NAWIGACJA ROBOTÓW HUMANOIDALNYCH

WYKŁAD 10

Nav2: Stos nawigacyjny dla robotów mobilnych

Architektura, konfiguracja i wdrożenie kompletnego systemu nawigacji w ROS 2

dr inż. Mateusz Pomianek

Nav2 | BehaviorTree | Costmap | Planner | Controller

Plan wykładu

Sekcja 1

Architektura Nav2

Czym jest Nav2, historia, komponenty, przepływ danych, lifecycle manager, BehaviorTree jako rdzeń

Sekcja 2

Instalacja i konfiguracja

Instalacja pakietów, pliki startowe, URDF, konfiguracja YAML, TF2 i mapa środowiska

Sekcja 3

Costmapy i planowanie

Global costmap, local costmap, warstwy kosztów, planista globalny (NavFn, Smac), kontroler lokalny (DWB, RPP, MPPI)

Sekcja 4

Wdrożenie i rozszerzenia

Niestandardowe wtyczki, debugowanie (RViz2, CLI), monitoring jakości, integracja z SLAM i humanoidem

01

SEKCJA

Architektura Nav2

Historia, komponenty, BehaviorTree i przepływ danych

- **Nav2** (Navigation 2) to oficjalny stos nawigacyjny ROS 2 – następca move_base z ROS 1
- **Cel**: autonomiczne przemieszczanie robota z punktu A do punktu B w mapie 2D lub 3D z unikaniem przeszkód
- **Architektura modularna**: każdy komponent jest osobnym node'em ROS 2 z interfejsem wtyczek (plugins)
- **Silnik decyzyjny**: BehaviorTree.CPP v4 – hierarchiczne drzewo zachowań steruje całą sekwencją nawigacji
- **Lifecycle management**: każdy node przechodzi stany configure → activate → deactivate → cleanup dla bezpiecznego uruchamiania
- **Kompatybilność**: działa z każdym robotem mającym poprawne TF2 i odometrię; nie wymaga konkretnego sprzętu
- **Licencja**: Apache 2.0; aktywnie rozwijany przez Open Navigation LLC i społeczność ROS 2
- **Dokumentacja**: https://docs.nav2.org – w pełni opisane API, parametry i przykłady

Główne komponenty Nav2

BT Navigator

Wykonuje drzewo BehaviorTree; koordynuje wszystkie inne komponenty; publikuje feedback dla akcji NavigateToPose

Planner Server

Planista globalny; wtyczki: NavFn (Dijkstra), Smac Planner (A*, Hybrid-A*); zwraca ścieżkę globalną do BT Navigator

Controller Server

Kontroler lokalny; wtyczki: DWB, RPP, MPPI; generuje komendy cmd_vel na podstawie localnej mapy i ścieżki

Costmap 2D

Global + local costmap; warstwy: static, inflation, obstacle, voxel; aktualizacja z LiDAR i RGB-D

Behavior Server

Zachowania awaryjne: Spin, BackUp, WaitAtWaypoint, ClearCostmap; wywoływane przez BT gdy navigation fails

Waypoint Follower

Sekwencyjne nawigowanie do listy celów; task executor dla każdego waypoint; kompatybilny z VLM goal list

Przepływ danych w Nav2 - od celu do cmd_vel

1

Cel nawigacji

Operator lub system misji wysyła akcję NavigateToPose z celem (x, y, theta) w ramce 'map' do BT Navigator

2

Global Plan

BT Navigator wywołuje Planner Server; Smac/NavFn wyznacza globalną ścieżkę na global costmap od bieżącej pozycji do celu

3

Local Control

Controller Server śledzi ścieżkę; DWB/RPP/MPPI oblicza (v, ω) na local costmap z uwzględnieniem przeszkód dynamicznych

4

Zachowania awaryjne

Gdy progress_checker wykryje utknięcie → BT wywołuje Spin lub BackUp; po odblokow. wraca do śledzenia ścieżki

5

Cel osiągnięty

goal_checker sprawdza odległość < xy_goal_tolerance i kąt < yaw_goal_tolerance; BT zwraca SUCCESS do klienta

- **BT (BehaviorTree.CPP v4)** to hierarchiczny automat stanów z modularną, konfigurowalną strukturą XML
- **Węzły BT**: Action (wywołaj plugin), Condition (sprawdź stan), Decorator (modyfikuj wynik), Sequence, Fallback
- **Domyślne drzewo Nav2** (navigate_w_replanning_and_recovery.xml): planuj → śledź → retry → recover → fail
- **Własne drzewo XML:** definiujesz kolejność prób, warunki replanu i zachowania awaryjne bez zmiany kodu C++
- **Hot-swappable**: można podmienić plik XML w czasie działania systemu przez parametr 'default_bt_xml_filename'
- **Condition nodes dostępne**: GoalReached, IsBatteryLow, IsStuck, IsPathValid, TransformAvailable
- **Integracja VLM**: BT może wywoływać node VLM jako Action i na podstawie wyniku wybierać gałąź nawigacji
- **Debugowanie**: Groot2 – graficzny monitor drzewa BT w czasie rzeczywistym; publikacja na porcie 1667

- **Lifecycle nodes** (rclcpp_lifecycle) mają stany: Unconfigured → Inactive → Active → Finalized
- **Nav2 Lifecycle Manager** uruchamia wszystkie komponenty w prawidłowej kolejności i zarządza przejściami
- **Kolejność startu**: map_server → AMCL → costmap_2d → planner_server → controller_server → bt_navigator
- **Przejście 'configure':** node ładuje parametry YAML, inicjalizuje wtyczki i alokuje zasoby
- **Przejście 'activate':** node zaczyna subskrybować tematy i publikować wyniki; robot gotowy do nawigacji
- **Przejście 'deactivate':** bezpieczne zatrzymanie nawigacji bez utraty stanu (np. przy przejściu do ładowarki)
- **autostart**: true w launch file automatycznie uruchamia cały stos po starcie systemu
- **bond_timeout**: jeśli node przestanie wysyłać heartbeat w ciągu X sekund → Lifecycle Manager wyzwala restart

02

SEKCJA

Instalacja i konfiguracja

Środowisko ROS 2, pliki startowe, YAML, TF2 i mapa

- **sudo apt install ros-jazzy-navigation2 ros-jazzy-nav2-bringup** – instalacja binarną paczką
- **Alternatywnie**: build ze źródeł – git clone https://github.com/ros-navigation/navigation2 && colcon build
- **Zależności**: slam_toolbox, robot_localization, tf2_ros, sensor_msgs, geometry_msgs, nav_msgs
- **Weryfikacja**: ros2 pkg list | grep nav2 – powinno zwrócić >15 pakietów nav2_*
- **Przykłady**: ros2 launch nav2_bringup tb3_simulation_launch.py – uruchomienie z TurtleBot3 w Gazebo
- **Wymagania środowiskowe**: poprawna ramka 'map', ramka 'base_link', ciągłe TF między nimi przez 'odom'
- **URDF robota**: musi zawierać wszystkie joint i link z dokładnymi wymiarami (footprint) dla costmap inflation
- **Testowe środowisko**: Nav2 Community Maps lub Gazebo World do pierwszych eksperymentów przed prawdziwym robotem

KOD PYTHON

Plik startowy Nav2 (Python launch file)

```python
from launch import LaunchDescription
from launch.actions import IncludeLaunchDescription
from launch_ros.actions import Node
from ament_index_python.packages import get_package_share_directory
import os

def generate_launch_description():
    nav2_dir = get_package_share_directory('nav2_bringup')
    params_file = os.path.join(
        get_package_share_directory('my_robot_nav'),
        'config', 'nav2_params.yaml')
    map_file = os.path.join(
        get_package_share_directory('my_robot_nav'),
        'maps', 'office_map.yaml')

    nav2_launch = IncludeLaunchDescription(
        os.path.join(nav2_dir, 'launch', 'bringup_launch.py'),
        launch_arguments={
            'map': map_file,
            'params_file': params_file,
            'use_sim_time': 'false',
            'autostart': 'true',
        }.items()
    )
    return LaunchDescription([nav2_launch])
```

params_file i map_file przekazane do nav2_bringup; autostart:true automatycznie aktywuje lifecycle nodes

KOD PYTHON

Plik parametrów Nav2 (nav2_params.yaml – fragment)

```yaml
bt_navigator:
  ros__parameters:
    use_sim_time: False
    global_frame: map
    robot_base_frame: base_link
    odom_topic: /odom
    default_bt_xml_filename: navigate_w_replanning_and_recovery.xml
    plugin_lib_names:
      - nav2_compute_path_to_pose_action_bt_node
      - nav2_follow_path_action_bt_node
      - nav2_back_up_action_bt_node
      - nav2_spin_action_bt_node

controller_server:
  ros__parameters:
    controller_frequency: 20.0
    progress_checker_plugin: "progress_checker"
    goal_checker_plugin: "goal_checker"
    controller_plugins: ["FollowPath"]
    FollowPath:
      plugin: "nav2_mppi_controller::MPPIController"
      time_steps: 56
      model_dt: 0.05
      batch_size: 2000
```

Każdy komponent konfigurowany w oddzielnej sekcji YAML; plugin: wybiera wtyczkę (NavFn / DWB / MPPI)

- **Drzewo TF obowiązkowe**: map → odom → base_link → [sensory: laser_frame, camera_frame]
- **map → odom**: publikowany przez AMCL lub slam_toolbox (korekta SLAM, może być skokowa)
- **odom → base_link**: publikowany przez robot_localization (EKF) lub driver odometrii; musi być ciągły
- **base_link → laser_frame**: statyczna transformacja z URDF robota; kalibracja extrinsic sensora
- **Mapa 2D**: plik .pgm (obraz) + .yaml (rozdzielczość, origin, occupied_thresh, free_thresh)
- **Tworzenie mapy**: slam_toolbox (async_slam_toolbox_node) lub nav2_map_server podczas eksploracji
- **Serwowanie mapy**: ros2 launch nav2_map_server map_server.launch.py map:=office_map.yaml
- **Weryfikacja TF**: ros2 run tf2_tools view_frames → wyświetla drzewo TF jako plik PDF

03

SEKCJA

Costmapy i planowanie

Warstwy kosztów, planista globalny i lokalny, strojenie

Costmap 2D: global i local

Global Costmap

- **Zasięg**: cały budynek lub piętro; rozdzielczość 5 cm
- **Warstwy statyczne**: static_layer (mapa .pgm) + inflation_layer
- **Aktualizacja**: rzadka (przy zmianie mapy lub przeszkodzie stałej)
- **Użycie**: Planner Server wyznacza trasę na global costmap
- **Pamięć**: ~50 MB dla biura 50×50 m przy rozdzielczości 5 cm
- **Parametry**: width, height, resolution, robot_radius, inflation_radius

Local Costmap

- **Zasięg**: 3–5 m wokół robota; rolujące okno (rolling window: true)
- **Warstwy**: obstacle_layer (LiDAR+RGB-D) + inflation_layer
- **Aktualizacja**: co 100–200 ms; uwzględnia obiekty dynamiczne
- **Użycie**: Controller Server śledzi ścieżkę na local costmap
- **observation_sources**: scan (LiDAR 2D), pointcloud (RGB-D)
- **Parametry**: update_frequency, publish_frequency, max_obstacle_height

- **static_layer**: wczytuje mapę .pgm; occupied_thresh → koszt=254, free_thresh → koszt=0, unknown → koszt=255
- **inflation_layer**: tworzy gradient kosztu wokół przeszkód; robot_radius=0.35 m; inflation_radius=0.55 m
- **obstacle_layer**: aktualizuje siatkę z obserwacji LiDAR i kamer; mark i clear przeszkód w czasie rzeczywistym
- **voxel_layer**: trójwymiarowa wersja obstacle_layer; pozwala wykrywać nawisy i przeszkody na różnych wysokościach
- **range_layer**: integruje dane z sensorów zasięgowych (sonar, IR); przydatne przy słabym LiDAR-ze
- **social_layer** (plugin): niestandardowa warstwa z anizotropowym Gaussem wokół pieszych (patrz W7)
- **spatio_temporal_voxel_layer**: zaawansowana warstwa 3D z decay time; obiekty dynamiczne znikają po T sekundach
- **Kolejność warstw w YAML**: plugins: ['static_layer', 'obstacle_layer', 'inflation_layer'] – kolejność ma znaczenie

Planista globalny: NavFn i Smac Planner

NavFnPlanner (Dijkstra / A*)

- Klasyczny A*/Dijkstra na siatce global costmap
- **Szybki dla prostych**, płaskich środowisk biurowych
- **plugin**: 'nav2_navfn_planner/NavfnPlanner'
- **use_astar**: true / false (domyślnie Dijkstra)
- **allow_unknown**: true (planowanie przez nieznane obszary)
- **Wada**: nie uwzględnia kinematyki robota; ostre zakręty
- **Zalecany**: TurtleBot, roboty różnicowe, szybkie prototypy

SmacPlanner (Hybrid-A*)

- **Uwzględnia kinematykę**: minimalny promień skrętu
- **SmacPlannerHybrid**: planowanie z orientacją (x, y, theta)
- **SmacPlanner2D**: szybszy, bez orientacji; jak NavFn+
- **plugin**: 'nav2_smac_planner/SmacPlannerHybrid'
- **minimum_turning_radius**: 0.4 (dla humanoida: 0.1)
- **Zalecany**: humanoidy, auta autonomiczne, holonomiczne
- Dłuższy czas obliczenia (~50 ms vs ~10 ms NavFn)

Kontrolery lokalne: DWB, RPP, MPPI

DWB Controller

Dynamic Window Approach;
ocena trajektorii przez krytyków;
plugin: 'dwb_core::DWBLocalPlanner';
min/max_vel_x, min/max_vel_theta;
Krytyki: GoalAlign, PathAlign, ObstacleFootprint, RotateToGoal

Regulated Pure Pursuit

Pure Pursuit z regulacją prędkości;
plugin: 'nav2_regulated_pure_pursuit';
+_controller/RegulatedPurePursuitController';
lookahead_dist: 0.6 m;
use_velocity_scaled_lookahead: true;
Lekki, wolny, bardzo dokładny

MPPI Controller

Model Predictive Path Integral;
próbkuje N trajektorii GPU-based;
plugin: 'nav2_mppi_controller
::MPPIController';
batch_size: 2000, time_steps: 56;
Najlepszy dla skomplikowanego terenu

KOD PYTHON

Konfiguracja MPPI - strojenie dla humanoida

```
controller_server:
  ros__parameters:
    controller_frequency: 20.0
    FollowPath:
      plugin: "nav2_mppi_controller::MPPIController"
      time_steps: 56           # horyzont predykcji = 56 * 0.05s = 2.8 s
      model_dt: 0.05           # krok czasowy [s]
      batch_size: 2000         # liczba losowanych trajektorii
      vx_std: 0.2              # odchylenie losowania v_x
      vy_std: 0.0              # 0 dla robotów nie-holonomicznych
      wz_std: 0.4              # odchylenie losowania omega_z
      vx_max: 0.5              # max prędkość do przodu [m/s]
      vx_min: -0.15            # min prędkość (cofanie) [m/s]
      wz_max: 1.0              # max prędkość kątowa [rad/s]
      critics: ["GoalCritic", "GoalAngleCritic",
                "ObstaclesCritic", "PathAlignCritic",
                "PathFollowCritic", "PreferForwardCritic"]
      ObstaclesCritic.collision_cost: 1000.0
      ObstaclesCritic.critical_weight: 20.0
```

Opis: batch_size=2000 próbkuje 2000 trajektorii w każdym kroku; critics oceniają każdą z nich; wybrana jest optymalna

- **goal_checker**: sprawdza czy robot osiągnął cel; warunki: xy_goal_tolerance i yaw_goal_tolerance
- **SimpleGoalChecker**: xy_goal_tolerance=0.25 m, yaw_goal_tolerance=0.25 rad – odpowiednie dla robotów usługowych
- **StoppedGoalChecker**: dodatkowo sprawdza czy robot jest nieruchomy (trans_stopped_velocity < 0.25 m/s)
- **progress_checker**: wykrywa brak postępu; wywołuje zachowania awaryjne gdy robot utknął
- **SimpleProgressChecker**: movement_time_allowance=10.0 s bez minimalnego przemieszczenia → trigger recovery
- **PoseProgressChecker (zaawansowany):** sprawdza zmianę pozy (dx, dy, dyaw) w oknie czasowym; dokładniejszy
- **Parametry strojenia**: zbyt mała tolerancja → robot kręci się przy celu; zbyt duża → niedokładne dotarcie
- **Dla humanoida**: xy_goal_tolerance=0.15 m, yaw_goal_tolerance=0.1 rad – precyzja wymagana przy manipulacji

04

SEKCJA

Wdrożenie i rozszerzenia Nav2

Wtyczki, debugowanie, monitoring, integracja z humanoidem

KOD PYTHON

Nawigacja do punktu przez Python API

```python
import rclpy
from rclpy.node import Node
from nav2_simple_commander.robot_navigator import BasicNavigator
from geometry_msgs.msg import PoseStamped

def main():
    rclpy.init()
    nav = BasicNavigator()

    # Pozycja startowa (opcjonalna, jeśli AMCL nie jest zainicjowany)
    initial_pose = PoseStamped()
    initial_pose.header.frame_id = 'map'
    initial_pose.pose.position.x = 0.0
    initial_pose.pose.position.y = 0.0
    initial_pose.pose.orientation.w = 1.0
    nav.setInitialPose(initial_pose)
    nav.waitUntilNav2Active() # czekaj aż Nav2 będzie gotowe

    goal = PoseStamped()
    goal.header.frame_id = 'map'
    goal.pose.position.x = 3.5
    goal.pose.position.y = 1.2
    goal.pose.orientation.w = 1.0

    nav.goToPose(goal) # wyślij cel
    while not nav.isTaskComplete():
        feedback = nav.getFeedback()
        print(f"Odległość do celu: {feedback.distance_remaining:.2f} m")
    print("Status:", nav.getResult())
```

nav2_simple_commander opakowuje ActionClient; goToPose wysyła NavigateToPose; getFeedback() zwraca postęp

KOD PYTHON

Nawigacja przez listę punktów (Waypoint Follower)

```python
from nav2_simple_commander.robot_navigator import BasicNavigator
from geometry_msgs.msg import PoseStamped
import rclpy

def make_pose(x, y, w=1.0):
    p = PoseStamped()
    p.header.frame_id = 'map'
    p.pose.position.x = x
    p.pose.position.y = y
    p.pose.orientation.w = w
    return p

def main():
    rclpy.init()
    nav = BasicNavigator()
    nav.waitUntilNav2Active()
        waypoints = [          # Trasa: korytarz -> sala -> powrót
        make_pose(2.0, 0.5),   # skrzyżowanie korytarza
        make_pose(4.5, 1.8),   # drzwi sali konferencyjnej
        make_pose(6.0, 1.8),   # środek sali
        make_pose(0.0, 0.0),   # powrót do stacji bazowej
    ]
    nav.followWaypoints(waypoints)
    while not nav.isTaskComplete():
        feedback = nav.getFeedback()
        idx = feedback.current_waypoint
        print(f"Aktualny waypoint: {idx+1}/{len(waypoints)}")
    print("Trasa zakończona!")
    nav.lifecycleShutdown(); rclpy.shutdown()
```

followWaypoints() sekwencyjnie nawiguje przez listę celów; WaypointFollower pozwala dodać task executor

KOD PYTHON

Szkielet niestandardowej wtyczki planisty globalnego (C++)

```cpp
#include "nav2_core/global_planner.hpp"
#include "pluginlib/class_list_macros.hpp"

class MyCustomPlanner : public nav2_core::GlobalPlanner {
public:
  void configure(
    const rclcpp_lifecycle::LifecycleNode::WeakPtr & parent,
    std::string name,
    std::shared_ptr<tf2_ros::Buffer> tf,
    std::shared_ptr<nav2_costmap_2d::Costmap2DROS> costmap_ros) override
  {
    node_ = parent.lock();
    name_ = name;
    costmap_ = costmap_ros->getCostmap();
    RCLCPP_INFO(node_->get_logger(), "MyCustomPlanner configured");
  }

  nav_msgs::msg::Path createPlan(
    const geometry_msgs::msg::PoseStamped & start,
    const geometry_msgs::msg::PoseStamped & goal) override
  {
    nav_msgs::msg::Path path;
    // ======= TU WSTAWIMY NASZ ALGORYTM PLANOWANIA =======
    return path;
  }
};

PLUGINLIB_EXPORT_CLASS(MyCustomPlanner, nav2_core::GlobalPlanner)
```

Implementuj GlobalPlanner lub Controller; zarejestruj przez pluginlib; wskaż plugin: 'mypkg::MyCustomPlanner' w YAML

KOD PYTHON

Bezpośredni klient akcji NavigateToPose w rclpy

```python
import rclpy
from rclpy.action import ActionClient
from rclpy.node import Node
from nav2_msgs.action import NavigateToPose
from geometry_msgs.msg import PoseStamped

class NavGoalClient(Node):
    def __init__(self):
        super().__init__('nav_goal_client')
        self.client = ActionClient(self, NavigateToPose, 'navigate_to_pose')

    def send_goal(self, x, y, yaw_w=1.0):
        goal = NavigateToPose.Goal()
        goal.pose = PoseStamped()
        goal.pose.header.frame_id = 'map'
        goal.pose.header.stamp = self.get_clock().now().to_msg()
        goal.pose.pose.position.x = x
        goal.pose.pose.position.y = y
        goal.pose.pose.orientation.w = yaw_w

        self.client.wait_for_server()
        future = self.client.send_goal_async(
            goal, feedback_callback=self.feedback_cb)
        future.add_done_callback(self.goal_response_cb)

    def feedback_cb(self, msg):
        fb = msg.feedback
        self.get_logger().info(
            f"remain={fb.distance_remaining:.2f} m, "
            f"eta={fb.estimated_time_remaining.sec}s")
```

Ten wariant pokazuje surowy mechanizm ActionClient: pełna kontrola nad nagłówkiem, feedbackiem i obsługą odrzucenia celu.

KOD PYTHON

Fragment własnego drzewa BehaviorTree dla Nav2

```
<root main_tree_to_execute="MainTree">
  <BehaviorTree ID="MainTree">
    <RecoveryNode number_of_retries="3" name="NavigateRecovery">
      <PipelineSequence name="NavigateWithReplanning">
        <RateController hz="1.0">
          <ComputePathToPose goal="{goal}" path="{path}"
                             planner_id="GridBased"/>
        </RateController>
        <FollowPath path="{path}" controller_id="FollowPath"/>
      </PipelineSequence>

      <ReactiveFallback name="RecoveryFallback">
        <GoalUpdated/>
        <Sequence name="ClearingActions">
          <ClearEntireCostmap name="ClearGlobal" service_name="global_costmap/clear_entirely_global_costmap"/>
          <ClearEntireCostmap name="ClearLocal" service_name="local_costmap/clear_entirely_local_costmap"/>
          <Spin spin_dist="1.57"/>
        </Sequence>
      </ReactiveFallback>
    </RecoveryNode>
  </BehaviorTree>
</root>
```

Drzewo wymusza replan co 1 Hz, śledzi ścieżkę i uruchamia kontrolowaną sekwencję recovery po niepowodzeniu

KOD PYTHON

Konfiguracja własnej warstwy Social Layer w local_costmap

```yaml
local_costmap:
  local_costmap:
    ros__parameters:
      update_frequency: 10.0
      publish_frequency: 5.0
      global_frame: odom
      robot_base_frame: base_link
      rolling_window: true
      width: 5
      height: 5
      resolution: 0.05
      plugins: ["obstacle_layer", "social_layer", "inflation_layer"]

      social_layer:
        plugin: "my_nav2_plugins::SocialLayer"
        enabled: true
        people_topic: /people_tracked
        front_sigma: 1.2       # większy koszt przed człowiekiem
        rear_sigma: 0.45       # mniejszy koszt za człowiekiem
        side_sigma: 0.65
        amplitude: 220.0
        cutoff: 15.0

      inflation_layer:
        plugin: "nav2_costmap_2d::InflationLayer"
        inflation_radius: 0.65
        cost_scaling_factor: 3.0
```

Warstwa społeczna modyfikuje lokalną mapę kosztów: kontroler omija ludzi z większym marginesem z przodu niż z tyłu

KOD PYTHON

Analiza rosbag2: długość trasy, czas misji i komendy cmd_vel

```python
import numpy as np
from rosbags.highlevel import AnyReader
from pathlib import Path

bag = Path('nav2_run_01')
poses, speeds = [], []
with AnyReader([bag]) as reader:
    conns = [c for c in reader.connections if c.topic in ['/odom', '/cmd_vel']]
    for conn, t, raw in reader.messages(connections=conns):
        msg = reader.deserialize(raw, conn.msgtype)
        if conn.topic == '/odom':
            p = msg.pose.pose.position
            poses.append((t * 1e-9, p.x, p.y))
        elif conn.topic == '/cmd_vel':
            speeds.append((t * 1e-9, msg.linear.x, msg.angular.z))

xy = np.array([(x, y) for _, x, y in poses])
segments = np.linalg.norm(np.diff(xy, axis=0), axis=1)
path_len = segments.sum()
mission_time = poses[-1][0] - poses[0][0]
mean_v = np.mean([abs(v) for _, v, _ in speeds])

print(f"path={path_len:.2f} m, time={mission_time:.1f} s")
print(f"mean |v|={mean_v:.2f} m/s")
```

Taki skrypt zamienia zapis rosbag2 na metryki porównawcze dla DWB, RPP i MPPI po identycznym scenariuszu testowym

- **RViz2**: wizualizacja global_costmap, local_costmap, globalnego planu, celu nawigacji i footprintu robota
- **Groot2**: graficzny monitor drzewa BehaviorTree w czasie rzeczywistym; połączenie przez ZMQ port 1667
- **ros2 topic echo /local_costmap/costmap** – wyświetla surowe wartości kosztów jako nav_msgs/OccupancyGrid
- **ros2 topic echo /plan** – globalny plan wyznaczony przez Planner Server (nav_msgs/Path)
- **ros2 action list** – wyświetla dostępne akcje Nav2; send_goal przez CLI: ros2 action send_goal /navigate_to_pose ...
- **ros2 service call /clear_entirely_both_costmaps** – czyści obie mapy kosztów w przypadku 'ghost obstacles'
- **nav2_lifecycle_manager**: ros2 service call /lifecycle_manager_navigation/manage_nodes – zmiana stanu nodes
- **rosbag2 record**: nagraj dane z sensorów i nawigacji; odtwórz w Gazebo do postmortem po incydencie

Porównanie kontrolerów i globalnych planistów

Komponent	Wtyczka	Czas [ms]	Kinematyka	Najlepsze użycie
NavFnPlanner	nav2_navfn_planner	~10	Brak	Proste środowiska, różnicowe roboty
SmacPlanner2D	nav2_smac_planner	~25	Ograniczona	Szybkie planowanie z pewną jakością
SmacPlannerHybrid	nav2_smac_planner	~50	Pełna (SE2)	Humanoidy, pojazdy autonomiczne
DWB Controller	dwb_core	~5	Różnicowa	Stabilne środowiska, prosta konfiguracja
Regulated PP	nav2_regulated_pp	~2	Różnicowa	Wąskie korytarze, precyzyjne śledzenie
MPPI Controller	nav2_mppi_controller	~15*	Dowolna	Złożone środowiska, dynamiczne przeszkody

Integracja Nav2 z SLAM i lokalizacją

SLAM Toolbox (mapowanie online)

- **async_slam_toolbox_node**: SLAM online w ROS 2
- **Publikuje**: /map (OccupancyGrid) i TF map→odom
- **Integracja**: Nav2 costmap subskrybuje /map
- **slam_toolbox_params.yaml**: resolution=0.05 m
- **Tryby**: online (eksploracja) i localization (znana mapa)
- **Zapis mapy**: ros2 service call /slam_toolbox/save_map
- **Kompatybilny z Nav2 bringup**: use_slam:=True

AMCL (lokalizacja w znana mapie)

- **Adaptive MCL**: filtr cząsteczkowy z obserwacjami LiDAR
- **Publikuje**: TF map→odom (skokowa korekta SLAM)
- **Parametry**: max_particles=2000, min_particles=500
- **laser_model_type**: likelihood_field (zalecany)
- **Inicjalizacja**: set_initial_pose przez RViz2 lub Python API
- **Wymagania**: /scan (sensor_msgs/LaserScan) + /map
- **Przy humanoidzie**: lidar wyżej = mniej szumów podłogi

- **Footprint**: dla humanoida użyj polygon zamiast robot_radius; odwzorowuje rzeczywisty kształt ciała
- **footprint**: [[0.25, 0.15], [0.25, -0.15], [-0.25, -0.15], [-0.25, 0.15]] – prostokąt 50×30 cm
- **cmd_vel → WBC**: node-przekaźnik konwertuje geometry_msgs/Twist z Nav2 na komendy footstep planera
- **Częstość kontrolera**: controller_frequency=10.0 Hz dla humanoida (vs 20 Hz dla kołowego); WBC działa 200 Hz
- **velocity_smoother (Nav2 plugin):** wygładza cmd_vel aby unikać gwałtownych zmian prędkości wywołujących niestabilność
- **3D costmap**: gdy humanoid chodzi po schodach → voxel_layer w local_costmap; max_obstacle_height=2.0 m
- **Footstep server jako Behavior Tree Action node**: BT wywołuje footstep planer zamiast bezpośrednio cmd_vel
- **Stabilność**: custom critic DWB/MPPI odrzuca trajektorie naruszające ZMP; safety_monitor vetuje cmd_vel (W9)

Kompletny pipeline wdrożenia Nav2 krok po kroku

1

Środowisko i robot

Zainstaluj ROS 2 + Nav2. Przygotuj URDF robota z poprawnymi wymiarami i joint. Zweryfikuj TF2 przez view_frames.

2

Mapa środowiska

Uruchom slam_toolbox i przeprowadź robota przez całe środowisko. Zapisz mapę: save_map. Weryfikuj w RViz2.

3

Konfiguracja YAML

Utwórz nav2_params.yaml: dobierz footprint, inflation_radius, wybierz planistę (SmacHybrid) i kontroler (MPPI). Zacznij od domyślnych wartości.

4

Uruchomienie i testy SIL

Uruchom Nav2 w Gazebo z przygotowaną mapą. Testuj trasy TC-01 do TC-04 (prosta trasa, statyczna i dynamiczna przeszkoda, waypoints).

5

Strojenie i wdrożenie

Dostosuj parametry na podstawie wyników testów. Uruchom na prawdziwym robocie. Monitoruj RMSE, kolizje i zużycie energii przez rosbag2.

- **/navigate_to_pose/_action/feedback**: distance_remaining, estimated_time_remaining, current_pose – co 100 ms
- **/local_costmap/costmap_updates i /global_costmap/costmap_updates**: weryfikacja aktualizacji warstw kosztów
- **/plan**: globalny plan wyznaczony przez Planner Server; liczba punktów i długość trasy
- **/cmd_vel**: aktualna komenda prędkości; logujesz do rosbag2 i analizujesz energię (integral τ*dq)
- **Nav2 diagnostics topic /diagnostics**: stan lifecycle nodes, częstość planowania, czas odpowiedzi serwera
- **rqt_robot_monitor**: graficzna wizualizacja /diagnostics; wbudowany w nav2_bringup
- **Custom metrics node**: subskrybuje /odom i cel; liczy RMSE pozycji, czas misji i liczbę replanu online
- **Grafana + ROSBridge**: eksport metryk Nav2 do dashboardu Grafana dla monitorowania floty robotów

PODSUMOWANIE

01	**Nav2** to modularny stos nawigacyjny ROS 2 oparty na BehaviorTree, lifecycle management i architekturze wtyczek.
02	**Dwa poziomy map:** global costmap (trasa A*) i local costmap (śledzenie z unikaniem przeszkód dynamicznych).
03	**Planiści globalni** (NavFn, SmacHybrid) i lokalni (DWB, RPP, MPPI) są wymieniane przez plik YAML bez rekompilacji.
04	**Integracja z humanoidem wymaga**: polygon footprint, cmd_vel→WBC bridge, ZMP critic i dostosowanej częstości kontrolera.
05	**Pipeline wdrożenia**: URDF → SLAM (mapa) → nav2_params.yaml → testy SIL/Gazebo → strojenie → robot rzeczywisty.

Pytania problemowe i zadanie do analizy

1. Dlaczego MPPI Controller generuje lepsze trajektorie niż DWB w zatłoczonym korytarzu? Jakie parametry mają kluczowy wpływ?

2. Co się stanie, gdy transformacja map→odom nie będzie dostępna przez >500 ms podczas nawigacji? Opisz zachowanie Nav2.

3. Jak skonfigurować Nav2, aby robot humanoidalny zatrzymał się 0.5 m przed człowiekiem i czekał na ruch? Które komponenty zmienisz?

4. Porównaj SmacPlannerHybrid i NavFnPlanner pod kątem czasu obliczenia, jakości trasy i uwzględnienia kinematyki humanoida.

Zadanie

Zaimplementuj kompletny launch file uruchamiający Nav2 z mapą biura, skonfiguruj MPPI jako kontroler lokalny i przetestuj nawigację do 3 waypoints w Gazebo. Zmierz RMSE pozycji i czas dotarcia. Porównaj MPPI z DWB na mapie z 3 dynamicznymi przeszkodami. Bonus: dodaj niestandardową warstwę Social Cost w local_costmap integrując SocialCostmap z Wykładu 7.