Bezzałogowe Obiekty Autonomiczne

Synteza i symulacja reaktywnego systemu sterowania dla wybranych struktur kinematycznych kołowego robota mobilnego

AiR S2-I/Rob

Sekcja 3

Piotr Gruchalski
Paweł Kaźmieruk
Maksymilian Skibiński
Mateusz Szczepanik

Wydział Automatyki, Elektroniki i Informatyki
Politechnika Śląska
26 września 2022 r.

1 Wstęp

Celem zajęć projektowych było zaprojektowanie reaktywnego systemu sterowania dla wybranego robota mobilnego. Taki system sterowania powinien zadbać o dotarcie robota do celu (lub możliwie blisko niego) oraz omijanie przeszkód. Praca ta została wykonana w środowisku MATLAB z użyciem toolboxa Mobile Robotics Simulation.

1.1 Teoria

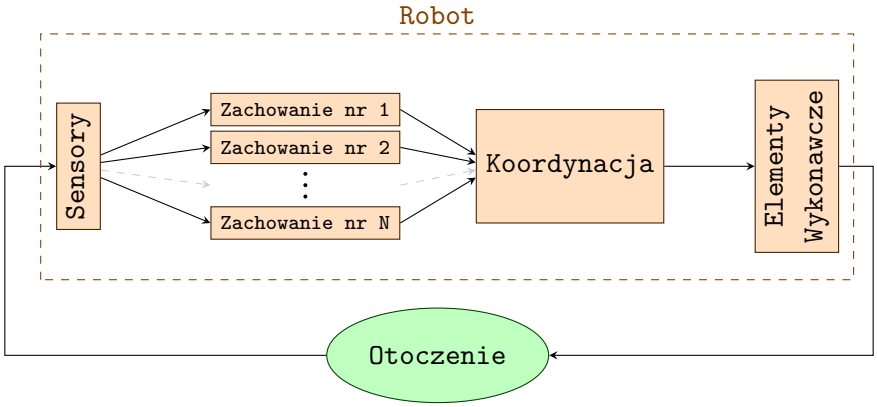
Reaktywne systemy sterowania (zwane też behawioralnymi) charakteryzuje dosyć szybkie wyznaczanie zachowań robota w odpowiedzi na sygnały sensorów w zamian za świadomą rezygnację z możliwości otrzymania optymalnej trasy dojścia do celu. Taka transakcja ma sens, gdyż niezwykle trudno jest wykonać taki sekwencyjny system sterowania, który poradzi sobie ze sprawnym przetwarzaniem wszystkich danych dostępnych z sensorów robota, tak by wyznaczyć jak najlepszą trasę. Podejście behawioralne, proponuje wyposażenie robota w pewien nieduży zbiór zachowań i ich koordynację, które pozwolą mu *dobrze* radzić sobie w pewnym środowisku.

Przy projektowaniu takiego systemu należy wykonać dwa zadania:

1. Wyposażyć robota w pewien zbiór zachowań;
2. Skoordynować je ze sobą.

Zbiór zachowań jest, ogólnie, zbiorem funkcji, które przetwarzają sygnały pochodzące z sensorów robota, w akcje. Teoretycznie, czym więcej zachowań, tym robot staje się bardziej inteligentny i może poradzić sobie w większej liczbie sytuacji. Większą ilość zachowań jest jednak trudniej ze sobą skoordynować oraz wiąże się z większym nakładem obliczeniowym, dlatego powinno się celować w mniejszy zbiór, który potrafi bardziej ogólnie, a mniej szczegółowo, opisywać zachowanie robota w pewnym wybranym środowisku.

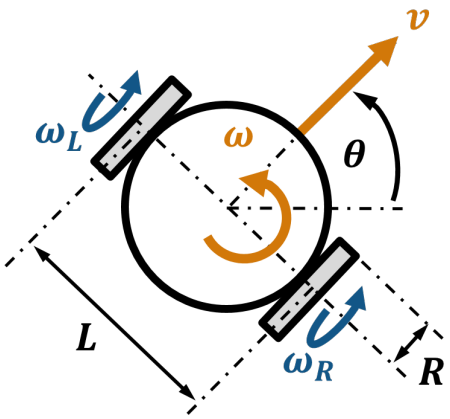
Kolejną częścia systemu jest układ koordynacji, który decyduje jak ostatecznie zachowa się robot. Pewne sygnały generują sprzeczne ze sobą akcje, pewne oddziałują na siebie w jakimś stopniu, a inne nie wpływają na siebie w ogóle. Zadaniem koordynacji jest pogodzić ze sobą wszystkie zachowania, i wyznaczyć ostateczne sterowanie w danej chwili czasu. Taki cel można osiągnąć korzystając np. z metod priorytetowych, które służą do oznaczenia zachowań ważniejszych i tym samym wykonywanych w danej chwili, bądź można skorzystać z fuzji zachowań, wyznaczając tym samym pewne wypadkowe sterowanie.



Rysunek 1: Uproszczona struktura systemu behawioralnego

1.2 Środowisko

Wykorzystany toolbox MATLABa oferuje kilka różnych struktur kinematycznych robotów mobilnych w postaci gotowych funkcji, służących do symulacji dynamiki robotów. W pracy skorzystano z mobilnego robota z napędem różnicowym.



Rysunek 2: Model kinematyczny robota z napędem różnicowym

Robota opisują 3 dostępne funkcje:

- `DifferentialDrive(R, L)` – powołuje do życia naszą maszynę. Jest ona opisana przez dwa parametry: promień kół (R) oraz odległość pomiędzy ich środkami (L).
- `forwardKinematics(vehicle, wL, wR)` – rozwiązanie prostego zadania kinematyki. Znając prędkości kątowe obu kół (ω_L, ω_R), otrzymujemy prędkość liniową robota (v) oraz jego prędkość kątową (ω).
- `inverseKinematics(vehicle, v, w)` – rozwiązanie odwrotnego zadania kinematyki. Wiąże wspomniane wyżej wielkości w odwrotnej kolejności.

Ostatnie dwie funkcje po prostu rozwiązują zapisane niżej równania:

`forwardKinematics`

$$v = \frac{R}{2}(\omega_R + \omega_L)$$

$$\omega = \frac{R}{L}(\omega_R - \omega_L)$$

`inverseKinematics`

$$\omega_L = \frac{1}{R}\left(v - \frac{\omega L}{2}\right)$$

$$\omega_R = \frac{1}{R}\left(v + \frac{\omega L}{2}\right)$$

Wielkości:

- R, L są zapisane w metrach [m],
- ω_R, ω_L, ω w radianach na sekundę $[\frac{\text{rad}}{\text{s}}]$,
- v w metrach na sekundę $[\frac{\text{m}}{\text{s}}]$.

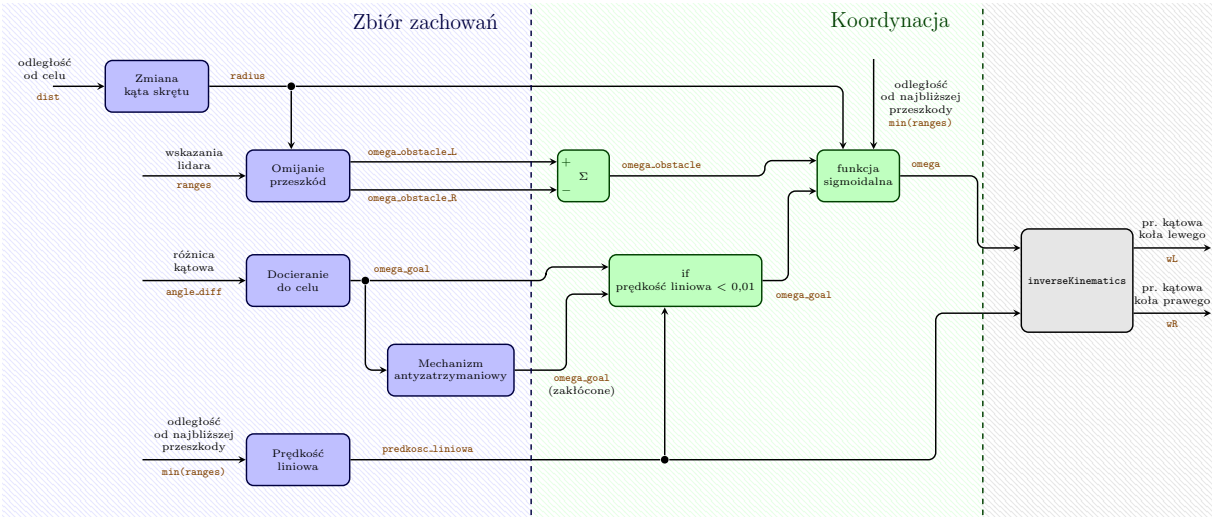
Robot wyposażony został w lidar, służący do wykrywania przeszkód. Składa się nań 10 równomiernie rozmieszczonych wiązek z zakresu kątów od $-\frac{\pi}{2}$ do $\frac{\pi}{2}$.

Wszystkie przedstawione w tym podpunkcie informacje można łatwo znaleźć w dokumentacji toolboxa (link).

2 System sterowania

Zgodnie z ideą sterowania behawioralnego, system dzieli się na zbiór zachowań oraz ich koordynację. W tym rozdziale zostanie omówiony ogólny sposób działania naszego systemu oraz przedstawione zostaną fragmenty kodu odpowiedzialne za omawiane funkcje. Cały kod został umieszczony na końcu raportu.

2.1 Ogólna koncepcja



Rysunek 3: Struktura naszego systemu sterowania

Na rysunku powyżej przedstawiony został schemat blokowy ilustrujący przepływ sygnałów z sensorów i ich wpływ na wynikowe sterowanie – na system składa się zbiór pięciu zachowań i trzech funkcji koordynujących, a w końcowej fazie wykorzystywana jest funkcja rozwiązująca odwrotne zadanie kinematyki (`inverseKinematics`), dająca wynikowe prędkości kątowe dla obu kół robota.

Na układ sensoryczny robota składa się lidar dostarczający informacji o odległości robota od przeszkód, oraz bliżej niezidentyfikowany sensor, dostarczający informacji o położeniu celu, robota i, w ten sposób, o różnicy pomiędzy ich pozycjami. Sygnały te trafiają do zbioru zachowań (a także, jeden z nich do modułu koordynacji), i służą do obliczeń odpowiednich sygnałów sterujących.

Za najistotniejsze dla działania naszego system sterowania można uznać dwa zachowania i dwie funkcje koordynujące:

– *Omijanie przeszkód*, *Docieranie do celu* – dla z. zachowań,

– Σ, *f. sigmoidalna* – dla koordynacji.

Omijanie przeszkód oblicza prędkości kątowe dla obu kół, takie by ominąć wykryte przez lidar przeszkody, a *Docieranie do celu* oblicza p. kątową robota, ze względu na różnicę kątową pomiędzy orientacją robota, a celem, sprowadzającą tę różnicę do zera. Sygnały wynikowe zachowania *Omijanie przeszkód*, są koordynowane przy pomocy ich superpozycji, a następnie ten sygnał, wraz z sygnałem wyjściowym zachowania *Docieranie do celu*, są koordynowane przy pomocy funkcji sigmoidalnej. Ta funkcja ma dwa parametry, ale jeden z nich jest ważniejszy – jest to odległość od przeszkody. Czym bliżej robot jest przeszkody, tym mocniejsze znaczenie ma

sterowanie obliczone przez zachowanie omijające przeszkody, a mniejsze to pozwalające dotrzeć do celu.

Pozostałe zachowania/koordynacje, pozwalają lepiej przygotować robota do operowania w środowisku:

- *Prędkość liniowa* jest jedyną funkcją regulującą pr. liniową robota. Ogólnie jest ona stała, ale czym bliżej robot znajduje się przeszkody, tym mocniej jest ona zmniejszana.
- *Mechanizm antyzatrzymaniowy* generuje zakłóconą pr. kątową (ze względu na cel), pozwalającą w sytuacjach gdy robot trafia do kąta przeszkody, opuścić go. Taka sytuacja jest u nas wykrywana poprzez sprawdzenie czy prędkość liniowa robota jest bardzo mała. Gdy taki warunek jest prawdziwy funkcja koordynująca (*if prędkość liniowa < 0,01*) przepuszcza ten sygnał zakłócony – jest to koordynacja na zasadzie priorytetowej.
- *Zmiana kąta skrętu* oblicza parametr `radius` służący jako argument funkcji sigmoidalnych wykorzystywanych w dwóch miejscach (*Omijanie przeszkód*, *funkcja sigmoidalna*). Zachowanie to pozwala nam na zmianę wagi sterowania omijającego przeszkody, gdy robot znajduje się bliżej celu – w ten sposób robot potrafi dotrzeć do celu przy przeszkodzie.

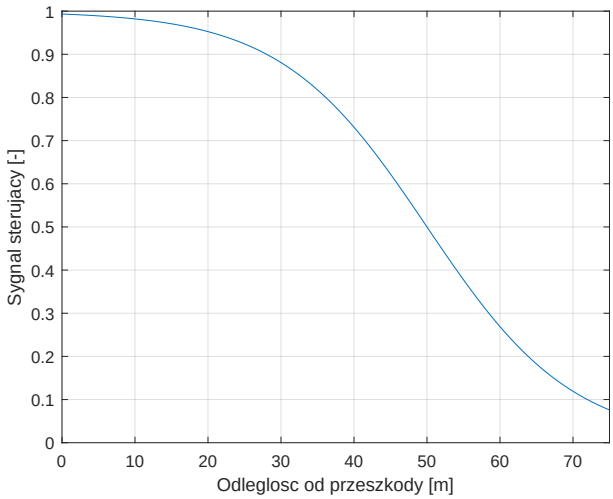
W następnych punktach te funkcje zostaną omówione szerzej (wraz z ich kodami).

2.2 Zachowania

Omijanie przeszkód

Robot wykrywa przeszkody przy pomocy lidara, który omiata otoczenie dziesięcioma wiązkami w zakresie kątów od $-\frac{\pi}{2}$ do $\frac{\pi}{2}$. Wygenerowane sterowanie zależy od 2 parametrów:

- pozycji wiązki lidara – im bliżej kąta zerowego, tym mocniejsza reakcja. Pozwala to lepiej zabezpieczyć robota przed przeszkodami, z którymi zderzenie może nastąpić szybciej.
- odległości od przeszkody – czym robot bliżej jest przeszkody, tym mocniej musi zmienić swoją prędkość kątową by ją ominąć.



Rysunek 4: Używana funkcja sigmoidalna

Do wypracowania odpowiedniego sygnału sterującego skorzystano z funkcji sigmoidalnej. Jak łatwo zauważyć, przeszkody, które znajdują się daleko ingerują słabiej w zmianę prędkości

kątowej, natomiast te które są bardzo blisko – już dużo mocniej. Wyjście samej funkcji jest bezwymiarowe - wymiar prędkości kątowej ($\frac{\text{rad}}{\text{s}}$) otrzymuje, gdy jest przemnożone przez wagę związaną z pozycją wiązki lidara.

Po uwzględnieniu tych założeń stworzono następujący kod.

```
73    % Prędkości obu kół wynikające z przeszkód.
74    omega_obstacle_L = sum( ...
75        f_sigmoidalna(0.1, radius, ranges(1:5)) .* linspace(0.01, 0.1, 5)');
76    omega_obstacle_R = sum( ...
77        f_sigmoidalna(0.1, radius, ranges(6:10)) .* linspace(0.1, 0.01, 5)');
```

Zmienna `ranges` przechowuje informację o wykrytych odległościach od przeszkód. Wektor wag tworzony poprzez funkcję `linspace` służy do uwzględnienia z jaką siłą będą działały wiązki „centralne", według omówionego wyżej pomysłu. 5 pierwszych wiązek działa z lewej strony, a ostatnie 5 z prawej – wypracowane sterowania są ze sobą sumowane i w ten sposób otrzymujemy pożądaną, według danego zachowania, prędkość kątową robota.

Pierwszy argument funkcji sigmoidalnej, odpowiadający za wygładzenie kształtu funkcji, (0.1) został dobrany eksperymentalnie. Podobnie jest z parametrem `radius`, który ogólnie wynosi 50, ale jego dokładna wartość jest powiązana z innym (omówionym później) zachowaniem.

Docieranie do celu

Robot stara się minimalizować różnicę kątową pomiędzy swoją orientacją a celem. W zależności od wartości tej różnicy, liniowo zmienia swoją prędkość kątową.

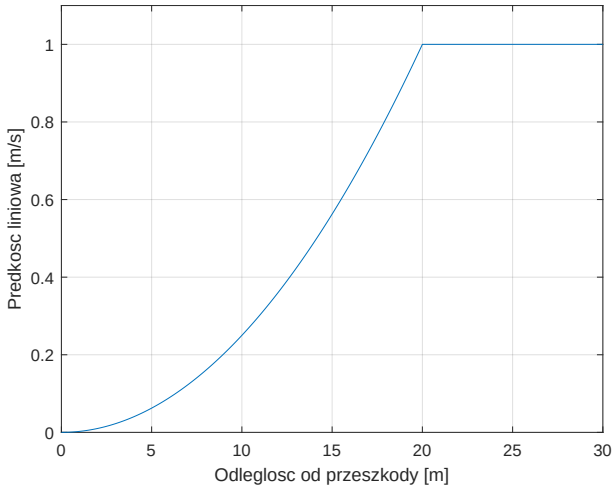
```
83    % Kąt celu.
84    angle_goal = atan2(goal(2) - pose(2), goal(1) - pose(1));
85
86    % Różnica kątowa wraz z rozwiązaniem problemu nieciągłości
87    % funkcji atan2.
88    angle_diff = atan2( ...
89        sin(angle_goal - pose(3)), ...
90        cos(angle_goal - pose(3)) ...
91    );
92
93    % Prędkość kątowa wynikająca z różnicy kątowej.
94    omega_goal = angle_diff * 0.1;
```

Prędkość liniowa

Robot stara się poruszać ze stałą prędkością liniową, ale gdy jest blisko przeszkody zmniejszą ją, by uniknąć zderzenia ze ścianą. To zachowanie nazwano „mechanizmem antykolizyjnym".

```
102   % Mechanizm antykolizyjny.
103   % Zmiany prędkości liniowej w funkcji odległości od przeszkody.
104   % Zabezpiecza robota przed kolizją.
105   if min(ranges) < 20
106       predkosc_liniowa = (1/20 * min(ranges))^2;
107   else
108       predkosc_liniowa = 1;
109   end
```

Kształt funkcji zmniejszającej prędkość liniową jak i sama odległość od celu, zostały dobrane doświadczalnie – ostatecznie zdecydowaliśmy się realizować to przy pomocy f. kwadratowej.



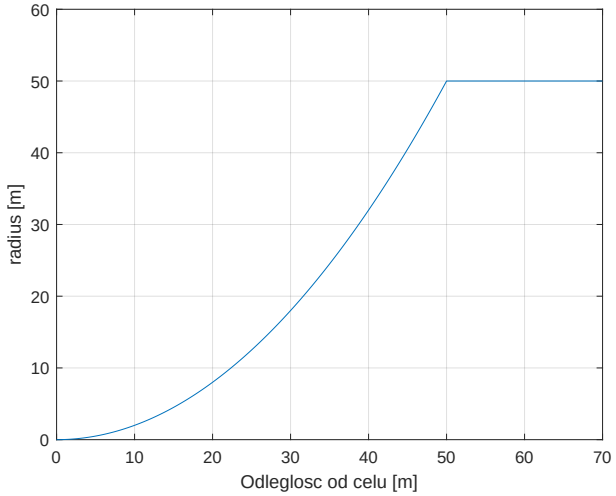
Rysunek 5: Zależność prędkości liniowej od odległości od przeszkody

Zmiana kąta skrętu - parametr `radius`

W sytuacji gdy robot jest blisko celu zaczyna zmniejszać promień skrętu, by trafić weń nawet wówczas, gdy tuż obok celu jest przeszkoda.

```
61      % Dystans od celu.
62      dist = sqrt((pose(1) - goal(1))^2 + (pose(2) - goal(2))^2);
63
64      % Jeśli odległość robota od celu jest mała,
65      % to zmieniamy kąt skretu - im bliżej tym ciaśniejszy.
66      if dist > 50
67          radius = 50;
68      else
69          radius = (1/sqrt(50) * dist).^2;
70      end
```

Parametr ten zmienia swoje wartości zgodnie z funkcją kwadratową. Zależność ta zaczyna jednak działać dopiero wówczas, gdy robot jest blisko celu – w przeciwnym razie zbyt mocno zniekształcałaby sterowanie wypracowywane przy przeszkodach.

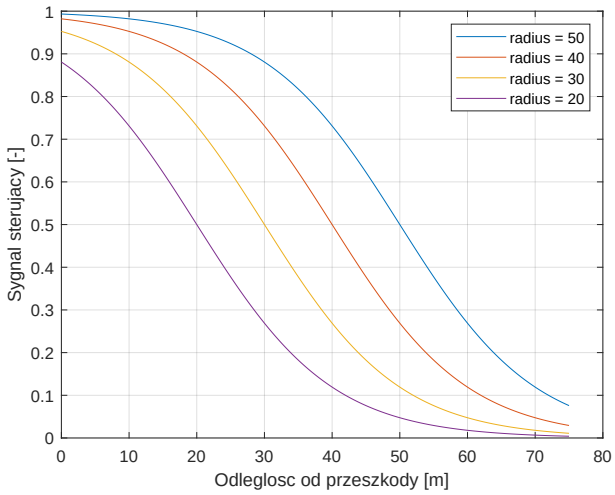


Rysunek 6: Zależność parametru `radius` od odległości do celu

Działanie tego parametru może nie być zbyt intuicyjne. Jest on wykorzystywany w dwóch

miejscach w naszym kodzie: przy wyznaczaniu prędkości kątowych wynikających z odległości od przeszkód, oraz przy wyznaczaniu współczynnika koordynacji, przy ostatecznym wyznaczaniu prędkości kątowej robota (co jest omówione później) – jest on drugim argumentem funkcji sigmoidalnej, która jest w tych sytuacjach wykorzystywana.

Na rysunku niżej, został przedstawiony wpływ zmian wartości tego parametru na kształt funkcji mapującej sygnał wpływający na pr. kątową w zależności od odległości od przeszkody. Zmniejszenie wartości tego parametru powoduje przesunięcie funkcji sigmoidalnej, co z kolei wpływa na zmniejszenie wpływu odległości od przeszkody na wypadkową prędkość kątową – sygnały wynikające z rozmieszczenia przeszkód zostają osłabione, przez co różnica kątowa względem celu, ma większy wpływ.



Rysunek 7: Wpływ `radius`

Mechanizm antyzatrzymaniowy

Czasem zdarza się, że robot zatrzymuje się przy przeszkodzie, bo generowane sterowanania do tego go zmuszają. W tej sytuacji nieznacznie zniekształcamy zadaną prędkość kątową, by zmienić tor jego ruchu. To zachowanie nazwano "mechanizmem antyzatrzymaniowym".

```
96      % Mechanizm antyzatrzymaniowy.
97      if predkosc_liniowa < 0.01
98          omega_goal = (angle_diff - pi/2) * 0.1;
99      end
```

2.3 Koordynacja

Układ koordynacji obsługuje zachowania ingerujące w prędkość kątową robota. Prędkość liniowa jest zmieniana poprzez jedną akcję, dlatego w jej wypadku moduł ten nie ma co czynić.

Prędkość kątowa wynikająca z przeszkód

Dotyczy dwóch zachowań robota, które tak naprawdę stanowią jedność – dwie prędkości kątowe, które zostają wyznaczone w zależności od odległości od wykrytych przeszkód. Prędkości te są ze sobą łączone na zasadzie superpozycji – jedna jest odejmowana od drugiej.

```
79    % Koordynacja obu tych prędkości.
80    omega_obstacle = omega_obstacle_L - omega_obstacle_R;
```

Mechanizm antyzatrzymaniowy

Mechanizm antyzatrzymaniowy ma wyższy priorytet niż ogólna prędkość kątowa wypracowana poprzez różnicę kątową względem celu. W przypadku, gdy prędkość liniowa jest bardzo mała (czyli odległość od przeszkody jest bardzo mała) prędkość kątowa jest zniekształcana.

```
96    % Mechanizm antyzatrzymaniowy.
97    if predkosc_liniowa < 0.01
98        omega_goal = (angle_diff - pi/2) * 0.1;
99    end
```

„Ogólna” prędkość kątowa

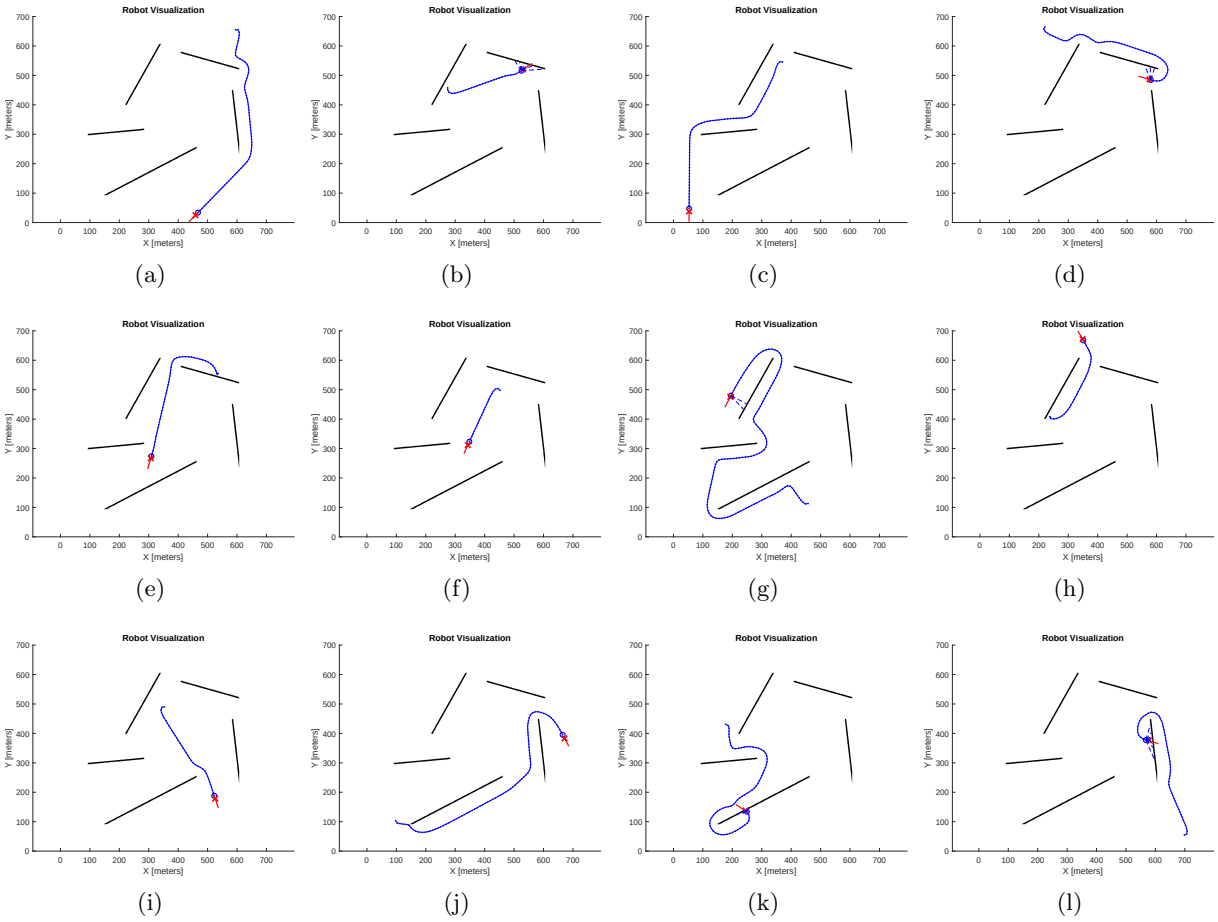
W większości sytuacji robot musi jednocześnie zmieniać swoją prędkość kątową w zależności od odległości od przeszkody i różnicy kątowej względem celu. Te dwa zachowania są ze sobą powiązane na zasdzie fuzji – czym bliżej robot jest przeszkody, tym ważniejsze jest sterowanie wygenerowane przez zasadę omijającą przeszkodę. Wykorzystano do tego wcześniej wspomnianą funkcję sigmoidalną.

```
112   % Współczynnik alfa, przyjmuje wartości z zakresu 0 do 1.
113   % Odpowiada za fuzję prędkości kątowych wyzacznonych od
114   % celu oraz przeszkód.
115   alfa = f_sigmoidalna(0.1, radius, min(ranges));
116
117   % Prędkość kątowa ostateczna.
118   omega = alfa * omega_obstacle + (1 - alfa) * omega_goal;
```

3 Testy

3.1 Prosta mapa

Zaprojektowany system sterowania radzi sobie bardzo dobrze w sytuacjach nieskomplikowanych, tj. gdy przeszkód jest niewiele.



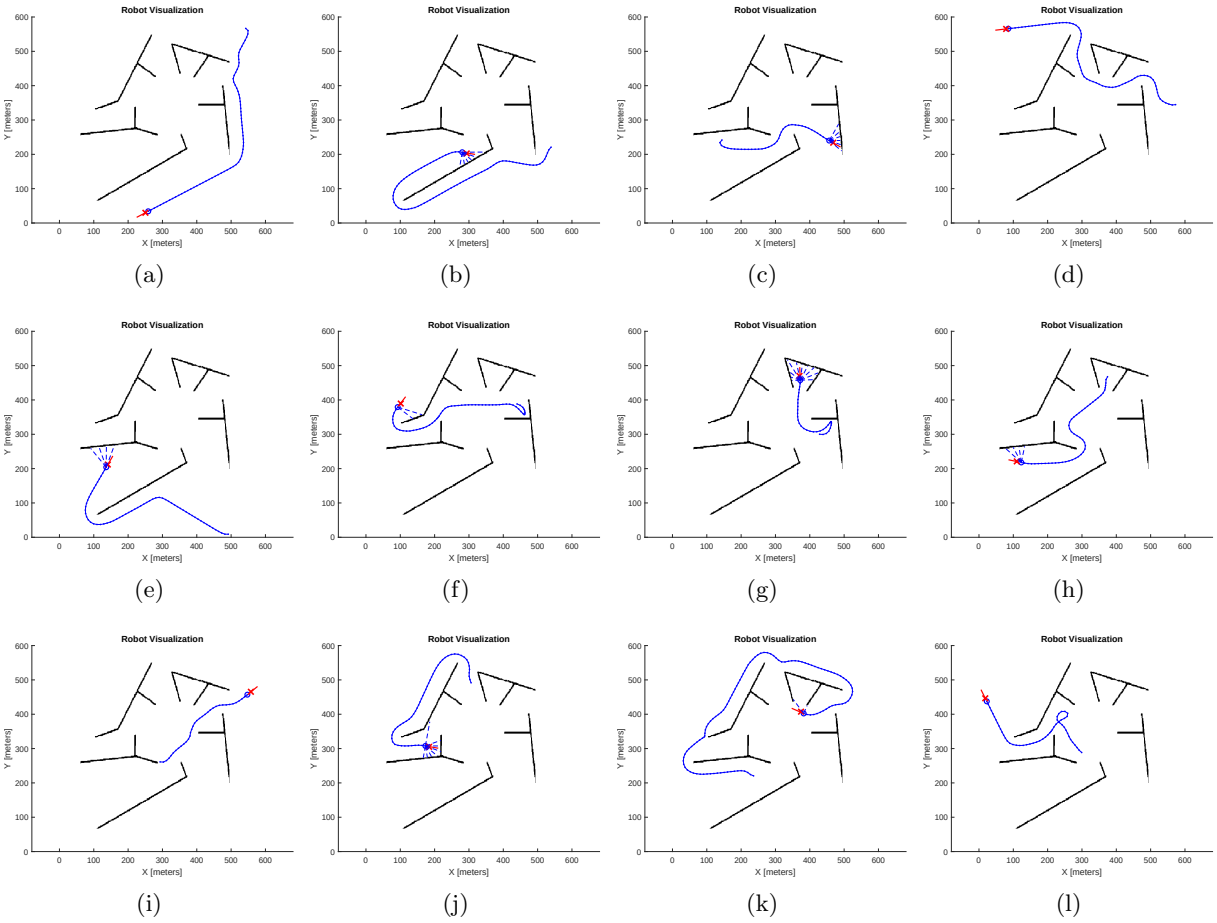
Rysunek 8: Przykłady prostych zadań

Na takiej mapie robot dobrze radzi sobie z płynną regulacją prędkością obrotową, tak by omijać przeszkody i docierać do celu. Wypracowane przez system sterowania trasy można nazwać całkiem intuicyjnymi (za wyjątek można by uznać przykład na rysunku (g)). Ponadto robot umie docierać do celu, który znajduje się zaraz przy przeszkodzie[1] (przykłady (k), (l)).

[1]Takie zadania dla robota były problematyczne wcześniej, na etapie poprzedniego szkicu naszego raportu – jest on już nieaktualny.

3.2 Dodanie kolejnych przeszkód

Poziom trudności mapy zwiększono, by sprawdzić działanie robota w bardziej złożonych sytuacjach. Trasa, którą obiera robot nie zawsze jest trasą najkrótszą, ale wynika to z jego ograniczonej wiedzy o otoczeniu.



Rysunek 9: Przykłady zadań o umiarkowanym poziomie trudności

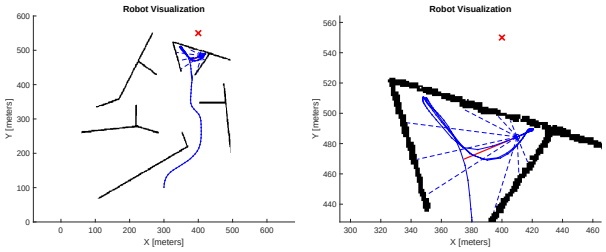
Ogólnie robot radzi sobie cały czas dobrze z omijaniem przeszkód i docieraniem do celu, ale na takiej mapie zdarza mu się zachowywać w „dziwny" sposób – za przykłady mogą posłużyć zadania z rysunków (f), (g), (k) oraz (l).

– w przypadku (f) oraz (g), pozycja początkowa robota, tj. koordynaty x, y oraz *orientacja*, powodują, że robot podjeżdża do ściany, a dopiero później zmienia swoją orientację względem celu, w bardziej gwałtowny sposób. Najlepiej byłoby, gdyby zaczął od „naprawienia" swojej orientacji, by podążać względem celu, w ten sposób zaooszczędziłby sobie swojej nadmiarowej, zupełnie niepotrzebnej drogi. Zaprojektowany przez nas system wymusza, by robot starał się podążąć ze stałą prędkością liniową, którą zmniejsza dopiero przy samych preszkodach, stąd taka trasa.

– podobny wpływ pozycji początkowej robota, na jego trasę do celu widać na rysunku (k). Gdyby orientacja początkowa, była równa ok. 0 rad, robot znacznie szybciej dotarłby do celu. Ten problem jednak dużo bardziej wynika z samej idei systemu behawioralnego, który stara się szybko wyznaczać następne sterowania, zamiast optymalnej trasy, niźli z naszych pomysłów (choć te też pewnie mogłyby być lepsze).

– na rysunku (l) widać, jak w pewnym momencie robot robi nadmiarowy obrót, przy skręcie. Wynika to po prostu z naszego sposobu koordynacji reguł wyznaczających prędkość kątową względem celu, oraz względem przeszkód. Koordynowane są na zasadzie superpozycji, z współczynnikiem, który zmienia swoją wartość w zależności od odległości od przeszkody. Znacznie lepiej byłoby, gdyby robot po prostu skręcił w lewo, zamiast wykonywać taki obrót, ale należołaby zastosować jakiś inny sposób koordynacji.

Niestety, na nowej mapie istnieje też przeszkoda, której robot nie potrafi pokonać.



(b) (c) zbliżenie

Rysunek 10: Robot wpada w pułapkę

System nie jest w stanie znaleźć drogi wyjścia i wpada w pętlę tych samych sterowań, oscylując od jednego punktu do drugiego. Ten problem, także wynika z naszego sposobu koordynacji zachowań wyznaczających prędkości kątowe. Pozycja celu zmusza robota do podążania trasą, gdzie czeka go, prawie czołowe, zderzenie z przeszkodą. Czym jest jej bliżej, tym zachowanie przeciwzderzeniowe, stara się mocniej zmieniać prędkość kątową i ostatecznie robot trafia do kąta przeszkody. W kącie, bardzo powoli, zmienia swoją pozycję, aż udaję mu się wydostać z zaułka, ale i tak, preszkody, wraz z pozycją celu, prowadzą robota do kolejnego zaułka.

Robot cały czas omija jednak ściany.

3.3 Labirynt

Znacznie trudniejszym zadaniem dla robota jest poruszanie się po labiryncie.

(a) (b) (c) (d)

(e) (f) (g) (h)

(i) (j) (k) (l)

(m) (n) (o) (p)

Rysunek 11: Przykłady zadań w labiryncie

W labiryncie robot radzi sobie znacznie gorzej. W przybliżeniu połowa testów kończy się sukcesem, a druga połowa porażką – ugrzęźnięciem w jednym z zaułków labiryntu. Nigdy jednak nie zderza się z przeszkodą. Wielokrotnie można też obserwować działanie mechanizmu antyzatrzymaniowego, który na mniej skomplikowanych mapach nie był zbyt często przydatny. Przykłady jego działania można zaobserwować np. na rysunkach (c), (e) czy (l).

Dużo częściej można było zaobserwować jak niewielkie zmiany pozycji początkowej robota czy celu, potrafią wypłynąć na ostateczną trasę. Na rysunkach (c) i (d) pozycja celu uległa drobnym zmianom, a trasa robota znacznie się skróciła.

Poza tym, zachowanie robota jest podobne do omówionego w poprzednich przykładach.

4 Podsumowanie

Projekt został ukończony pomyślnie. Zaimplementowano system sterowania, który dobrze radzi sobie z docieraniem do celu i w każdej sytuacji potrafi uniknąć kolizji z przeszkodą. System ma jednak swoje wady:

– Generowane przez system trasy są często niepotymalne, a w niektórych przypadkach bardzo skomplikowane. (Dobrym przykładem tego jest rysunek 7k.) Jest to w pewnym stopniu nieuniknione w systemie reaktywnym, ale prawdopodobnie inaczej zbudowany system reaktywny radziłby sobie nieco lepiej.

– W wielu przypadkach robot wpada w pułapki z których nie potafi się wydostać, i nieustannie krąży po tych samych ścieżkach.

Dalsze pomysły na rozwinięcie systemu:

– Wykorzystanie szumu/liczb losowych do unikania oscylacji w zaułkach po takich samych ścieżkach. Dzięki temu robot, po kilku iteracjach swojej pętli, mógłby wydostać się z pułapki.

– Na bardziej skomplikowanych mapach, takich jak pokazany powyżej labirynt, zastosowanie mógłby znaleźć algorytm do podążania wzdłuż ściany.

– Pewnego rodzaju pamięć, nawet krótka, mogłaby także usprawnić radzenie sobie ze wspomnianymi problemami, choć realizacja tego pomysłu byłaby prawdopodobnie znacznie trudniejsza.

Warto też poświęcić chwilę uwagi na wszystkie „liczby", które występują w zbudowanym przez nas systemie. Same wartości liczbowe parametrów nie są w żaden sposób „ostateczne", ani nie były one dla nas najważniejsze. Naszym celem było zbudowanie struktury samego systemu – zaproponowanie zestawu zachowań oraz sposobu ich koordynacji, tak by robot dobrze radził sobie z realizacją testowanych przez nas zadań. Wartości parametrów powinny być dobierane, w sposób doświadczalny, adekwatnie do problemu, przed którym stawiamy robota. Testowaliśmy wiele różnych wartości parametrów, i dla pewnych sytuacji jedne radzą sobie gorzej od innych. Dlatego też, umieszczane w kodzie wartości różnych współczynników, powinny być czasem zmieniane. Najważniejsza jest struktura systemu.

Dodatek A Kompletny kod

Poniżej zamieszczamy przykładowe użycie wyrzeźbionego przez nas kodu. Pewne parametry, głównie te znajdujące się przed pętlą symulacyjną, nie są w żaden sposób „finalnymi" wyrobami naszej pracy, a są jedynie przykładowymi parametrami symulacji działania robota.

```
clear; close all; clc;

% Mapa.
img = ~logical(rgb2gray(imread('./mapy/mapa_3_extended.png')));
map = binaryOccupancyMap(img);

% Pojazd.
R = 0.1; L = 0.5;
vehicle = DifferentialDrive(R, L);

% Wizualizacja.
viz = Visualizer2D;
viz.mapName = 'map';
goal = [ % Cel - pozycja losowa.
    rand * size(img, 1);
    rand * size(img, 2)
]';
goal = [300, 175]; % Cel - nielosowany.
viz.hasWaypoints = true;

% Lidar.
lidar = LidarSensor;
lidar.scanAngles = linspace(-pi/2, pi/2, 10);
lidar.maxRange = 75;
attachLidarSensor(viz, lidar);

% Dane początkowe.
pose = [ % Pozycja poczatkowa - losowana.
    rand * size(img, 1);
    rand * size(img, 2);
    (rand - 0.5) * 2*pi
];
pose = [500; 500; -pi];     % Pozycja początkowa - nielosowana.
predkosc_liniowa = 1;       % Pożądana prędkość liniowa.
wL = 10; wR = 10;           % Wartości początkowe pr. kół.
dt = 1;                     % Krok czasowy.



% Pętla symulacyjna.
for t = 0:dt:6000
    % Zadanie proste kinematyki.
    % Znając prędkości kół, dostajemy prędkość liniową i kątową.
    % Znając krok czasowy znamy jak zmieni się pozycja robota.
    [v, w] = forwardKinematics(vehicle, wL, wR);
    angle = w * dt;
    x = v * cos(pose(3)) * dt;
    y = v * sin(pose(3)) * dt;

    % Aktualizacja pozycji.
    pose = pose + [x; y; angle];

```

```
    % Odczyty lidara.
    ranges = lidar(pose);


    %%%
    %%%     ZACHOWANIA I KOORDYNACJA
    %%%

    % Dystans od celu.
    dist = sqrt((pose(1) - goal(1))^2 + (pose(2) - goal(2))^2);

    % Jeśli odległość robota od celu jest mała,
    % to zmieniamy kąt skretu - im bliżej tym ciaśniejszy.
    if dist > 50
        radius = 50;
    else
        radius = (1/sqrt(50) * dist).^2;
    end


    % Prędkości obu kół wynikające z przeszkód.
    omega_obstacle_L = sum( ...
        f_sigmoidalna(0.1, radius, ranges(1:5)) .* linspace(0.01, 0.1, 5)');
    omega_obstacle_R = sum( ...
        f_sigmoidalna(0.1, radius, ranges(6:10)) .* linspace(0.1, 0.01, 5)');

    % Koordynacja obu tych prędkości.
    omega_obstacle = omega_obstacle_L - omega_obstacle_R;


    % Kąt celu.
    angle_goal = atan2(goal(2) - pose(2), goal(1) - pose(1));

    % Różnica kątowa wraz z rozwiązaniem problemu nieciągłości
    % funkcji atan2.
    angle_diff = atan2( ...
        sin(angle_goal - pose(3)), ...
        cos(angle_goal - pose(3)) ...
    );

    % Prędkość kątowa wynikająca z różnicy kątowej.
    omega_goal = angle_diff * 0.1;

    % Mechanizm antyzatrzymaniowy.
    if predkosc_liniowa < 0.01
        omega_goal = (angle_diff - pi/2) * 0.1;
    end


    % Mechanizm antykolizyjny.
    % Zmiany prędkości liniowej w funkcji odległości od przeszkody.
    % Zabezpiecza robota przed kolizją.
    if min(ranges) < 20
        predkosc_liniowa = (1/20 * min(ranges))^2;
    else
        predkosc_liniowa = 1;
    end

```

```
111
112     % Współczynnik alfa, przyjmuje wartości z zakresu 0 do 1.
113     % Odpowiada za fuzję prędkości kątowych wyzacznonych od
114     % celu oraz przeszkód.
115     alfa = f_sigmoidalna(0.1, radius, min(ranges));
116
117     % Prędkość kątowa ostateczna.
118     omega = alfa * omega_obstacle + (1 - alfa) * omega_goal;
119
120
121     % Kinematyka odwrotna - otrzymujemy prędkości
122     % obu kół wykorzystywane w następnej chwili czasowej.
123     [wL, wR] = inverseKinematics(vehicle, predkosc_liniowa, omega);
124
125
126     %%%
127     %%%     RESZTA
128     %%%
129
130     % Aktualizacja wizualizacji, co pewien krok czasowy.
131     if mod(t, 10) == 0
132         viz(pose, goal, ranges);
133         pause(0.1);
134     end
135
136     % Jeśli robot jest bardzo blisko celu to wyłącz symulację.
137     if dist < 5
138         break;
139     end
140 end
141
142
143
144 function ret = f_sigmoidalna(alpha, D_L0, D_L)
145     % Jeśli D_L = NaN, to znaczy że lidar nic nie wykrył.
146     % Możemy zatem przyjąć, że przeszkoda jest w nieskończoności.
147     D_L(isnan(D_L)) = inf;
148
149     % Funkcja sigmoidalna nastrojona według argumentów wejściowych.
150     ret = 1 ./ (1 + exp(alpha .* (D_L - D_L0)));
151 end
```