POLITECHNIKA WARSZAWSKA

Wydział Elektroniki i Technik Informacyjnych

Instytut Systemów Elektronicznych

Maciej Gąbka

nr albumu: 198404

Praca dyplomowa magisterska

Zcentralizowany algorytm unikania kolizji w rozgrywkach ligi robocup

Praca wykonana pod kierunkiem

prof. dr hab. Jarosława Arabasa

Warszawa 2011

Spis treści

1 Wstęp **4**
1.1 Cel pracy 5

2 Liga Robocup **6**
2.1 Opis projektu Robocup 6
2.2 Szczegółowe omówienie ligi Small-size (F180) 8
2.2.1 Zasady 8
2.2.2 Schemat komunikacji 8
2.3 Budowa robota w *Small-size League* 10

3 Założenia projektowe **12**

4 Player/Stage/Gazebo **14**
4.1 Koncepcja 14
4.2 Architektura symulatora 15
4.2.1 Gazebo 16
4.2.2 Player 17
4.3 Modelowanie obiektów w Gazebo 19
4.3.1 Zasady modelowania w Gazebo 0.10 19
4.3.2 Realizacja środowiska Ligi RoboCup 25

5 Architektura aplikacji sterującej drużyną robotów **27**
5.1 Opis algorytmu sterujacego drużyną 27

6 Sterowanie modelem robota z *Small-size League* **31**
6.1 Omówienie omnikierunkowej bazy jezdnej 31
6.1.1 Opis położenia kół 32

6.1.2 Opis kinematyki oraz dynamiki bazy 33
6.2 Opis algorytmu wyznaczającego prędkość liniową robota 34
6.3 Dryblowanie z piłką 35

7 Algorytmy unikania kolizji **37**
7.1 Krótki przegląd algorytmów unikania kolizji 37
7.1.1 Algorytm Bug 37
7.1.2 Algorytm VHF 39
7.1.3 Technika dynamicznego okna 41
7.1.4 Algorytmy pól potencjałowych 41
7.1.5 Algorytm CVM (`Curvature Velocity Method`) 44
7.2 Zasada działania CVM 45
7.3 Algorytm RRT (`Rapidly-Exploring Random Tree`) 51
7.4 Zalety algorytmu RRT w stosunku do CVM 55
7.5 Opis implementacji algorytmu `RRT` 56
7.5.1 Parametry zastosowanego algorytmu 60

8 Testy zaimplementowanej wersji algorytmu `RRT` **61**
8.1 Opis środowisk testowych 62
8.2 Wyniki eksperymentów w środowisku statycznym 64
8.3 Wyniki eksperymentów środowisku dynamicznym 69

9 Testy architektury `STP` **76**
9.1 Nawigacja w dynamicznym środowisku 76
9.1.1 Wyniki testu nawigacji 77
9.2 Strzelanie po podaniu 77
9.2.1 Wyniki testu 78

10 Podsumowanie **79**

A Zawartość płyty CD **80**

B Instrukcja instalacji Gazebo **81**

C Szczegóły eksperymentów **82**

Rozdział 1

Wstęp

napisać nowy wstęp

Wstęp został póki co przeklejony z pracy inżynierskiej [14]. Termin *robot* został po raz pierwszy użyty w sztuce czeskiego pisarza Karela Čapka. Słowo to określało maszynę-niewolnika zastępującą człowieka w najbardziej uciążliwych zajęciach. Semantyka tego wyrazu doskonale wyraża motywację, którą kieruje się człowiek tworząc roboty. Dlatego w drugiej połowie XX wieku, kiedy technologia elektroniczna oferowała coraz większe możliwości, robotyka jako dziedzina nauki i techniki zaczęła się gwałtownie rozwijać. Początkowo konstruowano roboty do zastosowań przemysłowych. Były to manipulatory, czyli mechaniczne ramiona o kilku stopniach swobody, na których zamontowane były odpowiednie narzędzia. Pierwszy manipulator został wykorzystany w przemyśle samochodowym w 1961 roku przez firmę General Motors.

Wraz z rozwojem techniki zaczęto myśleć o wprowadzeniu robotów do codziennego życia przeciętnego człowieka. Koncepcja robotów usługowych zakłada, że będą one wyręczać ludzi z konieczności wykonywania żmudnych czynności, takich jak sprzątanie czy też koszenie trawników. Współczesne roboty usługowe mogą nawet pełnić funkcje przewodników po muzeach. Nie wyczerpuje to oczywiście wszystkich zastosowań robotów, które coraz częściej wykorzystywane są w medycynie, wojskowości oraz ratownictwie.

Innowacyjne rozwiązania stwarzają jednak potrzebę opracowania wymagającego środowiska testowego. W przypadku robotów mobilnych gra w piłkę nożną może za takie posłużyć. Podobnie jak w przypadku ludzi zawodnik w takiej grze powinien charakteryzować się dużą zwrotnością i szybkością. Ponadto powinien sprawnie reagować na zmiany sytuacji na boisku, szybko podejmować decyzje oraz (ponieważ jest to gra zespołowa) współpracować z pozostałymi zawodnikami. Z takiego za-

łożenia wyszli twórcy *RoboCup*, czyli rozgrywek robotów w piłkę nożną. Liga jest traktowana jako pole testowe dla konstrukcji mechanicznych oraz algorytmów sterowania. W ostatnich latach stworzono także osobne przedsięwzięcie o nazwie *RoboCup Rescue*, mające na celu testowanie robotów pod kątem użyteczności w ratownictwie podczas sytuacji kryzysowych.

Liga *RoboCup* budzi zainteresowanie wielu ludzi na całym świecie, także liczne środowiska akademickie prowadzą prace z nią związane. Pomimo że na Politechnice Warszawskiej nie istnieje jeszcze w pełni funkcjonalna drużyna robotów, testowane są już algorytmy sterowania pojedynczym zawodnikiem. Jednym z podstawowych problemów, na który napotyka się przy sterowaniu robotem mobilnym, jest bezkolizyjna nawigacja w dynamicznie zmieniającym się środowisku. To właśnie ten problem należy rozwiązać przed przystąpieniem do kolejnych, bardziej zaawansowanych prac.

1.1 Cel pracy

Celem niniejszej pracy było stworzenie środowiska symulacyjnego, umożliwiającego modelowanie rozgrywki robotów w piłkę nożną oraz dającego w przyszłości możliwość testowania różnorodnych rozwiązań sterowania drużyną.

Rozdział 2

Liga Robocup

2.1 Opis projektu Robocup

Projekt Robocup, jego idea jak i historia zostały opisane w pracy inżynierskiej [14], więcej informacji na temat mistrzost można także znaleźć na oficjalnej stronie projektu `http://www.robocup.org`. W niniejszej pracy problematyka rozgrywek robotów w piłkę nożną zostanie przedstawiona jedynie skrótowo ze szczególnym uwzględnieniem budowy robota wykorzystywanego w lidze na której wzorowano się podczas prac. Głównym celem przedsięwzięcia jest stworzenie do 2050 roku drużyny w pełni autonomicznych robotów humanoidalnych zdolnych wygrać rozgrywkę z aktualnymi mistrzami świata. Aby osiągnąć zamierzony cel należy połączyć osiągnięcia z różnych dziedzin nauki. Przede wszystkim ważna jest konstrukcja zarówno mechaniczna jak i elektroniczna robota. Zawodnik powinien być wyposażony w odpowiedni zestaw czujników umożliwiających osiągnięcie pełnej autonomiczności. Z drugiej strony natomiast należy dysponować funkcjonalnym oprogramowaniem umożliwiającym współpracę wielu robotów. Projekt jest realizowany nieprzerwanie od września 1993 roku. Początkowo brali w nim udział jedynie przedstawiciele środowisk naukowych z Japonii. Rozgrywki toczone są w kilku niezależnych od siebie ligach. Aktualnie wyróżnione zostały nastepujące ligi:

- liga symulacyjna
- *Small-size League*
- *Middle-size League*

- *Standard Platform League*
- *Humanoid League*

Liga symulacyjna jest pewnego rodzaju grą, w której uczestniczące drużyny implentują program decydujący o zachowaniu zawodników. Jest ona najstarszą z lig, towarzyszy przedsięwzięciu od samego początku jego istnienia. Zachowanie robotów jest symulowane za pomocą programu zwanego *Robocup Soccer Simulator*.

Kolejną z lig jest *Small-Size League*. W rozgrywkach tej ligi drużyna składa się maksymalnie z pięciu niewielkich robotów, takich jak widoczne na fotografii 2.1. Roboty nie są jednak w pełni autonomiczne, ponieważ nie posiadają własnych sensorów. Algorytm sterujący czerpie informację o położeniu piłki oraz robotów z kamery umieszczonej centralnie nad boiskiem. Lidze tej został poświęcony w całości paragraf **??**.

Rysunek 2.1: Roboty biorące udział w *Small-Size League*
(źródło: `www.robocup.org`)

Middle-size League to rozgrywki w pełni autonomicznych robotów. W odróżnieniu od wcześniej omawianej ligi małych robotów, globalny system wizyjny jest całkowicie zakazany. Każdy robot jest wyposażony w osobny zestaw czujników wizyjnych. Zadaniem zawodników jest reagowanie na sytuację na planszy i koordynowanie swoich działań w zależności od zachowań innych graczy.

W projekcie wyróżnione są także ligi *Standard Platform League* , czyli rozgrywki piesków *Aibo* konstruowanych przez firmę *Sony* oraz liga robotów humanoidalnych. W tej ostatniej biorę udział roboty przypominające swoją budową ludzi czyli posiadać korpus, nogi, ręce oraz głowę.

2.2 Szczegółowe omówienie ligi Small-size (F180)

2.2.1 Zasady

Zbiór zasad obowiązujący w lidze podlega co roku aktualizacji przez komitet techniczny ligi. Zazwyczaj około rok przed kolejnymi mistrzostwami znane są zasady na nich obowiązujące. Początkowo w rozgrywkach wszystkie roboty korzystały z globalnego systemu wizyjnego, jednak w trakcie kolejnych zmian w przepisach dopuszczono do rozgrywek roboty z własnym systemem wizyjnym, dzięki czemu zaczęto wykorzystywać także rozproszone algorytmy sterowania.

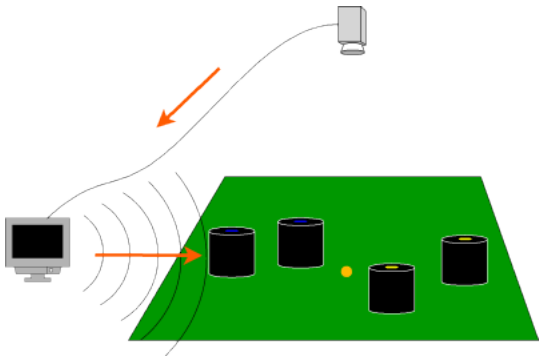
W rozgrywce na boisku o wymiarach 7.4 [m] na 5.4 [m] wyłożonej zielonym dywanem lub wykładziną biorą udział dwie drużyny składające się z maksymalnie pięciu robotów. Jeden z robotów może zostać oddelegowany do pełnienia funkcji bramkarza, jednak powinno to zostać zgłoszone przed rozpoczęciem meczu. Konstrukcja robota powinna zmieścić się w walcu o średnicy 18 [cm] oraz wysokości 15 [cm]. W przypadku robotów z własnymi sensorami wizyjnymi dopuszcza się wysokość do 22.5 [cm]. Robot może być wyposażony w urządzenie do prowadzenia piłki, jednak istnieją pewne ograniczenia dotyczące jego budowy oraz stosowania w czasie gry. Mianowicie robot może prowadzić piłkę maksymalnie przez dystans 50 [cm], po przejechaniu którego powinien albo podać ją innemu zawodnikowi, albo kopnąć przed siebie i dalej ją prowadzić. W przeciwnym wypadku sygnalizowane jest przewinienie. Natomiast konstrukcja urządzenia do dryblowania nie może uniemożliwiać kontaktu z piłką zawodnikowi z przeciwnej drużyny.

Rozgrywka jest całkowicie kontrolowana przez arbitra, który czuwa nad tym, aby regulamin był przestrzegany. Do jego zadań należy sygnalizowanie przewinień, zdobytych bramek oraz innych typowych sytuacji na boisku. Sędzia ma prawo zmienić swoją decyzje po konsultacjach z asystentem. Postanowienia arbitra są tłumaczone na sygnały elektryczne przez asystenta i wysyłane do wszystkich zawodników.

2.2.2 Schemat komunikacji

Jedną z najistotniejszych spraw podczas rozgrywki jest dostęp do informacji o położeniu piłki i pozostałych robotów. Aby rozwiązać ten problem drużyny uczestniczące w rozgrywce mogą korzystać z systemu wizyjnego widocznego na rysunku 2.2. Składa się on z kamery ustawionej centralnie nad boiskiem oraz specjalnej aplikacji

RoboCup VideoServer[1]. Obraz zarejestrowany przez kamerę jest przesyłany do komputera, na którym uruchomiony jest *RoboCup VideoServer*. Program przetwarza na bieżąco obraz z kamery i w wyniku swojego działania dostarcza informację o położeniu oraz prędkościach robotów i piłki. Dane te następnie są wykorzystywane przez rywalizujące ze sobą drużyny na potrzeby ich algorytmów sterujących zawodnikami. Algorytm sterujący jako wynik swojego działania powinien zwracać kierunek i prędkość zawodników. Ta ostateczna informacja jest wysyłana droga radiową do zawodnika. Sam program *RoboCup VideoServer* oferuje bardzo dużą funkcjonalność. Pozwala przykładowo dokonać dokładnej kalibracji nawet w przypadku, gdy boisko widziane przez kamerę ma kształt owalny (z powodu zniekształceń obrazu). Przed rozpoczęciem pracy z aplikacją należy zdefiniować początek układu współrzędnych oraz rozpoznawane przez serwer kolory.



Rysunek 2.2: Schemat komunikacji w *Small Size League*
(źródło: `www.robocup.org`)

RoboCup VideoServer może rozpoznawać nie tylko położenie poszczególnych robotów, ale także ich orientację na płaszczyźnie. Jednak do tego celu niezbędne jest zastosowanie specjalnych znaczników. Każdej z drużyn przed rozpoczęciem rozgrywki zostaje przypisana para kolorów, jakie zawiera znacznik. Dzięki zastosowaniu dwóch kolorów możliwe jest nie tylko rozróżniania robotów z różnych drużyn, ale także właśnie ich orientacji.

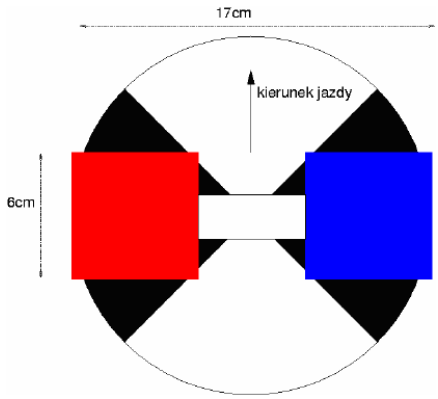
[1]Do pobrania z `http://sourceforge.net/projects/robocup-video`

(a) (b) (c)

Rysunek 2.3: Popularny model robota wykorzystywany w lidze *F180* (źródło: `www.robocup.org`)

2.3 Budowa robota w *Small-size League*

W oficjalnym regulaminie ligi nie zostały narzucone konkretne modele robotów, które mogą brać udział w rozgrywkach, jednak, obserwując kolejne mistrzostwa, łatwo zauważyć, że wśród zgłaszanych drużyn dominuje jedna konstrukcja mechaniczna. Została ona zaprezentowana na rysunkach 2.3. Podczas gry w piłkę nożną często wynika potrzeba zmiany orientacji w miejscu. W prezentowanym rozwiązaniu zdecydowano się na omnikierunkową bazę jezdną. Składa się ona z trzech kół szwedzkich, w tym dwóch niezależnie napędzanych. Koło szwedzkie posiada taką zaletę, iż dodatkowo poza obrotem wokół własnej osi umożliwia obrót wokół punktu styczności koła z podłożem oraz wokół osi rolek umieszczonych na kole. Dzięki zastosowaniu takiego rozwiązania uzyskano w pełni holonomiczną budową robota. Robot biorący udział w rozgrywkach musi być zdolny do prowadzenia piłki. Zastosowana konstrukcja jest wyposażona w urządzenie do dryblowania widoczne na rysunkach 2.3a oraz 2.3c. Zbudowane jest ono z walca nadającego piłce wsteczną rotację, przez co nie odbija się ona od robota, a także nie traci on nad nią kontroli w momencie hamowania lub obracania się. W regulaminie rozgrywek dopuszczono do stosowania jedynie urządzenia do dryblowania działające na piłkę siłą prostopadłą do podłoża rys. 2.3a (we wcześniejszych latach w



Rysunek 2.4: Znacznik umożliwiający systemowi wizyjnemu identyfikację robotów

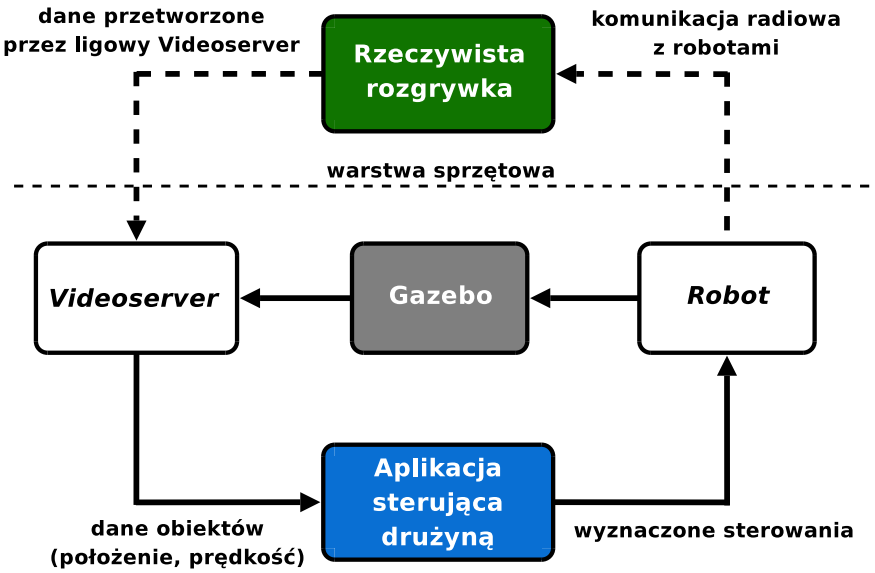
użyciu były urządzenia, w których obracany walec był umieszczony pionowo).

Ostatnim ważnym elementem, w który musi być wyposażony robot, jest znacznik (rys. 2.4). Znajduje się on w takim miejscu, aby kamera umieszczona centralnie nad boiskiem mogła go zarejestrować (przykrywa robota od góry). Znaczniki umożliwiają systemowi wizyjnemu określenie, do której drużyny należy dany robot, a także poprawne rozpoznanie jego pozycji, orientacji oraz prędkości na boisku.

Rozdział 3

Założenia projektowe

Podczas realizacji postawionego zadania starano się zachować w jak największym stopniu realia rozgrywek `Robocup`, ze szczególnym uwzględnieniem zasad ligi *Small-size League*. Z racji, iż wczesniejsze prace prowadzono na symulatorze *Player/Stage/Gazebo* zdecydowano się na dalsze prace na tej platformie. Zachowano schemat przepływu informacji z pracy inżynierskiej. Został on zamieszczony na rysunku 3.1. Stosowany w rozgrywach *Small-size League* został zamodelowany jako osobna warstwa aplikacji, komunikująca się bezpośrednio z symulatorem. Osobną warstwę stanowi także część aplikacji odpowiedzialna za sterowanie robotem.



Rysunek 3.1: Komunikacja pomiędzy warstwami aplikacji.

Dzięki takiej architekturze w łatwy sposób można przystosować aplikację do sterowania rzeczywistym robotem pobierającym dane z zewnętrznego serwera. Zdecydowano się jednak na odejście od modelu robota o napędzie różnicowym. Tego typu baza jezdna wprowadza jednak znaczące ograniczenia na sterowanie takim zawodnikiem. W symulatorze zamodelowane zostały rzeczywiste roboty biorące udział w rozgrywkach *Small-size League*. Jak już wspomniano w rozdziale 2 są to roboty posiadające trzy niezależnie napędzane koła szwedzkie. Sterowanie takim zawodnikiem jest dużo prostsze. Model został także wyposażony w urządzenie do dryblowania piłki oraz umożliwiające strzał lub podanie. Przed przystąpieniem do prac nad architekturą aplikacji sterującej dokonano przeglądu rozwiązań stosowanych w rozgrywkach. Ostatecznie zdecydowano się na architekturę `STP - Skill Tactics Play` oryginalny opis można znaleźć w [11]. Natomiast w niniejszej pracy szerzej została ona omówiona w rozdziale 5.

Rozdział 4

Player/Stage/Gazebo

Symulator `Player/Stage/Gazebo` został już opisany w pracy inżynierskiej [14] jednak z uwagi na jego ważną rolę i zmiany jakie zdecydowano się wprowadzić w symulowanym środowkisku podczas rozwijania aplikiacji zdecydowano się na krótkie przypomnienie koncepcji projektu.

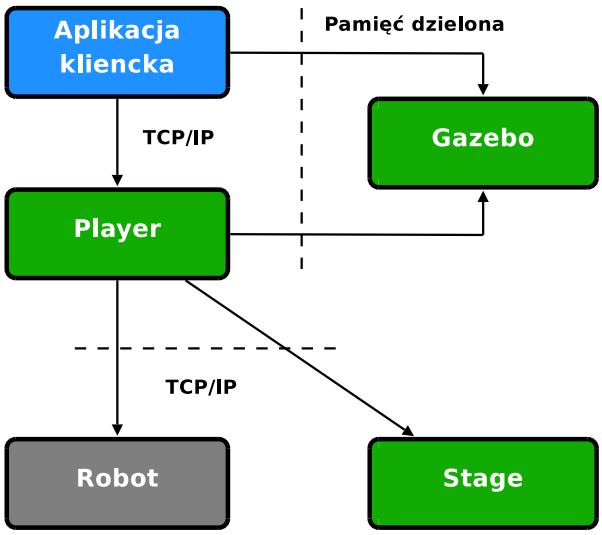
4.1 Koncepcja

Początki oprogramowania nazwanego *Project Player* (oznaczającego zestaw trzech aplikacji: *Player*, *Stage* oraz *Gazebo*) sięgają 1998 roku, kiedy to na Uniwersytecie Południowej Kaliforni w Los Angeles rozpoczęto prace nad stworzeniem oprogramowania pozwalającego na sterowanie grupą robotów mobilnych *Pioneer 2* firmy ActivMedia, będących na wyposażeniu tamtejszego laboratorium robotyki. Twórcami projektu byli Brian Gerkey i Richard Vaughan, później dołaczyli do nich Kasper Støy oraz Andrew Howard. Do tego czasu w skład stworzonego oprogramowania wchodziły: *Golem* (pozwalający na sterowanie robotem *Pioneer*) oraz *Arena* (symulator). W toku prac narodziła się idea opracowania uniwersalnego zestawu aplikacji, umożliwiającego kontrolę nad robotami niezależnie od zastosowanych rozwiązań sprzętowych, dobrze współpracującego z symulatorem oraz dostępnego za darmo (zgodnie z GNU GPL). Tak powstały aplikacje *Player* oraz *Stage*. *Player* miał za zadanie dostarczać narzędzi do sterowania podzespołami robota, a *Stage* stanowił prosty dwuwymiarowy symulator. Oprogramowanie to znalazło szeroko stosowane zarówno w uczelnianych laboratoriach, jak i wśród indywidualnych użytkowników. W 2002 roku rozpoczęto prace nad nowym symulatorem *Gazebo*, który miał służyć do mo-

delowania zachowań robotów w trójwymiarowym świecie z uwzględnieniem rzeczywistych oddziaływań fizycznych między obiektami. Całość projektu jest w dalszym ciągu intensywnie rozwijana przez deweloperów oraz aktywną społeczność użytkowników. Oprogramowanie działa na systemach Linux, Solaris, *BSD oraz Mac OSX. Szczegółowe informacje, dokumentacja oraz kody źródłowe każdej z aplikacji dostępne są pod adresem `http://playerstage.sourceforge.net/`. W trakcie prac nad opisywaną aplikacją wydana została stabilna wersja symulatora i zmieniona strona internetowa projektu, obecnie jest on dostępny pod `http://gazebosim.org/`.

4.2 Architektura symulatora

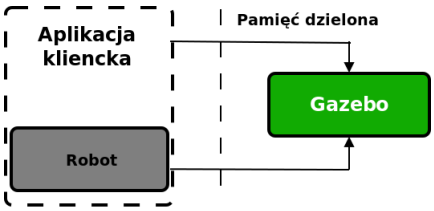
Schemat zależności pomiędzy komponentami *Player/Stage/Gazebo* został przedstawiony na rysunku 9.1. Dobrze obrazuje on rolę programu *Player*, pełniącego funkcję pośrednika pomiędzy aplikacją klienta (odpowiadającą za algorytm sterowania robotem) a rzeczywistym robotem lub symulatorami modelującymi jego zachowanie (*Stage* oraz *Gazebo*). Do komunikacji pomiędzy komponentami wykorzystywany jest protokół TCP/IP. Aplikacja kliencka łączy się z "serwerem", którego rolę pełni *Player*. Z punktu widzenia klienta nie ma znaczenia, czy interfejsy dostarczane przez *Playera* sterują robotem rzeczywistym, czy tylko jego wirtualnym odpowiednikiem (modelem) w jednym z symulatorów. Ponadto, przeniesienie kontroli pomiędzy sy-



Rysunek 4.1: Schemat komunikacji w środowisku *Player/Stage/Gazebo*

mulowanym a rzeczywistym robotem wymaga tylko nieznacznych modyfikacji kodu.

Istnieje jeszcze druga opcja – do każdego z symulatorów (*Stage* lub *Gazebo*) można odwoływać się bezpośrednio w przypadku, gdy korzystanie z *Playera* nie jest uzasadnione, lub gdy chce się dokonać pewnych modyfikacji w sposobie działania symulatora. Do tego celu zostały stworzone biblioteki (*libstage* oraz *libgazebo*), które dostarczają funkcji służących do komunikacji między aplikacją kliencką, a symulatorami. Podsumowując, struktura pakietu oprogramowania *Player/Stage/Gazebo*



Rysunek 4.2: Zrealizowany schemat komunikacji w środowisku *Player/Stage/Gazebo*

umożliwia tworzenie aplikacji sterujących robotami w sposób, który zapewnia przenośność stworzonych programów i ich działanie zarówno na symulatorach robotów, jaki i na rzeczywistych obiektach.

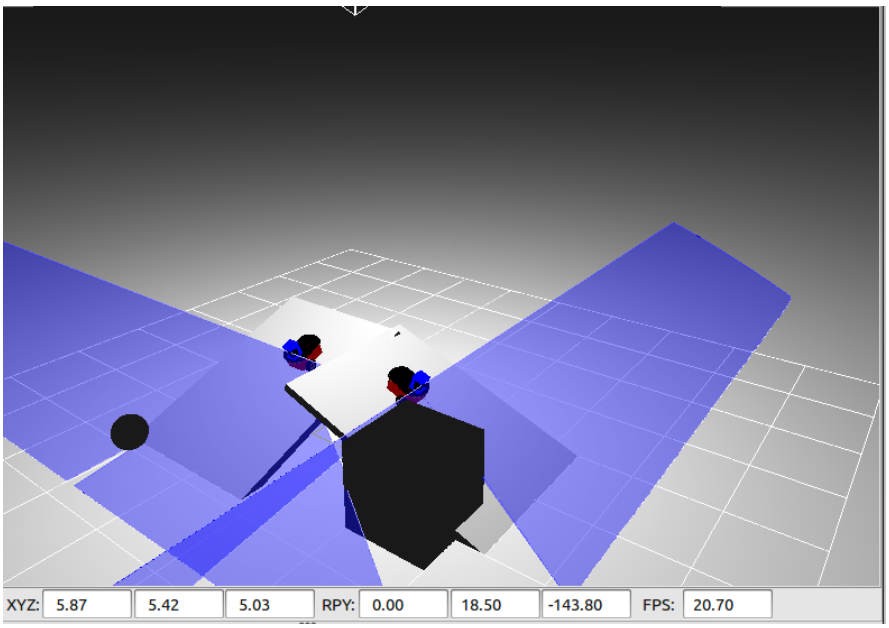
4.2.1 Gazebo

Program *Gazebo*, podobnie jak *Stage*, umożliwia symulację grup robotów mobilnych. Różnica polega na tym, że symulowane środowisko jest trójwymiarowe, uwzględnia dynamikę, cechy fizyczne oraz oddziaływania między modelami. Wiąże się to oczywiście ze wzrostem potrzebnej mocy obliczeniowej, dlatego też symulowane grupy robotów nie mogą być tak liczne, jak w przypadku *Stage*.

Gazebo zostało oparte na nastepujących komponentach:

- ODE[1], bibliotece odpowiadającej za symulację cech fizycznych brył oraz detekcję kolizji,
- OGRE[2], silniku graficznym umożliwiającym tworzenie wspomaganej sprzętowo grafiki 3D,

[1]Open Dynamics Engine, `http://www.ode.org`
[2]Object-Oriented Graphics Rendering Engine, `http://www.ogre3d.org/`



Rysunek 4.3: *Gazebo* – okno główne (wersja 0.10)

- FLTK[3], bibliotece dostarczającej przenośny interfejs użytkownika dla Gazebo,
- libXML2[4], parserze XML (stworzonym oryginalnie dla środowiska graficznego GNOME), umożliwiającym Gazebo proste wczytywanie plików konfiguracyjnych.

Gazebo pozwala na symulację standardowych sensorów (m.in. czujniki odległości, kamery, GPS). Dostarcza gotowe modele popularnych robotów (jak np. *Pioneer2DX*, *Pioneer 2AT* oraz *SegwayRMP*). Pozwala ponadto na tworzenie własnych modeli i definiowanie ich własności fizycznych (takich jak np. masa, współczynnik tarcia, sztywność), co zapewnia odpowiedni realizm symulacji i pozwala na interakcję z innymi obiektami (podnoszenie, przesuwanie brył itp.). Co więcej, dla każdego z modeli dostępne są kontrolery pozwalające na ich sterowanie, ale nic nie stoi na przeszkodzie, żeby zdefiniować własne według potrzeb i dodać je do *Gazebo*.

4.2.2 Player

Player jest serwerem sieciowym dostarczającym interfejs do kontroli nad sensorami i efektorami, w które wyposażony jest robot, za pośrednictwem protokołu TCP/IP.

[3]Fast Light Toolkit, `http://www.fltk.org/`

[4]`http://xmlsoft.org/`

Jak już wspomniano, został zaprojektowany do obsługi robotów z rodziny *Pioneer 2* firmy ActivMedia, jednak z czasem rozbudowano go dodając obsługę wielu innych urządzeń i algorytmów[5]. Najważniejsze cechy decydujące o atrakcyjności i funkcjonalności Playera to:

- niezależność od platformy i stosowanego języka programowania – program klienta łączący się z aplikacją *Playera* może zostać uruchomiony na dowolnym systemie operacyjnym; wymagane jest tylko istnienie połączenia sieciowego z robotem oraz to, żeby język programowania, w którym napisano progam kliencki, potrafił obsługiwać komunikację za pośrednictwem socketów TCP; *Player* dostarcza obecnie narzędzia ułatwiające komunikację z nim w językach C, C++, Tcl, Java, Python i Common LISP,

- brak ograniczeń na strukturę programu klienta przeznaczonego do sterowania robotem – może ona być wielowątkowa, napisana w prosty sposób reaktywny (pobierz dane - reaguj) lub bardziej skomplikowany – z perspektywy *Playera* nie ma to żadnego znaczenia,

- możliwość reprezentacji wielu urządzeń przez ten sam interfejs – dzięki temu można np. sterować dwoma zupełnie różnymi robotami za pomocą tego samego programu klienta (ich efektory odpowiadające za przemieszczanie robota będą w obu przypadkach reprezentowane przez interfejs *position*),

- wspieranie dowolnej liczby klientów; przykładowo, można stworzyć wiele połączeń do dowolnych instancji aplikacji *Playera* na dowolnym robocie i za pomocą jednej aplikacji sterować robotami, podczas gdy druga będzie odpowiedzialna za generowanie wykresów obrazujących odczyty z sensorów,

- możliwość zdalnej konfiguracji serwera w czasie jego pracy,

- łatwość testowania rozwiązań w symulatorach – *Player* jest kompatybilny ze Stage oraz Gazebo, a kod napisany w celu sterowania robotami może działać bez poprawek zarówno z symulatorem, jak i z rzeczywistym robotem.

[5]Aktualną listę można znaleźć pod adresem `http://playerstage.sourceforge.net/doc/Player-cvs/player/supported_hardware.html`

4.3 Modelowanie obiektów w Gazebo

Filozofia modelowania w *Gazebo* opiera się na tworzeniu opisu świata (o parametrach określonych w pliku *.world* za pomocą języka XML), w którym umieszczone zostają symulowane obiekty. Plik zawiera informacje o dodanych do świata modelach, ale także parametry określające przebieg symulacji, takie jak krok pracy symulatora czy sposób detekcji kolizji.

4.3.1 Zasady modelowania w Gazebo 0.10

Opis modelowanego świata powinien zawierać się w pliku głównym z rozszerzeniem *.world*. W pliku muszą być zapisane informacje związane z konfiguracją interfejsu użytkownika, sposobem renderowania sceny oraz fizyką symulacji. Elementy przykładowego pliku *.world* zostały zaprezentowane na listingach poniżej.

```
1  <?xml version="1.0"?>
2  <gazebo:world>
```

Bardzo istotnym elementem jest konfiguracja fizyki (ODE), interfejs umożliwia konfigurację globalnych parametrów takich jak:

```
3  <physics:ode>
4      <stepTime>0.001</stepTime>
5      <gravity>0 0 -9.8</gravity>
6      <erp>0.8</erp>
7      <cfm>0.05</cfm>
8      <stepType>quick</stepType>
9      <stepIters>25</stepIters>
10     <stepW>1.4</stepW>
11     <contactSurfaceLayer>0.007</contactSurfaceLayer>
12     <contactMaxCorrectingVel>100</contactMaxCorrectingVel>
13  </physics:ode>
```

Znaczenie poszczególnych parametrów jest następujące:

1. `stepTime` jest wyrażony w sekundach i określa czas pomiędzy kolejnymi aktualizacjami silnika `ODE`,
2. `gravity` jest wektorem określającym siłę grawitacji,
3. `erp` rozwija się na *error reduction parameter*, przyjmuje on wartości z przedziału od 0 do 1 i jest odpowiedzialny za redukcję błędów w połączeniach pomiędzy bryłami (problematyka zostanie szerzej poruszona na stronie 23);

`erp` ustawione na 0 powoduje, że żadna dodatkowa siła nie jest przykładana do danej bryły, natomiast wartość 1 powoduje, że wszytskie błędy połączeń zostaną naprawione w kolejnym kroku symulatora,

4. `cfm` jest skrótem od *constraint force mixing* i odpowiada za sztywność ograniczeń występujacych w kolizjach między bryłami, gdy wartość jest równa 0 ograniczenie wynikające z fizyki nie może zostać złamane, ustawienie parametru na wartość dodatnią powoduje, że ograniczenie staje się "miękkie", sprężyste,

5. `stepType` odpowiada za rodzaj używanej funkcji z `ODE` do detekcji kolizji, użytkownik może dokonać wybrou jednej z dwóch wartości:

   - *world*, używana jest wtedy funkcja `dWorldStep`, operująca na macierzy zawierającej wszystkie ograniczenia, złożoność obliczeniowa tej metody wynosi m^3, natomiast pamięciowa jest rzędu m^2, gdzie m określa ilość wierszy (ograniczeń) analizowanej macierzy,
   - *quick*, do detekcji kolizji stosowana jest metoda iteracyjna, w literaturze nazywana `SOR - Successive over-relaxation` należąca do rodziny metod Gaussa–Seidela, jej złożoność obliczeniowa jest rzędu $m * N$, a pamięciowa m, gdzie m ma znaczenie jak wyżej, natomiast N jest liczbą iteracji; dla dużych systemów metoda jest dużo bardziej wydajna jednak mniej dokładna, mogą także występować problemy z jej stabilnością; najprostszą metodą na poprawę stabilności jest zwiększanie `cfm`,

6. `stepIters` - ilość iteracji, gdy do detekcji kolizji wybrano metodę *quick*,

7. `stepW` określa czas relaksacji metody Gaussa–Seidela,

8. `contactSurfaceLayer` określa głębokość na jaką może wnikać bryła w podłoże, ustawienie parametru na małą wartość dodatnią zapobiega jitterowi w momencie, gdy ograniczenia są nieustannie tworzone i zrywane (na przykład podczas ruchu koła po podłożu),

9. `contactMaxCorrectingVel` jest maksymalną korektą prędkości jaka może wynikać z utworzonych tymczasowo kontaktów; domyślnie parametr przyjmuje nieskończoną wartość. Reducing this value can help prevent "popping" of de-

przetlumac na polski

eply embedded objects.

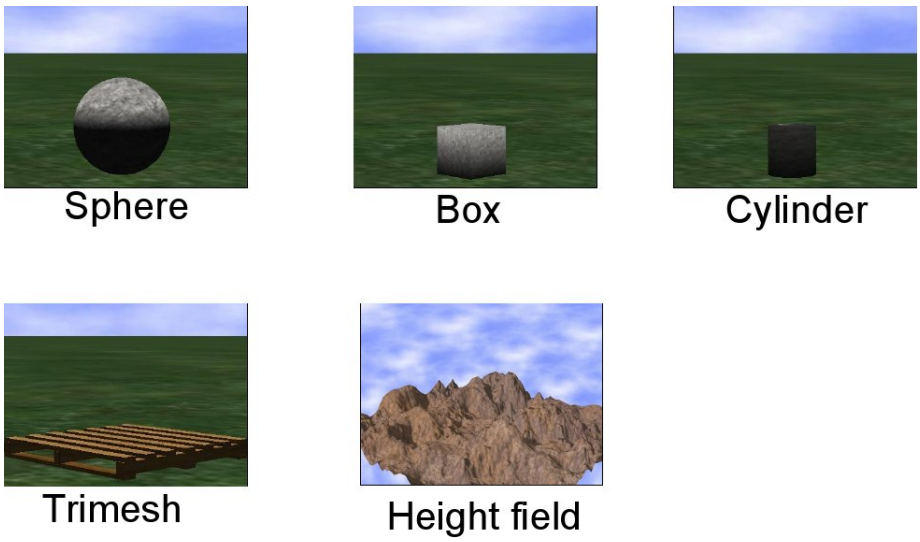
Warto wspomnieć, że niektóre parametry można zmieniać dla poszczególnych modeli lub połączeń *joints*.
Ustawienia interfejsu: typ (dostępne tylko *fltk*), rozmiaru okna i jego pozycja początkowej:

```
14      <rendering:gui>
15          <type>fltk</type>
16          <size>640 480</size>
17          <pos>0 0</pos>
18      </rendering:gui>
```

Określenie parametrów renderowanej sceny: techniki cieniowania, tekstury pokrywającej niebo (dostępne inne opcje, jak np. rodzaje oświetlenia, dodawanie mgły itp.):

```
19      <rendering:ogre>
20          <shadowTechnique>stencilAdditive</shadowTechnique>
21          <sky>
22                <material>Gazebo/CloudySky</material>
23          </sky>
24      </rendering:ogre>
25  </gazebo:world>
```

Do tak stworzonego pliku z opisem świata należy następnie dodać modele. Dodany obiekt musi zawierać co najmniej jeden element *body*, który jest złożony z elementów *geoms*. Sekcja opisująca przykładowy model piłki mogłaby wyglądać następująco:



Rysunek 4.4: Dostępne typy *geoms* – Gazebo 0.10 (źródło: [20])

```
1 <model:physical name="ball">
2     <static>false</static>
```

Na wstępie deklarowany jest model, którego parametr *static* ustawiono na *false*, co oznacza, że będzie on brał udział w symulacji fizycznej i może oddziaływać z innymi obiektami. Następnie należy zadeklarować "ciało" tworzonego obiektu:

```
3     <body:sphere name = "ball_body">
```

Wewnątrz *body*, które jest odpowiedzialne za dynamikę obiektu, należy umieścić elementy opisujące kształt obiektu (pozwalające tym samym na detekcję kolizji) – w tym przypadku zdefiniowano kulę o wymiarach i masie odpowiadającej piłce do golfa:

```
4         <geom:sphere name = "ball_geom">
5                 <xyz>0 0 0.01</xyz>
6                 <rpy>0 0 0 </rpy>
7                 <size>0.02</size>
8                 <mass>0.045</mass>
```

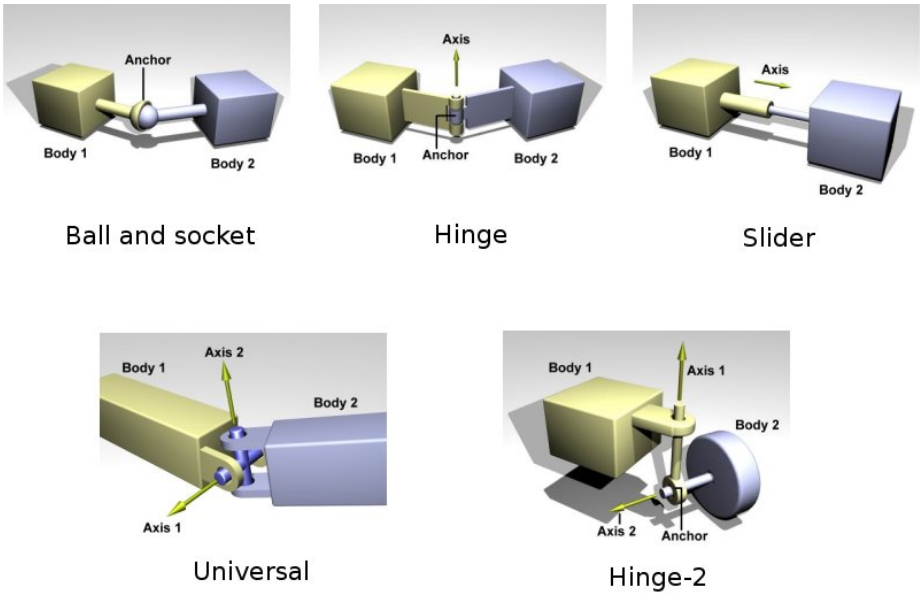
Sekcja *visual* bloku *geom* pozwala na przypisanie obiektowi wyglądu – może to być figura o dowolnym kształcie (stworzona np. w programie do grafiki 3D) i wybranej przez projektanta teksturze lub kolorze:

```
9                 <visual>
10                        <scale>0.02 0.02 0.02</scale>
```

```
11                          <size>0.02</size>
12                          <mesh>unit_sphere</mesh>
13                          <material>Gazebo/Orange</material>
14                  </visual>
15          </geom:sphere>
16      </body:sphere>
17  </model:physical>
```

Połączenia pomiędzy bryłami

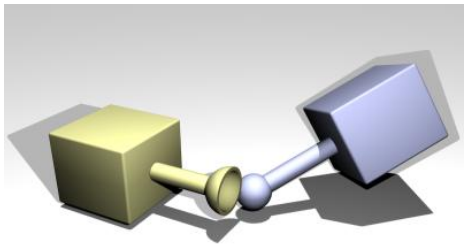
Tworząc modele, można korzystać z trzech podstawowych brył (kula, walec, prostopadłościan), a także z siatek *trimesh* oraz elementów *height field* (stworzonych do generowana terenu – por. rys. 4.4)). Żeby zamodelować robota, należy jeszcze stworzone bryły odpowiednio ze sobą powiązać. Do tego służą elementy typu *joint*, którymi można łączyć komponenty *body* (modelujące np. koła i podwozie robota). Stopnie swobody połączenia zależą od wybranego typu wiązania *joint* (dostępne typy przedstawiono na rys. 4.5). Obiekty połączone wiązaniem tworzą parę kinematyczną. *Gazebo* umożliwia nadawanie tak połączonym bryłom prędkości obrotowych względem siebie, co pozwala na modelowanie ruchomych elementów robotów.



Rysunek 4.5: Typy połączeń *joints* (Źródło: dokumentacja ODE)

Jak wspomniano wcześniej, w czasie realizowanej symulacji połączenia pomiędzy

bryłami moga ulec wypaczeniu, na skutek sił działających na model. Może to być tarcie, siła powodująca obrót bryły czy siły działające na model podczas kolizji. Sytacje taką przedstawiono na rysunku 4.6.



Rysunek 4.6: Zepsute połączenie (*joints*) (Źródło: dokumentacja ODE)

Błędy tego typu można redukować za pomocą parametru `erp` ustawianego globalnie lub dla każdego z połączeń indywidualnie.

Sterowniki modeli

Sterowanie zaprojektowanym modelem jest możliwe po uprzednim wyposażeniu go w sterownik (*controller*). Sterowniki implentowane są przez użytkownika w $C++$ jako klasa dziedzicząca po zdefiniowanym w *libgazebo* interfejsie. Klasa odpowiada za przetwarzanie danych z sensorów robota oraz pozwala na zadawanie prędkości bryłom połączonym wiązaniami. Komunikuje się on z programem sterującym za pośrednictwem interfejsów (bezpośrednio korzystając z *libgazebo* lub poprzez *Playera*). *Gazebo* oczywiście dostarcza gotowe sterowniki, pozwalające na kontrolowanie np. robota z napędem różnicowym, w wersji 0.10 dodano także przykładowy model robota holonomicznego zawarty w pliku *wizbot.world*. Aby z nich skorzystać, wystarczy przypisać wybrany sterownik do modelu robota, określić, które ze złączeń odpowiadają jego kołom i podać nazwę instancji interfejsu, za pomocą którego odbywać ma się komunikacja pomiędzy klientem a sterownikiem. W przypadku sterowania przemieszczeniem będzie to interfejs *position*, za pomocą którego możemy zadawać prędkości bryłom, ale *Gazebo* posiada też zaimplementowane interfejsy do sterowania innymi elementami np. do komunikacji z laserowymi czujnikami odległości, kamerą bądź chwytakiem manipulatora.

Listing 4.1: Przykład przypisanie sterownika do modelu

```
1 <model:physical name="pioneer_model">
2     <controller:pioneer2dx_position2d name="controller">
3         <leftJoint>left_hinge_joint</leftJoint>
4         <rightJoint>right_hinge_joint</rightJoint>
5         <interface:position name="position_iface"/>
6     </controller>
7 </model:physical>
```

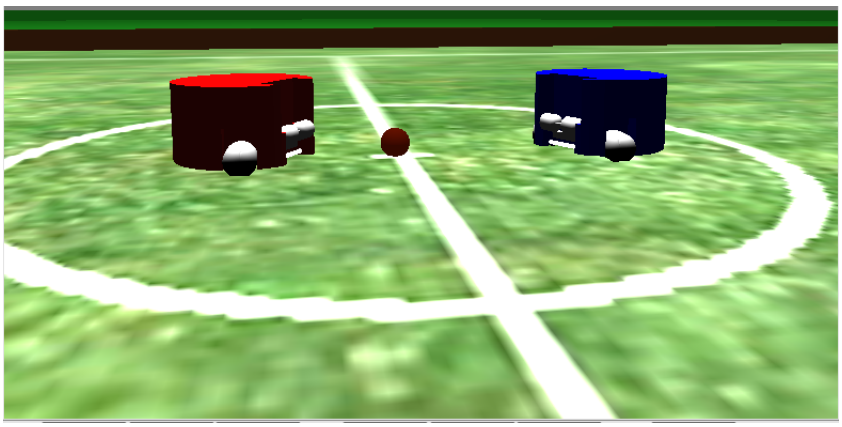
4.3.2 Realizacja środowiska Ligi RoboCup

W celu realizacji niniejszej pracy stworzono dla symulatora *Gazebo* modele odpowiadające robotom biorącym udział w *Small-size League*. Boisko zostało stworzone z wykorzystaniem rzeczywistych wymiarów($5.4[m]x7.4[m]$). Zgodnie z regułami Ligi z 2007 roku[6] utworzono również linie boiska, pozostawiając wystarczającą dla robota przestrzeń przy bandach tak, żeby było możliwe np. wykonywanie zagrań z autu. Model piłki odpowiada piłce do golfa, której używa się w oficjalnych rozgrywkach.

Wykonane modele robotów wzorowano na konstrukcji rzeczywistych robotów wykorzystywanych w *Small-size League*. Główna częścią robota jest walec o promieniu $6[cm]$ i wysokości $4[cm]$, umieszczony na trzech kołach rozmieszczonych symetrycznie na podstawie walca. Robot został dodatkowo wyposażony w *dribbler*[7] pomagający utrzymać kontrolę nad piłką. Mimo że *Gazebo* udostepnia sterownik dla napędu holo-

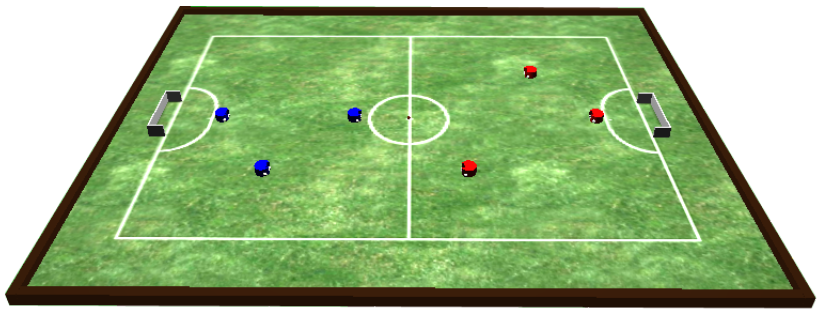
[6] `http://small-size.informatik.uni-bremen.de/rules/f180rules2007.html`

[7] urządzenie opisano w par. 2.3



Rysunek 4.7: Opracowane modele robotów (oparte na konstrukcji HMT)

nomicznego, sterowanie robotem wymagało napisania nowego kontrolera, ponieważ nie uwzględniał on obsługi *dribblera*. Kod źródłowy sterownika został zamieszczony na płycie CD razem z kodem źródłowym wykorzystywanej wersji symulatora.



Rysunek 4.8: Model boiska

W trakcie prac nad modelowaniem Ligi wprowadzono kilka poprawek w symulatorze. Aby zamodelować w pełni koło szwedzkie należało dodać możliwośc określenia w jakim kierunku dla danej bryły występuje tarcie. W tym celu w skecji opisującej bryłę dodano nowy parametr:

Listing 4.2: Parametr określający kierunek siły tarcia

```
1         <fDir1>1 0 0</fDir1>
```

Powyższa wartość oznacza, że tarcie występuje jedynie w kierunku osi OX danej bryły.

opisac jak zaimplentowano w ode brak tarcia

Rozdział 5

Architektura aplikacji sterującej drużyną robotów

Przed przystąpieniem do prac nad aplikacją sterującą drużyną robotów dokonano przeglądu stosowanych rozwiązań przez uczestników mistrzostw. Okazało się, że wstępnie opracowana architektura jest zbieżna z już istniejącym modelem nazwanym `STP - Skill Tactics Play` szerzej opisany w publikacji `STP: Skills, tactics and plays for multi-robot control in adversarial environments` [11] stworzonym przez grupę naukowców z Carnegie Mellon University. Rozwiązanie to było stosowane przez drużynę `CMD Dragons` w kilku mistrzostwach. Architektura `STP` zakłada planowanie działań drużyny na 3 poziomach:

1. Poziom położony najwyżej w hierachii: `Play`,
2. `Tactics`,
3. `Skills`.

Szczegóły algorytmu są zaprezentowane w kolejnych podrozdziałach.

5.1 Opis algorytmu sterujacego drużyną

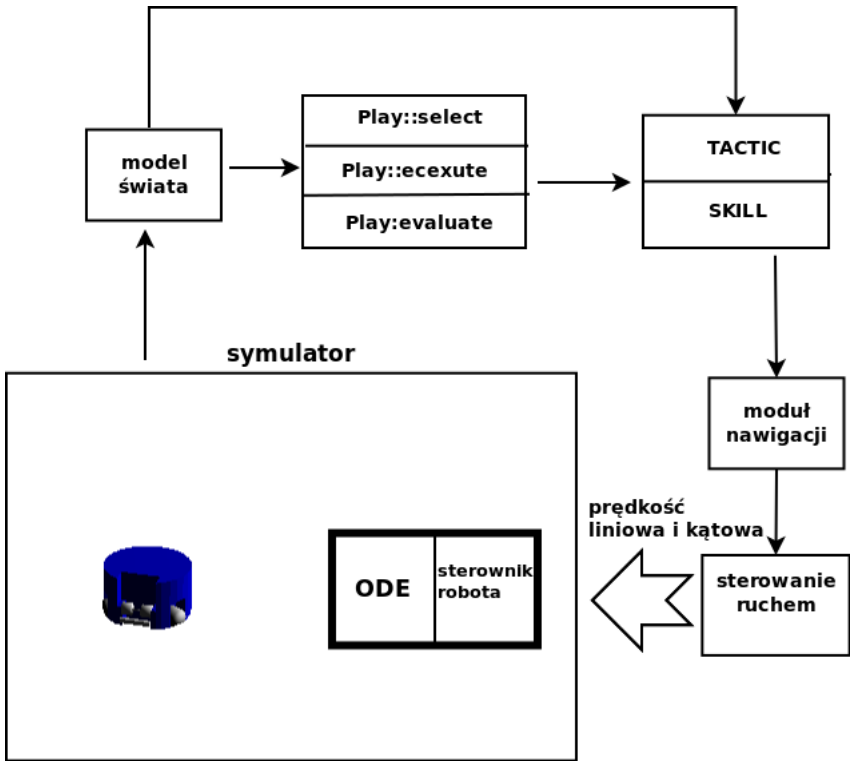
Problem sterowania i koorydynacji działań w obrębie drużyny robotów nie jest zadaniem trywialnym. Po pierwsze środowisko w którym znajdują się roboty jest dynamiczne, odrębym problemem jest analizowanie na bieżąco działań drużyny przeciwnej. Oczekiwania stawiane przed takim algorytmem są następujące:

- koordynacja działań w obrębie drużyny mająca za zadanie osiągnięcie długofalowych celów (oddanie strzału na bramkę, a w efekcie zwycięstwo),
- reagowanie w czasie rzeczywistym (*on-line*) na zachowania drużyny przeciwnej,
- budowa modelu dynamicznego świata na podstawie niepewnej informacji z czujników,
- modułowa architektura, umożliwiająca łątwą konfigurację i adaptację do bieżącej sytuacji.

Nazwa `STP` odnosi się bezpośrednio do modułu sztucznej inteligencji odpowiadającego za sterowanie drużyną robotów. Jak sam skrót wskazuje, podejscie zakłada planowanie działań na kilku poziomach. Hierarchia ma za zadanie ułatwić adaptację i paramteryzowanie poszczególnych zachowań. Oryginalny algorytm zakłada istnienie następujących modułów:

1. zawierającego informacje o świecie,
2. umożliwiający ocenę sytuacji na planszy (pozwalający na ocenę atrakcyjności zachowań, punktów docelowych etc),
3. sztucznej inteligencji; moduł ten powienien być odpowiedzialny za koordynację działań drużyny (tutaj właściwie realizowane jest `STP`),
4. modułu odpowiedzialnego za nawigację robota;moduł ten powinien być odpowiedzialny za tworzenie bezkolizyjnej ścieżki prowadzącej do zadanego celu,
5. moduł sterowania ruchem robota; moduł ten powinien wyznaczyć optymalne prędkości prowadzące do zadanego punktu (problem bezwładności robota, wyhamowanie przed punktem docelowym etc.),
6. moduł bezpośrednio odpowiedzialny za sterowanie warstwą fizyczną robota (zadawanie prędkości liniowej kątowej, uruchamianie urządzenia do prowadzenia piłki(*dribbler-a*), kopnięcie piłki).

Wszystkie warstwy algorytmu oraz przepływ informacji zostały zobrazowane na rysunku 5.1.



Rysunek 5.1: Architektura aplikacji sterującej drużyną.

Podstawowe pojęcia

Właściwe planowanie działań drużyny realizowane jest na 3 poziomach. Każdy odnosi się do innej warstwy abstrakcji.

1. Poziomem najwyżej w hierachii jest `Play`. Przez to pojęcie rozumiany jest plan gry dla całej drużyny, uwzględniana jest tutaj koordynacja poczynań pomiędzy zawodnikami. Plan zakłada przydział roli dla każdego zawodnika. W obrębie danego planu zawodnik wykonuje swoją rolę, aż do momentu zakończenia danego planu lub wyznaczenia kolejnego.

2. Przez `Tactics` rozumiany jest plan działań dla jednej roli. Można rozumieć to jako plan działań na szczeblu robota prowadzący do osiągnięcia pożądanego w danej sytuacji fektu. Przykładem może być strzał na bramkę. Robot dostaje polecenie oddania strzału na bramkę, zatem plan jego poczynań ma doprowadzić do sytuacji, w której osiągnie on pozycję umożliwiającą strzał na bramkę z zadanym powodzeniem. Plan na szczeblu pojedynczego robota jest wykonywany do momentu zmiany planu gry całej druzyny. Przykładowe plany

działań dla robotów:

- strzał na bramkę,
- podanie piłki,
- odebranie podania,
- blokowanie innego robota,
- wyjście na pozycję,
- bronienie pozycji,
- dryblowanie z piłką.

3. Pojęcie `Skills` odnosi się do konkretnych umiejętności robota, takich jak:

   - doprowadzenie piłki do celu piłki,
   - przemieszczenie robota do celu,
   - podążanie za innym robotem,

   Na tym szczeblu zachowanie robota zmieniane jest w każdym kroku gry. Z każdego zadania, w każdym momencie określone musi być przejście albo do nowego zadania bądź kontynuowanie tego samego zadania. Przykładowo jeśli zlecimy robotowi przemieszczenie do celu z piłką i piłka odskoczy robotowi, to powinien do niej podjechać i ponownie prowadzić lub jeśli nastąpi dobra okazja do strzału wykorzystać ją.

Koordynacja poczynań drużyny zapewniona jest na najwyższym poziomie. Dodatkowo każdy poziom wprowadza dodatkowe parametry wykorzystywane przy wykonywaniu zadania na najniższym poziomie. Wykonywanie każdego planu `Play` wymaga spełnienia określonych predykatów, np. czy mamy rozpoczęcie gry z autu, czy wystąpił rzut rożny.

dokonczyc rozdział opisac zrealizowana wersje algorytmu

opisać naiwny model swiata

opisać modul

Rozdział 6

Sterowanie modelem robota z *Small-size League*

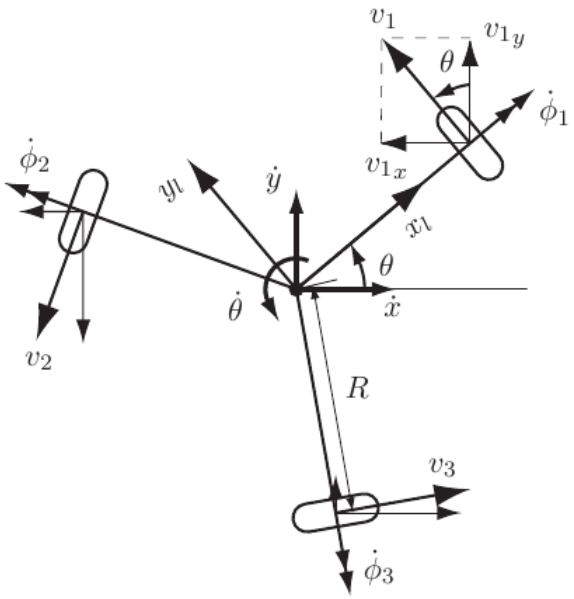
Budowę i napęd robota należy dobrać odpowiednio do postawionego zadania. W przypadku ligi *Small-size League*, najważniejszym elementem jest prostota i mobilność takiej jednostki, w pracy inżynierskiej [14] posługiwano się robotem modelem robota o napędzie różnicowym (dwa niezależnie napędzane koła), jednak zrezygnowano z tego modelu, ponieważ nie był wystarczająco funkcjonalny. Zdecydowano się natomiast na zbudowanie modelu wzorowanego na rzeczywistym zawodniku *Small-size League*.

6.1 Omówienie omnikierunkowej bazy jezdnej

Analogicznie jak podczas rzeczywistej rozgrywki, decydującym elementem jest budowa anatomiczna i zdolności motoryczne zawodników, tak samo podczas rozgrywek robotów istotna rolę odgrywa baza jezdna zawodników. W pracy inżynierskiej testom poddawany był robot o napędzie różnicowym. Baza ta posiadała jednak znaczące ograniczenia, które musiały zostać uwzględnione w algorytmach sterujących. Praktyczniejszą w użyciu jest baza omnikierunkowa. Korzystając z takiej bazy w wiekszości algorytmów robot może być traktowany jako punkt materialny.

6.1.1 Opis położenia kół

Baza omnikierunkowa składa się z umieszczonych symetrycznie co najmniej trzech kół szwedzkich tak jak zaprezentowano to na rysunku 6.1. Każde z kół posiada osobny napęd. Budowę koła omnikierunkowego przedstawiono na ilustracji 6.2. Koło



Rysunek 6.1: Rozkład kół w omnikierunkowej bazie jezdnej

szwedzkie posiada na swoim obwodzie zamontowane w odpowiedni sposób dodatkowe rolki. Umożliwiają one ruch koła w dowolnym kierunku, bez względu na to, jak koło jest zorientowane w przestrzeni. Dzięki temu umożliwia ruch robota w dowolnym kierunku, czyli należy do klasy robotów holonomicznych.

Rysunek 6.2: Konstrukcja przykładowego koła szwedzkiego

6.1.2 Opis kinematyki oraz dynamiki bazy

Podczas sterowania robotem istotnym problemem jest sposób w jaki prędkości i przyspieszenia obrotowe poszczególnych kół przekładają się na predkość/przyspieszenie kątowe i liniowe całego robota. Wszytskie poniższe obliczenia zostały wykonane przy założeniu, że koła nie ulegają poślizgowi, czyli że cały moment obrotowy śilników przekłada się na prędkość robota. Przyspieszenie liniowe i prędkość obrotowa środka masy takiego układu dane jest następującymi wzorami:

$$a = \frac{1}{M}(F1 + F2 + F3) \tag{6.1}$$

$$\dot{\omega} = \frac{R}{I}(f_1 + f_2 + f_3) \tag{6.2}$$

, gdzie f_i oznacza długość wektora siły przyłożonego do poszczególnego koła, a I jest momentem bezwładności. Przyspieszenie wzdłuż poszczególnych osi można obliczyć rozbijając siłę działającą na koło wzdłuż tychże osi, otrzymamy wtedy:

$$Ma_x = -f_1 sin\theta_1 - f_2 sin\theta_2 - f_3 sin\theta_3 \tag{6.3}$$

$$Ma_y = f_1 cos\theta_1 + f_2 cos\theta_2 + f_3 cos\theta_3 \tag{6.4}$$

Dla jednolitego cylindra moment bezwładności obliczany jest ze wzoru $I = \frac{1}{2}MR^2$, natomiast dla obręczy $I = MR^2$, gdzie R jest odpowiednio promieniem cylindra/obręczy natomiast M masą. Dla obiektów o rozkładzie masy pomiędzy obręczą, a cylindrem wprowadzany jest dodatkowy parametr α. Wzór przyjmuje wtedy postać: $I = \alpha MR^2$, gdzie $0 < \alpha < 1$. Używając zapisu macierzowego równania można przedstawić w postaci:

$$\begin{pmatrix} a_x \\ a_y \\ \dot{\omega} \end{pmatrix} = \begin{pmatrix} -sin\theta_1 & -sin\theta_2 & -sin\theta_3 \\ cos\theta_1 & cos\theta_2 & cos\theta_3 \\ \frac{MR}{I} & \frac{MR}{I} & \frac{MR}{I} \end{pmatrix} \begin{pmatrix} f_1 \\ f_2 \\ f_3 \end{pmatrix} \tag{6.5}$$

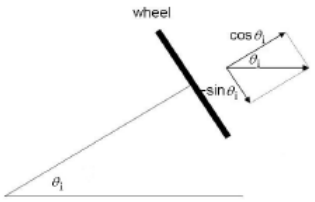
Podstawiając do powyższego wzoru $I = \alpha MR^2$ oraz zastępując $\dot{\omega}$ wyrażeniem $R\dot{\omega}$ otrzymujemy:

$$\begin{pmatrix} a_x \\ a_y \\ R\dot{\omega} \end{pmatrix} = \begin{pmatrix} -sin\theta_1 & -sin\theta_2 & -sin\theta_3 \\ cos\theta_1 & cos\theta_2 & cos\theta_3 \\ \frac{1}{\alpha} & \frac{1}{\alpha} & \frac{1}{\alpha} \end{pmatrix} \begin{pmatrix} f_1 \\ f_2 \\ f_3 \end{pmatrix} \tag{6.6}$$

Macierz z powyższego równania o wymiarze $3x3$ zostanie oznaczona symbolem C_α.

Jednak najbardziej interesujący jest sposób w jaki prędkość obrotowa kół przekłada się na prędkość liniową robota. Załóżmy, że robot porusza się wzdłuż osi OX, zatem wektor prędkości $(v_x, v_y, R\omega)$ wygląda następująco $(1, 0, 0)$. Rozważmy jedno z kół, tak jak to przedstawiono na rysunku 6.3, dokonując rozkładu wektora prędkości na dwie składowe, jedną zgodną z ruchem obrotowym dużego koła, a drugą zgodną z ruchem małych, poprzecznych kółek otrzymujemy odpowiednio prędkości $v = -sin\theta$ $v_y = cos\theta$. Przy wyznaczaniu prędkośći koła przyjęto założenie, że prędkość dodatnia powoduje obrót w kierunku wyznaczonym przez kciuk prawej dłoni, gdy pokrywa się ona z osią silnika. Otrzymujemy zatem następujące powiązanie pomiędzy prędkościami robota, a prędkościami poszczególnych silników:

$$\begin{pmatrix} v_1 \\ v_2 \\ v_3 \end{pmatrix} = \begin{pmatrix} -sin\theta_1 & cos\theta_1 & 1 \\ -sin\theta_2 & cos\theta_2 & 1 \\ -sin\theta_3 & cos\theta_3 & 1 \end{pmatrix} \begin{pmatrix} v_x \\ v_y \\ R\omega \end{pmatrix} \tag{6.7}$$



Rysunek 6.3: Prędkość liniowa dużego koła szwedzkiego i małych kółek, gdy robot porusza się wzdłuż osi OX

6.2 Opis algorytmu wyznaczającego prędkość liniową robota

Znając już powiązanie pomiędzy prędkościa obrotową kół, a prędkością liniową i obrotową robota, ostatnim elementem jest wyznaczenie prędkości prowadzących robota do zadanego punktu. Należy przy tym wziąć pod uwagę takie parametry robota jak przyspieszenie i opóźnienie. W tym celu skorzystano z metody opisanej w [15] oraz

[16]. Polega ona na dekompozycji dwuwymiarowego problemu sterowania robotem, do dwóch problemów jednowymiarowych, tak jak to zaprezentowano na rysunku 6.4. Ruch robota w kierunku osi OX i w kierunku osi OY jest rozpatrywany osobno. Podejście to jest znane w robotyce pod nazwą trapezoidalnego profilu prędkości. Zaczynając od punktu w którym robot się znajduje przyspiesza on ze swoim stałym przyspieszeniem, aż do osiągnięcia maksymalnej prędkości, następnie zaczyna hamować, tak aby zatrzymać się w punkcie docelowym. W szczególnym przypadku profil prędkości może przybrać formę trójkąta. Prędkość obliczana jest według następujących reguł:

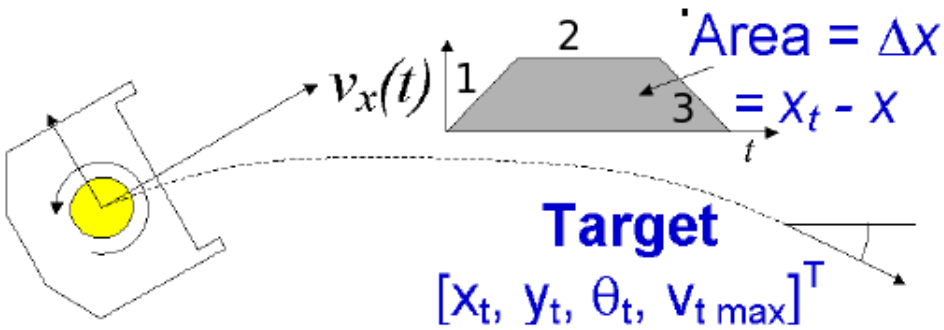
1. Jeśli bieżąca prędkość spowoduje oddalenie się robota od celu to wyhamuj do 0.
2. Jeśli robot poruszając się z bieżąca prędkością przejedzie cel to także należy zatrzymać go.
3. Gdy bieżąca prędkość przekracza maksymalną wyhamuj do prędkości maksymalnej.
4. Oblicz trójkątny profil prędkości prowadzącej do celu.
5. Jeśli w obliczonym rozkładzie prędkość przekracza w jakimkolwiek momencie maksimum to należy obliczyć trapezoidalny profil.

opiać matematykę dryblowania z piłką

- ograniczenia wynikające z budowy `dribblera`,
- wyznaczanie dopuszczalnych prędkośći.

6.3 Dryblowanie z piłką

- ograniczenia wynikające z budowy `dribblera`,
- wyznaczanie dopuszczalnych prędkośći.



Rysunek 6.4: Dekompozycja sterowania robotem w 2D na dwa niezależne zadania w 1D

Rozdział 7

Algorytmy unikania kolizji

W rozdziale zostanie szerzej zaprezentowany jeden z algorytmów unikania kolizji jakim jest RRT(`Rapidly-Exploring Random Tree`). Poruszona zostanie kwestia powodów, dla których wybrano tę, a nie inną metodę. Uzasadnione zostanie odejście od algorytmu opracowanego w ramach pracy inżynierskiej (CVM `Curvature Velocity Method`). Ponadto opisane zostaną inne metody planowania ścieżki, omówione zostaną ich właściwości. Na tej podsatwie zostanie uzasadniony wybór algorytmu RRT. Omówione zostaną także szczegóły implementacji algorytmu.

7.1 Krótki przegląd algorytmów unikania kolizji

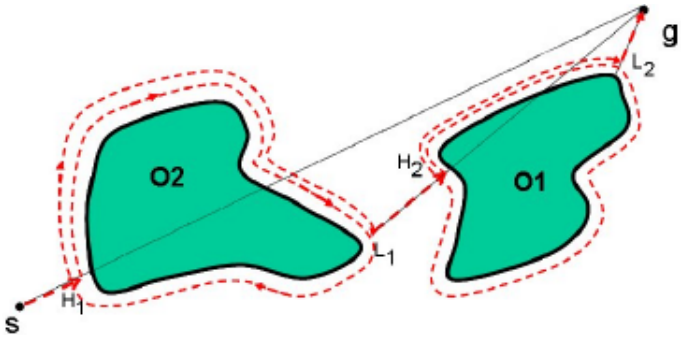
Jednym z podstawowych problemów podczas poruszania się każdej jednostki mobilnej jest wyznaczenie bezkolizijnej ścieżki prowadzącej do celu. Współczesna robotyka zna wiele algorytmów rozwiązujących z mniejszym bądź większym sukcesem to zadanie. Użycie wielu z nich jest jednak w pełni uzasadnione tylko w szczególnych okolicznościach.Poniżej zostaną zaprezentowane najbardziej znane algorytmy unikania kolizji (omówione także w pracy inżynierskiej [14]

7.1.1 Algorytm Bug

Najprostszym algorytmem unikania kolizji jest algorytm *Bug*, naśladujący zachowanie pluskwy. Gdy robot, podążając do punktu docelowego napotyka przeszkodę, okrąża ją całkowicie i zapamiętuje położenie w którym znajdował się najbliżej celu. Następnie ponownie podczas powtórnego okrążania przeszkody robot dąży do

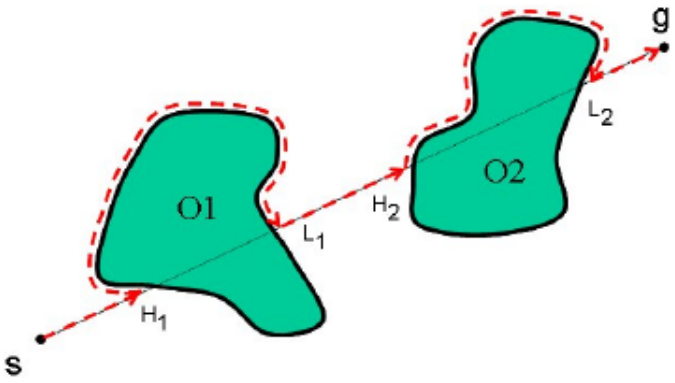
uprzednio zapamiętanego punktu po czym odrywa się od przeszkody i zaczyna podążać w stronę celu. Ścieżka wyznaczona za pomocą algorytmu została zaprezentowana na rysunku 7.1. Robot wykorzystujący ten algorytm powinien być wyposażony w dwa rodzaje czujników:

- czujnik celu – wskazujący kierunek do celu oraz umożliwiający pomiar odległości do celu
- czujnik lokalnej widoczności – umożliwiający podążanie wzdłuż konturu przeszkody



Rysunek 7.1: Bezkolizyjna ścieżka wyznaczona przez algorytm Bug

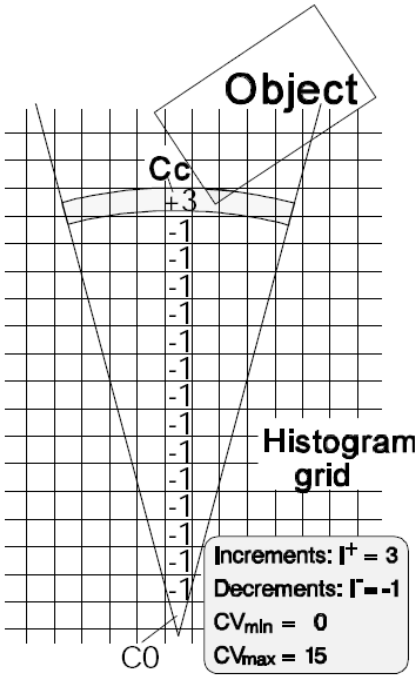
Ścieżka wyznaczona przez ten algorytm jest daleka od optymalnej. Robot niezależnie od wybranego kierunku omijania przeszkody musi ją okrążyć całkowicie. W pewnych sytuacjach całkowite okrążenie przeszkody jest niemożliwe, np jesli przeszkodą jest ściana. W takim przypadku algorytm zawodzi całkowicie. Efektywność algorytmu można poprawić sprawdzając podczas okrążania przeszkody czy punkt w którym aktualnie znajduje się robot jest poszukiwanym punktem położonym najbliżej celu. Przykładowa ścieżka wyznaczona za pomocą zmodyfikowanej wersji algorytmu jest widoczna na rysunku 7.2. Robot otacza przeszkodę w wybranym wcześniej kierunku i odłącza się od niej w momencie przecięcia prostej łączącej punkt startowy z punktem docelowym. Uzyskana w ten sposób ścieżka nadal zależy od wybranego a priori kierunku jazdy. Obie zaprezentowane wersje algorytmu Bug nie uwzględniaja ograniczeń wynikających z kinematyki robota, a w szczególności faktu, że rozpatrywany robot może być nieholonomiczny.



Rysunek 7.2: Zasada działania algorytmu Bug w wersji rozszerzonej

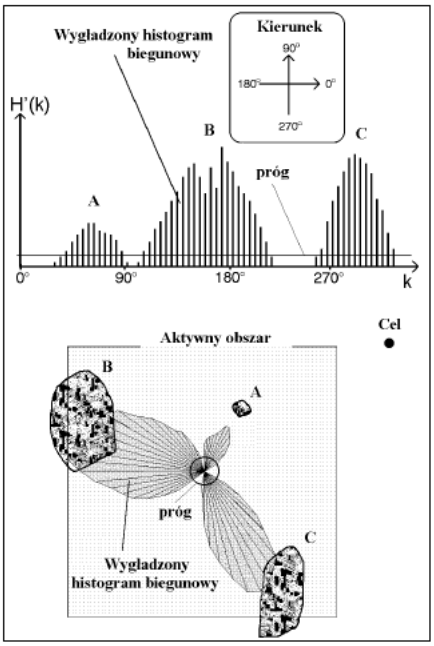
7.1.2 Algorytm VHF

Pełna anglojęzyczna nazwa metody brzmi *Vector Field Histogram*, na język polski można ją przetłumaczyć jako algorytm histogramu pola wektorowego. Należy ona do grupy metod, które w czasie rzeczywistym pozwalają na jednoczesne wykrywanie i omijanie przeszkód oraz kierowanie robota na cel. Algorytm ten został szczegółowo opisany w pracy inżynierskiej [14], więcej informacji można też znaleźć w publikacjach autorów, [8] oraz [9]. W skrócie jego zasada działania opiera się na konstrukcji dwuwymiarowego opisu świata w postaci siatki. Każdemu elementowi siatki przypisany jest poziom ufności oddający prawdopodobieństwo z jakim w danym położeniu może pojawić się przeszkoda (rysunek 7.3).



Rysunek 7.3: Zasada tworzenia dwuwymiarowego histogramu (na podstawie [9])

Tak skonstruowana mapa redukowana jest do jednowymiarowego histogramu biegunowego (rysunek 7.4),który dodatkowo wygładzany jest przez filtr dolnoprzepustowy. W ten sposób otrzymuje się informację o poziomie ufności wystąpienia przeszkody poruszając się w danym kierunku (świadomomie rezygnuje się z informacji o odległości od przeszkody). Na podstawie tego histogramu wyznaczane są sterowania dla robota. Histogram jest analizowany w poszukowaniu minimów lokalnych, wszystkie doliny, których poziom ufności znajduję się poniżej ustalonego umownie progu są rozważane przy wyznaczaniu kierunku jazdy robota. Jeśli minimów jest kilka wybierane jest to, którego kierunek prowadzi najbliżej do celu. Do poprawnego działania algorytm potrzebuje informacji o rozłożeniu przeszkód w otoczeniu robota, w oryginalnej implementacji była ona pozyskiwana z sensorów ultradźwiękowych, można je zastąpić z powodzeniem czujnikami laserowymi. Obraz video z kamery umieszczonej nad eksplorowanym światem także dostarcza tej informacji (jak w rozgrywkach *Small-size League*).



Rysunek 7.4: Histogram biegunowy dla przykładowego rozmieszczenia przeszkód (na podstawie [19])

7.1.3 Technika dynamicznego okna

Popularną, stosowaną w robotyce metodą unikania kolizji jest także technika dynamicznego okna. Algorytm ten analizuje jedynie możliwe do osiągnięcia w danej sytuacji prędkości. Więcej informacji na temat algorytmu można znaleźć w publikacji [10] lub w pracy inżynierskiej. Ujmując w jedym zdaniu algorytm polega na przeszukiwaniu zbioru dopuszczalnych prędkości liniowych i kątowych, tak aby maksymalizować funkcję celu. Sposób konstrukcji funkcji celu gwarantuje, że wybrane prędkości będą prowadzić robota w zamierzonym kierunku oraz, że nie natrafi na przeszkodę.

7.1.4 Algorytmy pól potencjałowych

Algorytm w wersji podstawowej

Kolejne podejście do problemu nawigacji robota mobilnego zaczerpnięto z fizyki. Problem znalezienia bezkolizyjnej ścieżki został sprowadzony do problemu konstrukcji funkcji opisującej rozkład energii w danym środowisku. W takim układzie dotarcie

do celu jest równoważne ze znalezieniem minimum funkcji opisującej rozkład sztucznego pola potencjałowego. Robot podąża w kierunku ujemnego gradientu tej fuknkcji, mamy zatem $c(t) = -\nabla U(c(t))$. W klasycznym podejściu sztuczne pole potencjałowe tworzy się w ten sposób, że przeszkody są źródłem ujemnego pola(odpychającego), natomiast cel emituje pole dodatnie. Ujęte jest to w następujących wzorach:

$$U(q) = U_{att}(q) + U_{rep}(q) \tag{7.1}$$

$$U_{att}(q) = \begin{cases} \frac{1}{2}\xi d^2 & \text{dla } d \leqq d^*_{goal} \\ \xi d^*_{goal} d - \frac{1}{2}(d^*_{goal})^2 & \text{dla } d \geqq d^*_{goal} \end{cases} \tag{7.2}$$

, gdzie $d = d(q, q_{goal})$ jest odległością danego położenia od celu, ε jest stałą określającą poziom przyciągania do celu, natomiast d^*_{goal} określa próg, po którym funkcja z kwadratowej przechodzi w trójkątną. Oddziaływanie przeszkód opisane jest następującym wzorem:

$$U_{rep}(q) = \begin{cases} \frac{1}{2}\eta(\frac{1}{D(q)} - \frac{1}{Q^*})^2 & \text{dla } D(q) \leqslant Q^* \\ 0 & \text{dla } D(q) > Q^* \end{cases} \tag{7.3}$$

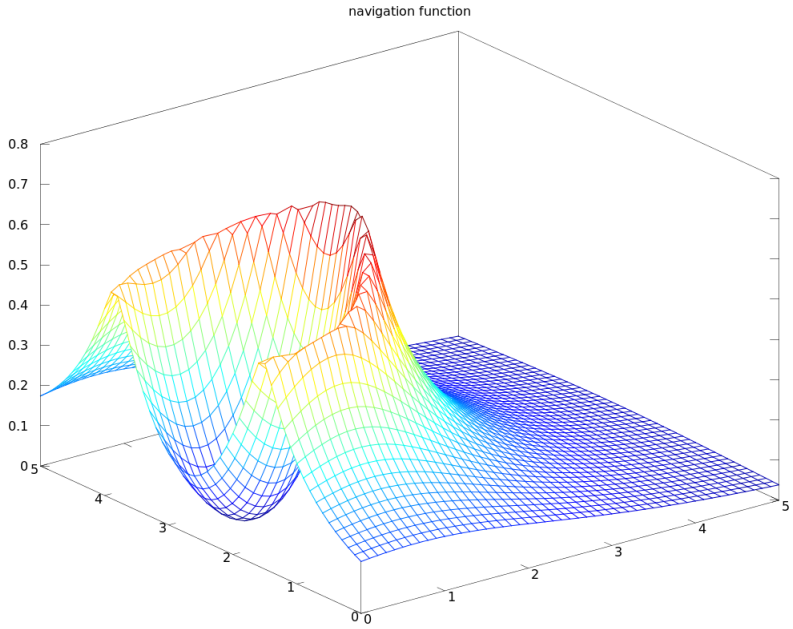
,gdzie Q^* jest zasięgiem pola odpychającego przeszkody, natomiast η odpowiada za siłę tego pola. Kierunek w którym podążać ma robot wyznaczany jest ze wzoru:

$$-\nabla U(q) = -\nabla U_{att}(q) - \nabla U_{rep}(q) \tag{7.4}$$

Powyższe podejście w stosunkowo prosty sposób umożliwa wyznaczenie kierunku bezkolizyjnej ścieżki do celu, jednak posiada pewną dość istotną wadę, mianowicie w pewnych, szczególnych sytuacjach robot może utknąć w minimum lokalnym. Sposób w jaki konstruowana jest funkcja celu w żaden sposób nie wyklucza występowania minimów lokalnych. W przypadku, gdy sterowany robot utknie w lokalnym minimum, konieczna jest reakacja ze strony wyższej warstwy nawigującej maszyną, przykładowo można wykorzystać algorytm błądzenia losowego.

Funkcja nawigacji

Istnieje jednak pewna szczególna funkcja, która posiada tylko globalne minimum, została ona zdefiniowana w publikacjach [17],[18]. Przykładowy wykres funkcji nawigacji jest zamieszczony poniżej (rysunek 7.5). Siła przyciągania tego minimum zależy od kilku istotnych parametrów, które zawarte są we wzorze opisującym rozkład sztucznego potencjału. Dla bieżącego położenia robota q funkcja nawigacji



Rysunek 7.5: Funkcja nawigacji dla $\kappa = 10$

zdefiniowana jest następująco:

$$\phi = \frac{(d(q, q_{goal}))^2}{(\lambda\beta(q) + d(q, q_{goal})^{2\kappa})^{\frac{1}{\kappa}}} \tag{7.5}$$

gdzie $\beta(q)$ jest iloczynem funkcji odpychających zdefiniowanych dla wszytskich przeszkód:

$$\beta \triangleq \prod_{i=0}^{N} \beta_i(q) \tag{7.6}$$

Natomiast dla każdej przeszkody($i > 0$) funkcja określona jest następująco:

$$\beta_i(q) = (d(q, q_i))^2 - r_i^2 \text{ dla i} = 1...\text{N, gdzie N jest liczbą przeszkód} \tag{7.7}$$

Powyższa funkcję definiuje się dla każdej z przeszkód o promieniu r_i, której środek znajduje się w punkcie q_i. Jak łatwo zauważyć, funkcja ta przyjmuje ujemne wartości wewnątrz okręgu opisującego przeszkodę, natomiast dodatnie na zewnątrz okęgu. Dodatkowo definiuje się funkcję $\beta_0(q) = -(d(q, q_0))^2 + r_0^2$, w której parametry r_0 oraz q_0 oznaczają odpowiednio promień świata w którym porusza się robot oraz środek tego świata. Wzór 7.5 posiada dwa istotne parametry, λ ogranicza przeciwdziedzinę do przedziału $[0, 1]$, gdzie wartość 0 jest tożsama z osiągnięciem celu, natomiast 1

jest osiągna na brzegu każdej z przeszkód. Drugi parametr κ odpowiednio dobrany powoduje, że blisko celu wykres funkcji ϕ przybiera kształt misy. Zwiększanie parametru κ powoduje przesuwanie globalnego minimum w kierunku położenia punktu docelowego, zwiększa zatem oddziaływanie przyciągającego pola emitowanego przez punkt docelowy.

Wykres funkcji nawigacji przedstawiony na rysunku 7.5 został sporządzony dla następujących parametrów(wszystkie jednostki wyrażone są w metrach):

1. eksplorowany świat ograniczony jest okręgiem o środku w punkcie $q_0(2.7; 3.7)$ i promieniu $r_0 = 7.4$,
2. $\lambda = 0.2$,
3. $\kappa = 10$,
4. przeszkody umiejscowione są w punktach $(2; 2), (3; 3)(4; 4), (3.5; 3.5)$ i mają stały promień, odpowiadający modelowi robota zastosowanemu podczas doświadczeń $r_i = 0.14$,
5. punkt docelowy ma współrzędne $q_g(2.7; 0.675)$,
6. funkcja nawigacji została wykreślona z krokiem 0.1.

7.1.5 Algorytm CVM (`Curvature Velocity Method`)

W pracy inżynierskiej [14] jako docelowy algorytm unikania kolizji wybrany został właśnie `CVM`. Metoda krzywizn i prędkości (ang. *Curvature Velocity Method*) została zaproponowana przez R. Simmonsa w [6]. Należy ona do grupy metod bazujących na przeszukiwaniu przestrzeni prędkości, a jej działanie jest zbliżone do techniki dynamicznego okna zaprezentowanej w 7.1.3.

Spośród wszystkich par (v, ω) wybierana jest taka, która osiąga największą wartość funkcji celu. Podejście takie umożliwia uwzględnienie przede wszystkim ograniczeń kinematycznych, ale także i dynamicznych. Wyznaczone przez algorytm sterowanie na następny krok definiuje łuk $c = \frac{\omega}{v}$, po którym będzie poruszał się robot do chwili wyznaczenia kolejnego sterowania.

Funkcja celu została tak skonstruowana, aby preferować prędkości, które prowadzą do celu, ale jednocześnie nie powodują kolizji. Ponadto dołożone zostało kryterium na maksymalizację prędkości liniowej robota. Funkcja celu jest zatem sumą

ważoną trzech składowych:

$$F(v,\omega) = \alpha_1\mathcal{V}(v,\omega) + \alpha_2\mathcal{D}(v,\omega) + \alpha_3\mathcal{G}(v,\omega)) \tag{7.8}$$

gdzie:

- $\alpha_1, \alpha_2, \alpha_3$ współczynniki wagowe,
- funkcja $\mathcal{V}(v,\omega)$ określa stosunek między ocenianą prędkością liniową a maksymalną,
- funkcja $\mathcal{D}(v,\omega)$ określa odległość od najbliższej przeszkody na którą napotka robot poruszający się po trajektorii wyznaczonej przez (v,ω),
- funkcja $\mathcal{G}(v,\omega)$ odpowiada za kierowanie się robota na cel.

Zachowaniem robota można sterować poprzez zmianę wartości współczynników wagowych. W skrajnych przypadkach, gdy któryś ze współczynników zostanie wyzerowany, traci on całkowicie wpływ na trajektorię po której porusza się robot. Szczegółowy opis poszczególnych składowych, jak i samego działania metody został zaprezentowany w paragrafie 7.2.

7.2 Zasada działania CVM

Omawianie mechanizmu wyboru najlepszego sterowania z wykorzystaniem algorytmu *CVM* należy rozpocząć od przypomnienia funkcji celu (7.8)

$$F(v,\omega) = \alpha_1\mathcal{V}(v,\omega) + \alpha_2\mathcal{D}(v,\omega) + \alpha_3\mathcal{G}(v,\omega))$$

W powyższym równaniu składowe odpowiadające za kierowanie na cel oraz za maksymalizację prędkości liniowej zostały zdefiniowane następująco:

$$\mathcal{V}(v,\omega) = \frac{v}{V_{max}} \tag{7.9}$$

$$\mathcal{G}(v,\omega) = 1 - \frac{|\theta_{cel} - T_{alg}\omega|}{\pi} \tag{7.10}$$

gdzie:

- V_{max} – maksymalna prędkość liniowa z jaką może poruszać się robot

- T_{alg} – czas na jaki wystawiane jest obliczone sterowanie
- θ_{cel} – orientacja do celu w układzie współrzędnych robota

Natomiast zdefiniowanie składowej odpowiedzialnej za omijanie przeszkód wymaga określenia pewnych dodatkowych funkcji wykorzystywanych w algorytmie. Należy dokonać przekształcenia opisu przeszkód we współrzędnych kartezjańskich do przestrzeni prędkości. Dokonywane jest to za pomocą odległości do punktu p, w którym wystąpi kolizja z przeszkodą o, jeśli robot będzie poruszał się po trajektorii $c = \frac{\omega}{v}$. Wartość ta jest oznaczana jako $d_c((0,0),p)$, gdzie p jest punktem zderzenia z przeszkodą. Następnie należy zdefiniować funkcję odległości robota od przeszkody:

$$d_v(v,\omega,p) = \begin{cases} d_c((0,0),p_i & \text{dla } v \neq 0 \\ \infty & \text{dla } v = 0 \end{cases} \tag{7.11}$$

Dla danego zbioru przeszkód O zawsze wybierana jest odległość do najbliższej przeszkody leżącej na trajektorii. Zatem funkcję dla danego zbioru przeszkód O można zapisać następująco:

$$D_v(v,\omega,O) = \inf_{o \in O} d_v(v,\omega,o) \tag{7.12}$$

W rzeczywistości jednak informacja o położeniu przeszkód jest ograniczona do pewnego obszaru, który może wynikać z zasięgu czujników w jakie wyposażony jest robot, bądź ze sztucznego ograniczenia zasięgu działania algorytmu w przypadku, gdy możliwy jest dostęp do globalnej informacji o położeniu przeszkód. Zasięg algorytmu będzie w dalszej część pracy oznaczany jako L. Funkcja (7.12) jest zatem ograniczona od góry w sposób następujący:

$$D_{ogr}(v,\omega,P) = \min(L, D(v,\omega,P)) \tag{7.13}$$

Po uprzednim zdefiniowaniu niezbędnych funkcji można podać postać funkcyjną składowej funkcji celu odpowiadającej za unikanie kolizji:

$$\mathcal{D}(v,\omega) = \frac{D_{ogr}}{L} \tag{7.14}$$

Aby sposób obliczania odległości do przeszkody był możliwie jak najprostszy, każda przeszkoda jest reprezentowana w algorytmie jako okrąg o promieniu r (wynikającym z jej rozmiarów) i współrzędnych środka (x_0, y_0). Można zatem dość łatwo wyznaczyć odległość, jaką przebędzie robot poruszający się po łuku wyznaczonym

przez parę (v, ω) do momentu przecięcia z okręgiem. Zostało to zobrazowane na rysunku 7.6.

Dla robota o początkowej orientacji zgodnej z osią OY, poruszającego się po trajektorii o krzywiźnie c przecinającej okrąg opisujący przeszkodę w punkcie $P(x, y)$ prawdziwe są następujące wzory:

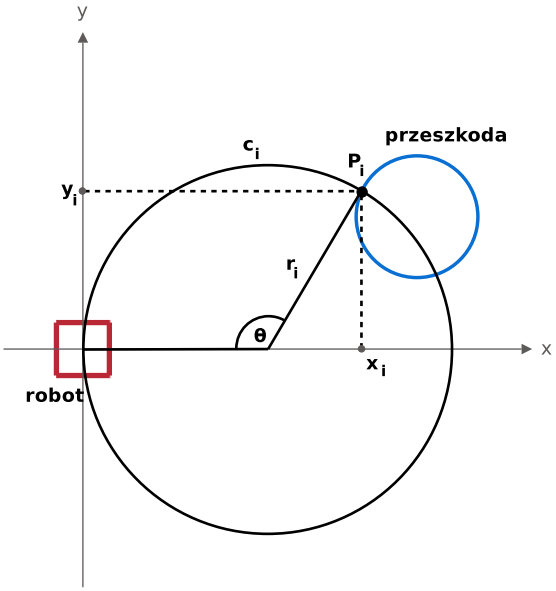
$$\theta = \begin{cases} \text{atan2}(y, x - \frac{1}{c}) & \text{dla } c < 0 \\ \pi - \text{atan2}(y, x - \frac{1}{c}) & \text{dla } c > 0 \end{cases} \tag{7.15}$$

$$d_c((0,0), P) = \begin{cases} y & \text{dla } c = 0 \\ |\frac{1}{c}|\theta & \text{dla } c \neq 0 \end{cases} \tag{7.16}$$

Zastosowana we wzorze (7.15) funkcja *atan2* zwraca wartości z przedziału $[-\pi; \pi]$, zatem uwzględnia w której ćwiartce leży kąt:

$$\text{atan2}(y, x) = \begin{cases} \arctan \frac{|y|}{|x|}\text{sgn}(y) & \text{dla } x > 0 \\ \frac{\pi}{2}\text{sgn}(y) & \text{dla } x = 0 \\ \pi - \arctan \frac{|y|}{|x|}\text{sgn}(y) & \text{dla } x < 0 \end{cases} \tag{7.17}$$

Jednakże obliczanie odległości od przeszkody dla każdego sterowania (v, ω) w sposób omówiony powyżej jest czasochłonne. Stąd konieczne jest kolejne uproszczenie. Łatwo zauważyć, że z punktu $(0, 0)$ w którym znajduje się robot można



Rysunek 7.6: Obliczanie odległości od przeszkody

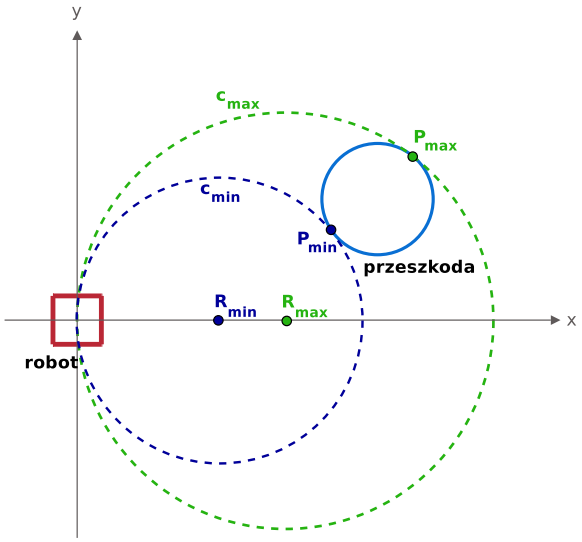
poprowadzić dwa łuki styczne do okręgu opisującego przeszkodę p. Zatem na zewnątrz krzywych stycznych odległość $d_c((0,0),p)$ jest nieskończona. Natomiast dla krzywizn zawierających się pomiędzy łukami stycznymi można dokonać przybliżenia wartością stałą. Omawiana sytuacja została zobrazowana na rysunku 7.7. Aby wyznaczyć okręgi styczne do okręgu opisującego przeszkodę należy znaleźć takie r dla którego poniższy układ równań ma dokładnie jedno rozwiązanie:

$$\begin{cases} (x-x_0)^2+(y-y_0)^2 = {r_0}^2 \\ (x-r)^2+y^2 = r^2 \end{cases} \tag{7.18}$$

Gdzie (x_0, y_0) jest środkiem okręgu opisującego przeszkodę, natomiast r_0 jego promieniem. W wyniku takiego postępowania można otrzymać punkty styczności P_{max} oraz P_{min} oraz odpowiadające im krzywizny c_{min} oraz c_{max}. Ostatecznie można już dokonać przybliżenia odległości po łuku od rozpatrywanej przeszkody p w następujący sposób:

$$d_v(v,\omega,p) = \begin{cases} \min(d_c((0,0),P_{max}), d_c((0,0),P_{min})) & c_{min} \leqslant c \leqslant c_{max} \\ \infty & \text{w p.p.} \end{cases} \tag{7.19}$$

Zaprezentowane rozumowanie prowadzi do sytuacji, w której zbiorowi przeszkód O odpowiada zbiór przedziałów krzywizn o stałej odległości od przeszkody. W dalszej części pracy przedział będzie rozumiany jako trójka liczb postaci $([c_1, c_2], d_{1,2})$, gdzie c_1,c_2 to krzywizny okręgów wyznaczających ten przedział, natomiast $d_{1,2}$ jest odległością zdefiniowaną równaniem (7.19).



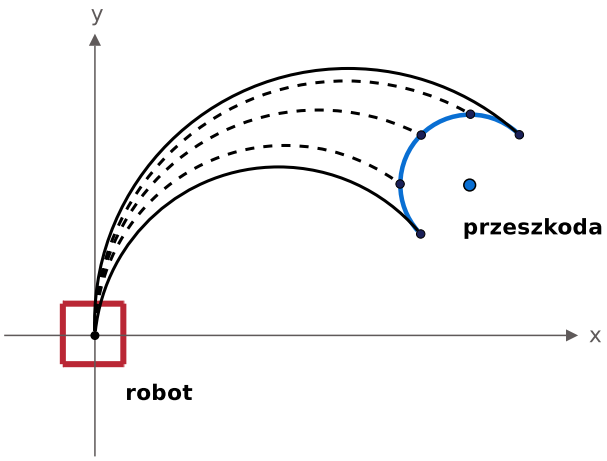
Rysunek 7.7: Zasada wyznaczania przedziału (c_{min},c_{max})

Ze zbioru przedziałów należy następnie utworzyć listę przedziałów $\mathbb{S}$, tak, aby zawierała przedziały s rozłączne o odległości do najbliższej przeszkody (zgodnie z równaniem (7.12)). W oryginalnej pracy na temat *CVM* [6] zaproponowany został algorytm (7.1) realizujący to zadanie.

Algorytm 7.1 tworzy listę rozłącznych przedziałów $\mathbb{S}$

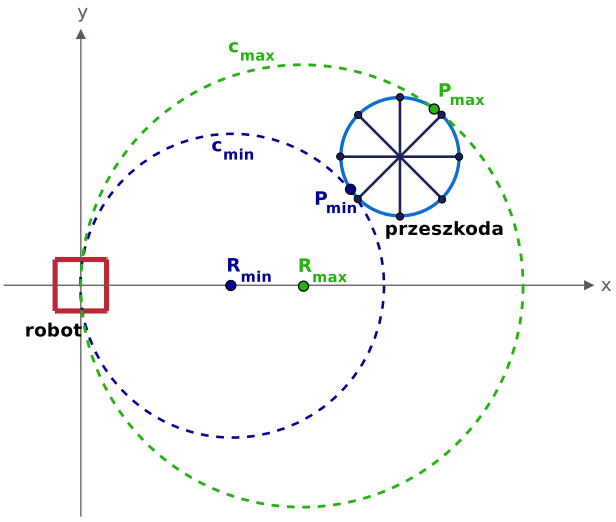
$\mathbb{S} := ((-\infty; \infty), L)$
wybierz kolejny nie dodany przedział $([c_{min}; c_{max}], d)$
for all $s \in \mathbb{S}$ **do**
  if zbiory są rozłączne **then**
    nic nie rób
  else if s zawiera się w $([c_{min}; c_{max}], d)$ **then**
    ustaw odległość $s.d = min(s.d, d)$
  else if s zawiera $([c_{min}; c_{max}], d)$ **then**
    if $d < s.d$ **then**
      podziel s na trzy przedziały:
      $([s.c_{min}; c_{min}], s.d)$, $([c_{min}; c_{max}], d)$, $([c_{max}; s.c_{max}], s.d)$
    else
      nic nie rób
  else if s częściowo zawiera $([c_{min}; c_{max}], d)$ **then**
    if $d < s.d$ **then**
      podziel s na dwa przedziały
      if $c_{max} > s.c_{max}$ **then**
        podziel na przedziały $([s.c_{min}; c_{min}], s.d)$ $([c_{min}; s.c_{max}], d)$
      else
        podziel przedziały $([s.c_{min}; c_{max}], d)$ $([c_{max}; s.c_{max}], s.d)$
    else
      nic nie rób

W zależności od rozmieszczenia przeszkód w środowisku przedziały wyznaczone przez krzywizny c_{min} oraz c_{max} charakteryzują się różną szerokością. Często przybliżenie zbioru długości krzywych należących do $[c_{min}, c_{max}]$ jedną stałą wartością jest niewystarczające. Zazwyczaj przedział ten zawiera zbiór łuków których odległość od przeszkody jest istotnie mniejsza niż ta wynikająca ze wzoru (7.19). Sytuacja taka została zilustrowana na rysunku 7.8.



Rysunek 7.8: Ukazanie sensu zastosowania segmentacji

Jednym z możliwych rozwiązań powyższego problemu jest podzielenie przedziału $[c_{min}, c_{max}]$ na kilka podprzedziałów i do każdego z nich zastosowanie równania (7.19) w celu wyznaczenia odległości od przeszkody. W pracy przyjęto rozwiązanie analogiczne jak opisane w [7]. Pierwszym etapem jest wyznaczenie punktu leżącego na okręgu reprezentującym przeszkodę położonego najbliżej robota zgodnie z metryką euklidesową. Następnie okrąg jest dzielony symetrycznie na k segmentów. Postępowanie takie zostało przedstawione na rysunku 7.9. Dla każdych dwóch sąsiadujących ze sobą punktów leżących pomiędzy punktami styczności P_{min} oraz P_{max} wyznaczane są krzywizny c_1 oraz c_2 i tworzony jest przedział. Jako odległość przyjmowana jest krótsza z odległości zgodnie ze wzorem (7.19). Tak otrzymane przedziały są dodawane do listy za pomocą algorytmu 7.1.



Rysunek 7.9: Efekt podziału okręgu na części

7.3 Algorytm RRT (Rapidly-Exploring Random Tree)

Kolejnym zupełnie odmiennym podejściem w planowaniu bezkolizyjnej ścieżki jest losowe przeszukiwanie dopuszczalnej przestrzeni położeń robota. Szczegółowe informacje na jego temat można znaleźć w [2] oraz w [3]. Algorytm RRT jest wydajnym algorytmem, który w czasie rzeczywistym może wyznaczyć drogę do celu, jednak nie jest ona optymalna pod względem dynamiki, kinematyki ani długości. Podstawową strukturą danych na której operuje algorytm jest drzewo. Jako korzeń przyjmowany jest stan reprezentujący położenie robota w momencie uruchomienia algorytmu. Budowa drzewa polega na dodawaniu kolejnych węzłów do drzewa w kierunku punktu obranego w danym kroku jako docelowy. Tymczasowy stan docelowy generowany jest w następujący sposób:

Algorytm 7.2 Funkcja obliczająca stan docelowy

```
function chooseTarget(goal:state) state;
p = uniformRandom in[0.0...1.0]
if 0.0 < p < goalProb then
  return goal;
else if goalProb < p < 1.0 then
  return RandomState()
```

Na listingu 7.2 użyta została funkcja `RandomState()` zwracająca losowy punkt ze zbioru wszystkich możliwych położeń robota. W najprostszej realizacji może ona zwracać współrzędne punktu dobierane z rozkładem jednostajnym z zadanego przedziału. W każdej iteracji algorytmu tymczasowy stan docelowy obliczany jest na nowo. Aktualne drzewo, na którym operuje algorytm przeszukiwane jest w celu znalezienia węzła `nearest` znajdującego się najbliżej tego celu. Węzeł `nearest` rozszerzany jest w kierunku tegóż celu za pomocą funckji `extend`. Operacja ta w najprostszym ujęciu polega na stworzeniu nowego węzła przesuniętego względem `nearest` o odcinek `rrtStep` w kierunku tymczasowego celu. Istotne jest, aby sprawdzić czy nowo wygenerowany stan znajduje się w dopuszczalnej przestrzeniu stanów $. W najprostszym wariancie można zastosować heurystykę sprawdzającą jedynie czy położenie to nie koliduje z żadną z przeszkód znajdujących się w danej sytuacji oraz czy robot jest w stanie do niego dotrzeć. Bardziej zaawansowane heurystyki powinny sprawdzać czy na przykład podczas przemieszczania się do nowej pozycji nie nastąpi kolizja z dynamiczną przeszkodą. W ogólnym ujęciu algorytm `RRT` wygląda zatem następująco:

Algorytm 7.3 Ogólna zasada algorytmu `RRT`

```
zbuduj model eksplorowanego świata ↦ env:environment;
zainicjuj stan początkowy ↦ initState:state;
zainicjuj stan końcowy ↦ goalState:state;

while goalState.distance(nextState) > minimalDistance do
  wybierz tymczasowy cel: target = chooseTarget(goalState)
  nearestState =  nearest(tree, target)
  extendedState = extend(env, nearest, target)
  if extendedState != NULL then
    addNode(tree, extended)
return tree;
```

W opisie algorytmu 7.3 użyte zostały nie omówione jeszcze funkcje, `distance` zwracająca odległość w sensie euklidesowym między dwoma węzłami drzewa, `addNode` dodająca węzeł do bieżącego drzewa. Parametr `rrtStep` jest natomiast wyrażony w metrach. Tak zaimplementowana wersja algorytmu może być zastosowana jako skuteczne narzędzie do bezkolizyjnej nawigacji robotem, jednak jego wydajność można poprawić poprzez modyfikacje opisane w kolejnym podrozdziale.

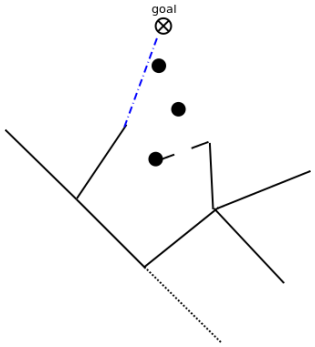
Modyfikacje algorytmu, czyli przejście z `RRT` do `ERRT`

Omawiany algorytm można zmodyfikować pod kątem szybkości działania. Po pierwsze elementy drzewa można przechowywać w strukturach ułatwiających przeszukiwanie drzewa, w literaturze stosowane są w tym celu `KD-drzewa`. Kolejną modyfikacją jest zapamiętywanie losowo wybranych węzłów ze ścieżki znalezionej w poprzednim uruchomieniu algorytmu. W takim wariancie tymczasowy punkt docelowy obierany jest następująco:

Algorytm 7.4 Zmodyfikowana funkcja obliczająca stan docelowy

```
function chooseTargetExt(goal:state) state;
p = uniformRandom in[0.0...1.0]
i = uniformRandom in[0...number_of_waypoints − 1]
if 0.0 < p < goalProb then
  return goal;
else if goalProb < p < goalProb + wayPointProb then
  return WayPoints[i]
else if goalProb + wayPointProb < p < 1.0 then
  return RandomState()
```

W publikacji [2] prawdopodobieństwo podążania do właściwego celu ostatecznie ustalane zostało na poziomie 0.1, natomiast `wayPointProb` wynosiło 0.7.



Rysunek 7.10: `RRT` z zapamiętanymi węzłami z porzedniego uruchomiena. Linią kropkowaną zaznaczono gałąź dodaną w kierunku losowego punktu, linią przerywaną w kierunku jednego z elementów z poprzedniej ścieżki, a linią niebiską w kierunku celu.

The distance metric can be modified to include not only the distance from the tree to a target state, but also the distance from the root of the tree, multiplied by some gain value.

7.4 Zalety algorytmu RRT w stosunku do CVM

Podczas kontynuacji prac nad rozgrywkami *RoboCup* zdecydowano się na zmianę algorytmu wyznaczającego bezkolizyjną ścieżkę. Zmiana została wymuszona decyzją o zmianie modelu sterowanego robota. Bazę jezdną o napędzie różnicowym zastąpiono holonomiczną. Algorytm *CVM* jest algorytmem dedykowanym do robotów o napędzie różnicowym, doskonale można ująć w nim ograniczenia na dopuszczalne prędkości. Jednak sterowanie za jego pomocą robotem holonomicznym nie pozwala na pełne wykorzystanie możliwości robota. Dodatkowo jak zostało udowodnione w pracy inżynierskiej [14], *CVM* w przypadku przeszkód dynamicznych osiągał skuteczność na poziomie 80%. Kolizje miały miejsce głównie w sytacjach, kiedy przeszkoda uderzała w robota z boku lub z tyłu. Wynikało to z faktu, że trajektoria ruchu takiej przeszkody znajdowała się poniżej osi OY w układzie współrzędnych związanym ze sterowanym robotem. Przeszkody takiego typu nie były uwzględnianie przez algorytm. *RRT* pozwala na uwzględnianie przeszkód rozsianych po całej planszy. Dodatkowo umożliwia poruszanie się po bardziej złożonych trajektoriach. Kolejną zaletą *RRT* jest prostota w ograniczaniu przestrzeni z której losowane są tymczasowe punkty docelowe. Dzięki temu w prosty sposób można ograniczyć poruszanie się robota tylko do wnętrza boiska, w przypadku *CVM* istniała konieczność modelowania bandy boiska jako zbioru okręgów, mnożyło to ilość obiektów z jakimi należało detekować kolizje. Kolejnym problemem *CVM* jest nawigacja do celu znajdującego się za robotem, w pracy inżynierskiej wprowadzono modyfikację, zmieniającą parametr odpowiedzialny za kierowanie robota na cel, uzależniając go od orientacji robota względem celu. Intencją było wymuszenie obrotu robota w kierunku celu, w sytuacji gdy orientacja do celu jest duża. Modyfikacja poprawiła skuteczność jednak, nawet dla najlepszego zestawu wag wynosiła ona 80%. W przypadku bazy holonomicznej obracanie się w kierunku punktu docelowego nie jest konieczne, stąd takie sterowanie nie jest optymalne.

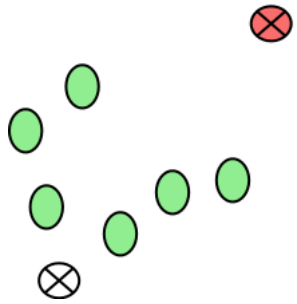
7.5 Opis implementacji algorytmu `RRT`

W trakcie testów wersji podstawowej algorytmu *RRT* opisanej w 7.3 zdecydowano się na dodatkowe modyfikacje algorytmu, część zaczerpnięto z 7.3 dodano też własne modyfikacje, które miały na celu poprawienie wydajności algorytmu i skrócenie czasu obliczeń. Zdecydowano się na implementację algorytmu w wersji z zapamiętywaniem części węzłów ze ścieżki wyznaczonej w poprzednim uruchomieniu algorytmu (pod warunkiem, że punkt docelowy nie uległ zmianie) i traktowanie ich jako tak zwanych `way points`. Jednak z uwagi na specyfikę eksplorowanego środowiska (brak wąskich korytarzy, przeszkody są wypukłe) zdecydowano się na eliminowanie z zapamiętanej ścieżki węzłów których odległość między sąsiadami jest mniejsza niż $5cm$. Zrezygnowano z przechowywania węzłów w postaci `KD-drzewa`.

Dodatkowo wprowadzono ograniczenie na maksymalną liczbę węzłów algorytmu. Sytuacja na planszy podczas rozgrywki szybko ulega zmianie, stąd wyznaczanie kompletnej ścieżki do celu pozbawione jest sensu, jeśli ma to być czasochłonne. Przy ponownym uruchomieniu algorytmu sytuacja może wyglądać inaczej i algorytm szybciej wyznaczy ścieżkę. Jeśli uruchomienie algorytmu nie zakończyło się wyznaczeniem kompletnej ścieżki do celu jako punkt docelowy dla robota wybierany jest węzeł znajdujący się najbliżej celu.

Zauważono także, że przy małym prawdopodobieństwie budowy ścieżki do celu, drzewo może ulegać nadmiernemu rozbudowaniu przy samym korzeniu. Ma to miejsce w sytuacji, gdy tymczasowe punkty losowane są z otoczenia korzenia, kolejne wylosowane węzły mogą powodować poszerzenie gałęzi, a tak rozbudowane gałęzie mogą znajdować się w podobnej odległości od celu. W efekcie algorytm przez dłuższy czas może próbować wyznaczać kilka ścieżek prowadzących do celu, co nie jest pożądane. Sytuację taką dla dwóch gałęzi przedstawiono na rysunku 7.11. W zaimplementowanej wersji ograniczono liczbę wezłów potomnych korzenia do 4.

Podobnie jak w pracy inżynierskiej [14] zdecydowano się na naiwną predykcję położenia dynamicznych przeszkód. Ponieważ baza jezdna robota zbliżona jest do okręgu w algorytmie jest ona analizowana jako okrąg o środku znajdującym się w danym położeniu robota (przeszkody) i promieniu $r = 2 * (r_{base} + l_{dribbler})$, gdzie r_{base} jest promieniem bazy jezdnej robota, a $l_{dribbler}$ długością dribblera. Do zbioru rzeczywistych przeszkód dodawne są przeszkody, których środek przesunięty jest o drogę jaką może pokonać dana przeszkoda w czasie trwania jednego uruchomienia algo-



Rysunek 7.11: Drzewo wyznaczające ścieżkę do celu.

rytmu. Podczas planowania ścieżki promienie przeszkód powiększane są dodatkowo o margines bezpieczeństwa, ma on za zadanie wyeliminowanie ewentualnych kolizji wynikających z poślizgu robota podczas hamowania, czy opóźnienia przy zmianie kierunku jazdy. Główną zaletą `RRT` jest jego szybkość, stąd przewidywanie położenia przeszkody na 1 krok w przód okazało się wystarczające. W przypadku `CVM` konieczne było dodawanie "wirtualnych" przeszkód symulujących położenie robotów za 15, 30 oraz 45 kroków symulatora.

W zaimplementowanej formie algorytm ma następujący przebieg:

Algorytm 7.5 Wersja `RRT` z wprowadzonymi modyfikacjami

ogranicz maksymalną przestrzeń do rozmiarów boiska
if punkt docelowy jest poza dopuszczalnymi współrzędnymi **then**
  zwróć błąd *waypoints*
if w poprzednim uruchomieniu znaleziono ścieżkę do celu **then**
  zainicjuj listę *waypoints*
dodaj statyczne oraz dynamiczne przeszkody;
if robot uległ kolizji (nie są brane pod uwagę przewidywane położenia przeszkód) **then**
  tak zwróć błąd;
if robot dojechał do celu **then**
  tak zwróć sukces;
if punkt docelowy jest w obrębie przeszkody **then**
  zwróć błąd;
if punkt docelowy jest bezpośrednio osiągalny **then**
  zwróć go jako wynik działania algorytmu;
while `goalState.distance(nextState)` > `minimalDistance` && `treeSize¡ maxNodeNumber` **do**
  wybierz tymczasowy cel: `tmp_target` = `chooseTarget(goalState)`
  nearestState = `nearest(tree, target)`
  `extendedState` = `extend(env, nearest, target)`
  if `extendedState` ! = `NULL` **then**
    odblokuj ustawianie `tmp_target` jako textttgoalState
    `addNode(tree, extended)`
  else if `tmp_target` należy do zbioru *waypoints* **then**
    usuń punkt ze zbioru *waypoints*
  else if `tmp_target` jest właściwym punktem docelowym **then**
    zablokuj ustawianie `tmp_target` jako textttgoalState
if `nearestState.distance(goalState)` > `minimalDistance` **then**
  return węzeł znajdujący się najbliżej celu
else
  zapamiętaj ścieżkę do celu
  return węzeł bezpośrednio osiągalny znajdujący się najbliżej celu;

W opisie algorytmu zastosowano fukcje `chooseTarget` oraz `extend`, których działanie również zostało lekko zmodyfikowane w stosunku do oryginalnej wersji algorytmu. Ich działanie zostało przedstawione poniżej:

Algorytm 7.6 funckcja `chooseTarget` wyznaczająca tymczasowy punkt docelowy

`function` *chooseTarget*(*goal:state*) *state*;
p = `uniformRandom in`[0.0...1.0]
if wyznaczono ścieżkę w poprzednim uruchomieniu algorytmu **then**
  if $0.0 < p < goalProb$ **then**
    if wybór właściwego punktu docelowego jest zablokowany **then**
      return `randomPoint`;
    return właściwy punkt docelowy
  else if $p < goalProb + wayPointProb$ **then**
    i = `uniformRandom in`[0...$number\_of\_waypoints - 1$]
    return *WayPoints[i]*
  else
    return *RandomState()*
else
  if $0.0 < p < goalProb$ && wybór właściwego punktu docelowego nie jest zablokowany **then**
    return `goal`;
  else
    return *RandomState()*

Algorytm 7.7 funckcja `extend(nearest,tmp_target)` rozszerzejąca drzewo w kierunku zadanego punktu

podczas detekcji kolizji przeszkody powiekszane są o margines bezpieczeństwa oraz uwzględniane są dodatkowe przeszkody wynikające z ruchu robotów
if punkt docelowy leży w obrębie przeszkody **then**
  return `NULL`
wyznacz nowy węzeł (`newNode`) odległy od `nearest` w kierunku `tmp_target`
if wyznaczony punkt leży w obrębie przeszkody **then**
  return `NULL`
return `newNode`

7.5.1 Parametry zastosowanego algorytmu

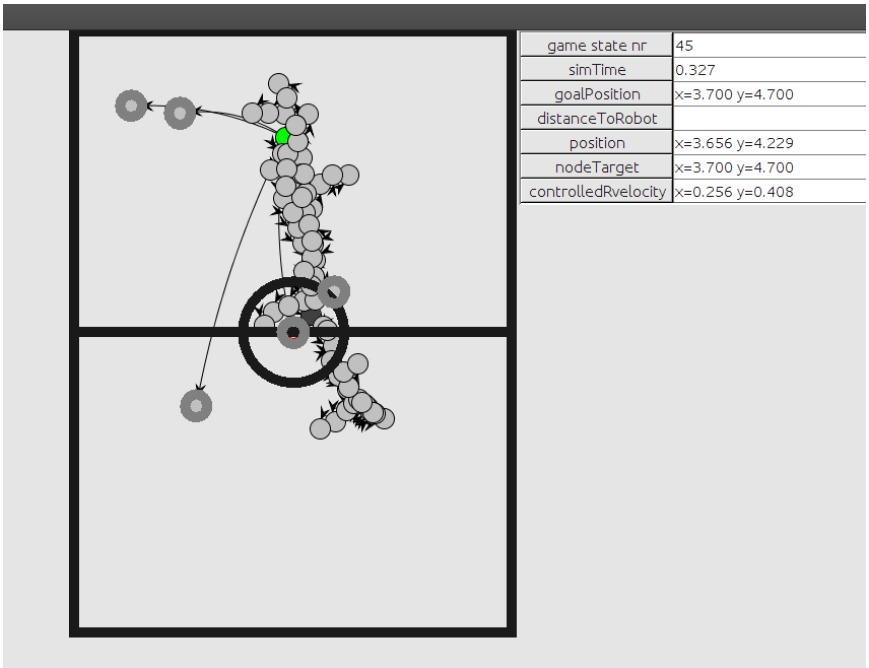
Zaimplementowowaną wersję algorytmu `RRT` można w łatwy sposób konfigurować poprzez zmiane kluczowych parametrów takich jak:

1. włączanie/wyłączanie predykcji ruchu przeszkod,
2. prawdopodobieństwo losowania tymczasowego punktu docelowego należącego do `wayPoints` (*goalProb*) lub jak punkt docelowy *wayPointProb*,
3. rozmiary dopuszczalnej przestrzeni z jakiej losowany jest tymczasowy cel, wynikajace z rozmiaru swiata,
4. ograniczenia na przestrzen przeszukiwan wynikajace z zaistnialej sytuacji na planszy, bądź z rodzaju zadania,
5. odległości po przekroczeniu której uznajemy, że robot dojechał do celu,
6. ścieżka z poprzedniego wywołania algorytmu, przekazywana tylko wtedy gdy punkt docelowy nie zmienił się,
7. maksymalna liczba węzłów drzewa `RRT`,
8. maksymalna liczba węzłów doczepionych do korzenia,
9. odległość między `wayPoints`; z przekazanej ścieżki scalane są te między którymi odległość jest mniejsza niż wartość parametru,
10. *rrtStep* odległość o jaką rozszerzany jest węzeł leżący najbliżej tymczasowego celu w jego kierunku,
11. odleglość punktu startowego od najbliższej przeszkody, brane są pod uwagę jedynie bieżące położenia robotów.

Rozdział 8

Testy zaimplementowanej wersji algorytmu RRT

W trakcie implementacji algorytmu RRT wyniknęła konieczność napisania programu umożliwiającego sprawdzenie poprawności działania, w tym celu powstała aplikacja RRT_debug. Umożliwia ona wczytanie serii drzew zapisanych w formacie xml. Program został napisany w języku Java, a do reprezentacji węzłów drzewa wykorzystano bibliotekę JUNG-Java Universal Network/Graph Framework. Aplikacja umożliwia śledzenie konstruowanych przez RRT drzew. Ilustracja 8.1 zawiera przykładowy zrzut ekranu z uruchomionej aplikacji. Program umożliwia wyświetlanie dodatkowych informacji o zaznaczonym węźle drzewa, oraz wykonywanie zbliżenia w celem podglądu szczegółowo wybranej części drzewa. W prawym górnym rogu wyświetlane są infomrmacje o czasie symulacji w jakim zostało stworzone drzewo, pozycji docelowej drzewa, tymczasowej pozycji w kierunku której rozszerzano drzewo przy dodawaniu zaznaczonego węzła, współrzędnych zaznaczonego węzła na planszy oraz prędkości sterowanego robota.



Rysunek 8.1: Aplikacja RRT_debug.

8.1 Opis środowisk testowych

Algorytm RRT poddano wstępnym testom w aplikacji RRT_debug. Po sprawdzeniu poprawności działania zdecydowano się na porównanie algorytmu pod względem skuteczności z metodą CVM opracowaną w pracy inżynierskiej [14]. W tym celu zastosowano ten sam zestaw środowisk testowych (załącznik C strona 83). W związku ze zmianą wymiarów boiska, sytuacje zaczerpnięte z pracy inżynierskiej zostały przeskalowane, odpowiednio wydłużony został także maksymalny czas. Dla środowisk dynamicznych wyniósł on 12.47$[s]$, a dla statycznych 8.73 . Po przeskalowaniu droga do piłki nie przekraczała 3$[m]$, maksymalną prędkość ograniczono do 1$[\frac{m}{s}]$. Jadąc prosto do celu z maksymalną prędkością robot powinien przejechać dystans w maksymalnie 3$[s]$ Testy przeprowadzono na 10 środowiskach statycznych oraz na 10 środowiskach dynamicznych. W przypadku eksperymentów w środowisku statycznym dokonano pewnego uproszczenia. Zrezygnowano z rozróżnienia eksperymentów na sytuacje, gdy robot zwrócony jest w stronę celu lub przeciwnie. Metoda opracowana w pracy inżynierskiej była zdecydowanie mniej skuteczna w sytuacjach, gdy cel znajdował się za robotem. Wynikało to ze specyfiki samego algorytmu, jak i rodzaju

zastosowanej bazy jezdnej (ograniczenia wynikające z więzów nieholonomicznych). Algorytm RRT jest niewrażliwy na położenie punktu docelowego względem sterowanego robota, zastosowana baza jezdna także pozwala na poruszanie się w dowolnym kierunku. Zadaniem sterowanego robota było dojechanie do piłki, tak aby uniknąć kolizji z innymi robotami, bandą boiska oraz bramkami. Zdecydowano się na zbadanie działania algorytmu dla różnych prawdopodobieństw wyboru tymczasowego punktu docelowego. Na każdej planszy algorytm uruchomiono 20-krotnie dla różnego zestawu prawdopodobieństw $goalProb$, $wayPointProb$. Przetestowano wszystkie możliwe warianty prawdopodobieństw z krokiem 0.1 których jest dokładnie 65. Ponumerowane zestawy współczynników zamieszczone są w załączniku C na stronie 83. Pomiarom zostały poddanne następujące wielkości:

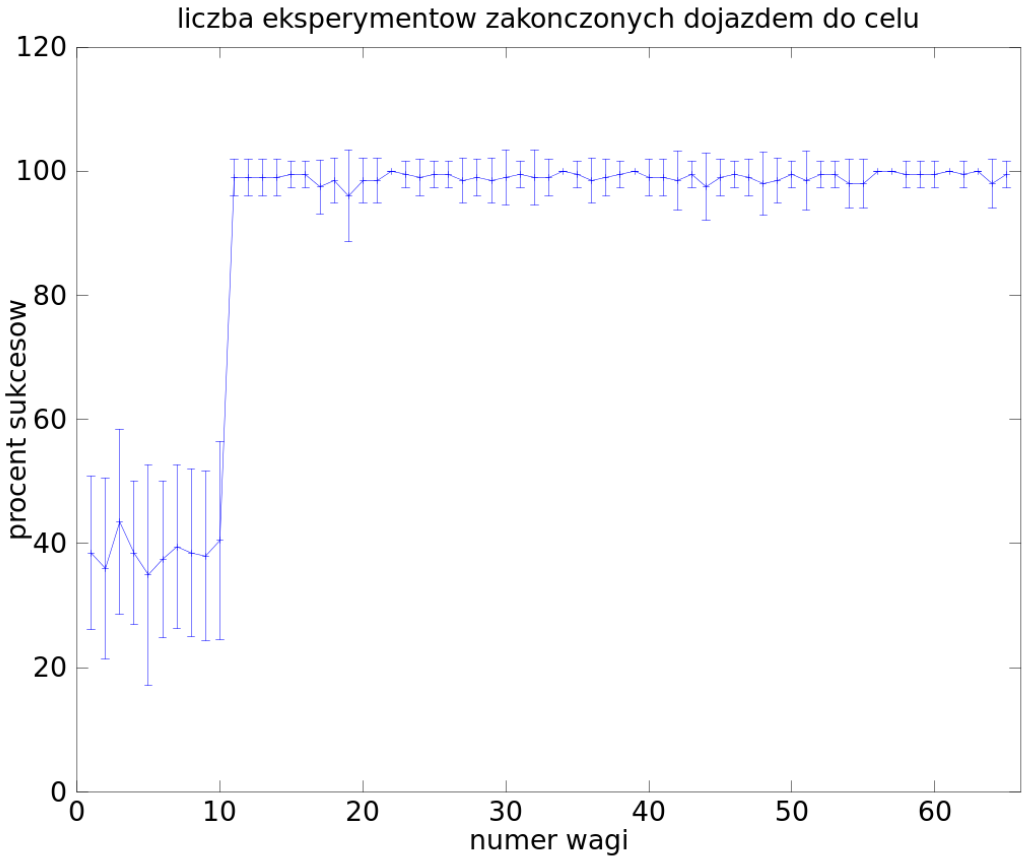
1. procent eksperymentów zakończonych sukcesem dla każdego zestawu wag,
2. czas dojazdu robota do celu, przy czym uwzględnine zostały jedynie sytuacje w których robot zakończył eksperyment sukcesem,
3. średni czas obliczeń jednego uruchomienia algorytmu, z pominięciem sytuacji w których cel był bezpośrednio osiągalny,
4. maksymalna długość wyznaczonej ścieżki w czasie trwania jednego eksperymentu, dla testowanego zestawy prawdopodobieństw,
5. maksymalna liczba węzłów drzewa wyznaczana w analogiczny sposób jak powyżej.

Dodatkowo zamieszczony został wykres przedstawiający ile razy dla danego zestawu wag i konkretnej planszy choć raz znaleziono ścieżkę do celu.

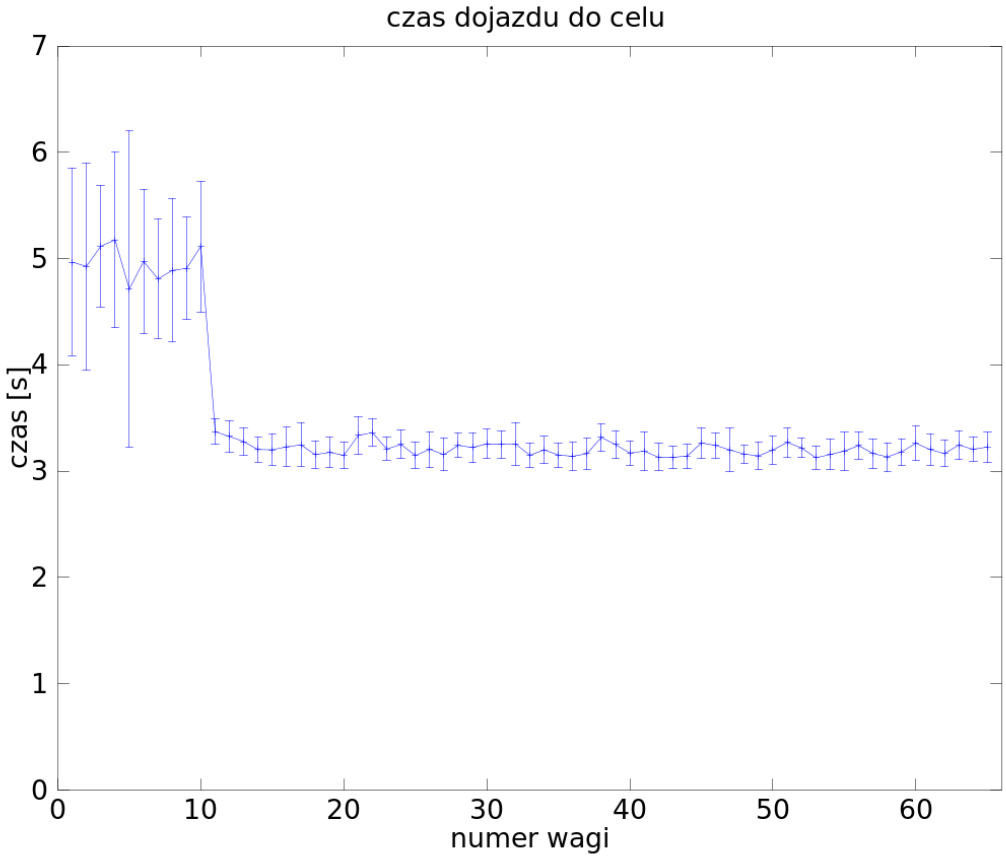
W trakcie przeprowadzanych eksperymentów krok, z jakim pracował symulator został ustawiony na $0.001[s]$, przy takiej wartości `real-time factor` kształtował się w okolicach $0.9 - 1.0$ na maszynie wyposażonej w procesor `Intel i5-2520M` z zegarem $2.5[GHz]$. W przypadku środowisk statycznych daje to maksymalny czas pełnego eksperymentu na poziomie 31 godzin, w przypadku środowisk dynamicznych górne ograniczenie wyniosło 45 godzin.

8.2 Wyniki eksperymentów w środowisku statycznym

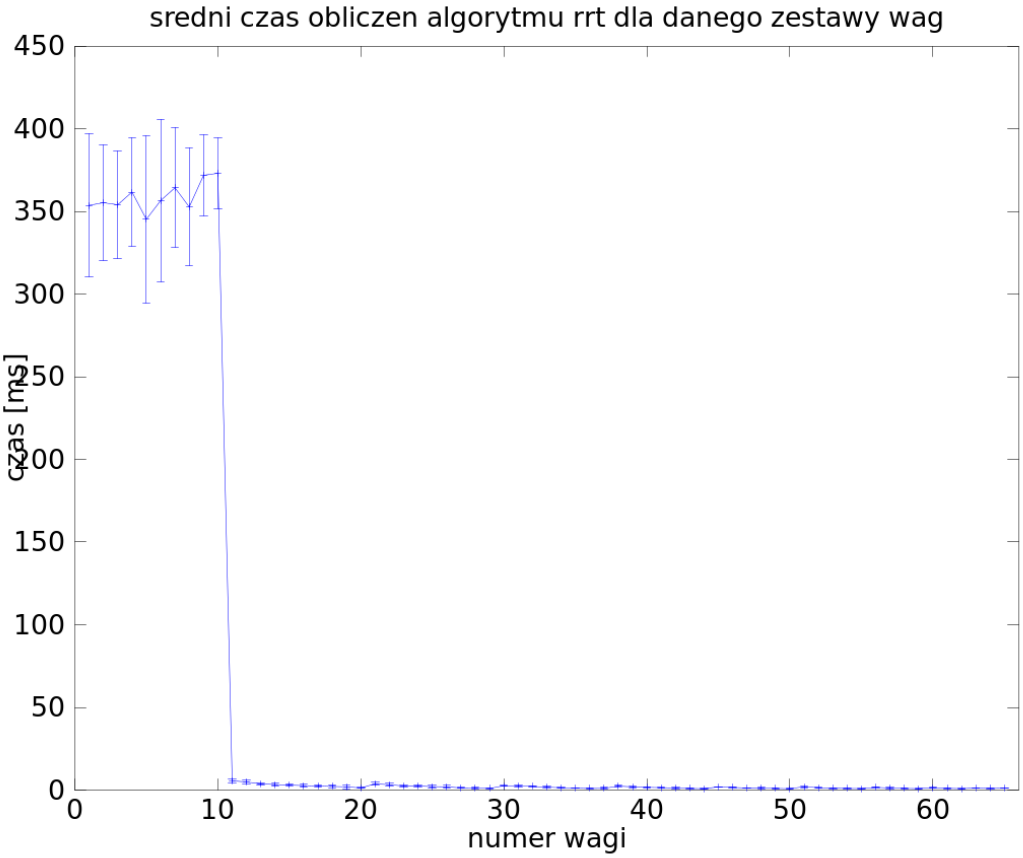
Poniższe wykresy prezentują wyniki otrzymane dla środowisk statycznych. Można zauważyć, że dla współczynników poniżej numeru 11, czyli gdy prawdopodobieństwo budowania drzewa do celu jest równe 0, algorytm przy odpowiednio długim uruchomieniu także potrafi wyznaczyć ścieżkę do celu, jednak czas obliczeń samego algorytmu jest długi, rzędu 350$[ms]$, a czas dojazdu jest na poziomie 5$[s]$. Przy takich parametrach, znalezienie ścieżki do celu wymaga zbudowania drzewa zawiarającego w granicach 800 węzłów.



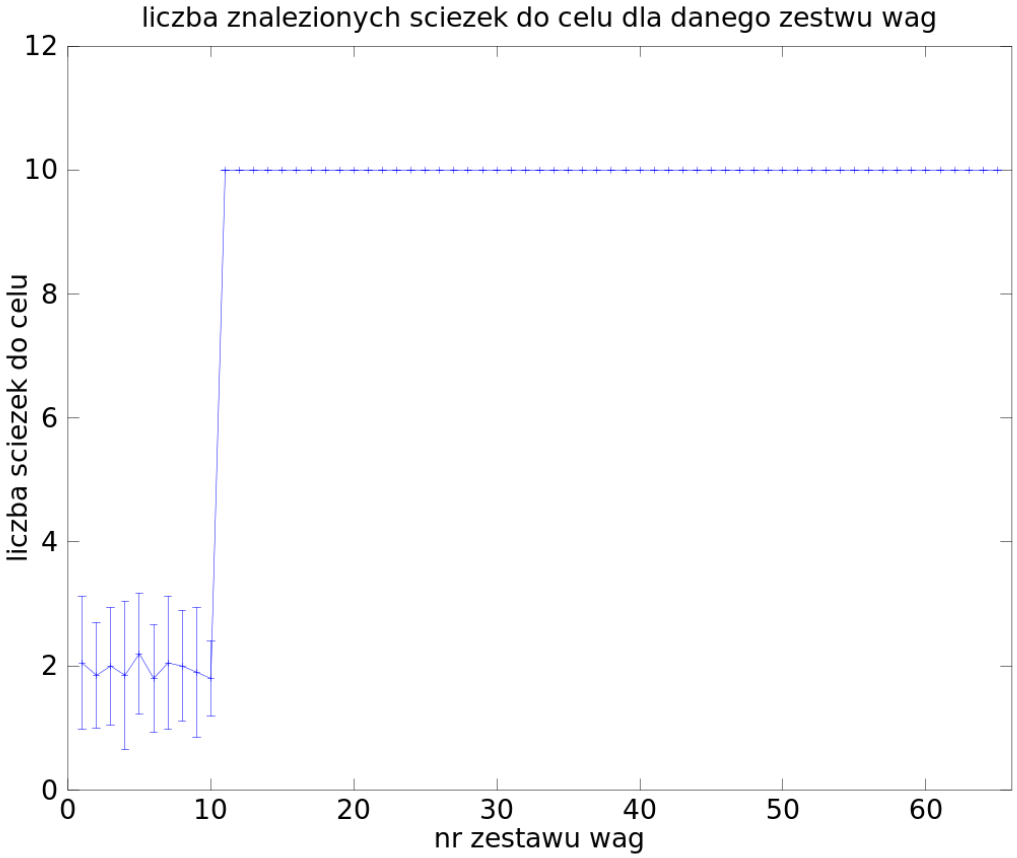
Rysunek 8.2: Procent eksperymentów zakończonych sukcesem.



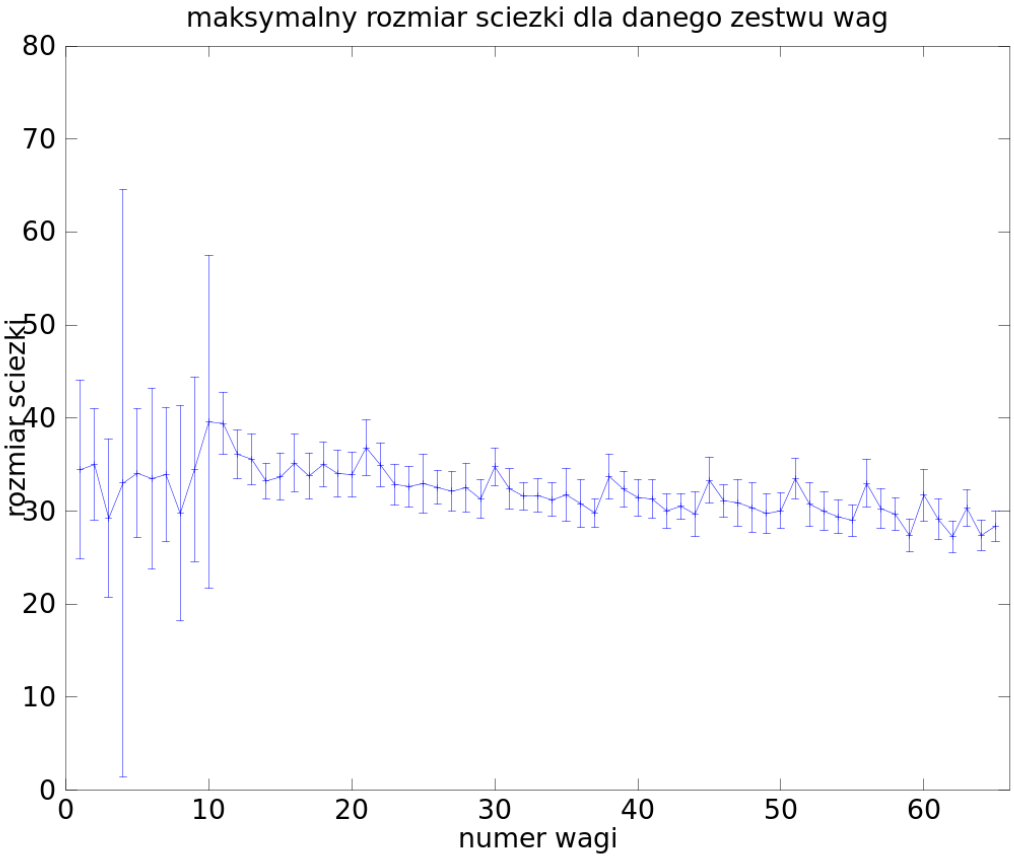
Rysunek 8.3: Czas dojazdu do celu.



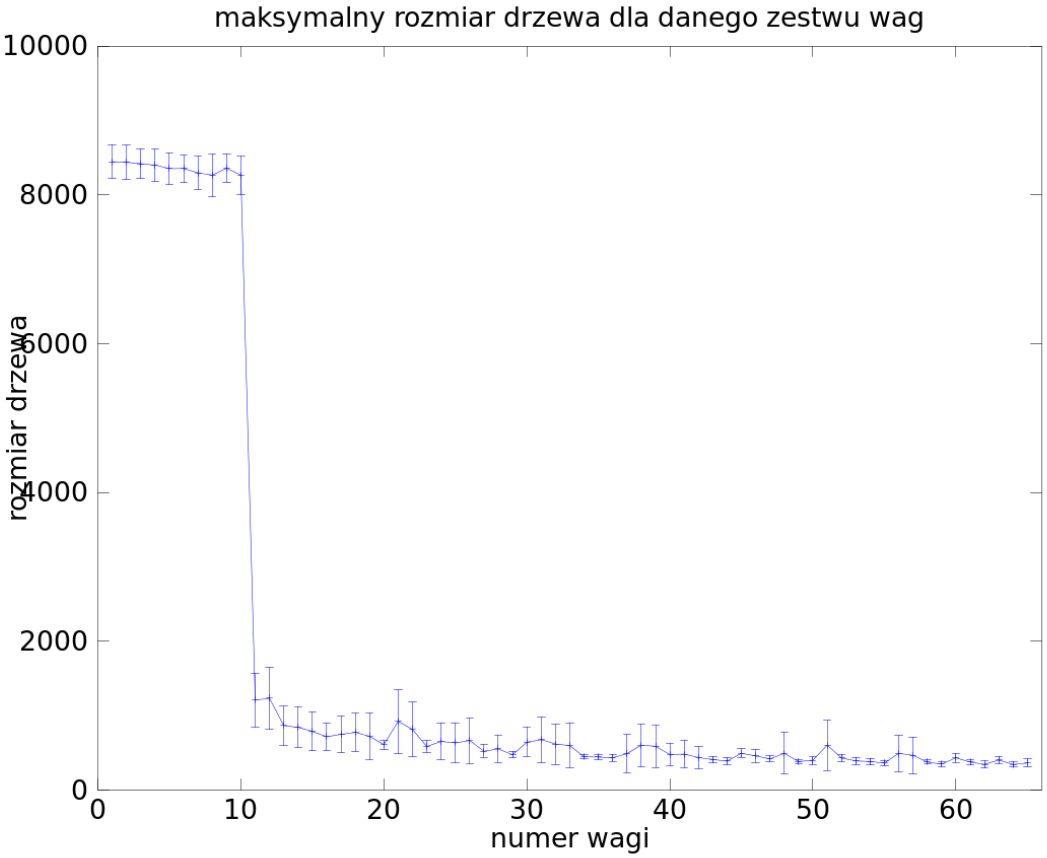
Rysunek 8.4: Czas jednego uruchomienia RRT .



Rysunek 8.5: Liczba znaleźionych ścieżek prowadzących do celu.



Rysunek 8.6: Maksymalny rozmiar ścieżki prowadzącej do celu.

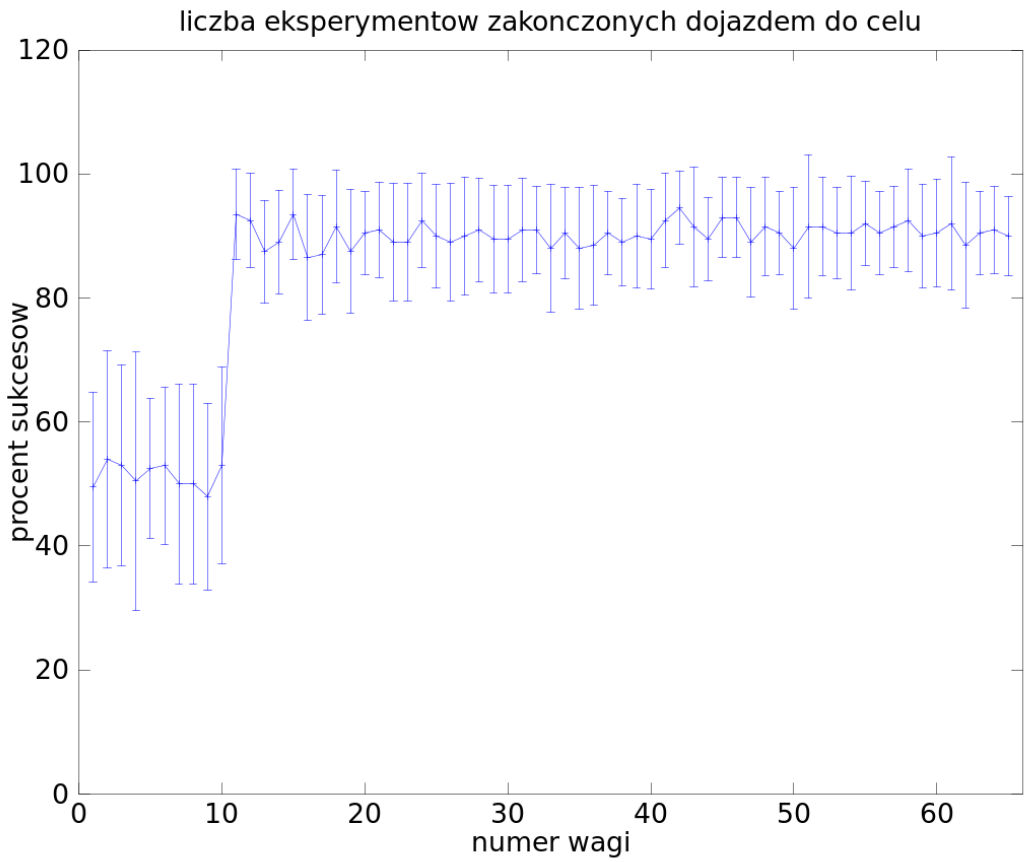


Rysunek 8.7: Maksymalny rozmiar drzewa znalezionego przez RRT.

zamiescic przykładowe trajektorie w środowisku statycznym

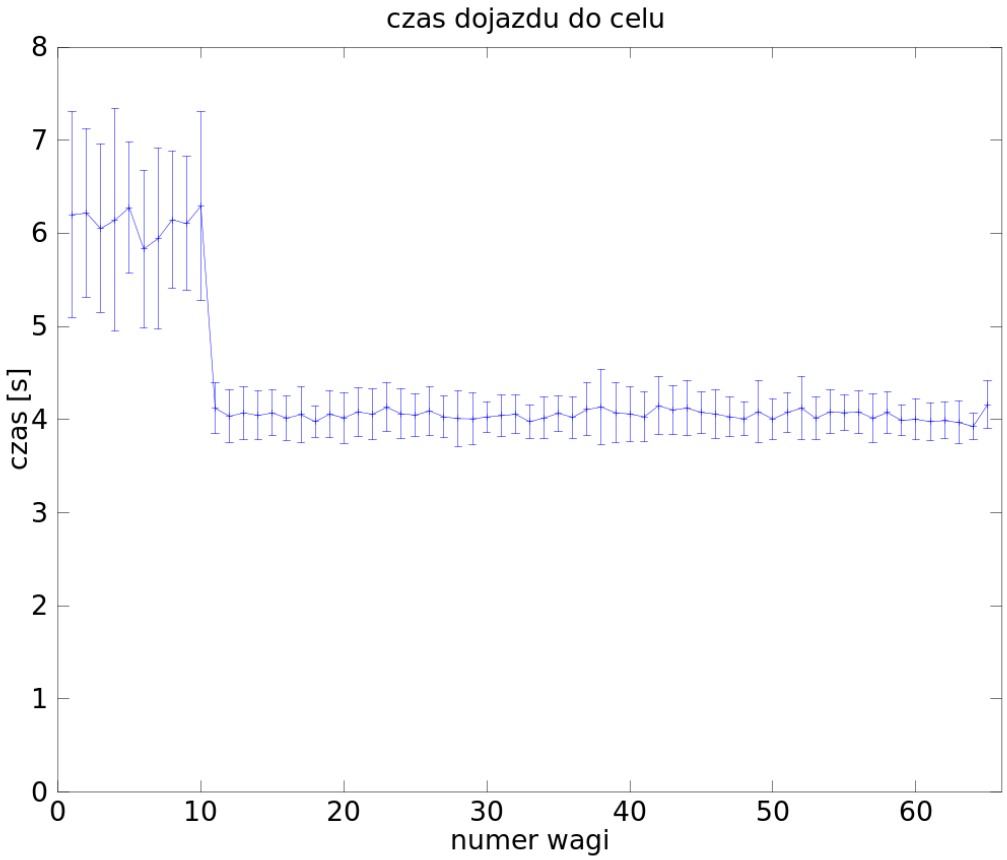
8.3 Wyniki eksperymentów środowisku dynamicznym

Rezultaty otrzymane podczas testów z ruchomymi przeszkodami zostały zaprezentowane poniżej. Można zauważyć, że procent sukcesów dla wag poniżej numeru 11 jest wyższy niż w przypadku eksperymentów statycznych. Jest to spowodowane tym, że po pewnym czasie piłka staje się bezpośrednio osiągalna i algorytm unikania kolizji nie jest konieczny. Podobnie jak w przypadku eksperymentów statycznych czas obliczeń w tym przypadku jest wysoki rzędu 320$[ms]$, a czas dojazdu przekracza 6$[s]$.

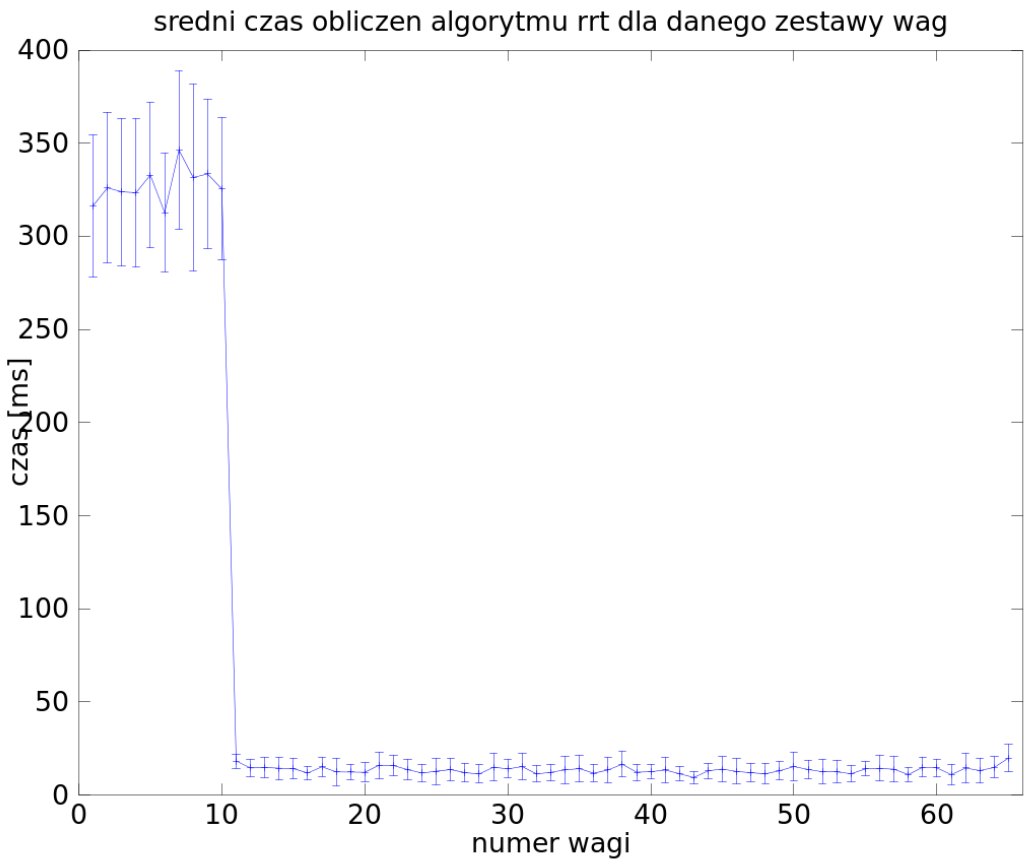


Rysunek 8.8: Procet eksperymentów zakończonych sukcesem.

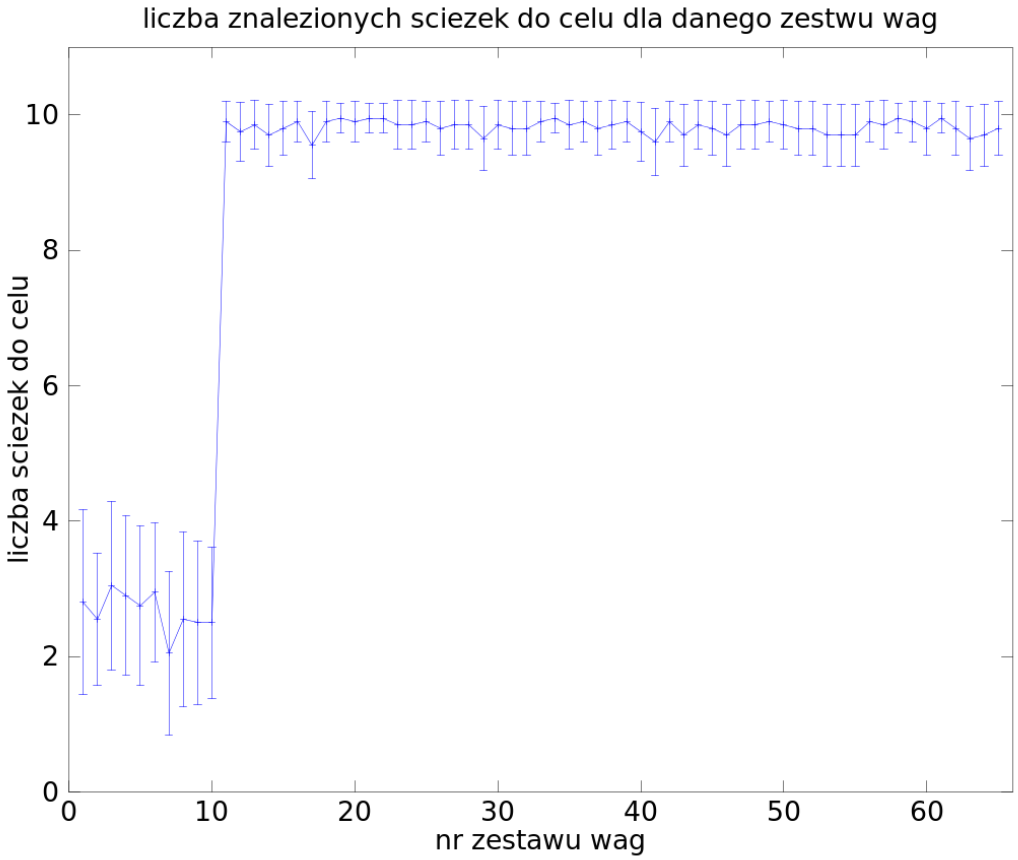
zamiescic przykładowe trajektorie w srodowisku dynamicznym



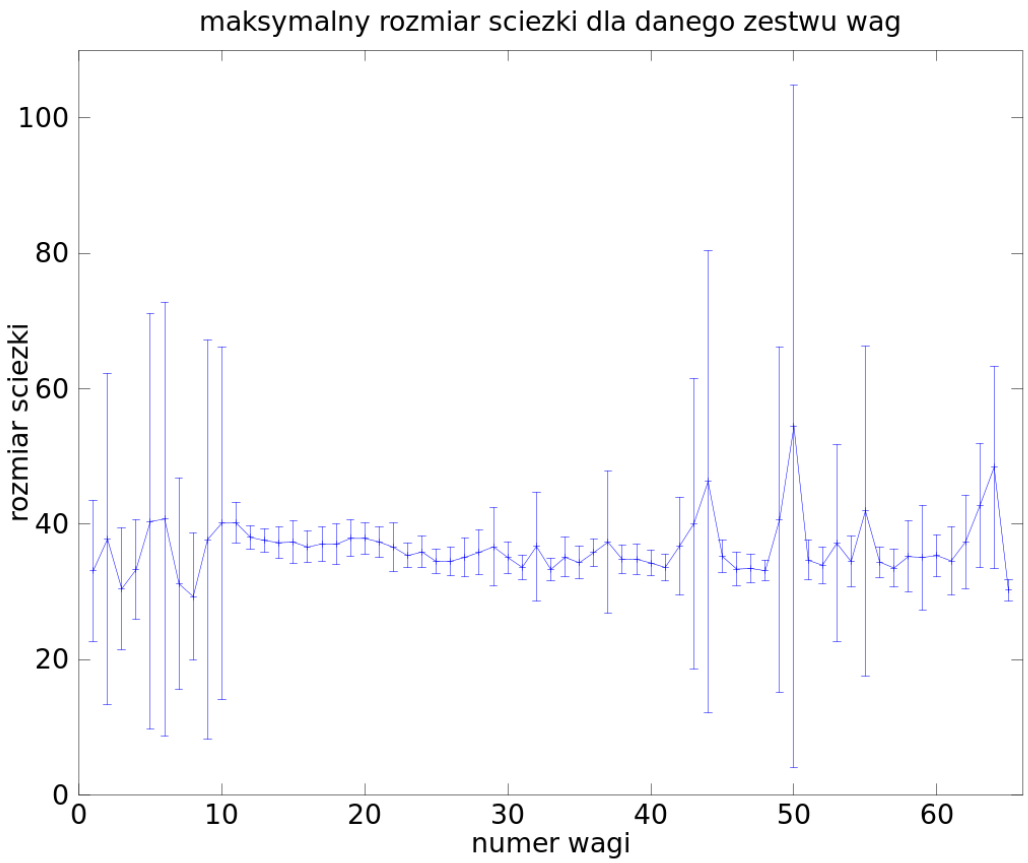
Rysunek 8.9: Czas dojazdu do celu.



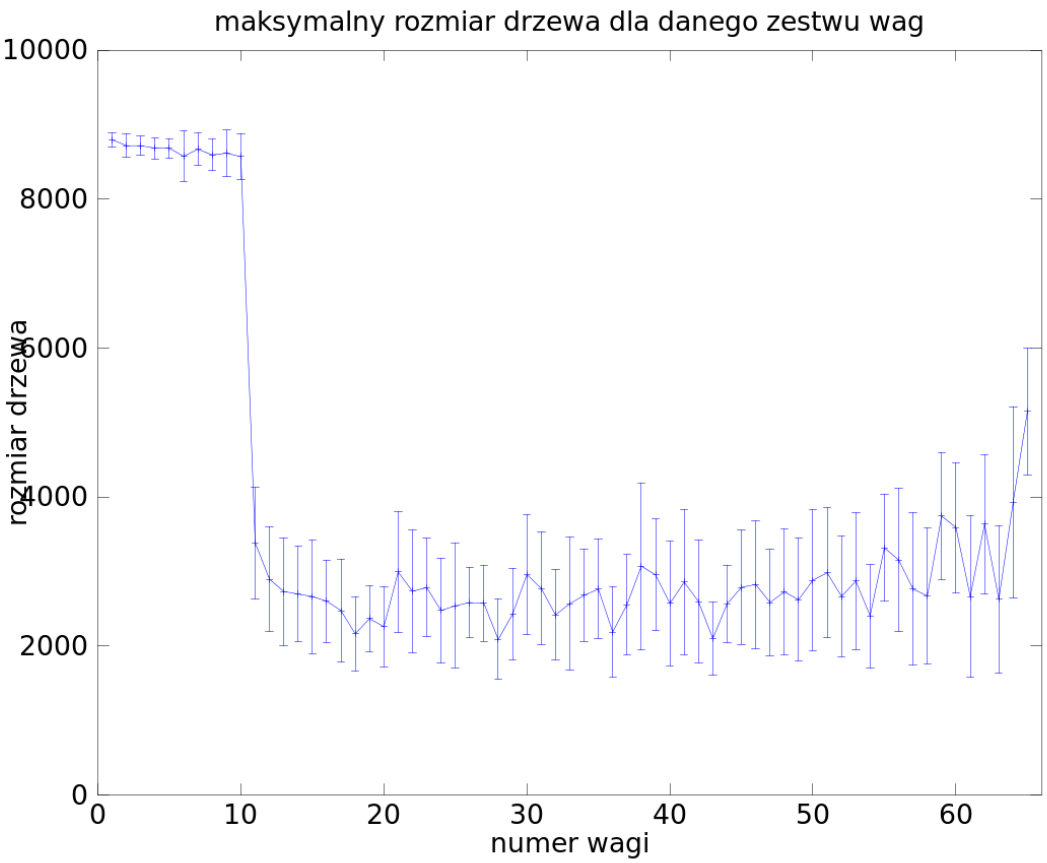
Rysunek 8.10: Czas jednego uruchomienia RRT .



Rysunek 8.11: Liczba znaleźionych ścieżek prowadzących do celu.



Rysunek 8.12: Maksymalny rozmiar ścieżki prowadzącej do celu.



Rysunek 8.13: Maksymalny rozmiar drzewa znalezionego przez RRT.

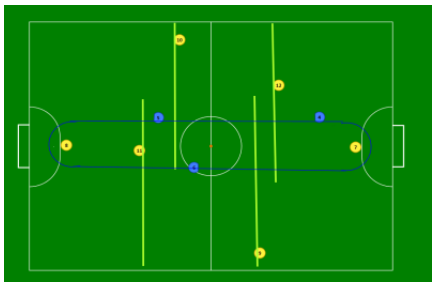
Rozdział 9

Testy architektury STP

Kolejnym etapem jakim poddano aplikację było przetestowanie zaimplementowanej warstwy wzorowanej na architekturze STP. Każda zgłoszona do rozgrywek drużyna, przed przystąpieniem do turnieju głównego musi przejść eliminacje sprawdzające jej poziom. Jako zadania testowe postanowiono wybrać niektóre rzeczywiste zadania elimincyjne, przez jakie musiały przejść drużyny w ostatnich latach. Więcej informacji na temat eliminacji można znaleźć na stronie projektu [1].

9.1 Nawigacja w dynamicznym środowisku

Pierwszym zadanie pochodzi z eliminacji do mistrzostw w 2011 roku. Celem próby jest sprawdzenie zdolności robotów do bezpiecznego poruszania się w dynamicznym środowisku. Poniżej zamieszczono rysunek przedstawiający środowisko testowe. Znajduje się na nim 6 robotów pełniących role przeszkód. Dwa z pośród nich są nieruchome, a pozostałe cztery poruszają się wzdłuż zaznaczonej linii prostej. Zasady



Rysunek 9.1: Plan środowiska testowego

eksperymentu są następujące:

1. Liczba startujących robotów jest ograniczona do trzech.
2. Uczestniczące roboty muszą poruszać się pomiędzy dwoma nieruchomymi przeszkodami.
3. Każdorazowo kiedy robot dotknie przeszkody otrzymuje punkt ujemny.
4. Każdy uczestnik, który pokona z powodzeniem trasę otrzymuje punkt.
5. Robot, który wykona okrążenie z piłką otrzymuje dodatkowo 2 punkty.
6. Test trwa 2 minuty.

9.1.1 Wyniki testu nawigacji

Rezultaty uczestników eliminacji

9.2 Strzelanie po podaniu

Kolejne zadanie pochodzi z 2009 roku. W zadaniu uczestniczy od 2 do 3 robotów z jednej drużyny. Ich zadaniem jest zdobycie jak największej ilości goli w przeciągu 120 sekund. Punkty przyznawane są następująco:

1. drużyna zdobywa 1 punkt w momencie gdy przed poprawnym strzałem na bramkę dwa roboty dotkną piłki (wykonane zostanie np. jedno podanie).
2. drużyna zdobywa 2 punkty gdy przed oddanym strzałem 3 roboty dotkną piłki.

Zasady eksperymentu zamieszczono poniżej:

1. Przed startem wszytkie roboty muszą być umieszczone w odległości nie przekraczającej $1[m]$ od własnej linii bramkowej,
2. Zawodnicy drużyny przeciwnej pełnią rolę statycznych przeszkód,
3. Przy każdym starcie piłka jest umieszczana w jedynm z narożników, na własnej połowie grającej drużyny,
4. Gol może być zdobyty tylko w sytuacji, gdy robot znajdduje się na połowie przeciwnika,

5. Po zdobytej bramce piłka ponownie wraca do jednego z narożników.

9.2.1 Wyniki testu

Rezultaty uczestników eliminacji

Rozdział 10

Podsumowanie

Dodatek A

Zawartość płyty CD

Dołączona do pracy płyta CD zawiera następujące elementy:

- pliki źródłowe symulatora *Gazebo* wraz z naniesionymi poprawkami opisanymi w
- modele elementów środowiska *RoboCup* do symulatora *Gazebo* – folder `modele`,
- kod źródłowy aplikacji sterującej robotem (projekt w środowisku Eclipse[1] dla C++) wraz z dokumentacją – folder `aplikacja_sterujaca`,
- filmy oraz zrzuty ekranu prezentujące wykonane modele oraz działanie aplikacji sterującej – folder `media`,
- plik `praca_magisterska.pdf` – wersja elektroniczna niniejszego dokumentu.

[1] Środowisko jest dostępne pod adresem `www.eclipse.org`.

Dodatek B

Instrukcja instalacji Gazebo

poprawic opis instalacji gazebo

Instrukcję instalacji można znaleźć w poradniku dostępnym pod adresem `http://playerstage.sourceforge.net/doc/Gazebo-manual-svn-html/install.html`. W razie problemów można skorzystać także z opisu dostępnego pod adresem `http://www.irobotics.org/gazebo08.f8.html`. Do instalacji należy użyć wersji *Gazebo* dostępnej na płycie CD dołączonej do pracy (jest to wersja 6782 dostępna poprzez repozytorium SVN, zawiera jednak poprawki opisane w par. 4.3.2). Do poprawnej pracy *Gazebo* wymagana jest obecność dodatkowych aplikacji. Najważniejsze z nich wykorzystano w niniejszej pracy w nastepujących wersjach:

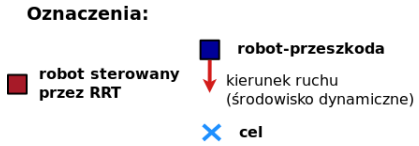
zmienic numery rewizji zastosowanych aplikacji

- Player (wymagany do kompilacji Gazebo) – pobrany z repozytorium SVN w wersji 6350,
- ODE pobrane z oficjalnego repozytorium SVN, wersja 1451,
- OGRE w wersji 1.4.7,
- pozostałe wymagane aplikacje zostały zainstalowane w wersjach zgodnych z opisem w podręczniku instalacji.

Dodatek C

Szczegóły eksperymentów

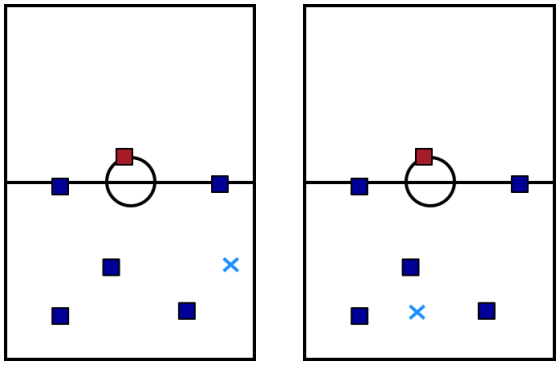
Dodatek zawiera poglądowe rysunki przedstawiające środowiska testowe, na których przeprowadzane były eksperymenty. Na każdym z nich zaznaczono inne roboty nie podlegające sterowaniu przez testowany algorytm oraz początkowe położenie i orientację sterowanego robota. W przypadku eksperymentów w środowisku dynamicznym zaznaczono także kierunek i zwrot prędkości z jaka poruszały się przeszkody. Poniżej zamieszczono legendę objaśniającą znaczenie użytych symboli.



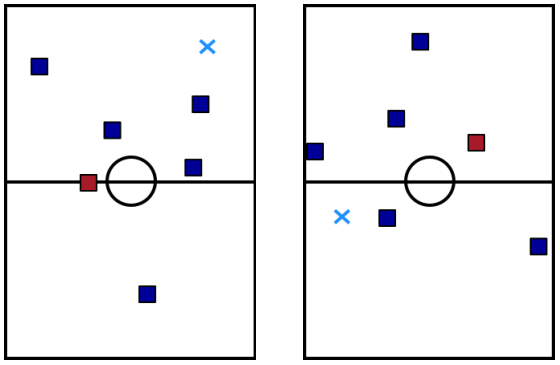
W drugiej części dodatku zamieszczono tabelę z zastosowanymi podczas eksperymentów z użyciem algorytmu *CVM* zestawami wag. Ponieważ każda ze składowych funkcji celu (7.8) algorytmu zwraca wartość z przedziału $[0; 1]$ zdecydowano się na przetestowanie takich trójek liczb z tego przedziału, które sumują się do jedności.

Środowiska testowe

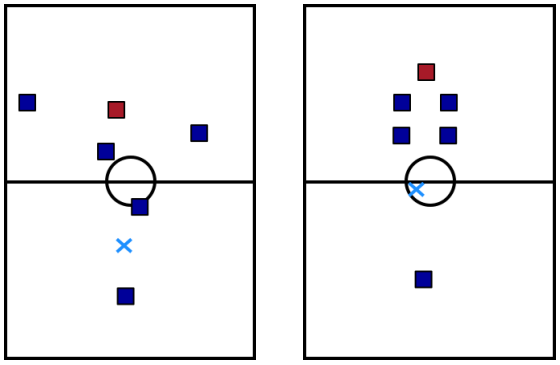
statyczne:



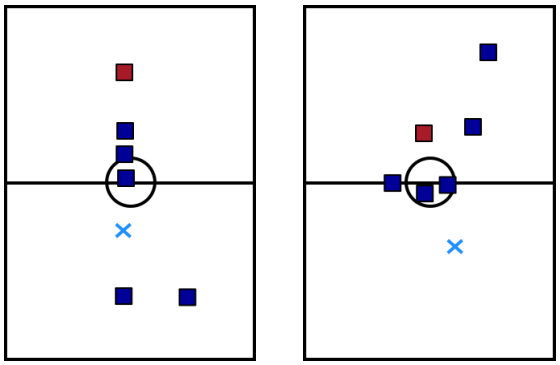
(a) (b)



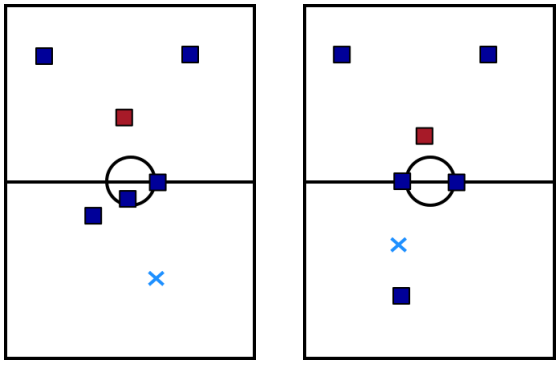
(c) (d)



(e) (f)

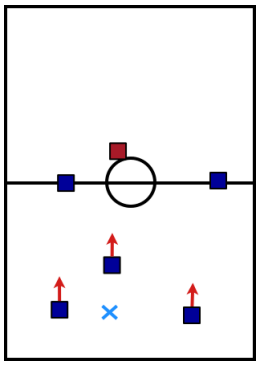


(g) (h)

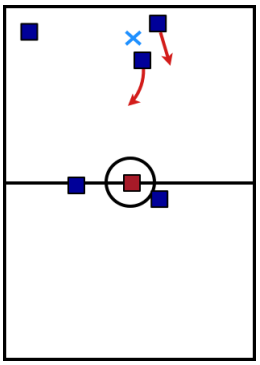


(i) (j)

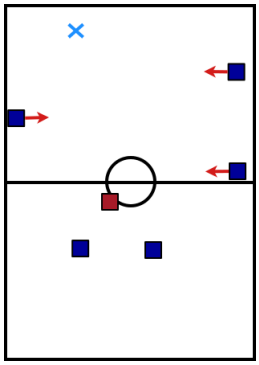
dynamiczne:



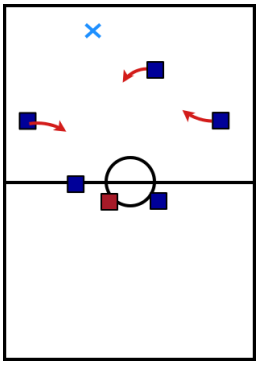
(k)



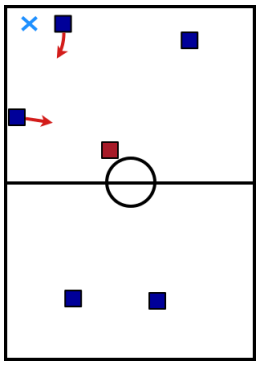
(l)



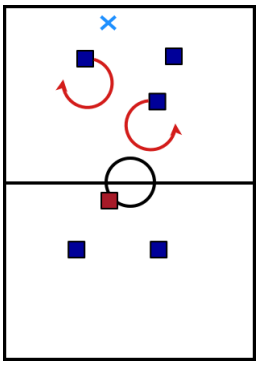
(m)



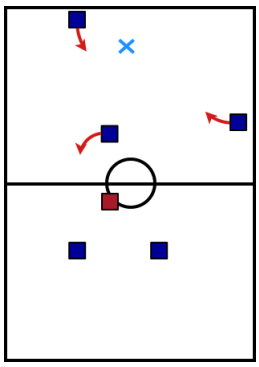
(n)



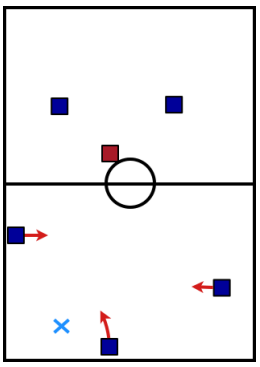
(o)



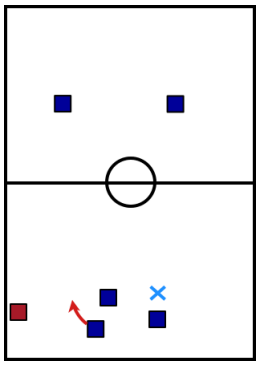
(p)



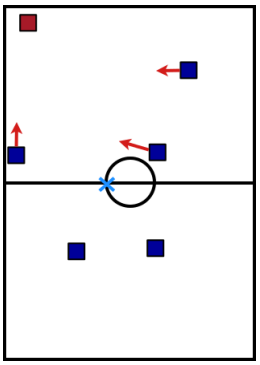
(q)



(r)



(s)



(t)

Zestawy wag używane w eksperymentach

Zestaw wag	$goalProb$	$wayPointProb$
1	0.0	0.0
2	0.0	0.2
3	0.0	0.3
4	0.0	0.4
5	0.0	0.5
6	0.0	0.6
7	0.0	0.7
8	0.0	0.8
9	0.0	0.9
10	0.0	1.0
11	0.1	0.0
12	0.1	0.1
13	0.1	0.2
14	0.1	0.3
15	0.1	0.4
16	0.1	0.5
17	0.1	0.6
18	0.1	0.7
19	0.1	0.8
20	0.1	0.9
21	0.2	0.0
22	0.2	0.1
23	0.2	0.2
24	0.2	0.3
25	0.2	0.4
26	0.2	0.5
27	0.2	0.6
28	0.2	0.7
29	0.2	0.8
30	0.3	0.0
31	0.3	0.1
32	0.3	0.2
33	0.3	0.3

Zestaw wag	$goalProb$	$wayPointProb$
34	0.3	0.4
35	0.3	0.5
36	0.3	0.6
37	0.3	0.7
38	0.4	0.0
39	0.4	0.1
40	0.4	0.2
41	0.4	0.3
42	0.4	0.4
43	0.4	0.5
44	0.4	0.6
45	0.5	0.0
46	0.5	0.1
47	0.5	0.2
48	0.5	0.3
49	0.5	0.4
50	0.5	0.5
51	0.6	0.1
52	0.6	0.2
53	0.6	0.3
54	0.6	0.4
55	0.7	0.0
56	0.7	0.1
57	0.7	0.2
58	0.7	0.3
59	0.8	0.0
60	0.8	0.1
61	0.8	0.2
62	0.8	0.2
63	0.9	0.0
64	0.9	0.1
65	1.0	0.0

Bibliografia

[1] Oficjalna strona Ligi *RoboCup* dostępna pod adresem:
`www.robocup.org`

[2] J. Bruce, M. Veloso: *Real-Time Randomized Path Planning for Robot Navigation*. Carnegie Mellon University

[3] J.Kim J.M. Esposito, V. Kumar: *An RRT-Based algorithm for testing and validating multi-robot controllers*. University of Pennsylvania, US Naval Academy

[4] I. Dulęba: *Metody i algorytmy planowania ruchu robotów mobilnych i manipulacyjnych*. Warszawa, Akademicka Oficyna Wydawnicza EXIT, 2001.

[5] T. Quasn, L.Pyeatt, J.Moore: *Curvature-Velocity Method for Differentially Steered Robots*. AI Robotics Lab Computer Science Department, Texas Tech University, 2003.

[6] R. Simmons: *The Curvature-Velocity Method for Local Obstacle Avoidance*. School of Computer Science, Carnegie Mellon University, 1996.

[7] M. Majchrowski: *Algorytm unikania kolizji przez robota mobilnego bazujący na przeszukiwaniu przestrzeni prędkości*. Praca Magisterska, Politechnika Warszawska, 2006.

[8] J. Borenstein, Y. Koren: *The vector field histogram – fast obstacle avoidance for mobile robots*. IEEE Transaction on Robotics and Automation, 1991.

[9] J. Borenstein,Y. Koren: *Histogramic in-motion mapping for mobile robot obstacle avoidance*. Department of Mechanical Engineering and Applied Mechanics, The University of Michigan, 1991.

[10] D. Fox, W. Burgard, S. Thrun: *The Dynamic Window Approach to Collision Avoidance*. Department of Computer Science, University of Bonn; Department of Computer Science, Carnegie Mellon University, Pittsburgh, 1991.

[11] B.Browning, J.Bruce, M.Bowling, M.Veloso: *STP: Skills, tactics and plays for multi-robot control in adversarial environments*. Carnegie Mellon University, Pittsburgh. 2004.

[12] R.Rojas, A.Gloye Föorster: *Holonomic Control of a robot with an omnidirectional drive.* Freie Universität Berlin, 2006.

[13] R.Rojas, A.Gloye Föorster: *Design of an omnidirectional universal mobile platform.* Eindhoven University of Technology, 2005.

[14] M.Gąbka, K.Muszyński Praca dyplomowa inzynierska *Środowisko symulacyjne i algorytm unikania kolizji robota mobilnego grającego w piłkę nożną* Politechnika Warszawska, 2008.

[15] J.Bruce, M.Bowling, B.Browning, M.Veloso *Multi-Robot Team Response to a Multi-Robot Opponent Team* Carnegie Mellon University, Pittsburgh.

[16] J.Bruce, M.Veloso *Real-time multi-robot motion planning with safe dynamics* Carnegie Mellon University, Pittsburgh.

[17] D. E. Koditschek, and E.Rimon *Robot navigation functions on manifolds with boundary* Advances in Applied Mathematics Volume 11 Issue 4, Dec. 1990.

[18] D. E. Koditschek, and E.Rimon *Exact robot navigation using artificial potential functions* IEEE Transactions on Robotics and Automation. Vol. 8, no. 5, pp. 501-518. Oct. 1992

[19] C. Zieliński, W. Szynkiewicz: *Konspekt do wykładu: Inteligentne Systemy robotyczne*. Politechnika Warszawska, 2008.

[20] N. Koenig: *Gazebo. The Instant Expert's Guide*. Player Summer School on Cognitive Robotics, Monachium 2007.

Todo list

napisać nowy wstęp . . . 4
przetlumaczyc na polski . . . 20
opisac jak zaimplentowano w ode brak tarcia . . . 26
dokonczyc rozdział opisac zrealizowana wersje algorytmu . . . 30
opisać naiwny model swiata . . . 30
opisać modul oceny . . . 30
wspomniec ze rrt to modul nawigacji . . . 30
opiać matematykę dryblowania z piłką . . . 35
zamiescic przykładowe trajektorie w srodowisku statycznym . . . 69
zamiescic przykładowe trajektorie w srodowisku dynamicznym . . . 69
poprawic opis instalacji gazebo . . . 81
zmienic numery rewizji zastosowanych aplikacji . . . 81