.
- **Uczenie wielo-zadaniowe:** jedna polityka dla wielu obiektów i pozycji docelowych dzięki kodowaniu celu w obserwacji.

## Załącznik A. Minimalna struktura repozytorium po Ćwiczeniu 3

```
g1 project/
├── README.md                    # wersje, GPU, commit, seed
├── environment.yml              # conda env export
├── ros2 ws/                     # z Ćwiczenia 1 i 2
├── sim/
|   └── IsaacLab/
|       └── source/isaaclab tasks/isaaclab tasks/g1 tasks/
|           ├──   init  .py
|           ├── g1 reach env.py     # własne środowisko
|           └── reward.py           # funkcja nagrody
├── configs/
|   ├── ppo g1 reach.yaml
|   └── bc g1 reach.json
├── datasets/
|   └── g1 reach v1.hdf5
├── logs/
|   ├── g1 reach ppo v1/
|   |   ├── config.yaml
|   |   ├── env info.txt
|   |   ├── checkpoints/
|   |   ├── tb/
|   |   └── eval dr.csv
|   └── g1 reach bc v1/
├── experiments/
|   └── realrobot/
|       ├── session 2024-01-15/
|       |   ├── session.bag
|       |   └── notes.md
|       └── ...
└── scripts/
    ├── train ppo.sh
    ├── train bc.sh
    ├── eval dr.py
    └── deploy policy.py         # wrapper SafetyFilter + SDK
```

## Załącznik B. Dobre praktyki RL i sim-to-real

- Każdy eksperyment ma unikalną nazwę, seed i commit. Bez tego wyniki są nieodtwarzalne.
- Nie usuwaj starych checkpointów — najlepszy model może być z epoki 300, a nie 1500.
- Przed transferem sim-to-real zawsze wizualizuj rollout polityki w symulacji z otwartym RViz. Anomalie widoczne na ekranie to anomalie, które zobaczycie na robocie.
- Limit czasu na epizod testowy na realnym robocie: nigdy nie pozwól polityce działać bez nadzoru dłużej niż 15 sekund.
- Jeśli polityka działa w symulacji, ale nie na robocie — nie modyfikuj nagrody. Najpierw zamknij lukę modelową przez DR, potem optymalizuj nagrodę.
- Demonstracje do IL nagrywaj w spokojnych warunkach, bez pośpiechu. Jedna zła demonstracja może zdominować trening BC.

## Załącznik C. Materiały źródłowe i literatura


- **Isaac Lab documentation:** isaac-sim.github.io/IsaacLab
- **rsl-rl:** github.com/leggedrobotics/rsl_rl
- **robomimic:** robomimic.github.io
- **PPO (Schulman et al., 2017):** arxiv.org/abs/1707.06347
- **Domain Randomization (Tobin et al., 2017):** arxiv.org/abs/1703.06907
- **RMA (Kumar et al., 2021):** arxiv.org/abs/2107.04034 — adaptacja online polityki robotycznych nóg
- **Unitree SDK2:** github.com/unitreerobotics/unitree_sdk2_python

# Ćwiczenie 4: Dłoń dexterous, uczenie wielo-zadaniowe i wdrożenie produkcyjne

**Uwaga**

**Skrót dla prowadzącego:** Dopuść zespół do Ćwiczenia 4 wyłącznie po potwierdzeniu M3.8, udanym transferze polityki z symulacji na realnego G1. To ćwiczenie łączy wszystkie poprzednie warstwy: bezpieczeństwo, ROS 2, percepcję, FSM, RL/IL i sim-to-real. Każdy etap wymaga odbioru przed przejściem do następnego.

## Mapa dokumentu

| Sekcja | Po co istnieje |
|---|---|
| 1. Cel i zakres | Definiuje wynik końcowy i warunki odbioru projektu. |
| 2. Dłoń dexterous — Dex3-1 i Inspire | Mechanika, API i bezpieczne uruchamianie końcówki. |
| 3. Planowanie chwytu | Grasp pose estimation, GraspNet, metryki jakości chwytu. |
| 4. Uczenie wielo-zadaniowe | Goal-conditioned RL, task embeddings, jedna polityka dla wielu celów. |
| 5. Percepcja w pętli uczenia | End-to-end z enkoderem wizyjnym, latent goal representations. |
| 6. Adaptacja online (RMA) | Szybka adaptacja do nowych parametrów fizyki bez restartu treningu. |
| 7. Architektura systemu produkcyjnego | Integracja ROS 2 + SDK + polityka + bezpieczniki w jeden potok. |
| 8. Testy systemowe i odbiór projektu | Protokół ewaluacji end-to-end, metryki i dokumentacja końcowa. |
| 9. Ćwiczenia i kierunki badawcze | Zadania finalne i propozycje tematów prac dyplomowych. |
| Załączniki | Checklisty, struktura repozytorium, literatura. |

## 1. Cel i zakres ćwiczenia

Ćwiczenie 4 jest **finałem cyklu**. Jego celem jest zbudowanie działającego systemu manipulacyjnego end-to-end: robot **widzi obiekt, planuje chwyt, podchodzi, chwyta i przenosi**, autonomicznie i powtarzalnie. Wszystkie warstwy z poprzednich ćwiczeń stają się modułami jednego systemu produkcyjnego.

### 1.1. Definicja zadania końcowego

**Definicja**

**Zadanie Pick-and-Place:** Robot G1 EDU wykrywa wskazany obiekt (np. kostka 5×5×5 cm) leżący na stole w odległości 30–60 cm od bazy, planuje chwyt z użyciem dłoni dexterous, przenosi obiekt

do oznaczonej strefy docelowej i odkłada go w pozycji pionowej. Całość wykonywana autonomicznie przez wytrenowaną politykę z percepcją online, bez interwencji operatora.

## 1.2. Kamienie milowe ćwiczenia 4

| Etap | Kryterium wyjścia |
|---|---|
| M4.1: Dłoń dexterous | Dex3-1/Inspire odpowiada na komendy SDK; 5 ruchów palców wykonanych bezawaryjnie. |
| M4.2: Grasp pose estimation | Węzeł percepcji publikuje PoseStamped chwytu z orientacją; walidacja w RViz. |
| M4.3: Polityka wielo-zadaniowa | Jedna polityka RL osiąga success rate > 65% dla ≥ 3 różnych obiektów w symulacji. |
| M4.4: Percepcja w pętli | Polityka z enkoderem wizyjnym działa stabilnie w symulacji przy losowych pozycjach obiektów. |
| M4.5: Adaptacja online | RMA enkoder redukuje degradację przy ±30% perturbacji masy do < 10 pp. |
| M4.6: Integracja systemowa | Pełny potok ROS 2 uruchamia się jednym launch file; wszystkie węzły startują bez błędów. |
| M4.7: Odbiór end-to-end | System wykonuje pick-and-place na realnym G1 z success rate ≥ 3/5 prób pod nadzorem. |

**Ostrzeżenie**

Nie przechodź do M4.3 bez sprawdzonego M4.2. Polityka wielo-zadaniowa nauczona na błędnych pozach chwytu będzie skuteczna w symulacji i katastrofalna na realnym robocie.

# 2. Dłoń dexterous - Dex3-1 i Inspire

Unitree G1 EDU może być wyposażony w dłoń **Dex3-1** (3 palce, 6 DOF) lub dłoń **Inspire** (5 palców, 12 DOF). Obie komunikują się przez SDK jako odrębne aktuatory. Sterowanie dłonią wymaga osobnej warstwy bezpieczeństwa i oddzielnego węzła ROS 2.

## 2.1. Parametry dłoni

| Parametr | Dex3-1 | Inspire |
|---|---|---|
| Liczba palców | 3 | 5 |
| DOF | 6 (2 na palec) | 12 (2–3 na palec) |
| Siła chwytu | do ~30 N | do ~20 N |
| Typ chwytu | Precision / power | Precision / pinch / cylindrical |
| Zasilanie | DC 24V przez robot | DC 24V przez robot |
| Interfejs | unitree_sdk2 HandCmd | unitree_sdk2 HandCmd |
| Uwaga bezp. | Sprawdź trasę kabla | Sprawdź trasę kabla przed każdym testem |

## 2.2. Checklista uruchomienia dłoni

- Trasa kabla dłoni sprawdzona — brak dogiętych miejsc przy stawach ramienia.
- Dłoń zamontowana stabilnie; śruby montażowe dokręcone zgodnie z instrukcją mechaniczną.
- Konfiguracja robota nie dopuszcza przysiadów ani pozycji leżącej z Dex3-1 (ograniczenie producenta).
- Program diagnostyczny weryfikuje odpowiedź dłoni przed przejściem do sterowania polityką.
- Zakres ruchu palców ograniczony w konfiguracji do bezpiecznych wartości startowych.

## 2.3. Pierwsze komendy dłoni przez SDK

```
from unitree sdk2py.core.channel import ChannelSubscriber, ChannelPublisher
from unitree sdk2py.idl.unitree hg.msg.dds  import HandCmd
import time

# Inicjalizacja publishera komend dłoni
pub = ChannelPublisher('rt/inspire/cmd', HandCmd )
pub.Init()

def set finger position(finger id: int, position: float) -> None:
    """position: 0.0 = otwarty, 1.0 = zamknięty"""
    cmd = HandCmd ()
    cmd.motor cmd[finger id].q   = position
    cmd.motor cmd[finger id].kp  = 2.0   # sztywność
    cmd.motor cmd[finger id].kd  = 0.05  # tłumienie
    pub.Write(cmd)

# Test: powolne zamykanie wszystkich palców
for pos in [0.0, 0.25, 0.5, 0.75, 1.0]:
    for finger in range(5):
        set finger position(finger, pos)
    time.sleep(0.8)

# Powrót do pozycji otwartej
for finger in range(5):
    set finger position(finger, 0.0)
```

**Wskazówka**

Zawsze zacznij od pozycji otwartej (0.0) i zwiększaj stopniowo. Nigdy nie wysyłaj komendy pełnego zamknięcia (1.0) bez sprawdzenia, czy w dłoni nie ma przeszkód.

## 2.4. Węzeł ROS 2 dla dłoni

```
import rclpy
from rclpy.node import Node
from std msgs.msg import Float32MultiArray
from unitree sdk2py.idl.unitree hg.msg.dds  import HandCmd

class HandControllerNode(Node):
    """Subskrybuje /hand/target positions [5 floats 0-1]"""
    def   init  (self) -> None:
```

```
        super().  init  ('hand controller node')
        self.sub = self.create subscription(
            Float32MultiArray, '/hand/target positions',
            self.cmd callback, 10
        )
        self. init sdk publisher()
        self.get logger().info('HandControllerNode gotowy')

    def cmd callback(self, msg: Float32MultiArray) -> None:
        if len(msg.data) != 5:
            self.get logger().warn('Oczekiwano 5 wartości pozycji palców')
            return
        cmd = HandCmd ()
        for i, pos in enumerate(msg.data):
            pos clamped = max(0.0, min(1.0, pos))  # bezpiecznik zakresu
            cmd.motor cmd[i].q  = pos clamped
            cmd.motor cmd[i].kp = 2.0
            cmd.motor cmd[i].kd = 0.05
        self. pub.Write(cmd)

    def  init sdk publisher(self) -> None:
        from unitree sdk2py.core.channel import ChannelPublisher
        self. pub = ChannelPublisher('rt/inspire/cmd', HandCmd )
        self. pub.Init()
```

## 3. Planowanie chwytu (Grasp Pose Estimation)

Zanim robot wykona chwyt, musi wiedzieć **gdzie i jak uchwycić obiekt.** Grasp Pose Estimation to zadanie wyznaczenia pozycji i orientacji dłoni, przy której chwyt będzie stabilny. Na zajęciach korzystamy z podejścia opartego o chmurę punktów i analityczne metryki jakości.

### 3.1. Potok wyznaczania pozy chwytu

| Etap | Narzędzie / metoda |
|---|---|
| 1. Segmentacja obiektu | Filtr Pass-Through + RANSAC do usunięcia płaszczyzny stołu (PCL / Open3D) |
| 2. Wyznaczenie bounding box | Oriented Bounding Box (OBB) z Open3D |
| 3. Generacja kandydatów chwytu | Analityczne: antipodal grasps; lub sieć: GraspNet-1Billion |
| 4. Ocena kandydatów | Friction cone test + jakość siłowa (force closure) |
| 5. Transformacja do base_link | TF2: camera_link → base_link |
| 6. Publikacja PoseStamped | Temat: /hand/grasp_pose (MoveIt 2 konsumuje jako cel) |

### 3.2. Segmentacja i wyznaczanie bounding box

```
import open3d as o3d
import numpy as np

def extract object bbox(
    pcd: o3d.geometry.PointCloud,
    table height z: float = 0.72,  # metry od base link
```

```
    z_margin: float = 0.02,
) -> o3d.geometry.OrientedBoundingBox | None:
    """Usuwa płaszczyznę stołu i zwraca OBB największego klastra."""
    pts = np.asarray(pcd.points)

    # Filtr wysokości — punkty powyżej stołu
    mask = pts[:, 2] > table_height_z + z_margin
    if mask.sum() < 50:
        return None
    obj_pcd = pcd.select_by_index(np.where(mask)[0])

    # Usunięcie szumu i klastrowanie DBSCAN
    obj_pcd = obj_pcd.remove_statistical_outlier(nb_neighbors=20, std_ratio=2.0)[0]
    labels = np.array(obj_pcd.cluster_dbscan(eps=0.02, min_points=10))
    if labels.max() < 0:
        return None

    # Największy klaster
    largest = np.argmax(np.bincount(labels[labels >= 0]))
    cluster = obj_pcd.select_by_index(np.where(labels == largest)[0])
    return cluster.get_oriented_bounding_box()
```

## 3.3. Analityczne wyznaczanie pozy chwytu antipodal

```
from dataclasses import dataclass
import numpy as np

@dataclass
class GraspCandidate:
    position:    np.ndarray    # środek chwytu [3]
    approach:    np.ndarray    # kierunek podejścia dłoni [3]
    quality:     float         # wynik [0, 1]

def generate_antipodal_grasps(
    bbox: 'OrientedBoundingBox',
    n_candidates: int = 16,
) -> list[GraspCandidate]:
    """Generuje kandydatów chwytu jako pary punktów antipodal na OBB."""
    center  = np.asarray(bbox.center)
    extents = np.asarray(bbox.extent)       # [dx, dy, dz]
    R       = np.asarray(bbox.R)             # macierz obrotu

    candidates = []
    for axis in range(3):                    # próbuj wzdłuż każdej osi OBB
        approach = R[:, axis]                # wektor osi
        width    = extents[axis]
        if width > 0.12:                     # za szeroki dla dłoni
            continue
        quality = 1.0 - width / 0.12         # bliżej 0 = lepszy chwyt
        candidates.append(GraspCandidate(
            position=center,
            approach=approach,
            quality=quality,
        ))
    return sorted(candidates, key=lambda g: -g.quality)
```

**Wskazówka**

Wizualizuj kandydatów chwytu w RViz jako MarkerArray ze strzałkami w kolorze zakodowanym jakością (zielony = dobry, czerwony = słaby) zanim uruchomisz jakikolwiek ruch ramienia.

### 3.4. Walidacja pozy chwytu: checklista

- Poza chwytu leży w osiągalnym workspace ramienia (weryfikacja IK w MoveIt 2).
- Kierunek podejścia nie koliduje z blatem stołu (sprawdź w RViz z collision mesh).
- Szerokość chwytu mieści się w zakresie dłoni: < 10 cm dla Dex3-1, < 8 cm dla Inspire.
- TF2 transform grasp_pose do base_link poprawny — brak skoków po 2–3 klatkach.
- Jakość chwytu (quality score) > 0.5 zanim wyślesz komendę do arm_controller.

## 4. Uczenie wielo-zadaniowe (Goal-Conditioned RL)

Zamiast trenować osobną politykę dla każdego obiektu i pozycji docelowej, trenujemy **jedną politykę uwarunkowaną celem** (goal-conditioned policy). Cel jest zakodowany jako część obserwacji, co pozwala polityce generalizować na nowe konfiguracje bez restartu treningu.

### 4.1. Kodowanie celu w obserwacji

**Definicja**

**Goal-conditioned policy:** π(a | s, g), gdzie g to reprezentacja celu — np. pozycja docelowa efektora, identyfikator obiektu lub embedding wizyjny. Polityka uczy się warunkować swoje zachowanie na g, co pozwala jednej sieci obsługiwać wiele zadań.

```
import torch
import torch.nn as nn

class GoalConditionedActor(nn.Module):
    def   init  (
        self,
        obs dim:  int = 67,    # q(29) + dq(29) + rpy(3) + gyro(3) + hand q(6) - 3
(goal placeholder)
        goal dim: int = 6,     # pos(3) + quat(3) lub pos(3) + one hot(3 objects)
        act dim:  int = 18,    # 12 stawy ramion + 6 palców
        hidden:   list = None,
    ) -> None:
        super().  init  ()
        hidden = hidden or [512, 256, 128]
        layers = []
        in dim = obs dim + goal dim
        for h in hidden:
            layers += [nn.Linear(in dim, h), nn.ELU()]
            in dim = h
        layers.append(nn.Linear(in dim, act dim))
        self.net = nn.Sequential(*layers)

    def forward(
        self,
        obs:  torch.Tensor,   # [B, obs dim]
        goal: torch.Tensor,   # [B, goal dim]
```

```
    ) -> torch.Tensor:        # [B, act_dim]
        return self.net(torch.cat([obs, goal], dim=-1))
```

## 4.2. Hindsight Experience Replay (HER)

Dla zadań z rzadką nagrodą (binary success) HER jest kluczową techniką: po każdym epizodzie cele są **retrospektywnie przepisywane** tak, aby stan końcowy epizodu był traktowany jako osiągnięty cel. Drastycznie zwiększa gęstość pozytywnych przykładów.

```
from dataclasses import dataclass, field
from typing import List
import numpy as np

@dataclass
class Transition:
    obs:    np.ndarray
    goal:   np.ndarray
    action: np.ndarray
    reward: float
    next_obs: np.ndarray
    done:   bool

def apply_her(
    episode: List[Transition],
    k: int = 4,               # ile celów HER na transition
    strategy: str = 'future' # 'future' | 'episode' | 'final'
) -> List[Transition]:
    """Zwraca oryginalne + k * HER transitions."""
    augmented = list(episode)
    T = len(episode)
    for t, tr in enumerate(episode):
        indices = np.random.randint(t, T, size=k)  # 'future' strategy
        for idx in indices:
            her_goal   = episode[idx].next_obs[:3]  # pozycja efektora jako cel
            her_reward = 1.0 if np.linalg.norm(tr.next_obs[:3] - her_goal) < 0.03 else 0.0
            augmented.append(Transition(
                obs=tr.obs, goal=her_goal, action=tr.action,
                reward=her_reward, next_obs=tr.next_obs, done=tr.done,
            ))
    return augmented
```

## 4.3. Konfiguracja środowiska wielo-zadaniowego

| Parametr | Wartość / zakres |
|---|---|
| Liczba obiektów (klasy) | 3–5 (np. kostka, cylinder, prostopadłościan) |
| Losowa pozycja obiektu | stół 40×60 cm; robot 30–60 cm od obiektu |
| Losowa pozycja docelowa | oznaczona strefa 20×20 cm; orientacja: ±30° |
| Kodowanie celu | pos(3) + orientacja cel. (3) + id obiektu one-hot(5) |
| Nagroda postępu | exp(-4·dist_ee_to_obj) + exp(-4·dist_obj_to_goal) |
| Bonus sukcesu | +15 za obiekt w strefie z prawidłową orientacją |

| Warunek końca | dist_obj_to_goal < 3 cm LUB timeout 12 s |
|---|---|

**Kluczowa zasada**

Trenuj z curriculum: zacznij od jednego obiektu i małej strefy; rozszerzaj obiekty i zakres jeden po drugim w miarę konwergencji. Multi-task od samego początku bez curriculum zwykle nie konwerguje.

## 5. Percepcja w pętli uczenia

Zamiast oddzielać percepcję od polityki, możemy wytrenować enkoder wizyjny **end-to-end razem z polityką**. Polityka wtedy bezpośrednio obserwuje obraz — lub jego latentną reprezentację — co eliminuje błędy wynikające z ręcznie zaprojektowanego pipeline percepcyjnego.

### 5.1. Architektura z enkoderem wizyjnym

```
import torch
import torch.nn as nn
from torchvision.models import resnet18

class VisualEncoder(nn.Module):
    def   init  (self, latent dim: int = 64) -> None:
        super().  init  ()
        backbone = resnet18(pretrained=False)
        # Usuń ostatnią warstwę klasyfikacyjną
        self.features = nn.Sequential(*list(backbone.children())[:-1])
        self.proj     = nn.Linear(512, latent dim)

    def forward(self, img: torch.Tensor) -> torch.Tensor:
        # img: [B, 3, H, W], wartości [0, 1]
        x = self.features(img).flatten(1)    # [B, 512]
        return self.proj(x)                   # [B, latent dim]

class VisualPolicy(nn.Module):
    def   init  (self, prop dim=61, latent dim=64, act dim=18) -> None:
        super().  init  ()
        self.encoder = VisualEncoder(latent dim)
        self.actor    = nn.Sequential(
            nn.Linear(prop dim + latent dim, 512), nn.ELU(),
            nn.Linear(512, 256),                   nn.ELU(),
            nn.Linear(256, act dim),
        )

    def forward(self, props: torch.Tensor, img: torch.Tensor) -> torch.Tensor:
        z = self.encoder(img)                    # [B, latent dim]
        return self.actor(torch.cat([props, z], dim=-1))
```

### 5.2. Strategie treningu enkodera

| Strategia | Kiedy stosować | Uwaga |
|---|---|---|
| End-to-end (pełny gradient) | Mały dataset, prosta scena | Wolna konwergencja; wymaga dużej VRAM |

| Zamrożony pretrained encoder | ResNet/ViT pretrained na ImageNet | Szybki start; słaba generalizacja na nowe obiekty |
|---|---|---|
| Fine-tuning ostatnich warstw | Środkowy kompromis | Rozmrożaj stopniowo po konwergencji aktora |
| Contrastive pretraining (R3M) | Duży zbiór demonstracji | Najlepsza generalizacja; wymaga danych teleop |

**Ostrzeżenie**

Trening end-to-end z kamerą wymaga co najmniej 2× więcej VRAM niż trening z obserwacjami własnościowymi. Sprawdź pamięć GPU przed uruchomieniem: nvidia-smi dmon -s m.

## 6. Adaptacja online - Rapid Motor Adaptation (RMA)

RMA (Kumar et al., 2021) to dwuetapowy framework uczenia, który pozwala polityce **szybko adaptować się do nowych parametrów fizyki** (masa, tarcie, uszkodzenia aktuatorów) bez restartu treningu. Jest szczególnie przydatny przy transferze sim-to-real na G1.

### 6.1. Architektura RMA

| Komponent | Rola |
|---|---|
| Polityka bazowa π(a \| s, z) | Aktor warunkowny na latentnym kontekście z; trenowany w fazie 1 z ground-truth z. |
| Enkoder środowiska E(e → z) | W fazie 1 mapuje jawne parametry środowiska (masa, tarcie) na z. Niedostępny na robocie. |
| Enkoder adaptacyjny Â(hist → z) | W fazie 2 uczy się przewidywać z tylko z historii obserwacji i akcji. Dostępny na robocie. |

```
import torch
import torch.nn as nn

# FAZA 1: Polityka bazowa z ground-truth enkoderem środowiska
class EnvironmentEncoder(nn.Module):
    """Przyjmuje jawne parametry środowiska (niedostępne na robocie)."""
    def   init  (self, env dim: int = 8, latent dim: int = 16) -> None:
        super().  init  ()
        self.net = nn.Sequential(
            nn.Linear(env dim,    64), nn.ELU(),
            nn.Linear(64, latent dim),
        )
    def forward(self, env params: torch.Tensor) -> torch.Tensor:
        return self.net(env params)    # [B, latent dim]

# FAZA 2: Enkoder adaptacyjny — uczy się z historii
class AdaptationEncoder(nn.Module):
    """Dostępny na realnym robocie. Estymuje z z historii (obs, act)."""
    def   init  (self,
                 obs dim: int = 67,
                 act dim: int = 18,
                 history len: int = 50,
```

```
                 latent dim: int = 16) -> None:
        super().  init  ()
        in dim = (obs dim + act dim) * history len
        self.net = nn.Sequential(
            nn.Linear(in dim, 256), nn.ELU(),
            nn.Linear(256,     128), nn.ELU(),
            nn.Linear(128, latent dim),
        )

    def forward(self, history: torch.Tensor) -> torch.Tensor:
        # history: [B, history len, obs dim + act dim]
        return self.net(history.flatten(1))    # [B, latent dim]

# Trening fazy 2: dopasuj Â do E przez MSE na tych samych trajektoriach
# loss = F.mse loss(adaptation encoder(history), env encoder(env params).detach())
```

## 6.2. Ocena skuteczności RMA

| Test | Oczekiwany wynik po pełnym RMA |
|---|---|
| Nominalne parametry fizyki | > 80% success rate (baseline) |
| ±20% masy segmentów | Degradacja < 8 pp vs baseline |
| ±30% tarcia stawów | Degradacja < 12 pp vs baseline |
| Symulowane uszkodzenie 1 stawu | Degradacja < 20 pp; robot kompensuje postawą |
| Nowa powierzchnia podłoża | Degradacja < 10 pp przy tarciu 0.3–0.9 |

# 7. Architektura systemu produkcyjnego

System produkcyjny to integracja wszystkich modułów w jeden spójny potok, który można uruchomić jednym poleceniem i który **automatycznie obsługuje błędy, loguje każdą sesję i zatrzymuje się bezpiecznie** przy przekroczeniu dowolnego progu bezpieczeństwa.

## 7.1. Diagram warstw systemu

| Warstwa | Moduły i odpowiedzialność |
|---|---|
| Warstwa sprzętowa | Pilot RC, SDK unitree_sdk2, sterownik dłoni — bezpośrednia komunikacja z robotem |
| Warstwa sterownikowa | g1_driver_node: most SDK ↔ ROS 2; joint_state_bridge; hand_controller_node |
| Warstwa percepcji | perception_node: kamera + lidar → grasp_pose; tf2_bridge |
| Warstwa polityki | policy_node: wczytuje checkpoint, AdaptationEncoder, SafetyFilter, publikuje akcje |
| Warstwa logiki zadania | task_manager_node: FSM pick-and-place, timeouty, warunki błędu |
| Warstwa monitorowania | safety_monitor_node: limity momentów, zakresy stawów, heartbeat; emergency stop |
| Warstwa logowania | ROS bag, CSV akcje+stany, TensorBoard online (jeśli sieć dostępna) |

## 7.2. Węzeł safety_monitor

```
import rclpy
from rclpy.node import Node
from std msgs.msg import Bool
from sensor msgs.msg import JointState
import numpy as np

# Progi bezpieczeństwa — dostosuj do konfiguracji sprzętowej
TORQUE LIMIT   = np.array([80.0] * 12 + [30.0] * 6 + [5.0] * 6)   # Nm
POSITION LIMIT = np.array([[-2.5, 2.5]] * 29)   # rad (uproszczenie)
HEARTBEAT TIMEOUT = 0.5   # s — brak akcji → emergency stop

class SafetyMonitorNode(Node):
    def   init  (self) -> None:
        super().  init  ('safety monitor node')
        self.sub joints  = self.create subscription(
            JointState, '/unitree/joint states', self. check state, 10)
        self.pub estop   = self.create publisher(Bool, '/emergency stop', 1)
        self. last action t = self.get clock().now()
        self.create timer(0.1, self. check heartbeat)

    def  check state(self, msg: JointState) -> None:
        efforts = np.array(msg.effort)
        if np.any(np.abs(efforts) > TORQUE LIMIT[:len(efforts)]):
            self.get logger().error('PRZEKROCZENIE MOMENTU — EMERGENCY STOP')
            self. publish estop()

    def  check heartbeat(self) -> None:
        dt = (self.get clock().now() - self. last action t).nanoseconds * 1e-9
        if dt > HEARTBEAT TIMEOUT:
            self.get logger().warn(f'Brak akcji przez {dt:.2f}s — stop')
            self. publish estop()

    def  publish estop(self) -> None:
        self.pub estop.publish(Bool(data=True))
```

## 7.3. Główny launch file systemu produkcyjnego

```
# g1 bringup/launch/production.launch.py
from launch import LaunchDescription
from launch.actions import TimerAction
from launch ros.actions import Node

def generate launch description():
    driver     = Node(package='g1 driver',     executable='g1 driver node')
    hand ctrl  = Node(package='g1 control',    executable='hand controller node')
    perception = Node(package='g1 perception', executable='perception node')
    policy     = Node(package='g1 control',    executable='policy node',
                      parameters=[{'checkpoint': 'logs/final/checkpoint final.pt',
                                   'velocity scale': 0.10}])
    monitor    = Node(package='g1 control',    executable='safety monitor node')
    task mgr   = Node(package='g1 control',    executable='task manager node')

    # Kolejność startowania: driver → percepcja → polityka → monitor → FSM
    return LaunchDescription([
        driver,
```

```
        TimerAction(period=2.0, actions=[hand ctrl, perception]),
        TimerAction(period=4.0, actions=[policy, monitor]),
        TimerAction(period=6.0, actions=[task mgr]),
    ])
```

**Wskazówka**

Opóźnienia TimerAction gwarantują, że driver zdąży nawiązać połączenie z robotem zanim perception i policy zaczną subskrybować dane. Bez nich wyścig inicjalizacji powoduje trudne do zdebugowania błędy startowe.

## 8. Testy systemowe i odbiór projektu

Odbiór projektu wymaga udokumentowanego testu end-to-end na realnym robocie. Nie wystarczy pokaz — wymagane są **zarejestrowane metryki, bag z sesji i protokół odbioru** podpisany przez prowadzącego i cały zespół.

### 8.1. Protokół testu end-to-end

| Krok | Opis i warunek zaliczenia |
|---|---|
| 1. Rozgrzewka diagnostyczna | Program z Ćw. 1 potwierdza stabilne odczyty stawów i IMU. Brak NaN. |
| 2. Test dłoni | 5 cykli otwórz-zamknij dłoni bez komend z polityki. Brak drgań. |
| 3. Test percepcji | Węzeł publikuje grasp_pose dla obiektu testowego. Wizualizacja w RViz. |
| 4. Test ramienia (sucha trajekt.) | MoveIt 2 planuje trajektorię do grasp_pose bez ruchu dłoni. Zaakceptowana. |
| 5. Test pick w miejscu | Robot wykonuje chwyt bez przenoszenia. 3/3 udane chwyty. |
| 6. Pełny pick-and-place | 5 prób. Zaliczone ≥ 3/5. Każda próba logowana do bag. |

### 8.2. Metryki odbioru

| Metryka | Definicja | Próg zaliczenia |
|---|---|---|
| Success Rate | % prób z obiektem w strefie docelowej | ≥ 60% (3/5 prób) |
| Grasp Success Rate | % prób z udanym chwytem (obiekt podniesiony) | ≥ 80% |
| Czas wykonania | Od startu FSM do odkładania obiektu | < 25 s / próba |
| Max moment stawu | Maksymalny zmierzony moment podczas sesji | < 90% progu bezp. |
| Emergency Stop | Liczba automatycznych zatrzymań przez monitor | 0 w zaliczanych próbach |
| Bag Coverage | % czasu z pełnym logowaniem (wszystkie tematy) | ≥ 95% |

### 8.3. Dokumentacja końcowa projektu

Każdy zespół dostarcza dokumentację zawierającą następujące elementy:

1. **README.md:** wersje, seed, commit, GPU, instrukcja uruchomienia jednym poleceniem.
2. **Raport techniczny (2–4 strony):** opis architektury polityki, wyniki treningu (krzywe TensorBoard), wyniki ewaluacji DR, analiza luki sim-to-real, tabela metryk odbioru.
3. **Repozytorium:** kompletna struktura (patrz Załącznik A), checkpointy, konfiguracje YAML/JSON, launch files.
4. **Nagrania sesji:** bag z co najmniej 3 próbami pick-and-place na realnym robocie, nagranie wideo z kamery zewnętrznej (opcjonalne, ale zalecane).
5. **Prezentacja (10 min + 5 min Q&A):** demonstracja live lub nagranie wideo, omówienie napotkanych problemów i podjętych decyzji projektowych.

### 8.4. Checklista końcowego odbioru

- System uruchamia się jednym: ros2 launch g1_bringup production.launch.py
- safety_monitor_node aktywny i potwierdzony przez prowadzącego przed każdą próbą.
- Wszystkie 5 prób pick-and-place zarejestrowane do bag z pełnym pokryciem tematów.
- Metryki z tabeli 8.2 wyliczone i wpisane do protokołu odbioru.
- Repozytorium z tagiem v1.0 i plikiem environment.yml.
- Dokumentacja końcowa złożona przed terminem podanym przez prowadzącego.

## 9. Ćwiczenia laboratoryjne i kierunki badawcze

### 9.1. Ćwiczenia finalne

- Zaimplementuj węzeł HandControllerNode i wykonaj 10 cykli otwórz-zamknij dla każdego palca osobno. Zmierz opóźnienie komendy vs odpowiedź w stanie stawu dłoni.
- Uruchom pipeline percepcji: segmentacja chmury punktów → OBB → kandydaci chwytu → najlepszy PoseStamped. Wizualizuj wynik w RViz dla 3 różnych obiektów.
- Wytrenuj goal-conditioned politykę dla 2 obiektów z HER. Porównaj success rate z polityką bez HER (identyczne hiperparametry, 3 seedy każda).
- Zaimplementuj AdaptationEncoder i wykonaj trening fazy 2 RMA. Porównaj odporność na ±30% perturbacji masy: polityka z i bez RMA.
- Zintegruj wszystkie węzły w production.launch.py z TimerAction. Potwierdź poprawne uruchomienie przez ros2 node list i ros2 topic list.
- Przeprowadź pełny odbiór end-to-end: 5 prób pick-and-place na realnym G1. Wypełnij protokół odbioru i oblicz wszystkie metryki z sekcji 8.2.

### 9.2. Propozycje tematów prac dyplomowych

- **Adaptacja online z małą próbką:** Meta-RL (MAML, RL²) dla szybkiej adaptacji polityki do nowych obiektów na podstawie kilku demonstracji.
- **Hierarchiczne RL dla długich sekwencji:** Opcje i subgoals dla zadań wymagających wieloetapowego planowania (np. sortowanie wielu obiektów).
- **Percepcja bez kalibracji:** Polityki odporne na dryft kalibracji kamery przez data augmentation i domain randomization obrazu.
- **Dłoń dexterous w pętli RL:** Trening polityki sterującej zarówno ramieniem jak i palcami end-to-end dla zadań in-hand manipulation.
- **Bezpieczna eksploracja w RL:** Constrainted RL (CPO, FOCOPS) z jawnym ograniczeniem momentów stawów jako constraint, nie kara.
- **Wielorobotowa koordynacja:** Dwa G1 EDU współpracujące przy przenoszeniu dużych obiektów — shared policy lub MARL.

# Załącznik A. Kompletna struktura repozytorium

```
g1 project/
├── README.md                          # wersje, GPU, seed, instrukcja uruchomienia
├── environment.yml                    # conda env export
├── PROTOCOL ODBIORU.md                # wypełniony protokół z Ćw. 4
|
├── ros2 ws/
|   └── src/
|       ├── g1 bringup/
|       |   └── launch/
|       |       ├── full system.launch.py    # z Ćw. 2
|       |       └── production.launch.py     # z Ćw. 4
|       ├── g1 driver/
|       ├── g1 control/
|       |   ├── g1 control/
|       |   |   ├── task manager node.py
|       |   |   ├── arm controller node.py
|       |   |   ├── hand controller node.py
|       |   |   ├── policy node.py
|       |   |   ├── safety monitor node.py
|       |   |   └── safety filter.py
|       ├── g1 perception/
|       |   └── g1 perception/
|       |       ├── perception node.py
|       |       ├── grasp estimator.py
|       |       └── tf2 bridge.py
|       └── g1 interfaces/
|           └── msg/  srv/
|
├── sim/
|   └── IsaacLab/
|       └── source/isaaclab tasks/.../g1 tasks/
|           ├── g1 reach env.py
|           ├── g1 pickplace env.py
|           ├── g1 multitask env.py
|           └── reward.py
|
├── models/
|   ├── rma policy.py
|   ├── visual policy.py
|   └── goal conditioned actor.py
|
├── configs/
|   ├── ppo g1 reach.yaml
|   ├── ppo g1 multitask.yaml
|   └── bc g1 reach.json
|
├── datasets/
|   └── g1 reach v2.hdf5
|
├── logs/
|   ├── g1 multitask ppo v1/
|   |   ├── checkpoints/
|   |   ├── tb/
|   |   └── eval dr.csv
|   └── g1 reach bc v1/
|
├── experiments/
```

```
|    └── realrobot/
|        └── session final/
|            ├── session.bag
|            ├── metrics.csv
|            └── notes.md
|
└── scripts/
    ├── bootstrap.sh
    ├── train ppo.sh
    ├── train bc.sh
    ├── eval dr.py
    ├── deploy policy.py
    └── compute metrics.py
```

## Załącznik B. Dobre praktyki

- **Bezpieczeństwo przede wszystkim:** każda sesja z realnym robotem zaczyna się od checklisty i pilota w ręce operatora. Bez wyjątku.
- **Modularność kodu:** jeden pakiet, jedna odpowiedzialność. safety_monitor nigdy nie jest łączony z task_manager.
- **Odtwarzalność eksperymentów:** seed, commit, wersje, GPU — w każdym katalogu eksperymentu. Bez tego wynik jest anegdotą, nie wynikiem naukowym.
- **Waliduj w symulacji przed realnym robotem:** każda zmiana polityki, nagrody lub architektury musi przejść test w symulacji z DR zanim dotknie G1.
- **Curriculum i cierpliwość:** jeśli trening nie konwerguje po 500 iteracjach, zmień curriculum lub nagrodę, nie architekturę sieci.
- **Dokumentuj niepowodzenia:** opis co poszło źle i dlaczego jest wartościowszy niż opis co poszło dobrze. Przyszłe zespoły skorzystają.

## Załącznik C. Literatura i materiały źródłowe


- **PPO:** Schulman et al. (2017), arxiv.org/abs/1707.06347
- **HER:** Andrychowicz et al. (2017), arxiv.org/abs/1707.01495
- **RMA:** Kumar et al. (2021), arxiv.org/abs/2107.04034
- **GraspNet-1Billion:** Fang et al. (2020), arxiv.org/abs/2004.12057
- **R3M (Visual Encoder):** Nair et al. (2022), arxiv.org/abs/2203.12601
- **robomimic / BC:** Mandlekar et al. (2021), arxiv.org/abs/2108.03298
- **Isaac Lab:** isaac-sim.github.io/IsaacLab
- **Unitree G1 SDK2:** github.com/unitreerobotics/unitree_sdk2_python
- **Open3D:** open3d.org/docs
- **MoveIt 2:** moveit.ros.org

# Ćwiczenie 5: Modele Vision–Language–Action (VLA) i World-Model–Action (WMA) na Unitree G1

## Wprowadzenie

W laboratoriach z robotem Unitree G1 studenci poznali podstawy sterowania humanoidem, korzystając z instrukcji G1 oraz narzędzia ROS 2. Kolejne ćwiczenie koncentruje się na zaawansowanych modelach uczenia maszynowego, które integrują percepcję wizualną, rozumienie języka naturalnego i generowanie działań dla robota. Rodzina modeli UnifoLM obejmuje dwa kluczowe warianty: Vision–Language–Action (VLA) oraz World-Model–Action (WMA). Modele te są otwarte na platformie Hugging Face i stanowią fundament do budowy uniwersalnych agentów humanoidalnych.

## 1. Vision–Language–Action (VLA)

### 1.1 Koncepcja i motywacja

Modele Vision–Language–Action poszerzają tradycyjne modele wizja–język (VLM) o komponent predykcji akcji. Dzięki temu potrafią nie tylko odpowiadać na pytania o obrazy, ale także planować i wykonywać sekwencje działań w środowisku fizycznym. UnifoLM-VLA-0 bazuje na otwartym modelu Qwen 2.5-VL-7B i jest dalej trenowany na multimodalnym zbiorze danych obejmującym zadania robotyczne oraz sceny ogólne. W toku uczenia model łączy instrukcje tekstowe z informacją o 2D i 3D w scenie, a ponadto uczy się dynamiki manipulacji, co umożliwia mu planowanie dłuższych sekwencji akcji.

### 1.2 Architektura modelu VLA

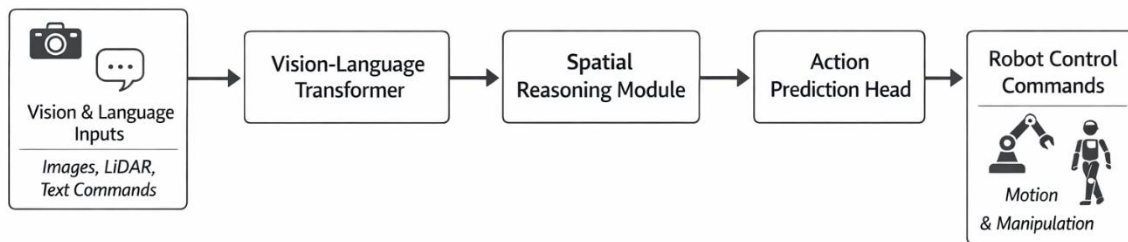
Model UnifoLM-VLA-0 składa się z kilku komponentów:

- Warstwa wejściowa pobiera obraz z kamer robota oraz komendę tekstową.
- Wspólna sieć transformera łączy sygnały wizualne i językowe, tworząc bogate reprezentacje multimodalne.
- Moduł wnioskowania przestrzennego integruje geometryczne cechy (położenia obiektów, relacje) z reprezentacją semantyczną.
- Głowa predykcji akcji generuje dyskretne lub ciągłe sekwencje ruchów ciała i dłoni G1. Integracja predykcji akcji oparta jest na chunkingu sekwencji oraz uwzględnia dynamikę forward i inverse.

Model używa ok. 340 godzin rzeczywistych danych z G1 oraz danych syntetycznych z symulacji. Dane obejmują m.in. zadania:

- G1_Stack_Block,
- G1_Bag_Insert,
- G1_Erase_Board,
- G1_Clean_Table,
- G1_Pack_PencilBox,
- G1_Pour_Medicine,
- G1_Pack_PingPong,
- G1_Prepare_Fruit,
- G1_Organize_Tools,
- G1_Fold_Towel,
- G1_Wipe_Table,

- G1_DualRobot_Clean_Table.




Rys. 1. Schemat architektury modelu VLA

### 1.3 Integracja z ROS 2

Aby wykorzystać model VLA na robocie G1, należy stworzyć pipeline integrujący przetwarzanie wizji, poleceń tekstowych i sterowania. Typowy przepływ obejmuje:

1. uruchomienie serwera modelu (np. za pomocą interfejsu Hugging Face lub repozytorium unifolm-vla),
2. węzeł ROS 2 odbierający obraz z kamer G1 i przekazujący go do modelu,
3. moduł NLP, który zamienia komendy użytkownika na wektory wejściowe,
4. węzeł decyzyjny wysyłający wynik predykcji akcji do sterowników G1 za pośrednictwem topics/serwisów ROS 2. Do komunikacji z robotem można wykorzystać pakiet unitree_ros2 lub pythonowy SDK.

W artykule Robonomics opisano pierwsze kroki z G1: przygotowano środowisko, zbudowano paczki ROS 2 z repozytorium unitree_ros2 i skonfigurowano komunikację z robotem. Odbierano stany kontrolera i uruchamiano przykłady, choć przykłady związane z ruchem wymagały dodatkowego testowania. Integracja z ROS 2 zapewnia skalowalność i strukturalność projektu.

## 2. World-Model–Action (WMA)

### 2.1 Koncepcja i motywacja

Świat-Model–Akcja (WMA) to architektura, w której model generatywny przewiduje przyszłe interakcje robota z otoczeniem. W UnifoLM-WMA-0 world-model uczy się praw fizycznych i dynamiki interakcji. Model działa w dwóch trybach:

- **Tryb decyzyjny** – world-model przewiduje wyniki interakcji, co pozwala ulepszać politykę decyzyjną.
- **Tryb symulacyjny** – world-model generuje realistyczne odpowiedzi środowiska na akcje robota, umożliwiając trening na danych syntetycznych.

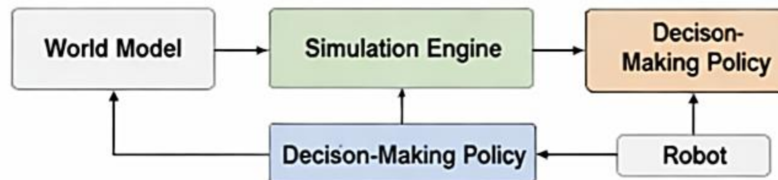
### 2.2 Architektura modelu WMA

Architektura UnifoLM-WMA-0 składa się z:

1. generatora wideo – po przystosowaniu na zbiorze Open-X generuje sekwencje wideo zgodne z poleceniami i obserwacjami;
2. world-modelu – sieć przewidująca przyszły stan świata na podstawie bieżących obserwacji i hipotezy akcji;
3. action head – polityki decyzyjnej uczonej w oparciu o przewidywania world-modelu; oraz
4. robota – realnego robota G1 lub Z1, który realizuje wygenerowane trajektorie.

Strategia treningowa obejmuje kolejno: fine-tuning generatora wideo na danych Open-X, post-trening world-modelu w trybie decyzyjnym na docelowych zadaniach, a następnie post-trening w trybie symulacyjnym. Wersje WMA_Base i WMA_Dual różnią się zestawem danych:

- WMA_Base trenuje na jednym zbiorze, natomiast
- WMA_Dual wykorzystuje pięć zestawów zadań Unitree (Z1_StackBox, Z1_DualArm_StackBox, Z1_DualArm_StackBox_V2, Z1_DualArm_Cleanup_Pencils, G1_Pack_Camera).




Rys. 2. Schemat architektury modelu WMA

### 2.3 Implementacja i pipeline

Implementacja WMA wymaga uruchomienia serwera world-modelu oraz modułu decyzyjnego. Unitree udostępnia repozytorium unifolm-world-model-action oraz checkpointy (WMA_Base, WMA_Dual) na Hugging Face. Przebieg treningu obejmuje:

- przygotowanie własnego zestawu danych w formacie LeRobot, konwersję do formatu HDF5 i RLDS;
- konfigurację parametrów (m.in. liczby stopni swobody) w plikach konfiguracyjnych;
- uruchomienie skryptu train.sh w celu przeprowadzenia treningu;
- uruchomienie trybu inference, w którym model generuje przewidywane stany i wideo na podstawie aktualnych obrazów oraz planowanej sekwencji akcji.

## 3. Propozycja eksperymentu laboratoryjnego

Ćwiczenie zakłada wykorzystanie robota Unitree G1 wraz z modelem VLA lub WMA.

1. Przygotowanie środowiska: zainstalowanie paczek unitree_ros2 oraz wymaganych bibliotek, uruchomienie serwerów modeli VLA/WMA na lokalnym komputerze.
2. Integracja z G1: napisanie węzła ROS 2, który pobiera obraz z kamery oraz komendy tekstowe (np. “podnieś czerwoną kostkę”). Węzeł powinien wysyłać je do modelu VLA/WMA i odbierać przewidywane akcje.
3. Realizacja zadania manipulacyjnego: wykorzystując model VLA, robot ma za zadanie umieścić obiekt na wyznaczonym miejscu (np. G1_Stack_Block).
4. Porównanie trybów WMA: w drugim etapie studenci uruchamiają model WMA w trybie decyzyjnym i symulacyjnym. W trybie symulacyjnym należy wygenerować sekwencje wideo i ocenić ich zgodność z rzeczywistym przebiegiem.
5. Ocena i dyskusja: studenci dokumentują liczbę udanych manipulacji, czas realizacji, liczbę błędów oraz oceniają, która architektura (VLA czy WMA) lepiej radzi sobie w danych zadaniach.

## 4. Metryki oceny wyników

Do porównania modeli i oceny eksperymentu można zastosować następujące metryki:

- Skuteczność zadania: odsetek zadań wykonanych poprawnie w określonym czasie.
- Czas realizacji: średni czas realizacji zadania od chwili otrzymania komendy do zakończenia manipulacji.
- Stabilność działania: liczba interakcji, w których robot zachował równowagę i nie popełnił błędów kolizji.
- Jakość generacji (dla WMA): porównanie przewidywanych sekwencji wideo z faktycznymi ruchami; oceniane subiektywnie przez obserwatora lub z wykorzystaniem miar porównujących trajektorie.

W niniejszym ćwiczeniu studenci poznają zaawansowane techniki łączenia wizji, języka i kontroli działania robotów. Modele UnifoLM stanowią kluczowy krok w kierunku uniwersalnych humanoidów, a ich integracja z rosłym ekosystemem ROS 2 pozwala na praktyczne eksperymenty w warunkach laboratoryjnych.

# 5. Instrukcja praktyczna krok po kroku

## 5.1 Cel i rezultat końcowy

Celem rozszerzonego ćwiczenia jest uruchomienie pełnego łańcucha decyzyjnego dla zadania manipulacyjnego: obraz z kamery -> komenda tekstowa -> predykcja celu lub akcji -> trajektoria ruchu -> zapis wyników do późniejszej analizy. Ćwiczenie należy wykonać dwuetapowo: najpierw w trybie mock na komputerze laboratoryjnym, a następnie – po weryfikacji logiki i bezpieczeństwa – z podłączeniem realnych źródeł danych z robota G1.

1. Po zakończeniu ćwiczenia student powinien umieć uruchomić pipeline ROS 2 bez robota, zinterpretować wynik modelu i zamienić go na trajektorię ramienia.
2. Student powinien umieć wskazać granicę między percepcją, decyzją i sterowaniem oraz uzasadnić, dlaczego nie należy łączyć tych warstw w jednym węźle.
3. Student powinien zapisać log epizodu do pliku i wykorzystać go jako punkt wyjścia do prostego eksperymentu WMA lub analizy offline.

**Uwaga praktyczna**
Tryb mock jest obowiązkowy przed pierwszym uruchomieniem na G1. Pozwala sprawdzić topiki, QoS, opóźnienia i logikę sterowania bez ryzyka kolizji oraz bez obciążania platformy obliczeniowej robota.

## 5.2 Wymagania środowiskowe

| Element | Wymaganie minimalne | Komentarz praktyczny |
|---|---|---|
| **System** | Ubuntu 22.04 + ROS 2 Humble | Wersja zgodna z docelowym środowiskiem produkcyjnym zespołu. |
| **Python** | Python 3.10 | Kod w przykładach jest przygotowany dla ament_python. |
| **Pakiety ROS 2** | rclpy, sensor_msgs, geometry_msgs, trajectory_msgs, cv_bridge | Wystarczają do uruchomienia trybu mock. |
| **Wizja** | OpenCV + NumPy | W przykładzie wykorzystywana jest detekcja czerwonego obiektu. |
| **Robot / SDK** | Unitree SDK2 Python lub adapter ROS 2 | Warstwa wykonawcza na robocie powinna być odseparowana od węzła VLA. |
| **Kamera** | RealSense D435i lub źródło syntetyczne | Na starcie rekomendowany jest generator obrazu z przykładu. |

| **Logowanie** | Plik JSONL | Ten sam zapis można później wykorzystać do analizy WMA. |
|---|---|---|

## 5.3 ARCHITEKTURA

Zalecana architektura laboratoryjna powinna być modularna i przewidywalna czasowo. Na potrzeby ćwiczenia rekomendowany jest następujący podział na warstwy:

| Warstwa | Pakiet / node | Wejścia | Wyjścia | QoS / uwagi |
|---|---|---|---|---|
| **Percepcja** | synthetic_camera_node lub realsense2_camera_node | — | /camera/color/image_raw | SensorData QoS, best_effort, depth=5 |
| **Wejście zadania** | task_command_node | — | /lab/task_command | reliable + transient_local |
| **Decyzja VLA** | mock_vla_bridge_node lub klient modelu VLA | obraz + komenda | /vla/target_pose, /vla/debug | reliable, oddzielony od sterowania |
| **Sterowanie** | trajectory_executor_node | PoseStamped | /lab/joint_trajectory lub topic kontrolera | reliable, ograniczenia workspace |
| **Logowanie** | episode_recorder_node | komenda + cel + trajektoria | plik JSONL | podstawa do analizy WMA |
| **Bezpieczeństwo** | safety_supervisor_node (produkcyjnie) | trajektoria / stan robota | stop / permit | na sprzęcie wymagany jako osobny węzeł |

Na realnym sprzęcie węzły związane z kamerą, sterownikiem i adapterem do G1 powinny działać jako Lifecycle Nodes, aby można było przejść przez stany configure -> activate -> deactivate bez restartu całego systemu. Węzeł decyzyjny VLA/WMA nie powinien mieć bezpośredniego dostępu do napędów – jego zadaniem jest jedynie publikowanie celu, trajektorii nominalnej albo polityki wysokiego poziomu.

## 5.4 PRZEPŁYW DANYCH

1. Kamera publikuje obraz na topic /camera/color/image_raw.
2. Węzeł task_command_node publikuje komendę tekstową na /lab/task_command z trwałością transient_local, aby nowy subskrybent otrzymał ostatnią komendę bez ponownego wpisywania.
3. Węzeł VLA łączy ostatni obraz i ostatnią komendę, wykonuje inferencję i publikuje cel manipulacyjny na /vla/target_pose.
4. Węzeł trajectory_executor_node zamienia cel na prostą trajektorię stawów, stosuje ograniczenia workspace i publikuje wynik.
5. Węzeł episode_recorder_node zapisuje komendę, cele i trajektorie do pliku JSONL, co umożliwia późniejszą ocenę jakości działania.
6. Po przejściu weryfikacji w trybie mock zamieniamy tylko źródło obrazu oraz temat wyjściowy trajektorii, bez przepisywania logiki decyzyjnej.

## 5.5 Tryb mock i tryb realny

| Element pipeline'u | Tryb mock | Tryb realny |
|---|---|---|
| **Źródło obrazu** | synthetic_camera_node | RealSense D435i / sterownik ROS 2 |
| **Model decyzyjny** | mock_vla_bridge_node | serwer VLA lub klient inferencji lokalnej |
| **Wyjście sterujące** | /lab/joint_trajectory | topic kontrolera lub adapter SDK2 |
| **Ryzyko kolizji** | praktycznie zerowe | wymaga nadzoru, limitów i operatora awaryjnego |

| **Cel etapu** | walidacja topików i logiki | wykonanie zadania na platformie G1 |
| --- | --- | --- |

### 5.6 Zasady bezpieczeństwa

Przed wejściem w tryb realny należy uwzględnić ograniczenia samego robota. Unitree G1 jest platformą o masie ponad 35 kg, z czujnikami RealSense D435i, lidarem 3D oraz zdalnym sterowaniem o zasięgu terenowym ponad 100 m. Oznacza to, że nawet „niewielki” błąd w logice ruchu może być istotny z punktu widzenia bezpieczeństwa operatora i otoczenia.

- Pierwsze testy ruchu wykonujemy wyłącznie na płaskim i twardym podłożu, z wolnym obszarem wokół robota.
- Student przy komputerze nie może jednocześnie pełnić roli operatora bezpieczeństwa; potrzebne są co najmniej dwie osoby.
- Jeżeli na robocie zamontowana jest zręczna dłoń Dex3-1, należy unikać pozycji leżenia i przysiadu podczas eksperymentu.
- Przed przejściem na realny topic sterowania należy włączyć debug / dry-run i zweryfikować, czy trajektorie są ograniczone do bezpiecznego workspace.
- Inferencję VLA/WMA trzeba ograniczać czasowo – brak odpowiedzi modelu nie może blokować warstwy sterowania w nieskończoność.

## 6. Minimalny pakiet ROS 2 – działający przykład

Poniższy przykład można uruchomić na zwykłym komputerze z ROS 2 Humble, bez realnego robota i bez modelu VLA. Został przygotowany tak, aby studenci mogli najpierw uruchomić przepływ danych end-to-end, a dopiero później wymienić elementy mock na źródło obrazu z kamery oraz właściwy moduł inferencji.

### 6.1 STRUKTURA PLIKÓW

```
g1_vla_lab/
├── package.xml
├── setup.py
├── setup.cfg
├── launch/
│   └── vla_lab.launch.py
├── resource/
│   └── g1_vla_lab
├── tools/
│   └── evaluate_episode.py
└── g1_vla_lab/
    ├── __init__.py
    ├── synthetic_camera_node.py
    ├── task_command_node.py
    ├── mock_vla_bridge_node.py
    ├── trajectory_executor_node.py
    └── episode_recorder_node.py
```

W archiwum dołączonym do instrukcji znajduje się pełna wersja pakietu. W samym dokumencie pokazano najważniejsze pliki potrzebne do zrozumienia architektury i przebiegu ćwiczenia.

### 6.2 KOD – synthetic_camera_node.py

*Listing 1. Generator obrazu publikujący syntetyczną scenę z czerwonym obiektem.*

```
from __future__ import annotations

import math

import cv2
import numpy as np
import rclpy
from cv_bridge import CvBridge
from rclpy.node import Node
from rclpy.qos import HistoryPolicy, QoSProfile, ReliabilityPolicy
from sensor_msgs.msg import Image
```

```
class SyntheticCameraNode(Node):
    def   init  (self) -> None:
        super().  init  ('synthetic camera node')
        self.declare parameter('width', 640)
        self.declare parameter('height', 480)
        self.declare parameter('fps', 10.0)
        self.declare parameter('animate', True)
        self.declare parameter('square size', 80)

        self.width = int(self.get parameter('width').value)
        self.height = int(self.get parameter('height').value)
        fps = float(self.get parameter('fps').value)
        self.animate = bool(self.get parameter('animate').value)
        self.square size = int(self.get parameter('square size').value)

        qos = QoSProfile(
            history=HistoryPolicy.KEEP LAST,
            depth=5,
            reliability=ReliabilityPolicy.BEST EFFORT,
        )
        self.publisher = self.create publisher(Image, '/camera/color/image raw', qos)
        self.bridge = CvBridge()
        self.start time = self.get clock().now()
        self.timer = self.create timer(1.0 / fps, self.publish frame)

    def publish frame(self) -> None:
        image = np.full((self.height, self.width, 3), 255, dtype=np.uint8)

        if self.animate:
            elapsed = (self.get clock().now() - self.start time).nanoseconds / 1e9
            center x = int((self.width * 0.5) + (self.width * 0.25) * math.sin(0.5 *
elapsed))
            center y = int((self.height * 0.5) + (self.height * 0.15) * math.cos(0.8 *
elapsed))
        else:
            center x = int(self.width * 0.6)
            center y = int(self.height * 0.45)

        half = self.square size // 2
        x0 = max(center x - half, 0)
        y0 = max(center y - half, 0)
        x1 = min(center x + half, self.width - 1)
        y1 = min(center y + half, self.height - 1)

        cv2.rectangle(image, (x0, y0), (x1, y1), (0, 0, 255), -1)
        cv2.putText(
            image,
            'Synthetic red block',
            (20, 40),
            cv2.FONT HERSHEY SIMPLEX,
            1.0,
            (20, 20, 20),
            2,
            cv2.LINE AA,
        )

        msg = self.bridge.cv2 to imgmsg(image, encoding='bgr8')
        msg.header.stamp = self.get clock().now().to msg()
        msg.header.frame id = 'camera color optical frame'
        self.publisher.publish(msg)


def main() -> None:
    rclpy.init()
    node = SyntheticCameraNode()
    try:
        rclpy.spin(node)
    finally:
        node.destroy node()
        rclpy.shutdown()


if   name   == '  main  ':
    main()
```

## 6.3 KOD – task_command_node.py

*Listing 2. Węzeł publikujący komendę tekstową z QoS transient_local.*

```
from   future   import annotations

import rclpy
from rclpy.duration import Duration
from rclpy.node import Node
from rclpy.qos import DurabilityPolicy, HistoryPolicy, QoSProfile, ReliabilityPolicy
from std msgs.msg import String


class TaskCommandNode(Node):
    def   init  (self) -> None:
```

```
        super().  init  ('task command node')
        self.declare parameter('command text', 'pick the red block and move slightly
left')
        self.declare parameter('publish period sec', 2.0)

        self.command text = str(self.get parameter('command text').value)
        publish period = float(self.get parameter('publish period sec').value)

        qos = QoSProfile(
            history=HistoryPolicy.KEEP LAST,
            depth=1,
            reliability=ReliabilityPolicy.RELIABLE,
            durability=DurabilityPolicy.TRANSIENT LOCAL,
        )
        self.publisher = self.create publisher(String, '/lab/task command', qos)
        self.timer = self.create timer(publish period, self.publish command)
        self.last publish = self.get clock().now()

    def publish command(self) -> None:
        msg = String()
        msg.data = self.command text
        self.publisher.publish(msg)
        self.get logger().info(f'Published command: {msg.data}')


def main() -> None:
    rclpy.init()
    node = TaskCommandNode()
    node.publish command()
    try:
        rclpy.spin(node)
    finally:
        node.destroy node()
        rclpy.shutdown()


if   name   == '  main  ':
    main()
```

## 6.4 KOD – mock_vla_bridge_node.py

*Listing 3. Mock VLA: łączy obraz i komendę, wykrywa czerwony obiekt i publikuje PoseStamped.*

```
from   future   import annotations

from dataclasses import dataclass
from typing import Optional, Tuple

import cv2
import numpy as np
import rclpy
from cv bridge import CvBridge
from geometry msgs.msg import PoseStamped
from rclpy.node import Node
from rclpy.qos import (
    DurabilityPolicy,
    HistoryPolicy,
    QoSProfile,
    ReliabilityPolicy,
)
from sensor msgs.msg import Image
from std msgs.msg import String


@dataclass
class DetectionResult:
    centroid u: float
    centroid v: float
    confidence: float


class MockVLABridgeNode(Node):
    def   init  (self) -> None:
        super().  init  ('mock vla bridge node')
        self.bridge = CvBridge()
        self.latest image: Optional[Image] = None
        self.latest command: str = ''

        sensor qos = QoSProfile(
            history=HistoryPolicy.KEEP LAST,
            depth=5,
            reliability=ReliabilityPolicy.BEST EFFORT,
        )
        command qos = QoSProfile(
            history=HistoryPolicy.KEEP LAST,
            depth=1,
            reliability=ReliabilityPolicy.RELIABLE,
            durability=DurabilityPolicy.TRANSIENT LOCAL,
        )
```

```
        self.create subscription(Image, '/camera/color/image raw', self.on image,
sensor qos)
        self.create subscription(String, '/lab/task command', self.on command,
command qos)
        self.pose publisher = self.create publisher(PoseStamped, '/vla/target pose', 10)
        self.debug publisher = self.create publisher(String, '/vla/debug', 10)
        self.timer = self.create timer(0.2, self.run inference)

    def on image(self, msg: Image) -> None:
        self.latest image = msg

    def on command(self, msg: String) -> None:
        self.latest command = msg.data.lower().strip()
        self.get logger().info(f'Received command: {self.latest command}')

    def detect red object(self, image bgr: np.ndarray) -> Optional[DetectionResult]:
        hsv = cv2.cvtColor(image bgr, cv2.COLOR BGR2HSV)
        lower red 1 = np.array([0, 100, 80], dtype=np.uint8)
        upper red 1 = np.array([10, 255, 255], dtype=np.uint8)
        lower red 2 = np.array([170, 100, 80], dtype=np.uint8)
        upper red 2 = np.array([180, 255, 255], dtype=np.uint8)

        mask 1 = cv2.inRange(hsv, lower red 1, upper red 1)
        mask 2 = cv2.inRange(hsv, lower red 2, upper red 2)
        mask = cv2.bitwise or(mask 1, mask 2)

        moments = cv2.moments(mask)
        area = moments['m00']
        if area < 1000.0:
            return None

        u = moments['m10'] / area
        v = moments['m01'] / area
        confidence = min(1.0, area / 20000.0)
        return DetectionResult(centroid u=u, centroid v=v, confidence=confidence)

    def command offsets(self) -> Tuple[float, float, float]:
        x offset = 0.0
        y offset = 0.0
        z offset = 0.0

        if 'left' in self.latest command:
            y offset += 0.06
        if 'right' in self.latest command:
            y offset -= 0.06
        if 'up' in self.latest command:
            z offset += 0.05
        if 'down' in self.latest command:
            z offset -= 0.05
        if 'far' in self.latest command:
            x offset += 0.05
        if 'near' in self.latest command:
            x offset -= 0.05

        return x offset, y offset, z offset

    def run inference(self) -> None:
        if self.latest image is None or not self.latest command:
            return

        image bgr = self.bridge.imgmsg to cv2(self.latest image,
desired encoding='bgr8')
        detection = self.detect red object(image bgr)
        if detection is None:
            return

        height, width = image bgr.shape[:2]
        normalized u = (detection.centroid u / width) - 0.5
        normalized v = (detection.centroid v / height) - 0.5
        x offset, y offset, z offset = self.command offsets()

        pose = PoseStamped()
        pose.header.stamp = self.get clock().now().to msg()
        pose.header.frame id = 'base link'

        pose.pose.position.x = 0.45 + x offset
        pose.pose.position.y = (-normalized u * 0.30) + y offset
        pose.pose.position.z = 0.22 + (-normalized v * 0.20) + z offset

        pose.pose.orientation.x = 0.0
        pose.pose.orientation.y = 1.0
        pose.pose.orientation.z = 0.0
        pose.pose.orientation.w = 0.0

        self.pose publisher.publish(pose)

        debug = String()
        debug.data = (
            f'detected red confidence={detection.confidence:.2f} '
            f'pixel=({detection.centroid u:.1f},{detection.centroid v:.1f}) '
            f'pose=({pose.pose.position.x:.3f},'
            f'{pose.pose.position.y:.3f},'
            f'{pose.pose.position.z:.3f})'
```

```
        )
        self.debug_publisher.publish(debug)


def main() -> None:
    rclpy.init()
    node = MockVLABridgeNode()
    try:
        rclpy.spin(node)
    finally:
        node.destroy_node()
        rclpy.shutdown()


if __name__ == '__main__':
    main()
```

## 6.5 KOD – trajectory_executor_node.py

*Listing 4. Warstwa sterowania zamieniająca cel kartezjański na prostą trajektorię stawów.*

```
from __future__ import annotations

from typing import List

import rclpy
from builtin_interfaces.msg import Duration
from geometry_msgs.msg import PoseStamped
from rclpy.node import Node
from std_msgs.msg import String
from trajectory_msgs.msg import JointTrajectory, JointTrajectoryPoint


def clamp(value: float, lower: float, upper: float) -> float:
    return max(lower, min(upper, value))


class TrajectoryExecutorNode(Node):
    def __init__(self) -> None:
        super().__init__('trajectory_executor_node')
        self.declare_parameter('trajectory_topic', '/lab/joint_trajectory')
        self.declare_parameter(
            'joint_names',
            [
                'left_shoulder_pitch_joint',
                'left_shoulder_roll_joint',
                'left_elbow_joint',
                'left_wrist_pitch_joint',
            ],
        )
        trajectory_topic = str(self.get_parameter('trajectory_topic').value)
        self.joint_names: List[str] = list(self.get_parameter('joint_names').value)

        self.publisher = self.create_publisher(JointTrajectory, trajectory_topic, 10)
        self.status_publisher = self.create_publisher(String, '/lab/executor_status',
10)
        self.create_subscription(PoseStamped, '/vla/target_pose', self.on_target_pose,
10)

    def on_target_pose(self, msg: PoseStamped) -> None:
        x = clamp(msg.pose.position.x, 0.25, 0.60)
        y = clamp(msg.pose.position.y, -0.25, 0.25)
        z = clamp(msg.pose.position.z, 0.05, 0.40)

        shoulder_pitch = clamp(0.55 - 1.2 * (x - 0.25), -1.0, 1.0)
        shoulder_roll = clamp(2.4 * y, -0.8, 0.8)
        elbow = clamp(1.30 - 2.4 * (z - 0.05), 0.2, 1.8)
        wrist_pitch = clamp(-0.3 + 0.5 * (x - 0.25), -0.8, 0.2)

        trajectory = JointTrajectory()
        trajectory.header.stamp = self.get_clock().now().to_msg()
        trajectory.joint_names = self.joint_names

        point = JointTrajectoryPoint()
        point.positions = [shoulder_pitch, shoulder_roll, elbow, wrist_pitch]
        point.velocities = [0.0] * len(self.joint_names)
        point.accelerations = [0.0] * len(self.joint_names)
        point.time_from_start = Duration(sec=1, nanosec=500000000)

        trajectory.points.append(point)
        self.publisher.publish(trajectory)

        status = String()
        status.data = (
            f'Published trajectory with joints='
            f'{[round(value, 3) for value in point.positions]}'
        )
        self.status_publisher.publish(status)
        self.get_logger().info(status.data)
```

```
def main() -> None:
    rclpy.init()
    node = TrajectoryExecutorNode()
    try:
        rclpy.spin(node)
    finally:
        node.destroy_node()
        rclpy.shutdown()


if __name__ == '__main__':
    main()
```

## 6.6 KOD – launch/vla_lab.launch.py

*Listing 5. Plik startowy uruchamiający kompletny pipeline w trybie mock.*

```
from launch import LaunchDescription
from launch_ros.actions import Node


def generate_launch_description() -> LaunchDescription:
    return LaunchDescription(
        [
            Node(
                package='g1_vla_lab',
                executable='synthetic_camera_node',
                name='synthetic_camera_node',
                output='screen',
                parameters=[{'animate': True, 'fps': 10.0}],
            ),
            Node(
                package='g1_vla_lab',
                executable='task_command_node',
                name='task_command_node',
                output='screen',
                parameters=[
                    {'command_text': 'pick the red block and move slightly left'},
                ],
            ),
            Node(
                package='g1_vla_lab',
                executable='mock_vla_bridge_node',
                name='mock_vla_bridge_node',
                output='screen',
            ),
            Node(
                package='g1_vla_lab',
                executable='trajectory_executor_node',
                name='trajectory_executor_node',
                output='screen',
                parameters=[{'trajectory_topic': '/lab/joint_trajectory'}],
            ),
            Node(
                package='g1_vla_lab',
                executable='episode_recorder_node',
                name='episode_recorder_node',
                output='screen',
            ),
        ]
    )
```

## 6.7 Komendy uruchomieniowe

```
mkdir -p ~/ros2_ws/src
cp -r g1_vla_lab_example ~/ros2_ws/src/g1_vla_lab

cd ~/ros2_ws
source /opt/ros/humble/setup.bash
colcon build --symlink-install
source install/setup.bash

ros2 launch g1_vla_lab vla_lab.launch.py
ros2 topic echo /vla/target_pose --once
ros2 topic echo /lab/joint_trajectory --once
tail -n 5 /tmp/g1_vla_lab_episode.jsonl
```

Dla komendy „pick the red block and move slightly left" wynik powinien pojawić się jednocześnie na topicu /vla/target_pose oraz /lab/joint_trajectory. Jeśli topiki działają poprawnie, logika przepływu danych została uruchomiona poprawnie.

## 6.8 POTENCJALNE PROBLEMY

- Brak danych na /vla/target_pose najczęściej oznacza problem z obrazem wejściowym albo brak zgodności QoS między publisherem i subskrybentem.
- Jeżeli komenda tekstowa nie jest widoczna po uruchomieniu nowego węzła, trzeba sprawdzić, czy publisher działa z durability=transient_local.
- cv_bridge i OpenCV muszą być zainstalowane w tym samym środowisku ROS 2; mieszanie kilku interpreterów Pythona zwykle kończy się błędem importu.
- Na realnym robocie nie wolno mapować wyniku modelu bezpośrednio na joint commands bez dodatkowych limitów i filtracji.

## 6.9 OPTYMALIZACJA

- Na etapie demonstratora wystarcza rozdzielczość 640x480; wyższa rozdzielczość zwiększa opóźnienie, a nie daje proporcjonalnego zysku w zadaniu typu pick-and-place.
- Węzeł inferencji powinien mieć osobny timer i własny timeout; brak odpowiedzi modelu nie może blokować sterowania.
- Na sprzęcie warto rozdzielić percepcję i model na osobne procesy lub kontenery, aby łatwiej mierzyć wykorzystanie CPU/GPU i restartować tylko wadliwy komponent.
- Wynik trajektorii należy wygładzać i ograniczać w osobnej warstwie bezpieczeństwa, zamiast „poprawiać” model VLA ręcznie w każdym zadaniu.

# 7. Rozszerzenie ćwiczenia do WMA

Ten sam pipeline można rozszerzyć do wariantu WMA bez przebudowy całego projektu. Najważniejsza zmiana polega na tym, że zaczynamy traktować zapisany epizod jako sekwencję obserwacji i działań, na podstawie której można analizować zgodność przewidywań z rzeczywistym przebiegiem zadania.

## 7.1 ARCHITEKTURA

W wariancie WMA pozostawiamy percepcję i sterowanie bez zmian, a pomiędzy logowaniem i analizą dodajemy warstwę world-modelu. W praktyce student powinien najpierw zbierać stabilne logi z komendami, celami i trajektoriami, a dopiero później wykorzystywać je do trenowania, walidacji lub porównania przewidywań kolejnego kroku.

## 7.2 PRZEPŁYW DANYCH

1. Uruchom pipeline w trybie mock albo w trybie realnym z ograniczoną prędkością.
2. Zapisz co najmniej kilka epizodów dla tej samej komendy, aby porównać powtarzalność wyniku.
3. Użyj logu JSONL do policzenia liczby predykcji i trajektorii na jedną komendę.
4. Dopiero po weryfikacji jakości logów przejdź do konwersji na bardziej złożony format danych.

## 7.3 KOD – prosta analiza zapisanego epizodu

*Listing 6. Narzędzie pomocnicze zliczające rekordy w pliku JSONL.*

```
#!/usr/bin/env python3
from __future__ import annotations

import json
import sys
from pathlib import Path


def main() -> None:
    if len(sys.argv) != 2:
        raise SystemExit('Usage: evaluate_episode.py /tmp/g1_vla_lab_episode.jsonl')

    path = Path(sys.argv[1])
    if not path.exists():
        raise SystemExit(f'File not found: {path}')
```

```
    commands = 0
    target_poses = 0
    joint_trajectories = 0

    with path.open('r', encoding='utf-8') as handle:
        for line in handle:
            record = json.loads(line)
            record_type = record.get('type')
            if record_type == 'command':
                commands += 1
            elif record_type == 'target_pose':
                target_poses += 1
            elif record_type == 'joint_trajectory':
                joint_trajectories += 1

    print(f'commands={commands}')
    print(f'target_poses={target_poses}')
    print(f'joint_trajectories={joint_trajectories}')
    if commands > 0:
        print(f'poses_per_command={target_poses / commands:.2f}')
        print(f'trajectories_per_command={joint_trajectories / commands:.2f}')


if __name__ == '__main__':
    main()
```

```
python3 tools/evaluate_episode.py /tmp/g1_vla_lab_episode.jsonl
```

### 7.4 POTENCJALNE PROBLEMY

- Logi z różnych uruchomień bez spójnych znaczników czasu są trudne do późniejszej synchronizacji.
- Jeżeli student zapisuje tylko trajektorie końcowe, a nie zapisuje wejściowych komend i celów, analiza WMA traci sens poznawczy.
- Dane z realnego robota należy czyścić z epizodów, w których interweniował operator lub supervisor bezpieczeństwa.

### 7.5 OPTYMALIZACJA

- Do analiz WMA zapisuj tylko te sygnały, które rzeczywiście wykorzystasz: timestamp, komenda, obserwacja, target pose, trajektoria, status wykonania.
- Rozdzielaj logi per zadanie i per dzień laboratoryjny; przyspiesza to późniejsze wyszukiwanie błędów i budowę datasetu.
- Jeżeli dostępne jest GPU, world-model uruchamiaj poza procesem sterowania i komunikuj się przez kolejkę lub usługę z ograniczonym timeoutem.

## 8. Propozycja przebiegu zajęć laboratoryjnych

| Etap | Czas orientacyjny | Co robi student | Wynik |
|---|---|---|---|
| **1. Budowa pakietu** | 20 min | Kompiluje pakiet i uruchamia launch w trybie mock. | Działające topiki /vla/target_pose i /lab/joint_trajectory |
| **2. Analiza logów** | 15 min | Sprawdza zapis JSONL i uruchamia evaluate_episode.py. | Krótki raport liczbowy |
| **3. Zamiana źródła obrazu** | 20 min | Podmienia synthetic_camera_node na kamerę rzeczywistą. | Obraz z kamery w tym samym pipeline'ie |
| **4. Integracja z G1** | 25 min | Przekierowuje wyjście sterowania do adaptera G1 lub kontrolera. | Bezpieczna trajektoria testowa |
| **5. Porównanie VLA/WMA** | 10 min | Opisuje, który wariant lepiej nadaje się do danego etapu zadania. | Wnioski do sprawozdania |

## 9. Co powinno znaleźć się w sprawozdaniu

1. schemat przepływu danych lub lista topików użytych podczas ćwiczenia;
2. krótki opis różnicy między warstwą percepcji, warstwą decyzyjną i sterowaniem;
3. zrzuty wyników ros2 topic echo dla /vla/target_pose i /lab/joint_trajectory;
4. metryki: skuteczność zadania, czas pojedynczego cyklu oraz liczba interwencji bezpieczeństwa;
5. jedna sytuacja błędna wraz z diagnozą przyczyny i sposobem naprawy;
6. wniosek, czy w danym zastosowaniu lepiej sprawdza się VLA, czy WMA i dlaczego.

## 10. Podsumowanie praktyczne

Najważniejszym rezultatem ćwiczenia nie jest „samo uruchomienie modelu”, lecz zbudowanie stabilnego, mierzalnego i bezpiecznego pipeline'u. W praktyce to właśnie poprawne rozdzielenie warstw, ustawienie QoS, obsługa timeoutów i logowanie epizodów decydują o tym, czy model VLA/WMA będzie użyteczny na platformie humanoidalnej. Wersja mock powinna stać się standardowym krokiem przygotowawczym przed każdą próbą na realnym G1.

---

g1_vla_lab_example.zip

Więcej na:
https://ai-robot-lab.github.io/_LAB-page/
https://matpomgit.github.io/RoboHub/

---

## *Torchę teorii*

## Modele WMA i VLA w robotach humanoidalnych na przykładzie Unitree G1 EDU

Modele **WMA (world-model–action)** i **VLA (Vision–Language–Action)** to dwa podejścia do „uczenia sterowania" robotem humanoidalnym, które różnią się tym, *gdzie* powstaje „inteligencja" i *jak* dochodzi do decyzji ruchu. WMA można myśleć jak o **wewnętrznym symulatorze/predyktorze przyszłości**: model świata przewiduje, co się stanie po wykonaniu danych akcji, a część „action" wykorzystuje te przewidywania do lepszego wyboru ruchu lub do wytwarzania danych treningowych. W przypadku Unitree jest to np. **UnifoLM-WMA-0**, gdzie świat-model działa jako **Simulation Engine** (generowanie syntetycznych danych) i jako **Policy Enhancement** (wspieranie podejmowania decyzji przez przewidywanie przyszłych interakcji). VLA to podejście „bardziej bezpośrednie": model dostaje **obraz (vision)** i **instrukcję (language)** i zwraca **sekwencję akcji (action)** – często jako „pakiet" kilku kroków sterowania (tzw. action chunk). Unitree unifikuje to podejście w **UnifoLM-VLA-0** i opisuje je jako drogę od samego „rozumienia obrazu i tekstu" (VLM) do „ucieleśnionego mózgu" zdolnego do fizycznej interakcji. Dla **Unitree G1 EDU** praktyczny wybór wygląda następująco: jeśli celem jest **szybkie uruchomienie wykonywania poleceń/manipulacji** (np. „posprzątaj stół", „złóż ręcznik"), zwykle zaczyna się od VLA (polityka end-to-end). Jeśli celem jest **planowanie, predykcja skutków, generowanie danych, długa sekwencja zachowań** i/lub wsparcie decyzji poprzez „wyobrażanie" przyszłości, naturalnym kandydatem jest WMA. Unitree udostępnia kody wdrożeniowe WMA w trybie klient–serwer (inference na serwerze, robot jako klient), co jest istotne, gdy robot nie ma wystarczającej mocy obliczeniowej na pokładzie.

### Modele WMA i VLA w prostych słowach

WMA (world-model–action) to architektura, w której „mózg" robota zawiera **model świata**: komponent uczący się, jak wygląda i zmienia się środowisko oraz jak środowisko reaguje na akcje robota. Unitree opisuje UnifoLM-WMA-0 jako architekturę z rdzeniem w postaci world-modelu rozumiejącego fizyczne interakcje robot–środowisko, który działa w dwóch rolach: **Simulation Engine** (interaktywny symulator generujący dane syntetyczne) oraz **Policy Enhancement** (połączenie z „action head", który korzysta z predykcji przyszłości do poprawy decyzji). VLA (Vision–Language–Action) to (najczęściej) **jedna polityka** ucząca się mapowania: *(obrazy + instrukcja) → (akcje sterujące)*. W UnifoLM-VLA-0 Unitree podkreśla, że model ma wyjść poza ograniczenia klasycznych VLM w interakcji fizycznej i stać się „embodied brain" dzięki kontynuowanemu pre-treningowi na danych manipulacyjnych. W praktyce oba podejścia często się łączy: VLA może być szybkim „sterownikiem umiejętności", a WMA – warstwą „przewidywania i planowania", która sprawdza konsekwencje i stabilizuje zachowanie (szczególnie w dłuższych horyzontach). Tę ideę (łączenie world modelu i VLA) widać też w innych projektach open-source, np. RynnVLA-002, które jawnie łączą model świata i model akcji w jednej ramie.

| Cecha | WMA (world-model–action) | VLA (vision–language–action) |
|---|---|---|
| **Intuicja** | Robot **przewiduje przyszłość** i wybiera ruch | Robot **patrzy + czyta polecenie** i od razu generuje ruch |
| **Typowe wejście** | Obraz + stan robota + (opcjonalnie instrukcja) + **kandydackie akcje** do symulacji | Obraz (często multi-view) + stan/proprio + instrukcja |

| Typowe wyjście | Predykcja przyszłych obserwacji/stanów **lub** wsparcie w wyborze akcji (action head) | Sekwencja akcji (często „chunk”, np. horyzont 16 kroków) |
|---|---|---|
| **Najmocniejsza strona** | Planowanie, długi horyzont, generowanie danych, „co jeśli” | Uniwersalne sterowanie zadaniami z języka, multi-task jedna polityka |
| **Ograniczenia** | Ciężkie obliczeniowo; ryzyko błędnych „halucynacji dynamiki”; złożony pipeline | Koszt inferencji i real-time; wrażliwość na dystrybucję danych i perturbacje |
| **Przykład Unitree** | UnifoLM-WMA-0 (tryb decision-making I simulation) | UnifoLM-VLA-0 (12 kategorii zadań na G1, jedna polityka) |
| **Dane i format** | Repo mówi o treningu na danych w formacie **LeRobot v2.1** (ważne dla przygotowania datasetu) | Dane Unitree na HF opisują m.in. pozycje kamer i 30 Hz rejestracji (przydatne do integracji sensoryki) |

### Jak „uczą się świata” i „planują akcje” w praktyce

WMA (wg repo UnifoLM-WMA-0) ma jasno opisany plan treningowy: najpierw **fine-tuning modelu generacji wideo** jako world-model na dużym zbiorze (Open-X), potem „post-training” w **decision-making mode**, a następnie (opcjonalnie) w **simulation mode**. To ważne, bo sugeruje, że world-model jest w dużej mierze modelem generatywnym przewidującym przebieg interakcji (często w formie predykcji kolejnych klatek/obserwacji), a planowanie przypomina **MPC / best-of-N**: sprawdzamy kilka wariantów akcji „w wyobraźni” i wybieramy najlepszy. VLA zwykle nie „planuje przez symulację” (choć może), ale generuje akcje bezpośrednio. Dla UnifoLM-VLA-Base można podejrzeć parametry architektury w `config.yaml`: backbone VLM to **Qwen2.5-VL** (pole `model_type: qwen2_5_vl`), a część akcyjna ma parametry typowe dla generowania sekwencji (np. `action_horizon: 16`, `future_action_window_size: 15`) oraz elementy sugerujące podejście dyfuzyjne/iteracyjne (`repeated_diffusion_steps`, `num_inference_timesteps`). W tym samym configu widać też, że trening przewiduje użycie obrazu z nadgarstka i propriocepcji (`use_wrist_image: true`, `use_proprio: true`).

## Architektura i komponenty na przykładzie Unitree G1 EDU

Dla praktycznej integracji z percepcją i stanem robota potrzebujesz co najmniej trzech strumieni: **Obraz z kamer**. W datasetach Unitree związanych z UnifoLM-VLA-0 podano m.in.: rozdzielczość 640×480, 30 Hz oraz rozmieszczenie kamer: „wrist-mounted (monocular) + head-mounted (binocular)”. To jest bardzo użyteczne, bo pokazuje realistyczny układ sensoryki używany do uczenia polityk manipulacji. **IMU / orientacja tułowia**. W ekosystemie Unitree SDK2 i symulatorze `unitree_mujoco` widać, że dla G1 istnieje strumień IMUState na temacie `rt/secondary_imu` (opisane jako „G1 only”). To typowy sygnał do stabilizacji, estymacji postawy i bezpiecznego ograniczania ruchu (np. gdy robot traci równowagę). **Propriocepcja i stany silników**. W dokumentacji ROS2/SDK Unitree pojawia się pojęcie „lowstate” zawierające m.in. `IMUState` i stany motorów (to kluczowy kanał do pętli sprzężenia zwrotnego).

## Przykładowe przepływy danych i architektury

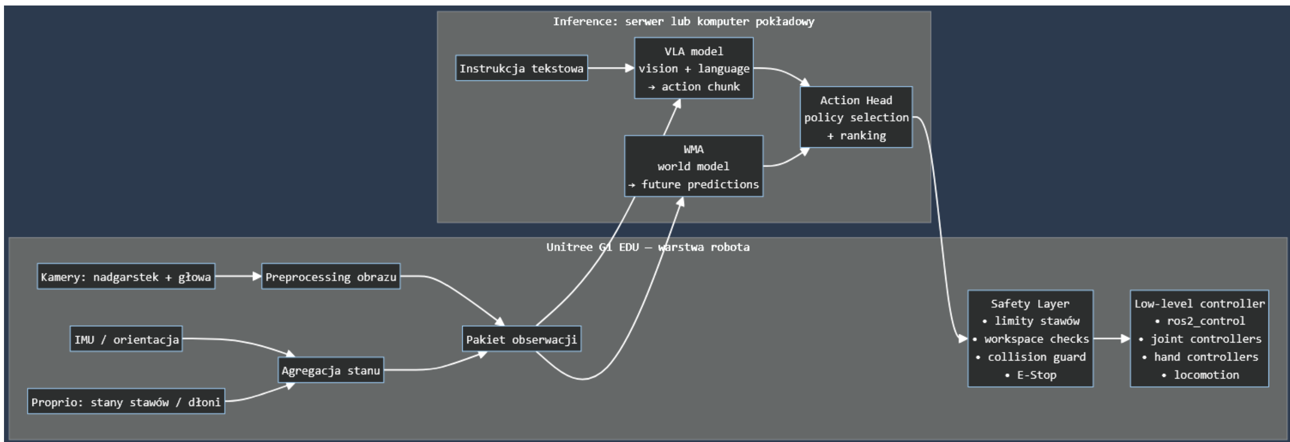
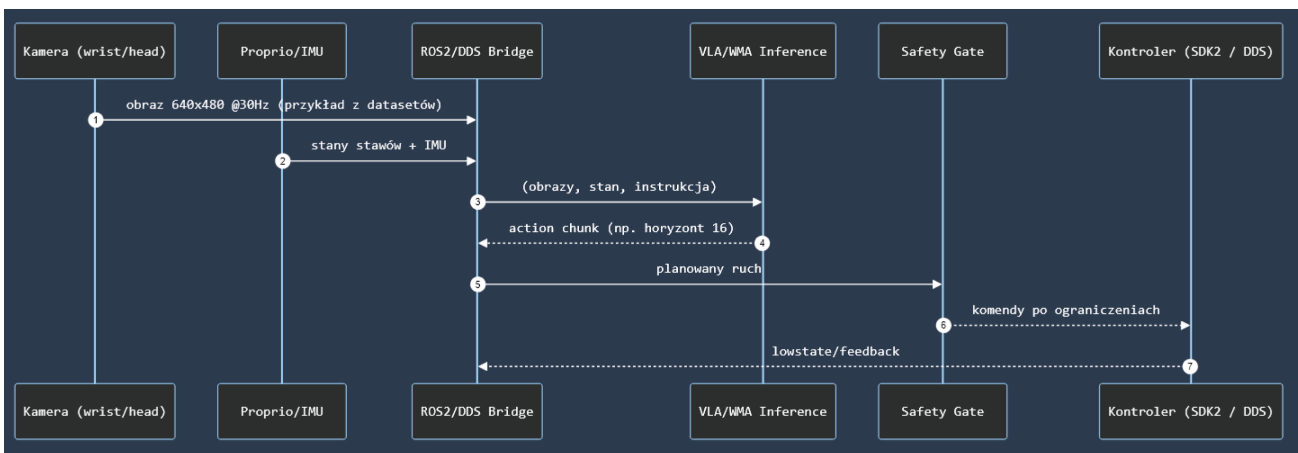



Diagram odzwierciedla dwa style podejmowania decyzji: bezpośredni (VLA) oraz planujący (WMA + action head), zgodnie z opisem UnifoLM-WMA-0 (decision-making vs simulation) i UnifoLM-VLA-0 (jedna polityka, multi-task).




Parametr „action_horizon: 16” oraz użycie proprio i wrist image wynikają bezpośrednio z konfiguracji UnifoLM-VLA-Base. Minimalne wymagania sprzętowe i programowe zależą od tego, czy inference robisz na robocie, czy na serwerze.

## Minimalny zestaw „wdrożeniowy” (rekomendowany dla dużych modeli)

| Element | Minimum praktyczne | Uzasadnienie / źródło |
|---|---|---|
| **Robot** | **Unitree G1 EDU** (nie zwykły G1) | Wersja G1 nie wspiera secondary development; EDU jest wskazane do rozwoju. |
| **Sterowanie I komunikacja** | Unitree **SDK2** (C++ lub Python) + DDS | `unitree_sdk2` opisuje środowisko (Ubuntu 20.04, gcc 9.4) i jest główną biblioteką; komunikacja oparta o DDS. |
| **Python na robocie/kliencie** | Python ≥3.8 (praktycznie: 3.10) | `unitree_sdk2_python` wskazuje Python ≥3.8 i zależności jak cyclonedds/opencv; projekty Unitree IL/LeRobot i WMA używają Pythona 3.10. |
| **ROS2 (opcjonalnie, ale bardzo użytecznie)** | Ubuntu 22.04 + ROS2 Humble (rekomendacja z Unitree ROS2) | `unitree_ros2` podaje testowane systemy i rekomenduje Humble na Ubuntu 22.04. |
| **Serwer inference (WMA/VLA)** | x86_64 + GPU (VRAM zależne od modelu) | Model UnifoLM-VLA-Base ma repo ~19 GB (same pliki wag), co zwykle wymaga GPU/VRAM lub agresywnej kwantyzacji. |

**Uwaga o mocy obliczeniowej:** nie ma w źródłach „twardego” minimum VRAM/RAM dla UnifoLM-VLA-Base/WMA, więc liczby typu „24 GB VRAM” należy traktować jako *praktyczne*

*heurystyki*, a nie wymagania. Najbardziej pewna informacja to rozmiar repo z wagami (19 GB) oraz fakt, że konfiguracja wspomina `flash_attention_2`, co zwykle oznacza GPU-centryczny tryb pracy.

### Wymagania programowe z repo Unitree

WMA (UnifoLM-WMA-0) – repo podaje konkretną instalację środowiska: Python 3.10.18, `pinocchio=3.2.0`, `ffmpeg=7.1.1`, klonowanie z submodułami i instalację `external/dlimp`. Unitree IL + LeRobot – repo `unitree_IL_lerobot` opisuje instalację LeRobot i środowiska, w tym pinocchio i ffmpeg oraz instalację `unitree_sdk2_python` do komunikacji DDS. LeRobotDataset v3.0 – jeśli budujesz własne dane na Hub, v3 zapewnia streaming i redukuje presję na system plików; dokumentacja podkreśla konieczność `finalize()` przy tworzeniu datasetu.

## Praktyczna implementacja: scenariusze i przykładowy kod

Poniższe przykłady są świadomie „proste” i mają pokazać, jak spiąć sensorykę, model i sterowanie. Interfejsy sterowania (np. konkretne API G1) mogą się różnić wersją robota/firmware – to część, którą trzeba zweryfikować na Twoim egzemplarzu (patrz sekcja wdrożenia).

### Scenariusz podążania za obiektem

Wariant A: **klasyczny** (bez VLA/WMA) – dobry jako test plumbing (kamera → decyzja → komenda).
Wariant B: **VLA-asystowany** – VLA dostaje obraz i prostą instrukcję „podążaj za X” i generuje akcje.

```
# Pseudokod/Python: podążanie za obiektem na podstawie obrazu
# Założenie: masz w ROS2 node odbierający obraz (np. /camera/image_raw)
# i możesz wysłać komendę prędkości (loco) przez SDK2/DDS lub ROS2 bridge.
import numpy as np
class FollowObjectController:
def __init__(self, loco_iface):
self.loco = loco_iface # TODO: adapter do Unitree SDK2 (G1 LocoClient) lub ROS2
cmd_vel
self.kp_yaw = 0.8
self.kp_forward = 0.4
def detect_target_center(self, rgb_image):
"""
Najprostsza detekcja: segmentacja koloru / marker ArUco / gotowy detektor.
Tu: placeholder zwracający (cx, cy) w pikselach lub None.
"""
return None # TODO
def step(self, rgb_image):
h, w = rgb_image.shape[:2]
target = self.detect_target_center(rgb_image)
if target is None:
# Bez celu: zatrzymaj się (bezpiecznie)
self.loco.set_velocity(vx=0.0, wz=0.0)
return
cx, cy = target
err_x = (cx - w/2) / (w/2) # -1..1
err_size = 0.0 # TODO: np. z bbox area → dystans
wz = -self.kp_yaw * err_x
vx = self.kp_forward * (0.2 - err_size) # prosta regulacja podejścia
# Safety clamp
vx = float(np.clip(vx, -0.3, 0.3))
wz = float(np.clip(wz, -0.6, 0.6))
self.loco.set_velocity(vx=vx, wz=wz)
```

Jeśli zamiast klasycznej reguły chcesz oprzeć „co zrobić” o VLA, możesz zastąpić `detect_target_center` i regulator wywołaniem polityki VLA (obraz + prompt). Sensowne jest wtedy generowanie „chunków” akcji i wykonywanie ich w pętli sterowania (np. 10–20 Hz), co jest spójne z ideą `action_horizon` i okna przyszłych akcji w konfiguracji UnifoLM-VLA-Base.

### Scenariusz reagowania na polecenia tekstowe/głosowe

Najprostszy, wdrożeniowy podział to:

1. **Wejście polecenia**: tekst (np. UI) lub mowa→tekst (ASR).
2. **Model**: VLA generuje akcje; WMA może weryfikować/przewidywać skutki.
3. **Bezpieczeństwo**: filtr poleceń, limity ruchu, warunek „robot gotowy".
4. **Sterowanie**: wysłanie do kontrolera (ramiona, dłonie, locomotion). W ekosystemie Unitree spotyka się interfejs audio po stronie robota (TTS/ASR). Wg opisu interfejsów wysokopoziomowych SDK2 dla G1 istnieje `AudioClient` obejmujący m.in. TTS i ASR; w praktyce dostępność może zależeć od wersji/serwera usług na robocie. Przykład interakcji audio przez ROS2 (TTS) pokazuje zewnętrzny driver/poradnik: wywołanie usługi z `request_data: 'volume=...;speak= ...'`.

```
# Pseudokod: "tekst -> VLA -> action chunk -> wykonanie"
# Zakładamy adaptery: get_observation(), vla_policy(), execute_action()
def vla_loop(vla_policy, robot_iface):
while True:
obs = robot_iface.get_observation() # obrazy + proprio + IMU
instruction = robot_iface.get_latest_instruction() # tekst z UI/ASR
if not instruction:
continue
# VLA zwraca sekwencję akcji (np. 16 kroków)
action_chunk = vla_policy.predict(obs, instruction)
# Prosty safety gate
if not robot_iface.is_safe_to_move():
robot_iface.stop()
continue
for a in action_chunk:
robot_iface.execute_action(a) # np. docelowe joint deltas / ee deltas
```

Jeżeli chcesz użyć WMA jako „kontrolera ryzyka", możesz wstawić krok: „zasymuluj 2–3 warianty chunków akcji" i wybrać ten, który minimalizuje ryzyko (np. kolizji, wyjścia poza workspace). To jest zgodne z opisem, że world-model w decision-making mode „predicts information about future physical interactions to assist the policy in generating actions".

### Kroki wdrożenia na Unitree G1 EDU

| Warstwa | Co instalujesz/uruchamiasz |
|---|---|
| SDK i komunikacja | `unitree_sdk2` (Ubuntu 20.04, gcc 9.4) oraz/lub `unitree_sdk2_python` (cyclonedds==0.10.2, opencv) |
| Środowisko WMA | conda env `unifolm-wma` (Python 3.10.18), `pinocchio=3.2.0`, `ffmpeg=7.1.1`, repo z submodułami |
| Środowisko IL/VLA (LeRobot) | conda env `unitree_lerobot` (Python 3.10), pinocchio, ffmpeg, instalacja LeRobot i `unitree_sdk2_python` |
| Dane do VLA | datasety Unitree „UnifoLM-VLA-0 Collection" (np. czyszczenie stołu) |
| Dane do WMA | repo WMA wymaga datasetów w formacie LeRobot v2.1; jest skrypt konwersji |
| Symulacja (opcjonalnie, ale zalecane) | `unitree_mujoco` (most sim2real, obsługa IMUState `rt/secondary_imu` dla G1) |

## Wdrożenie WMA w trybie klient–serwer

Repo UnifoLM-WMA-0 opisuje architekturę, w której inference działa na serwerze, a robot działa jako klient zbierający obserwacje i pytający o akcje. Kluczowe kroki (wprost z repo) to:

- Serwer: uruchomienie skryptu `run_real_eval_server.sh` po skonfigurowaniu checkpointu i datasetów w YAML.
- Robot-klient: przygotowanie środowiska `unitree_deploy` i usług/kontrolerów na robocie (tu możliwe, że część wymaga dostępu do dokumentacji producenta lub uprawnień).
- Połączenie: tunel SSH na port usługi inference (przykład: `-L 8000:127.0.0.1:8000`).

- Start klienta: uruchomienie `robot_client.py` z parametrami m.in. `--action_horizon 16` i `--control_freq 15`. Warto zaznaczyć dwie rzeczy praktyczne:
- **Model WMA na Hugging Face jest gated** (wymaga zaakceptowania warunków i udostępnienia danych kontaktowych), co może być realną barierą w pipeline CI/CD.
- Modele Unitree AI (w tym VLA/WMA) są publikowane na licencji **CC BY-NC-SA 4.0**, co zwykle wyklucza zastosowania komercyjne i wymaga kontroli zgodności prawnej.

## Jak sprawdzić informacje nieokreślone (np. firmware / wersje usług)

Firmware G1 zależy od egzemplarza i aktualizacji OTA. Najbardziej praktyczne sposoby weryfikacji:

- Sprawdź, czy w Twoim strumieniu stanu (np. „lowstate" w używanym bridge/driverze) jest pole `version` – przykładowy opis wiadomości lowstate zawiera `uint32[2] version`.
- Odnieś się do Unitree Document Center, do którego odsyła SDK2 (`support.unitree.com`) – część materiałów może wymagać konta/uprawnień producenta.
- Zrób test funkcji usług: znane są przypadki, w których pewne API mogą zwracać „not implemented on server" (przykład z issue dot. akcji ramion). To nie dowodzi, że u Ciebie będzie tak samo, ale jest dobrą motywacją, by dodać testy diagnostyczne przed uruchomieniem polityk.

## Ograniczenia, ryzyka i środki bezpieczeństwa

Najważniejsze ryzyka przy WMA/VLA na humanoidzie wynikają z: (a) real-time i opóźnień, (b) nieprzewidywalnych zachowań poza rozkładem danych, (c) bezpieczeństwa „wejść" (obrazy/polecenia), (d) ograniczeń integracji sprzętowej.

**Real-time / opóźnienie inference.** VLA potrafią być kosztowne obliczeniowo; istnieją prace przyspieszające inference, np. VLA-Cache (konsumuje ciągłość czasową i używa ponownie tokeny wizualne w KV-cache), aby zwiększyć częstotliwość sterowania. W ekosystemie LeRobot pojawiają się też metody stricte wdrożeniowe, jak

**Real-Time Chunking (RTC)**: asynchroniczne generowanie kolejnego chunka, gdy robot wykonuje bieżący, aby uniknąć „pauz/ szarpnięć" przy dużej latencji.

**Bezpieczeństwo na poziomie percepcji.** Badania pokazują, że VLA mogą być podatne na ataki/adwersarialne obrazy prowadzące do „zamrożenia" (robot przestaje reagować na instrukcje). FreezeVLA formalizuje ten problem i raportuje wysokie ASR w benchmarkach, co jest argumentem za filtrowaniem wejść wizualnych i dodaniem watchdogów wykonania.

**Ryzyko „halucynacji świata" w WMA.** Klasyczne prace o world-modelach pokazują, że politykę można trenować nawet „we śnie" generowanym przez model świata, a potem transferować do realnego środowiska – to potężne, ale wzmacnia potrzebę walidacji sim2real i ograniczeń bezpieczeństwa.

**Ograniczenia licencyjne i dostępu.** Modele Unitree (VLA/WMA) są CC BY-NC-SA 4.0, a przynajmniej część artefaktów może być gated (WMA-0-Base). Ponadto nie wszystkie funkcje usług mogą być dostępne w danej konfiguracji robota/serwera usług.

**Minimalne środki bezpieczeństwa przy wdrożeniu**

- **Zewnętrzny E-Stop** i procedura „safe stop" w oprogramowaniu (np. watchdog na brak feedbacku lowstate).
- **Twarde limity prędkości/siły/pozycji** na warstwie `Safety Gate` (nie ufaj wyjściu modelu bezpośrednio). W szczególności dla dłoni z siłowym sterowaniem i czujnikami dotyku.
- „**Tryb niskiej energii**" podczas pierwszych testów: małe prędkości, duże marginesy workspace, praca z miękkimi obiektami.

- **Telemetria i logowanie**: nagrywanie obrazów i stanów (w stylu datasetów Unitree: 30 Hz, multimodal) ułatwia diagnostykę i późniejsze douczenie.

## Propozycje dalszych badań i ulepszeń

Najbardziej obiecujące (i praktyczne) kierunki rozwoju dla G1 EDU to:

**Hybryda VLA + WMA**. W wielu zastosowaniach humanoida warto rozdzielić „rozumienie zadania" od „sprawdzenia konsekwencji": VLA wybiera ruch/umiejętność, a WMA ocenia czy plan jest bezpieczny (np. kolizje, rozlanie, upuszczenie). Takie łączenie modeli świata i modeli akcji jest aktywnym trendem (np. projekty łączące „action world model" i VLA w jednym frameworku).

**Optymalizacja inference**. Poza RTC i cache'owaniem tokenów, w praktyce możesz też rozważyć: kwantyzację, ograniczanie liczby kamer, zmniejszenie rozdzielczości wejścia (np. do 224×224 jak w configu UnifoLM-VLA-Base), lub heterogeniczne pipeline'y (VLM na serwerze, action expert lokalnie).

**Rozszerzanie modalności**. Jeśli Twój G1 EDU ma dłonie z dotykiem, możesz rozwijać polityki o sensory dotykowe; Dex3-1 ma warianty z macierzami czujników dotykowych.

**Skalowanie danych i streaming**. Przy własnych datasetach na HF Hub, LeRobotDataset v3.0 daje streaming i lepsze skalowanie plików; to ułatwia trenowanie większych polityk bez trzymania wszystkiego lokalnie.

**Lepsze „podstawowe klocki" polityk**. Jeśli Twoje zadania są kontaktowe i wielomodalne, warto znać fundamenty: Diffusion Policy (akcje jako proces dyfuzyjny) oraz ACT (Action Chunking with Transformers). Te prace stanowią bazę dla wielu współczesnych VLA.

## Linki do repozytoriów i modeli

Wskazówka praktyczna: jeśli ktoś planuje „prawdziwe wdrożenie" na G1 EDU, proponuję zacząć od repo `unitree_IL_lerobot` (łatwiej przejść ścieżkę: dane → trening → test) oraz od WMA repo (gotowy schemat klient–serwer). Dopiero potem rozbudowuj o hybrydę WMA+VLA i optymalizacje real-time.

```
Hugging Face (Unitree)
https://huggingface.co/unitreerobotics
https://huggingface.co/unitreerobotics/UnifoLM-VLA-Base
https://huggingface.co/unitreerobotics/UnifoLM-VLA-Libero
https://huggingface.co/unitreerobotics/UnifoLM-WMA-0-Base
https://huggingface.co/collections/unitreerobotics/unifolm-vla-0

Przykładowe datasety G1 (UnifoLM-VLA-0 Collection)
https://huggingface.co/datasets/unitreerobotics/G1_Clean_Table
https://huggingface.co/datasets/unitreerobotics/G1_Fold_Towel
https://huggingface.co/datasets/unitreerobotics/G1_Wipe_Table

GitHub (Unitree)
https://github.com/unitreerobotics/unifolm-world-model-action
https://github.com/unitreerobotics/unitree_IL_lerobot
https://github.com/unitreerobotics/unitree_sdk2
https://github.com/unitreerobotics/unitree_sdk2_python
https://github.com/unitreerobotics/unitree_mujoco

LeRobot (docs)
https://huggingface.co/docs/lerobot/en/lerobot-dataset-v3
https://huggingface.github.io/lerobot/

Prace / implementacje referencyjne (VLA i bezpieczeństwo)
https://github.com/openvla/openvla
https://huggingface.co/papers/2307.15818  (RT-2)
https://huggingface.co/papers/2509.19870  (FreezeVLA)
```