NAWIGACJA ROBOTÓW HUMANOIDALNYCH

WYKŁAD 3

Planowanie ruchu i unikanie przeszkód

dr inż. Mateusz Pomianek



Planowanie globalne | Planowanie lokalne | Footstep planning | Kody Python

Plan wykładu

Sekcja 1

Planowanie globalne: przestrzeń stanów, Dijkstra, A*, heurystyki, cost map, wygładzanie

Sekcja 2

Planowanie lokalne: DWA, potential fields, MPC, przeszkody dynamiczne, replanowanie

Sekcja 3

Footstep planning: sekwencja kroków, ograniczenia kinematyczne, schody, trajektoria CoM

Kody Python

A* na grid map, A* z cost map, pole potencjału, ocena trajektorii, generacja kroków

W tym wykładzie przechodzimy przez trzy poziomy planowania ruchu robota humanoidalnego. Najpierw omawiamy planowanie globalne, czyli wyznaczenie trasy od pozycji startowej do celu na podstawie mapy. Następnie przechodzimy do planowania lokalnego, które odpowiada za bieżące omijanie przeszkód i reagowanie na zmiany w otoczeniu. Ostatnia część dotyczy planowania kroków, czyli przełożenia ścieżki geometrycznej na fizyczną sekwencję ustawień stóp humanoida. Kody w Pythonie pokazują uproszczone implementacje najważniejszych algorytmów.

01

SEKCJA

Planowanie globalne



Reprezentacje środowiska, Dijkstra, A, heurystyki i cost map*

Miejsce planowania w architekturze systemu

Wejście: mapa + lokalizacja

Planista dostaje mapę kosztów i bieżącą pozycję robota z estymatora stanu

Wejście: cel misji

Punkt docelowy lub sekwencja celów pośrednich z modułu zarządzania misją

Wyjście: ścieżka globalna

Ciąg punktów w przestrzeni (x,y) lub (x,y,z) przekazywany do planisty lokalnego

Wyjście: cost map

Mapa kosztów przesyłana do lokalnego planisty jako mapa naviagacyjna

Planowanie ruchu znajduje się pomiędzy percepcją, lokalizacją, zarządzaniem misją i kontrolą ruchu. Planista nie działa w izolacji: korzysta z mapy środowiska, aktualnej pozycji robota oraz celu misji. Wynikiem jego pracy jest ścieżka globalna oraz mapa kosztów, które są następnie przekazywane do planisty lokalnego. W praktycznym systemie humanoidalnym globalny plan określa kierunek działania, a lokalny planista decyduje, jak bezpiecznie wykonać ruch w danym momencie.

Problem planowania ścieżki

- **Przestrzeń stanów**: zbiór wszystkich dopuszczalnych konfiguracji robota $C_{free} \subseteq C$
  - **Stan początkowy** q_{start} : aktualna pozycja i orientacja robota w mapie
  - **Stan docelowy** q_{goal} : żądana pozycja końcowa misji
- **Ograniczenia**: przeszkody, granice mapy, kinematyka, stabilność robota
- **Funkcja kosztu**: $c(q_{start}, q_{goal})$ minimalizowana wzdłuż ścieżki (czas, energia, ryzyko)
- **Ścieżka optymalna**: ciąg $\gamma : [0,1] \rightarrow C_{free}$ $taki, że$ $\gamma(0) = q_{start}, \gamma(1) = q_{goal}$
- **Zupełność**: algorytm znajdzie ścieżkę, jeśli istnieje; optymalność: znajdzie najlepszą

Problem planowania ścieżki można formalnie zapisać jako szukanie przejścia ze stanu początkowego do stanu docelowego w przestrzeni dopuszczalnych konfiguracji. Dla humanoida przestrzeń ta jest ograniczona nie tylko przez przeszkody geometryczne, ale również przez kinematykę, stabilność i możliwość wykonania kroku. Funkcja kosztu określa, co uznajemy za „dobrą” ścieżkę: może to być najkrótszy czas, najniższa energia, największy dystans od przeszkód albo najmniejsze ryzyko utraty równowagi. Ważne pojęcia to zupełność i optymalność algorytmu.

Reprezentacje grafowe środowiska

Siatka komórek (Grid)

- Regularna siatka N×M komórek
- Komórka: wolna / zajęta / koszt
- Łatwa implementacja A*
- Duże mapy = wysoka pamięć
- Rozdzielczość ogranicza precyzję

Graf topologiczny

- Węzły = charakterystyczne miejsca
- Krawędzie = przejazdy między nimi
- Szybkie planowanie wysokopoziomowe
- Niska rozdzielczość geometryczna
- Wymaga mapowania semantycznego

Visibility/Roadmap

- Graf widzialności krawędzi przeszkód
- Trasy wzdłuż wklęsłości przeszkód
- Optymalne w 2D dla znanych map
- Trudne do uogólnienia na 3D
- Rzadko stosowane dla humanoida

Środowisko można reprezentować na kilka sposobów, a wybór reprezentacji wpływa na szybkość i jakość planowania. Siatka komórek jest najprostsza i dobrze współpracuje z A*, ale przy dużych mapach zużywa dużo pamięci. Graf topologiczny dobrze nadaje się do planowania wysokopoziomowego, na przykład przejścia między pomieszczeniami, ale traci szczegóły geometryczne. Graf widzialności i roadmapy są użyteczne w znanych mapach 2D, jednak dla humanoidów często okazują się zbyt uproszczone, ponieważ nie uwzględniają stabilności i ograniczeń kroków.

Algorytm Dijkstry

Pseudokod Dijkstry

```
1.  dist[start] = 0;  dist[v] = ∞ dla pozostałych
2.  priority_queue Q: (dist, vertex)
3.  Q.push((0, start))
4.  while Q nie jest pusta:
5.      (d, u) = Q.pop_min()
6.      if d > dist[u]: continue
7.      for (v, w) in sąsiedzi(u):
8.          nowy_koszt = dist[u] + w
9.          if nowy_koszt < dist[v]:
10.             dist[v] = nowy_koszt
11.             poprzednik[v] = u
12.             Q.push((nowy_koszt, v))
13. odtwórz ścieżkę przez poprzednik[]
```

Właściwości

- Gwarantuje optymalność
- **Brak heurystyki**, eksploruje we wszystkich kierunkach
- **Złożoność**: O((V+E) log V)
- **Powolny** dla dużych map
- Dobry jako punkt odniesienia
- **Podstawa** dla A* i Bellmana-Forda

Algorytm Dijkstry wyznacza najkrótszą ścieżkę w grafie z nieujemnymi kosztami krawędzi. Jego zaletą jest gwarancja optymalności, ale wadą brak kierunkowania poszukiwań w stronę celu. Oznacza to, że algorytm eksploruje wiele węzłów, które niekoniecznie prowadzą do rozwiązania. Dijkstra jest dobrym punktem odniesienia przy testowaniu innych algorytmów, takich jak A*, oraz przy budowie map typu cost-to-go. W robotyce mobilnej używa się go głównie wtedy, gdy mapa jest mała albo gdy nie mamy jednej konkretnej pozycji docelowej.

Algorytm A* - planowanie z heurystyką

Pseudokod A*

```
f(n) = g(n) + h(n)
  g(n) – koszt dotarcia od startu do n
  h(n) – heurystyka: szacunek do celu

1.  open_set = {start}
2.  g[start] = 0;  f[start] = h(start)
3.  while open_set nie jest pusta:
4.      u = węzeł z min f w open_set
5.      if u == goal: odtwórz ścieżkę
6.      open_set.remove(u)
7.      for v in sąsiedzi(u):
8.          tentative_g = g[u] + cost(u,v)
9.          if tentative_g < g[v]:
10.             g[v] = tentative_g
11.             f[v] = g[v] + h(v)
12.             parent[v] = u
```

Właściwości

- Optymalny, gdy h admissible
- **Szybszy** od Dijkstry dzięki h
- **h euklidesowa**: $\sqrt{(dx^2+dy^2)}$
- **h Manhattan**: |dx|+|dy|
- Im lepsza h, tym mniej węzłów
- **Złożoność**: O(b^d), b=branching

A* można traktować jako rozwinięcie algorytmu Dijkstry o heurystykę, czyli oszacowanie odległości do celu. Funkcja kosztu składa się z dwóch części: kosztu dotarcia do danego węzła oraz przewidywanego kosztu dojścia do celu. Dzięki temu algorytm przeszukuje graf bardziej selektywnie i zwykle działa znacznie szybciej. Kluczowa jest jakość heurystyki: jeśli nie przeszacowuje kosztu, A* zachowuje optymalność. Dla siatek 4-kierunkowych typowo stosuje się metrykę Manhattan, a dla ruchu bardziej ciągłego metrykę euklidesową.

KOD PYTHON

Algorytm Dijkstry: implementacja na siatce 2D

Kompletna implementacja z kolejką priorytetową i odtwarzaniem ścieżki

```python
import heapq
import numpy as np
from typing import Optional

def dijkstra(grid: np.ndarray, start: tuple,
             goal: tuple) -> Optional[list]:
    """
    Dijkstra na siatce 2D (ruch 8-kierunkowy).
    grid[r][c] = 0: wolne, 1: zajęte.
    Zwraca listę komórek lub None, gdy brak ścieżki.
    """
    rows, cols = grid.shape
    # Koszt przejścia: prosty = 1.0, skos = sqrt(2)
    MOVES = [(-1,0,1.0),(1,0,1.0),(0,-1,1.0),(0,1,1.0),
             (-1,-1,1.414),(-1,1,1.414),(1,-
1,1.414),(1,1,1.414)]

    dist = {start: 0.0}; prev = {start: None}
    pq = [(0.0, start)] # (koszt, wierzchołek)

    while pq:
        d, u = heapq.heappop(pq)
        if d > dist.get(u, float('inf')):
            continue
        if u == goal:
            # Odtworzenie ścieżki
            path, node = [], u
            while node is not None:
                path.append(node); node = prev[node]
            return path[::-1]
        r, c = u
        for dr, dc, cost in MOVES:
            v = (r + dr, c + dc)
            if 0 <= v[0] < rows and 0 <= v[1] < cols and grid[v] == 0:
                nd = d + cost
                if nd < dist.get(v, float('inf')):
                    dist[v] = nd; prev[v] = u
                    heapq.heappush(pq, (nd, v))
    return None   # brak ścieżki
```

Dijkstra, a A*

Dijkstra eksploruje wszystkie wierzchołki do celu; brak heurystyki.

Kiedy używać Dijkstry:
- Mapa bez wyraźnego celu (Multi-target search)
- Weryfikacja optymalności
- Budowa cost-to-go mapy
- Mapy < 200x200 komórek

Złożoność czasowa:
$O((V+E) \cdot \log V)$

Ruch 8-kier. koszt skosu:
$\sqrt{2} \approx 1.414$ (metryczny)

A* z h=0 === Dijkstra

8-kierunkowy Dijkstra; heapq jako min-heap; O((V+E) log V); nie wymaga heurystyki.

KOD PYTHON

A* na mapie zajętości (grid map)

```python
import heapq, numpy as np

def astar(grid, start, goal):
    """A* na siatce 2D. grid[r][c]=0: wolne, 1: zajete."""
    rows, cols = grid.shape
    h = lambda r, c: abs(r-goal[0]) + abs(c-goal[1])
    g = {start: 0}
    parent = {start: None}
    pq = [(h(*start), start)]
    while pq:
        _, u = heapq.heappop(pq)
        if u == goal:
            path = []
            while u: path.append(u); u = parent[u]
            return path[::-1]
        for dr, dc in [(-1,0),(1,0),(0,-1),(0,1)]:
            v = (u[0]+dr, u[1]+dc)
            if 0<=v[0]<rows and 0<=v[1]<cols and grid[v]==0:
                ng = g[u] + 1
                if ng < g.get(v, 1e9):
                    g[v]=ng; parent[v]=u
                    heapq.heappush(pq,(ng+h(*v),v))
```

Opis: Manhattan h – dopuszczalna i spójna; 4-kierunkowa sąsiedztwo; zwraca listę komórek ścieżki

Właściwości i zastosowanie Heurystyki w A*

Heurystyka Manhattan (L1)

- $h(n) = |dx| + |dy|$
- Idealna dla siatek z ruchem 4-kierunkowym
- **Dopuszczalna**: nigdy nie przeszacowuje kosztu
- **Bardzo szybka:** prosta operacja arytmetyczna
- **Dla ruchu 8-kierunkowego**: lekko nieprecyzyjna

Heurystyka euklidesowa (L2)

- $h(n) = \sqrt{dx^2 + dy^2}$
- Odpowiednia dla ciągłego ruchu w 2D/3D
- Dopuszczalna dla humanoida (metryczny koszt)
- **Droższa obliczeniowo** (sqrt), lecz precyzyjniejsza
- Zalecana gdy ścieżka jest wygładzana

Heurystyka wpływa bezpośrednio na liczbę sprawdzanych węzłów i czas działania A*. Heurystyka Manhattan jest bardzo szybka i dobrze pasuje do ruchu po siatce w czterech kierunkach, gdzie robot porusza się tylko poziomo i pionowo. Heurystyka euklidesowa lepiej opisuje rzeczywisty ruch robota w przestrzeni ciągłej, zwłaszcza gdy dopuszczamy ruch po skosie lub późniejsze wygładzanie ścieżki. W praktyce wybór heurystyki powinien być spójny z modelem ruchu robota i sposobem liczenia kosztu przejścia.

Planowanie z uwzględnieniem kosztów terenu

- **Cost map** zastępuje binarną mapę wolne/zajęte gradientem kosztów
- **Cel**: preferowanie bezpiecznych, szerokich przejść zamiast najkrótszej trasy geometrycznej
- **Warstwy kosztu**: inflacja wokół przeszkód, nachylenie terenu, wskaźnik stabilności, gęstość ruchu
- **Implementacja**: komórki siatki mają wartość $c \in [0, 255]$; A* używa $cost(u, v) = 1 + \frac{c(v)}{50}$
- **Efekt**: trasa omija obrzeża mebli i wąskie przejścia nawet gdy są geometrycznie wolne
- **Dynamiczna aktualizacja**: nowe obserwacje LiDAR/kamery zmieniają cost map online
- **Parametryzacja**: wagi warstw dobierane eksperymentalnie lub przez uczenie maszynowe

Cost map rozszerza klasyczną mapę zajętości o informację o jakości i ryzyku przejazdu przez dany obszar. Zamiast rozróżniać tylko pola wolne i zajęte, każda komórka otrzymuje koszt. Wysoki koszt może oznaczać bliskość przeszkody, śliską nawierzchnię, nachylenie terenu albo strefę o dużym natężeniu ruchu. Dzięki temu robot nie wybiera tylko najkrótszej trasy geometrycznej, lecz trasę bezpieczniejszą i bardziej komfortową. W humanoidach cost map ma szczególne znaczenie, ponieważ bliskość przeszkód zwiększa ryzyko kolizji kończyn i utraty stabilności.

KOD PYTHON

A* z mapą kosztów terenu

```python
import heapq, numpy as np

def astar_cost(grid, cost_map, start, goal):
    """A* z kosztem komorki z cost_map [0-100]."""
    rows, cols = grid.shape
    h = lambda r, c: np.hypot(r-goal[0], c-goal[1])  # euklidesowa
    g = {start: 0.0}
    parent = {start: None}
    pq = [(h(*start), start)]
    while pq:
        _, u = heapq.heappop(pq)
        if u == goal:
            path, node = [], u
            while node: path.append(node); node = parent[node]
            return path[::-1]
        for dr, dc in [(-1,0),(1,0),(0,-1),(0,1),(-1,-1),(1,1),(-1,1),(1,-1)]:
            v = (u[0]+dr, u[1]+dc)
            if 0<=v[0]<rows and 0<=v[1]<cols and grid[v]==0:
                step = np.hypot(dr,dc) * (1 + cost_map[v]/50.0)
                if (ng := g[u]+step) < g.get(v,1e9):
                    g[v]=ng; parent[v]=u
                    heapq.heappush(pq,(ng+h(*v),v))
```

Opis: 8-kierunkowe sąsiedztwo; koszt przejścia = odległość * (1 + wartość_kosztu/50); ruch po skosie droższy

Wygładzanie ścieżki globalnej

- A* na siatce zwraca ścieżkę z ostrymi załamaniami przy każdej zmianie kierunku
- **Wygładzanie**: konieczne przed przekazaniem ścieżki do generatora kroków lub MPC
- **Metoda 1, interpolacja B-spline**: krzywa przechodzi blisko punktów kontrolnych, zachowuje ciągłość krzywizny
- **Metoda 2, skracanie (shortcutting):** iteracyjne usuwanie pośrednich punktów, jeśli prosta jest wolna od przeszkód
- **Metoda 3, filtr Gaussa 1D**: ważona średnia po sąsiednich punktach wzdłuż ścieżki
- **Metryka**: kąt ugięcia < 15° między kolejnymi odcinkami jako warunek akceptacji
- **Ważne**: wygładzona ścieżka musi pozostawać w wolnej przestrzeni – weryfikacja kolizji po wygładzeniu

Ścieżka zwrócona przez A* na siatce zwykle składa się z odcinków łamanych i ostrych zakrętów. Taka trajektoria nie nadaje się bezpośrednio do wykonania przez robota humanoidalnego, ponieważ wymagałaby gwałtownych zmian kierunku. Wygładzanie pozwala uzyskać trasę bardziej naturalną dla generatora kroków i kontrolera ruchu. Można użyć interpolacji B-spline, skracania ścieżki lub filtracji. Po każdym wygładzeniu trzeba jednak ponownie sprawdzić kolizje, ponieważ wygładzona krzywa może wejść w obszar zajęty, mimo że oryginalna ścieżka była poprawna.

KOD PYTHON

Wygładzanie ścieżki: interpolacja B-spline

Usuwa ostre zakręty zwrócone przez A; zachowuje ciągłość krzywizny*

```python
import numpy as np
from scipy.interpolate import splprep, splev

def smooth_path_bspline(path: list,
                        num_points: int = 300,
                        smoothing: float = 0.0,
                        degree: int = 3) -> np.ndarray:
    """
    Wygładza ścieżkę A* interpolacja B-spline (scipy).
    path       – lista (row, col) lub (x, y) z planisty
    num_points – gęstość punktów wyjściowych
    smoothing  – 0 = przechodzi dokładnie przez punkty;
                 > 0 = aproksymacja (mniej załamania)
    degree     – stopień splajnu (3 = kubiczny)
    Zwraca ndarray shape (num_points, 2) w metrach lub komórkach.
    """
    pts = np.array(path, dtype=float)
    # Usuń zduplikowane kolejne punkty (splprep wymaga unikalnych)
    diffs = np.diff(pts, axis=0)
    mask  = np.any(diffs != 0, axis=1)
    pts   = np.vstack([pts[:-1][mask], pts[-1]])

    if len(pts) < degree + 1:
        return pts   # za mało punktów, zwróć oryginalną

    tck, _ = splprep(
        [pts[:, 0], pts[:, 1]],
        s=smoothing, k=degree
    )
    u_fine = np.linspace(0, 1, num_points)
    xs, ys = splev(u_fine, tck)
    return np.column_stack([xs, ys])


def check_path_collision(smooth_pts: np.ndarray,
                         grid: np.ndarray,
                         resolution: float = 0.05) -> bool:
    """Sprawdza, czy wygładzona ścieżka nie koliduje z siatką."""
    origin = grid.shape[0] // 2
    for x, y in smooth_pts:
        cx = int(origin + x / resolution)
        cy = int(origin + y / resolution)
        if 0 <= cx < grid.shape[1] and 0 <= cy < grid.shape[0]:
            if grid[cy, cx] > 50:   # zajete
                return True
    return False
```

Kiedy wygładzac?

Zawsze przed przekazaniem ścieżki do:
- footstep plannera
- generatora CoM
- kontrolera MPC

Parametr degree:
3 = kubiczny (zalecany)
5 = dla bardzo gładkich tras (koszt: oscylacje)

Smoothing s:
0.0 – interp. dokladna
0.5 – aproksymacja

Weryfikacja kolizji:
check_path_collision() po kazdym wygładzeniu!

splprep/splev z scipy; s=0 interpolacja dokładna; po wygładzeniu obowiązkowo sprawdź kolizje.

KOD PYTHON

Wygładzanie metodą skracania ścieżki (shortcutting)

Iteracyjne usuwanie zbędnych punktów pośrednich – bez bibliotek zewnętrznych

```python
import numpy as np

def shortcut_path(path: list,
                  grid: np.ndarray,
                  resolution: float = 0.05,
                  max_iter: int = 200) -> list:
    """
    Greedy shortcutting: jesli prosta miedzy i a j
    jest wolna od przeszkód, usuń punkty pomiędzy.
    path       – lista (row, col) po A*
    grid       – siatka zajętości (0=wolne, 1=zajete)
    resolution – metry/komorkę
    """
    pts = list(path)
    improved = True
    iteration = 0
    while improved and iteration < max_iter:
        improved = False; i = 0
        while i < len(pts) - 2:
            j = len(pts) - 1
            while j > i + 1:
                if _line_free(pts[i], pts[j], grid):
                    # Skrot: usun punkty i+1 .. j-1
                    pts = pts[:i+1] + pts[j:]
                    improved = True
                    break
                j -= 1
            i += 1
        iteration += 1
    return pts

def _line_free(p1: tuple, p2: tuple,
               grid: np.ndarray,
               steps: int = 40) -> bool:
    """Sprawdza czy odcinek p1-p2 jest wolny (Bresenham)"""
    r1, c1 = p1; r2, c2 = p2
    for t in np.linspace(0, 1, steps):
        r = int(round(r1 + t * (r2 - r1)))
        c = int(round(c1 + t * (c2 - c1)))
        if not (0 <= r < grid.shape[0] and
                0 <= c < grid.shape[1]):
            return False
        if grid[r, c] != 0:
            return False
    return True
```

Shortcutting vs B-spline

Shortcutting:
+ Gwarantuje brak kolizji
+ Działa bez scipy
+ Zachowuje punkty oryg.
- Kanciate zakręty
- O(n^2) per iteracja

B-spline:
+ Ciagla krzywa
+ Kontrola gladkosci

Zalecenie:
Shortcutting → B-spline → check_collision()
Obie metody kolejno!

_line_free sprawdza 40 punktow wzdluz odcinka; O(n^2) per iteracja ale szybkie dla n<500

02

SEKCJA

Planowanie lokalne i unikanie przeszkód



DWA, potential fields, MPC, przeszkody dynamiczne

Po co potrzebny planista lokalny?

Świat dynamiczny

Mapa globalna nie uwzględnia ludzi, wózków, otwieranych drzwi. Planista lokalny reaguje na bieżąco

Niedoskonała mapa

Sensory mają szum, mapa jest zbudowana ze starego skanu – lokalna mapa jest zawsze świeższa

Ograniczenia dynamiki

Robot nie może skręcić dowolnie ostro ani zatrzymać się natychmiast – kinematyka musi być uwzględniona

Horyzont predykcji

Decyzje na 1–3 s do przodu wystarczają; pełny replan jest zbyt kosztowny w każdym cyklu (10–50 Hz)

Planista globalny zakłada pewien obraz świata, ale rzeczywiste środowisko jest dynamiczne i niepewne. Ludzie mogą wejść na trasę, drzwi mogą się zamknąć, a sensory mogą wykryć przeszkodę, której nie było na mapie. Planista lokalny działa z większą częstotliwością i reaguje na bieżące dane sensoryczne. Jego zadaniem jest bezpieczne podążanie za ścieżką globalną, ale z uwzględnieniem aktualnych ograniczeń ruchu, dynamiki robota i przeszkód w najbliższym otoczeniu. Dla humanoida lokalne decyzje muszą dodatkowo respektować stabilność chodu.

Pola potencjału (Potential Fields)

Wzory sił

$U_{attr}(q) = \frac{1}{2} \cdot k_{att} \cdot d^2(q, q_{goal})$
– attrakcja do celu; rośnie z odległością

$U_{rep}(q) = \frac{1}{2} \cdot k_rep \cdot (1/d - 1/d_0)^2$
gdy $d < d_0$, inaczej 0 – odpychanie od przeszkód
d_0 – promień wpływu przeszkody

$F(q) = -\nabla U(q)$
$= F_{attr} + F_{rep}$

Ruch: $q_{new} = q + \alpha \cdot F(q)$
α – krok iteracji

Właściwości

- Prosta implementacja
- Działa w czasie rzeczywistym
- Problem: minima lokalne
- Oscylacje w wąskich przejściach
- Trudna parametryzacja k_{att}, k_{rep}
- **Rozwiązanie**: potencjały harmoniczne lub RRT

Metoda pól potencjału opisuje ruch robota jako wynik działania sił: przyciągania do celu i odpychania od przeszkód. Jest to intuicyjna i szybka metoda, dlatego nadaje się do prostych zadań lokalnej nawigacji. Jej ograniczeniem są minima lokalne, czyli sytuacje, w których suma sił się zeruje, mimo że robot nie osiągnął celu. Problem może wystąpić na przykład w narożnikach lub wąskich przejściach. W praktycznych systemach pola potencjału często łączy się z innymi metodami, takimi jak A*, RRT, DWA lub okresowe przeplanowywanie.

KOD PYTHON

Pole potencjału - lokalny wektor ruchu

```python
import numpy as np

def potential_field_move(robot_pos, goal_pos, obstacles,
                         k_att=1.0, k_rep=100.0, d0=1.5):
    """Wyznacza wektor ruchu z pol potencjalnych."""
    robot = np.array(robot_pos, dtype=float)
    goal  = np.array(goal_pos,  dtype=float)
    # Sila atrakcji do celu
    f_att = k_att * (goal - robot)
    # Suma sil odpychania od przeszkód
    f_rep = np.zeros(2)
    for obs in obstacles:
        diff = robot - np.array(obs)
        d    = np.linalg.norm(diff) + 1e-6
        if d < d0:
            f_rep += k_rep * (1/d - 1/d0) * diff / d**3
    total = f_att + f_rep
    return total / (np.linalg.norm(total) + 1e-6)  # wektor jednostkowy
```

Opis: Zwraca znormalizowany wektor ruchu; mnożymy przez dt i prędkość robota, by uzyskać przemieszczenie

KOD PYTHON

Ocena kandydatów trajektorii (uproszczone DWA)

```python
import numpy as np

def score_trajectory(v, omega, robot_state, goal, obstacles,
                     T=2.0, dt=0.1,
                     alpha=0.4, beta=0.4, gamma=0.2):
    """Ocenia trajektorie (v, omega) w horyzoncie T sekund."""
    x, y, yaw = robot_state
    min_clearance = 1e9
    for _ in range(int(T / dt)):
        x += v * np.cos(yaw) * dt
        y += v * np.sin(yaw) * dt
        yaw += omega * dt
        for obs in obstacles:
            min_clearance = min(min_clearance, np.hypot(x-obs[0], y-obs[1]))
    heading   = -abs(np.arctan2(goal[1]-y, goal[0]-x) - yaw)
    clearance = min_clearance
    return alpha*heading + beta*clearance + gamma*v
```

Opis: Wyższy score = lepsza trajektoria; wywoływane dla N par (v, ω), wybierana para z max score

Idea Dynamic Window Approach (DWA)

- **Idea**: w każdym kroku czasowym szukaj najlepszej $para\ prędkości\ (v, \omega)$ z okna osiągalnego
- **Okno dynamiczne**: $\{(v,\omega)|v \in [v - a_{\max} \cdot dt, v + a_{\max} \cdot dt],\ \omega\ analogicznie\}$
- **Ocena trajektorii**: dla każdego kandydata (v, ω) symuluj ruch przez T sekund (np. T=2 s)
- **Funkcja celu**: $G(v,\omega) = \alpha \cdot heading + \beta \cdot clearance + \gamma \cdot velocity$
- **heading**: wyrównanie kierunku ruchu z celem globalnym
- **clearance**: odległość od najbliższej przeszkody wzdłuż trajektorii
- **velocity**: nagroda za wyższą prędkość do przodu
- **wybierz** $(v*, \omega*) = \boldsymbol{argmax\ G}$ – wykonaj jeden krok, przelicz w kolejnym cyklu

Dynamic Window Approach wybiera najlepszą parę prędkości liniowej i kątowej, którą robot może osiągnąć w krótkim czasie, biorąc pod uwagę ograniczenia przyspieszenia. Dla każdej kandydackiej pary symulowana jest krótka trajektoria, a następnie oceniana funkcją celu. Funkcja ta premiuje zgodność z kierunkiem do celu, bezpieczny dystans od przeszkód oraz odpowiednią prędkość ruchu. DWA dobrze nadaje się do planowania lokalnego, ponieważ działa cyklicznie i może szybko reagować na nowe przeszkody. Dla humanoida trzeba dodatkowo filtrować trajektorie pod kątem stabilności.

Unikanie ludzi i przeszkód dynamicznych

Detekcja i predykcja

- **Detekcja ludzi**: YOLO, PointNet na chmurze LiDAR
- **Śledzenie** (tracking): filtr Kalmana per obiekt
- **Predykcja**: stałe velocity, Social Force Model
- **Horyzont predykcji**: 1–3 s wystarczy
- **Niepewność**: elipsa kowariancji wokół ścieżki

Planowanie z uwzględnieniem ruchu

- **Strefa bezpieczeństwa**: okrąg lub elipsa wokół człowieka
- **Prędkość** przeszkody skaluje rozmiar strefy
- **ORCA** (Optimal Reciprocal Collision Avoidance)
- **Social navigation**: min. 1,2 m od człowieka
- Zatrzymanie gdy człowiek zbliży się < 0,5 m

W środowisku z ludźmi samo wykrycie przeszkody nie wystarcza. Robot musi śledzić jej ruch i przewidywać, gdzie znajdzie się za kilka sekund. Typowy pipeline obejmuje detekcję człowieka, estymację pozycji i prędkości, predykcję trajektorii oraz wyznaczenie strefy bezpieczeństwa. W nawigacji społecznej ważne jest nie tylko uniknięcie kolizji, ale też zachowanie komfortowego dystansu. Robot powinien poruszać się przewidywalnie, nie przecinać agresywnie toru ruchu człowieka i zatrzymać się, gdy odległość spadnie poniżej krytycznego progu.

KOD PYTHON

Śledzenie pieszego: filtr Kalmana (model CV)

Model stałej prędkości (Constant Velocity) dla każdego wykrytego pieszego

```python
import numpy as np

class PedestrianKF:
    """
    Filtr Kalmana dla śledzenia pieszego – model CV.
    Stan: [x, y, vx, vy]   Pomiar: [x, y]
    """
    def __init__(self, x0: float, y0: float, dt: float = 0.1):
        self.dt = dt
        # Wektor stanu: pozycja + prędkość
        self.x = np.array([x0, y0, 0.0, 0.0])
        # Macierz kowariancji: wysokie P0 = duża niepewność
        self.P = np.diag([0.5, 0.5, 1.0, 1.0])
        # Macierz przejścia stanu (kinematyka CV)
        self.F = np.array([[1, 0, dt, 0],
                           [0, 1, 0, dt],
                           [0, 0, 1,  0],
                           [0, 0, 0,  1]])
        # Macierz obserwacji: obserwujemy tylko [x, y]
        self.H = np.array([[1, 0, 0, 0],
                           [0, 1, 0, 0]])
        # Szum procesu (przyspieszenie pieszego ~0.5 m/s^2)
        q = 0.5
        self.Q = q**2 * np.diag([dt**3/3, dt**3/3, dt, dt])
        # Szum pomiaru (LiDAR/kamera ~0.1 m)
        self.R = np.diag([0.1**2, 0.1**2])

    def predict(self):
        """Krok predykcji (przed nowym pomiarem)."""
        self.x = self.F @ self.x
        self.P = self.F @ self.P @ self.F.T + self.Q

    def update(self, z: np.ndarray):
        """Korekcja z pomiaru z = [x_obs, y_obs]."""
        innov = z - self.H @ self.x # innowacja
        # kowariancja innowacji
        S = self.H @ self.P @ self.H.T + self.R
        K = self.P @ self.H.T @ np.linalg.inv(S) # wzmocnienie
        self.x = self.x + K @ innov
        self.P = (np.eye(4) - K @ self.H) @ self.P

    def get_pos_vel(self):
        return self.x[:2], self.x[2:]
```

Model CV vs CA

Constant Velocity (CV)
- v = const między krokami
- Dokładny do ~1 s
- Prosty, szybki

Const. Acceleration
- Lepszy przy skretach
- Stan: [x,y,vx,vy,ax,ay]
- Większy koszt

Wieloosobowe śledzenie
- Jeden KF per obiekt
- IoU lub dist matching
- Nowy KF gdy nowy detektor
- Usuń gdy brak pomiaru
- przez > 0.5 s

Chi2 test innowacji
- Outlier detection
- NIS > chi2(0.95, 2) → skip

dt=0.1 s (10 Hz LiDAR); Q modeluje zmiany prędkości; R = szum detektora; predict()+update() co klatkę.

KOD PYTHON

Predykcja trajektorii i zarządzanie śledzonymi obiektami

MultiTracker: propagacja KF, dopasowanie detektorów, predykcja na H kroków

```python
import numpy as np
from typing import Dict, List

class MultiPedestrianTracker:
    """Śledzenie wielu pieszych z automatycznym tworzeniem i usuwaniem KF."""
    def __init__(self, dt=0.1, max_miss=5):
        self.trackers: Dict[int, dict] = {}   # id -> {kf, miss_count}
        self.dt = dt; self._next_id = 0
        self.max_miss = max_miss  # klatki bez pomiaru przed usunięciem

    def update(self, detections: List[np.ndarray]) -> Dict[int, np.ndarray]:
        """
        detections: lista pomiarów [x, y] (m) z bieżącej klatki.
        Zwraca słownik {id: stan [x,y,vx,vy]} dla każdego śledzonego.
        """
        # 1. Predykcja wszystkich KF
        for tid in self.trackers:
            self.trackers[tid]['kf'].predict()

        # 2. Proste dopasowanie: najbliższy pieszy (greedy)
        matched = set()
        for det in detections:
            best_id, best_d = None, float('inf')
            for tid, tr in self.trackers.items():
                if tid in matched: continue
                kf_pos = tr['kf'].x[:2]; d = np.linalg.norm(det - kf_pos)
                if d < best_d and d < 1.5: # próg 1.5 m
                    best_d, best_id = d, tid
            if best_id is not None:
                self.trackers[best_id]['kf'].update(det)
                self.trackers[best_id]['miss_count'] = 0
                matched.add(best_id)
            else:
                # Nowy obiekt – nowy KF
                from w3_extra_code import PedestrianKF  # importuj klasę KF
                new_kf = PedestrianKF(det[0], det[1], self.dt)
                self.trackers[self._next_id] = {'kf': new_kf, 'miss_count': 0}
                self._next_id += 1

        # 3. Zwiększ licznik braku pomiaru; usuń stare
        stale = [tid for tid, tr in self.trackers.items()
                 if tr['miss_count'] > self.max_miss]
        for tid in stale: del self.trackers[tid]
        for tid in self.trackers:
            if tid not in matched:
                self.trackers[tid]['miss_count'] += 1
        return {tid: tr['kf'].x for tid, tr in self.trackers.items()}

    def predict_trajectory(self, tid: int, horizon: int = 20) -> np.ndarray:
        """Predykcja H kroków do przodu dla obiektu tid (model CV)."""
        kf = self.trackers[tid]['kf']; pos = kf.x[:2].copy()
        vel = kf.x[2:].copy(); traj = [pos.copy()]
        for _ in range(horizon):
            pos = pos + vel * self.dt; traj.append(pos.copy())
        return np.array(traj)   # (H+1, 2)
```

predict_trajectory(): model CV na H kroków; dla lepszej predykcji zastąp Social Force lub LSTM

KOD PYTHON

Uproszczone ORCA: Optimal Reciprocal Collision Avoidance

Robot przyjmuje pełną odpowiedzialność za unikanie kolizji (pieszy nie stosuje ORCA)

```python
import numpy as np

def orca_velocity(robot_pos: np.ndarray, robot_vel: np.ndarray,
                  v_pref: np.ndarray, pedestrians: list,
                  tau: float = 2.0, r_robot: float = 0.35,
                  r_ped: float = 0.25, v_max: float = 1.0) -> np.ndarray:
    """
    Uproszczone ORCA dla robota wśród pieszych.
    Robot bierze PEŁNA odpowiedzialność (pieszy ignoruje ORCA).
    pedestrians: lista {'pos': [x,y], 'vel': [vx,vy]}
    tau: horyzont czasowy unikania [s]
    Zwraca skorygowana prędkość v_new.
    """
    # Zbierz half-plane'y ORCAi dla każdego pieszego
    constraints = [] # lista (p: punkt na half-plane, n: normalny do wewnątrz)

    for ped in pedestrians:
        rel_pos = np.array(ped['pos']) - robot_pos # od robota do pieszego
        rel_vel = robot_vel - np.array(ped['vel']) # względna prędkość
        combined_r = r_robot + r_ped    # suma promieni
        dist = np.linalg.norm(rel_pos) + 1e-9

        # Sprawdź, czy kolizja jest możliwa w horyzoncie tau
        if dist > combined_r + v_max * tau:
            continue   # zbyt daleko

        # Oblicz punkt na granicy strefy kolizji (uproszczone)
        n = rel_pos / dist # normalny od pieszego do robota
        u = n * combined_r / tau - rel_vel # wektor korekty prędkości

        # Half-plane: prędkość musi być po stronie u względem rel_vel
        constraints.append((rel_vel + u, n))   # (punkt, normalny)

    if not constraints:
        # Brak zagrożenia, idź z preferowana prędkością
        v_new = v_pref.copy()
    else:
        # Projekcja v_pref na bezpieczną prędkość (uproszczone: minimalna zmiana)
        v_new = v_pref.copy()
        for pt, n in constraints:
            # Jeśli v_new narusza half-plane -> skoryguj
            penetration = np.dot(v_new - pt, n)
            if penetration < 0:
                v_new = v_new - penetration * n

    # Ogranicz do v_max
    speed = np.linalg.norm(v_new)
    if speed > v_max: v_new = v_new / speed * v_max
    return v_new
```

tau=2.0 s: horyzont unikania; dla pełnego ORCA użyj biblioteki RVO2 (C++/Python binding)

Model Predictive Control, czyli taka „intuicja”

- **MPC** optymalizuje sekwencję sterowań $u_0, u_1, \dots, u_{\{N-1\}}$ na horyzoncie N kroków
- **Model predykcji**: $x_{\{t+1\}} = f(x_t, u_t)$ – uproszczony model kinematyczny/dynamiczny robota
- **Funkcja celu**: minimalizacja odchylenia od ścieżki + kosztu sterowania + penalizacja za przeszkody
- **Ograniczenia**: $|v| \leq v_{\max}, |\omega| \leq \omega_{\max}$, brak kolizji, stabilność ZMP
- **Receding horizon**: po wyznaczeniu sekwencji wykonuje się tylko u_0, a plan jest aktualizowany
- **Zalety**: jawne ograniczenia dynamiczne, optymalne zarządzanie kolizjami w horyzoncie
- **Wada**: wysoki koszt obliczeniowy (solver QP/NLP); wymaga dedykowanego CPU lub GPU

MPC planuje sterowanie w horyzoncie czasowym, czyli przewiduje kilka przyszłych kroków i wybiera taką sekwencję akcji, która minimalizuje określony koszt. W odróżnieniu od prostych metod reaktywnych MPC może jawnie uwzględniać ograniczenia prędkości, skrętu, kolizji i stabilności. W każdym cyklu wykonywany jest tylko pierwszy element zaplanowanej sekwencji, po czym cały proces jest liczony ponownie. To pozwala reagować na zmiany otoczenia. Wadą MPC jest koszt obliczeniowy oraz konieczność posiadania modelu robota i wydajnego solvera optymalizacyjnego.

KOD PYTHON

Uproszczony MPC: Model Predictive Control lokalny

Optymalizacja sekwencji sterowań na horyzoncie N kroków bez solvera QP

```python
import numpy as np

def mpc_local_planner(robot_state: np.ndarray,
                      ref_path: np.ndarray,
                      obstacles: list,
                      N: int = 10,
                      dt: float = 0.1,
                      v_max: float = 0.8,
                      omega_max: float = 1.0,
                      n_samples: int = 200) -> np.ndarray:
    """
    Uproszczony MPC oparty na probkowaniu trajektorii (CEM/MPPI lite).
    robot_state: [x, y, yaw]
    ref_path: ndarray (M, 2) – referencja geometryczna
    obstacles: lista {'pos': [x,y], 'r': float}
    Zwraca optymalny wektor akcji [v, omega] dla pierwszego kroku.
    """
    best_cost = float('inf')
    best_action = np.array([0.0, 0.0])

    for _ in range(n_samples):
        # Losuj stala sekwencje (v, omega) na caly horyzont (uproszczenie)
        v     = np.random.uniform(0.0, v_max)
        omega = np.random.uniform(-omega_max, omega_max)

        # Symuluj trajektorie
        x, y, yaw = robot_state
        cost = 0.0
        collision = False

        for k in range(N):
            x   += v * np.cos(yaw) * dt
            y   += v * np.sin(yaw) * dt
            yaw += omega * dt

            # Koszt: odleglosc od referencji (najblizszy punkt)
            dists_ref = np.linalg.norm(ref_path - np.array([x, y]),
                                       axis=1)
            cost += dists_ref.min()

            # Kara za przeszkody
            for obs in obstacles:
                d = np.linalg.norm(np.array([x,y]) - np.array(obs['pos']))
                if d < obs.get('r', 0.5) + 0.35:
                    cost += 200.0  # kara kolizji
                    collision = True

            # Nagroda za predkosc (do przodu)
            cost -= 0.3 * v * np.cos(yaw - robot_state[2])

        # Nie wybieraj trajektorii kolizyjnych jesli jest lepsza
        if not collision or best_cost == float('inf'):
            if cost < best_cost:
                best_cost   = cost
                best_action = np.array([v, omega])

    return best_action   # [v [m/s], omega [rad/s]]
```

n_samples=200 trajektorii × N=10 kroków; prawdziwy MPC uzywa solvera QP (osqp/casadi) z ograniczeniami ZMP

Kiedy i jak przeplanowywać online

Replan globalny (rzadki)

- Gdy cel misji zostaje zmieniony
- Gdy mapa zasadniczo się zmienia *(zamknięte przejście)*
- Co 5–30 sekund lub na żądanie operatora
- Kosztowny obliczeniowo – całkowite A*
- Alternatywa: D* Lite *(przyrostowy A*)*

Replan lokalny (częsty)

- Co 50–100 ms w każdym cyklu planisty
- Okno lokalne wokół bieżącej pozycji robota
- Reaguje na nowe przeszkody dynamiczne
- Możliwy na CPU embarked 1–2 ms
- Brak zmiany ścieżki globalnej

Przeplanowywanie może odbywać się na dwóch poziomach. Globalne przeplanowanie jest kosztowne i wykonuje się je rzadko, na przykład gdy cel się zmieni albo główne przejście zostanie zablokowane. Lokalne przeplanowanie działa znacznie częściej, nawet co kilkadziesiąt milisekund, i odpowiada za reakcje na przeszkody w pobliżu robota. W praktyce oba poziomy powinny współpracować: lokalny planista próbuje ominąć problem, a jeśli przeszkoda blokuje drogę przez dłuższy czas, system uruchamia nowy plan globalny.

Konflikt między planem, a stabilnością

- **Problem**: optymalny manewr unikania może wymagać szybkiego skrętu lub dużej prędkości bocznej
- **Konsekwencja**: środek masy wychodzi poza wielobok podparcia → utrata równowagi
- **Reguła nadrzędna**: stabilność > unikanie kolizji > optymalizacja trasy
- **Implementacja**: każda kandydatka trajektoria lokalnej jest filtrowana przez weryfikator ZMP
- **Weryfikator ZMP**: odrzuca trajektorie, gdzie przewidywany ZMP wychodzi poza margines bezpieczeństwa
- **Efekt**: planista lokalny preferuje wolniejsze, bardziej konserwatywne manewry w trudnym terenie
- **Parametr**: safety_margin_zmp = 2–4 cm; dla schodów i pochyłości zwiększony do 5–8 cm

Najkrótsza lub najszybsza trajektoria nie zawsze jest wykonalna dla humanoida. Robot dwunożny ma ograniczony wielobok podparcia i musi utrzymać środek masy oraz ZMP w bezpiecznych granicach. Gwałtowny skręt, duża prędkość boczna lub zbyt szeroki krok mogą doprowadzić do utraty równowagi. Dlatego w hierarchii decyzji stabilność ma wyższy priorytet niż optymalność trasy. Planista powinien odrzucać trajektorie, które są geometrycznie poprawne, ale niestabilne biomechanicznie lub dynamicznie.

03

SEKCJA

Planowanie kroków i trajektorii humanoida



Footstep planning, ograniczenia, schody, trajektoria CoM

Od ścieżki geometrycznej do sekwencji kroków

1

Ścieżka bazy

Globalna trasa (x,y) lub (x,y,z) wyznaczona przez A*; reprezentuje ruch CoM lub miednicy

2

Wzorzec chodu

Parametry: długość kroku l, szerokość w, czas fazy T_step, T_swing; regularne lub adaptacyjne

3

Wyznaczenie stóp

Dla każdego punktu ścieżki wyznacz cele stopy: offset lewo/prawo ±w/2, orientacja = yaw trasy

4

Weryfikacja

Sprawdź długość kroku, osiągalność kinematyczną, kontakt z terenem, margines ZMP i stabilność

Ścieżka globalna opisuje zwykle ruch punktu referencyjnego, na przykład środka masy, miednicy albo bazy robota. Humanoid nie może jednak po prostu „jechać” po tej ścieżce, ale musi postawić konkretne kroki. Footstep planner przekształca trasę w sekwencję pozycji lewej i prawej stopy. Uwzględnia długość kroku, szerokość rozstawu, orientację stóp i ograniczenia kontaktu z podłożem. Każdy krok musi zostać zweryfikowany pod kątem kinematyki, dostępności miejsca kontaktu oraz stabilności całego ciała.

KOD PYTHON

Weryfikacja wykonalności kroku

Zasięg nogi, zakres stawów, delta kąta, margines ZMP, strefa kontaktu

```python
import numpy as np
from dataclasses import dataclass

@dataclass
class FootstepConstraints:
    max_step_len:   float = 0.50  # [m] max długoś kroku wzdłużna
    max_step_width: float = 0.28  # [m] max odległość boczna miedzy stopami
    max_delta_yaw:  float = 0.35  # [rad] max skręt w jednym kroku (~20 deg)
    min_step_width: float = 0.08  # [m] stopy nie mogą nachodzić
    max_height:     float = 0.30  # [m] max różnica wysokości stopni
    zmp_margin:     float = 0.025 # [m] wymagany margines ZMP od krawędzi
    foot_len:       float = 0.24  # [m] długość stopy
    foot_width:     float = 0.10  # [m] szerokość stopy


def verify_footstep(prev: dict, curr: dict,
                    cfg: FootstepConstraints = FootstepConstraints()
                    ) -> dict:
    """
    Weryfikuje krok z prev → curr.
    Słowniki: {'x','y','yaw','z','left'} (z w metrach)
    Zwraca {'ok': bool, 'reason': str, 'metrics': dict}
    """
    dx   = curr['x'] - prev['x']
    dy   = curr['y'] - prev['y']
    dz   = curr.get('z', 0.0) - prev.get('z', 0.0)
    dyaw = abs(curr['yaw'] - prev['yaw'])
    # Normalizacja kata do [-pi, pi]
    dyaw = abs((dyaw + np.pi) % (2*np.pi) - np.pi)

    # Długoś w kierunku ruchu i na boki
    yaw_ref = prev['yaw']
    step_fwd  = dx*np.cos(yaw_ref) + dy*np.sin(yaw_ref)
    step_side = abs(-dx*np.sin(yaw_ref) + dy*np.cos(yaw_ref))
    step_dist = np.hypot(dx, dy)

    checks = {
        'step_len_ok':   step_dist <= cfg.max_step_len,
        'step_width_ok': step_side >= cfg.min_step_width,
        'max_width_ok':  step_side <= cfg.max_step_width,
        'yaw_ok':        dyaw <= cfg.max_delta_yaw,
        'height_ok':     abs(dz) <= cfg.max_height,
    }
    all_ok = all(checks.values())
    reason = 'OK' if all_ok else ', '.join(k for k, v in checks.items() if not v)
    metrics = {
        'step_fwd': step_fwd, 'step_side': step_side,
        'step_dist': step_dist, 'delta_yaw_deg': np.degrees(dyaw),
        'delta_z': dz
    }

    return {'ok': all_ok, 'reason': reason, 'metrics': metrics}
```

Wywołuj **verify_footstep()** przed każdym krokiem; odrzuć krok jeśli **ok=False** i zaplanuj alternatywę.

Parametry i ograniczenia planowania kroków

- **Długość kroku**: typowo 0,2–0,5 m; zależy od prędkości i stabilności (przy niskiej prędkości krócej)
- **Szerokość kroku**: 0,1–0,25 m; za mała = utrata stabilności, za duża = kolizja nóg
- **Orientacja stopy**: dostosowana do krzywizny ścieżki i pochylenia terenu (normal surface)
- **Strefy dozwolonego kontaktu** (foothold regions): płaskie, niegrzybowe obszary ≥ stopa robota
- **Kolejność kroków**: lewa–prawa naprzemiennie; podwójne podparcie ≥ 20% cyklu
- **Wykluczenie**: krawędzie, śliskie powierzchnie, zbyt strome pochylenie (>20°), wąskie progi
- **Planowanie w 3D**: wymagane gdy teren ma zmienną wysokość (schody, kamienie, nierówności)

Parametry kroków decydują o tym, czy chód będzie szybki, stabilny i wykonalny. Dłuższy krok zwiększa prędkość, ale zmniejsza margines stabilności i może przekroczyć zasięg nogi. Zbyt mała szerokość kroku grozi utratą równowagi, a zbyt duża może powodować nienaturalny chód lub kolizję nóg. Orientacja stopy powinna wynikać z kierunku ścieżki oraz geometrii terenu. Na nierównym podłożu szczególnie ważne jest wyznaczanie bezpiecznych obszarów kontaktu, czyli takich, które są płaskie, wystarczająco duże i oddalone od krawędzi.

KOD PYTHON

Generacja sekwencji kroków (footstep planner)

```python
import numpy as np

def generate_footsteps(path_xy, step_length=0.35,
                       step_width=0.18) -> list:
    """Generuje naprzemienne kroki L/P wzdłuż ścieżki"""
    footsteps = []
    left_foot = True  # zaczyna lewa noga
    dist_accum = 0.0
    for i in range(1, len(path_xy)):
        dx = path_xy[i][0] - path_xy[i-1][0]
        dy = path_xy[i][1] - path_xy[i-1][1]
        dist_accum += np.hypot(dx, dy)
        if dist_accum >= step_length:
            yaw   = np.arctan2(dy, dx)
            sign  = 1.0 if left_foot else -1.0
            foot_x = path_xy[i][0] - np.sin(yaw)*sign*step_width/2
            foot_y = path_xy[i][1] + np.cos(yaw)*sign*step_width/2
            footsteps.append({'x':foot_x,'y':foot_y,'yaw':yaw,'left':left_foot})
            left_foot = not left_foot
            dist_accum = 0.0
    return footsteps
```

Opis: Offset ±step_width/2 prostopadle do kierunku ruchu; 'left' flaga steruje naprzemiennością L/P

KOD PYTHON

Generacja trajektorii CoM z sekwencji kroków (model LIPM)

Simple ZMP preview: środek masy podąża za sekwencją ZMP wyznaczoną z footstepów

```python
import numpy as np

def generate_com_trajectory(footsteps: list,
                            com_height: float = 0.85,
                            g: float = 9.81,
                            T_step: float = 0.6,
                            T_dsp: float = 0.12,
                            dt: float = 0.005) -> np.ndarray:
    """
    Uproszczona generacja trajektorii CoM na podstawie sekwencji kroków.
    Uzywa modelu LIPM: x_ddot = omega^2 * (x - x_zmp)
    footsteps: lista {'x','y','yaw','left'}
    Zwraca ndarray (N, 4): [t, x, y, vx_vy...] pary CoM
    """
    omega = np.sqrt(g / com_height)   # ~3.4 rad/s przy h=0.85 m
    traj  = []

    # Inicjalizacja CoM nad srodkiem obu stop (zakladamy stojacy start)
    x_com  = footsteps[0]['x']
    y_com  = footsteps[0]['y']
    vx_com = 0.0
    vy_com = 0.0
    t      = 0.0

    for i in range(len(footsteps) - 1):
        fs_curr = footsteps[i]
        fs_next = footsteps[i + 1]

        # ZMP w fazie podporu jednostronnego: środek bieżącej stopy
        zmp_x = fs_curr['x']
        zmp_y = fs_curr['y']

        # Czas pojedynczego kroku
        t_end = t + T_step

        while t < t_end:
            # Rownanie LIPM: calkowanie Eulera
            ax = omega**2 * (x_com - zmp_x)
            ay = omega**2 * (y_com - zmp_y)
            vx_com += ax * dt
            vy_com += ay * dt
            x_com  += vx_com * dt
            y_com  += vy_com * dt
            traj.append([t, x_com, y_com, vx_com, vy_com])
            t += dt

        # Faza podwójnego podporu: ZMP przesuwa się do następnej stopy
        t_dsp_end = t + T_dsp
        while t < t_dsp_end:
            alpha  = (t - (t_end)) / T_dsp
            zmp_x  = (1-alpha)*fs_curr['x'] + alpha*fs_next['x']
            zmp_y  = (1-alpha)*fs_curr['y'] + alpha*fs_next['y']
            ax = omega**2 * (x_com - zmp_x)
            ay = omega**2 * (y_com - zmp_y)
            vx_com += ax * dt; vy_com += ay * dt
            x_com  += vx_com * dt; y_com  += vy_com * dt
            traj.append([t, x_com, y_com, vx_com, vy_com])
            t += dt

    return np.array(traj)   # (N, 5): [t, x, y, vx, vy]
```

$T_{step} = 0,6\ s, T_{dsp} = 0,12\ s$: typowe dla chodu $1,0\ m/s$; weryfikuj ZMP(t) w każdej klatce po generacji

Ograniczenia kinematyczne i stabilnościowe

Ograniczenia kinematyczne

- **Zasięg nogi**: max. $odległość\ stopa–biodro \leq l_{\max}$
- **Min. odstęp**: stopy nie mogą nachodzić na siebie
- **Zakres stawów**: każdy staw w przedziale $[q_{min}, q_{max}]$
- **Prędkość stawów**: $\omega_{joint} \leq \omega_{max}$ *(przepustowość aktuatora)*
- **Konfiguracja singularna**: unikanie pełnego wyprostu kolana

Ograniczenia stabilnościowe

- **ZMP** wewnątrz wieloboku podparcia z marginesem
- **Capture Point** w dopuszczalnej strefie landing
- **Odchylenie tułowia** $\leq \pm 15°$ od pionu
- **Prędkość CoM** ograniczona przez marginesy LIPM
- **Przy schodach**: dodatkowy margines boczny +3 cm

Planowanie kroków musi respektować zarówno ograniczenia kinematyczne, jak i stabilnościowe. Ograniczenia kinematyczne wynikają z budowy robota: długości kończyn, zakresów stawów, maksymalnych prędkości aktuatorów i unikania konfiguracji singularnych. Ograniczenia stabilnościowe określają, czy robot może utrzymać równowagę podczas wykonywania kroku. Najważniejsze jest utrzymanie ZMP w wieloboku podparcia z odpowiednim marginesem. Przy trudnym terenie, schodach i pochyłościach marginesy powinny być większe niż podczas ruchu po płaskiej powierzchni.

Planowanie na schodach i nierównościach

- **Detekcja schodów**: segmentacja chmury punktów; klasyfikacja płaskich poziomych płaszczyzn
- **Wysokość stopnia**: robot musi unieść stopę o $(h_{step} + clearance)$ ponad krawędź
- **Miejsca kontaktu**: wyznaczane na środku głębokości stopnia z marginesem ≥ 2 cm od krawędzi
- **Kolejność działań**: wejście przodem – lewa noga wyżej → prawa noga dołącza, powtórz;
- **Trajektoria stopy w powietrzu**: krzywa beziera nad stopniem; max. $wysokość = h_{step} + 5\ cm$
- **Kontrola ciała**: tułów pochyla się nieznacznie do przodu przy wchodzeniu, do tyłu przy schodzeniu
- **Zabezpieczenie**: jeśli wysokość stopnia > 25 cm lub brak miejsca kontaktu = zatrzymaj misję

Schody wymagają planowania w przestrzeni 3D, ponieważ oprócz pozycji x i y trzeba uwzględnić wysokość kontaktu stopy. Robot musi wykryć poziome płaszczyzny stopni, oszacować ich wysokość i głębokość oraz wybrać bezpieczne punkty kontaktu. Stopa powinna lądować z marginesem od krawędzi, a trajektoria swing foot musi zapewnić odpowiedni clearance nad stopniem. Przy zbyt wysokim stopniu lub braku stabilnego miejsca kontaktu robot powinien zatrzymać misję zamiast próbować wykonać ryzykowny krok.

KOD PYTHON

Weryfikacja ZMP względem wieloboku podparcia

Sprawdzenie czy trajektoria CoM zachowuje margines stabilności w każdym kroku

```python
import numpy as np
from scipy.spatial import ConvexHull

def build_support_polygon(active_feet: list, foot_len: float = 0.24,
                          foot_width: float = 0.10) -> np.ndarray:
    """
    Buduje wielobok podparcia jako wypukła obwiednie stóp.
    active_feet: lista {'x','y','yaw'} - aktualnie oparte stopy
    Zwraca ndarray (N, 2) wierzchołków wieloboku.
    """
    corners = []
    for f in active_feet:
        cx, cy, yaw = f['x'], f['y'], f['yaw']
        # 4 naroza kazdej stopy (prostokąt obrócony o yaw)
        for dx, dy in [( foot_len/2,  foot_width/2),
                       ( foot_len/2, -foot_width/2),
                       (-foot_len/2,  foot_width/2),
                       (-foot_len/2, -foot_width/2)]:
            rx = cx + dx*np.cos(yaw) - dy*np.sin(yaw)
            ry = cy + dx*np.sin(yaw) + dy*np.cos(yaw)
            corners.append([rx, ry])
    pts = np.array(corners)
    hull = ConvexHull(pts)
    return pts[hull.vertices]   # wierzcholki wieloboku

def zmp_margin(zmp: np.ndarray,
               support_polygon: np.ndarray) -> float:
    """
    Oblicza minimalna odległość ZMP od krawędzi wieloboku [m].
    Wartość ujemna = ZMP poza wielobokiem (utrata równowagi!).
    """
    n = len(support_polygon)
    min_dist = float('inf')
    for i in range(n):
        p1 = support_polygon[i]; p2 = support_polygon[(i + 1) % n]
        edge = p2 - p1; normal = np.array([-edge[1], edge[0]])
        normal /= (np.linalg.norm(normal) + 1e-9)
        d = np.dot(zmp - p1, normal); min_dist = min(min_dist, d)
    return float(min_dist)

def verify_com_trajectory(traj: np.ndarray,
                          footsteps: list,
                          com_height: float = 0.85,
                          g: float = 9.81,
                          margin_req: float = 0.02) -> dict:
    """Weryfikuje trajektorie CoM: oblicza ZMP i sprawdza
margines."""
    results = {'ok': True, 'min_margin': float('inf'), 'violations':
[]}
    active = footsteps[:2] if len(footsteps) >= 2 else footsteps
    polygon = build_support_polygon(active)
    for row in traj:
        t, x, y, vx, vy = row
        ax = np.gradient([vx], [t])[0] if len(traj) > 1 else 0.0
        ay = 0.0; zmp_x = x - (com_height / g) * ax
        zmp = np.array([zmp_x, y]); m = zmp_margin(zmp, polygon)
        results['min_margin'] = min(results['min_margin'], m)
        if m < margin_req:
            results['ok'] = False
            results['violations'].append({'t': t, 'margin': m})
    return results
```

$zmp_{margin}() > 0 = ZMP$ wewnątrz; < 0 = utrata równowagi; $margin_{req} = 0{,}02\,m$ (2 cm) to min. dla płaskiej nawierzchni

KOD PYTHON

Detekcja schodów z chmury punktów i plan kroków 3D

Segmentacja poziomych płaszczyzn z LiDAR, generacja celów stóp na stopniach

```python
import numpy as np

def detect_stairs(point_cloud: np.ndarray, z_min: float = 0.03, z_max: float = 0.30,
                  plane_tol: float = 0.03, min_pts: int = 50) -> list:
    """
    Detekcja stopni schodów z chmury punktów (uproszczona).
    point_cloud: (N, 3) – punkty [x, y, z] w ramce base_link
    Zwraca listę stopni: [{'z_height', 'x_center', 'y_center', 'depth'}]
    """
    # Filtruj punkty w zakresie wysokości schody
    mask = (point_cloud[:, 2] > z_min) & (point_cloud[:, 2] < z_max)
    pts  = point_cloud[mask]
    if len(pts) < min_pts:
        return []

    # Grupuj punkty według wysokości z (bin 3 cm)
    z_vals  = pts[:, 2]
    z_bins  = np.arange(z_min, z_max, plane_tol)
    stairs  = []

    for z_lo, z_hi in zip(z_bins[:-1], z_bins[1:]):
        layer = pts[(z_vals >= z_lo) & (z_vals < z_hi)]
        if len(layer) < min_pts:
            continue
        # Sprawdź, czy warstwa jest pozioma (plaska płaszczyzna)
        z_std = layer[:, 2].std()
        if z_std < plane_tol * 0.5:
            stairs.append({
                'z':       float(layer[:, 2].mean()),
                'x':       float(layer[:, 0].mean()),
                'y':       float(layer[:, 1].mean()),
                'depth':   float(layer[:, 0].max() - layer[:, 0].min()),
                'width':   float(layer[:, 1].max() - layer[:, 1].min()),
                'n_pts':   len(layer)
            })
    return sorted(stairs, key=lambda s: s['z']) # rosnąco wg wysokości


def plan_stair_footsteps(stairs: list, step_width: float = 0.18,
                         margin: float = 0.03) -> list:
    """
    Generuje naprzemienne L/P cele stóp dla wykrytych stopni.
    margin: [m] odległość od krawędzi stopnia
    """
    footsteps = []
    left = True
    for step in stairs:
        foot_x = step['x'] - margin    # cofnij od krawędzi przedniej
        foot_y = step['y'] + (step_width/2 * (1 if left else -1))
        footsteps.append({
            'x': foot_x, 'y': foot_y, 'z': step['z'],
            'yaw': 0.0, 'left': left,
            'clearance': step['z'] + 0.05 # unieś stopę o 5 cm nad stopień
        })
        left = not left
    return footsteps
```

Binowanie po z co 3 cm; STD < 1.5 cm = plaska płaszczyzna; margin=3 cm od krawędzi stopnia (bezpieczeństwo)

Trajektoria środka masy, a sekwencja kroków

- **CoM** (Centre of Mass) musi śledzić zaplanowane kroki przez model LIPM (Linear Inverted Pendulum)
- **Przy podwójnym podparciu**: CoM przesuwa się od poprzedniej stopy do następnej
- **Przy pojedynczym podparciu** (faza swing): CoM podąża parabolą wyznaczoną przez ZMP
- **Sprzężenie**: sekwencja kroków → trajektoria ZMP → trajektoria CoM (generator QP)
- **Weryfikacja online**: ZMP(t) ∈ convex hull(stopa_oparta) przez cały cykl kroku
- **Narzędzie**: biblioteki Choreonoid, OpenHRP, ROS2 Whole Body Controller
- **Kompromis**: krótsze, częstsze kroki = lepsza stabilność CoM kosztem prędkości ruchu

Sekwencja kroków określa, gdzie mają znaleźć się stopy, ale stabilny chód wymaga również zaplanowania trajektorii środka masy. Model LIPM pozwala uprościć dynamikę humanoida i powiązać ruch CoM z położeniem ZMP. W fazie pojedynczego podparcia środek masy przemieszcza się względem jednej stopy podporowej, a w fazie podwójnego podparcia ZMP przechodzi między stopami. Im krótsze i częstsze kroki, tym łatwiej utrzymać stabilność, ale kosztem prędkości i efektywności ruchu.

Dobór planisty do konkretnej aplikacji

Laboratorium badawcze

A* + DWA; prosta cost map; footstep planer 2D; iteracyjna weryfikacja ZMP; MuJoCo do testów

Robot usługowy (biuro)

A* z cost map + Social DWA; detekcja YOLO; adaptacyjne kroki; zatrzymanie przy ludziach < 1 m

Ratownictwo (gruz)

3D A* lub RRT*; footstep w 3D z height map; MPC lokalny; konserwatywne marginesy ZMP

Środowisko zatłoczone

ORCA lub S-MPC; krótki horyzont predykcji (0,5 s); priority queue stref bezpiecznych; niskie prędkości

Schody i poziomy

Hierarchiczny planer: globalny graf pięter + lokalny 3D footstep; detekcja stopni z point cloud

Sport/zawody

MPC z agresywnym horyzontem; dynamiczny chód biegowy; uproszczony footstep dla szybkości kosztem marginesu ZMP

Nie istnieje jeden uniwersalny planista najlepszy dla wszystkich zastosowań. W laboratorium można użyć prostszej kombinacji A*, DWA i podstawowej weryfikacji ZMP. W robocie usługowym ważniejsze są reakcje na ludzi, dlatego potrzebna jest detekcja pieszych, social navigation i bezpieczne zatrzymanie. W ratownictwie lub na gruzowisku konieczne jest planowanie 3D, height mapy i konserwatywne marginesy stabilności. Dobór algorytmu powinien wynikać z charakteru środowiska, dostępnych sensorów, mocy obliczeniowej oraz poziomu ryzyka akceptowanego w danej aplikacji.

PODSUMOWANIE

01 **Planowanie globalne** (A*, Dijkstra) wyznacza optymalną trasę; heurystyki i cost map sterują wyborem ścieżki

02 **Planista lokalny** (DWA, MPC, potential fields) reaguje na zmiany środowiska i przeszkody dynamiczne

03 **Stabilność** ma wyższy priorytet niż optymalność trasy – planista musi uwzględniać margines równowagi

04 **Footstep planning** tłumaczy geometryczną ścieżkę na fizyczną sekwencję kroków z weryfikacją kinematyczną

05 **Trzy poziomy decyzji** – globalny, lokalny, footstep – są sprzężone i muszą działać w spójnej pętli

Zadanie projektowe i pytania problemowe

1. Dlaczego heurystyka Manhattan nie jest dopuszczalna dla siatek z ruchem 8-kierunkowym?
2. Co się dzieje z polem potencjału, gdy robot wpadnie w lokalne minimum? Zaproponuj wyjście.
3. Jaką wartość parametru α i β wybrałbyś dla DWA w zatłoczonym korytarzu? Uzasadnij.

Zadanie projektowe: zaprojektuj planistę kroków dla humanoida poruszającego się w korytarzu 1,5 m szerokości z napotykanymi ludźmi. Opisz: jakie dane wejściowe są potrzebne, jak wyznaczasz miejsca kontaktu stóp, jak weryfikujesz każdy krok, jak reaguje planista gdy droga jest zablokowana przez >3 s.

Bonus: zaimplementuj wygładzanie B-spline dla ścieżki zwróconej przez A* i porównaj liczbę kroków koniecznych do pokonania tej samej trasy przed i po wygładzeniu.