NAWIGACJA ROBOTOW HUMANOIDALNYCH

Laboratorium

Autonomiczna nawigacja robota

Unitree G1 EDU z Nav2

dr inż. Mateusz Pomianek

Politechnika Rzeszowska | Katedra Informatyki i Automatyki

Numer laboratorium	LAB-01
Tryb pracy	Symulacja Gazebo + opcjonalnie: prawdziwy robot G1
Wymagania wstępne	W1 - W4 z wykladow; podstawy ROS 2 Humble; Python 3.10+
Wersja dokumentu	v1.0 | ROS 2 Humble | Nav2 Humble

1. Informacje ogólne i zasady bezpieczeństwa

Celem zajęć jest praktyczne opanowanie pełnego stosu nawigacyjnego Nav2 dla robota humanoidalnego Unitree G1 EDU. Po ukończeniu laboratorium student będzie potrafił:

- uruchomić i konfigurować stos Nav2 wraz ze sterownikami robota G1;
- zbudować mapę zajętości metoda SLAM (slam_toolbox) na podstawie danych z LiDAR;
- ustawić i zoptymalizować parametry planisty globalnego i lokalnego;
- napisać klienta nawigacyjnego w Pythonie (nav2_simple_commander);
- analizować działanie cost mapy i modyfikować jej parametry.

1.1. Platforma Unitree G1 EDU - krótkie przypomnienie

Masa / wysokość	35 kg / 127 cm (pozycja stojaca)
Stopnie swobody	23 x DoF (12 nog + 7 reka x2 + szyja)
Sensory (podstawowe)	Livox Mid360 LiDAR, 2x kamera głębi RealSense D430, IMU 200 Hz
Komputer pokładowy	NVIDIA Orin NX 16 GB + Intel i7 (moduł percepcji)
System operacyjny	Ubuntu 22.04 + ROS 2 Humble
Komunikacja	Ethernet (LAN 192.168.123.x) + Wi-Fi 6
Czas pracy	~2 h (akumulator 10 Ah / 48 V)

1.2. Zasady bezpieczeństwa

ZASADY BEZPIECZENSTWA - PRZECZYTAJ PRZED URUCHOMIENIEM ROBOTA

1. Przed uruchomieniem robota sprawdź przestrzeń pracy: min. 3 x 3 m wolnej powierzchni bez kabli, przeszkód i osób postronnych.
2. Robot G1 waży 35 kg - upadek może spowodować poważne obrażenia. Nigdy nie stój bezpośrednio przed robotem.
3. Zachowaj dystans min. 1,5 m od robota podczas ruchu. Strefa bezpieczna oznaczona jest taśmą na podłodze stanowiska.
4. Przycisk E-STOP (czerwony, na pilocie i na obudowie) musi być trzymany w ręku prowadzącego przez cały czas pracy z fizycznym robotem.
5. Nigdy nie uruchamiaj trybu chodzenia bez nadzoru prowadzącego - wymagane jest wydanie wyraźnej zgody ustnej.
6. W przypadku niekontrolowanego ruchu lub błędu systemowego - natychmiast naciśnij E-STOP i cofnij się.
7. Kable Ethernet i zasilające układaj wzdłuż ściany; żadna osoba nie może przez nie przechodzić.
8. Nie kładź żadnych przedmiotów na robocie ani nie dotykaj aktuatorów podczas pracy.
9. W trybie SYMULACJI powyższe zasady nie obowiązują na stanowisku, ale są obowiązkowe na egzaminie praktycznym.
10. Wszelkie awarie, dziwne dźwięki lub nieoczekiwane ruchy zgłaszaj natychmiast prowadzącemu.

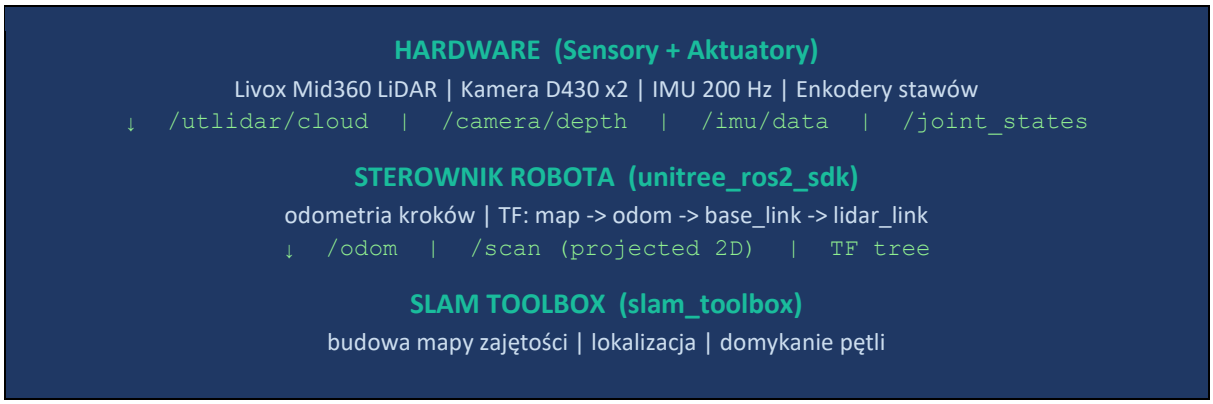
1.3. Wyposażenie stanowiska laboratoryjnego

Stacja robocza	PC z Ubuntu 22.04, ROS 2 Humble, Gazebo Fortress
Symulacja G1	unitree_ros2_sim + model URDF G1 z Livox Mid360
Stos nawigacyjny	Nav2 Humble (nav2_bringup, slam_toolbox, nav2_simple_commander)
Wizualizacja	RViz2 z gotowa konfiguracja *.rviz
Edytor kodu	VS Code + Python 3.10 + rozszerzenia ROS
Prawdziwy robot (opcja)	Unitree G1 EDU na stole + pilot E-STOP (tylko z prowadzącym)

2. Architektura systemu G1 + ROS 2 + Nav2

Zanim przystąpisz do ćwiczeń, zapoznaj się z pełną architektura stosu, którą będziemy uruchamiać. Każdy komponent odpowiada dokładnie jednemu modułowi omawianemu na wykładach W1-W4.

2.1. Schemat przepływu danych





2.2. Kluczowe topiki ROS 2

/utlidar/cloud	PointCloud2 z Livox Mid360 (150 Hz)
/scan	LaserScan 2D (projekcja chmury punktów) - wejście do SLAM i Nav2
/imu/data	sensor_msgs/Imu - orientacja + przyspieszenie (200 Hz)
/odom	nav_msgs/Odometry - odometria kroków (100 Hz)
/joint_states	sensor_msgs/JointState - kąty stawów (500 Hz)
/cmd_vel	geometry_msgs/Twist - komendy prędkości z Nav2 -> kontroler chodu
/map	nav_msgs/OccupancyGrid - mapa zajętości z SLAM
/goal_pose	geometry_msgs/PoseStamped - cel nawigacyjny z RViz2 lub klienta

2.3. Drzewo transformacji TF2

```
map
 \__ odom              (publikuje: SLAM toolbox - korekta dryfu)
      \__ base_link    (publikuje: unitree_ros2 - odometria kroków)
           |__ lidar_link   (stała transformacja kalibracyjna)
           |__ camera_link  (stała transformacja kalibracyjna)
           |__ imu_link     (stała transformacja kalibracyjna)
```

Najczęstszy błąd nawigacji w ROS 2

- Niespójne drzewo TF jest powodem >60% błędów startowych w Nav2.
- Zawsze sprawdź drzewo TF poleceniem: ros2 run tf2_tools view_frames
- Jeśli brakuje transformacji map->odom: SLAM nie jest uruchomiony lub nie ma jeszcze skanu.

3. Przygotowanie środowiska

3.1. Struktura workspace

Wszystkie pliki laboratorium są już zainstalowane w katalogu ~/ros2_ws. Sprawdź strukturę:

```
~/ros2_ws/
  src/
    unitree_ros2/          # sterownik robota G1
      unitree_g1_description/  # model URDF + mesh
      unitree_g1_sim/          # symulacja Gazebo
      unitree_g1_nav2/         # konfiguracja Nav2 dla G1
```

```
  install/                # po kompilacji
  build/
  log/
```

3.2. Weryfikacja instalacji

Wykonaj poniższe polecenia w terminalu i sprawdź, czy wszystkie pakiety są dostępne:

```
# 1. Załaduj środowisko ROS 2
source /opt/ros/humble/setup.bash
source ~/ros2_ws/install/setup.bash

# 2. Sprawdź kluczowe pakiety
ros2 pkg list | grep -E "(nav2|slam_toolbox|unitree)"

# 3. Sprawdź dostępność nav2_simple_commander
python3 -c "from nav2_simple_commander.robot_navigator import BasicNavigator;
print('OK')"

# 4. Sprawdź wersje Gazebo
ign gazebo --version
```

⚠ Jeśli pakiety nie sa widoczne

- Skompiluj workspace: cd ~/ros2_ws && colcon build --symlink-install
- Ponownie załaduj środowisko: source ~/ros2_ws/install/setup.bash
- W razie problemu zgłoś się do prowadzącego - nie traci czasu na samodzielna instalacje.

3.3. Aliasy dla wygody

Dodaj poniższe aliasy do ~/.bashrc (już dodane na stacjach laboratoryjnych):

```
# ~/.bashrc - juz skonfigurowane na stanowiskach:
alias humble='source /opt/ros/humble/setup.bash && source
~/ros2_ws/install/setup.bash'
alias g1sim='ros2 launch unitree_g1_sim g1_sim.launch.py'
alias g1nav='ros2 launch unitree_g1_nav2 g1_navigation.launch.py'
alias g1slam='ros2 launch unitree_g1_nav2 g1_slam.launch.py'
alias tf_tree='ros2 run tf2_tools view_frames && evince frames.pdf'
```

▶ Ćwiczenie 1: Uruchomienie i weryfikacja systemu

czas: **30 minut**

Cel: zapoznaj się ze struktura topików ROS 2 dla robota G1, wizualizuj dane sensoryczne w RViz2 i zweryfikuj poprawność drzewa transformacji TF.

Krok 1.1 - Uruchomienie symulacji Gazebo

Otwórz nowy terminal (Ctrl+Alt+T) i uruchom symulacje:

```
# Terminal 1: symulacja Gazebo z modelem G1
humble && g1sim

# Poczekaj ~30 s na pełne załadowanie Gazebo.
# Pojawi sie okno symulatora z robotem G1 stojącym w pustym pokoju.
# Dodatkowo automatycznie otworzy się RViz2 z wstępną konfiguracja.
```

Tryb prawdziwego robota (tylko z prowadzącym)

- Zamiast g1sim uruchom: ros2 launch unitree_g1_nav2 g1_driver.launch.py
- Adres IP robota: 192.168.123.161 (polacz się przez kabel Ethernet).
- Upewnij się, ze robot stoi stabilnie na stole przed wydaniem jakiekolwiek komendy.

Krok 1.2 - Weryfikacja topików

W nowym terminalu (Terminal 2) sprawdź, jakie topiki są aktywne:

```
# Terminal 2: weryfikacja topikow
humble

# Lista wszystkich topikow
ros2 topic list

# Topiki zwiazane z nawigacja
ros2 topic list | grep -E "(imu|odom|scan|lidar|joint|cmd_vel)"

# Czestotliwosc publikacji
ros2 topic hz /imu/data
ros2 topic hz /scan
ros2 topic hz /odom
```

Uzupelnij tabele wynikami pomiarow:

Topik	Oczekiwany Hz	Zmierzony Hz	Typ wiadomosci
/imu/data	~200		sensor_msgs/Imu
/odom	~100		nav_msgs/Odometry
/scan	~10		sensor_msgs/LaserScan
/joint_states	~500		sensor_msgs/JointState
/utlidar/cloud	~100		sensor_msgs/PointCloud2

Krok 1.3 - Wizualizacja danych w RViz2

W oknie RViz2 dodaj nastepujace wizualizacje (panel Displays po lewej -> przycisk Add):

1. Dodaj wyswietlacz RobotModel - ustaw description topic na /robot_description.
2. Dodaj wyswietlacz LaserScan - ustaw topic na /scan, rozmiar pointow: 0.03 m.
3. Dodaj wyswietlacz PointCloud2 - ustaw topic na /utlidar/cloud (oryginalna chmura 3D).
4. Dodaj wyswietlacz Odometry - ustaw topic na /odom; zaznacz "Show arrow".
5. Dodaj wyswietlacz TF - odznacz wszystkie ramki oprocz: map, odom, base_link, lidar_link.
6. Ustaw Fixed Frame na: odom (uzyj map po uruchomieniu SLAM).

✓ Co powinieneś zobaczyć

- Model 3D robota G1 stojącego w pozycji wyprostowanej.
- Zielony okrąg punktów LaserScan wokół robota (skan poziomy 2D z projektowanego LiDAR).
- Biale punkty chmury 3D z Livox Mid360.
- Strzałka odometrii przy podstawie robota.

Krok 1.4 - Sprawdzenie drzewa TF

```
# Wygeneruj i wyświetl drzewo TF
ros2 run tf2_tools view_frames
evince frames.pdf  # lub xdg-open frames.pdf
```

W wygenerowanym diagramie zaznacz i opisz poniższe relacje:

☐	Relacja map -> odom jest widoczna (publikuje: slam_toolbox lub static broadcaster)
☐	Relacja odom -> base_link jest widoczna (publikuje: unitree_ros2)
☐	Relacja base_link -> lidar_link jest widoczna (stala kalibracja)
☐	Każda transformacja ma aktualny timestamp (brak ostrzeżeń o starych danych)

Krok 1.5 - Skrypt monitorowania sensorów

Uruchom ponizszy skrypt Pythona w nowym pliku ~/lab01/monitor_sensors.py i sprawdź odczyty:

```
# ~/lab01/monitor_sensors.py
# Monitorowanie i walidacja danych sensorycznych robota G1

import rclpy
from rclpy.node import Node
from sensor_msgs.msg import Imu, LaserScan
from nav_msgs.msg import Odometry
import numpy as np

class SensorMonitor(Node):
    def __init__(self):
        super().__init__('sensor_monitor')
        self.imu_hz = self.scan_hz = self.odom_hz = 0
        self.last_imu_orientation = None

        self.create_subscription(Imu, '/imu/data', self.imu_cb, 10)
        self.create_subscription(LaserScan, '/scan', self.scan_cb, 10)
        self.create_subscription(Odometry, '/odom', self.odom_cb, 10)
        self.create_timer(2.0, self.report)
        self.get_logger().info('Monitor uruchomiony. Czekam na dane...')

    def imu_cb(self, msg: Imu):
        self.imu_hz += 1
        q = msg.orientation
        # Konwersja kwaternion -> kata yaw
        siny = 2.0 * (q.w * q.z + q.x * q.y)
        cosy = 1.0 - 2.0 * (q.y * q.y + q.z * q.z)
        yaw_deg = np.degrees(np.arctan2(siny, cosy))
        self.last_yaw = yaw_deg

    def scan_cb(self, msg: LaserScan):
        self.scan_hz += 1
        valid_ranges = [r for r in msg.ranges if np.isfinite(r) and r > 0]
```

```
        self.scan_min = min(valid_ranges) if valid_ranges else float('inf')

    def odom_cb(self, msg: Odometry):
        self.odom_hz += 1
        self.pos_x = msg.pose.pose.position.x
        self.pos_y = msg.pose.pose.position.y

    def report(self):
        yaw = getattr(self, 'last_yaw', 0.0)
        scan_min = getattr(self, 'scan_min', 0.0)
        pos_x = getattr(self, 'pos_x', 0.0)
        pos_y = getattr(self, 'pos_y', 0.0)
        self.get_logger().info(
            f'[IMU] yaw={yaw:.1f} deg  '
            f'[SCAN] min_dist={scan_min:.2f} m  '
            f'[ODOM] pos=({pos_x:.3f}, {pos_y:.3f}) m  '
            f'[Hz] imu={self.imu_hz//2}, scan={self.scan_hz//2},
odom={self.odom_hz//2}'
        )
        self.imu_hz = self.scan_hz = self.odom_hz = 0

def main():
    rclpy.init()
    rclpy.spin(SensorMonitor())
    rclpy.shutdown()

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

```
# Uruchomienie skryptu
cd ~/lab01 && python3 monitor_sensors.py
```

Zadanie 1.5: Zmodyfikuj skrypt tak, aby zgłosił ostrzeżenie (WARN) gdy minimalna odległość ze skanu laserowego jest mniejsza niż 0,5 m.

Wskazówka: uzyj self.get_logger().warn(...).

▶ Ćwiczenie 2: Budowa mapy środowiska metoda SLAM

czas: **55 minut**

Cel: zbuduj mapę zajętości 2D metoda SLAM toolbox prowadzać robota przez pomieszczenie. Zapisz mapę i przeanalizuj jej jakość.

Krok 2.1 - Konfiguracja SLAM toolbox

Przejrzyj plik konfiguracyjny SLAM toolbox przygotowany dla G1:

```
cat ~/ros2_ws/src/unitree_g1_nav2/config/slam_toolbox_g1.yaml
```

```
# slam_toolbox_g1.yaml - konfiguracja dla G1 EDU z Livox Mid360
slam_toolbox:
  ros__parameters:
```

```
    # Ramki TF
    odom_frame: odom
    map_frame: map
    base_frame: base_link
    scan_topic: /scan

    # Tryb: mapping (budowanie) lub localization (tylko lokalizacja)
    mode: mapping

    # Parametry mapy
    resolution: 0.05          # 5 cm/komórka - kompromis szybkość/dokładność
    max_laser_range: 12.0     # max zasięg LiDAR [m]
    minimum_travel_distance: 0.5   # min dystans przed nowym skanem
    minimum_travel_heading: 0.5    # min obrót przed nowym skanem [rad]

    # Domykanie petli (loop closure)
    do_loop_closing: true
    loop_search_maximum_distance: 3.0
    loop_match_minimum_chain_size: 10
```

Kluczowe parametry do zapamiętania

- resolution: 0.05 = komórka mapy 5x5 cm. Większe -> szybsza mapa, mniejsza dokładność.
- do_loop_closing: true - SLAM koryguje skumulowany dryft przy powrocie do odwiedzonego miejsca.
- minimum_travel_distance: 0.5 - SLAM nie dodaje nowego skanu jeśli robot stal w miejscu.

Krok 2.2 - Uruchomienie SLAM

```
# Terminal 3: uruchomienie SLAM (symulacja musi juz dzialac)
humble && g1slam

# Co zostanie uruchomione:
# - slam_toolbox w trybie mapping
# - node projekcji 3D -> 2D (pointcloud_to_laserscan)
# - static TF: base_link -> lidar_link
```

Przełącz RViz2 na Fixed Frame: map i dodaj wyświetlacz Map (topic: /map, Color Scheme: map).
Po kilku sekundach powinna pojawić się budująca się mapa.

Krok 2.3. Teleoperate - eksploracja pomieszczenia

W nowym terminalu uruchom sterowanie klawiatura i przeprowadź robota po całym pomieszczeniu:

```
# Terminal 4: sterowanie klawiatura
humble
ros2 run teleop_twist_keyboard teleop_twist_keyboard --ros-args --remap
cmd_vel:=/cmd_vel

# Klawisze:
#   i / , - do przodu / do tylu
#   j / l - skręt w lewo / w prawo
#   k     - zatrzymanie
#   q / z - zwiększ / zmniejsz prędkość (max: v=0.5 m/s, omega=0.8 rad/s)
```

⚠Zasady bezpiecznej eksploracji w symulacji

- Maksymalna prędkość w symulacji: 0.5 m/s. Wyższa może powodować niestabilność modelu.
- Jeśli robot "wchodzi" w ściany: prędkość jest zbyt duża lub mapa nie zdążyła się zaktualizować.
- Zatrzymaj się (k) gdy zmieniasz kierunek ruchu.

Plan eksploracji - przejdź co najmniej:

7. Cały obwód pomieszczenia (trasa zamknięta - kluczowe dla domykania pętli!).
8. Środek pomieszczenia z obu osi.
9. Powrót do pozycji startowej (zamkniecie pętli).

✓ Jak ocenić jakość mapy na żywo

- Ściany musza być ciągłymi liniami - brak "przerw" oznacza dobra pokrycie skanem.
- Mapa powinna wyglądać jak plan pomieszczenia - widoczne ściany, wolna przestrzeń biała.
- Jeśli widzisz "duchy" przeszkód w pustej przestrzeni: prędkość jest zbyt duża.

Krok 2.4 - Zapis mapy

```
# Terminal 5: zapis gotowej mapy
mkdir -p ~/lab_maps
ros2 run nav2_map_server map_saver_cli -f ~/lab_maps/pomieszczenie_g1 \
  --ros-args -p save_map_timeout:=10.0

# Zostaną zapisane dwa pliki:
#   ~/lab_maps/pomieszczenie_g1.yaml  - metadane mapy
#   ~/lab_maps/pomieszczenie_g1.pgm   - obraz mapy (wolne=białe, zajęte=czarne)
```

```
# Sprawdź metadane zapisanej mapy
cat ~/lab_maps/pomieszczenie_g1.yaml

# Podglądnij obraz mapy
eog ~/lab_maps/pomieszczenie_g1.pgm  # lub: gimp, feh
```

Krok 2.5 - Analiza jakości mapy

Uzupełnij tabele obserwacji:

Kryterium oceny	Wpisz swoją obserwację
Czy ściany są ciągle (brak luk)?	
Czy wolna przestrzeń jest biała (brak szarych obszarów)?	
Czy widoczne są artefakty (duchy przeszkód)?	
Czy mapa jest spójnie zamknięta po powrocie do startu?	
Ile skanów SLAM przetworzyło? (ros2 topic echo /slam_toolbox/scan_count)	

Zadanie 2.5: Zbuduj dwie mapy tej samej sceny: jedna z resolution: 0.05 m i jedna z resolution: 0.10 m. Porównaj rozmiary plikow .pgm i widoczne różnice w jakości. Zapisz nazwy obu map i wyniki porównania w raporcie.

▶ Ćwiczenie 3: Autonomiczna nawigacja z Nav2

czas: **60 minut**

Cel: uruchom pełny stos Nav2 z zapisana mapa i wyślij cele nawigacyjne z RViz2. Obserwuj działanie planisty globalnego (A*) i lokalnego (DWB). Porównaj trasy z rożnymi parametrami cost mapy.

Krok 3.1 - Uruchomienie Nav2 z mapa

Zatrzymaj SLAM (Ctrl+C w Terminalu 3). Uruchom Nav2 z zapisana mapa:

```
# Terminal 3 (po zatrzymaniu SLAM): uruchomienie Nav2
humble
ros2 launch unitree_g1_nav2 g1_navigation.launch.py \
  map:=$HOME/lab_maps/pomieszczenie_g1.yaml \
  use_sim_time:=True

# Zostana uruchomione:
# - nav2_map_server    (serwer mapy)
# - nav2_amcl          (lokalizacja Monte Carlo)
# - nav2_planner       (planista globalny: NavFn/A*)
# - nav2_controller    (planista lokalny: DWB)
# - nav2_bt_navigator (koordynator misji)
# - nav2_lifecycle_manager (zarzadzanie cyklem zycia)
```

Dlaczego Nav2 używa lifecycle manager?

- Każdy node Nav2 ma stany: unconfigured -> inactive -> active -> deactivating.
- Lifecycle manager uruchamia je w odpowiedniej kolejności i można je uruchamiać/zatrzymywać bez zamykania procesu.
- Jeśli jakiś node nie aktywuje sie: ros2 lifecycle get /nav2_amcl pokazuje aktualny stan.

Krok 3.2 - Inicjalizacja pozycji startowej (AMCL)

AMCL (Adaptive Monte Carlo Localization) musi wiedzieć, gdzie robot startuje na mapie. W RViz2:

10. W górnym pasku RViz2 kliknij przycisk "2D Pose Estimate" (strzałka).
11. Kliknij na mapie w przybliżeniu startowej pozycji robota i przeciągnij w kierunku jego orientacji.
12. W panelu Displays dodaj wyświetlacz PoseArray z topikiem /particlecloud - zobaczysz chmurę hipotez AMCL.
13. Porusz robotem teleopem (~0.5 m naprzód i powrót) - obserwuj jak hipotezy skupiają się wokół prawdziwej pozycji.

Jak rozpoznać dobra inicjalizacje AMCL

- Chmura cząsteczek (/particlecloud) powinna skupić się w jednym, małym obszarze.
- Jeżeli chmura jest rozproszona po całej mapie: podaj lepsze przybliżenie pozycji startowej.
- Można też użyć: ros2 topic pub --once /initialpose geometry_msgs/PoseWithCovarianceStamped "..."

Krok 3.3 - Wysłanie celu nawigacyjnego z RViz2

14. W górnym pasku RViz2 kliknij "2D Nav Goal" (strzałka z celem).
15. Kliknij na mapie w dowolnym miejscu wolnej przestrzeni i przeciągnij w kierunku docelowej orientacji.
16. Obserwuj planowanie trasy: w RViz2 pojawi się niebieska linia planu globalnego.
17. Obserwuj ruch lokalnego planisty: zielona ścieżka w bliskim zasięgu robota.
18. Czekaj, aż robot dotrze do celu lub zanotuj błędy w konsoli.

Dodaj do RViz2 następujące wyświetlacze do obserwacji planistów:

- Path (topic: /plan) - plan globalny (kolor: niebieski, grubość: 0.03)
- Path (topic: /local_plan) - plan lokalny DWB (kolor: zielony, grubosc: 0.02)
- Map (topic: /global_costmap/costmap) - globalna mapa kosztów
- Map (topic: /local_costmap/costmap) - lokalna mapa kosztów

Krok 3.4 - Analiza cost mapy i parametrów

Przejrzyj konfiguracje cost mapy dla G1:

```
cat ~/ros2_ws/src/unitree_g1_nav2/config/nav2_params_g1.yaml | head -80
```

```
# Fragment nav2_params_g1.yaml
local_costmap:
  local_costmap:
    ros__parameters:
      robot_radius: 0.30        # KLUCZOWE: min. promień od przeszkody [m]
      inflation_radius: 0.55    # strefa narastania kosztu
      cost_scaling_factor: 3.0  # szybkość narastania kosztu do centrum
przeszkody

global_costmap:
  global_costmap:
    ros__parameters:
      robot_radius: 0.30
      inflation_radius: 0.55
```

Eksperyment: zmień parametry i obserwuj efekt.

Parametr	Wartość domyślna	Obserwacja efektu
inflation_radius	0.55 m	
cost_scaling_factor	3.0	
robot_radius	0.30 m	

Aby zmienić parametr w czasie rzeczywistym bez restartu:

```
# Zmiana inflation_radius w locie (tylko local_costmap)
ros2 param set /local_costmap/local_costmap inflation_layer.inflation_radius 0.8

# Sprawdz aktualny stan parametrow planisty
ros2 param list /global_costmap/global_costmap
ros2 param get /local_costmap/local_costmap inflation_layer.inflation_radius
```

Krok 3.5 - Analiza logów nawigacji

```
# Obejrzyj log z jednej misji nawigacyjnej (zatrzymaj po dotarciu do celu)
ros2 topic echo /navigate_to_pose/_action/feedback | head -40


# Informacje o aktualnym stanie planisty
ros2 service call /nav2_controller/get_velocity_commands
nav2_msgs/srv/GetVelocityCommand
```

Zadanie 3.5: Wyslij ten sam cel nawigacyjny dwa razy: raz z inflation_radius = 0.3 m, raz z 0.8 m. Czy trasy sa identyczne? Która jest bezpieczniejsza i dlaczego? Opisz w raporcie.

▶ Cwiczenie 4: Klient nawigacyjny w Pythonie

czas: 50 minut

Cel: napisz klienta nawigacyjnego korzystającego z nav2_simple_commander. Zaimplementuj nawigacje sekwencyjna przez listę punktów kontrolnych i monitoruj postęp misji.

Krok 4.1 - Podstawy nav2_simple_commander

Biblioteka nav2_simple_commander to wysokopoziomowe API Nav2 dla Pythona. Główna klasa BasicNavigator pozwala na:

- goToPose(goal) - nawigacja do pojedynczego punktu,
- followWaypoints(poses) - sekwencyjna nawigacja przez liste punktow,
- isTaskComplete() - sprawdzenie czy misja jest skonczona,
- getFeedback() - informacje zwrotne (odleglosc, czas),
- cancelTask() - przerwanie biezacej misji,
- waitUntilNav2Active() - czekanie az Nav2 bedzie gotowe.

Krok 4.2 - Klient podstawowy: nawigacja do jednego punktu

Utwórz plik ~/lab01/nav_to_goal.py:

```
# ~/lab01/nav_to_goal.py
# Prosty klient nawigacyjny Nav2 - nawigacja do jednego punktu

import rclpy
import math
import time
from geometry_msgs.msg import PoseStamped
from nav2_simple_commander.robot_navigator import BasicNavigator, TaskResult


def make_pose(navigator, x: float, y: float, yaw_deg: float = 0.0) ->
PoseStamped:
    """Tworzy wiadomosc PoseStamped z pozycja i orientacja."""
    pose = PoseStamped()
    pose.header.frame_id = 'map'
    pose.header.stamp = navigator.get_clock().now().to_msg()
    pose.pose.position.x = x
    pose.pose.position.y = y
    # Przeliczenie kata yaw na kwaternion (obrot wokol osi Z)
    yaw_rad = math.radians(yaw_deg)
```

```
    pose.pose.orientation.z = math.sin(yaw_rad / 2.0)
    pose.pose.orientation.w = math.cos(yaw_rad / 2.0)
    return pose


def navigate_to(navigator, x: float, y: float, yaw_deg: float = 0.0) -> bool:
    """Nawiguje do podanego punktu i czeka na wynik."""
    goal = make_pose(navigator, x, y, yaw_deg)
    navigator.goToPose(goal)

    start_time = time.time()
    while not navigator.isTaskComplete():
        feedback = navigator.getFeedback()
        if feedback:
            dist = feedback.distance_remaining
            elapsed = time.time() - start_time
            print(f'  Odleglosc do celu: {dist:.2f} m  |  Czas: {elapsed:.1f}
s')
        time.sleep(0.5)

    result = navigator.getResult()
    if result == TaskResult.SUCCEEDED:
        print(f'[OK] Cel ({x:.2f}, {y:.2f}) osiagniety w {time.time()-
start_time:.1f} s')
        return True
    else:
        print(f'[BLAD] Nawigacja nieudana: {result}')
        return False


def main():
    rclpy.init()
    navigator = BasicNavigator('nav_client')

    print('Czekam az Nav2 bedzie aktywne...')
    navigator.waitUntilNav2Active()
    print('Nav2 gotowe!')

    # Nawigacja do jednego punktu (dopasuj do swojej mapy!)
    success = navigate_to(navigator, x=1.5, y=0.5, yaw_deg=90.0)

    if success:
        print('Misja zakonczona pomyslnie.')
    else:
        print('Misja nieudana.')

    rclpy.shutdown()


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

```
# Uruchomienie (Nav2 musi byc aktywne!)

cd ~/lab01 && python3 nav_to_goal.py
```

Krok 4.3 - Nawigacja przez listę punktów kontrolnych

Utwórz plik ~/lab01/waypoint_mission.py - pełna misja przez wiele punktów:

```
# ~/lab01/waypoint_mission.py
# Misja nawigacyjna: sekwencyjna nawigacja przez listę punktów + raport wyników
```

```
import rclpy
import math, time, datetime
from geometry_msgs.msg import PoseStamped
from nav2_simple_commander.robot_navigator import BasicNavigator, TaskResult


WAYPOINTS = [
    # (x [m], y [m], yaw [deg], nazwa)
    (1.0,  0.0,   0.0,  'Punkt A'),
    (1.0,  1.5,  90.0,  'Punkt B'),
    (0.0,  1.5, 180.0,  'Punkt C'),
    (0.0,  0.0, -90.0,  'Powrot do bazy'),
]


def make_pose(nav, x, y, yaw_deg):
    pose = PoseStamped()
    pose.header.frame_id = 'map'
    pose.header.stamp = nav.get_clock().now().to_msg()
    pose.pose.position.x = x
    pose.pose.position.y = y
    yaw = math.radians(yaw_deg)
    pose.pose.orientation.z = math.sin(yaw / 2.0)
    pose.pose.orientation.w = math.cos(yaw / 2.0)
    return pose


def run_mission(navigator):
    results = []
    total_start = time.time()

    for idx, (x, y, yaw, name) in enumerate(WAYPOINTS):
        print(f'\n>>> [{idx+1}/{len(WAYPOINTS)}] Nawigacja: {name} -> ({x}, {y},
{yaw} deg)')
        goal = make_pose(navigator, x, y, yaw)
        navigator.goToPose(goal)

        t_start = time.time()
        while not navigator.isTaskComplete():
            fb = navigator.getFeedback()
            if fb:
                print(f'  Dystans: {fb.distance_remaining:.2f} m', end='\r')
            time.sleep(0.3)

        elapsed = time.time() - t_start
        result = navigator.getResult()
        ok = result == TaskResult.SUCCEEDED
        status = 'OK' if ok else f'BLAD ({result})'
        print(f'  [{status}] Czas: {elapsed:.1f} s')
        results.append({
            'name': name, 'x': x, 'y': y,
            'success': ok, 'time': elapsed
        })

        if not ok:
            print('[!] Misja przerwana z powodu bledu.')
            break

    # --- Raport koncowy ---
    total = time.time() - total_start
    print('\n' + '='*50)
    print('RAPORT MISJI')
    print('='*50)
    for r in results:
        sym = '[OK]  ' if r['success'] else '[FAIL]'
        print(f"{sym} {r['name']:20s}  {r['time']:.1f} s")
    success_count = sum(r['success'] for r in results)
```

```
    print(f'Laczny czas: {total:.1f} s  |  Sukces:
{success_count}/{len(results)}')
    print('='*50)
    return results


def main():
    rclpy.init()
    nav = BasicNavigator('waypoint_client')
    print('Czekam na Nav2...')
    nav.waitUntilNav2Active()
    print('Nav2 gotowe. Startuje misje.')

    run_mission(nav)
    rclpy.shutdown()


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

⚠Ważne: dopasuj współrzędne do swojej mapy!

- Współrzędne na liście WAYPOINTS musza leżeć w wolnej przestrzeni Twojej mapy.
- Sprawdź współrzędne klikając PPM w RViz2 -> "Publish Point" i odczytując z /clicked_point.
- Jeśli robot nie może znaleźć ścieżki: punkt jest w kolizji lub poza zasięgiem mapy.

```
# Jak odczytać współrzędne z mapy w RViz2
# 1. W RViz2: kliknij przycisk "Publish Point" (lub: Panels->Tool Properties->Publish Point)
# 2. Kliknij na mapę w punkcie docelowym
# 3. Odczytaj współrzędne:
ros2 topic echo /clicked_point --once
```

Krok 4.4 - Rozszerzenie: monitorowanie stabilności ZMP

Dodaj do klienta nawigacyjnego monitoring marginesu ZMP podczas ruchu:

```
# ~/lab01/nav_with_stability.py (fragment - dodaj do waypoint_mission.py)

import threading
from sensor_msgs.msg import Imu
from nav_msgs.msg import Odometry
import numpy as np

class StabilityMonitor(threading.Thread):
    """Monitoruje kat pochylenia tułowia jako proxy stabilnosci ZMP."""

    def __init__(self, node):
        super().__init__(daemon=True)
        self.node = node
        self.max_pitch = 0.0
        self.max_roll  = 0.0
        self._sub = node.create_subscription(
            Imu, '/imu/data', self._imu_cb, 10)

    def _imu_cb(self, msg: Imu):
        q = msg.orientation
        # Roll z kwaterniona
        sinr = 2.0 * (q.w * q.x + q.y * q.z)
        cosr = 1.0 - 2.0 * (q.x**2 + q.y**2)
```

```
        roll  = abs(np.degrees(np.arctan2(sinr, cosr)))
        # Pitch z kwaterniona
        sinp  = 2.0 * (q.w * q.y - q.z * q.x)
        pitch = abs(np.degrees(np.arcsin(np.clip(sinp, -1, 1))))
        self.max_roll  = max(self.max_roll,  roll)
        self.max_pitch = max(self.max_pitch, pitch)
        if roll > 10.0 or pitch > 12.0:
            self.node.get_logger().warn(
                f'[STABILNOSC] Duze odchylenie! roll={roll:.1f}
pitch={pitch:.1f} deg')

    def report(self) -> str:
        return (f'max_roll={self.max_roll:.1f} deg, '
                f'max_pitch={self.max_pitch:.1f} deg')

    def reset(self):
        self.max_roll = self.max_pitch = 0.0
```

Zadanie 4.4: Zintegruj StabilityMonitor z waypoint_mission.py tak, aby po każdym punkcie wyświetlał raport maksymalnych odchyleń. Które punkty trasy powodują większe odchylenia i dlaczego?

▶ Ćwiczenie 5: Parametry planisty i adaptacja cost mapy

czas: **30 minut**

Cel: zbadaj wpływ kluczowych parametrów DWB (planista lokalny) i inflacji cost mapy na zachowanie robota. Porównaj strategie planowania w wąskim korytarzu.

Krok 5.1 - Parametry DWB Controller

Planista lokalny DWB (Dynamic Window-Based) ocenia kandydujące trajektorie według funkcji:

$$G(v,\omega) = \alpha \times heading(v,\omega) + \beta \times clearance(v,\omega) + \gamma \times velocity(v,\omega)$$

heading (alpha)	Wyrównanie kierunku ruchu z celem globalnym. Wysokie alpha -> robot celuje prosto.
clearance (beta)	Odległość od najbliższej przeszkody wzdłuż trajektorii. Wysokie beta -> robot trzyma się dalej.
velocity (gamma)	Nagroda za wyższa prędkość liniowa. Wysokie gamma -> robot jedzie szybciej.

Przejrzyj aktualne wartości parametrów:

```
# Sprawdź parametry krytyku DWB
ros2 param list /controller_server | grep -E "(scale|weight)"

# Odczytaj konkretny parametr
ros2 param get /controller_server FollowPath.GoalAlignCritic.scale
```

```
# Fragment konfiguracji DWB w nav2_params_g1.yaml
controller_server:
  ros__parameters:
    controller_frequency: 10.0    # Hz planisty lokalnego
    min_x_velocity_threshold: 0.001
    min_y_velocity_threshold: 0.5
    min_theta_velocity_threshold: 0.001
    FollowPath:
      plugin: "dwb_core::DWBLocalPlanner"
      max_vel_x:   0.5    # max predkosc do przodu [m/s]
      max_vel_theta: 0.8  # max predkosc katowa [rad/s]
      min_vel_x:  -0.2    # cofanie dozwolone
      acc_lim_x:   2.5    # max przyspieszenie liniowe
      acc_lim_theta: 3.2  # max przyspieszenie kątowe
      critics: ["RotateToGoal", "Oscillation", "BaseObstacle",
                "GoalAlign", "PathAlign", "PathDist", "GoalDist"]
      GoalDist.scale:   24.0    # waga krytyku odleglosci do celu
      PathDist.scale:   32.0    # waga krytyku odleglosci od globalnej sciezki
      GoalAlign.scale:  24.0    # waga wyrownania do celu
      PathAlign.scale:  32.0    # waga wyrownania do globalnej sciezki
```

Krok 5.2 - Eksperyment: bezpieczeństwo vs. szybkość

Wykonaj 3 nawigacje tym samym celem, zmieniajac konfiguracje po kazdej:

#	Konfiguracja	Zmienione parametry	Czas trasy [s]	Obserwacje
1	Domyslna	`inflation=0.55, max_vel_x=0.5`	...	
2	Bezpieczna	`inflation=0.8, max_vel_x=0.3`	...	
3	Szybka	`inflation=0.3, max_vel_x=1.0`	...	

```
# Zmiana parametrów DWB w locie
ros2 param set /controller_server FollowPath.max_vel_x 0.3
ros2 param set /local_costmap/local_costmap inflation_layer.inflation_radius 0.8

# WAZNE: po zmianie inflation_radius wymagany restart costmap lub
# wymuszenie ponownego obliczenia:
ros2 service call /local_costmap/clear_entirely_local_costmap \
  nav2_msgs/srv/ClearEntireCostmap {}
```

Zadanie 5.2: Na podstawie obserwacji odpowiedz na pytanie: Jaka kombinacja parametrów byłaby optymalna dla robota humanoida poruszającego się wśród pieszych (wg norm z Wykładu 7)? Uzasadnij dobór wartości odwołując się do stref proksemicznych Halla.

6. Zadanie zaliczeniowe

Misja: Nawigacja przez 4 punkty kontrolne z raportem metryk

Czas realizacji: 20-25 minut | Ocena: grupowa (1 raport na stanowisko)

6.1. Tresc zadania

Napisz program misji nawigacyjnej (rozszerzenie waypoint_mission.py z Ćwiczenia 4), który realizuje następujące wymagania:

1. Robot nawiguje przez dokładnie 4 punkty kontrolne rozmieszczone w czterech narożnikach pomieszczenia, tworząc zamknięty czworobok.

2. Przed wysłaniem każdego celu program wyświetla jego numer, współrzędne i szacowany dystans od bieżącej pozycji.
3. Podczas nawigacji program mierzy i wyświetla odległość do celu co 0.5 sekundy.
4. Po każdym punkcie program zapisuje do listy: czas trwania, sukces/porażka, maksymalny kat pochylenia z StabilityMonitor.
5. Po zakończeniu całej misji program generuje raport z kluczowymi metrykami (patrz niżej).

6.2. Wymagany raport programu

Program musi wyświetlić (w konsoli) poniższy raport po zakończeniu misji:

```
======================== RAPORT MISJI ============================
Data:          2024-10-15 14:32:01
Mapa:          pomieszczenie_g1.yaml
Liczba punktow: 4
------------------------------------------------------------------
#  | Nazwa         | Sukces |  Czas[s] | Max pitch[deg]
1  | Punkt A       |  TAK   |    12.3  |     3.2
2  | Punkt B       |  TAK   |    18.7  |     5.1
3  | Punkt C       |  TAK   |    14.2  |     4.0
4  | Baza          |  TAK   |    11.9  |     3.8
------------------------------------------------------------------
Laczny czas misji:  57.1 s
Sukces misji:       4/4  (100.0%)
Max pitch w misji:  5.1 deg
Sredni czas/punkt:  14.3 s
==================================================================
```

6.3. Kryteria oceny

Program uruchamia się bez błędów (0/5)	0 lub 5 pkt
Robot dotrze do wszystkich 4 punktów (0/15)	do 15 pkt (po 5/punkt)
Raport generowany poprawnie z wymaganymi metrykami (0/5)	0 lub 5 pkt
Stabilność: max_pitch < 8 deg na każdym segmencie (0/5)	0 lub 5 pkt
Czytelność kodu: komentarze, nazwy zmiennych po angielsku (0/5)	0 do 5 pkt
Odpowiedzi ustne na pytania kontrolne (0/10)	0 do 10 pkt

> **i Wskazówki do zadania zaliczeniowego**
>
> - Bazuj bezpośrednio na kodzie z Ćwiczeń 4 i 5 - nie zaczynam od zera.
> - Współrzędne 4 punktów kontrolnych zapisz jako listę WAYPOINTS na samym początku pliku.
> - Użyj klasy StabilityMonitor z Ćwiczenia 4 jako wątkowej subskrypcji IMU.
> - Pamiętaj: nav.waitUntilNav2Active() musi być wywołane PRZED pierwszym goToPose().

7. Pytania kontrolne

Przygotuj się ustnie na poniższe pytania. Będziemy je zadawać losowo podczas obrony zadania zaliczeniowego.

Pytania z zakresu architektury systemu:

1. Opisz role każdego z pięciu głównych komponentów: unitree_ros2, slam_toolbox, nav2_planner, nav2_controller, nav2_bt_navigator.

2. Dlaczego ramka odom jest "ciągła, lecz dryfująca", a ramka map jest "nieciągła, ale globalna"? Co to oznacza w praktyce?
3. Co to jest domykanie pętli (loop closure) w SLAM toolbox i jak wpływa na jakość mapy?
4. Jaka jest różnica miedzy global_costmap a local_costmap? Która jest ważniejsza dla bezpieczeństwa?

Pytania z zakresu algorytmów:

5. Algorytm A* wyznacza najkrótszą ścieżkę geometrycznie. Dlaczego trasa Na wizyta przez Nav2 może być dłuższa od najkrótszej? Co to oznacza z perspektywy bezpieczeństwa humanoida?
6. Opisz funkcje oceny trajektorii DWB. Jakie wartości parametrów alpha i beta zaproponowałbyś dla humanoida poruszającego się w szpitalu (wg norm z Wykładu 7)?
7. Co to jest inflation_radius i jak wpływa na planowanie? Co się stanie gdy ustawisz go na 0.0 m?
8. Jak AMCL (Adaptive Monte Carlo Localization) inicjalizuje się po włączeniu robota? Co jest powodem rozproszenia chmury punktów /particlecloud?

Pytania praktyczne (z laboratorium):

9. Uruchom ros2 topic hz /imu/data i wyjaśnij, dlaczego wartości nie są dokładnie równe 200 Hz.
10. Opisz, co zaobserwowałeś porównując ścieżki dla inflation_radius = 0.3 i 0.8. Które rozwiązanie wybrałbyś w biurze z wąskim korytarzem 1.2 m?
11. Co to jest typ wiadomości TaskResult.SUCCEEDED i jakie inne wartości mogą się pojawić? Kiedy może wystąpić TaskResult.FAILED?

Dodatek A: Pełna konfiguracja Nav2 dla G1 EDU

Pełny plik konfiguracyjny Nav2 dostosowany do robota Unitree G1 EDU:

```
# ~/ros2 ws/src/unitree g1 nav2/config/nav2 params g1.yaml

bt navigator:
  ros  parameters:
    use sim time: True
    global frame: map
    robot base frame: base link
    odom topic: /odom
    bt loop duration: 10
    default server timeout: 20

controller server:
  ros  parameters:
    use sim time: True
    controller frequency: 10.0
    controller plugins: ["FollowPath"]
    FollowPath:
      plugin: "dwb core::DWBLocalPlanner"
      debug trajectory details: True
      min vel x: -0.2      # cofanie
      max vel x:  0.5      # max predkosc do przodu
      max vel theta: 0.8   # max predkosc obrotowa
      acc lim x: 2.5
      acc lim theta: 3.2
      sim time: 1.5        # horyzont predykcji DWA [s]
      vx samples: 20       # liczba prob predkosci liniowej
      vtheta samples: 40   # liczba prob predkosci katowej
      critics: ["RotateToGoal", "Oscillation", "BaseObstacle",
                "GoalAlign", "PathAlign", "PathDist", "GoalDist"]
      GoalDist.scale: 24.0
```

```
      PathDist.scale: 32.0

planner server:
  ros  parameters:
    expected planner frequency: 20.0
    use sim time: True
    planner plugins: ["GridBased"]
    GridBased:
      plugin: "nav2 navfn planner/NavfnPlanner"
      tolerance: 0.5
      use astar: True       # True = A*, False = Dijkstra
      allow unknown: True   # dozwolone wejście w nieznany obszar

local costmap:
  local costmap:
    ros__parameters:
      update frequency: 5.0
      publish frequency: 2.0
      global frame: odom
      robot base frame: base link
      use sim time: True
      rolling window: true
      width: 3
      height: 3
      resolution: 0.05
      robot radius: 0.30   # promien robota G1 [m]
      plugins: ["obstacle layer", "inflation layer"]
      obstacle layer:
        plugin: "nav2 costmap 2d::ObstacleLayer"
        enabled: True
        observation sources: scan
        scan:
          topic: /scan
          max obstacle height: 2.0
          clearing: True
          marking: True
          data type: 'LaserScan'
          raytrace_max_range: 3.0
          obstacle max range: 2.5
      inflation layer:
        plugin: "nav2_costmap_2d::InflationLayer"
        cost scaling factor: 3.0
        inflation radius: 0.55

global costmap:
  global costmap:
    ros__parameters:
      update frequency: 1.0
      publish frequency: 1.0
      global_frame: map
      robot base frame: base link
      use sim time: True
      robot radius: 0.30
      resolution: 0.05
      track unknown space: true
      plugins: ["static layer", "obstacle layer", "inflation layer"]
      static layer:
        plugin: "nav2 costmap 2d::StaticLayer"
        map subscribe transient local: True
      obstacle layer:
        plugin: "nav2 costmap 2d::ObstacleLayer"
        enabled: True
        observation sources: scan
        scan:
          topic: /scan
          max obstacle height: 2.0
          clearing: True
          marking: True
          data type: 'LaserScan'
      inflation layer:
        plugin: "nav2 costmap 2d::InflationLayer"
        cost scaling factor: 3.0
```

```
        inflation radius: 0.55
```

Dodatek B: Ściąga z kluczowych komend ROS 2 i Nav2

Uruchamianie systemu

```
# Pełne uruchomienie laboratorium (ważna jest kolejność!)
humble && g1sim                          # 1. Symulacja
humble && g1slam                         # 2. SLAM
humble && g1nav map:=~/lab_maps/X.yaml   # 3. Nav2 (zamiast SLAM)
```

Diagnostyka i debugowanie

```
# TF tree
ros2 run tf2_tools view_frames && evince frames.pdf
ros2 run tf2_ros tf2_echo map base_link     # transformacja na zywo

# Topiki
ros2 topic list   ros2 topic hz /scan
ros2 topic echo /amcl_pose --once           # pozycja AMCL
ros2 topic echo /clicked_point --once       # współrzędne z RViz2

# Nav2 status
ros2 action list
ros2 lifecycle get /nav2_controller
ros2 service list | grep nav2

# Parametry
ros2 param list /controller_server
ros2 param get  /controller_server FollowPath.max_vel_x
ros2 param set  /controller_server FollowPath.max_vel_x 0.3
```

Mapa i SLAM

```
# Zapis mapy
ros2 run nav2_map_server map_saver_cli -f ~/lab_maps/NAZWA

# Wczytanie mapy (bez Nav2)
ros2 run nav2_map_server map_server --ros-args -p
yaml_filename:=~/lab_maps/NAZWA.yaml

# Sprawdzenie informacji o mapie
ros2 topic echo /map --once | grep -E "(width|height|resolution)"
```

Przydatne skróty klawiszowe

Ctrl+C	Zatrzymanie bieżącego programu/node-a
Ctrl+Alt+T	Nowe okno terminala
Ctrl+Shift+C / V	Kopiuj/Wklej w terminalu
Tab	Autouzupełnianie komend ros2
strzalka gora	Poprzednia komenda w historii
ros2 run ... &	Uruchomienie w tle (można pisać kolejne komendy)