Instrukcja Wprowadzająca dla Studentów: ROS 2 i Programowanie Robota Unitree G1 EDU

Wstęp: Czym jest ROS 2 i dlaczego go używamy?

Zanim zagłębimy się w kod, ważne jest zrozumienie, czym jest ROS 2 (Robot Operating System 2) i jaką rolę odgrywa w programowaniu zaawansowanych robotów, takich jak Unitree G1 EDU.

Wyobraź sobie robota jako złożony organizm: ma "oczy" (kamery), "uszy" (mikrofony), "mózg" (komputer pokładowy) i "mięśnie" (silniki stawów). Aby to wszystko działało sprawnie, potrzebują one wspólnego systemu komunikacji i koordynacji.

ROS 2 jest właśnie takim systemem operacyjnym dla robotów. Jest to zbiór narzędzi, bibliotek i konwencji, które ułatwiają programistom tworzenie złożonych aplikacji robotycznych. Zamiast pisać od zera kod, który steruje każdym pojedynczym silnikiem, ROS 2 pozwala nam skupić się na logice wysokiego poziomu (np. "idź do przodu", "podnieś ramię"), delegując szczegóły na pakiety sterujące.

Kluczowe zalety ROS 2:

1. **Modułowość:** Każda funkcja robota (np. zbieranie danych z kamery, planowanie ruchu, sterowanie silnikami) jest oddzielnym programem zwanym **węzłem (node)**. Węzły komunikują się ze sobą, co ułatwia testowanie i rozbudowę systemu.
2. **Abstrakcja sprzętu:** Dzięki ROS 2, kod sterujący robotem jest w dużej mierze niezależny od konkretnego modelu. Możesz użyć tego samego interfejsu do sterowania ramieniem G1, ramieniem innej firmy, a nawet wirtualnym robotem w symulacji.
3. **Społeczność i narzędzia:** ROS 2 ma ogromną społeczność deweloperów. Dostępne są gotowe pakiety do nawigacji, wizji komputerowej, uczenia maszynowego i wielu innych zadań.

Podstawowe Koncepcje Komunikacji w ROS 2

Nasza demonstracja wykorzystuje dwa główne mechanizmy komunikacji, które są fundamentem ROS 2:

1. Tematy (Topics) – Komunikacja Strumieniowa

- **Zasada działania:** Temat działa jak strumień danych lub kanał informacyjny. Jeden węzeł publikuje dane na ten temat (np. "jestem tutaj"), a inne węzły, które go subskrybują, otrzymują te dane w czasie rzeczywistym. To jest jednokierunkowa komunikacja typu "jeden do wielu".
- **Zastosowanie w G1:** Używamy tematu **/cmd_vel** do sterowania robotem. Węzeł g1_demo_node.py publikuje na tym temacie komendę prędkości (Twist). Kontroler robota G1 subskrybuje ten temat i interpretuje komendę jako instrukcję do chodzenia. To jest idealne do ciągłego, dynamicznego ruchu.

2. Akcje (Actions) – Komunikacja Zadaniowa

- **Zasada działania:** Akcje służą do wykonywania złożonych zadań, które trwają pewien czas i wymagają sprzężenia zwrotnego. Działają na zasadzie "cel-informacja zwrotna-wynik". Węzeł wysyła cel (Goal), a serwer akcji na bieżąco informuje o postępie (Feedback) i na końcu zwraca ostateczny wynik (Result).
- **Zastosowanie w G1:** Używamy akcji **/arm_controller/follow_joint_trajectory** do sterowania ramieniem. Zamiast wysyłać ciągłe komendy prędkości dla każdego stawu z osobna, wysyłamy jeden cel: "Wykonaj trajektorię machania ręką". Robot wykonuje całe zadanie, a my dostajemy informację zwrotną, kiedy machanie się zakończyło. Jest to niezbędne przy sterowaniu precyzyjnym.

Dlaczego kod demonstracyjny jest napisany w Pythonie?

W programowaniu robotów ROS 2 najczęściej używa się języków C++ i Python. Nasz kod demonstracyjny został napisany w Pythonie z kilku kluczowych powodów:

1. **Łatwość nauki:** Python jest znacznie prostszy do nauki i zrozumienia dla początkujących studentów. Skupiamy się na logice robotyki, a nie na złożonej składni C++.
2. **Szybkie prototypowanie:** Python pozwala szybko pisać i testować nowe pomysły. Idealnie nadaje się do tworzenia logiki wysokiego poziomu i interfejsów użytkownika.
3. **Wysokopoziomowa kontrola:** W kontekście robota G1, wewnętrzne sterowniki silników są zaimplementowane w C++ ze względów wydajnościowych. Python pełni rolę "mózgu decyzyjnego", który wysyła proste polecenia (np. "idź do przodu") do tych wydajnych sterowników.

Przegląd Kodu Demonstracyjnego g1_demo_node.py

Poniżej znajduje się omówienie kluczowych części kodu, które zostaną zaprezentowane podczas demonstracji:

1. Struktura Węzła (class G1DemoNode(Node))

- **Inicjalizacja (__init__)**: W tym miejscu konfigurujemy wszystkie połączenia (Tematy i Akcje), które będą używane przez węzeł. W naszym przypadku są to:
    - arm_action_client: Klient do sterowania ramieniem.
    - base_action_client: Klient do zmiany postawy całego ciała robota (np. wstawanie).
    - cmd_vel_publisher: Wydawca komend prędkości dla chodu.

2. Sekwencja Demonstracyjna (run_demo())

Główna funkcja run_demo() zawiera sekwencję, która będzie wykonywana:

- **Krok 1: Wstawanie (set_robot_posture('stand'))**: Używamy Akcji, aby bezpiecznie przejść z pozycji spoczynkowej do pozycji stojącej. To pokazuje, jak robimy złożone, precyzyjne ruchy.
- **Krok 2: Chód do przodu (move_robot_base(...))**: Używamy Tematu (/cmd_vel) do wysłania komendy prędkości. Robot interpretuje to jako chód. Po upływie określonego czasu, węzeł wysyła komendę zatrzymania.
- **Krok 3: Machanie ramieniem (perform_arm_wave())**: Ponownie używamy Akcji, aby wykonać zdefiniowaną trajektorię ruchu ramienia. To demonstruje precyzyjną kontrolę nad manipulatorem.
- **Krok 4: Siadanie (set_robot_posture('sit'))**: Powrót do pozycji spoczynkowej za pomocą Akcji.

3. Kluczowe Funkcje Pomocnicze

- **send_trajectory_goal(...)**: Ta funkcja jest sercem sterowania ramieniem i postawą. Przyjmuje listę punktów (trajektorię) i wysyła je do kontrolera robota.
- **move_robot_base(...)**: Ta funkcja tworzy i publikuje wiadomość Twist (prędkość liniowa i kątowa) na temat /cmd_vel.

Podsumowanie: Nasz kod Pythonowy działa jako "dyrygent" orkiestry ROS 2, używając różnych interfejsów (Tematów i Akcji) do koordynacji złożonych ruchów robota Unitree G1 EDU.

Ten materiał zawiera kompletny pakiet ROS 2 do demonstracji podstawowych możliwości robota Unitree G1 EDU. Kod jest zaprojektowany tak, aby w sposób przystępny pokazać studentom kluczowe koncepcje programowania robotów: komunikację przez tematy (Topics) oraz akcje (Actions).

Cel Demonstracji

Głównym celem jest zaprezentowanie, jak programowo:

1. Uruchomić robota do pozycji stojącej.
2. Wykonać podstawowy ruch (chód do przodu).
3. Sterować ramieniem robota (np. wykonać gest "machania").
4. Zakończyć działanie robota (pozycja spoczynkowa).

Struktura Pakietu ROS 2

Stworzymy prosty pakiet o nazwie g1_demo_package. Pakiet ten będzie zawierał jeden główny skrypt Pythona (g1_demo_node.py) oraz pliki konfiguracyjne.

Wymagania wstępne:

- Środowisko ROS 2 (np. Humble, Iron) zainstalowane na komputerze sterującym.
- Pakiety ROS 2 specyficzne dla Unitree G1 (np. unitree_ros2) muszą być zainstalowane i skonfigurowane zgodnie z dokumentacją producenta. Kod zakłada, że robot G1 jest dostępny w sieci i jego kontrolery są poprawnie zainicjowane.

1. Kod Główny (Python)

Poniższy skrypt Pythona implementuje logikę demonstracyjną. Wykorzystuje on standardowe interfejsy ROS 2:

- **Akcje (control_msgs/action/FollowJointTrajectory):** Do sterowania precyzyjnymi ruchami ramion i nóg. Jest to standardowy sposób wysyłania celów dla kontrolerów stawów.
- **Tematy (geometry_msgs/msg/Twist):** Do wysyłania komend prędkości dla poruszania się.

g1_demo_node.py

code Python
downloadcontent_copy
expand_less

```
# !/usr/bin/env python3

import rclpy
from rclpy.node import Node
from rclpy.action import ActionClient
from rclpy.callback_groups import MutuallyExclusiveCallbackGroup

from geometry_msgs.msg import Twist
from trajectory_msgs.msg import JointTrajectory, JointTrajectoryPoint
from control_msgs.action import FollowJointTrajectory

import time
import math

class G1DemoNode(Node):
    """
    Węzeł demonstracyjny dla robota Unitree G1 EDU.
    Demonstruje:
    1. Inicjalizację i pozycję stojącą (poprzez Action Client).
    2. Podstawowy chód do przodu (poprzez Topic Publisher).
    3. Sterowanie ramieniem - ruch "machania" (poprzez Action Client).
    4. Pozycję spoczynkową (poprzez Action Client).
    """

    def __init__(self):
        super().__init__('g1_demo_node')
        self.get_logger().info('Initializing G1 Demo Node...')

        # Użycie grup callbacków dla oddzielenia logiki Actions od Topics
        self.action_callback_group = MutuallyExclusiveCallbackGroup()
        self.topic_callback_group = MutuallyExclusiveCallbackGroup()

        # Konfiguracja klienta akcji dla kontrolera ramienia
        # Zakładamy, że robot używa standardowego interfejsu ROS 2 dla ramienia,
        # np. '/arm_controller/follow_joint_trajectory'.
        self.arm_action_client = ActionClient(
            self,
```

```
            FollowJointTrajectory,
            '/arm_controller/follow_joint_trajectory',
            callback_group=self.action_callback_group
        )

        # Konfiguracja klienta akcji dla kontrolera całego ciała/postawy
        # Używamy tego do przechodzenia do pozycji "stand" i "sit".
        # Nazwa topicu/akcji może się różnić w zależności od konkretnej
konfiguracji Unitree G1.
        self.base_action_client = ActionClient(
            self,
            FollowJointTrajectory,
            '/full_body_controller/follow_joint_trajectory',
            callback_group=self.action_callback_group
        )

        # Konfiguracja wydawcy tematu dla komend prędkości (chodzenia)
        # Standardowy temat ROS 2 do sterowania prędkością.
        self.cmd_vel_publisher = self.create_publisher(
            Twist,
            '/cmd_vel',
            10,
            callback_group=self.topic_callback_group
        )

        self.get_logger().info('Clients and Publishers created successfully.')
        self.run_demo()

    def send_trajectory_goal(self, action_client, joint_names, points,
timeout=5.0):
        """
        Pomocnicza funkcja do wysyłania celu ActionClient.
        :param action_client: Klient akcji (np. self.arm_action_client).
        :param joint_names: Lista nazw stawów (np. ['joint1', 'joint2']).
        :param points: Lista obiektów JointTrajectoryPoint.
        :param timeout: Limit czasu oczekiwania na połączenie z serwerem akcji.
        """
        self.get_logger().info(f'Sending trajectory goal for
{action_client._action_name}...')

        # 1. Sprawdzenie dostępności serwera akcji
        if not action_client.wait_for_server(timeout_sec=timeout):
            self.get_logger().error(f'Action server {action_client._action_name}
not available after {timeout} seconds.')
            return False

        # 2. Tworzenie celu akcji
        goal_msg = FollowJointTrajectory.Goal()
        goal_msg.trajectory.joint_names = joint_names
        goal_msg.trajectory.points = points

        # 3. Wysyłanie celu i oczekiwanie na odpowiedź
        send_goal_future = action_client.send_goal_async(goal_msg)
        self.get_logger().info('Waiting for result...')

        # Czekamy na wynik asynchronicznie, aby nie blokować pętli ROS 2
        rclpy.spin_until_future_complete(self, send_goal_future)

        if send_goal_future.result():
            result_future = send_goal_future.result().get_result_async()
            rclpy.spin_until_future_complete(self, result_future)
            if result_future.result().result.error_code ==
result_future.result().result.SUCCESSFUL:
                self.get_logger().info('Goal completed successfully.')
```

```
                return True
            else:
                self.get_logger().warn(f'Goal failed with error code:
{result_future.result().result.error_code}')
                return False
        else:
            self.get_logger().error('Failed to send goal or goal was rejected.')
            return False

    def move_robot_base(self, linear_velocity, angular_velocity, duration):
        """
        Wysyła komendy prędkości na temat /cmd_vel przez określony czas.
        :param linear_velocity: Prędkość liniowa (m/s).
        :param angular_velocity: Prędkość kątowa (rad/s).
        :param duration: Czas trwania ruchu (sekundy).
        """
        self.get_logger().info(f'Moving robot at velocity: {linear_velocity} m/s
for {duration} seconds...')
        twist = Twist()
        twist.linear.x = linear_velocity
        twist.angular.z = angular_velocity

        start_time = self.get_clock().now().nanoseconds / 1e9
        while (self.get_clock().now().nanoseconds / 1e9 - start_time) <
duration:
            self.cmd_vel_publisher.publish(twist)
            time.sleep(0.01)

        # Zatrzymaj robota po upływie czasu
        twist.linear.x = 0.0
        twist.angular.z = 0.0
        self.cmd_vel_publisher.publish(twist)
        self.get_logger().info('Robot stopped.')

    def perform_arm_wave(self):
        """
        Wykonuje gest machania ramieniem.
        Definicja trajektorii dla przykładowego ruchu.
        Zakładamy, że ramie ma 6 stopni swobody (joint names w przykładzie są
uproszczone).
        """
        self.get_logger().info('Starting arm wave demonstration...')

        # Nazwy stawów ramienia G1 (przykładowe, należy zweryfikować w
dokumentacji Unitree)
        arm_joint_names = [
            'arm_shoulder_roll_joint',
            'arm_shoulder_pitch_joint',
            'arm_shoulder_yaw_joint',
            'arm_elbow_pitch_joint',
            'arm_wrist_yaw_joint',
            'arm_wrist_pitch_joint'
        ]

        # Trajektoria machania: pozycja neutralna -> pozycja "w górę" -> powrót
do neutralnej
        # Punkt 1: Pozycja neutralna (przyjmujemy 0)
        p1 = JointTrajectoryPoint()
        p1.positions = [0.0] * len(arm_joint_names)
        p1.time_from_start.sec = 0

        # Punkt 2: Ramię w górę i lekko do przodu (gest machania)
        p2 = JointTrajectoryPoint()
```

```
        p2.positions = [0.0, -1.0, 0.0, 0.5, 0.0, 0.0]  # Przykładowe wartości
kątów (rad)
        p2.time_from_start.sec = 1

        # Punkt 3: Powrót do pozycji neutralnej
        p3 = JointTrajectoryPoint()
        p3.positions = [0.0] * len(arm_joint_names)
        p3.time_from_start.sec = 2

        # Stworzenie trajektorii z punktów
        trajectory_points = [p1, p2, p3]

        # Wysłanie trajektorii do kontrolera ramienia
        self.send_trajectory_goal(self.arm_action_client, arm_joint_names,
trajectory_points)

    def set_robot_posture(self, posture_name):
        """
        Ustawia postawę robota (np. "stand" lub "sit").
        :param posture_name: Nazwa docelowej postawy.
        """
        self.get_logger().info(f'Setting robot posture to: {posture_name}')

        # Nazwy wszystkich stawów robota G1 (przykładowe, należy zweryfikować)
        full_body_joint_names = [
            'left_hip_roll_joint', 'left_hip_pitch_joint',
'left_knee_pitch_joint',
            'right_hip_roll_joint', 'right_hip_pitch_joint',
'right_knee_pitch_joint',
            'arm_shoulder_roll_joint', 'arm_shoulder_pitch_joint',
'arm_elbow_pitch_joint'
            # ... oraz inne stawy ramion i tułowia
        ]

        # Definicja pozycji "stand" i "sit" (przykładowe wartości kątów)
        if posture_name == 'stand':
            # Ustawienie pozycji stojącej (wartości zależą od kalibracji robota)
            joint_positions = [0.0] * len(full_body_joint_names)
        elif posture_name == 'sit':
            # Ustawienie pozycji spoczynkowej
            joint_positions = [0.0] * len(full_body_joint_names) # Zastąpić
rzeczywistymi wartościami "sit"
        else:
            self.get_logger().error(f"Unknown posture: {posture_name}")
            return False

        # Utworzenie punktu trajektorii dla docelowej postawy
        p = JointTrajectoryPoint()
        p.positions = joint_positions
        p.time_from_start.sec = 2  # Czas trwania przejścia do postawy

        self.send_trajectory_goal(self.base_action_client,
full_body_joint_names, [p])


    def run_demo(self):
        """
        Główna sekwencja demonstracyjna.
        """
        self.get_logger().info('--- Starting G1 Educational Demo ---')

        # Krok 1: Inicjalizacja do pozycji stojącej
        self.get_logger().info('Step 1: Stand Up')
        # Zastąpienie realnymi danymi dla 'stand_up' (puste dla demo)
```

```
        self.set_robot_posture('stand')
        time.sleep(3)

        # Krok 2: Chód do przodu (użycie topicu /cmd_vel)
        self.get_logger().info('Step 2: Walk Forward')
        self.move_robot_base(linear_velocity=0.2, angular_velocity=0.0,
duration=4.0)
        time.sleep(2)

        # Krok 3: Demonstracja ramienia (użycie akcji FollowJointTrajectory)
        self.get_logger().info('Step 3: Arm Manipulation (Waving)')
        self.perform_arm_wave()
        time.sleep(2)

        # Krok 4: Zakończenie - pozycja spoczynkowa
        self.get_logger().info('Step 4: Sit Down')
        self.set_robot_posture('sit')
        time.sleep(3)

        self.get_logger().info('--- Demo Finished ---')
        rclpy.shutdown()

def main(args=None):
    rclpy.init(args=args)
    demo_node = G1DemoNode()
    rclpy.spin(demo_node)

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

2. Pliki konfiguracyjne pakietu

Aby uruchomić skrypt jako węzeł ROS 2, potrzebujemy standardowych plików konfiguracyjnych pakietu.

package.xml

Utwórz plik package.xml w katalogu pakietu (g1_demo_package).

code Xml
downloadcontent_copy
expand_less

```
<?xml version="1.0"?>
<?xml-model href="http://download.ros.org/schema/package_format3.xsd"
schematypens="http://www.w3.org/2001/XMLSchema"?>
<package format="3">
  <name>g1_demo_package</name>
  <version>0.0.0</version>
  <description>Package to demonstrate Unitree G1 EDU functionalities for
students.</description>
  <maintainer email="user@example.com">Your Name</maintainer>
  <license>Apache-2.0</license>

  <depend>rclpy</depend>
  <depend>geometry_msgs</depend>
  <depend>trajectory_msgs</depend>
  <depend>control_msgs</depend>
```

```
  <test_depend>ament_copyright</test_depend>
  <test_depend>ament_flake8</test_depend>
  <test_depend>ament_pep257</test_depend>
  <test_depend>python3-pytest</test_depend>

  <export>
    <build_type>ament_python</build_type>
  </export>
</package>
```

setup.py

Utwórz plik setup.py w katalogu pakietu (g1_demo_package).

code Python
downloadcontent_copy
expand_less

```
from setuptools import setup

package_name = 'g1_demo_package'

setup(
    name=package_name,
    version='0.0.0',
    packages=[package_name],
    data_files=[
        ('share/ament_index/resource_index/packages',
            ['resource/' + package_name]),
        ('share/' + package_name, ['package.xml']),
    ],
    install_requires=['setuptools'],
    zip_safe=True,
    maintainer='Your Name',
    maintainer_email='user@example.com',
    description='Package to demonstrate Unitree G1 EDU functionalities for
students.',
    license='Apache-2.0',
    tests_require=['pytest'],
    entry_points={
        'console_scripts': [
            'run_demo = g1_demo_package.g1_demo_node:main'
        ],
    },
)
```

3. Przygotowanie i uruchomienie

Krok 1: Struktura katalogów

Upewnij się, że Twoja struktura katalogów wygląda następująco:

code Code
downloadcontent_copy
expand_less

```
ros2_ws/
```

```
├── src/
│   ├── g1_demo_package/
│   │   ├── g1_demo_package/
│   │   │   ├── g1_demo_node.py
│   │   ├── setup.py
│   │   └── package.xml
```

Krok 2: Kompilacja i instalacja pakietu

W terminalu w katalogu ros2_ws wykonaj:

code Bash
downloadcontent_copy
expand_less

```
colcon build --packages-select g1_demo_package
```

Następnie załaduj środowisko ROS 2 (jeśli jeszcze tego nie zrobiłeś):

code Bash
downloadcontent_copy
expand_less

```
source install/setup.bash
```

Krok 3: Uruchomienie robota (G1)

Upewnij się, że robot G1 jest włączony, w trybie "idle" i że pakiety Unitree (np. unitree_ros2) są uruchomione. Zazwyczaj jest to realizowane przez uruchomienie launch file dostarczonego przez Unitree, który uruchamia kontrolery ROS 2.

code Bash
downloadcontent_copy
expand_less

```
ros2 launch unitree_g1_config unitree_g1_launch.py # Przykładowa komenda launch
```

Krok 4: Uruchomienie demonstracji

W nowym terminalu uruchom węzeł demonstracyjny:

code Bash
downloadcontent_copy
expand_less

```
ros2 run g1_demo_package run_demo
```

Analiza Kodu dla Studentów

Podczas prezentacji dla studentów, należy podkreślić następujące punkty:

1. Architektura ROS 2:

- **Węzeł (Node):** G1DemoNode jest pojedynczym programem w systemie ROS 2. Pełni funkcję mózgu, koordynując akcje.
- **Temat (Topic):** cmd_vel_publisher wysyła ciągłe polecenia prędkości. Służy do sterowania dynamicznym, nieprecyzyjnym ruchem (np. "idź do przodu").
- **Akcja (Action):** arm_action_client i base_action_client służą do wykonywania złożonych, docelowych zadań (np. "zrób gest machania", "przejdź do pozycji stojącej"). Akcje są lepsze niż tematy, gdy potrzebny jest feedback o postępie i możliwość przerwania zadania.

2. Standardowe interfejsy ROS 2:

- **geometry_msgs/Twist:** Standardowy komunikat do sterowania bazą mobilną. Studenci mogą używać go do sterowania dowolnym robotem mobilnym, który implementuje ten interfejs.
- **control_msgs/FollowJointTrajectory:** Standardowy interfejs do sterowania ramionami robotów i innymi manipulatrami. Umożliwia wysyłanie sekwencji punktów (trajektorii) do kontrolerów stawów.

3. Trajektoria vs. Prędkość (Walk vs. Arm Wave):

- **Chód (Walk):** Używamy Twist (prędkość) do prostego ruchu. Robot Unitree G1 ma wewnętrzny algorytm chodzenia, który interpretuje komendę Twist na skomplikowany ruch nóg.
- **Ramię (Arm Wave):** Używamy FollowJointTrajectory, aby precyzyjnie zdefiniować sekwencję pozycji stawów ramienia. To pokazuje, jak sterować precyzyjnie manipulatorem.

4. Bezpieczeństwo:

- Zawsze zaczynamy od pozycji bezpiecznej (sit) i przechodzimy do pozycji roboczej (stand) przed wykonaniem jakichkolwiek ruchów.
- Funkcja move_robot_base publikuje komendę Twist przez określony czas, a następnie publikuje zero, aby zatrzymać robota, co jest dobrą praktyką bezpieczeństwa.

Sugerowane ćwiczenia dla studentów:

1. **Modyfikacja trajektorii:** Zmodyfikuj funkcję perform_arm_wave tak, aby robot machał ramieniem w innym kierunku lub z inną prędkością.
2. **Chód w kwadrat:** Utwórz nową funkcję, która wykorzysta move_robot_base do przejścia po ścieżce kwadratu (np. idź prosto, obróć się o 90 stopni, powtórz 4 razy).

3. **Wyczuwanie otoczenia:** Dodaj do kodu subskrypcję na dane z czujników (np. LiDAR lub kamery) i stwórz warunek: jeśli w pobliżu robota wykryty zostanie obiekt, zatrzymaj się.