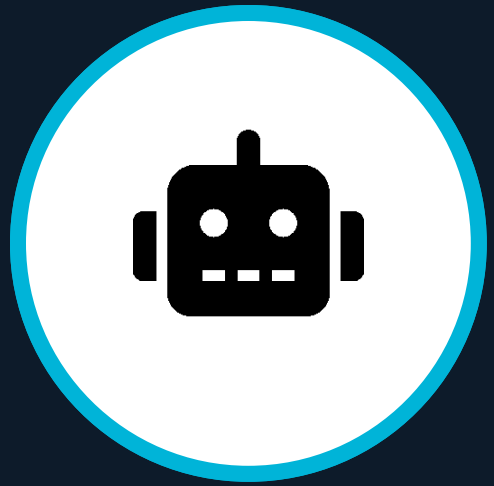
NAWIGACJA ROBOTÓW HUMANOIDALNYCH

WYKŁAD 1

Wprowadzenie do percepcji i nawigacji robota humanoidalnego



dr inż. Mateusz Pomianek

Percepcja | Mapowanie | Sensory | Architektura

Plan wykładu

Sekcja 1

Problem nawigacji humanoida, moduły systemu, ramy odniesienia, miary jakości

Sekcja 2

Sensory i percepcja: IMU, LiDAR, kamery, filtracja i synchronizacja danych

Sekcja 3

Reprezentacja środowiska: mapy zajętości, cost map, mapowanie lokalne/globalne

Kody Python

Struktury modułów, filtracja LiDAR, detekcja przeszkód, budowa grid map

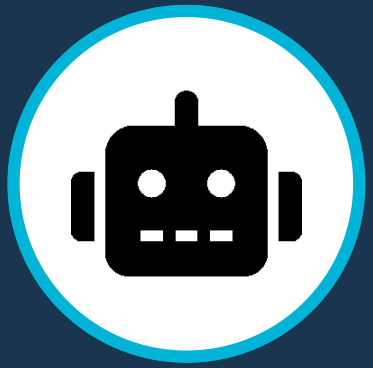
Najpierw omówimy sam problem nawigacji humanoida, główne moduły systemu, układy odniesienia oraz miary jakości. Następnie przejdziemy do sensorów i percepcji, czyli IMU, LiDAR-u, kamer, filtracji i synchronizacji danych. W trzeciej części zajmiemy się reprezentacją środowiska: mapami zajętości, cost mapami oraz mapowaniem lokalnym i globalnym. Po drodze będziemy też analizować krótkie przykłady w Pythonie.

01

SEKCJA

Problem nawigacji humanoida

Definicja, architektura systemu i miary jakości nawigacji dwunożnej



Nawigacja humanoida a platforma kołowa

Robot kołowy (AGV / różnicowy)

- Stabilna baza – zawsze min. 2 punkty kontaktu
- Prosta kinematyka – odometria kół wystarczy
- Brak ryzyka wywrotki przy ruchu do przodu
- Typowe środowisko: płaska posadzka
- Dojrzałe algorytmy SLAM i planowania

Robot humanoidalny

- Dynamiczna równowaga: stopa uniesiona w połowie cyklu
- Złożona kinematyka – 30+ stopni swobody
- Każdy krok to potencjalna utrata stabilności
- Środowisko: schody, progi, nierówności, drzwi
- Konieczność planowania kroków i trajektorii CoM

Proszę zwrócić uwagę, że humanoid nawigacyjnie bardzo różni się od platformy kołowej. Robot kołowy ma stabilną bazę, prostą kinematykę i zwykle porusza się po płaskiej posadzce, więc wiele klasycznych metod działa tam bardzo dobrze. Humanoid natomiast funkcjonuje w warunkach dynamicznej równowagi, ma złożoną kinematykę, ryzyko utraty stabilności przy każdym kroku i musi radzić sobie ze schodami, progami, nierównościami oraz drzwiami. Dlatego w jego przypadku sama trajektoria nie wystarcza, trzeba planować także kroki i trajektorię środka masy.

Główne moduły systemu nawigacji

Percepcja

IMU, enkodery, kamery, LiDAR, czujniki sił

Lokalizacja

Estymacja stanu, fuzja sensoryczna, SLAM

Mapowanie

Occupancy grid, height map, mapa semantyczna

Planowanie globalne

Dijkstra, A*, graph-based path planning

Planowanie lokalne

DWA, potential fields, MPC

Sterowanie chodem

ZMP, LIPM, footstep planning, WBC

Typowy system nawigacji humanoida składa się z sześciu głównych modułów. Najpierw mamy percepcję, która zbiera dane z IMU, enkoderów, kamer, LiDAR-u i czujników sił. Potem lokalizacja, czyli estymacja stanu, fuzja sensoryczna i często SLAM. Dalej mapowanie, na przykład occupancy grid, height map albo mapa semantyczna. Na tej podstawie działa planowanie globalne, później planowanie lokalne, a na końcu sterowanie chodem, które zamienia decyzje planistyczne na rzeczywiste kroki, trajektorie CoM i komendy całego ciała.

Przepływ informacji w systemie nawigacji

1. **Sensory** → **Percepcja**: surowe dane z IMU, enkoderów, LiDAR, kamer
2. **Percepcja** → **Estymacja stanu**: przefiltrowane obserwacje do fuzji sensorycznej
3. **Estymacja stanu** → **Planista globalny**: aktualna pozycja i mapa środowiska
4. **Planista globalny** → **Planista lokalny**: globalna ścieżka i mapa kosztów
5. **Planista lokalny** → **Generator kroków**: lokalna trajektoria do wykonania
6. **Generator kroków** → **Kontroler chodu**: sekwencja ustawień stóp i trajektoria CoM
7. **Kontroler chodu** → **Aktuatory**: komendy momentów i pozycji stawów
8. **Aktuatory** → **Sensory**: zamknięta pętla sprzężenia zwrotnego

Przepływ informacji w takim systemie jest sekwencyjny, ale zamknięty pętlą sprzężenia zwrotnego. Sensory dostarczają surowe dane do percepcji, percepcja przekazuje przefiltrowane obserwacje do estymacji stanu, a estymator podaje aktualną pozycję i mapę do planisty globalnego. Następnie planista globalny wyznacza ścieżkę, planista lokalny dopasowuje ją do bieżącej sytuacji, generator kroków tworzy sekwencję ustawień stóp, kontroler chodu zamienia to na trajektorie i momenty, a aktuatory wykonują ruch. Potem sensory znowu mierzą efekt tej akcji i pętla się zamyka.

Ramy odniesienia i układy współrzędnych

- **world** – bezwzględna ramka inercjalna, stała w przestrzeni
- **map** – ramka mapy globalnej; związana z world przez transformację SLAM
- **odom** – ramka odometrii; ciągła, ale kumuluje dryft w długim horyzoncie
- **base** – środek ciężkości lub pelvis robota; baza kinematyki odwrotnej
- **foot_L / foot_R** – ramki stóp; kluczowe podczas fazy podparcia
- **camera** – ramka optyczna kamery; kalibracja extrinsic względem base
- **lidar** – ramka skanera; wymaga precyzyjnej kalibracji kąta i przesunięcia
- **Drzewo transformacji TF (ROS):** spójna hierarchia wszystkich układów

W nawigacji musimy bardzo precyzyjnie rozróżniać układy współrzędnych. Ramka world jest inercjalna i stała w przestrzeni. Ramka map opisuje mapę globalną. Ramka odom jest ciągła, ale z czasem akumuluje dryft. Ramka base jest związana z miednicą albo środkiem ciężkości robota. Do tego dochodzą ramki stóp, kamery i LiDAR-u. Całość musi być spięta spójnym drzewem transformacji TF, bo bez tego nawet bardzo dobre algorytmy zaczną produkować błędne wyniki.

Ruch całego ciała a nawigacja bazowa

Nawigacja bazowa (CoM / pelvis)

- Trajektoria środka masy w przestrzeni 3D
- Pozycja miednicy i orientacja tułowia
- Globalny kierunek i prędkość przemieszania
- Wejście do globalnego i lokalnego planisty

Ruch kończyn (whole-body)

- Sekwencja ustawień stóp i czas kontaktu
- Trajektorie końcówek roboczych rąk
- Ruchy kompensacyjne tułowia dla równowagi
- Wyjście kontrolera chodu do aktuatorów

Warto odróżnić nawigację bazową od ruchu całego ciała. W nawigacji bazowej interesuje nas przede wszystkim trajektoria środka masy, pozycja miednicy, orientacja tułowia, globalny kierunek ruchu i prędkość przemieszczania. Natomiast ruch kończyn obejmuje sekwencję ustawień stóp, trajektorie rąk oraz ruchy kompensacyjne tułowia niezbędne do utrzymania równowagi. Innymi słowy: planista myśli o tym, dokąd robot ma iść, a kontroler całego ciała myśli o tym, jak dokładnie ma to wykonać.

Miary jakości nawigacji

Dokładność lokalizacji

RMSE pozycji [m], błąd orientacji [°]

Czas dotarcia do celu

Czas od startu do osiągnięcia punktu docelowego [s]

Zużycie energii

Sumaryczny pobór mocy aktuatorów [Wh/m]

Liczba korekt kroku

Ponowne planowanie stóp, wskaźnik reaktywności

Liczba kolizji

Kontakt z przeszkodą / upadek w trakcie misji

Zapas stabilności

Marginesy ZMP i Capture Point względem granicy

Jakość nawigacji mierzymy wieloma wskaźnikami jednocześnie. Interesuje nas dokładność lokalizacji, czyli na przykład RMSE pozycji i błąd orientacji. Interesuje nas czas dotarcia do celu, zużycie energii, liczba korekt kroku, liczba kolizji oraz zapas stabilności, wyrażony choćby przez marginesy ZMP i Capture Point. Dobra nawigacja humanoida nie jest tylko szybka — ona musi być jednocześnie dokładna, bezpieczna i stabilna.

Środowiska pracy humanoida

- **Korytarze biurowe** – wąskie przejścia, dynamiczne przeszkody (ludzie), równe podłoże
- **Schody i rampy** – zmienne wysokości, konieczność planowania kroków 3D
- **Drzwi i progi** – precyzyjny ruch chwytnych, klamki, siłowniki, progi podłogowe
- **Nierówne podłoże** – gruz, kostka brukowa, miękkie nawierzchnie, pochyłości terenu
- **Otoczenie ludzkie** – przewidywanie ruchu, strefa bezpieczeństwa, protokoły interakcji
- **Środowiska zmieniające się dynamicznie** – przesuwane meble, otwierane drzwi, ruchome obiekty

Humanoid pracuje w środowiskach znacznie trudniejszych niż klasyczny robot mobilny. W korytarzach biurowych ma wąskie przejścia i ludzi jako przeszkody dynamiczne. Na schodach i rampach musi uwzględniać wysokość terenu. Przy drzwiach i progach potrzebuje precyzji ruchu oraz często manipulacji. Na nierównym podłożu dochodzi problem kontaktu stopy z gruntem. W otoczeniu ludzkim pojawia się kwestia bezpieczeństwa i przewidywania ruchu człowieka, a w środowiskach dynamicznych mapa może się zmieniać praktycznie cały czas.

Studium przypadku: robot w budynku biurowym

- **Misja**: przemieszczenie z punktu A (wejście) do punktu B (sala konf.) w budynku 3-piętrowym
- **Etap 1** – **Lokalizacja startowa**: inicjalizacja SLAM, wczytanie mapy budynku
- **Etap 2** – **Planowanie globalne**: wyznaczenie trasy A* z uwzględnieniem wind lub schodów
- **Etap 3** – **Nawigacja korytarzem**: DWA z detekcją ludzi i dynamicznym omijaniem
- **Etap 4** – **Przejście przez drzwi**: detekcja klamki, planowanie kroków przy progu
- **Etap 5** – **Weryfikacja celu**: potwierdzenie lokalizacji względem mapy semantycznej
- **Kluczowe wyzwania**: synchronizacja sensorów, obsługa wind, interakcja z ludźmi

Rozważmy prosty przypadek: robot ma przejść od wejścia do sali konferencyjnej w trzypiętrowym biurze. Najpierw musi ustalić pozycję startową i zainicjalizować SLAM. Potem planuje globalną trasę z uwzględnieniem schodów albo windy. Następnie porusza się korytarzem, omijając ludzi i inne przeszkody dynamiczne. Gdy napotyka drzwi, musi wykryć klamkę, próg i zaplanować kroki. Na końcu weryfikuje osiągnięcie celu względem mapy semantycznej. Ten przykład pokazuje, że w praktyce nie ma jednego problemu nawigacji, jest cały łańcuch problemów połączonych w jeden system.

KOD PYTHON

Struktura modułów systemu nawigacji

```python
class SensorManager:
    def __init__(self):
        self.imu_data = None
        self.lidar_scan = None
        self.depth_frame = None
    def update(self, imu, lidar, depth):
        self.imu_data, self.lidar_scan, self.depth_frame = imu, lidar, depth

class StateEstimator:
    def estimate(self, sensors: SensorManager) -> dict:
        return {'pos': [0.0, 0.0, 0.0], 'yaw': 0.0, 'contact': [True, False]}

class Planner:
    def plan(self, state: dict, goal: list) -> list:
        return [] # lista kroków (footsteps)

class GaitController:
    def execute(self, footsteps: list) -> list:
        return [] # komendy stawowe [Nm]
```

Opis: Szkielet architektury oprogramowania – cztery główne klasy systemu nawigacji humanoida.

02

SEKCJA

Sensory i percepcja



Charakterystyka sensorów, propriocepcja, percepcja wizyjna, LiDAR i filtracja danych

Sensory stosowane w humanoidzie

IMU

Żyroskop + akcelerometr; orientacja i prędkość kątowa, f = 100-1000 Hz

Enkodery

Kąty stawów, prędkości; f = 100-500 Hz, wysoka rozdzielczość

Kamera RGB

Obraz kolorowy; detekcja obiektów, śledzenie, rozpoznawanie sceny

Kamera RGB-D

Obraz + głębia (structured light / ToF); 30-90 Hz, zasięg 0,3-5 m

LiDAR

Chmura punktów 2D/3D; dokładność ~2 cm, zasięg do 100m

Czujniki sił

Siły i momenty w stopach (6-DoF F/T); detekcja kontaktu i fazy chodu

Humanoid korzysta z kilku klas sensorów jednocześnie. IMU daje orientację i prędkość kątową przy bardzo dużej częstotliwości. Enkodery dostarczają kąty i prędkości stawów. Kamera RGB umożliwia detekcję obiektów i rozpoznawanie sceny. Kamera RGB-D dodaje do obrazu głębię, co jest bardzo użyteczne przy nawigacji w pomieszczeniach. LiDAR daje precyzyjną geometrię otoczenia, a czujniki sił w stopach pozwalają wykrywać kontakt i fazę chodu. Każdy z tych sensorów widzi inny fragment rzeczywistości, dlatego pojedynczy sensor prawie nigdy nie wystarcza.

Kluczowe parametry sensorów

- **Zasięg**: od kilku cm (kamery RGB-D) do 100 m (LiDAR zewnętrzny)
- **Częstotliwość**: IMU 200-1000 Hz, kamery 30-90 Hz, LiDAR 10-20 Hz
- **Szum i dryft**: IMU dryfuje ~1°/min; kamera wrażliwa na oświetlenie
- **Opóźnienie** (latencja): kamera 30-100 ms; IMU < 5 ms – krytyczne dla stabilności
- **Koszt obliczeniowy**: przetwarzanie chmur punktów i segmentacja semantyczna to ~80% CPU
- **Odporność środowiskowa**: LiDAR wrażliwy na kurz, kamera na silne kontrastowe oświetlenie
- **Kalibracja extrinsic**: każdy sensor wymaga precyzyjnego wyznaczenia pozycji względem base

Przy doborze sensorów patrzymy na kilka parametrów. Po pierwsze zasięg, bo kamera głębi pracuje na krótkich dystansach, a LiDAR może sięgać dziesiątek metrów. Po drugie częstotliwość, bo IMU pracuje dużo szybciej niż kamera czy LiDAR. Po trzecie szum i dryft, które są krytyczne dla estymacji. Dalej mamy latencję, koszt obliczeniowy, odporność na warunki środowiskowe i wreszcie kalibrację extrinsic, czyli dokładne położenie każdego sensora względem bazy robota. Sam zakup sensora nie rozwiązuje problemu, równie ważna jest jego poprawna integracja.

IMU i propriocepcja

- **Żyroskop**: pomiar prędkości kątowej [rad/s] – integracja daje orientację, ale dryfuje
- **Akcelerometr**: siły inercjalne [m/s^2] – wyznaczenie kierunku grawitacji = kąt nachylenia
- **Enkodery stawowe**: konfiguracja ciała q(t); wejście do kinematyki prostej i odwrotnej
- **Kontakt stóp** (czujniki F/T): detekcja fazy podparcia – kluczowe dla odometrii humanoida
- **Propriocepcja** to minimum informacji do zamkniętej pętli równowagi bez zewnętrznych sensorów
- **Filtr komplementarny / EKF** łączy szybki żyroskop z wolnym, ale niedryfującym akcelerometrem

Propriocepcja to ta część percepcji, która mówi robotowi o nim samym. Żyroskop mierzy prędkość kątową, akcelerometr dostarcza informacji o grawitacji i przyspieszeniach, a enkodery stawowe dają konfigurację ciała w czasie. Czujniki sił w stopach pozwalają określić, czy stopa jest w kontakcie z podłożem. Te informacje są absolutnie kluczowe dla utrzymania równowagi i odometrii humanoida. Zwykle łączymy szybki, ale dryfujący żyroskop z wolniejszym, ale długoterminowo stabilnym akcelerometrem za pomocą filtru komplementarnego albo EKF.

Percepcja wizyjna 2D i 3D

Percepcja 2D (obraz RGB)

- **Detekcja cech** (ORB, SIFT): punkty orientacyjne do lokalizacji
- **Detekcja obiektów** (YOLO, SSD): ludzie, drzwi, schody
- **Segmentacja semantyczna:** klasy terenu (podłoga, ściana, przeszkoda)
- **Śledzenie** (KLT optical flow): ruch obiektów w scenie

Percepcja 3D (RGB-D / stereo)

- **Mapa głębi:** odległość do każdego piksela
- **Rekonstrukcja chmury punktów** otoczenia
- **Detekcja przeszkód w 3D**: weryfikacja wolnego przejścia
- **Szacowanie normalnych powierzchni**: kąt nachylenia terenu (wektory normalne)

W percepcji wizyjnej rozróżniamy podejście 2D i 3D. W obrazie RGB możemy wykrywać cechy, obiekty, klasy semantyczne i śledzić ruch w scenie. W percepcji 3D, na przykład z RGB-D albo stereo, dostajemy głębię, rekonstrukcję chmury punktów, detekcję przeszkód przestrzennych i możliwość estymacji nachylenia powierzchni. Dla humanoida to istotne, bo on nie pyta tylko, czy coś stoi przed nim, ale też czy na danym fragmencie terenu można bezpiecznie postawić stopę.

LiDAR i chmury punktów

- **Zasada działania**: pomiar czasu przelotu impulsu laserowego (ToF) lub przesunięcia fazowego
- **Konfiguracje**: LiDAR 2D (skaner obrotowy), LiDAR 3D (wielowiązkowy, np. Velodyne 16-64)
- **Typowe parametry**: zasięg 5-100 m, kąt 360°, rozdzielczość kątowa 0,1-0,4°, f = 10-20 Hz
- **Reprezentacja**: chmura punktów {(x, y, z, intensity)} – surowy format PCL
- **Zalety w środowisku wewnętrznym**: niezależność od oświetlenia, precyzja geometryczna
- **Ograniczenia**: brak informacji o kolorze i teksturze; problemy z lustrami i przezroczystymi powierzchniami
- **Zastosowanie**: budowa mapy 2D/3D, lokalizacja ICP, detekcja przeszkód i wolnej przestrzeni

LiDAR jest jednym z najważniejszych sensorów geometrycznych. Działa na zasadzie pomiaru czasu przelotu albo przesunięcia fazowego impulsu laserowego. Może być 2D lub 3D, ma duży zasięg, pełne pole widzenia i dobrą dokładność geometryczną. Jego ogromną zaletą w pomieszczeniach jest niezależność od oświetlenia. Ograniczenia też są istotne: LiDAR nie daje koloru ani tekstury, ma problemy z lustrami i przezroczystymi powierzchniami, ale mimo to świetnie nadaje się do budowy map, lokalizacji, detekcji przeszkód i wyznaczania wolnej przestrzeni.

Detekcja przeszkód statycznych i dynamicznych

Przeszkody statyczne

- Ściany, meble, kolumny – stałe w czasie
- Budowa globalnej mapy zajętości
- Klasyfikacja: wolna przestrzeń / zajęte / nieznane
- Wystarczy pojedynczy skan do detekcji

Przeszkody dynamiczne

- Ludzie, wózki, inne roboty – ruch w czasie
- Wymagana filtracja czasowa (ostatnie N skanów)
- Predykcja trajektorii (Kalman, LSTM)
- Strefa bezpieczeństwa zależna od prędkości obiektu

Musimy odróżniać przeszkody statyczne od dynamicznych. Ściany, meble czy kolumny są względnie stałe i nadają się do budowy mapy globalnej. Wystarczy często pojedynczy skan, aby je poprawnie sklasyfikować. Natomiast ludzie, wózki czy inne roboty poruszają się w czasie, więc potrzebujemy filtracji czasowej, śledzenia i często predykcji trajektorii. W przypadku obiektów dynamicznych strefa bezpieczeństwa powinna zależeć od ich prędkości i przewidywanego ruchu.

Filtracja danych sensorycznych

- **Filtr medianowy**: usuwa impulsowe błędy *(np. refleksy lustrzane w skanerze LiDAR)*
- **Średnia ruchoma** *(moving average):* wygładzanie szumowych serii pomiarowych
- **Progowanie odległości**: odrzucenie pomiarów poza zakresem [d_min, d_max] sensora
- **Odrzucanie wartości odstających** *(outlier removal):* statystyczne odchylenie od sąsiedztwa
- **RANSAC**: dopasowanie płaszczyzny podłogi przy silnym szumie i dużej liczbie punktów
- **Voxel grid downsampling**: redukcja gęstości chmury punktów przed przetwarzaniem

Surowe dane sensoryczne niemal nigdy nie nadają się do bezpośredniego użycia. Stosujemy więc filtr medianowy, średnią ruchomą, progowanie odległości, odrzucanie obserwacji odstających, RANSAC do dopasowania płaszczyzny podłogi oraz voxel grid downsampling do redukcji rozmiaru chmury punktów. Celem nie jest upiększenie danych, tylko uzyskanie takiego sygnału, na którym estymacja i planowanie będą działały stabilnie i przewidywalnie.

KOD PYTHON

Filtrowanie danych z LiDAR-a

```python
import numpy as np

def filter_lidar_scan(ranges: list,
                      min_dist: float = 0.15,
                      max_dist: float = 10.0,
                      kernel: int = 5) -> np.ndarray:
    """Oczyszczanie skanu LiDAR: próg + filtr medianowy."""
    arr = np.array(ranges, dtype=float)
    # 1. Zamień wartości poza zakresem sensora na NaN
    arr[(arr < min_dist) | (arr > max_dist)] = np.nan
    # 2. Filtr medianowy – usuwa impulsowe błędy refleksji
    valid_mask = ~np.isnan(arr)
    indices = np.where(valid_mask)[0]
    for idx in indices:
        neighbors = arr[max(0, idx-kernel//2) : idx+kernel//2+1]
        arr[idx] = np.nanmedian(neighbors)
    return arr
```

Opis: Oczyszczanie skanu: progowanie zasięgu i medianowe usuwanie błędnych odbić.

KOD PYTHON

Detekcja przeszkody z kamery głębi

```python
import numpy as np

def detect_obstacle(depth_image: np.ndarray,
                    roi_top: int = 100, roi_bottom: int = 300,
                    roi_left: int = 200, roi_right: int = 440,
                    safety_dist: float = 0.6) -> bool:
    """Sprawdz, czy w obszarze ROI jest przeszkoda blizej niz safety_dist [m]."""
    region = depth_image[roi_top:roi_bottom, roi_left:roi_right]
    # Pomijamy bledne piksele (0 = brak pomiaru)
    valid = region[(region > 0) & (region < 8000)]
    if valid.size == 0:
        return False  # Brak danych – brak kolizji
    min_depth_m = valid.min() / 1000.0  # mm -> m
    collision_alarm = min_depth_m < safety_dist
    return collision_alarm
```

Opis: ROI to centralny kadr obrazu głębi; wartości w mm konwertowane na metry.

Synchronizacja czasowa sensorów

1. **Problem**: IMU działa z f = 200 Hz, kamera 30 Hz, LiDAR 10 Hz – różne timestampy
2. **Każdy sensor powinien dołączać znacznik czasu** (ROS: Header.stamp) przy akwizycji
3. **Buforowanie wiadomości**: kolejka FIFO z timestampami dla każdego sensora
4. **Interpolacja liniowa**: wyznaczenie stanu sensora A w chwili pomiaru sensora B
5. **Synchronizator aproksymacyjny** (ROS: message_filters): dobór pomiarów w oknie czasowym
6. **Efekt braku synchronizacji**: błędy fuzji i duchy przeszkód w chmurze punktów

Bardzo ważna jest synchronizacja czasowa sensorów. IMU może pracować z częstotliwością 200 herców, kamera z 30, a LiDAR z 10, więc jeśli nie będziemy pilnować znaczników czasu, to zaczniemy łączyć pomiary pochodzące z różnych chwil. W praktyce każdy sensor powinien publikować timestamp przy akwizycji, dane powinny być buforowane, a brakujące chwile odtwarzane przez interpolację albo synchronizację aproksymacyjną. Bez tego w fuzji pojawią się błędy, a w chmurach punktów mogą powstawać tak zwane duchy przeszkód.

03

SEKCJA

Reprezentacja środowiska i mapowanie



Typy map, mapa 2D kontra 3D, koszt przejścia, mapowanie lokalne i globalne

Reprezentacje map środowiska

Occupancy Grid

Siatka 2D/3D komórek: zajęte/wolne/nieznane; prostota, wysoka pamięć

Voxel Map

Siatka 3D wokseli (np. OctoMap); kompresja drzewem oktalnym

Point Cloud Map

Nieskompresowana chmura punktów; wysoka wierność, duże wymagania RAM

Topological Map

Graf węzłów i krawędzi; szybkie planowanie, niska dokładność geometryczna

Semantic Map

Mapa wzbogacona o etykiety klas (podłoga, ściana, drzwi, stół)

Height Map

Mapa wysokości terenu (elevation map); kluczowa dla planowania kroków

Środowisko można reprezentować na różne sposoby. Occupancy grid to siatka komórek zajęte, wolne albo nieznane. Voxel map jest przestrzennym odpowiednikiem siatki 3D, często kompresowanym przez drzewo oktalne. Point cloud map daje wysoką wierność, ale jest pamięciożerna. Topological map upraszcza środowisko do grafu relacji, a semantic map dodaje etykiety znaczeniowe. Dla humanoida szczególnie ważna jest też height map, bo wysokość terenu bezpośrednio wpływa na planowanie kroków.

Mapa 2D kontra mapa 3D dla humanoida

Kiedy wystarczy mapa 2D

- Równe, płaskie podłoże (biuro, magazyn)
- Planowanie trasy i unikanie dużych przeszkód
- Mała pamięć i szybkie przetwarzanie
- Kompatybilność z klasycznym Nav Stack (ROS)
- Algorytmy A*, Dijkstra bezpośrednio na grid

Kiedy potrzebna jest mapa 3D

- Schody, rampy, progi: wymagana wysokość terenu
- Planowanie stóp z uwzględnieniem nierówności
- Wolna przestrzeń nad stopą (clearance)
- Wykrywanie nawisów i przeszkód górą
- Nawigacja po nierównym terenie zewnętrznym

Mapa 2D wystarcza wtedy, gdy środowisko jest płaskie i interesuje nas głównie wyznaczenie ścieżki oraz omijanie dużych przeszkód. Jest szybka, oszczędna pamięciowo i zgodna z klasycznym stosem nawigacyjnym. Natomiast mapa 3D jest konieczna, gdy pojawiają się schody, rampy, progi, nierówności, nawisy albo przeszkody nad stopą. Humanoid porusza się w przestrzeni, w której wysokość naprawdę ma znaczenie, więc w wielu zastosowaniach 2D okazuje się niewystarczające.

Koszt przejścia w środowisku humanoida

- **Teren łatwy**: niska wartość kosztu – szeroki, płaski, bez przeszkód w pobliżu
- **Teren wąski**: podwyższony koszt – ograniczone miejsce na krok i manewr tułowia
- **Teren pochyły**: koszt zależny od kąta nachylenia i kierunku ruchu (pod górę = więcej energii)
- **Teren śliski**: wysoki koszt lub blokada – niestabilny kontakt stóp, ryzyko poślizgu
- **Inflacja wokół przeszkód**: sąsiedztwo obiektu ma podwyższony koszt (gradient odpychający)
- **Funkcja kosztu**: C(x,y) = w_dist * d_obstacle + w_slope * slope + w_narrow * clearance_penalty
- **Cel**: preferowanie trasy bezpiecznej, nie tylko najkrótszej geometrycznie

Koszt przejścia nie może zależeć wyłącznie od długości trasy. Teren szeroki, płaski i daleki od przeszkód powinien mieć niski koszt. Teren wąski, pochyły albo śliski powinien mieć koszt wyższy lub wręcz być zablokowany. Dodatkowo stosujemy inflację wokół przeszkód, czyli gradient kosztu, który odpycha trajektorię od obiektów. W praktyce funkcja kosztu uwzględnia odległość od przeszkód, nachylenie terenu i karę za mały prześwit. Celem jest znalezienie trasy nie najkrótszej geometrycznie, lecz najbezpieczniejszej i najbardziej stabilnej.

KOD PYTHON

Budowa prostej mapy zajętości *(occupancy grid)*

```python
import numpy as np

def build_occupancy_grid(obstacle_points: list,
                         grid_size: int = 100,
                         resolution: float = 0.05) -> np.ndarray:
    """Tworzy siatke zajętości z listy punktów przeszkód [m]."""
    grid = np.zeros((grid_size, grid_size), dtype=np.uint8)
    origin_x = origin_y = grid_size // 2
    for (px, py) in obstacle_points:
        cell_x = int(origin_x + px / resolution)
        cell_y = int(origin_y + py / resolution)
        if 0 <= cell_x < grid_size and 0 <= cell_y < grid_size:
            grid[cell_y, cell_x] = 100  # zajete
    return grid
```

Opis: Siatka 100x100 komórek, rozdzielczość 5 cm/komórkę; wartość 100 = zajęte, 0 = wolne

KOD PYTHON

Obliczanie mapy kosztów z inflacją

```python
from scipy.ndimage import distance_transform_edt
import numpy as np

def compute_cost_map(grid: np.ndarray,
                     inflation_radius: float = 3.0,
                     max_cost: float = 100.0) -> np.ndarray:
    """Dodaje gradient kosztu wokół przeszkód (inflacja)."""
    obstacle_mask = (grid >= 100).astype(np.uint8)
    # Odleglosc euklidesowa kazdej komorki od najblizszej przeszkody
    dist = distance_transform_edt(1 - obstacle_mask)
    # Koszt: max dla przeszkody, maleje z odlegloscia
    cost = np.clip(max_cost * (1 - dist / inflation_radius), 0, max_cost)
    cost[obstacle_mask == 1] = max_cost
    return cost.astype(np.float32)
```

Opis: Inflacja tworzy gradient kosztu wokół przeszkód. Planista preferuje trasy z dala od obiektów

Mapowanie lokalne i globalne

Mapa lokalna *(robocza)*

- Budowana z **bieżących pomiarów sensorów**
- **Ograniczony zasięg**: ~5-10 m od robota
- **Wysoka aktualność**: uwzględnia obiekty dynamiczne
- Podstawa decyzji **planisty lokalnego**
- **Regularnie czyszczona** i przebudowywana

Mapa globalna *(trwała)*

- Budowana w **długim horyzoncie** (SLAM)
- **Zasięg**: cały budynek, piętro
- Wolniej aktualizowana, mniej dynamiczna
- Podstawa planowania **globalnej trasy A***
- Może zawierać **informacje semantyczne**

W praktyce prawie zawsze rozdzielamy mapowanie lokalne i globalne. Mapa lokalna ma mały zasięg, jest bardzo aktualna, uwzględnia obiekty dynamiczne i służy planowaniu lokalnemu. Mapa globalna obejmuje całe piętro albo budynek, jest aktualizowana wolniej i stanowi podstawę planowania trasy długodystansowej. Często zawiera również informacje semantyczne. To rozdzielenie jest kompromisem między aktualnością a skalą środowiska.

Ograniczenia mapowania w czasie rzeczywistym

1. **Pamięć**: mapa 3D OctoMap z rozdzielczością 5 cm dla biura 50x50 m to setki MB RAM
2. **Rozdzielczość, a wydajność**: wyższa rozdzielczość = dokładność wzrasta, czas przetwarzania rośnie
3. **Opóźnienia** *(latencja):* SLAM nie może działać wolniej niż cykl sterowania *(~100 ms)*
4. **Zmienność środowiska**: dynamiczne przeszkody nie mogą na stałe zaznaczać komórek jako zajętych
5. **Kompromis**: optymalny dobór rozdzielczości i zasięgu mapy do konkretnego zadania
6. **Strategie**: rozdzielona mapa lokalna *(wyzsza rozdzielczość)* i globalna *(niższa rozdzielczość)*

Mapowanie w czasie rzeczywistym ma poważne ograniczenia. Duże mapy 3D zużywają setki megabajtów pamięci, a zwiększanie rozdzielczości poprawia dokładność kosztem czasu obliczeń. Do tego SLAM nie może być wolniejszy niż pętla sterowania, a środowisko dynamiczne nie może być na stałe zapisywane jako zajęte. W praktyce trzeba dobrać kompromis między zasięgiem, rozdzielczością i aktualnością mapy, często stosując wyższą rozdzielczość lokalnie i niższą globalnie.

PODSUMOWANIE

01 Nawigacja humanoida wymaga integracji percepcji, lokalizacji, planowania i sterowania chodem.

02 Sensory (IMU, LiDAR, RGB-D) dostarczają komplementarnych danych o otoczeniu i stanie robota.

03 Filtracja i synchronizacja danych sensorycznych to warunek niezawodnej percepcji.

04 Reprezentacja mapy (siatka zajętosci, mapa wokselowa) warunkuje skuteczne planowanie.

05 Funkcja kosztu przejscia musi uwzgledniać stabilnosc, a nie tylko geometrie trasy.

Pytania problemowe i zadanie do analizy

1. Jakie sensory wybrałbyś dla humanoida patrolującego biuro w nocy (słabe oświetlenie)?
2. Dlaczego synchronizacja IMU z kamerą jest krytyczna dla odometrii wizyjno-inercyjnej?
3. Czym różni się occupancy grid od height map i kiedy potrzebujesz obu jednocześnie?
4. Jak zmodyfikować funkcję kosztu, aby robot unikał zawilgoconej posadzki w pobliżu toalety?

Zadanie

Zaprojektuj strukturę modułów nawigacji dla humanoida poruszającego się po schodach.

Jakich sensorów i jakich map potrzebujesz, których nie potrzebujesz na płaskiej podłodze?