



Politechnika Warszawska  
Wydział Elektroniki i  
Technik Informacyjnych

WYDZIAŁ ELEKTRONIKI  
I TECHNIK INFORMACYJNYCH

Praca dyplomowa Inżynierska

Informatyka

Tytuł:

# Symulacja dookólnej bazy mobilnej

Autor:

Radosław Świątkiewicz

Opiekun naukowy:

dr hab. inż. Wojciech Szynkiewicz

Warszawa, 29 stycznia 2018

## Streszczenie

**Tytuł:** Symulacja dookólnej bazy mobilnej

**Słowa kluczowe:** symulacja, Gazebo, ROS, koła Mecanum, wielokierunkowa platforma mobilna, robot Velma, model dynamiki, ODE.

Niniejsza praca dotyczy projektowania i budowy środowiska symulacyjnego dla wielokierunkowej platformy mobilnej, poruszającej się za pomocą kół szwedzkich. Platforma jest bazą mobilną dla dwuramiennego robota Velma.

Celem pracy jest stworzenie możliwie dokładnego modelu symulacyjnego rzeczywistej bazy mobilnej. Model ten będzie służyć do wstępnych badań algorytmów planowania ruchu i sterowania robotem mobilnym.

Rozpatrzone są wymagania i problemy przy tworzeniu każdego ze składników środowiska. Na system składają się wirtualne efektory i receptory obsługujące odpowiednią maszynę symulacyjną, a także pakiety wspomagające symulację. Do wykonania zadania użyto programowej struktury ramowej ROS i symulatora Gazebo.

W ramach pracy, stworzone zostały modele dynamiczne, oraz kinematyczne platformy. Stworzono modele skanerów laserowych, jednostki inertjalnej, oraz enkoderów, a także narzędzia wspomagające testowanie i uruchomienie odpowiednich składników systemu.

Wykonano również szereg testów, mających na celu weryfikację działania opracowanego modelu.

## Abstract

**Title:** Simulation of an omnidirectional mobile base

**Keywords:** simulation, ROS, Gazebo, Mecanum wheels, omnidirectional mobile base, Velma robot, dynamic model, ODE physics engine

This thesis describes design and implementation of a simulation environment for omnidirectional mobile platform with four Mecanum wheels. This platform is a mobile base for a two-arm robot Velma.

The platform model and it's sensors models will then support the development of path-planning and movement algorithms, in order to safely test them before running on hardware.

The main goal of this thesis is to create a dynamic model of the real mobile platform.

To execute the task, ROS framework and Gazebo simulator were used. Simulation environment consists of dynamic and kinematic models, laser scanner, inertial measurement unit and simulated encoder. Also, many other components have been created in order to ease testing, data logging and execution of simulation.

In order to verify the correctness of the implemented model, a set of tests has been performed.

# Spis treści

<b>1 Wstęp</b>	<b>6</b>
1.1 Cel pracy . . . . .	6
1.2 Zakres pracy . . . . .	6
1.3 Podział tej pracy na sekcje . . . . .	7
<b>2 Budowa bazy mobilnej</b>	<b>8</b>
2.1 Dookólna platforma mobilna . . . . .	8
2.2 Koła szwedzkie . . . . .	11
2.2.1 Opis ruchu . . . . .	13
2.3 Enkodery . . . . .	14
2.4 Silniki kół . . . . .	15
2.5 Skaner laserowy . . . . .	15
2.5.1 Zasada działania . . . . .	15
2.5.2 Podstawowe cechy . . . . .	16
2.6 Jednostka inercyjna . . . . .	17
2.7 Sterowanie urządzeniami . . . . .	17
<b>3 Środowisko programistyczne</b>	<b>18</b>
3.1 <i>Robot Operating System (ROS)</i> . . . . .	18
3.2 Gazebo . . . . .	19
3.3 V-Rep . . . . .	20
3.4 Pozostałe narzędzia . . . . .	21
<b>4 Środowisko symulacyjne</b>	<b>22</b>
4.1 Zapis agentowy . . . . .	23
4.2 Model kinematyki . . . . .	24
4.2.1 Zachowanie . . . . .	25
4.3 Model dynamiki . . . . .	25
4.3.1 Zachowanie . . . . .	26
4.4 Model skanera laserowego . . . . .	26
4.5 Model jednostki inercyjnej . . . . .	26
4.6 Model kinematyki odwrotnej . . . . .	27
4.7 Manualne sterowanie . . . . .	27
4.7.1 Tryby działania . . . . .	27
4.8 Generator sterowania . . . . .	29
4.9 Wyłuskanie podwiadomości . . . . .	29
4.10 Podłoże o zmiennym współczynniku tarcia . . . . .	29
4.11 Algorytm usuwania szumu z danych jednostki inercyjnej . . . . .	29
4.12 Obserwator symulacji . . . . .	30
4.13 Scena z symulacją . . . . .	30
4.14 Rozdzielacz wiadomości . . . . .	30

4.15 Prosty program sterujący . . . . .	30
4.16 Struktury pakietów wiadomości . . . . .	31
4.17 Zewnętrzne pakiety ROSa . . . . .	31
4.17.1 Rysownik wykresów . . . . .	31
4.17.2 Wizualizer pomiarów . . . . .	31
4.17.3 Algorytm określania lokalizacji . . . . .	31
<b>5 Implementacja</b>	<b>32</b>
5.1 Istniejące implementacje . . . . .	32
5.2 Model 3D . . . . .	32
5.3 Ogólne typy pakietów . . . . .	33
5.3.1 Program wykonywalny w ROS . . . . .	33
5.3.2 Wtyczka Gazebo . . . . .	36
5.4 Mechanika przekształceń układów współrzędnych . . . . .	37
5.5 Instalacja ROSa . . . . .	38
5.5.1 Tworzenie pakietów . . . . .	39
5.6 Format SDF . . . . .	39
5.7 Model kinematyki . . . . .	40
5.7.1 Komunikacja . . . . .	40
5.7.2 Problemy implementacji . . . . .	40
5.8 Model dynamiki . . . . .	41
5.8.1 Wierność modelu . . . . .	41
5.8.2 Model koła z przywracaną orientacją . . . . .	41
5.8.3 Modyfikacja kierunków i wartości wektorów tarcia . . . . .	42
5.8.4 Ustawienie mas i momentów bezwładności . . . . .	44
5.8.5 Model silników . . . . .	44
5.8.6 Komunikacja . . . . .	45
5.8.7 Rozszerzenie modelu . . . . .	45
5.9 Model skanera laserowego . . . . .	46
5.9.1 Obliczenia symulatora . . . . .	46
5.9.2 Różnice między czujnikiem, a modelem . . . . .	46
5.9.3 Komunikacja . . . . .	47
5.9.4 Model w Gazebo . . . . .	47
5.9.5 Błędy . . . . .	48
5.10 Model jednostki inercyjnej . . . . .	49
5.11 Manualne sterowanie . . . . .	50
5.11.1 Program . . . . .	50
5.11.2 Komunikacja . . . . .	51
5.12 Generator sterowania . . . . .	51
5.13 Podłoże o zmiennym współczynniku tarcia . . . . .	52
5.14 Algorytm usuwania szumu z danych jednostki inercyjnej . . . . .	52
5.15 Obserwator symulacji . . . . .	52
5.16 Model kinematyki odwrotnej . . . . .	53
5.17 Scena z symulacją . . . . .	53
5.18 Rozdzielacz wiadomości . . . . .	53
5.19 Prosty program sterujący . . . . .	53
<b>6 Testy środowiska symulacyjnego</b>	<b>54</b>
6.1 Weryfikacja działania modelu dynamiki . . . . .	54
6.1.1 Porównanie modeli . . . . .	55
6.1.2 Powtarzalność testów . . . . .	58
6.2 Porównanie modelu z robotem . . . . .	59

6.2.1	Trasa bez rotacji . . . . .	59
6.2.2	Trasa z rotacją . . . . .	62
6.3	Jednostka inercyjna . . . . .	65
6.3.1	Akcelerometr . . . . .	65
6.3.2	Żyroskop . . . . .	68
<b>7</b>	<b>Podsumowanie</b>	<b>70</b>

# Rozdział 1

## Wstęp

### 1.1 Cel pracy

Celem pracy inżynierskiej jest budowa środowiska symulacyjnego robota mobilnego z kołami szwedzkimi. Dla realizacji tego celu należy opracować model 3D oraz model dynamiki dookółnej bazy jezdnej z 4 kołami szwedzkimi. Jednym z przyjętych założeń jest wymaganie, aby opracowany model był możliwie dokładny i jego działanie było zbliżone do rzeczywistego robota. Opisywana platforma będzie używana jako baza wielokierunkowa do przemieszczania dwuramiennego robota manipulacyjnego Velma.

Celem jest stworzenie modelu, który będzie reagował na siły podobnie do rzeczywistego robota i będzie sterowany tak samo, jak rzeczywisty robot. To spowoduje, że możliwe będzie stworzenie jednego wspólnego programu sterującego, do użycia zarówno w symulacji, jak i w robocie.

Testowanie oprogramowania sterującego na rzeczywistym obiekcie może prowadzić do jego uszkodzeń, dlatego wpierw należy się upewnić o poprawności projektowanych rozwiązań na bezpiecznym modelu wirtualnym. Środowisko symulacyjne pozwala także na skomplikowane scenariusze testów, które w rzeczywistości mogłyby być niemożliwe do wykonania lub koszty jego wykonania byłyby zbyt wysokie. Szybciej i taniej jest przeprowadzić symulacje, niż fizyczne eksperymenty, w dodatku błąd działania programu sterującego przy symulacji nie grozi zniszczeniem rzeczywistego robota. Dopiero po osiągnięciu satysfakcjonującej jakości sterowania w symulacji, można zastosować algorytmy sterowania do rzeczywistego obiektu bez ryzyka uszkodzeń urządzenia.

Oprócz modelu bazy jezdnej, środowisko symulacyjne musi również udostępniać modele czujników, w które wyposażony jest robot. Odczyty z symulatorów czujników są następnie wykorzystywane w układzie sterowania do generacji odpowiednich sygnałów sterujących. W celu możliwie wiernej symulacji działania czujników, do wartości pomiarów dodaje się szum pomiarowy i zakłócenia.

### 1.2 Zakres pracy

Do realizacji celu postawionego w pracy, należy wykonać następujące czynności:

1. Opracować model kinematyczny dookółnej bazy mobilnej z czterema kołami szwedzkimi.
2. Stworzyć model dynamiczny platformy do uruchomienia w symulatorze, zachowując wartości wszystkich parametrów rzeczywistego robota. Bryły składowe modelu muszą przypominać kształtem części z których składa się robot, należy im także ustawić parametry fizyczne, jak masę, moment bezwładności, materiał, itp.
3. Opracowanie modeli więzów dla koła, rolek oraz przegubu, aby maszyna symulacyjna poprawnie symulowała obiekt. Taki model powinien na tym stanie poprawnie reagować na wirtualne siły.
4. Opracowanie wtyczki symulatora, sterującej modelem, odczytującej odpowiednie dane z zewnątrz i wywołującej funkcje maszyny symulacyjnej, aby modyfikować ruch modelu. Na tym

poziomie można dobudować zamiennik programu sterującego, jedynie do podawania prostych wartości bez odczytywania pomiarów i sterowania. Porównanie z obiektem kinematycznym pozwala sprawdzić, czy model zachowuje się zgodnie z przewidywaniami.

5. Stworzenie wtyczki symulującej enkodery, aby generowały dane, bazując na pozycjach i prędkościach kół wirtualnych.
6. Dodanie czujników wirtualnych, zarówno tych, mających swoje odpowiedniki w rzeczywistości, jak i również całkowicie symulacyjnych.
7. Stworzenie interfejsów do zmiany różnych parametrów działania modelu.
8. Opracowanie programów pomocniczych, zbierających i wyświetlających dane oraz wspomagających przeprowadzanie testów.
9. Dodanie modelu skanera laserowego.
10. Dodanie modelu jednostki inercyjnej.
11. Przeprowadzenie testów, z porównaniem działania modeli i robota, w celu weryfikacji poprawności opracowanych rozwiązań.
12. Ustawianie parametrów modelu w celu przybliżenia go do zachowania platformy.

Po stworzeniu symulatora, następnym krokiem jest tworzenie głównego programu sterującego, którego testowanie będzie przeprowadzone na opracowywanym środowisku symulacyjnym. Program jest wspólny dla obu bytów — wirtualnego i rzeczywistego.

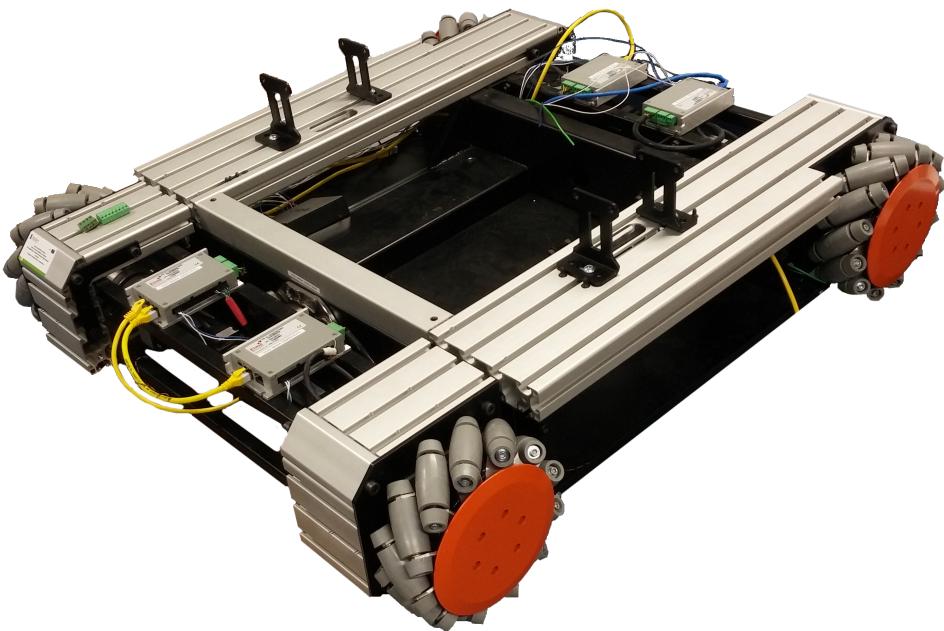
### **1.3 Podział tej pracy na sekcje**

Kolejne rozdziały kolejno opisują różne aspekty pracy. Rozdział 1 zawiera ogólny opis pracy. W rozdziale 2 opisano konstrukcję rzeczywistej platformy i czujników oraz zasadę jej działania. Następnie w rozdziale 3, opisano narzędzia programistyczne, użyte przy tworzeniu modeli i przeprowadzaniu testów. W rozdziale 4 opisano działanie opracowanych modeli i innych pakietów składających się na środowisko. Rozdział 5 zawiera opis implementacji opracowanych modeli, problemy i niedoskonałości z nimi związane. Opis poszczególnych dodatkowych składników systemu, używanych w symulacji, testowaniu, wizualizacji i wspomagających tworzenie. W rozdziale 6 przedstawiono wyniki badań oraz wnioski. W ostatnim rozdziale 7 zamieszczono podsumowanie pracy i dalsze możliwości rozwoju.

## Rozdział 2

# Budowa bazy mobilnej

### 2.1 Dookólna platforma mobilna



Rysunek 2.1: Dookólna baza mobilna na kołach szwedzkich.

Jest to duża, prostokątna baza dookólna poruszająca się na czterech kołach szwedzkich (rysunek 2.1). Koła są stałe, parami przytwierdzone do dwóch osi. Każde koło jest napędzane niezależnie przez podłączony bezpośrednio serwomotor, zatem może mieć prędkość i kierunek obrotu niezależny od pozostałych kół. Każdy z serwomotorów ma także wbudowany enkoder, który zwraca aktualny kąt i prędkość obrotu.

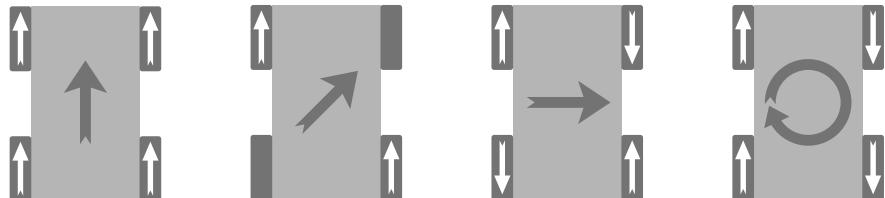
Jest to jeden z najpopularniejszych typów dookólnych platform mobilnych, mających zastosowanie także w innych robotach, jak na przykład Kuka Youbot (rysunek 2.2). Należy zwrócić uwagę na charakterystyczne ustawienie kół, identyczne jak w opisywanej platformie na rysunku 2.1.

Istnieją także roboty z trzema kołami szwedzkimi, w których koła rozstawione są na wierzchołkach trójkąta równobocznego. Pomimo prostszej budowy i takiej samej liczby stopni swobody, jak cztero-kołowa wersja, stabilność takiej konstrukcji jest gorsza [7]. Ponieważ jest to robot transportowy, to stabilność odgrywa tu istotną rolę i cztero-kołowa konstrukcja jest wskazana.



Rysunek 2.2: Platforma robota Kuka Youbot [18].

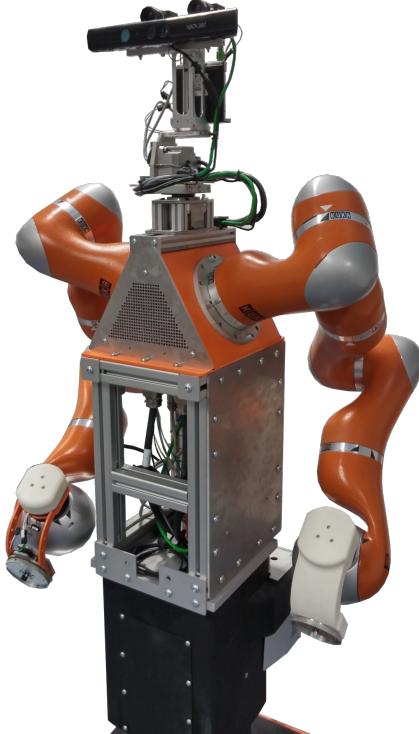
Odpowiedni obrót kół pozwala na jej ruch w dowolnym kierunku, niezależnym od orientacji robota, patrz rysunek 2.3. Jest możliwe także obracanie bazą, gdy ta porusza się w dowolnym kierunku, bądź stoi w miejscu.



Rysunek 2.3: Podstawowe kierunki ruchu robota z napędem wielokierunkowym.

Przykładowo, poruszając tylko przeciwnymi kołami po przekątnej, robot będzie mógł poruszać się po skosie, bez zmiany orientacji bazy. A jeśli do tego dodać obrót kół drugiej przekątnej, w odwrotnym kierunku, wtedy pojazd zacznie się poruszać w bok, pomimo faktu, że koła nie są skrętne i nie mogą ustawić się zgodnie z kierunkiem jazdy. Trasa, po której porusza się robot, przy stałej prędkości kół, zawsze jest łukiem okręgu. W szczególnych przypadkach można uznać prostą za okrąg o nieskończonym promieniu, a punkt za okrąg o zerowym. Wynika to z faktu, że każdy obiekt, który ma jednostajną prędkość i stały kierunek w lokalnym układzie współrzędnych oraz stałą prędkość kątową, będzie się poruszał po łuku. Zależność promienia okręgu  $R$  od prędkości liniowej  $v$  i prędkości kątowej  $\omega$ , wyraża się wzorem  $R = \frac{v}{\omega}$ .

Baza mobilna będzie podstawą dwuramiennego robota Velma, tworząc razem manipulator mobilny. Velma to dwuramienny robot, każde ramkę o 7 stopniach swobody, wyposażony w dwa chwytaki (patrz rysunek 2.4). Taka budowa wymaga szerokiej podstawy, aby zachować bezpieczną równowagę całości. Jeżdżąc na tej bazie, robot może się przemieszczać i obracać w dowolnym kierunku, aby zwiększyć swoją przestrzeń roboczą.



Rysunek 2.4: Dwuramienny robot Velma.

Platforma mobilna jest niesymetrycznie podzielona na dwie części, przednią i tylną, w sposób pokazany na rysunku 2.5. Przegub o jednym stopniu swobody jest jedynym łącznikiem pomiędzy tymi dwoma częściami. Zadaniem tego przegubu jest zmniejszanie wpływu nierówności podłożu na ruch bazy, aby każde koło zachowało stały kontakt z podłożem. Bez tego przegubu, ruch po nierównym terenie uniemożliwiałby sprawne sterowanie platformą na skutek nieprzewidywalnych zaników kontaktu kół z podłożem, powodując nieplanowane skręty. Takie zaniki kontaktu kół są niewykrywalne w bezpośredni sposób, jak to zostało opisane w [8].

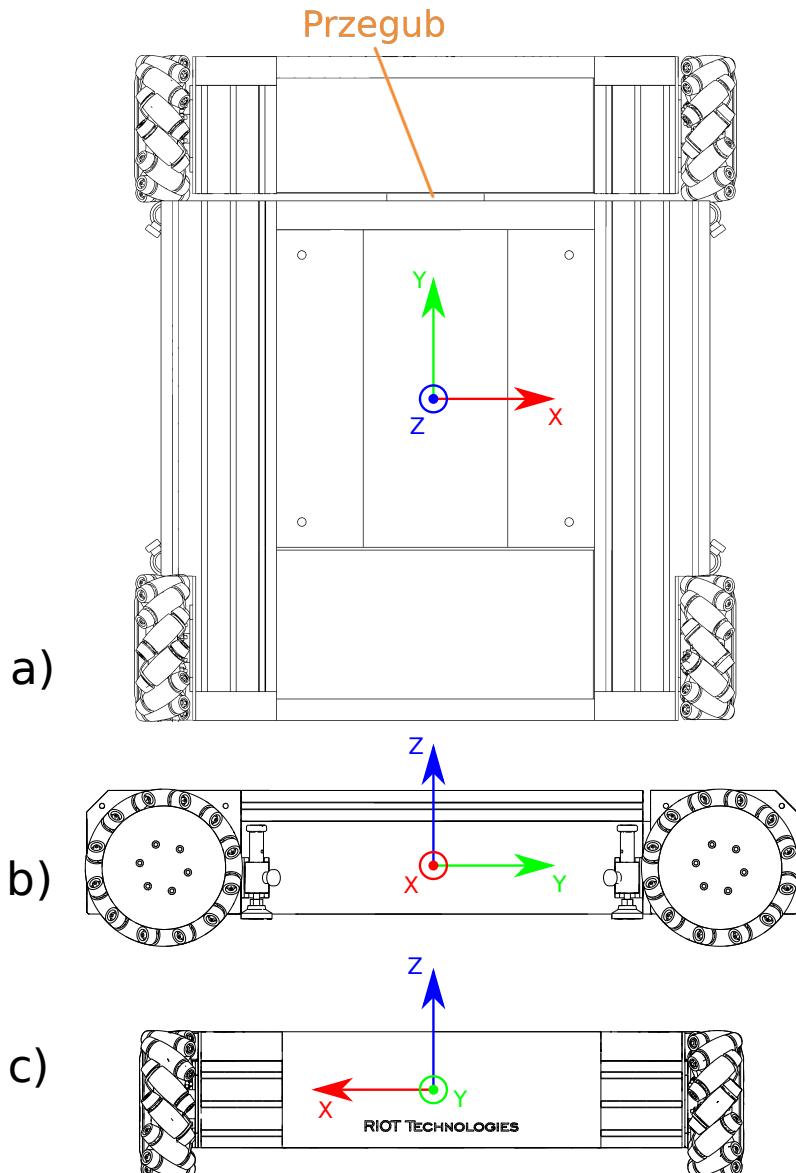
Platforma ma kształt prostokąta o wymiarach  $0,72 \times 0,76$  m (rysunek 2.6 i tabela 2.1). Koła ustawione są na wierzchołkach tego prostokąta.

Platforma może w trakcie hamowania poruszać się innym kierunku, niż tuż przed zatrzymaniem. Jest to spowodowane tym, że konstrukcja rolek powoduje poślizg platformy w kierunku obrotu rolki, mającej aktualnie kontakt z podłożem, a ten kierunek zależy od aktualnej orientacji bazy, nie od kierunku w jakim się porusza. Należy także uwzględnić inne cechy tego typu kół, jak nierówne tarcie poszczególnych rolek o powierzchnię [6].

Platforma ma 3 stopnie swobody.

- Przesunięcie wzdłuż osi X.
- Przesunięcie wzdłuż osi Y.
- Obrót wokół osi prostopadłej do podłoża.

Kierunek osi X i Y (patrz rysunek 2.5) układu jest zgodny z kierunkami przyjętymi w sterowaniu robotem.



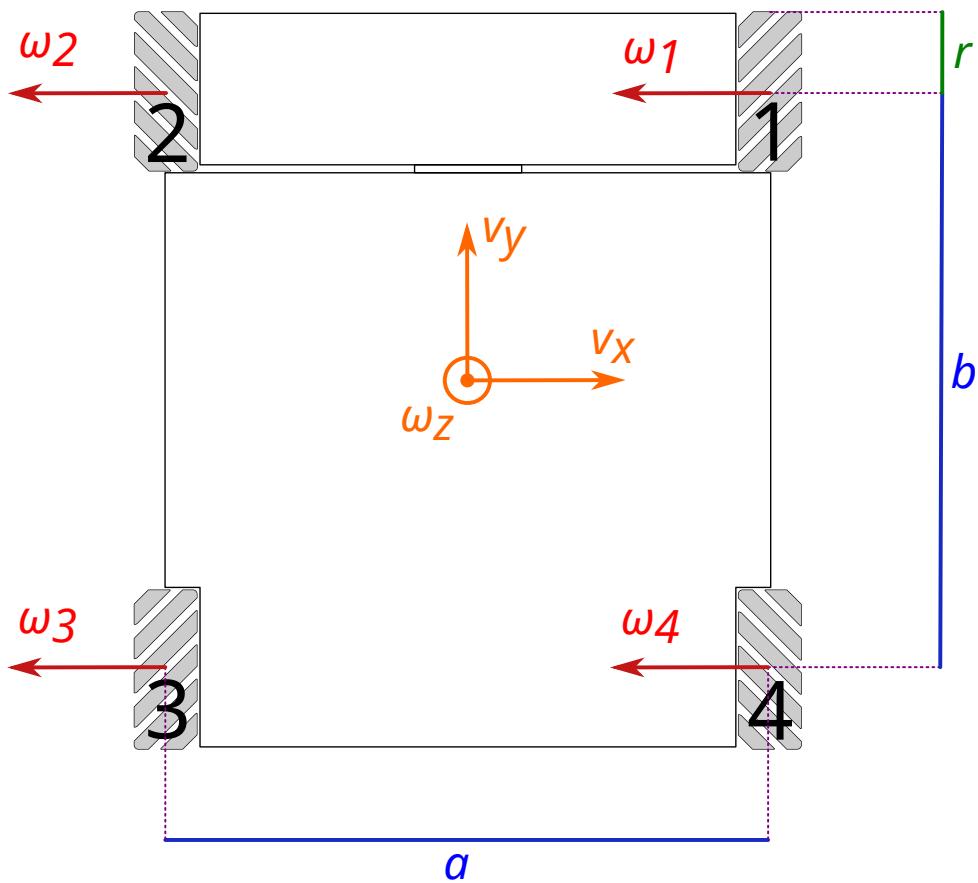
Rysunek 2.5: Platforma mobilna w lokalnym układzie współrzędnych. Widoki: a) od góry, b) od prawej strony i c) od przodu. Przegub obrotowy łączy dwie części.

## 2.2 Koła szwedzkie

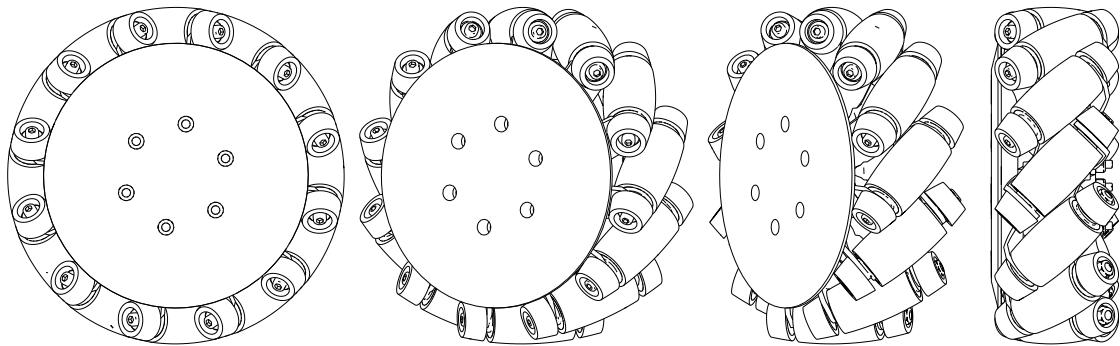
Koła szwedzkie, zwane także kołami Mecanum, to specjalne koła z dodatkowymi rolkami na obwodzie, ustawionymi pod kątem  $45^\circ$  do osi koła. Rolki są pasywne i obracają się niezależnie od siebie. Każde koło ma 12 takich rolek (patrz rysunek 2.7). W platformie ich osie ustawione są w ten sposób, że osie najwyższych, lub najniższych, rolek dwóch kół z tej samej strony platformy przecinają się pod kątem prostym. Innymi słowy, robot ma identycznie ustawione koła na przeciwnie wierzchołkach, i razem ustawione są w kształt litery X, patrząc na nie z góry. Należy pamiętać, iż oś dolnej rolki jest prostopadła do osi górnej rolki.

Koła zamontowane w platformie zostały wyprodukowane przez amerykańską firmę AndyMark [17]. Koła mają średnice o okrągłej liczbie 8 cali, czyli 20,32 cm oraz grubość 7,41 cm. Każde koło waży 1,6 kg i jest w stanie udźwignąć masę ok. 200 kg.

Każda rolka podzielona jest na 3 części (patrz rysunek 2.7) w celu lepszego zamocowania na kole, każda część może obracać się niezależnie. Teoretycznie zatem każde koło ma 36 rolek, lecz nie ma to znaczenia z punktu widzenia symulacji.



Rysunek 2.6: Parametry geometryczne bazy i prędkości.



Rysunek 2.7: Widok 12 rolkowego koła szwedzkiego platformy wielokierunkowej.

Oznaczenie	Wartość	Opis
$r$	0,1 m	Promień koła.
$a$	0,76 m	Rozstaw kół na tej samej osi.
$b$	0,72 m	Rozstaw osi.
$\omega_i$		Prędkość kątowa $i$ -tego koła.
$v_x$		Składowa prędkości liniowej wzdłuż osi X.
$v_y$		Składowa prędkości liniowej wzdłuż osi Y.
$\omega_z$		Prędkość kątowa wokół osi Z, wektor skierowany w góre.

Tablica 2.1: Parametry i zmienne modelu.

Każde koło ma 3 stopnie swobody [1].

- Obrót koła w osi prostopadłej do płaszczyzny koła i przechodzącej przez jego środek.
- Obrót pojedynczych rolek.
- Poślizg rolki w miejscu styku rolki z podłożem.

Na podstawie rysunku 2.7 można zauważyć, że krzywizna rolki jest tak dobrana, aby punkt kontaktu rolki z podłożem w czasie obrotu koła płynnie przechodził na następną rolkę. Celem jest utrzymanie stałej odległości osi obrotu koła od płaszczyzny podłożu. Nie powinno być efektu przeskoku z jednej rolki na drugą, gdyż powoduje to nierówne tarcie, losowe poślizgi i nadmierne zużycie elementów wykonawczych. Kształt pojedynczej rolki jest wycinkiem paraboloidy, wzory opisujące kształt rolki są złożone. Zazwyczaj przybliża się taką rolkę wycinkiem torusa, w celu uproszczenia produkcji [4].

Istnieją także koła o innej konstrukcji, złożone z wielu małych rolek, tak aby w każdym momencie więcej jak jedna rolka dotykała podłożu. Można także złożyć kilka tego typu kół obok siebie w jedno koło. Przydatne jest to dla robotów transportujących duże masy, gdyż zmniejsza to obciążenie pojedynczych rolek. Niestety, taka konstrukcja jest chroniona aktywnym patentem, więc pojedyncze koło, na które patent już wygasł, jest jedynym powszechnie używanym [3].

Pomimo skomplikowanej budowy, występują poślizgi rolek po powierzchni. Odległość osi obrotu koła od płaszczyzny podłożu nieznacznie zmienia się przy przenoszeniu obciążenia z rolki na rolkę, co przy dużych prędkościach powoduje drgania i jeszcze większe błędy wyznaczania pozycji na podstawie odometrii.

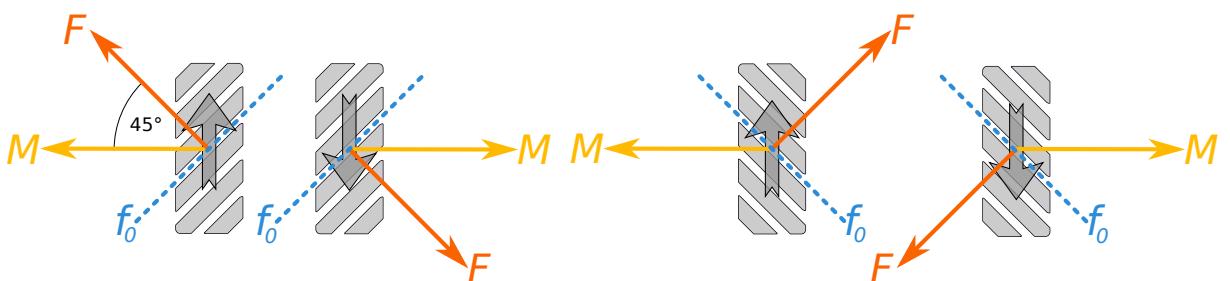
### 2.2.1 Opis ruchu

W zwykłym kole, dzięki sile tarcia, moment siły przekształcany jest na przyspieszenie w kierunku równoległym do podłożu i płaszczyzny koła. Dodatkowo wektory sił tarcia są równe we wszystkich kierunkach. To znaczy, koło będzie stawało identyczny opór, niezależnie czy wektor siły będzie równoległy do osi koła, czy prostopadły do niego.

Specjalne koło Mecanum nie powoduje sił tarcia w jednym z kierunków, oznaczonych jako  $f_0$  nad rysunku 2.8. Ten kierunek obrócony jest o  $45^\circ$  w stosunku do osi koła i jest zgodny z kierunkiem obrotu dolnej rolki.

Po przyłożeniu momentu siły  $M$ , siła tarcia zadziała jedynie w kierunku prostopadłym do  $f_0$ , gdyż tylko w tym kierunku dolna rolka nie będzie w stanie się obracać. Czyli wypadkowy wektor siły  $F$ , nadający przyspieszenie kołu, również będzie zwrócony o kąt  $45^\circ$  w stosunku do osi obrotu rolki.

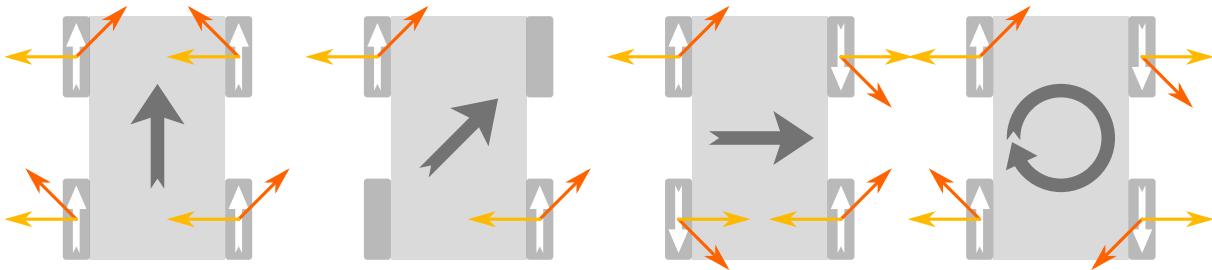
Suma wektorów  $F$  od wszystkich kół platformy nada jej odpowiedni kierunek ruchu.



Rysunek 2.8: Wektory momentu siły i siły dla koła Mecanum, widzianego z góry.

Ustawiając koła w odpowiedni sposób (rysunek 2.6), można wywołać odpowiednie znoszenie się wektorów sił, a w efekcie umożliwić platformie poruszanie się w kierunkach nieosiągalnych dla pojazdów o standardowych kołach.

Można przedstawić te wektory na wcześniejszym rysunku 2.3.



Rysunek 2.9: Ruchy platformy widziane z góry, z nałożonymi składowymi wektorów sił.

Należy wytłumaczyć podstawowe ruchy platformy. Kierunki osi są takie same, jak opisane wcześniej (rysunek 2.5).

1. Składowe wektorów sił w kierunku X znoszą się, ponieważ mają przeciwnie zwroty na lewej i prawej parze kół. Pozostają jedynie składowe równoległe do osi Y, które powodują prostoliniowy ruch w tym kierunku.
2. Dwa koła nie mają nadanego momentu siły. Wektory sił od pozostałych kół nadają platformie ruch pd kątem  $45^\circ$  do osi Y. Warto zwrócić uwagę, że ruch odbywa się równolegle do kierunku  $f_0$  dwóch zatrzymanych kół, zatem zgodnie z obrotem ich dolnych rolek, więc te koła nie powodują powstawania siły tarcia.
3. Tutaj również składowe wektorów sił, równoległe do osi Y, znoszą się, podobnie jak w przypadku ruchu naprzód. Pozostają składowe równoległe do osi X, które nadają platformie przyspieszenie w bok.
4. Prędkość kątowa wokół osi Z powstaje, gdy wypadkowa siła od kół po jednej stronie platformy znowu się z wypadkową siłą po drugiej stronie.

Warto nadmienić, że w trakcie ruchu równolegle do osi Y, w idealnym przypadku rolki nie obracają się. Inaczej mówiąc, prędkość rolki będzie tym większa, im bardziej ruch koła wymuszany jest równolegle do osi koła.

Przykładowo, przy ruchu w przód, rolki koła się nie obracają, lecz przy ruchu w bok ich prędkość obrotu jest znacznie większa. Ma to wpływ na zużywanie się tych elementów, nie tylko z punktu widzenia ilości obrotów danej rolki na pokonanym dystansie, ale także sposobu w jaki wymuszany jest jej ruch.

Rolki kół przy jeździe zawsze obracają się nieznacznie w obie strony, ze względu na nierówności powierzchni. Zatem przejazd przykładowego odcinka, przy platformie ustawionej przodem do kierunku jazdy, lub bokiem, będzie w różnym stopniu i w różny sposób zużywał elementy wykonawcze robota. To, jak dokładnie zużywają się przeguby i jaki styl jazdy opłaca się zastosować, aby zminimalizować uszkodzenia elementów jest dużą, odrębną dziedziną nauki. Odpowiednio skomplikowany algorytm sterowania może brać pod uwagę tą właściwość rolek.

## 2.3 Enkodery

Silniki kół mają zamontowane enkodery. Czujniki te umożliwiają pomiar prędkości kątowej i kąta obrotu koła. Korzystając z modelu kinematyki, można obliczyć z tych danych wypadkową prędkość robota, a następnie, za pomocą całkowania, wyznaczyć aktualną względną pozycję w stosunku do pozycji startowej.

## 2.4 Silniki kół

W opisywanym robocie, silniki kół są w stanie nadać im duży moment siły. To oznacza, że wartości prędkości kątowej kół, wykrytej przez enkodery, są zbliżone do prędkości zadanej. Ta właściwość jest potwierdzona testami (sekcja 6.2.1). Eksperymenty pokazały, że zasilacz robota ma ograniczony przesył prądu, więc jeśli silniki będą generować zbyt duże momenty sił, może dojść do awaryjnego odłączenia się zasilacza, co jest głównym powodem dla którego należy ograniczać nadawanie zbyt dużych przyspieszeń platformie.

## 2.5 Skaner laserowy

Poślizg kół powoduje, że odometria, bazująca na danych generowanych przez enkodery, obarczona jest błędami losowymi i nie może być użyta jako jedyna metoda wyznaczania pozycji względnej w trakcie jazdy robota [5].

Dodatkowym czujnikiem, używanym do wyznaczania pozycji platformy, jest skaner laserowy. Platforma wyposażona jest w dwa czujniki typu LiDAR firmy SICK. LiDAR to zbitek wyrazów *light* i *radar*.



Rysunek 2.10: Skaner laserowy SICK LMS100-10000.

### 2.5.1 Zasada działania

Wszystkie skanery tego typu mają bardzo podobną zasadę działania. W środku urządzenia znajduje się obrotowe lusterko, zwrócone pod kątem  $45^\circ$  do osi obrotu. Równolegle do osi jego obrotu znajduje się laser, który emituje impulsy z zakresie podczerwieni. Aktualna pozycja lusterka jest mierzona przez enkoder. Obok lasera jest czujnik, który bada odbite od obiektu światło.

We wbudowanym mikrokontrolerze następuje pomiar czasu przelotu i natężenia światła. Na tej podstawie wyznaczana jest odległość obiektu od skanera. Skaner odpowiada także za usunięcie nie-

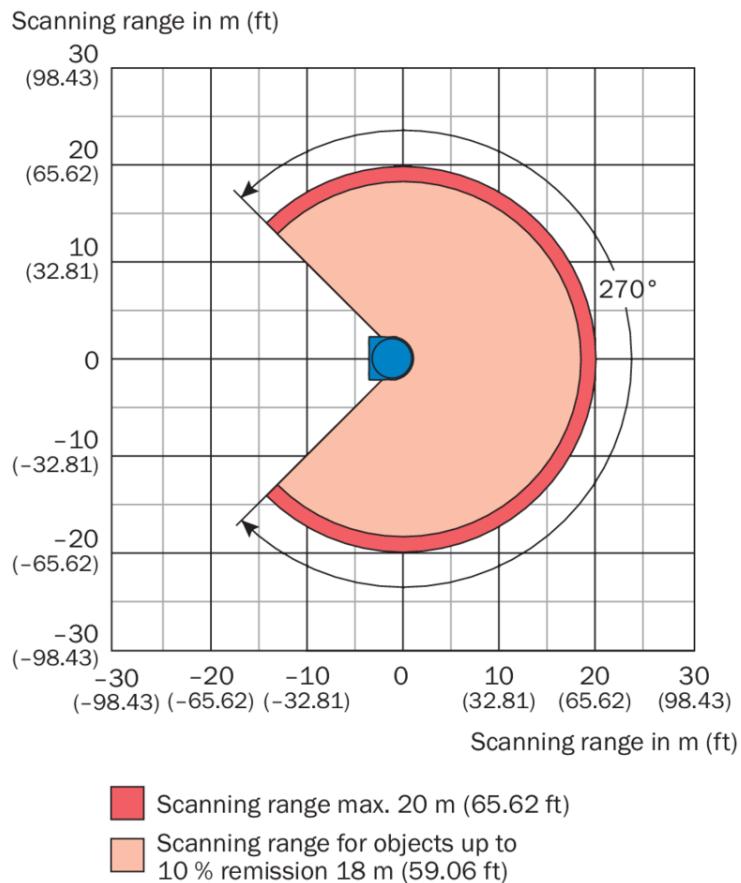
Cecha	Wartość
Zakres kątowy pracy	270°
Długość fali światła lasera	905 nm (podcerwien)
Częstotliwość skanowania	25 Hz / 50 Hz
Maksymalna odległość obiektu	≈ 20 m
Rozdzielcość kątowa	0,25° / 0,5°
Systematyczny błąd pomiarowy	± 0,03 m
Przypadkowy błąd pomiaru odległości	0,012 m

Tablica 2.2: Podstawowe cechy czujnika laserowego.

których błędów pomiarowych i ewentualnych odbić promienia. Komunikacja z urządzeniem odbywa się po sieci Ethernet.

### 2.5.2 Podstawowe cechy

Czujnik składa się z dwóch części, głównego trzonu, oraz nakładki. Połączenie tych elementów powoduje, że jego zakres pomiaru posiada martwy kąt. Przedstawia to dobrze grafika producenta 2.11.



Rysunek 2.11: Wykres przedstawiający zasięg pomiarowy czujnika [14].

Na podstawie danych z tabeli 2.2 można obliczyć, że w jednym przebiegu po całym zakresie kątowym urządzenia, emitowane jest około 1080 lub 540 impulsów (w zależności od trybu działania). Taka liczba promieni wymagana jest w symulacji, aby wiernie odwzorować urządzenie.

## 2.6 Jednostka inercyjna

Ten czujnik to małe urządzenie, posiadające zazwyczaj zestaw wewnętrznych czujników, przydatnych przy określaniu prędkości, orientacji i przyspieszeń modułu. Dodatkowo, wiele zestawów tego typu posiada także czujniki pola magnetycznego lub nawet termometry.

Czujnik użyty w platformie to ADIS16460AMLZ, firmy Analog Devices [15]. Czujnik jest wyposażony w:

- Trzyosiowy żyroskop.
- Trzyosiowy akcelerometr.
- Czujnik temperatury.
- Sprzętowe wspomaganie korekcji błędów i kalibracji.

Błędy pomiarowe akcelerometra są bardzo duże, w stosunku do błędów pomiarowych żyroskopu i skanera laserowego. Aby użyć tych danych w programie, należy zastosować algorytmy filtrujące szum i uśredniające wyniki.

W symulacji nie jest używana informacja o temperaturze otoczenia, zatem nie ma potrzeby jej symulować.

## 2.7 Sterowanie urządzeniami

Urządzenia robota przyjmują sterowanie w formie odpowiednich dla siebie pakietów sieciowych. Struktury wiadomości, używane w komunikacji w ROSie, nie mogą być bezpośrednio użyte do sterowania sprzętowego. W tym celu platforma podłączona jest do specjalnego komputera, pracującego z systemem operacyjnym czasu rzeczywistego, który to odpowiada za bezpośrednią komunikację z platformą.

Komendy dla robota w formie pakietów sieciowych, zawierających bezpośrednie wiadomości środowiska ROS, są nadawane poprzez sieć do tego pośrednika, który na ich podstawie odpowiednio obsługuje urządzenia. Możliwe jest także odbieranie pomiarów z czujników w podobny sposób.

## Rozdział 3

# Środowisko programistyczne

W tym rozdziale opisane są narzędzia użyte do wykonania zadania.

Środowisko symulacji składa się z symulatora, który posiada maszynę do symulacji fizyki, odpowiedzialną za obliczenia fizyczne, a także API do obsługi całej symulacji. Oprócz symulatora istnieją także pakiety generujące dane, filtrujące dane i zapisujące dane.

Zaawansowana maszyna symulacyjna fizyki powinna poprawnie modelować tarcia, więzy na ruch obiektów, przyłożone siły, materiały fizyczne dla określania tarcia i sprężystości obiektów oraz wszystko to, co potrzebne do jak najwierniejszego odtworzenia zachowania rzeczywistego obiektu.

Istnieje wiele różnych maszyn, zarówno do symulacji w czasie rzeczywistym, jak i wsadowym. Istnieją technologie otwartoźródłowe oraz o zamkniętym kodzie. Mogą używać tylko procesora, lub też być wspomagane przez kartę graficzną (na przykład *PhysiX*). Niektóre maszyny symulują, prócz zderzeń obiektów, także rozpływ cieczy, dymy, płótna, ciała sprężyste i strukturę wewnętrzną brył, lecz te funkcjonalności nie są potrzebne dla symulacji opisywanej platformy. Nazywa się je czasami „silnikami symulacji fizyki”, co jest bezpośrednim tłumaczeniem nazwy *physics engine* z języka angielskiego. W projekcie użyto maszyny dostępnej domyślnie w symulatorze Gazebo.

### 3.1 *Robot Operating System (ROS)*

ROS nie jest to systemem operacyjnym, lecz programową strukturą ramową (*framework*), zawierającą odpowiednie biblioteki i narzędzia do tworzenia programów sterujących [12]. Dostępne są algorytmy wyznaczania tras, budowy map, manipulowania robotycznymi ramionami, wizualizacji danych, itp.

System składa się z demona ROS, odpowiedzialnego za działanie systemu, węzłów, które są programami wykonywalnymi oraz strumieni komunikacyjnych, które pozwalają na wymianę informacji pomiędzy węzłami. W kanałach komunikacyjnych przesyła się wiadomości określonego typu. Zainteresowany węzeł zgłasza chęć publikacji lub subskrybowania danego typu wiadomości na dany temat, a demon automatycznie kieruje dane pomiędzy procesami. Nie jest wymagane istnienie nadawcy, aby zainicjalizować odbiorcę i nie musi istnieć odbiorca nadawanej wiadomości.

Struktura danych oparta jest o pakiety. Każdy pakiet może zawierać programy wykonywalne, dane, lub definicje. Pakiety są zależne od siebie w kwestii wykonania lub komplikacji. Programy mogą być pisane w C++ lub Pythonie i mogą używać funkcji bibliotecznych dostarczanych przez system.

Pakiet jest katalogiem zawierającym w sobie pliki opisujące jego parametry i skrypty CMake, używane do komplikacji. W symulacji opisywanej platformy, modele są pakietami, zawierającymi biblioteki ładowane dynamicznie i opisy budowy modeli, uruchamiane przez jeszcze inny pakiet symulatora Gazebo.

ROS potrzebuje także działającego demona w tle. Odpowiada on za komunikację i kontroluje stany wszystkich węzłów. Z punktu widzenia konstrukcji systemu, można porównać go do jądra systemu operacyjnego, a węzły do działających procesów. Dlatego też nazwa *Robot Operating System* nie jest przypadkowa.

Na stronie internetowej ROSa znajduje się bogata biblioteka pakietów. Każdy może także umieścić tam swój własny pakiet, aby mógł być on wykorzystany w projektach tworzonych przez inne osoby.

Komunikacja pomiędzy programami odbywa się w sposób ciągły przez strumienie wiadomości lub pojedyncze asynchroniczne wywołania, zwracające wynik, jak odwołanie się klienta do serwera. Można buforować wiadomości, podglądać strumienie, podłączać nadawcę do kilku odbiorników, podglądać graf komunikacji pomiędzy węzłami, itp. Do wszystkiego służy bogaty zestaw komend i wbudowanych narzędzi.

Używane wbudowane narzędzia z tej struktury ramowej to:

`rosbag` Narzędzie do zbierania i odtwarzania danych, przesyłanych przez kanał komunikacyjny.

`catkin` System budujący pakiety, działający na skryptach CMake.

`roslaunch` Program do wykonywania skryptu uruchamiającego węzeł w określony sposób.

`rosrun` Program do uruchamiania pliku wykonywalnego z pakietu.

`rostopic` Narzędzie do zarządzania węzłami, wysyłania i odbierania wiadomości i podglądania strumieni komunikacyjnych.

`roscore` Demon ROS, zarządzający wszystkimi węzłami.

Dodatkowo, używane funkcjonalności funkcji bibliotecznych:

- Rejestracja nowego węzła w demonie ROS.
- Nadawanie danych do strumienia wiadomości.
- Odbieranie danych od strumienia wiadomości.
- Zapisywanie danych do dziennika (wysyłanie wpisów do logów).
- Zarządzanie funkcją macierzy przekształceń jednorodnych.
- Zawieszenie programu.

## 3.2 Gazebo

Gazebo [10] jest symulatorem graficznym, działającym na podstawie uprzednio przygotowanych plików konfiguracyjnych. Zazwyczaj używany w trybie wsadowym, uruchamiany z argumentami z linii poleceń i plikiem opisującym symulację. Plik ten zawiera nazwy i ścieżki umieszczanych w symulacji modeli i wtyczek. Z tego powodu interfejs graficzny jest dość ubogi.

Program wykonuje symulację z wykorzystaniem podanych modeli, używając jednego z czterech popularnych maszyn symulacyjnych: ODE, Bullet, Simbody lub DART. Wszystkie te symulatory są wolnym oprogramowaniem i używane są także w innych programach, na przykład w edytorze Blender. Zmiana maszyny w symulatorze wiąże się z rekompilacją całego programu.

Symulator oprócz tego ma wbudowany edytor modeli, w którym można składać i ustawiać odpowiednie obiekty razem w przestrzeni trójwymiarowej i generować plik opisujący symulację. Funkcjonalność tych edytorów jest bardzo ograniczona, brak jest tak podstawowych funkcji, jak cofanie ruchu. Dlatego lepiej jest zdefiniować model w pliku tekstowym. Również tworząc modele poza edytorem, posiada się nad nimi większą kontrolę, a parametry składowych da się ustawiać z dowolną dokładnością.

Gazebo przyjmuje definicje modeli w specjalnym formacie SDF. Jest to standaryzowany, zdefiniowany niezależnie od symulatora format do opisywania budowy robotów i czujników. Dzięki temu plik SDF może być użyty w innej symulacji, w innym programie, pod warunkiem przestrzegania standardu.

Składnia jest zgodna ze standardowym językiem XML, co znaczy że może być tworzona na dowolnym edytorze tekstowym.

Wtyczka do sterowania modelem jest skompilowaną biblioteką, dołączaną na starcie programu. Tworzy się ją w C++ lub Pythonie jako klasę dziedziczącą po abstrakcyjnej klasie dostarczonej przez Gazebo. Dzięki temu może korzystać z wielu funkcji systemu operacyjnego. Z punktu widzenia ROSa, programy uruchamiane w Gazebo są osobnymi węzłami, które komunikują się z węzłem symulatora za pomocą dodatkowych kanałów komunikacyjnych.

Program jest w pełni wspierany na dystrybucji GNU/Linuksa Ubuntu ale bez problemu można go także skompilować na innych dystrybucjach. Nie wspiera innych systemów operacyjnych. Interfejs jest dopracowany i przestrzega systemowych ustawień DPI, lecz nie korzysta z dedykowanych bibliotek do tworzenia interfejsów typu Qt lub GTK. Uruchamianie programu jest proste i nie wymaga dodatkowych ustawień, wywoływania skryptów inicjalizujących, tworzenia odpowiednich katalogów, czy definiowania zmiennych systemowych. Tworzy ukryty katalog w katalogu domowym użytkownika, gdzie składa się wszystkie dostarczone modele, ustawienia i pliki dziennika.

Gazebo jest dystrybuowany także jako pakiet ROS. Kolejne wersje Gazebo są powiązane z kolejnymi wersjami ROSa, nie można użyć przestarzałej wersji Gazebo z nowszym ROSem i odwrotnie. Ze względu na chęć zachowania wysokiej kompatybilności pakietów ROSa, nie zawsze najnowsza wersja symulatora jest dostarczana razem z najnowszą wersją programowej struktury ramowej.

Gazebo implementuje prawie wszystkie elementy standardu SDF, ale tylko niektóre będą używane. Posiada także kilka narzędzi do wizualizacji wygenerowanych danych, lecz będą użyte te narzędzia, które zostały dostarczone przez środowisko ROS.

- Symulacja fizyki za pomocą maszyny do symulacji ODE.
- Całkowanie prędkości poprzez umieszczenie obiektu kinematycznego w przestrzeni wirtualnej.
- Możliwość modyfikacji wektorów tarcia obiektem.
- Dane o prędkościach i pozycjach wszystkich obiektów na scenie.
- Symulacja skanera laserowego.
- Symulacja jednostki inercyjnej.
- Wizualizacja modeli za pomocą siatki trójkątów i kolorów.
- Wizualizacja kolizji, kształtów i inercji obiektów.
- Wizualizacja położenia i orientacji poszczególnych obiektów, ich lokalne układy współrzędnych.

### 3.3 V-Rep

V-Rep [11], to duże i złożone środowisko, reklamowane posiadaniem zaawansowanych mechanizmów i funkcji przydatnych w robotyce. Pomimo otwartego kodu, użycie komercyjne jest płatne. Dla zastosowań akademickich użycie jest bezpłatne. Bogaty interfejs graficzny zakłada budowę i symulację wszystkiego w tym jednym programie.

W środowisku używa się dwóch z maszyn symulacyjnych, wykorzystywanych w Gazebo, czyli ODE i Bullet, oraz dodatkowo Vortex i Newton. Z tej czwórki tylko Vortex ma zamknięty kod.

Głównym mankamentem programu jest sposób zapisywania modeli. Program tworzy pliki binarne własnego formatu, co uniemożliwia edycję i wizualizację modelu bez uruchamiania całego programu i importowania modelu do symulacji. Brak przenośności, czy wsparcia systemu kontroli wersji dla takich nietekstowych plików także jest problemem.

Pisanie wtyczek najczęściej odbywa się w języku Lua. Poprzez komunikację sieciową, są też też dostępne inne języki, jak C, Matlab, Java, itp. Komunikacja z innymi programami odbywa się poprzez

specjalne wtyczki do środowiska. API pozwala stworzyć mały, wbudowany w edytor interfejs graficzny do sterowania symulacją poprzez przyciski i suwaki.

Ze strony producenta pobrać można gotowe archiwum z programem, który nie wymaga żadnej instalacji i posiada wszystkie potrzebne zasoby do pracy i nauki, jak przykładowe modele istniejących komercyjnych robotów. Program działa w trzech najpopularniejszych systemach operacyjnych — Windows, Linux i OS X.

Przy wykonywaniu zadania, użyty będzie jeden z gotowych modeli robotów wielokierunkowych, wspomniana wcześniej Kuka Youbot, aby na jego podstawie zbudować model opisywanej platformy. Zostanie zapisany skrypt w Lua, który będzie używał wbudowanych mechanik do komunikacji ze strukturą ramową ROS. Maszyna do symulacji fizyki ODE, ta sam co w Gazebo, będzie zastosowana w symulacji, ponieważ daje najlepsze wyniki w porównaniu z innymi. Interfejs graficzny do sterowania robotem nie będzie używany.

### 3.4 Pozostałe narzędzia

Do tworzenia oprogramowania na systemach Unixowych można użyć dowolnych edytorów, gdyż standardowo wszystko jest potem komplikowane za pomocą narzędzi wiersza poleceń i skryptów. Jednak warto sobie ułatwić pracę zaawansowanymi środowiskami graficznymi.

**CMake** to popularny i używany przez ROS i Gazebo system budowy kodu. Program tworzy na podstawie swoich plików konfiguracyjnych plik `makefile` do komplikacji źródeł i łączenia bibliotek.

**GCC** będzie użyty do komplikacji, gdyż jest to najpopularniejszy tego typu program używany w GNU/Linux. Same symulatory zostały w nim skompilowane.

**KDevelop** jest graficznym edytorem tekstowym i nadaje się do pisania komplikowanego kodu wtyczek. Posiada mechanizm interpretacji błędnych linii kodu przed komplikacją, debugowania i podobne.

**Bash** będący bardzo popularnym językiem skryptowym nadaje się do automatyzacji pracy i uruchamiania testów w kontrolowany i prosty sposób.

**Git** jest narzędziem kontroli wersji, używanym przy bardzo wielu projektach informatycznych. Pozwala na łatwe umieszczenie kodu w repozytorium GitHub.

**FreeCAD i Blender** służyły do zarządzania modelem CAD bazy w celu wygenerowania kształtów modeli i siatek trójkątów do wizualizacji.

Dodatkowo, narzędzia użyte w tworzeniu tej pracy naukowej:

**Gnuplot** służy do generowania wykresów z danych, zapisanych w pliku tekstowym.

**Dia** to graficzny edytor do tworzenia diagramów UML.

**LATEX** jest systemem do tworzenia dokumentów.

**Kile** to edytor kodu LATEXa.

**Inkscape** to program do edycji rysunków wektorowych, użyty do stworzenia większości obrazków.

## Rozdział 4

# Środowisko symulacyjne

W tym rozdziale opisane są stworzone składniki systemu, czyli modele dynamiki i kinematyki, modele czujników oraz pakiety wspomagające testowanie.

Aby uruchomić symulację, nie wystarczy uruchomienie symulatora Gazebo z modelami, należy zadbać także o nadawanie sterowania i odbieranie danych. Do tego potrzebne są programy wspomagające w formie pakietów ROS, które łączy się w różne konfiguracje, w zależności od scenariusza testowego. Ze względu na niezależność pakietów od siebie, można ich także użyć przy komunikacji z robotem.

Niektóre typy wiadomości ROS posiadają wbudowany nagłówek, inne istnieją w dwóch wersjach, z nagłówkiem i bez. Dopisek Stamped informuje o posiadaniu nagłówka. Nagłówek ma trzy pola:

- Numer sekwencyjny, zwiększany przez program wysyłający po każdej wysłanej wiadomości.
- Czas nadania wiadomości, z dokładnością do nanosekund.
- Identyfikator macierzy przekształcenia jednorodnego, według której podano dane, ta funkcjonalność została opisana dokładniej w sekcji 5.4.

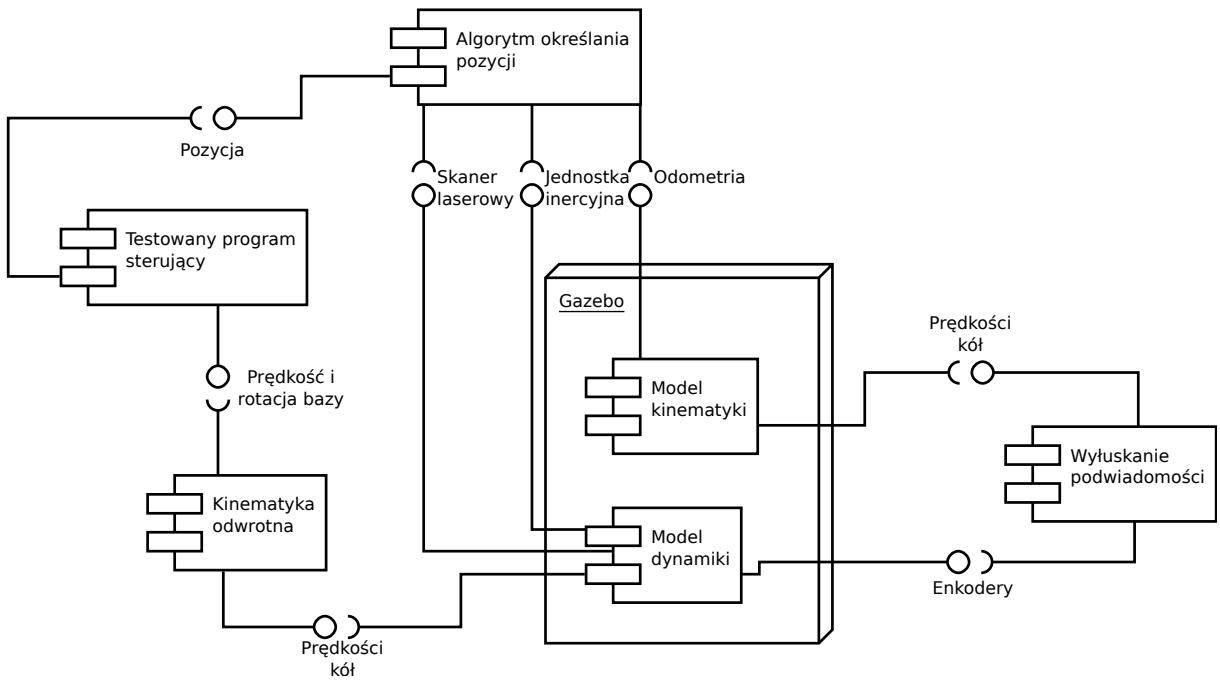
Węzły można podzielić na trzy typy:

- Generujące dane.
- Przekazujące filtrujące dane.
- Zbierające dane.

W tym rozdziale każdy pakiet opisany jest bardziej szczegółowo, wraz z jego interfejsem. W trakcie testowania symulatora, podłączenie pakietów będzie wyglądać jak na rysunku 4.1.

Typ	Opis
omnivelma_msgs/Encoders	Prędkości kątowe i kąty obrotu kół z enkodera.
omnivelma_msgs/Vels	Prędkości kątowe kół.
omnivelma_msgs/SetFriction	Nadanie tarcia elementowi modelu.
omnivelma_msgs/SetInertia	Nadanie mas i momentu bezwładności obiektywu.
geometry_msgs/Pose	Pozycja obiektu w przestrzeni kartezjańskiej.
geometry_msgs/Twist	Prędkość względna obiektu.
sensor_msgs/LaserScan	Jedno skanowanie skanera laserowego.
omnivelma_msgs/Relative	Odległość i kąt pomiędzy obiektywami.
nav_msgs/Odometry	Pozycja obiektu z macierzą kowariancji.
sensor_msgs/Imu	Dane generowane przez jednostkę inercyjną.

Tablica 4.1: Typy wiadomości przekazywanych pomiędzy węzłami.



Rysunek 4.1: Komunikacja podstawowych pakietów systemu w trakcie testowania programu sterującego.

Ważną rolę odgrywa tutaj algorytm określania pozycji, bazujący na odometrii, jednostce inercyjnej i danych ze skanera laserowego. Odometria jest generowana za pomocą modelu kinematyki, sterowanego danymi z enkoderów modelu dynamiki. Sam model dynamiki sterowany jest pośrednio przez program, który generuje zadane prędkości liniowe i prędkość kątową robota. W uproszczeniu: program sterujący wysyła sterowanie do modelu, bazując na jego pozycji, określonej z danych generowanych przez modele czujników.

W każdym miejscu przepływu danych można zebrać i zwizualizować przesyłane wartości. Program sterujący może także korzystać ze skanerów laserowych w celu wykrycia przeszkody, nie tylko w celu określenia pozycji. Bardziej zaawansowany program sterujący mógłby generować zadane prędkości kół bezpośrednio, nie bazować na modelu kinematyki odwrotnej.

## 4.1 Zapis agentowy

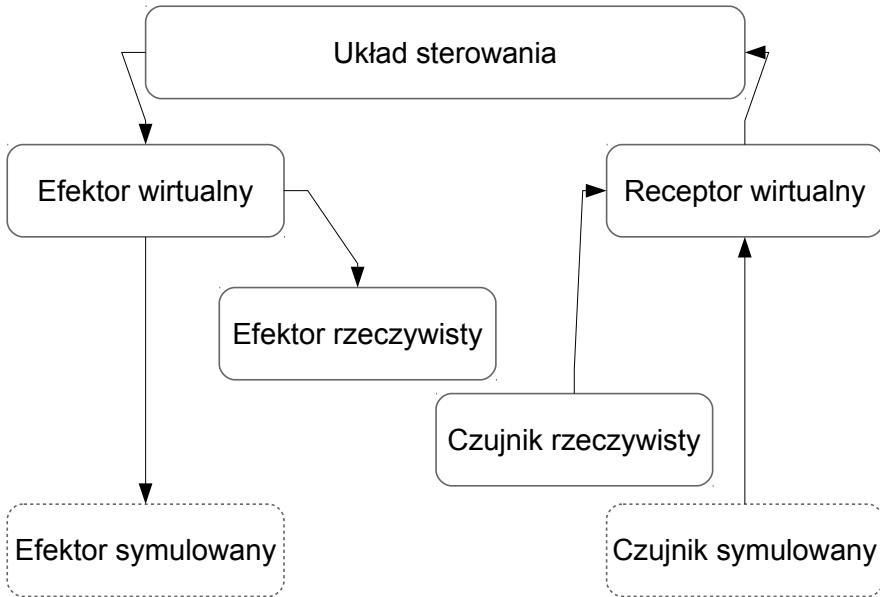
Aby zachować kompatybilność programu sterującego platformy mobilnej z jej modelem, należy stworzyć efektory i receptory wirtualne, do których program sterujący będzie wysyłał i z których będzie odbierał dane. Te wirtualne byty będą dalej przekazywać informacje zarówno do modeli, jak i do robota w taki sposób, że główny program sterujący nie będzie miał żadnej informacji o tym, do czego jest podłączony.

Można to przedstawić za pomocą zapisu agentowego (rysunek 4.2). Agent upustociowiony składa się z kilku modułów, komunikujących się ze sobą za pomocą różnych interfejsów.

Nadrzędnym modułem jest układ sterowania, który na podstawie odczytów z czujników generuje sterowanie dla efektorów. Ważne jest, aby komunikacja z rzeczywistymi urządzeniami była identyczna, jak z ich modelami, dzięki czemu taki system będzie przenośny i niezależny od implementacji modelu.

Efektor rzeczywisty, na przykład servomotor, jest sterowany za pomocą efektora wirtualnego, który zamienia wyjście układu sterowania na sygnały sterujące dla silnika napędowego. Przykładowo, zmienia odebraną liczbę, oznaczającą zadaną prędkość, na odpowiednie napięcie na wyjściu układu sterującego.

Zamodelowany efektor symulowany również przyjmuje te same sygnały do układu sterowania,



Rysunek 4.2: Struktura agenta upustaciowanego.

co efektor rzeczywisty, lecz nie zamienia ich na sygnały sterujące, a wywołuje odpowiednie funkcje maszyny symulacyjnej, nadające siły i prędkości obiektom w przestrzeni wirtualnej.

Receptor wirtualny pobiera surowe dane z czujnika, przekształca na odpowiedni format, usuwa błędy i szum tak, aby program sterujący mógł wykorzystać te dane w prosty sposób. Doskonałym przykładem jest tutaj urządzenie Kinect (widoczne na robocie Velma na rysunku 2.4), w którym to zachodzi odczytanie obrazu z kilku kamer. Następnie obraz przesyłany jest do komputera, w którym sterowniki interpretują dane, usuwając błędy, tworzą mapę głębokości, wykrywają szkielety i sylwetki osób. Te dane mogą być wykorzystane łatwo w grach i programach sterujących.

Modelowanie receptora, tak jak w przypadku efektora, polega na wygenerowaniu odpowiednich danych, używając odpowiednich funkcji w przestrzeni wirtualnej. Mogą one polegać na emitowaniu półprostych, symulujących laser, lub wręcz renderowaniu obiektów, aby uzyskać obraz z wirtualnej kamery. Receptor symulowany ma pełną wiedzę o symulowanym świecie, dokładne położenia i orientacje wszystkich obiektów, dane o kolizjach, itp. Pozwala to na łatwe symulowanie receptorów nie mogących mieć odwzorowania w rzeczywistości, co przydatne jest w pierwszych stadiach testowania i wyznaczaniu statystyk. Takim przykładem jest model czujnika dokładnego położenia, orientacji i prędkości w kartezjańskim układzie współrzędnych. Czujniki typu GPS, lub żyroskopy nie generują tak dokładnych pomiarów.

## 4.2 Model kinematyki

Kinematyka opisuje ruch obiektów bez rozważania sił powodujących ten ruch. Nie uwzględnia się przy opisie ruchu takich czynników jak masa, moment bezwładności, czy siły.

Ten program jest wtyczką simulatora Gazebo oraz modelem w przestrzeni wirtualnej. Dzięki temu pozwala na obliczanie i wizualizację pozycji.

Model kinematyki określa równania prostego zadania kinematyki. Rozwiążanie tego zadania polega na obliczeniu prędkości liniowej i kątowej bazy mobilnej na podstawie aktualnych prędkości kół. Symulator pozwala również na całkowanie tych prędkości, aby uzyskać aktualną pozycję platformy, z dokładnością do pozycji startowej.

Równania modelu kinematyki najwygodniej przedstawić w postaci macierzowej [2]. Dokładna pod-

stać wzoru zależy od kolejności numerowania kół i interpretacji wymiarów. Dla opisanego tutaj przypadku, (stałe zdefiniowane są w tabeli 2.1, numeracja kół jest pokazana na rysunku 2.6):

$$\begin{bmatrix} v_x \\ v_y \\ \omega_z \end{bmatrix} = \frac{r}{4} \begin{bmatrix} -1 & 1 & -1 & 1 \\ 1 & 1 & 1 & 1 \\ \frac{2}{a+b} & \frac{-2}{a+b} & \frac{-2}{a+b} & \frac{2}{a+b} \end{bmatrix} \begin{bmatrix} \omega_1 \\ \omega_2 \\ \omega_3 \\ \omega_4 \end{bmatrix} \quad (4.1)$$

Uzyskane wartości należy zastosować w funkcjach symulatora, aby nadać obiektom wirtualnym odpowiednie prędkości.

Sterowanie pozycją modelu kinematycznego odbywa się wyłącznie poprzez powyższy wzór, zatem w jego symulacji nie uczestniczy maszyna symulacyjna fizyki. Ten model nie reaguje na kolizje z innymi obiektami, nie reaguje na różnicę terenu i nie używa informacji o współczynnikach tarcia materiałów.

#### 4.2.1 Zachowanie

Platforma ignoruje inne obiekty znajdujące się na scenie. Po nadaniu stałych prędkości kół, następuje ruch zgodnie z rysunkiem 2.3.

Program sterujący co każdy krok symulacji (okres zależy od zasobów procesorowych komputera) zwraca aktualne położenie i orientację oraz prędkość liniową i kątową obiektu.

### 4.3 Model dynamiki

Maszyna do symulacji fizyki używa informacji o kształtach, masach i złączach pomiędzy ogniwami robota. Należy zatem stworzyć obiekt, złożony z modeli ogniw, i umieścić w symulatorze. Wtyczka Gazebo będzie nadawać prędkość kątową kołom i odczytywać dane z czujników. Potem należy nadać przegubom odpowiednie siły, aby otrzymać wyniki przybliżone do tego, jak zachowywałaby się rzeczywista baza mobilna.

Baza mobilna jest bryłą, na którą składają się następujące części składowe:

- Główna część korpusu.
- Ruchoma, mniejsza część korpusu, z przodu robota.
- 4 koła, 2 podłączone do głównej części korpusu, a 2 do przedniej.
- Po 12 rolek na każdym kole.
- Przegub obrotowy, łączący dwie części korpusu.
- 4 przeguby obrotowe z silnikami, łączące części bazy z kołami.
- 12 przegubów obrotowych na każdym kole, łączących koła z rolkami.
- Dwa skanery laserowe, przytwierdzone do głównej części korpusu.
- Mała jednostka inercyjna.

Jest to dość złożony obiekt do symulacji, dlatego należy dążyć do uproszczenia modelu w celu zmniejszenia ilości obliczeń symulatora. Istnieje wiele podejść do stworzenia odpowiedniego modelu, na przykład jak najdokładniejsze odwzorowanie budowy platformy za pomocą wzorów różniczkowych [6] [9].

Dodatkową funkcjonalnością jest brak nadawania prędkości kołom w przypadku odebrania pakietu zawierającego ciche nie-liczby (NaN). To jest charakterystyczne tylko dla modelu platformy (patrz sekcja 5.8.7).

W tym przypadku umieszczono model o odpowiednich parametrach w symulatorze, aby jego maszyna do symulacji fizyki obliczała prędkości i pozycje obiektów. Sam pakiet nie posiada modelu jednostki inercji i skanerów laserowych, a importuje je z innych pakietów.

### 4.3.1 Zachowanie

Model reaguje na siły przyłożone do jego ogniw, porusza się, reagując na otoczenie. Bierze udział w kolizjach, nadaje prędkości innym obiektom. Współczynniki tarcia podłożą i kół mają znaczenie w symulacji. Powstają niedokładności wyznaczania pozycji, spowodowane dużą ilością zmiennych, uczestniczących w symulacji i precyzją symulatora.

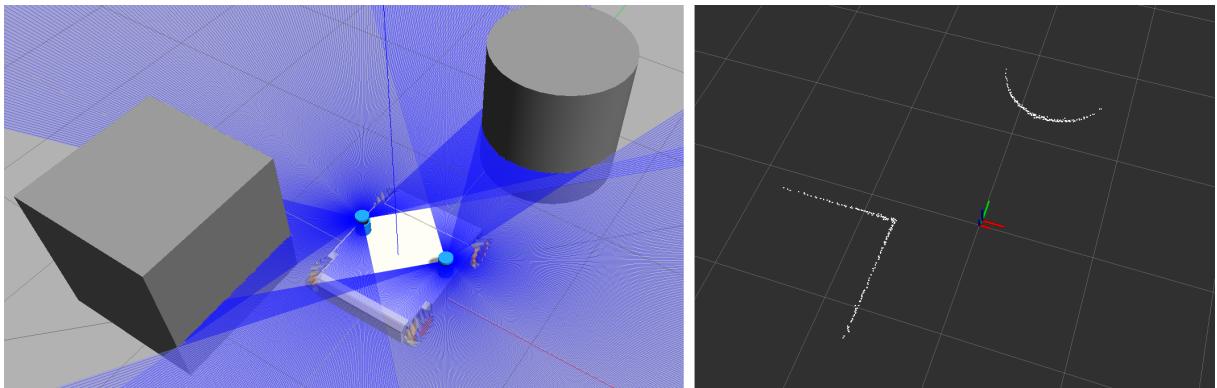
Zachowanie modelu silnie zależy od jego parametrów, a zwłaszcza od mas i momentów obrotowych ogniw.

## 4.4 Model skanera laserowego

Simulator generuje odpowiednie dane, emitując promienie w przestrzeni wirtualnej. Następnie maszyna symulacyjna fizyki oblicza kolizje promieni z obiektami na scenie, biorąc pod uwagę także ich albedo, sprecyzowane w parametrach modelu. Jest to operacja bardzo kosztowna obliczeniowo.

Do położen punków dodawany szum o rozkładzie normalnym, aby symulować błędy pomiarowe, powstałe przy odczycie odbicia lasera. Model symuluje zarówno pomiary odległości od obiektu, jak i ich intensywność.

Odczyt z rzeczywistego skanera pokazuje, że sztucznie wygenerowane dane są podobne do danych zebranych przez czujnik.



Rysunek 4.3: Zrzut ekranu platformy z Gazebo i wygenerowane dane, obserwowane w Rviz.

Ten pakiet składa się z wtyczki obsługującej model skanera i pliku opisującego parametry samego skanera, jak i również wygląd fizyczny, kształt, masę, moment obrotowy obudowy. Dane z tego pakietu importowane są przez pakiet modelu dynamiki i umieszczane w odpowiednim miejscu modelu robota z odpowiednim przegubem.

## 4.5 Model jednostki inercyjnej

Rzeczywisty czujnik obarczony jest bardzo dużymi błędami pomiarowymi. Model symuluje przyspieszenie oraz prędkość kątową robota. Dodaje sztuczny szum do wygenerowanych danych, aby przybliżyć zachowanie modelu do jednostki inercyjnej.

Niestety, ponieważ ten czujnik jest bardzo czuły na drgania robota, nie wszystkie jego cechy da się poprawnie zasymulować. Dodatkowo, wygenerowane dane w dużym stopniu zależą od momentów bezwładności ogniw robota.

Podobnie, jak pakiet modelu skanera laserowego, jest on zaimplementowany jako model umieszczany w symulacji i wtyczka, importowane przez model dynamiki.

## 4.6 Model kinematyki odwrotnej

Model kinematyki odwrotnej jest przeciwnieństwem modelu kinematyki, opisanego w sekcji 4.2. Ten program przyjmuje prędkość liniową i kątową platformy, następnie oblicza prędkości kątowe kół, wymagane aby nadać platformie zadaną prędkość.

Ten model działa bez symulatora, jako węzeł ROS, filtrujący wiadomości i nie posiada możliwości całkowania wygenerowanych prędkości. Całkowanie prędkości kół i tak nie ma większego sensu, ponieważ pozwoliłoby to jedynie obliczyć ich aktualny kąt obrotu. Z wyjątkiem porównania tych danych z danymi z enkoderów, nie ma to innego zastosowania.

Wzory kinematyki odwrotnej mogą być przedstawione w postaci macierzowej [2] (stałe zdefiniowane są w tabeli 2.1, numeracja kół jest pokazana na rysunku 2.6).

$$\begin{bmatrix} \omega_1 \\ \omega_2 \\ \omega_3 \\ \omega_4 \end{bmatrix} = \frac{1}{r} \begin{bmatrix} 1 & -1 & \frac{a+b}{2} \\ 1 & 1 & -\frac{a+b}{2} \\ 1 & -1 & -\frac{a+b}{2} \\ 1 & 1 & \frac{a+b}{2} \end{bmatrix} \begin{bmatrix} v_y \\ v_x \\ \omega_z \end{bmatrix} \quad (4.2)$$

Dodatkowo, program pozwala na obrót wektora wejściowej prędkości o kąt prosty lub półpełny. Jest to spowodowane tym, że różne pakiety i różne modele robotów przyjmują różną orientację wyjściową robota. Czasami przód modelu skierowany jest w dodatnią stronę osi X, a czasami Y. W związku z tym, ta funkcjonalność jest w stanie przekonwertować dane wejściowe dla innego robota tak, aby mogły być użyte do sterowania modelem platformy.

## 4.7 Manualne sterowanie

To zaawansowany program do manualnego generowania zadanych prędkości kół lub prędkości liniowej i kątowej platformy. Zaimplementowany jako pakiet ROS. Ponieważ jest niezależny od reszty systemu, może być użyty do sterowania rzeczywistym robotem. Pozwala także na wyświetlanie aktualnych prędkości kół, generowanych przez enkodery.

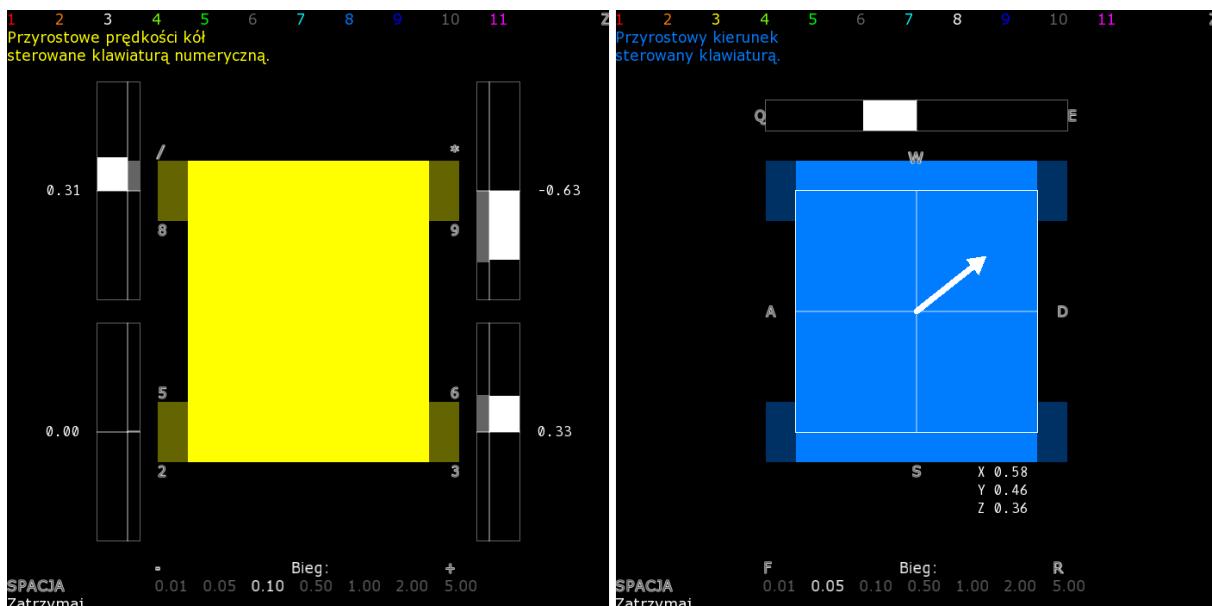
Sterowanie można nadać poprzez klawiaturę, kontroler do gier lub myszkę. Program otwiera graficzne okno, w którym wyświetla aktualne dane i wskaźniki prędkości.

### 4.7.1 Tryby działania

Program posiada 11 trybów działania, w których generuje różne wiadomości w różny sposób. Globalny mnożnik wyjścia pozwala na łatwe ograniczenie generowanych danych i ustawienia dokładności. Naciśnięcie klawisza spacji awaryjnie zeruje wszystkie wyjścia.

1. Za pomocą ósmu klawiszy klawiatury numerycznej, można nadać platformie określone prędkości kół. W tym trybie koło może stać w miejscu, albo obracać się z odpowiednią prędkością kątową. Takie sterowanie powoduje poślizgi platformy. W trakcie braku aktywności użytkownika, generowane jest zerowe sterowanie.
2. Podobnie do poprzedniego trybu, lecz przy braku naciśnięcia klawisza, generuje cichą nie-liczbę, aby zachować aktualną prędkość kół modelu. Ta dodatkowa funkcjonalność opisana jest szerzej w sekcji 5.8.7.
3. Naciśnięcie klawisza płynnie zwiększa lub zmniejsza prędkość koła. W trakcie braku aktywności użytkownika, generowane jest stałe sterowanie, bazujące na aktualnym ustawieniu programu.
4. Podobnie, co w poprzednim trybie, lecz pozwala na schodkowe ustawienie prędkości kół co 0,1 rad/s (pomnożone przez mnożnik wyjścia). Dzięki temu możliwe jest w miarę dokładne powtórzenie manualnych testów platformy.

5. Poprzedni tryb, lecz przy ustawieniu prędkości zerowej, generuje nie-liczbę.
6. Sterowanie prędkościami kół za pomocą gałek kontrolera. Większość kontrolerów posiada dwa, dwuosiowe joysticki, co daje cztery osie, zmieniające się w zakresie  $(-1; 1)$ . Można za ich pomocą bezpośrednio ustawiać prędkości kół. Ten tryb jest nieintuicyjny w działaniu.
7. Ten tryb generuje zadaną prędkość liniową i kątową, a nie prędkości kątowe kół, jak poprzednie tryby. Za pomocą klaviatury można nadać platformie jeden z ośmiu kierunków poruszania się i jeden z dwóch kierunków obrotu wokół osi Z. Ta metoda sterowania powoduje skoki prędkości i poślizgi. Przypomina sterowanie pojazdami w grach komputerowych. Puszczenie klawiszy powoduje zatrzymanie się platformy.
8. Podobny tryb do poprzedniego, ale naciśnięcie klawisza płynnie dodaje wartość do prędkości liniowej i kątowej platformy. Brak aktywności użytkownika powoduje, że generowane jest stałe sterowanie.
9. Schodkowe sterowanie prędkością platformy w krokach co  $0,1$  m/s. Pozwala na ustawienie prędkości i obrotu platformy z zadaną dokładnością i dokładniejsze manualne przeprowadzanie testów.
10. Sterowanie prędkością liniową i kątową platformy za pomocą kontrolera. Trzy osie są używane, dwie do nadania prędkości liniowej, jedna do nadania prędkości kątowej. Jest to prawdopodobnie najczęstszy sposób kontrolowania robotów wielokierunkowych za pomocą kontrolera. Sposób bardzo intuicyjny i używany także przez inne pakiety do manualnego sterowania robotami na kołach Mecanum.
11. Sterowanie za pomocą myszki, najdokładniejsze sterowanie zadaną prędkością liniową i kątową robota. Kursor myszy wskazuje końcówkę strzałki reprezentującej kierunek ruchu robota, za pomocą kółka można ustawić zadaną prędkość kątową robota wokół osi Z. Ponieważ większość myszek ma skokowe obroty kółek, wprowadza to nieznaczne poślizgi w nadawanej prędkości kątowej. Można także modyfikować tę wartość klaviaturą w płynnym trybie przyrostowym.



Rysunek 4.4: Zrzuty ekranu dwóch trybów działania programu.

Interfejs graficzny (patrz rysunek 4.4) składa się z listy trybów, wyświetlanych na górze, i nazwy aktualnego trybu. Wyszarzone tryby nie mogą być aktywowane, w tym przypadku z powodu braku podłączenia kontrolera.

Na lewym zrzucie widać zarys platformy i białe wskaźniki aktualnych prędkości kół, wraz ze wspólnikiem wypełnienia. Obok nich znajdują się szare wskaźniki prędkości kątowych kół, zwrócone przez modele enkoderów. Małe, szare znaki to nazwy klawiszy, używanych w tym trybie do modyfikowania prędkości.

Na prawym rysunku jest tryb generowania kierunku i obrotu. Strzałka wskazuje wektor prędkości liniowej, a górny pasek prędkość kątową platformy.

Na dole jest lista „biegów” urządzenia, są to zwyczajne mnożniki wyjścia w celu wygodnego przedstawiania dokładności z jaką platforma powinna się poruszać.

Wszystkie dane są w jednostkach SI, tzn, efektywna prędkość koła będzie się równać liczbę podaną przy kole, pomnożonej przez aktualny bieg. Dla obrotów to są rad/s, dla prędkości to m/s.

Jeszcze jeden parametr pozwala na ustawienie częstotliwości z jaką wysyłane mają być wiadomości przez strumień komunikacyjny.

## 4.8 Generator sterowania

Podstawą przeprowadzania testów modelu jest powtarzalność eksperymentów oraz dokładność nadanego sterowania. Potrzeba zatem jest sposobu na automatyczne wygenerowanie przesyłanych wiadomości z określonymi danymi.

Ten pakiet ROS generuje powtarzalne sterowanie, bazując na wczytanym pliku tekstowym i podanej częstotliwości wysyłania pakietów. Wyjściem są zadane prędkości liniowe i prędkość kątowa bazy.

Dane zapisane są w formie wierszy, w którym każdy określa czas i dane, które program ma nadawać. Po zakończeniu wykonywania instrukcji, węzeł generuje wiadomość o zerowych parametrach w celu zatrzymania bazy.

## 4.9 Wyłuskanie podwiadomości

Każda wiadomość przekazywana pomiędzy węzłami jest zwykle zagnieżdżoną strukturą. Ten pakiet ROS działa jak filtr dla strumienia wiadomości.

Czasami może zdarzyć się, że jakiś węzeł potrzebuje jedynie wewnętrznej podstruktury wiadomości. Mógłby subskrybować całą wiadomość, lecz to powodowałoby potrzebę przesyłania dodatkowych, nieużywanych danych oraz nie pozwoliłoby na zachowanie niezależności pakietu od innych.

Takie zjawisko występuje przy przekazywaniu informacji o prędkości kątowej i kącie obrotu kół, generowanej przez model czujnika enkoderów, do programu manualnego sterowania lub do modelu kinematyki.

ROS nie pozwala na automatyczne przesłanie tylko części pakietu pomiędzy węzłami, dlatego powstał ten program.

## 4.10 Podłoże o zmiennym współczynniku tarcia

Symulacja nie składa się jedynie z robota i czujnika, ale także z podłożem, na którym musi się poruszać. Ponieważ podłoże również wpływa na symulację, powinien istnieć sposób na ustawienie jego współczynnika tarcia.

Ten pakiet jest modelem ładowanym do symulatora Gazebo, przyjmuje on asynchroniczne wywołania, nadające podłożu odpowiednie tarcie. W ten sposób można testować zachowanie się modelu w różnych przypadkach testowych.

## 4.11 Algorytm usuwania szumu z danych jednostki inercyjnej

Jak wcześniej wspomniano, jednostka inercyjna i jej model zwracają bardzo duże błędy pomiarowe.

Ten filtr uśrednia dane w prosty sposób, licząc średnią z określonej ilości poprzednich pomiarów. Działa to bardzo dobrze przy uśrednianiu szumu przy zerowym przyspieszeniu lub w trakcie ruchu robota z prędkością jednostajną. Nie sprawdza się jednak przy odczycie faktycznego przyspieszenia, gdyż może generować zaniżone wartości.

W przyszłości zastosować trzeba będzie bardziej zaawansowany algorytm uśredniania odczytów. Może on być również testowany na tym modelu.

Stworzenie tego programu pozwoliło zbadać, czy model jednostki inercyjnej oraz jednostka inercyjna platformy reagują na ruch w odpowiednim kierunku, gdyż pomiary są obarczone tak dużymi błędami, że wizualizacja odczytów na wykresie nie daje gwarancji upewnienia się o działaniu modelu.

## 4.12 Obserwator symulacji

Wtyczka uruchamiana w symulatorze. Oblicza i zwraca statystyki międzymodelowe w przestrzeni symulacji, takie jak odległość i kąt. Pozwala zbadać, jak model dynamiki zachowuje się w stosunku do modelu kinematyki, to znaczy, czy pozycja, obliczona przez maszynę symulacyjną fizyki jest zbliżona do pozycji obliczonej równaniami kinematycznymi po scałkowaniu.

## 4.13 Scena z symulacją

Symulator Gazebo przy uruchomieniu ładuje plik opisujący symulację. Ten pakiet nie jest programem wykonywalnym, a zestawem kilku plików czytanych przez Gazebo. W tych plikach zawierają się także ustawienia symulacji, jak przyspieszenie grawitacyjne, typ maszyny symulacyjnej fizyki ze współczynnikiem, czy ustawienia wirtualnej atmosfery.

Zapisane są tutaj nazwy modeli importowanych z innych pakietów.

## 4.14 Rozdzielacz wiadomości

Jeśli dwóm węzłom nadać te same nazwy interfejsów strumienia wiadomości, to ROS będzie przekazywał pomiędzy nimi informacje. To jednak nie zawsze jest możliwe, aby mieć całkowitą kontrolę nad nazwami interfejsów wszystkich węzłów. Dlatego też, potrzebny jest program do przekazywania i ewentualnego rozdzielania wiadomości dla różnych odbiorników.

Ten program wykonywalny pobiera i generuje wiadomości zawierające zadane prędkości kół. Pozwala to na sterowanie kilkoma robotami o identycznym interfejsie ze wspólnego źródła. W szczególności przydaje się to przy rozdzielaniu wartości prędkości kół dla modelu platformy dynamicznej i kinematycznej.

Ten pakiet może być zastąpiony przez kilkukrotne uruchomienie wbudowanego w ROS narzędzia relay, które przekazuje pakiet z jednego strumienia komunikacyjnego do innego.

## 4.15 Prosty program sterujący

Jest to uproszczona wersja programu, który docelowo ma być tworzony na podstawie budowanego systemu modeli.

Program periodycznie wysyła dane o zadanej prędkości, zależnej od aktualnego stanu. W zależności od danych z czujników laserowych, program zmienia swój stan i obraca kierunek obrotu o  $90^\circ$ . Ten sterownik dla uproszczenia nie generuje poleceń obrotu kątowego, sterowany obiekt powinien zachować swoją orientację.

Prosty algorytm programu gwarantuje omijanie przeszkód, jednak nie bierze pod uwagę celu jazdy. To znaczy, że platforma będzie poruszać się od przeszkody do przeszkody w losowy sposób.

## 4.16 Struktury pakietów wiadomości

Ten pakiet nie jest plikiem wykonywalnym, a definicjami struktur danych, używanymi przez wiadomości ROSa w projekcie, jeśli standard nie obejmuje potrzebnego typu wiadomości.

Dodatkowo zdefiniowane typy wiadomości to:

- Dane prędkości kątowej i kącie obrotu kół, zwarcane przez model enkoderów.
- Dane o względnym położeniu i orientacji obiektów na scenie.
- Zadane prędkości kątowe kół.
- Asynchroniczne wywołanie do ustawienia mas i momentów bezwładności ogniw modelu.
- Asynchroniczne wywołanie do ustawienia współczynników tarcia obiektu.

## 4.17 Zewnętrzne pakiety ROSa

Istnieje kilka tysięcy różnych pakietów i programów, tworzonych przez społeczność ROSa. Te zostały użyte w projekcie.

### 4.17.1 Rysownik wykresów

Pakiet `rqt-multiplot` jest wtyczką do większego programu `rqt`. Pozwala na generowanie dwuwykresów, bazując na dwóch dowolnych wartościach z odbieranych pakietów, łącznie z czasem nadania. Pozwala porównać różne wykresy na jednym układzie.

W szczególności, przy ustawieniach wyświetlania składowych położenia Y względem X, pobranych z pakietu pozycji, pozwala narysować trajektorię ruchu platform.

### 4.17.2 Wizualizer pomiarów

Oryginalnie napisany dla robota o tej samej nazwie, `rviz` prezentuje trójwymiarową przestrzeń, w której można wyświetlać dane odebrane z innych węzłów.

Pozwala to na przykład umieścić znacznik reprezentujący pozycję platformy i chmury punktów, odebranych z czujników laserowych. Jest lżejszy na zasobach w działaniu niż Gazebo i pokazuje tylko informacje z odebranych danych, a nie całe środowisko symulacji. Nie posiada własnego simulatora fizyki, nie generuje żadnych danych samodzielnie.

### 4.17.3 Algorytm określania lokalizacji

Pakiet `laser_scan_matcher` pozwala na określenie pozycji robota, bazując na danych ze skanera laserowego oraz opcjonalnie jednostki inercyjnej i enkoderów w celu minimalizacji błędów. Bazuje na porównywaniu kolejnych odczytów chmury punktów pomiarowych. Istnieją bardziej zaawansowane algorytmy określania pozycji, na przykład bazujące na modelu mapy, lecz ten pakiet jest prosty w użyciu i nie wymaga dodatkowych danych, ani kalibracji.

# Rozdział 5

## Implementacja

W tym rozdziale opisane są szczegóły techniczne zastosowanych rozwiązań.

### 5.1 Istniejące implementacje

Istnieją także inne modele jeżdżących robotów na kołach szwedzkich. Można z nich brać przykład i sugerować się źródłami kodu i budową modeli.

Kuka Youbot jest popularnym robotem wielokierunkowym. Jego modele są domyślnie dostępne w różnych symulatorach, między innymi w Gazebo i V-Repie. Tylko w przypadku V-Rep, istnieje wstępny sterownik do którego da się wysyłać odpowiednie wartości zadanej prędkości liniowej i kątowej, a on nadaje odpowiednie prędkości kołom, aby odpowiednio poruszać modelem (zawiera model kinematyki). Wersja dla Gazebo jest statycznym obiektem z błędnie ustanowionymi przegubami, jego efektry nie są zaimplementowane. Dodatkowo, V-Rep posiada modele dwóch innych pojazdów o napędach kół Mecanum i czujnikach laserowych.

Ze względu na niezwykle zaawansowany obiekt kół i kształt rolek, ważne jest aby uprościć model poprzez zamianę niektórych ogniw i dodanie sztucznych więzów. Model zbliżony do budowy platformy może być zbyt skomplikowany, aby maszyny symulacji mogły go obliczać w czasie rzeczywistym. Proponowane uproszczenia modeli opisane są w sekcji 4.3.

### 5.2 Model 3D

Model 3D bazy mobilnej, opisany równaniami matematycznymi, powinien mieć zachowanie zbliżone do oryginału najbardziej jak to tylko możliwe. Musi uwzględniać masy i momenty bezwładności składowych ogniw, a także wszystkie tarcia. Model obejmuje więzy na ruchome ogniwach, takie jak koła i rolki, aby umożliwić symulację przegubów.

Model składa się z ogniw, odwzorowujących rzeczywiste części składowe bazy mobilnej. Posiadają one takie cechy, jak:

- Pozycja w modelu.
- Masa.
- Moment bezwładności.
- Kształt fizyczny.
- Materiał fizyczny.
- Wygląd.

Dodatkowo, należy uwzględnić wszystkie więzy w postaci symulowanych przegubów. W przypadku tej bazy, istnieje typ przegubów o jednym stopniu swobody, używany przy połączeniu przedniej i tylnej części platformy oraz jako piasty kół i rolek. Przeguby bez stopni swobody używane są do trwałego połączenia czujników z platformą i transportowanym robotem dwuramiennym. Więzy mogą oddziaływać momentem siły na elementy do których są podłączone, symulując silniki.

Ogniwa i symulowane przeguby oddziałują bezpośrednio z maszyną do symulacji fizycznej. To kształt, masy i momenty bezwładności brył są argumentami funkcji liczących. Maszyna symulacyjna oblicza odpowiednie prędkości i nadaje je podanym obiektom.

Do modelu doczepia się wirtualne czujniki, generujące odpowiednie dane na podstawie symulacji i rozkładu losowego. Nie są to pełne dane o stanie modelu, jakie posiada maszyna do symulacji, gdyż czujniki fizyczne również nigdy nie mają pełnej informacji o stanie urządzenia. Należy dodać losowy szum i błędy, aby przybliżyć ich zachowanie do rzeczywistych czujników.

Dla odpowiedniej wizualizacji symulacji wykorzystano istniejący model CAD do stworzenia siatki trójwymiarowej i nadania symulowanemu obiekowi wyglądu zbliżonego do robota.

## 5.3 Ogólne typy pakietów

Pakiety można podzielić na kilka typów, w zależności od sposobu implementacji. Wszystkie programy napisane są w języku C++ i korzystają z nowoczesnych funkcji języka, jak referencje i sprytne wskaźniki.

### 5.3.1 Program wykonywalny w ROS

Jest to prosty program, który korzysta z kilku funkcji ROSa, między innymi:

- Inicjalizacja węzła.
- Subskrypcja strumienia wiadomości.
- Publikowanie danych do strumienia wiadomości.
- Zawieszenie procesu.
- Generowanie wpisów dziennika, jak informacje pomocnicze, błędy i ostrzeżenia.

#### Inicjalizacja

Na początku głównej funkcji programu `int main(int argc, char** argv)`, należy podać argumenty wywołania programu do funkcji inicjalizującej ROSa. To, ponieważ program może być uruchomiony z dodatkowymi parametrami, które dotyczą działania ROSa, a nie samego działania programu. Ta funkcja odczytuje odpowiednie parametry i modyfikuje zmienne w razie problemów, usuwając niektóre argumenty, aby reszta kodu parsowała dane przeznaczone dla programu.

```
void ros::init (int& argc,
                char** argv,
                const std::string& name,
                uint32_t options = 0)
```

Ta funkcja przyjmuje także nazwę nowego węzła, jaka zostanie zgłoszona do demona ROS oraz flagi opisujące inicjalizację. Nazwa musi być unikalna dla całego systemu, nie mogą działać dwa węzły o tej samej nazwie. Flagi opisują, czy program powinien mieć zawieszone wypisywanie logów, czy nazwa powinna posiadać losowy dopisek (co pozwala na uruchomienie wielu instancji tego samego programu) oraz czy program powinien się zakończyć na otrzymanie sygnału systemowego SIGINT, czy w programie zdefiniowana jest funkcja obsługująca odebranie tego sygnału.

## Nadawanie wiadomości do strumienia

Kolejnym krokiem jest otwarcie komunikacji poprzez strumienie wiadomości z innymi węzłami. W tym celu tworzy się obiekt węzła, obiekt klasy `ros::NodeHandle`, którego konstruktor nie przyjmuje argumentów. Ten obiekt służy do komunikacji ze środowiskiem ROSa, pozwala na tworzenie publikatorów i subskrybentów danych. Programista musi jedynie się upewnić, aby zachować referencję do obiektu przez cały czas życia programu, gdyż destrukcja obiektu odłącza program od demona ROSa.

```
template<class M>
Publisher ros::NodeHandle::advertise (const std::string& topic,
                                         uint32_t queue_size,
                                         bool latch = false)
```

Korzystając z obiektu węzła można zarejestrować, za pomocą powyższej funkcji, nowego publikatora do strumienia wiadomości. Jest to metoda szablonowa, to oznacza że przy wywołaniu należy podać typ wiadomości, jaką węzeł powinien nadawać.

Za pomocą operatora negacji można sprawdzić, czy stworzenie obiektu przebiegło pomyślnie, a jeśli nie, poinformować użytkownika i zakończyć program.

Ponieważ kod jest komplikowany, programy w C++, w przeciwieństwie do programów w Pythonie, nie są w stanie dynamicznie odbierać i wysyłać wiadomości różnych typów. To znaczy, że każdy węzeł zaimplementowany w tym języku będzie działał jedynie z określonymi wcześniej danymi.

Następne argumenty to nazwa strumienia, wielkość bufora wysłanych wiadomości oraz opcjonalny argument, decydujący o tym czy ostatnia nadana wiadomość powinna być buforowana. Metoda zwraca obiekt publikatora, który to może być użyty do nadawania wiadomości.

```
template<class M>
void ros::Publisher::publish(const M& message) const
```

Wysłanie wiadomości to stworzenie nowego obiektu klasy wiadomości i podanie go do powyższej metody publikatora. To także jest funkcja szablonowa, korzystająca z typu podanego przy tworzeniu obiektu.

## Odbiór wiadomości ze strumienia

W nieco inny sposób należy stworzyć subskrybenta strumienia wiadomości. Przy każdej odebranej wiadomości, wywoływana jest podana w argumencie funkcja. Można to porównać do działania przerwań systemowych.

```
template<class M>
Subscriber ros::NodeHandle::subscribe(const std::string& topic,
                                         uint32_t queue_size,
                                         void(*)(M) handler,
                                         const TransportHints& transport_hints = TransportHints())
```

Podana metoda, podobnie, jak w nadajniku, jest metodą obiektu węzła. Przyjmuje nazwę strumienia wiadomości, wielkość bufora, wskaźnik na funkcję obsługi oraz opcjonalnie informacje dotyczące połączenia.

Metoda posiada wiele różnych odmian, w zależności od sposobu podania funkcji obsługującej. Zamiast funkcji, może być to na przykład funktor. Wywołanie tworzy nowy obiekt subskrybenta, na którym przy normalnym działaniu nie trzeba wywoływać żadnych metod. Jednakże, nadal należy posiadać referencję do niego, gdyż destrukcja obiektu automatycznie zamyka połączenie.

Jeśli strumień wiadomości o takiej nazwie nie istnieje, metoda nie zwraca błędu. Tak jest, ponieważ nadal może w przyszłości pojawić się nadajnik o odpowiedniej nazwie. Dodatkowo, to utrudniałoby jednoczesne uruchamianie kilku węzłów zależnych od siebie, gdyż należałoby się upewnić, że nadajniki uruchomią się pierwsze. A także nie pozwalałoby na połączenie strumieni wiadomości w cykle.

## Zapisywanie danych do dziennika

Pomimo, że program może zwracać dane na standardowe wyjście i błąd, użycie funkcji ROSa pozwala na selektywne ustawienie minimalnej ważności zwracanych powiadomień. ROS także koloruje tekst i dodaje przedrostek z czasem nadania powiadomienia.

```
ROS_INFO(...)  
ROS_INFO_STREAM(...)
```

Rysunek 5.1: Makra ROSa, wypisujące powiadomienia.

Używa się makr z rysunku 5.1, wszystkie makra są podane w dwóch typach, pierwsze pozwala na użycie argumentów w sposób charakterystyczny dla systemowej funkcji printf, a drugie przez strumienie języka C++.

Tekst INFO może być zastąpiony przez jeden z pięciu priorytetów powiadomienia: DEBUG, INFO, WARN, ERROR i FATAL.

## Zawieszenie procesu

Wywołanie `ros::spin()` powoduje zawieszenie się głównego wątku programu. W ten sposób program będzie działał jedynie na odbiór wiadomości. Istnieją funkcje systemowe, które pozwalają na dokonanie tego samego, lecz API ROSa jest prostsze w użyciu, a także może wykonywać dodatkowe akcje związane z obsługą demona.

## Przykład

Przykładowe użycie powyższych metod przy implementacji prostego filtra wiadomości. Ten kod tworzy publikatora i subskrybenta wiadomości, zawierającej prędkość liniową i kątową, a następnie na odbiór każdej wiadomości, przekazuje dalej jedynie prędkość liniową w płaszczyźnie platformy i prędkość kątową wokół osi Z. Dzięki temu robot nie otrzyma sterowania w niemożliwym do poruszania się kierunku. Ten kod nie jest użyty w żadnym z pakietów, model ignoruje dane z nieistotnych pól.

Następnie wątek główny zostaje uśpiony, gdyż program działa jedynie w systemie akcja-reakcja. Wysyła wiadomość jedynie po otrzymaniu innej.

```
#include <iostream>  
#include <string>  
//nagłówek głównego API ROSa  
#include <ros/ros.h>  
//nagłówek dla funkcji piszących do dziennika  
#include <ros/console.h>  
//nagłówek z typem wiadomości  
#include <geometry_msgs/Twist.h>  
  
//obiekt publikatora  
ros::Publisher publisher;  
  
//funkcja obsługi odbioru wiadomości  
void callbackFun(const geometry_msgs::Twist::ConstPtr& msg)  
{  
    //nowy obiekt typu wiadomość  
    //konstruktor zeruje wszystkie pola  
    geometry_msgs::Twist newTwist;  
    newTwist.linear.x = msg->linear.x;
```

```

        newTwist.linear.y = msg->linear.y;
        newTwist.angular.z = msg->angular.z;
        //wysyłanie wiadomości
        publisher.publish(newTwist);
    }

int main(int argc, char** argv)
{
    //inicjalizacja
    ros::init(argc, argv, "filtrownica");

    //nazwa strumienia wejściowego
    std::string inTopic;
    //nazwa strumienia wyjściowego
    std::string outTopic;

    //...parsowanie argumentów argc i argv
    //w celu odczytania powyższych zmiennych

    //obiekt węzła
    ros::NodeHandle handle;

    //stworzenie publikatora
    publisher = handle.advertise<geometry_msgs::Twist>(outTopic, 1000);
    if(!publisher)
    {
        ROS_FATAL_STREAM("Nie udało się stworzyć publikatora " << outTopic);
        return -1;
    }

    //stworzenie subskrybenta
    ros::Subscriber subscriber;
    subscriber = handle.subscribe<geometry_msgs::Twist>(inTopic,
        1000, callbackFun);
    if(!subscriber)
    {
        ROS_FATAL_STREAM("Nie udało się stworzyć subskrybenta " << inTopic);
        return -1;
    }

    //zawieszenie wykonywania głównego wątku
    //ROS obsługuje sygnał SIGINT i odblokuje program
    ros::spin();
    return 0;
}

```

### 5.3.2 Wtyczka Gazebo

Wtyczka do symulatora Gazebo jest klasą, w przypadku modeli dziedziczącą po ModelPlugin, w przypadku modelu czujnika po SensorPlugin lub dla wtyczki obsługującej symulację — WorldPlugin. Jest komplikowana do postaci biblioteki i ładowana dynamicznie przy uruchomieniu symulatora. Gazebo, prócz wywołania ewentualnego konstruktora, wywołuje metodę wirtualną, zależną od typu wtyczki.

```

void Load(physics::ModelPtr parent, sdf::ElementPtr sdf)
void Load(sensors::SensorPtr parent, sdf::ElementPtr sdf)
void Load(physics::WorldPtr world, sdf::ElementPtr sdf)

```

która to posiada argument typu sprytny wskaźnik na obiekt, który obsługuje ta wtyczka oraz wskaźnik na element pliku SDF, odpowiedzialny za załadowanie programu do symulatora. Przez drugi argument można przekazać dodatkowe dane do programu. Jeśli plik ma zdefiniowane dodatkowe elementy w tym miejscu, można ich wartości odczytać.

Jeśli chodzi o komunikację z ROSem, to może ona być zrealizowana identycznie, jak w sekcji 5.3.1, z tą różnicą, że nie jest wymagana inicjalizacja, gdyż logicznie cały symulator działa jak jeden węzeł. Również, jeśli chodzi o referencje do obiektów, to należy je zachować po opuszczeniu metody Load, to znaczy że obiekty węzła, nadawców i odbiorców powinny być przechowywane jako pola stworzonej klasy, inaczej węzeł odłączy się od demona i strumieni wiadomości zaraz po inicjalizacji.

### Wywołanie na kroki symulacji

Główną mechaniką używaną u wtyczek jest periodyczne wywołanie kodu co każdy krok symulacji. Pozwala to na przykład nadawać aktualną pozycję modelu. Gazebo obsługuje kilka zdarzeń, wywoływanych na różne części symulacji. Można się podłączyć do odpowiedniego zdarzenia za pomocą funkторa w poniższej metodzie. Ta metoda jest implementowana osobno dla każdego typu zdarzenia.

```
ConnectionPtr Connect(const boost::function< T > & _subscriber)
```

Przykładowe podłączenie do własnej zdefiniowanej metody OnUpdate do zdarzenia WorldUpdateBegin może wyglądać w ten sposób:

```
event::ConnectionPtr connection = event::Events::ConnectWorldUpdateBegin(
    std::bind(&MyModelDriver::OnUpdate, this));
```

Należy zatrzymać referencję do zwróconego obiektu.

## 5.4 Mechanika przekształceń układów współrzędnych

Komunikacja poprzez pakiety wiadomości nie jest jedynym sposobem na przekazywanie informacji w środowisku ROS. Istnieje także mechanika macierzy przekształceń jednorodnych TF2. Jest to idea podobna do niezaimplementowanej funkcjonalności Gazebo, ale nie jest automatyczna i nie ogranicza się tylko do jednego programu.

Macierz przekształcenia jest informacją o aktualnym położeniu i orientacji jakiegoś obiektu względem innego. Polega na wysłaniu pakietu typu geometry\_msgs/TransformStamped prosto do demona ROS, tworząc odpowiedni obiekt publikatora. Pakiet zawiera:

- Nagłówek z czasem nadania wiadomości i identyfikatorem oraz nazwą przekształcenia względem którego podane są poniższe dane.
- Nazwa nowej pozycji, jaka powstanie po zastosowaniu podanego przekształcenia do tego określonego w nagłówku.
- Lokalne położenie.
- Lokalna orientacja.

Demon ROSa następnie zbiera wszystkie dane ze wszystkich nadających węzłów i oblicza hierarchię przekształceń obiektów. Zwraca te dane na zapytania od innych węzłów.

Przykładowo, gdyby symulacja robota nie odbywały się w przestrzeni wirtualnej, w maszyngie symulacyjnej fizyki, informacja o dokładnym położeniu obiektu składowego w lokalnym układzie

Punkt wzgledny	Identyfikator
Stałý środek mapy	map
Środek modelu dynamiki	omnivelmana
Środek modelu kinematyki	pseudovelma
Pozycja lusterka prawego skanera laserowego	monokl_r_heart
Pozycja lusterka lewego skanera laserowego	monokl_l_heart

Tablica 5.1: Nazwy identyfikatorów przekształceń, używanych w symulatorze.

Nazwa	Punkt wzgledny	Identyfikator
Położenie i orientacja modelu dynamiki	map	omnivelmana
Położenie i orientacja modelu kinematyki	map	pseudovelma
Położenie i orientacja prawego skanera	map	monokl_r_heart
Położenie i orientacja lewego skanera	map	monokl_l_heart

Tablica 5.2: Przekształcenia wysyłane do demona ROS.

współrzędnych wcale nie musiałaby być łatwo dostępna. Ma to szczególne znaczenie dla skomplikowanych mechanizmów, na przykład wielosegmentowego ramienia manipulacyjnego. Obliczenie położenia i orientacji końcówki ramienia wymagałoby informacji o aktualnych pozycjach wszystkich poniższych segmentów. Która część systemu miałaby zajmować się obliczeniami i gdzie przekazywać te informacje?

Demon ROSa działa tutaj jak trzecia strona, zbierająca dane od sterowników i obliczającą położenia i orientacje wszystkich punktów. W takim przypadku, każdy segment symulacji mógłby przekazywać swój identyfikator, identyfikator obiektu którym steruje i jego pozycję do demona ROSa. Inne programy, na przykład do wizualizacji, mogłyby wtedy zapytać się demona o dokładne pozycje przegubów w przestrzeni kartezjańskiej, a on obliczyłby je i zwrócił wynik.

W symulacji platformy wielokierunkowej mechanika przekształceń układów współrzędnych jest potrzebna, gdyż wiadomość zwierająca pomiary z czujnika laserowego nie posiada informacji o aktualnej pozycji samego czujnika w przestrzeni, a jedynie wspomniany identyfikator. Orientacja i położenie potrzebne są programowi obliczającemu pozycję z czujników i ewentualnemu wizualizatorowi samych danych.

## 5.5 Instalacja ROSa

Instalacja programu na systemie operacyjnym jest złożona. Z wyjątkiem wersji Ubuntu długiego wsparcia, nie ma łatwego sposobu na instalację tej programowej struktury ramowej na innych dystrybucjach. Na przeszkodzie stoją błędy komilacji dla nowszych wersji kompilatorów, zależności od dokładnych wersji zewnętrznych bibliotek i inne problemy w czasie wykonywania, jak naruszenie ochrony pamięci.

Rozwiązaniem tego problemu jest instalacja odpowiedniego systemu operacyjnego na zewnętrznym nośniku danych, aby mógł być uruchomiony niezależnie od domyślnego systemu operacyjnego komputera, na tym systemie wtedy prosto zainstalować środowisko ROS.

Najnowszą wersją ROSa jest *Lunar Loggerhead* z maja 2017, jednak nie jest to wersja długiego wsparcia, a co za tym idzie, nie posiada wszystkich pakietów zewnętrznych twórców, potrzebnych przy wizualizacji symulacji. Odpowiedniejszą wersją jest *Kinetic Kame* z marca 2016 roku o bardzo dobrym wsparciu. Pakiety składające się na system ROS nadal są regularnie aktualizowane, lecz nie zawierają nowych funkcjonalności, a jedynie poprawki błędów. Symulator Gazebo jest w tej samej wersji na obu dystrybucjach, co też oznacza że przeniesienie projektu na najnowszą wersję ROSa jest trywialne.

Uruchomienie platformy programistycznej na systemie wymaga wielu dodatkowych komend inicjalizujących, a także dopisywania do tworzonych projektów licznych plików konfiguracyjnych za pomocą dostarczonych skryptów. Używanie pakietów z linii poleceń wymaga ustawienia kilku zmiennych sys-

temowych. Użycie niektórych funkcji ROS wymaga uruchomionego demona serwera w tle.

### 5.5.1 Tworzenie pakietów

Każdy pakiet jest katalogiem, w którym obowiązkowo znajdują się pliki `package.xml` i `CMakeLists.txt`.

Pierwszy zawiera metadane pakietu, takie jak nazwa, wersja, autor, opis. Posiada także listę zależności od innych pakietów.

Drugi jest skryptem programu CMake, który definiuje sposób budowy pakietu i także definiuje zależności. Oba pliki posiadają te same dane, skrypt kompilacyjny zgłosi błąd, jeśli nie będą się zgadzać między sobą.

W zewnętrznym katalogu uruchamia się skrypt kompilacyjny `catkin`, który kolejno sprawdza zawartość katalogów i wywołuje ich skrypty `CMakeLists.txt`. Tworzy dwa osobne katalogi, jeden z wygenerowanymi definicjami, drugi z plikami powstałymi w trakcie kompilacji. Dba o odpowiednie podawanie ścieżek do kompilatora i kolejność komplikacji pakietów, uwzględniając zależności. Na przykład, jeśli pakiet wymaga pliku nagłówkowego, generowanego przez komplikację innego pakietu, plik ten może być załączony w kodzie tak, jakby był systemowy. CMake zadba o wywołanie kompilatora z odpowiednimi argumentami, aby skompilować program wykonywalny lub bibliotekę.

Następnie, aby uruchomić program, wywołuje się komendę `rosrun` która odczytuje wygenerowane w trakcie komplikacji metadane, aby określić położenie bibliotek ładowanych dynamicznie i plików wykonywalnych.

## 5.6 Format SDF

*Simulation Description Format* (SDF) [13] jest formatem XML, pozwalającym na zdefiniowanie modeli i zależności pomiędzy nimi w przestrzeni trójwymiarowej, w szczególności budowy i rozmieszczenia robotów. Powstał jako zamiennik poprzedniego formatu URDF, ze względu na jego skomplikowaną semantykę i brak możliwości określania środowiska w którym poruszają się roboty.

W przeciwieństwie do poprzednika, w którym model był zapisany w strukturze drzewiastej, SDF określa wszystkie ogniva modelu na tej samej wysokości zagnieżdżenia w strukturze gwiazdistej. Za ognivo uznaje się część składową modelu, a także przegub.

Element typu `world` zawiera informacje o środowisku symulacji oraz elementy modeli biorących udział w symulacji. Dodatkowo, można dodać informację o ustawieniach maszyny symulującej fizykę, wyglądzie sceny, wietrza, grawitacji, polu magnetycznym, itp.

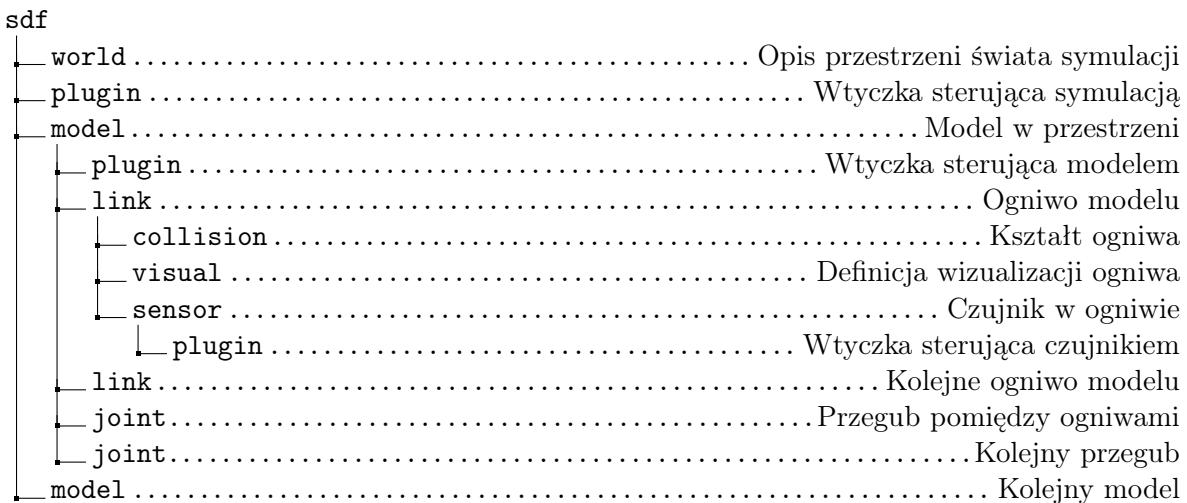
Element `model` definiuje model, wszystkie definicje ogniw tego modelu są logicznie rozmieszczone równolegle, jako jego dzieci tego elementu. Elementy więzów definiują interakcje pomiędzy ogniwami. Każdy model zawiera nazwę, początkową pozycję, sposób symulacji i wtyczki programów obsługujących zaawansowane zachowanie modelu.

Poprzez model w tym standardzie rozumie się nie tylko robota, ale także obiekty typu przeszkody, źródła światła, elementy animowane i tym podobne.

Elementy typu `link`, definiują normalne ogniva modelu. Każdy z nich jest osobną, kompletną częścią robota, na przykład kołem, fragmentem ramienia chwytaka, kadłubem, czujnikiem. Składa je w sobie informacje o pozycji względem lokalnego środka układu współrzędnych modelu, masie, kształcie, fizycznym kształcie, materiale fizycznym i wyglądzie.

Ogniva zawierają jedynie informacje o swoim początkowym umiejscowieniu w modelu, ale nie o sposobie poruszania się i nałożonych więzach. Do tego potrzebne są elementy `joint` określające przeguby. Każdy przegub określa, między jakimi obiektami się łączy.

Model lub jego fragment mogą być zimportowane z innego pliku, lecz nie zmieni to struktury gwiazdowej, a co za tym idzie, może dojść do utraty informacji. Jest tak, ponieważ to nie cały importowany model jest umieszczany wewnątrz pierwszego modelu, a jedynie jego ogniva. Zatem cała informacja zawarta w elemencie `model` importowanego pliku jest tracona, z wyjątkiem przedrostków nazw ogniw. Ten przypadek zachodzi przy modelach skanerów laserowych, opisanych w rozdziale 4.4.



Rysunek 5.2: Najważniejsze elementy formatu SDF.

## 5.7 Model kinematyki

Model kinematyki bazuje jedynie na prędkościach obiektów. Stworzony jest jako wtyczka do programu Gazebo, patrz sekcja 5.3.2. Użycie symulatora Gazebo służy do jednoczesnej wizualizacji pozycji obiektu, jak i całkowania prędkości w celu obliczenia pozycji.

### 5.7.1 Komunikacja

Komunikacja programu sterującego platformą odbywa się przez kanały komunikacyjne ROSa.

Wiadomość zawierająca dane prędkości kół czterokołowego robota nie mieści się w standardzie, zatem został stworzony specjalny typ `omnivelo_msgs/Vels`. Ta struktura zawiera cztery wartości zmiennoprzecinkowe podwójnej precyzji, oznaczające prędkości kątowe w rad/s.

Program w każdym cyklu symulacji nadaje wiadomości:

- `geometry_msgs/PoseStamped` z aktualnym położeniem i orientacją platformy oraz nagłówkiem z identyfikatorem i czasem nadania pakietu.
- `geometry_msgs/TwistStamped` z aktualną prędkością liniową platformy oraz podobnym nagłówkiem.
- `nav_msgs/Odometry` z obiema powyższymi danymi i nagłówkiem. Służy przy obliczaniu ruchów platformy na podstawie enkoderów kół.

Ponadto program przyjmuje dane:

- `omnivelo_msgs/Vels` z zadanymi prędkościami kątowymi kół.

### 5.7.2 Problemy implementacji

Gazebo nie ma zaimplementowanego pełnego wsparcia dla standardu SDF. W szczególności nie działa struktura elementów `frame`. Jest to mechanika macierzy przekształceń jednorodnych, podobna do ROSowego systemu, opisanego w sekcji 5.4. Nie jest to opisane w dokumentacji, a jedynie zgłoszone od kilku lat w systemie kontroli wersji jako błędy.

Oznacza to, że wszystkie elementy typu `link`, będąc dziećmi `model`, nie zachowują swojej pozycji w lokalnym układzie współrzędnych. Powoduje to, że nadając prędkość kątową modelowi, nadajemy ją każdemu ogniwu osobno.

Opis	Wartość
Główna część korpusu	60 kg
Przednia część korpusu	15 kg
Koło	1,6 kg
Skaner laserowy	1,1 kg

Tablica 5.3: Masy ogniw przyjęte w modelu.

Z punktu widzenia symulacji fizycznej ma to sens, gdyż nie można zakładać, że ogniva modelu są w jakikolwiek sposób podłączone. To wprowadzałoby także nieścisłości w typie zastosowanych domyślnie przegubów.

Aby przeciwdziałać temu zjawisku, należy umieścić wszystkie ogniva jako fragmenty siatki trójkątów, używanej przy wizualizacji. To znaczy, ustawić je nie jako dane używane przez maszynę symulacyjną fizyki, ale jako dane służące do rysowania obiektów na ekranie. W ten sposób traktowane są jako jedna całość, a nie osobne składowe. Nie można użyć tutaj więzów statycznych, gdyż te są wykorzystywane przez maszynę symulacyjną fizyki i ignorowane przy kinematycznym ruchu.

Powstała niedogodność jest taka, że trudniej jest sterować obrotem obiektów kół, gdyż są zarządzane przez kompletnie inny system symulatora, służący do graficznego renderowania sceny, działający na innym wątku, jednak w żaden sposób nie wpływa na ruch modelu bazy.

## 5.8 Model dynamiki

W modelu dynamiki przyjęto masy ogniw zgodne z tabelą 5.3. Masy przegubów i jednostki inercji są pomijalnie małe w porównaniu do najcięższych elementów. Momenty bezwładności ogniw zostały wyznaczone jak dla podstawowych brył, prostopadłościanów dla korpusu oraz walców dla kół i skanerów. Założono identyczną gęstość obu części korpusu.

Jest wiele sposobów na zamodelowanie takiego obiektu, poniżej przykładowe implementacje.

### 5.8.1 Wierność modelu

Należy zamodelować wszystkie ogniva modelu i nadać im kształt za pomocą odpowiednich modeli 3D. Kształt obiektów może być także przybliżony jednym z prymitywów jak sześcian, kula, łamana, walec, czy płaszczyzna. Takie przybliżenie znacznie przyspiesza obliczenia, gdyż może być specjalnie traktowane przez algorytmy obliczeń fizyki w maszynie symulacyjnej.

Niestety, rolki mają złożony kształt [4]. Przybliżenie takiego kształtu walcem powoduje problemy przy przenoszeniu punktu podparcia na kolejną rolkę, gdyż koło będzie musiało przez chwilę oprzeć się o krawędź rolki. Taki model wprowadzałby także drgania, zwiększąc tym samym i tak duże niedokładności symulacji. Podejście to zostało także zaproponowane w pracach [9] i [6].

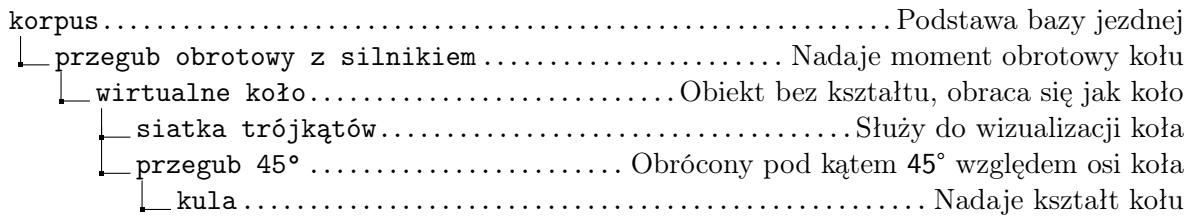
Przybliżenie rolki siatką trójkątów jedynie zmniejsza powyższy efekt, gdyż sama siatka zbudowana jest z prostych krawędzi. Zwiększając jej gęstość można teoretycznie poprawić jakość symulacji, kosztem olbrzymiego skoku ilości obliczeń. Obliczenia związane z wykryciem kolizji obiektów modelowanych za pomocą siatek trójkątów są najbardziej zasobniejsze ze wszystkich metod wykrywania kolizji.

### 5.8.2 Model koła z przywracaną orientacją

Sposób symulowania koła w ten sposób został użyty w modelach w symulatorze V-Rep.

Polega on na tym, iż koło, podłączone do korpusu za pomocą przegubu obrotowego, posiada drugi przegub obrotowy. Drugi przegub obrócony pod kątem  $45^\circ$  w stosunku do osi koła. Jego obrót jest zgodny z obrotem dolnej rolki koła, mającej kontakt z podłożem. Do drugiego przegubu jest podłączony obiekt kuli oddziałującej z podłożem, odpowiedzialny za kolizje. Zastosowano kulę, gdyż

jest prymitywem geometrycznym i obliczenia jej kolizji są najmniej wymagające obliczeniowo dla maszyny symulacyjnej. Orientacja kuli jest przywracana do orientacji początkowej w każdym kroku symulacji.



Rysunek 5.3: Zagnieżdżenie obiektów koła z przywracaną orientacją rolki.

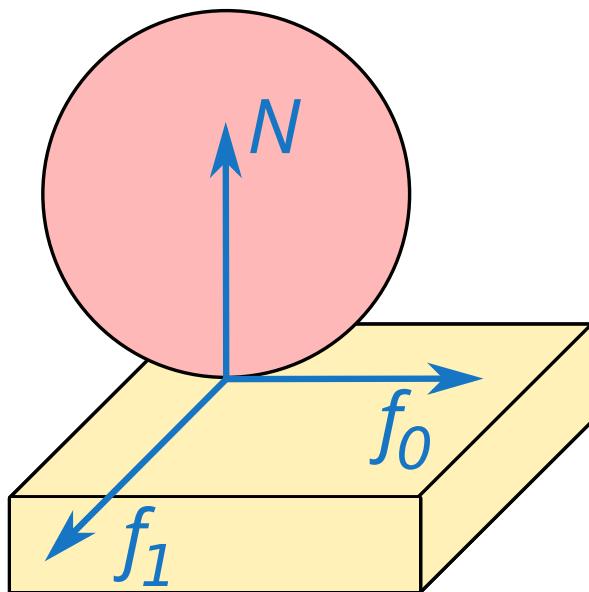
Wywołuje to takie działanie, jak gdyby koło w danej chwili mogło jednocześnie obracać się w dwóch kierunkach, wokół osi obrotu dolnej rolki oraz wokół osi koła. Przez następny krok symulacji, model zachowuje się poprawnie, aż pod wpływem obrotu koła, druga oś przestaje być równoległa do osi dolnej rolki co zaczyna negatywnie wpływać na symulację. Zanim jednak ten efekt się nasili, orientacja koła wirtualnego jest przywracana do pozycji początkowej, razem z kierunkiem drugiej osi. Ponieważ jest to powodowane nadpisaniem poprzedniej orientacji, a nie nadaniem momentu obrotowego obiektyowi, maszyna symulacyjna nie bierze w takim przypadku pod uwagę sił tarcia kuli o podłoże.

Niestety, nie jest możliwe zastosowanie tego rozwiązania wprost w Gazebo, gdyż struktura drzewiasta obiektów nie jest zaimplementowana, jak to wcześniej zostało opisane. Co więcej, metody natychmiastowo zmieniające pozycje obiektu nie działają poprawnie. W dodatku, potrzebna jest także możliwość ustawiania orientacji przegubu, elementu joint, co nie jest wystawione do modyfikacji w API.

Bardzo skomplikowany sposób działania kół skłania do szukania innych rozwiązań.

### 5.8.3 Modyfikacja kierunków i wartości wektorów tarcia

Ta implementacja została wybrana dla dynamicznego modelu platformy. Dla zrozumienia niżej wymienionych problemów należy opisać w jaki sposób są interpretowane dotyk i kolizja w maszynie symulującej dynamikę.

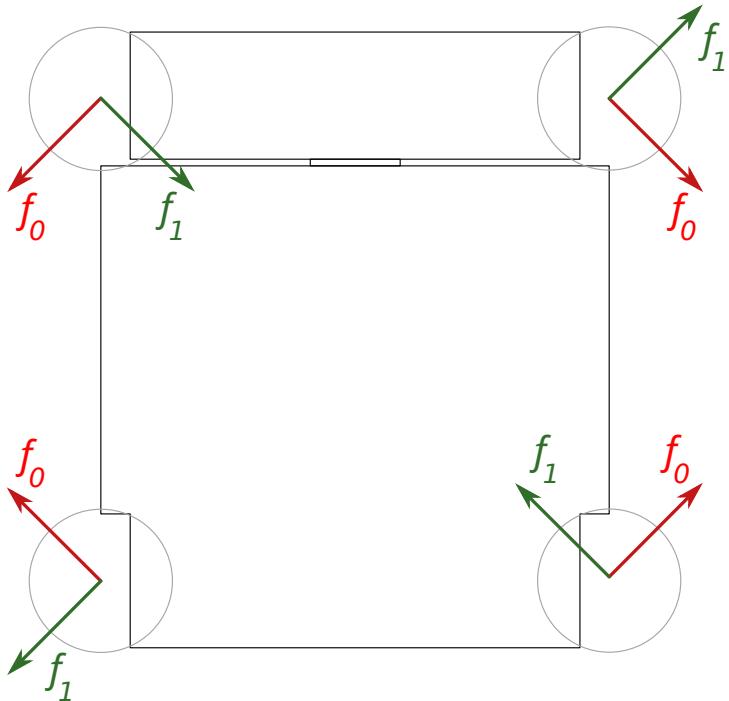


Rysunek 5.4: Wersory punktu kolizji.

Po wykryciu punktu kolizji i wyznaczeniu wektora normalnego  $N$  do dotykających się obiektów, system powinien obliczyć odpowiednie wartości sił, aby zatrzymać lub odbić obiekty od siebie. Dodatkowo, ponieważ prędkości obiektów nie muszą być równoległe do  $N$ , należy zasymulować siłę tarcia z odpowiednią dla współczynnika tarcia wartością. Można to uzyskać, nadając obiektom w punkcie kolizji siłę  $F$  prostopadłą do  $N$ , ten wektor może być rozpisany przy pomocy dwóch wektorów jednostkowych  $f_0$  i  $f_1$ . Te wektory zawsze są prostopadłe do  $N$ , równoległe do płaszczyzny kolizji.

W standardowej symulacji fizyki nigdy nie potrzeba osobno modyfikować współczynników tarcia i kierunku tych wektorów, gdyż zazwyczaj powierzchnie symulowanych obiektów mają jednakowe współczynniki tarcia w każdym kierunku. Jednakże modyfikując te wektory statycznie, lub dynamicznie, można uzyskać bardzo interesujące efekty. Instrukcja silnika symulacji podaje przykład, w którym aby zamodelować tarcie opon samochodu na zakręcie, prostopadłe do kierunku jazdy, należy dynamicznie zmieniać współczynnik tarcia dla wektora  $f_0$ , lub  $f_1$  w kierunku osi koła. Ten współczynnik tarcia, prostopadły do kierunku jazdy, może być liniowo zależny od prędkości. Spowoduje to, że im większa prędkość samochodu, tym boczna siła odrzutowa bardziej wpłynie na tor jego jazdy, co ma odwzorowanie w rzeczywistości. Więcej informacji można znaleźć na stronie instrukcji maszyny symulacyjnej ODE [16].

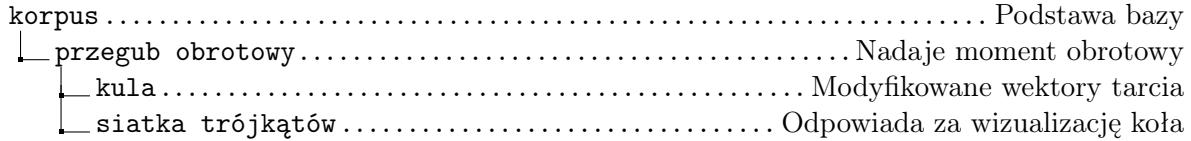
W opisywanym tutaj modelu, modyfikuje się kierunek wektora  $f_0$  oraz współczynniki tarcia dla obu wektorów, aby przybliżyć zachowanie się rolki do rzeczywistości. Ponieważ wektory  $f_0$  i  $f_1$  są określone w lokalnym dla koła układzie współrzędnych, w każdej iteracji maszyny symulacji należy obrócić je względem aktualnej pozycji bazy i odwrotności obrotu koła. Idealna rolka obraca się całkowicie bez tarcia, a ruch równoległy do jej osi jest niemożliwy. Można więc ustawić zerowy współczynnik tarcia w kierunku prostopadłym do osi, oraz nieskończonie duży dla wektora równoległego do osi.



Rysunek 5.5: Kierunki wektorów dla których należy nadać współczynniki tarcia przy symulacji platformy, widok z góry. Tarcie w kierunku  $f_0$  powinno być nieskończonie, a w  $f_1$  zerowe.

Niestety, w rzeczywistości rolki wykonane są ze śliskiego plastiku, który zezwala na poślizg kół wzdłuż ich osi. Osie kolek również nie obracają się płynnie, pod naciskiem platformy tarcie jest jeszcze większe. Każda rolka obraca się z innym tarciem wprowadzając kolejne zakłócenia. Podłoż, po którym porusza się robot, także nie jest tu bez znaczenia. Należy zatem wystawić interfejs do łatwej zmiany współczynników tarcia, aby później dobierać odpowiednie wartości na podstawie zachowania rzeczywistego robota.

Podobnie, jak w poprzednich przypadkach, modeluje się tylko najniższą, dotykającą podłożą rolkę. Jak wcześniej wspomniano, ma ona bardzo skomplikowany kształt, lecz można przybliżyć całe koło kulą. Zatem w miejscu każdego koła ustawiona jest kula z dynamicznie modyfikowanym tarciem i siatką trójkątów w kształcie koła do wizualizacji, oraz przegub z motorem łączący odpowiednią część bazy z kołem. To najprostsza budowa modelu (a zatem najszybsza) z poprzednich.



Rysunek 5.6: Logiczne zagnieźdżenie obiektów koła w strukturze drzewiastej z modyfikowanymi wektorami tarcia.

Takie rozwiązanie wiąże się z pewnym ryzykiem. Wymaga, aby symulator używał maszyny ODE, co zmniejsza przenośność modelu. ODE jest domyślnym symulatorem w Gazebo. Maszyna Bullet również liczy kolizje w podobny sposób i ma modyfikowalne wersory tarcia i współczynniki, lecz nie daje podobnych wyników. Być może jest to spowodowane brakiem odpowiedniej konfiguracji lub innym wewnętrznym traktowaniem modelu.

#### 5.8.4 Ustawienie mas i momentów bezwładności

W standardzie SDF zdefiniowano element `inertial`, który definiuje w sobie masę oraz macierz inercji. Dla uproszczenia wszystkie ogniva mają ustawione swoje środki w punktach środków mas, co oznacza, że każda z macierzy inercji jest macierzą przekątną. Masy zostały nadane zgodnie z danymi z tabeli 5.3, a macierze inercji obliczone ze wzorów na macierze inercji prostopadłościanu lub walca. Gazebo pozwala na wizualizację nadanych parametrów w formie narysowanych prostopadłościanów.

Środki mas zostały obliczone, bazując na uproszczonych kształtach ogniw, przy założeniu jednorodnego rozkładu gęstości w korpusie.

Jeśli dane mas i momentów bezwładności nie są poprawnie ustawione lub zostawione na domyślnych wartościach, model może powodować błędy maszyny symulacyjnej. Występują wtedy „glitche”, czyli losowy ruch i obrót obiektu w różnych kierunkach. Model wpada w taki stan i leci w losowym kierunku. Taki efekt czasami występuje w grach komputerowych. Jest to spowodowane tym, że dochodzi do błędów obliczeń numerycznych z powodu za dużych różnic w parametrach obiektów i obliczeń kolizji.

#### 5.8.5 Model silników

Platforma wyposażona jest w cztery niezależne serwomotory. Model takiego obiektu jest przegubem obrotowym, któremu można nadawać odpowiednią prędkość, w zależności od odebranych zadanych wartości. Ponieważ każda maszyna symulacyjna fizyki inaczej interpretuje tę wartość, należy zwrócić szczególną uwagę na sposób nadawania prędkości obiektem.

Przede wszystkim, prędkość jest zawsze reakcją na przyłożoną siłę tarcia lub moment siły. W rzeczywistości nie istnieje sposób na natychmiastowe nadanie prędkości obiektyowi, jak to jest możliwe w symulacji. Pomimo że maszyna posiada wewnętrzną informację o aktualnej prędkości, jej bezpośrednia modyfikacja może wprowadzać nietypowe zachowania. Przede wszystkim należy brać pod uwagę reakcję ze strony połączonego obiektu, w tym przypadku korpusu, a także momentów ustawiania tej prędkości. Ustawianie zadanej prędkości powinno być zsynchronizowane z krokami symulacji a nie częstotliwością odbioru wiadomości ze strumienia komunikacyjnego.

Najlepsze efekty daje nie nadawanie prędkości obiektom, a nadanie docelowej prędkości i maksymalnego momentu obrotowego. Maszyna symulacji fizyki wtedy odpowiednio zinterpretuje dane, obliczając również reakcję na obu końcach przegubu. Jest także możliwość ustawienia algorytmu sterownika, na przykład PID, ale takim ustawieniem należy określić odpowiednie parametry do pracy.

W modelu odwołano się bezpośrednio do zmiennych maszyny symulacyjnej ODE za pomocą interfejsu symulatora Gazebo. Jest to metoda wirtualna, implementowana do komunikacji z każdą maszyną symulacyjną osobno.

```
virtual bool SetParam (const std::string& key,  
                      unsigned int index,  
                      const boost::any& value)
```

Metoda przyjmuje nazwę modyfikowanej wartości, oś przegubu (dla silników to zawsze będzie jedyna oś 0) oraz nadawaną wartość. Modyfikowane wartości to:

fmax Maksymalny moment obrotowy w N m, używany przez maszynę symulacyjną do nadania docelowej prędkości kątowej przegubowi. Ustawiany raz przy inicjalizacji modelu. Ponieważ w platformie ta wartość nigdy nie będzie osiągnięta ze względu na ograniczenia energetyczne, można założyć tutaj wysoką liczbę 100 N m.

vel Docelowa prędkość obrotowa w rad/s zapamiętana i ustawiana w każdym kroku symulacji, nadpisywana na odbiór nowych wartości z komunikacyjnego strumienia wiadomości.

### 5.8.6 Komunikacja

Ze względu na wiele ustawień elementów bazy, należy stworzyć bogaty interfejs. W każdym cyklu symulacji, program sterujący modelem platformy nadaje wiadomości:

- geometry\_msgs/PoseStamped z aktualnym położeniem i orientacją platformy oraz nagłówkiem z czasem i identyfikatorem.
- geometry\_msgs/TwistStamped z aktualną prędkością platformy i nagłówkiem.
- omnivelma\_msgs/EncodersStamped z odczytaną ze stanu obiektów kół aktualną prędkością kątową i kątem obrotu, z nagłówkiem. To jest symulator enkoderów wbudowanych w silniki platformy.

Przyjmowane są także dane:

- omnivelma\_msgs/Vels z zadanymi prędkościami kół.
- Wywołanie ustawiające współczynniki tarcia wzdłuż wektorów  $f_0$  i  $f_1$ .
- Wywołanie ustawiające masy i momenty obrotowe niektórych elementów składowych konstrukcji.

### 5.8.7 Rozszerzenie modelu

Ponieważ komputerowa reprezentacja liczby zmiennoprzecinkowej pozwala na zapisanie nie tylko liczbowych wartości, można rozszerzyć model o dodatkową funkcjonalność, wywoływaną wysłaniem do modelu cichej nie-liczby (*NaN*) w wiadomości, w polu prędkości odpowiedniego koła. Cicha nie-liczba powstaje w procesorze, w module operacji zmiennoprzecinkowych, przy przeprowadzaniu nieprawidłowych, acz niekrytycznych obliczeń, na przykład dzielenie przez zero, lub dzielenie nieskończoności przez minus-nieskończoność (także zapisywane jako forma liczby zmiennoprzecinkowej). Takie operacje nie powodują błędu programu, jedynie wynik w postaci nie-liczby propaguje przez wszystkie pozostałe operacje.

Nadanie prędkości modelem w przestrzeni wirtualnej polega na wywołaniu odpowiedniej funkcji maszyny symulującej fizykę. Można zadać pytanie, jak zachowa się model, jeśli dla niektórych kół nie zmieniać prędkości po każdym odebraniu pakietu?

Wobec tego, jeśli w pakiecie z nowymi prędkościami kół znajdzie się cicha nie-liczba, program sterujący nie nada nowej prędkości temu kołu. Jest to podobne do nadania tej samej prędkości, jaką posiada aktualnie obiekt koła (jaką zwróciłby enkoder).

Zwraca to uwagę również na potrzebę, aby program do komunikacji z rzeczywistym robotem nie skończył się błędem po odebraniu jednej z takich nieokreślonych wiadomości. Ponieważ przekształca liczby zmiennoprzecinkowe, zawarte w ROSowych pakietach, na dane zrozumiałe przez sterownik silnika, które zazwyczaj są liczbami stałoprzecinkowymi, program może zachować się nieprzewidywalnie.

## 5.9 Model skanera laserowego

Ponieważ skanery laserowe tego typu są popularnie używane w robotyce, standard SDF posiada dedykowane elementy do umieszczenia takich obiektów w symulacji. Również Gazebo posiada możliwość renderowania zasymulowanych impulsów lasera.

### 5.9.1 Obliczenia symulatora

Skaner laserowy jest bardzo łatwo zasymulować w przestrzeni wirtualnej za pomocą rzutowania półprostych. Ta technika używana jest w bardzo wielu aspektach komputerowego generowania obrazu i symulacji fizyki.

Półposta jest emitowana z ustalonego punktu w pewnym kierunku w przestrzeni trójwymiarowej. Następnie system próbuje znaleźć pierwszy punkt jej kolizji z każdym z obiektów o fizycznym kształcie, uczestniczących w symulacji.

Ponieważ zasoby komputera zawsze są ograniczone, długość promienia także musi mieć pewien limit. Zwykle jest on jednak na tyle duży, że z punktu widzenia obiektów uczestniczących w symulacji, w opisywanym tutaj zagadnieniu, można uznać tą odległość za nieskończoną.

Algorytm obliczania kolizji z półpostą bazuje na kosztowym porównywaniu pozycji każdego obiektu fizycznego na scenie. Istnieją oczywiście sposoby na zmniejszenie ilości obliczeń, na przykład metoda prostopadłościanów zawierających obiekt, ale sposób radzenia sobie z tym zagadnieniem nie jest częścią tematu pracy. Wystarczy wspomnieć, że symulacja dużej ilości laserów oraz obiektów jest operacją kosztowną.

Testy pokazują, że samo ich rysowanie spowalnia symulację około czterokrotnie. To ze względu na bardzo dużą ich ilość, mogącą przekroczyć 1000 obliczeń kolizji w jednej klatce symulacji.

### 5.9.2 Różnice między czujnikiem, a modelem

Półposta emitowana jest z punktu reprezentującego środek skanera. Model upraszcza rzeczywisty czujnik (budowa skanera laserowego została opisana w sekcji 2.5). Uproszczenie to polega na tym, iż nie ma wewnętrz zamodelowanego obiektu żadnego odpowiednika obracającego się lusterka. W rzeczywistym czujniku ponadto jest jeden laser, emitujący脉sy w określonych odstępach czasu. W modelu warto zatem emitować osobne półproste, dla każdego pulsu lasera.

Można zauważyc tym samym, że model skanera wydaje się funkcjonalnie lepszym, niż rzeczywisty LiDAR. W danej chwili model emmituje promień we wszystkich kierunkach w zakresie jednocześnie, podczas gdy skaner jednym pulsem może dokonać tylko jednego pomiaru, i tylko o kącie w którym aktualnie znajduje się lusterko. Jednakże dyskretny sposób symulacji i sposób komunikacji urządzenia z odbiornikiem danych powodują że w obu przypadkach dane są podawane w zbiorach. Sterownik skanera jest w stanie wysłać pakiet z danymi z ostatniego pomiaru, podczas gdy program modelujący skaner jest obsługiwany na zasadzie przerwań czasowych po każdym kroku symulacji i tylko wtedy może wywołać funkcje zwracające dane zasymulowanych pomiarów. To oznacza, że interfejsy do ich obsługi zachowują się podobnie.

Drugą rzeczą, w której model przoduje, jest nieskończona (z punktu widzenia symulacji), odległość pomiaru. Nie tylko jako najdalszy wykryty punkt, ale także i najbliższy. Skaner może pomijać pomiary przypadające za blisko krawędzi dozwolonego obszaru, gdyż znacznie spada w tych miejscach dokładność pomiaru lub zwraca niedokładne dane. Symulator ma całkowitą dowolność w ustawianiu progu, dla którego obcina pomiar.

Podobnie, jak w poprzednim przypadku, symulator posiada niezmienną w odległości dokładność pomiaru. Błąd pomiarowy sknera laserowego zależy od odległości od mierzonego obiektu.

Jednakże, w zależności od obciążenia maszyny na której uruchomiony jest symulator, model skanera jest podatny na opóźnienia w odczytywaniu stanu. Skaner zawsze działa z tą samą częstotliwością, a jego program sterujący jest wbudowany w mikrokontroler i spełnia sztywne ramy czasowe.

### 5.9.3 Komunikacja

Bazując na architekturze opisanej wcześniej na rysunku 4.2, należy tak zbudować system, aby program sterujący mógł się komunikować w identyczny sposób z modelem skanera, jak i samym skanerem. Służą do tego specjalne typy wiadomości ROSa `sensor_msgs/LaserScan`. Program obsługujący model skanera generuje i wysyła pakiety zawierające:

- Nagłówek z czasem pomiaru, numerem sekwencyjnym i identyfikatorem macierzy przekształcenia jednorodnego pozycji czujnika.
- Kąty początkowe i końcowe pomiaru.
- Odległość kątowa pomiędzy kolejnymi promieniami.
- Czas pomiędzy kolejnymi emisjami lasera.
- Czas pomiędzy tym, a poprzednim przebiegiem urządzenia.
- Minimalny i maksymalny dystans mierzonego obiektu od skanera.
- Dane odległości.
- Dane intensywności.

### 5.9.4 Model w Gazebo

Tak, jak w modelu platformy, należy stworzyć odpowiedni plik SDF. Warto umożliwić stosowanie modelu czujnika w modelach innych robotów. Zatem jego implementacja powinna być niezależna od implementacji platformy, do której będzie przytwierdzony. Dodatkowo, w końcowym modelu istnieć będą dwa takie czujniki, budowa pliku powinna pozwolić na wielokrotne importowanie tych samych danych do tego samego modelu, ale jednak aby były interpretowane w różny sposób (gdyż nadawcy danych muszą być rozróżnialni).

Model składa się z dwóch elementów: korpusu i samego „mechanizmu” urządzenia. Mechanizm przytwierdzony jest w odpowiednim miejscu korpusu platformy za pomocą stałego przegubu (elementu `joint`).

Korpus czujnika posiada siatkę trójkątów, reprezentującą uproszczony wygląd urządzenia, a także dwa element ustawiający kształt walca, odpowiedzialny za kolizje fizyczne. Element korpusu odpowiada także za przesunięcie samego lasera względem podstawy, na której całe urządzenie jest montowane, i pozwala na wygodną referencję z innego modelu, w celu utworzenia przegubu. Jak już wcześniej wspomniano, model zawsze ma strukturę gwiazdową i przeguby po stronie robota nie mogą wskazywać na element `model` modelu skanera, a mogą na obiekt korpusu.

Główna część obiektu czujnika, skaner, posiada ozdobną siatkę trójkątów, udającą czarną szybkę LiDARa, oraz element SDF `sensor`, odpowiedzialny za sam czujnik. W kolejnych podelementach zawierają się parametry urządzenia, takie jak ilość symulowanych laserów, ich zasięg, kąt pierwszego i ostatniego lasera, oraz współczynnik błędu pomiarowego. Ten element celowo nie ma fizycznego kształtu, aby nie blokować wychodzących półprostych. Nie wpływa to na symulację, gdyż w środowisku, w którym znajduje się robot, i tak nie powinno dochodzić do kolizji modeli czujników z jakimikolwiek innymi modelami. Również czujniki nie są w stanie wykryć siebie nawzajem, gdyż zwrocone są do siebie martwymi kątami, a co za tym idzie nie muszą symulować nieprzezroczystych brył dla

innnych sensorów. Półproste emitowane są z punktów leżących kilka centymetrów od środka skanera, aby symulator nie wykrył ich kolizji z samym korpusem skanera.

### Połączenie modeli

Jak wcześniej wspomniano w sekcji 5.6, model SDF ma strukturę gwiaździstą. Zagnieżdżenie modeli spowodowałoby, że powstałaby inna struktura, drzewiasta. Dlatego też, element import nie umieszcza w swoim miejscu całego modelu z innego pliku, a raczej importuje jego ogniva i umieszcza równolegle do istniejących. To oznacza, że zadbać trzeba także o więzy joint, łączące element podstawy platformy z podstawą czujnika, inaczej symulator uznałby łączony obiekt za dwa osobne modele. Potrzebna jest zatem znajomość nazw elementów ogniw importowanego modelu. Element model importowanego modelu jest tracony, pozostaje jedynie przedrostek nazwy w zimportowanych składowych. Zatem program sterujący czujnikiem powinien na podstawie tylko nazwy swojego obiektu ustawić przedrostek swojego interfejsu nadawania wiadomości.

Taka mechanika działania wydaje się mało zrozumiała i nieintuicyjna, jednak doskonale dba o zachowanie spójności modelu. Wszystko nadal pozostaje strukturą gwiaździstą i każdy element musi być odpowiednio połączony z pozostałymi, aby dokładnie określić fizykę interakcji. Nie powstają niedopowiedziane sytuacje, w których zachowanie jakichś ogniw byłoby nieokreślone.

Alternatywnie, zawsze jest możliwość stworzenia dwóch, osobnych modeli skanerów, tudzież całość zapisać w jednym pliku. Jednak takie rozwiązanie niszczy pakietową budowę środowiska i nie pozwala na użycie modeli skanerów w innych modelach.

Należy zadbać o ustawienie odpowiedniej struktury macierzy przekształceń jednorodnych, opisanej w sekcji 5.4. Symulator platformy zawiera drugi program, który w każdym cyklu symulacji nadaje demonowi ROS położenia i orientacje środków czujników laserowych, dla uproszczenia względem początku układu współrzędnych, punktu (0,0,0). Program sterujący modelem samej platformy także nadaje przekształcenie z pozycją platformy względem globalnego środka układu współrzędnych. Dokładnie taki sam efekt byłby, gdyby nadawać przekształcenie ze stałą pozycją czujników laserowych, ale względem platformy (której przekształcenie nadawane jest przez inny sterownik). Stałą pozycją, ponieważ czujniki nie zmieniają swojej pozycji na platformie, są przytwierdzone na stałe.

### 5.9.5 Błędy

Jak podano wcześniej w tabelce 2.2, wyróżnione są dwa typy błędów pomiaru, systematyczny i po-miarowy. Dodatkowo istnieje także błąd gruby. Model czujnika powinien uwzględniać wszystkie błędy, aby zwracać dane jak najbardziej zbliżone do LiDARa.

#### Błąd gruby

Najprostszy typ błędu polega na dużych odchyłach niektórych pomiarów od pozostałych wartości. W trakcie przetwarzania odczytu, te punkty powinno się odrzucić. Pomimo, że sterownik skanera usuwa takie błędy, można łatwo dodać je do generowanych danych w celu testowania algorytmów określania pozycji.

Najczęstszym przypadkiem błędu grubego jest brak odbioru wysłanego impulsu. To skutkuje nadaniem aktualnemu pomiarowi wartości maksymalnej, co jest bardzo łatwo wykryć i usunąć.

Innym problemem może być odebranie światła niepochodzącego od emitera urządzenia, a jakiegoś zewnętrznego źródła.

Ponieważ rozkład i częstotliwość tych błędów zależy od środowiska w jakim działa czujnik, bardzo ciężko jest dobrać odpowiedni algorytm ich generacji.

#### Błąd systematyczny

Ten błąd jest stałą wartością, dodaną do każdego pomiaru. Spowodowany jest niedoskonałością budowy elementów pomiarowych, niewłaściwą kalibracją, zużyciem, lub otoczeniem w jakim pracuje

czujnik.

Rzeczywisty LiDAR powinien być skalibrowany przed użyciem właśnie po to, aby wewnętrzny program sterujący mógł obliczyć aktualne zboczenia pomiarów i skorygować dane przed wysłaniem ich wyżej. Skaner może także wysyłać czyste i obarczone błędami dane do programu sterującego, który samodzielnie je skoryguje. Pozwoli to na zastosowanie dowolnych algorytmów oczyszczania danych, kosztem większego obciążenia programu sterującego.

Symulator czujnika nie jest podatny na takie błędy, więc nie ma sensu dodawać interfejsu do kalibracji urządzenia. Można co najwyżej symulować niepoprawnie skalibrowany skaner poprzez sztuczne dodanie takiego błędu.

### Błąd pomiarowy

Jest to mała, losowa wartość, dodana do każdego pomiaru. Wynika ona z niedoskonałości samego czujnika, nieznanych zakłóceń i niezbadanych efektów kwantowych. Nie da się w żaden sposób usunąć, zmniejszyć, lub przewidzieć tego typu błędów. Jedynym sposobem jest obliczenie średniej błędu na podstawie dużej ilości pomiarów.

Błąd pomiarowy ma zwykle rozkład normalny o określonym odchyleniu standardowym. Standard SDF przewiduje element określający tę liczbę, a Gazebo może wewnętrznie obliczyć i dodać do wyników odpowiednią wartość. Również producent podał w tabeli danych urządzenia obliczony rozkład standardowy.

W związku z tym, wartość podana przez producenta, podana w tabelce 2.2, może być bezpośrednio zapisana do elementu odchylenia standardowego, w pliku SDF opisującym czujnik. Wadą takiego rozwiązania jest niemożność modyfikacji tego parametru w trakcie wykonywania programu, gdyż Gazebo nie wystawia API do modyfikacji tej wartości. Aby temu zaradzić, wystarczy obliczać błąd standardowy w programie sterującym i manualnie dodawać go do zwróconej przez symulator tablicy danych. Funkcje do obliczania błędu standardowego zostały wprowadzone do standardu języka C++.

Na zrzucie ekranu 4.3 można zobaczyć, iż punkty pomiarów, wizualizowane w RViz, nie leżą idealnie na figurach powstałych poprzez przecięcia skanowanych brył. Dodany jest szum, jak gdyby rozmazujący punkty.

## 5.10 Model jednostki inercyjnej

Ponieważ czujniki tego typu są często stosowane w robotyce, ROS i Gazebo wspierają jego symulację i struktury przekazywanych danych.

- ROS posiada specjalną wiadomość typu `sensor_msgs/Imu`, do przekazywania pomiarów pomiędzy węzłami.
- SDF definiuje element typu `imu` w sekcji czujników, gdzie można mu zdefiniować położenie w robocie i współczynniki błędów pomiarowych.
- Gazebo daje wsparcie klasy czujnika inercji z odczytem wygenerowanych przez maszynę symulacji wartości.

Warto tutaj nadmienić, że struktura wiadomości ROSa posiada pola dla danych, które nie koniecznie mogą być wygenerowane przez czujnik rzeczywisty. Takimi polami jest struktura rotacji, zapisana jako quaternion, oraz macierze kowariancji, wyznaczane zewnętrznie eksperymentalnie.

Macierze kowariancji definiują wpływ danych z jednej osi na drugą i mnożniki wyjścia. Nierówność pomiarów na przykład może być to spowodowana odchyleniem akcelerometrów względem kąta prostego, co powoduje, że ruch w osi jednego czujnika może być także wykryty przez czujniki innych osi. Podobnie jest, gdy czujniki nie generują dokładnie tych samych danych na taki sam ruch wzdłuż ich

Opis	Wartość
Średnia	0
Odchylenie standardowe czujnika prędkości kątowej	0,003
Odchylenie standardowe czujnika przyspieszenia liniowego	0,1

Tablica 5.4: Wartości szumu o rozkładzie Gaussa, zastosowanego w modelu jednostki inercyjnej.

osi, co może być spowodowane niedokładnością wykonania elementów. Macierz pozwala zastosować te cechy sprzętowe do danych w celu poprawy ich jakości.

W większości programów używających przestrzeni wirtualnej, rotację zapisuje się w postaci kwaterniona, jako cztery liczby. Taka postać odporna jest na zjawisko utraty jednego ze stopni swobody (*gimbal lock*). Gdy dwie z trzech osi pokryją się, niemożliwy staje się obrót obiektu wokół trzeciej osi. Niestety, taka postać rotacji nie ma odwzorowania w rzeczywistej przestrzeni.

Symulacja żyroskopu w maszynie symulacyjnej fizyki jest bardzo prosta, gdyż algorytm wyznaczania pozycji obiektów na podstawie nadanych sił korzysta wewnętrznie z wartości prędkości dla każdego obiektu. Zatem kwestia symulacji tego sensora polega na odczycie odpowiednich struktur z maszyny symulacji. Wtyczka do Gazebo, zapisana podobnie do wtyczki czujnika laserowego, zbiera w każdym cyklu maszyny symulacyjnej dane i wysyła je za pomocą pakietu ROSa.

Na podstawie testu opisanego w sekcji 6.3, wyznaczono w tabeli 5.4 parametry dodatkowego szumu, dodawanego do odczytów w celu dokładniejszego zamodelowania błędów pomiarowych.

Akcelerometr jest bardziej złożonym problemem, gdyż maszyna symulacji działa w czasie dyskretnym, co utrudnia różniczkowanie prędkości w celu otrzymania przyspieszenia. Ta wartość nie jest także nigdzie indziej używana i musi być obliczona specjalnie dla symulacji tego czujnika. Fakt, że małe odchylenia w zmianie prędkości powodują duże skoki danych przyspieszenia wprowadza naturalny szum do generowanych danych.

Pomimo, że odczytanie tych wartości jest tak samo proste, jak prędkości kątowej, to generowane dane różnią się jakością. Szum jest większy, a co za tym idzie, potrzeba dodatkowego programu do filtracji wygenerowanych wartości. Ten pakiet jest opisany w sekcji 4.11.

## 5.11 Manualne sterowanie

### 5.11.1 Program

Ten pakiet jest plikiem wykonywalnym, skompilowanym ze źródeł w C++, w sposób opisany w sekcji 5.3.1. Wykorzystując bibliotekę graficzną SFML, generuje okno z powierzchnią do rysowania na nim za pomocą OpenGL. Biblioteka ta pozwala również na bezproblemowe przechwytywanie zdarzeń z klawiatury, takich jak wcisnięcie i puszczenie klawisza, a także na obsługę kontrolera gier i myszki. Za pomocą gałek kontrolera, można nadawać robotowi niebinarne prędkości kół, co jest niezbędne do płynnego i bezpiecznego kontrolowania urządzeniem. Użyto także sterowania kursorem myszy, w razie gdyby użytkownik nie posiadał kontrolera.

Aplikacja tego typu mogłaby bez większego problemu pracować z interfejsem tekstowym w terminalu, aby być bardziej przenośna i lżejsza w zasobach, lecz nie mogłaby wykrywać zdarzeńпусzczenia klawisza (bez bezpośredniego czytania z urządzenia /dev/input/eventX, do czego są potrzebne prawa roota). Dodatkowo, interfejs graficzny pozwala na wyświetlenie dokładniejszych wskaźników i elementów wskazujących.

Program uruchamiany jest z wiersza polecenia, z argumentami dotyczącymi nazw strumieni i początkowej konfiguracji urządzenia.

Program stworzony jest jako maszyna stanów, która odpowiada za odpowiednią interpretację naciśniętych klawiszy, w zależności od trybu w jakim się znajduje. Bloki instrukcji warunkowych rysują na ekranie odpowiedni interfejs w każdym cyklu rysowania.

Program działa na dwóch wątkach. Pierwszy z nich odpowiada za rysowanie danych na ekranie i obsługę wejścia, drugi odpowiada za komunikację ze strumieniem wiadomości. W ten sposób opóźnienia spowodowane wysyłaniem sterowania do programu nie wpłyną na częstotliwość wysyłania wiadomości.

### 5.11.2 Komunikacja

Program potrafi generować dwie typy wiadomości.

Pierwszą są prędkości kół `omnivelma_msgs/Vels`, jakie w danej chwili platforma powinna przyjąć na sterowanie. Pozwala to na dokładne przetestowanie zachowania się modelu platformy. Można także wywołać takie prędkości, które nie powinny być używane przy rzeczywistym sterowaniu, gdyż wprowadzają duże nieścisłości ruchu na skutek poślizgów (przykładowo, obracanie przednich i tylnych kół tak, aby ich wektory prędkości się znosiły będzie nadawać niedeterministyczny ruch, spowodowany niedoskonałościami pojedynczych rolek).

Drugi typ wiadomości, `geometry_msgs/Twist`, to nadana prędkość względna platformy. To intuicyjny sposób, w jaki użytkownik steruje platformą i w jaki sposób mógłby także sterować nią prosty program sterujący. Jednak ponieważ model platformy nie jest w stanie poruszać się bez informacji, jakimi kołami z jaką prędkością obracać, ten typ wiadomości musi być jeszcze konwertowany przez pakiet węzel kinematyki odwrotnej, opisany w sekcji 4.6. Program może być bezpośrednio podłączony do robota.

Program opcjonalnie przyjmuje także wiadomość `omnivelma_msgs/Vels`, aby wyświetlić dane encoderów. Należy zauważyć, że nie przyjmuje całej wiadomości `omnivelma_msgs/EncodersStamped`, jaka jest generowany przez model platformy, a jedynie mały jej wycinek, gdyż tylko te informacje jest w stanie wyświetlić i tylko takie potrzebuje. Dzięki temu może być użyty niezależnie od innych pakietów i programów. Jednak może być wymagane użycie dodatkowego programu do wyłuskania tej informacji z większej wiadomości (patrz pakiet 4.9).

## 5.12 Generator sterowania

Program wczytuje plik danych, w którym znajdują się rekordy, każdy opisuje:

- Prędkość liniową platformy w osi X.
- Prędkość liniową platformy w osi Y.
- Prędkość kątową platformy wokół osi Z.
- Czas  $t$ , przez który program ma generować wiadomość z podanymi wyżej danymi.

Program czyta także z argumentu okres  $T$  odświeżania wiadomości.

Korzystając z dwóch liczników systemowych, program generuje co  $T$  sekund wiadomość typu `geometry_msgs/Twist` z danymi aktualnie wykonywanej linii pliku.

Pierwszy licznik nadaje regularnie aktualne sterowanie, aby podtrzymywać prędkość robota, drugi licznik asynchronicznie wywołuje się na zmiany sterowania i aktualizuje wysyłane dane.

Pomimo, że mechanika liczników czasu jest dostępna w API ROSa, użycie liczników systemowych POSIX pozwalała zastosowanie ich mechaniki bezpośredniego czasu, czas nadania wiadomości jest definiowany względem stopki czasu, a nie poprzedniego wywołania. Dzięki temu nie ma narastającego błędu numerycznego, jak w funkcjach ROSa.

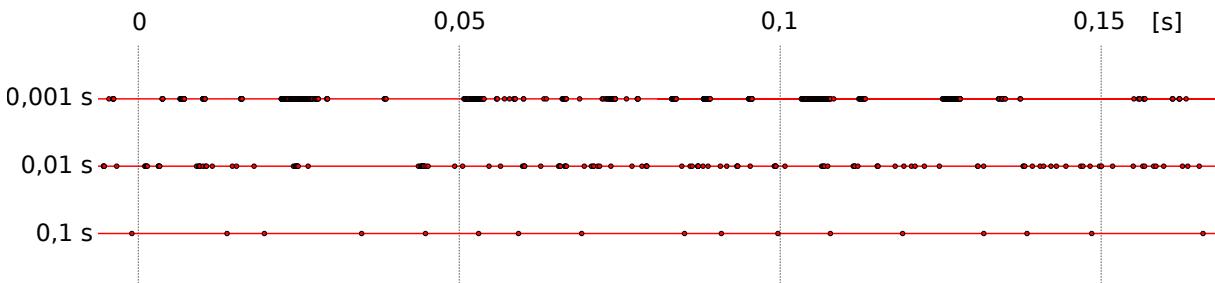
Dane z wejściowego pliku wczytywane są do listy, z której po kolejnych wywołaniach licznika zdejmowane są kolejne instrukcje.

Po wykorzystaniu danych, algorytm nadal generuje wiadomości z zerowymi prędkosciami, aby zatrzymać model platformy i podtrzymać aktywne hamowanie kół.

W ten sposób, możliwe jest proste generowanie sterowania robota, bazujące na czasie. Przykładowo, program może generować sterowanie:

1. Ruch przez 3,2 s, z prędkością 0,2 m/s, w kierunku (2,1).
2. Zatrzymanie na czas 0,9 s.
3. Obrót przez czas 10 s, z prędkością kątową 0,02 rad/s.

Ponieważ jednak system operacyjny, na którym pracuje program i symulator, nie spełnia wymogów systemu czasu rzeczywistego, generowane wiadomości nie muszą być (i nie są) wysyłane dokładnie w określonych momentach. Można przeprowadzić prosty eksperyment, aby zaprezentować ten problem.



Rysunek 5.7: Czas odbioru pakietu, wysyłanego przez program działający z jednym z trzech okresów.

Na rysunku 5.7 przedstawiono czas odbioru wiadomości generowanej przez program z określonymi okresami czasu. Działa to tak samo, jakby każdą wiadomość wysyłać z losowym opóźnieniem. Gdy nadsygnal jest za dużo (górny wykres), system zaczyna je buforować, a co za tym idzie, nadaje je w grupach o losowej wielkości. Wiadomość po nadaniu może być przekazywana przez wiele węzłów, każdy może nadawać losowe opóźnienie dla wiadomości.

Aby rozwiązać ten problem, należałoby uruchomić całe środowisko ROSa na systemie operacyjnym czasu rzeczywistego, również z tym węzłem generującym dane.

## 5.13 Podłożo o zmiennym współczynniku tarcia

Robi się to wewnętrznie w identyczny sposób, jak w przypadku kół platformy, co zostało opisane w sekcji 5.8.3.

W tym przypadku jednak powinno się ustawić identyczne wektory tarć  $f_0$  i  $f_1$ , aby podłożo symulowało równe tarcie we wszystkich kierunkach. Tak samo, jak w przypadku modelu platformy dynamicznej, program nadający podłożu odpowiednie wektory tarcia przyjmuje asynchroniczne wywołanie typu `omnivelman_msgs/SetFriction`, zawierające dwie wartości zmiennoprzecinkowe.

## 5.14 Algorytm usuwania szumu z danych jednostki inercyjnej

Odczytuje nazwę nadajnika i odbiornika wiadomości typu `sensor_msgs/Imu` oraz wielkość bufora. Używany jest w przeprowadzeniu testów w sekcji 6.3. Program działa na zasadzie listy, do którego początku dodaje dane z nowych wiadomości, a z końca usuwa dane ze starych. Z każdą otrzymaną wiadomością, liczy średnią z aktualnie przechowywanych wyników. Na początku działania liczy średnią tylko z tej ilości danych, którą aktualnie posiada.

## 5.15 Obserwator symulacji

Wtyczka symulatora Gazebo, oblicza i zwraca ciąg danych, reprezentujący odległość i kąt pomiędzy modelem dynamiki, a kinematyki, wiadomość `omnivelman_msgs/Relative`. Te dane pozwalają sprawdzić, na ile symulacja fizyczna różni się względem matematycznego modelu.

Ten program nie może być zaimplementowany jako zewnętrzny pakiet, gdyż wiadomości zawierające pozycje platform będą przychodzić asynchronicznie. Nie da się w takim przypadku obliczyć dokładnych odległości pomiędzy platformami w danej chwili. Wprowadzałoby to także spore opóźnienie, gdyż program musiałby czekać na odbiór obu wiadomości, dodatkowo zachowując pewność że oba pochodzą z tego samego kroku symulacji.

Z punktu widzenia symulacji, program nie steruje modelem, a całą symulacją. Jest umieszczany w elemencie `word` pliku opisującego symulację, a nie w elemencie `model`. W związku z tym nie posiada symulowanej pozycji i prędkości. Jednakże, bazując na nazwie, może sterować dowolnym obiektem w przestrzeni symulacji. W związku z tym, na diagramie 6.1, został przedstawiony jako niezależny element uczestniczący w symulacji.

## 5.16 Model kinematyki odwrotnej

Zaimplementowany jest w standardowy sposób typu akcja-reakcja. Odbiera wiadomość z zadanimi prędkościami liniowymi i rotacją, a zwraca wiadomość z prędkościami kół.

Używa wzoru przedstawionego w sekcji 4.6.

## 5.17 Scena z symulacją

Pakiet dostarcza trzy pliki opisujące symulację.

Plik typu `world` jest plikiem SDF, podobnym do tych, które służą do określenia wewnętrznej budowy robotów. Posiada listę elementów `import` ze ścieżkami modeli, a także nazwy programów działających bez modelu, jak obserwator symulacji, opisany w sekcji 4.12.

Ten pakiet nie posiada kodu wykonywalnego.

## 5.18 Rozdzielacz wiadomości

Ten program wykonywalny pobiera i generuje wiadomości typu `omnivelm_msgs/Vels`, zawierające prędkości kół. Pozwala to na sterowanie kilkoma robotami o identycznym interfejsie ze wspólnego źródła. W szczególności przydaje się to przy rozdzielaniu wartości prędkości kół dla modelu platformy dynamicznej i kinematycznej.

Może być zastąpiony przez wbudowane narzędzie, lecz ten jest zaimplementowany w C++, a gotowe narzędzia korzystają z wolniejszego języka interpretowanego Python.

## 5.19 Prosty program sterujący

Program porusza robotem równolegle do osi lokalnego układu współrzędnych. Nie nadaje prędkości kątowej.

Przyjmuje dane ze skanerów laserowych, a periodycznie generuje zadane sterowanie platformie.

Dane ze skanerów laserowych dzielone są, w zależności od kąta pomiaru, na cztery ćwiartki lokalnego układu współrzędnych. Rozpatrywane są tylko te ćwiartki, w których kierunku porusza się platforma. Jeśli pomiar wypadnie wystarczająco blisko platformy, kierunek jest obracany w odwrotnym do tej ćwiartki kierunku.

Na przykład, jeśli platforma porusza się w prawo i wykryje obiekt w trzeciej ćwiartce (czyli po prawej stronie względem aktualnego kierunku poruszania się), to zacznie poruszać się prosto, aby uniknąć przeszkody.

Taki program gwarantuje omijanie przeszkód, aby platforma nie zderzyła się z jakimś obiektem.

## Rozdział 6

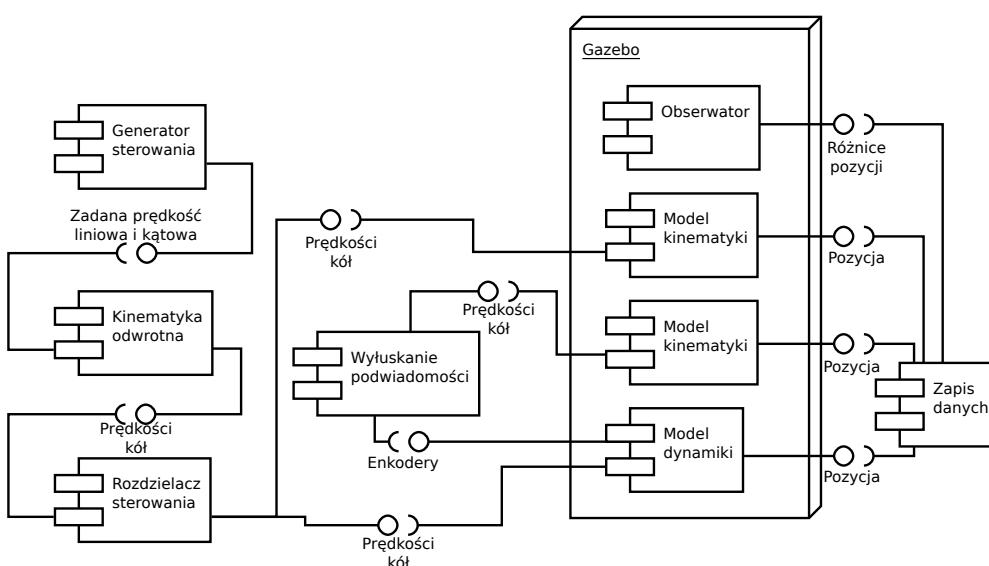
# Testy środowiska symulacyjnego

W tym rozdziale przedstawione są różne konfiguracje pakietów, wraz z wykresami ruchów platform, oraz wnioski płynące z tych zachowań.

Uruchomiono odpowiednią konfigurację pakietów i połączono je strumieniami konfiguracyjnymi. Wiadomości przesyłane w odpowiednich strumieniach zostały zapisane do pliku za pomocą narzędzia `rosbag`, dostępnego w środowisku ROS. Za pomocą narzędzia `rostopic`, wyeksportowano dane z plików do pliku rekordów, gdzie każde pole oddzielone jest przecinkiem (*Comma Separated Value(CSV)*). Używając skryptu w programie Gnuplot, narysowano wykresy w formacie PDF, które następnie załączono poniżej.

### 6.1 Weryfikacja działania modelu dynamiki

W tym teście porównano działanie modelu dynamiki oraz jej enkoderów w stosunku do zadanego sterowania. W tym celu uruchomiono scenę posiadającą dwa modele kinematyki i jeden dynamiki oraz obserwator symulacji, patrz rysunek 6.1. Jeden model kinematyki użyto do przekształcania zadanego prędkości kół do prędkości liniowej i kątowej platformy, którą to za pomocą symulatora scałkowano w celu określenia zadanej trasy robota. Drugi model, korzystając z enkoderów, realizował mechanikę odometrii, wyznaczał trasę na podstawie danych o obrotach kół. Wiadomości, zawierające pozycje modeli w czasie, były zapisywane do pliku. Dodatkowo, wtyczka symulatora, obserwator symulacji, generowała dane porównujące odległość i kąt obrotu modeli od siebie.



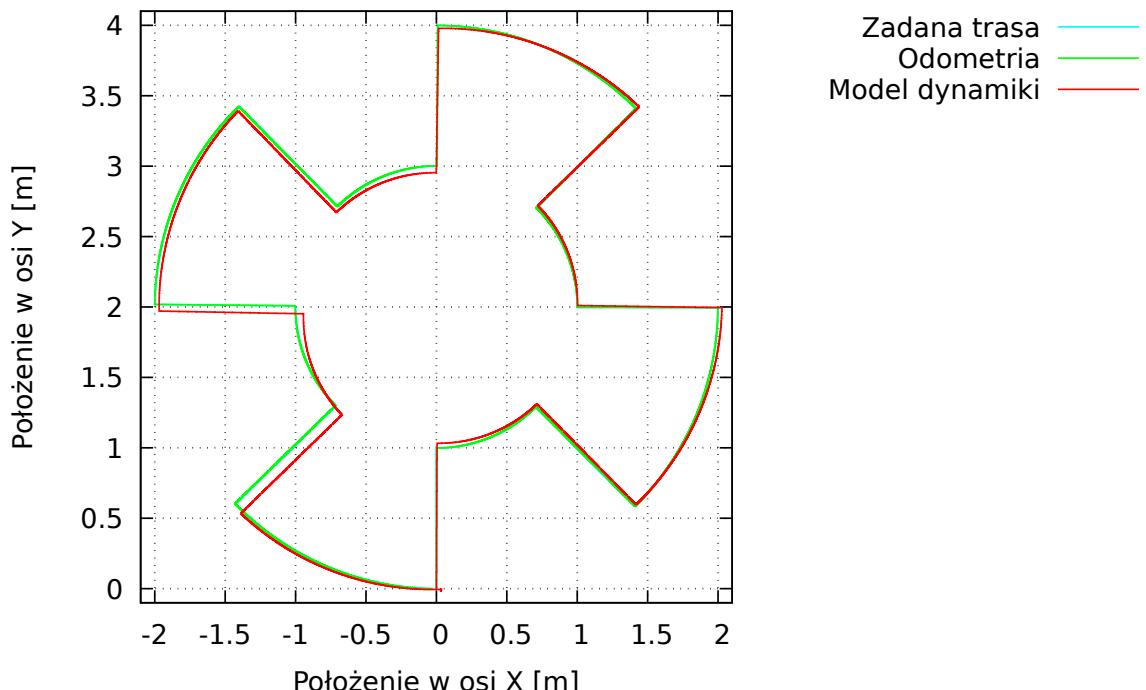
Rysunek 6.1: Połączenie pakietów w teście porównującym modele.

### 6.1.1 Porównanie modeli

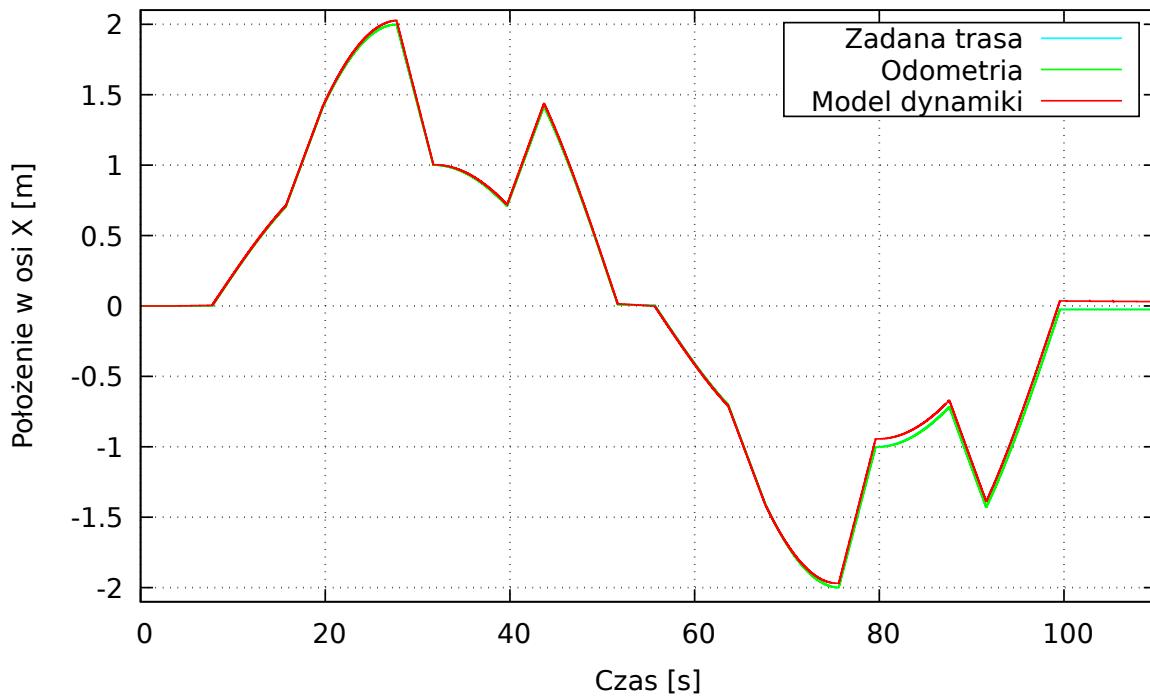
Za pomocą pakietu do nadawania sterowania, opisanego w sekcji 4.8, nadano trasę o kształcie przedstawionym na rysunku 6.2. W tym teście robot kolejno poruszał się:

1. W przód z prędkością  $0,25 \text{ m/s}$  przez  $5 \text{ s}$ .
  2. W prawo z prędkością  $\pi/32 \text{ m/s}$  oraz prędkością kątową o  $\pi/32 \text{ rad/s}$  wokół osi Z, przez  $8 \text{ s}$ .
  3. W tył z tą samą prędkością i przez ten sam czas, jak w punkcie 1.
  4. W prawo z prędkością  $\pi/16 \text{ m/s}$  oraz prędkością kątową  $\pi/32 \text{ rad/s}$  wokół osi Z, przez  $8 \text{ s}$ .

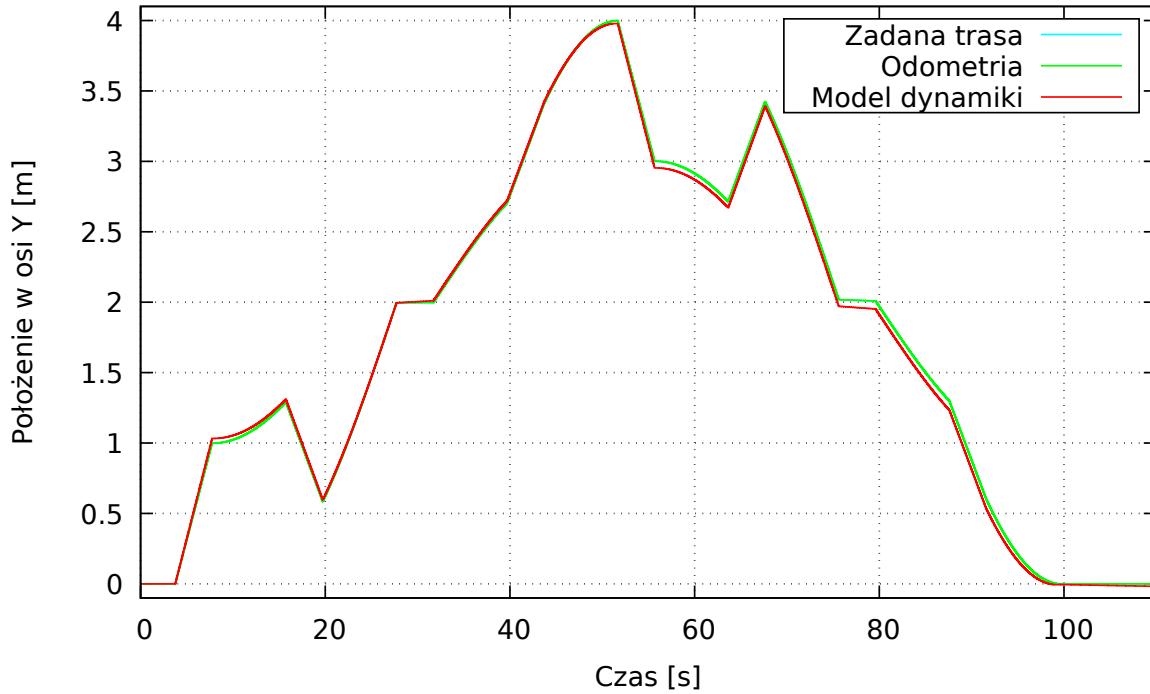
Powyższe akcje wykonał czterokrotnie.



Rysunek 6.2: Trasa modelu platformy w teście porównującym modele. Różnica między zadaną trasą, a trasą wyznaczoną przez enkodery, jest niewielka.



Rysunek 6.3: Składowa pozycji modelu wzdłuż osi X w czasie.



Rysunek 6.4: Składowa pozycji modelu wzdłuż osi Y w czasie.

### Dokładność odometrii

Trasa obliczana na podstawie prędkości kątowej kół, byłaby identyczna do zadanej trasy w przypadku gdyby koła obracały się dokładnie z zadanymi prędkościami kątowymi. To wymagałoby, aby silniki kół posiadały nieskończony moment siły, lub na tyle duży, aby opór obrotu, spowodowany tarciem,

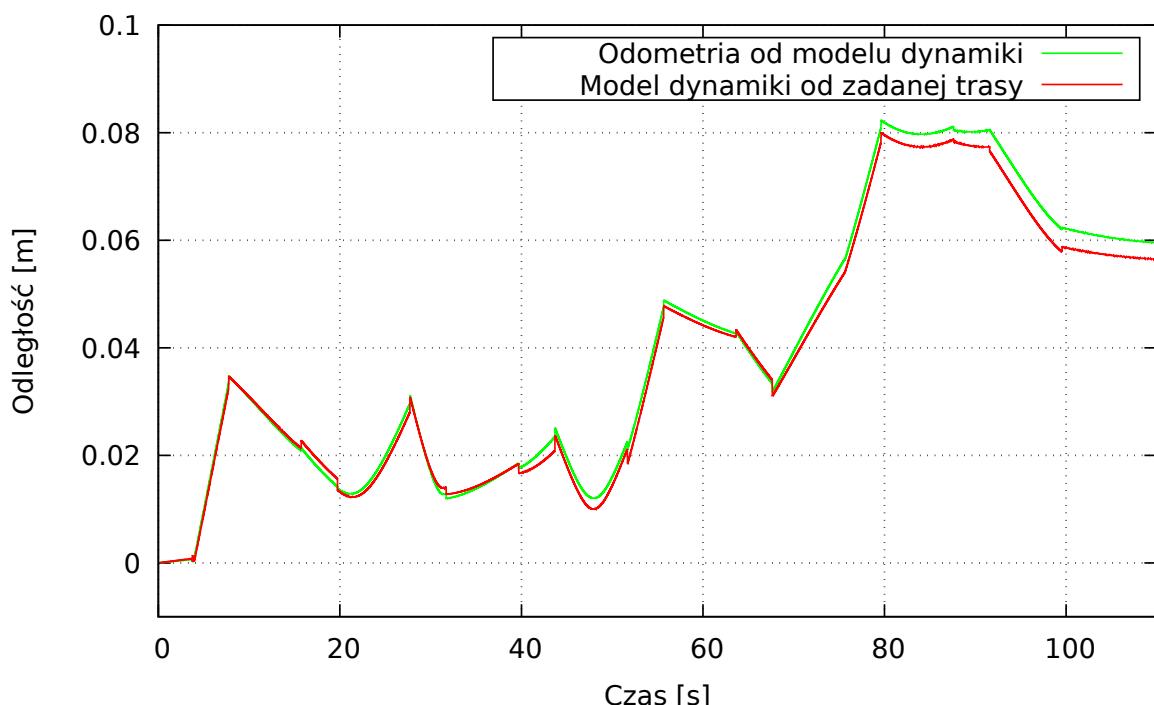
nie wpływał znacząco na ich prędkość kątową. Innymi słowy, nawet jeśli platforma uderzyłaby w przeszkode, koła nadal powinny obracać się zgodnie z zadanimi wartościami. To nie oznacza, że rzeczywista trasa robota jest zgodna z obliczoną z odometrii, gdyż uwzględnia także poślizgi kół, czego enkodery nie są w stanie wykryć.

Różnice w pozycji zadanej, a pozycji obliczonej za pomocą danych wygenerowanych przez enkodery biorą się z tego, że silniki mogą nie mieć wystarczającej mocy, aby nadać kołom odpowiednie prędkości i przeciwdziałać oporom. Ponieważ założono dużą moc silników, jak zostało to opisane w sekcji 2.4, modelowane koła są mniej podatne na opory, a co za tym idzie, ich prędkość będzie bardziej zbliżona do prędkości zadanej i także trasa wyznaczona w ten sposób z odometrii będzie niemalże pokrywać się z zadaną trasą.

Zauważalne różnice pomiędzy przebiegami trasy odometrycznej i zadanej pojawiają się przy wywoływaniu dużych przyspieszeń i poślizgów na modelu platformy.

### Trasa modelu dynamiki

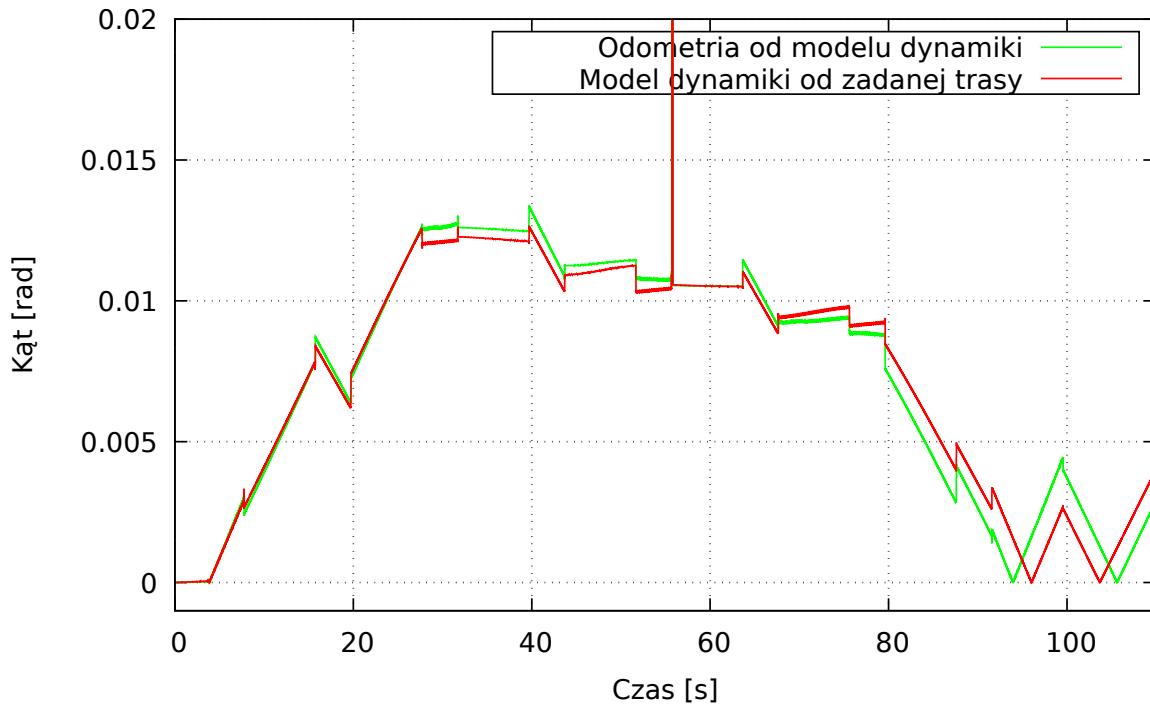
Różnica pomiędzy położeniem modelu dynamiki, a kinematyki (położenia zgodnego z zadaną trasą), rośnie w czasie. Jest tak na skutek błędów numerycznych maszyny do symulacji fizyki, braku systemu operacyjnego czasu rzeczywistego, braku zamodelowania wszystkich oporów i skomplikowanej budowy modelu. Model dynamiki uwzględnia współczynniki tarcia kół o podłoże i jest w stanie modelować ich poślizgi.



Rysunek 6.5: Różnica pomiędzy położeniem modelu dynamiki i modelem kinematyki modelującym zadaną pozycję i pozycję odometryczną.

### Zachowanie się modelu dynamiki

Na wykresie 6.5 zamieszczono różnice między położeniami odpowiednich modeli. Różnice nie zmieniają się monotonicznie ze względu na zmiany kierunków prędkości liniowej robota i jego prędkości kątowej wokół osi Z. Dlatego też, zdarzają się przypadki w których model dynamiki zbliża się do modelu kinematyki (zadanej pozycji) po tym gdy oddalił się od niej już wcześniej.



Rysunek 6.6: Różnica pomiędzy orientacją modelu dynamiki i modelem kinematyki modelującym zadaną pozycję i pozycję odometryczną.

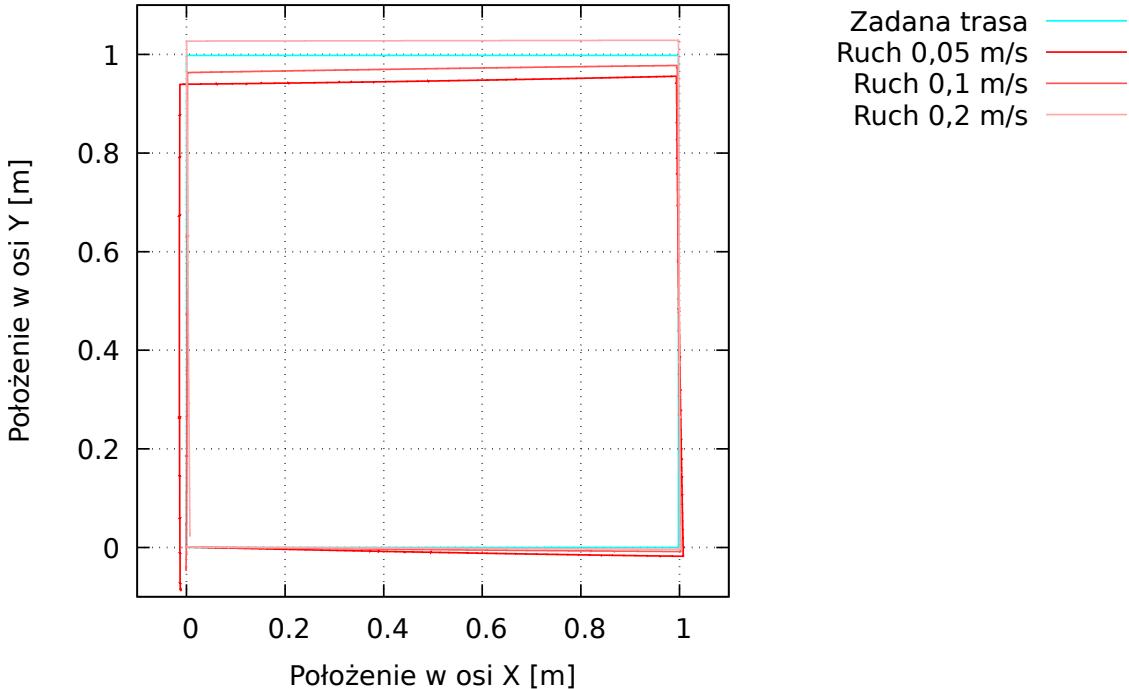
Różnica pomiędzy wykresami jest mała, gdyż na obie wartości wpływają poślizgi kół w trakcie pokonywania trasy. Enkodery są w stanie wykryć zwiększoną siłę tarcia kół o podłożu na skutek obliczenia różnicy zadanego obrotu, a faktycznego. Taka zwiększoną siłą tarcia towarzyszy nagłym zmianom prędkości platformy, tak samo jak poślizg, lecz to nie powoduje że na podstawie pomiarów z enkoderów da się bezpośrednio wyznaczyć wartości poślizgu. Inaczej mówiąc, przyspieszenia platformy powodują poślizgi i trudności w obrocie kołami na skutek sił tarcia, lecz korzystając z danych z enkoderów można wyznaczyć jedynie opory obrotu kół.

Przykładowe różnice pomiędzy wykresami występują na przykład w 15 sekundzie ruchu. W tym miejscu nastąpił nagły zwrot bazy, który spowodował tarcie, a co za tym idzie i zwiększyły opór ruchu obrotowego kół. Ten opór został wykryty przez enkodery, lecz poślizg nadal wpłynął na pozycję modelu dynamiki w większym stopniu. Dlatego też różnica między pozycją odometryczną, a pozycją modelu dynamiki wzrosła mniej, niż różnica pomiędzy pozycją modelu dynamiki, a pozycją wynikającą z zadanej trasy.

Na wykresie 6.6 są pokazane różnice pomiędzy kątem obrotu wokół osi Z tych modeli. Podobnie, jak w przypadku położenia, bazując na danych z enkoderów, można obliczyć zmiany kierunku ruchu platformy. Na zmianę różnicy kąta obrotu nie wpływają jedynie zmiany prędkości kątowej, a również losowy kąt obrotu nadany platformie w przypadku poślizgów. Można zauważyć, że różnica między orientacją modelu dynamiki, a orientacją wyliczoną na podstawie odometrii zwykle zmienia się mniej dynamicznie od różnicy między modelem dynamiki, a zadaną pozycją.

### 6.1.2 Powtarzalność testów

Zadano modelowi jazdę z różnymi prędkościami po trasie kwadratu, wyznaczona trasa została przedstawiona na rysunku 6.7.



Rysunek 6.7: Trasa modeli platformy, poruszających się z różną prędkością liniową po trasie kwadratu. W tym eksperymencie nie nadano prędkości kątowej.

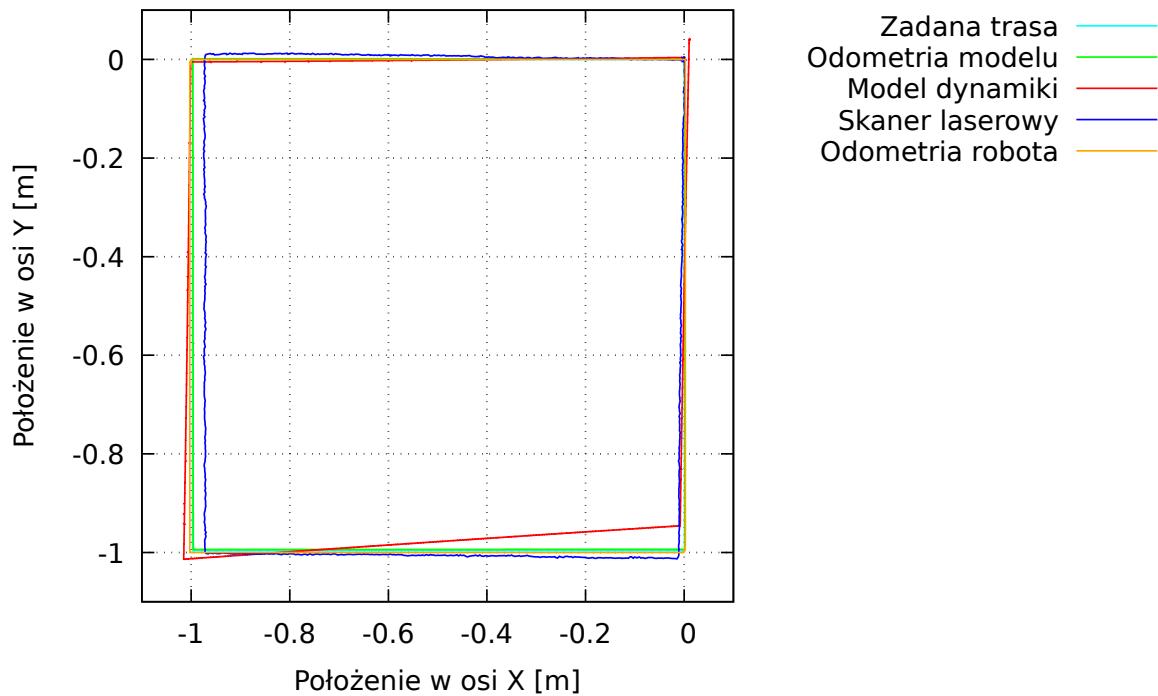
Eksperyment pokazuje, że zależnie od prędkości przejazdu, mogą występować niedokładności ruchu. Dzieje się to głównie w kierunku Y, gdyż ruch platformy w kierunku X jest zawsze podobny. Im wolniej platforma się porusza, tym mniejszą odległość pokonuje w odpowiednio dłuższym czasie. Tak jest na skutek wewnętrznego działania maszyny do symulacji fizyki. Potrzeba dokładniejszych testów, aby stwierdzić bezpośredni powód takiego zachowania.

Kilkukrotne wykonanie tego samego eksperymentu pokazuje, że różnica tras pomiędzy przejazdami nie jest aż tak duża.

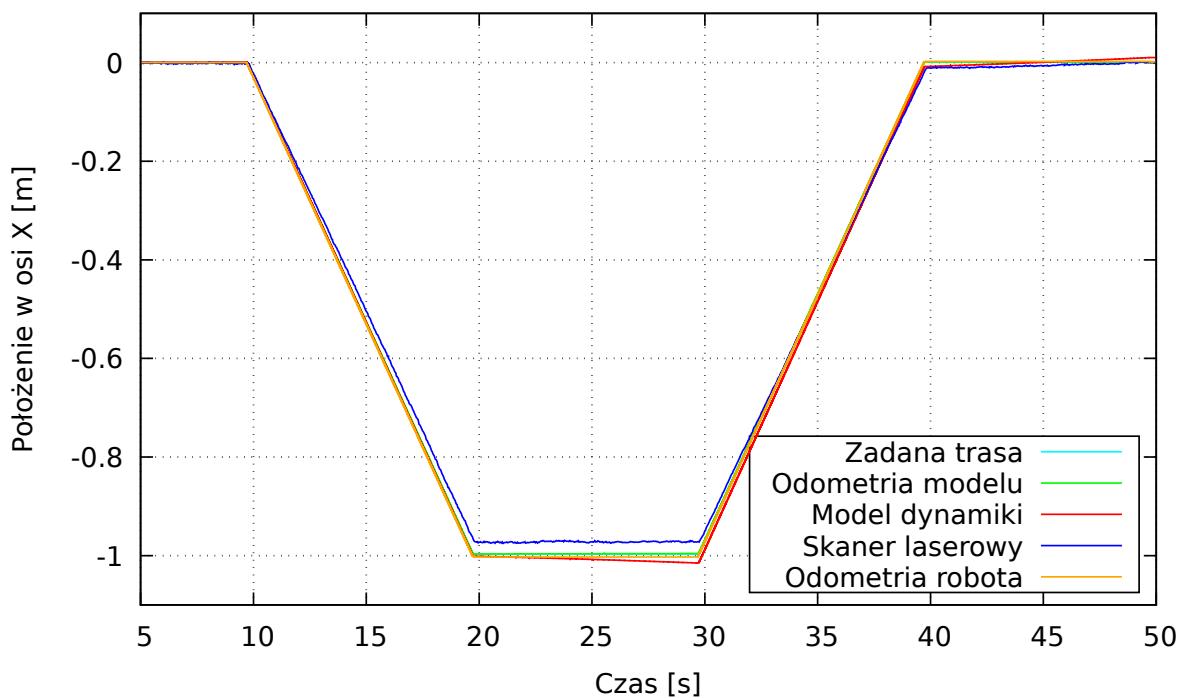
## 6.2 Porównanie modelu z robotem

### 6.2.1 Trasa bez rotacji

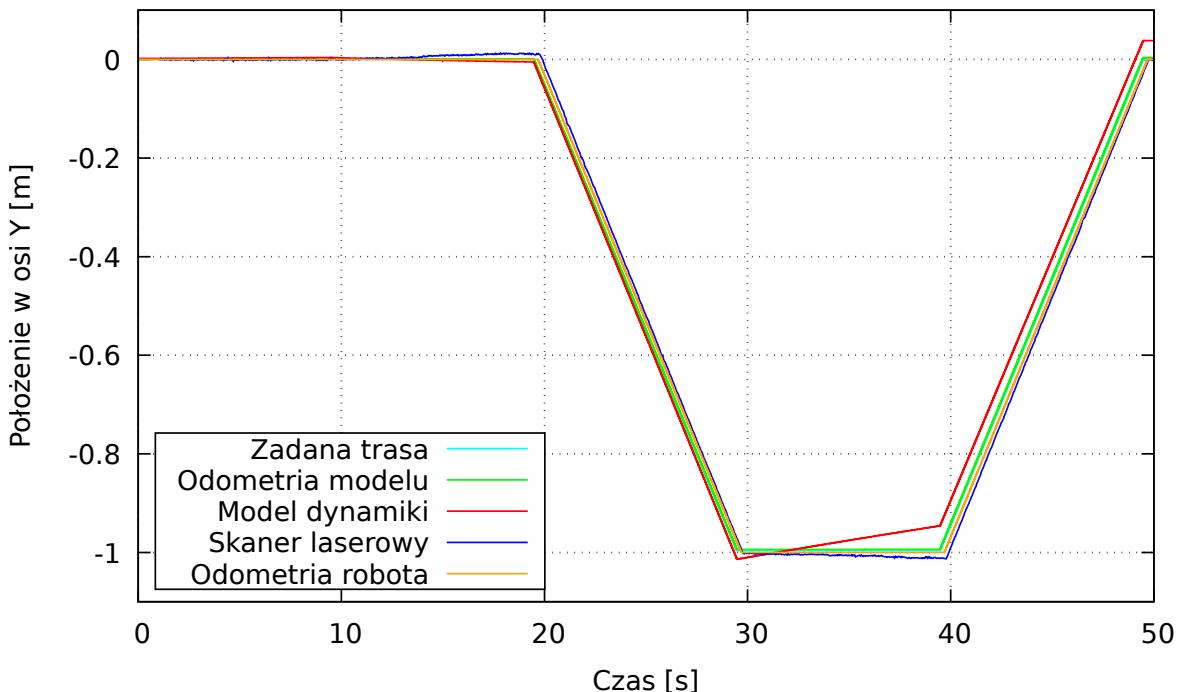
Za pomocą generatora sterowania, zadano platformie ruch po trasie kwadratu w prędkością 0,1 m/s, zaczynając od ruchu wzdłuż osi X. Zapisano dane generowane przez enkodery, obliczaną odometrię, pomiary skanerów laserowych oraz samo sterowanie. Następnie, używając tych samych danych, przeprowadzono identyczny eksperyment na modelu, łącząc pakiety w sposób pokazany na rysunku 6.1. Korzystając z pakietu `laser_scan_matcher`, obliczono rzeczywistą pozycję platformy, korzystając z danych skanerów laserowych. Określenie pozycji jest obarczone błędem, wynikającym z błędów pomiarowych czujników, stąd obliczona trasa nie składa się z odcinków i posiada szum. Trasy modeli i platformy pokazano na wykresie 6.8.



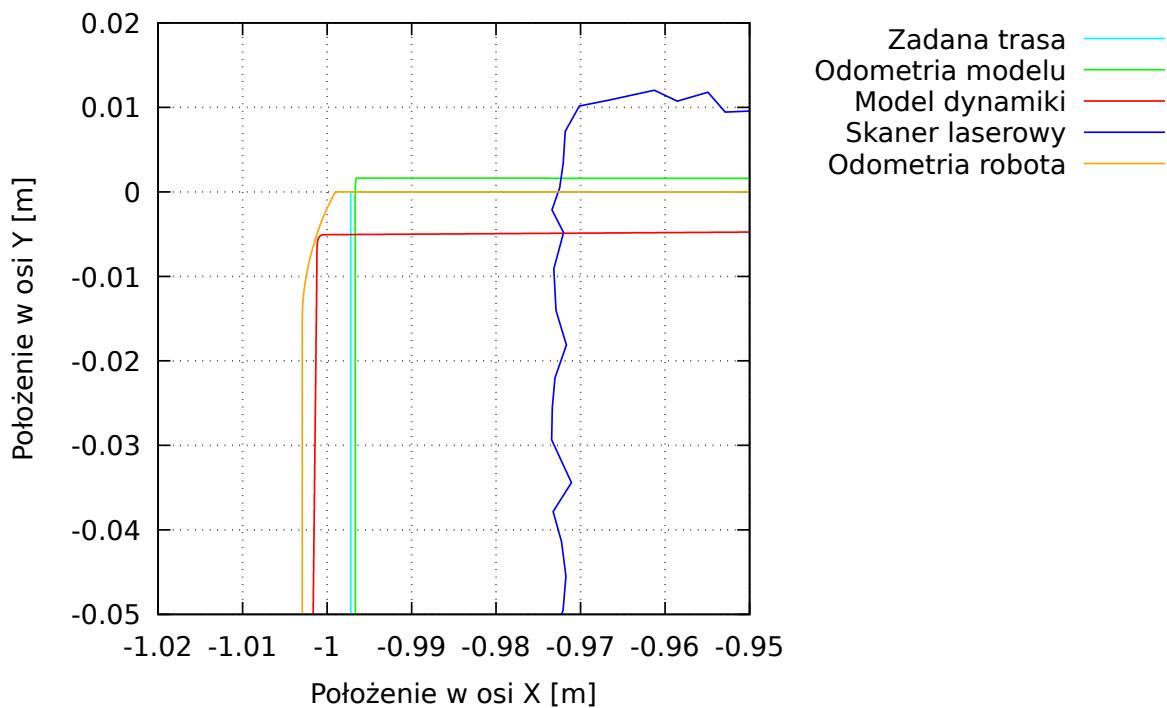
Rysunek 6.8: Trasa modeli robota przy jeździe po trasie kwadratu.



Rysunek 6.9: Składowna pozycji modeli wzdłuż osi X.



Rysunek 6.10: Składowa pozycji modeli wzdłuż osi Y.



Rysunek 6.11: Przybliżenie wykresu 6.8

Charakterystyczną cechą wykresu jest to, że robot przejechał mniejszą odległość wzdłuż osi X, niż wzdłuż osi Y. Może być to spowodowane różnicą w wyjściowym współczynniku tarcia kół w zależności od wyjściowego wektora prędkości kół. Zgodnie z cechą kół, opisaną w sekcji 2.2.1, rolki kół obracają się z największą prędkością przy ruchu w bok, a co za tym idzie, w ruchu ma udział także ich tarcie obrotowe. Przy ruchu w przód, rolki nie obracają się, więc i to dodatkowe tarcie nie wpływa na ruch

platformy. Dlatego też platforma pokonała mniejszą odległość w jednym z kierunków.

Model dynamiki doznał poślizgu w trakcie 30 sekundy ruchu, która nadała mu niespodziewaną zmianę orientacji, przez co różnica w odległości od zadanej trasy znacząco wzrosła.

### Nagła zmiana kierunku jazdy

Na wykresie 6.11 przedstawiono przybliżenie trasy modeli i robota w trakcie drugiego skrętu. Opisane niżej cechy są szczególnie dobrze widoczne na wektorowych wykresach w przypadku dokumentu elektronicznego.

**Zadana trasa** nie pokrywa się dokładnie z przewidzianą trasą kwadratu, następują jej nieznaczne odchylenia. Jest to spowodowane działaniem symulacji i programu generującego sterowanie na systemie operacyjnym, który nie był systemem czasu rzeczywistego. Dodatkowo, pakiety sieciowe, zawierające wiadomości ROSa, przesyłane były przez sieć Ethernet, która nie gwarantuje przesyłu w deterministycznym czasie. To powoduje, że obliczona trasa nie koniecznie pokrywa się z zadaną trasą. Co więcej, na skutek niedeterministycznego przekazywania wiadomości przez strumienie komunikacyjne w czasie symulacji, każdy model i platforma mogły otrzymać nieco inne sterowanie, a co za tym idzie, różnice pomiędzy ich położeniami mogły być spowodowane tym efektem.

**Odometria modelu** Jak wspomniano w sekcji 6.1.1, trasa obliczona na jej podstawie niemal pokrywa się z zadaną trasą. Jednakże, w trakcie nagłej zmiany kierunku ruchu, tarcie kół spowodowało nieznaczną różnicę prędkości kół w stosunku do zadanych wartości, co jest wykryte jako ścieśnięcie wykresu w kierunku Y. W przypadku modelu jest ono bardzo niewielkie, ale w tym samym kierunku, jak obliczone przez odometrię robota. To oznacza, że model enkoderów działa poprawnie, lecz być może jego parametry, czyli moment siły kół lub tarcie rolek nie są odpowiednio dobrane.

**Model dynamiki** wykazał bezwładność w trakcie ruchu, to znaczy, jego prędkość liniowa w kierunku osi Y nie wzrosła natychmiastowo, lecz pod wpływem przyspieszenia, tak samo prędkość w kierunku X, która także nie spadła natychmiast. Jest to widoczne jako zaokrąglenie na wykresie. Opóźnienie w kierunku X było większe, niż przyspieszenie w kierunku Y, tzn. model szybciej wyhamował składową X prędkości liniowej, niż nadał zadaną składową Y prędkości liniowej, co się objawia jako wydłużenie trasy w danym kącie. Takie samo zachowanie wykazały trasy odometrii.

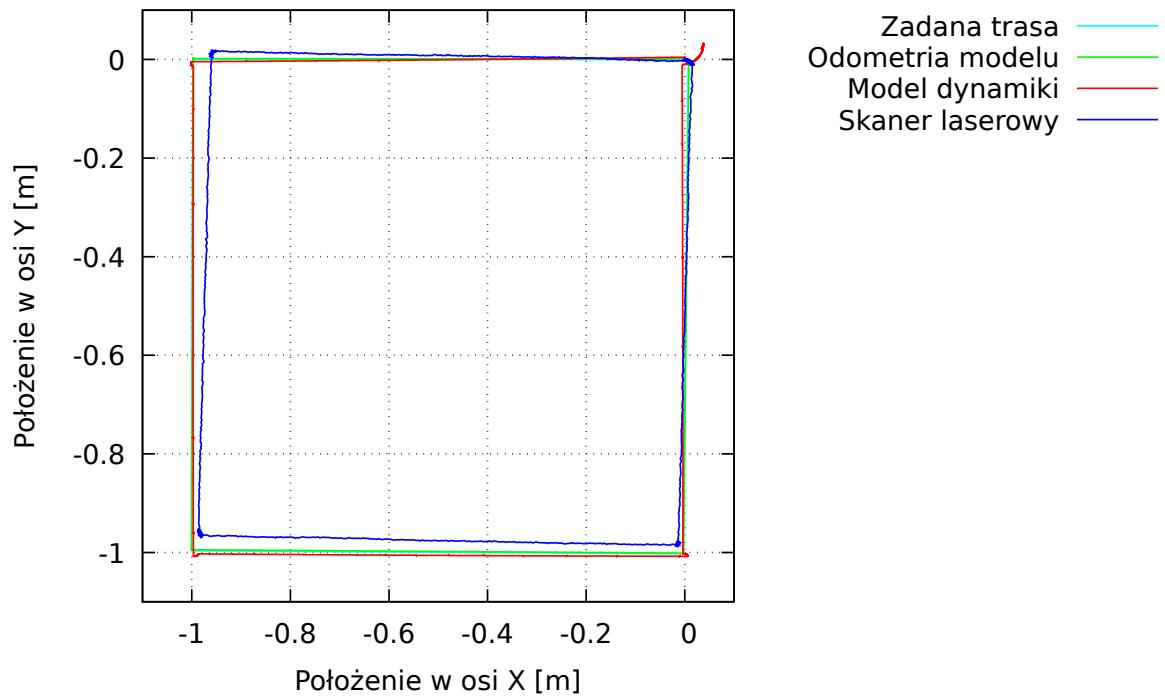
**Skaner laserowy** Ponieważ pomiar z tego czujnika jest obarczony dużym błędem oraz małą częstotliwością pomiarów, nie można jednoznacznie określić bezpośredniego zachowania platformy w trakcie skrętu.

**Odometria robota** Korzystając z enkoderów, program sterujący platformą obliczył jej pozycję. Nie jest to dokładna pozycja platformy, jaka jest określona przez skaner laserowy, lecz trasa wyznaczona tym sposobem jest zbliżona do zadanej trasy tak samo, jak trasa odometrii modelu. Wykazuje takie same własności w trakcie skrętu, jak jej model, tylko w większym stopniu. Różnica w przyspieszeniu i opóźnieniu robota wskazuje, że robot szybciej hamuje w kierunku X, niż przyspiesza w kierunku Y. Taki sam efekt występuje po drugiej stronie trasy, przy trzecim skręcie, lecz nie w drugim skręcie.

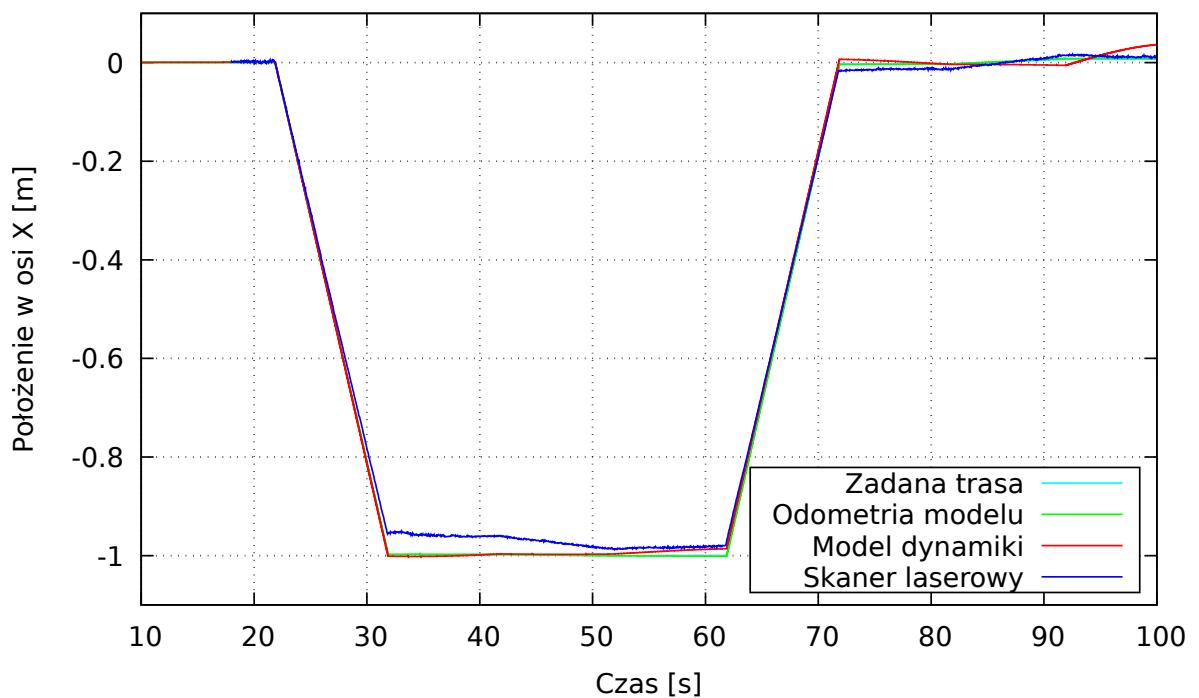
### 6.2.2 Trasa z rotacją

Sterowanie robotem odbyło się w sposób podobny do testu 6.2.1, z tą różnicą, że przy wcześniejszej zmianie kierunku prędkości liniowej obracano platformą o  $90^\circ$  wokół osi Z. To znaczy, że platforma poruszała się w prawo z prędkością 0,1 m/s, a następnie obróciła o  $90^\circ$  z prędkością kątową  $\pi/20$  rad/s i ponownie poruszała się w kierunku -X lokalnego układu współrzędnych. Trasa takiego przejazdu

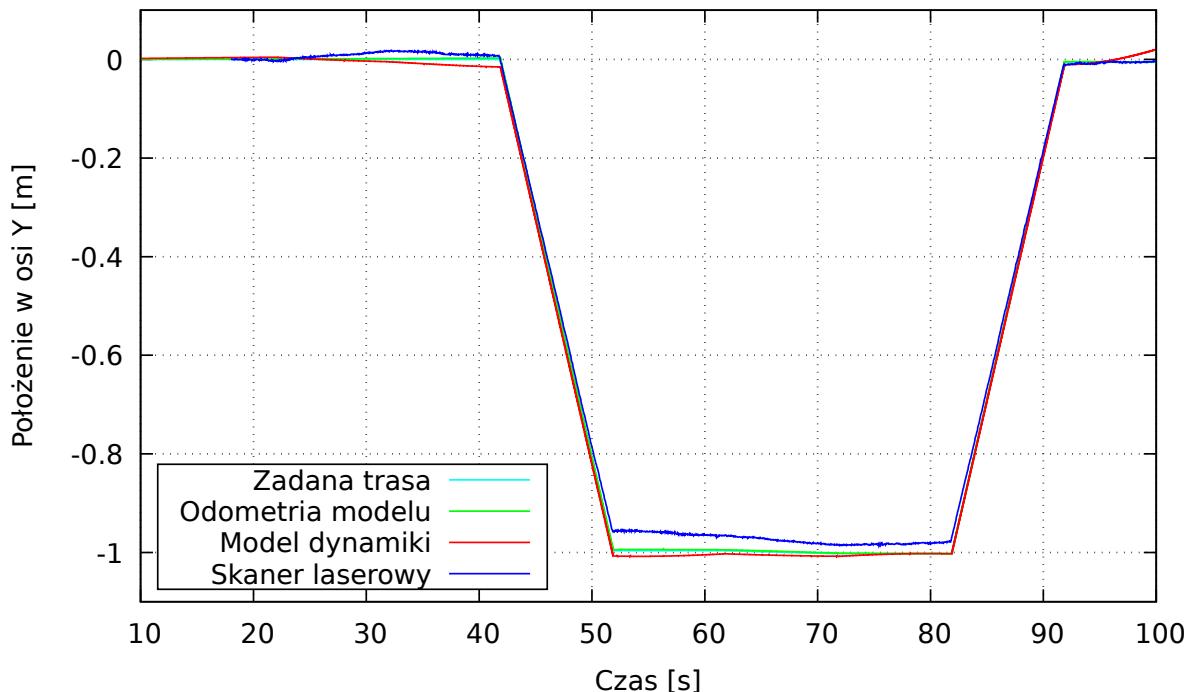
została przedstawiona na wykresie 6.12. Można na niej zauważyć poślizg przy ruszaniu platformy, który nadał jej losowy kąt obrotu.



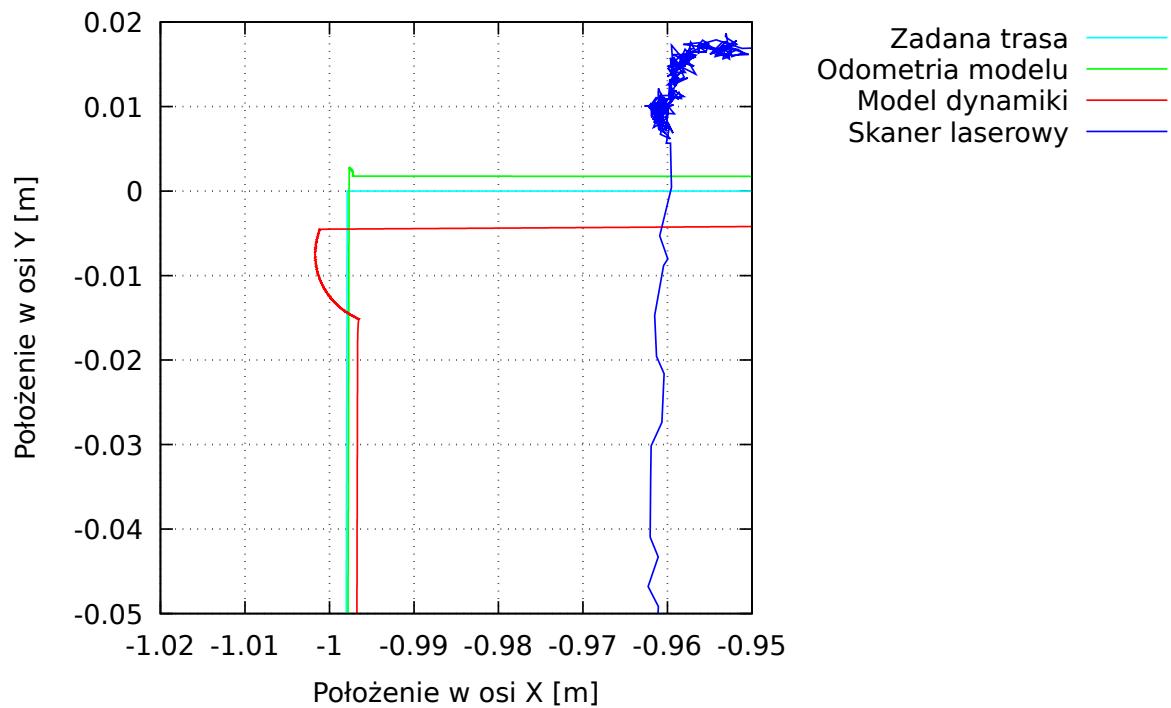
Rysunek 6.12: Trasa modeli robota przy jeździe po trasie kwadratu z nadaniem prędkości kątowej wokół osi Z.



Rysunek 6.13: Składająca się pozycji modeli wzdłuż osi X.



Rysunek 6.14: Składowa pozycji modeli wzdłuż osi Y.



Rysunek 6.15: Przybliżenie wykresu 6.14

W tym przypadku nie ma informacji o odometrii wygenerowanej przez robota, gdyż nie obsługuje ona poprawnie rotacji platformy.

Na przybliżeniu wykresu (rysunek 6.15) widać moment jak model dojechał do odpowiedniej pozycji i zaczął się obracać wokół osi Z. Wykres pokazuje, że obrót następował wokół punktu znajdującego się o około 0,5 cm na prawo od środka modelu platformy. Nie jest wiadome, dlaczego tak się stało, gdyż

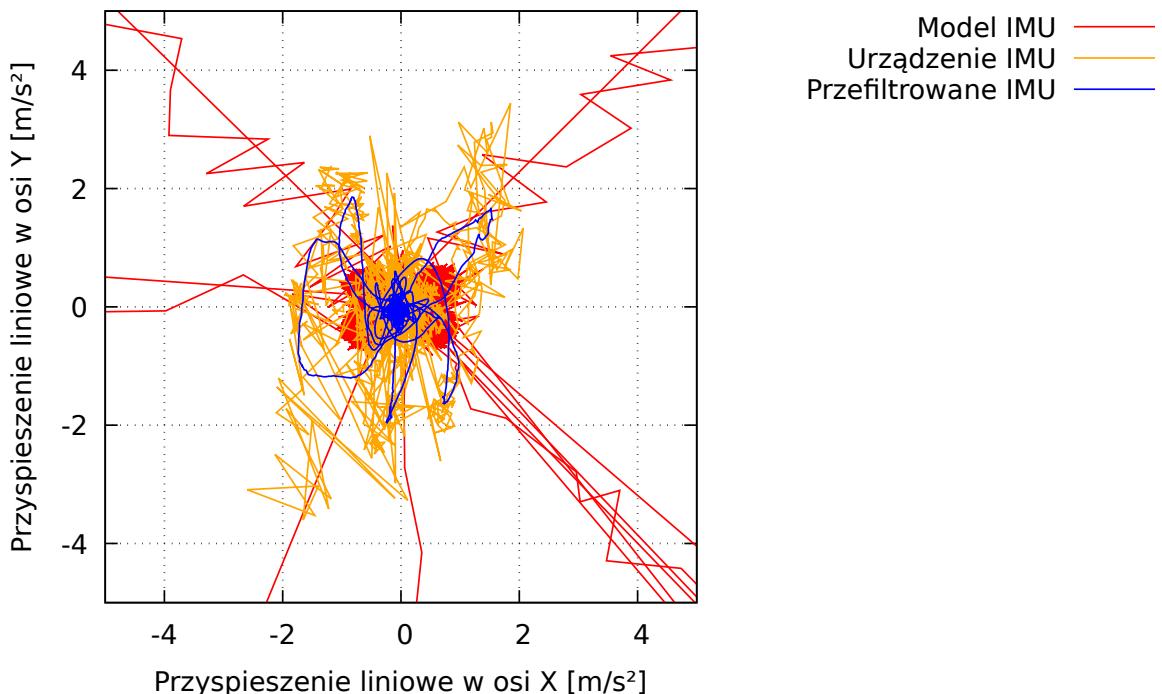
model jest zdefiniowany symetrycznie, a środek układu współrzędnych, względem którego rysowany jest wykres, znajduje się na środku platformy. Według enkoderów, punkt obrotu znajdował się z kolei na lewo od środka platformy. Takie zachowanie może być spowodowane na przykład asymetrią siatki definiującej kształt platformy, kolejnością obliczeń wykonywanych przy symulowaniu ogniw obiektu lub innymi własnościami maszyny do symulacji fizyki.

## 6.3 Jednostka inercyjna

W trakcie powyższych testów zabrano również dane wygenerowane przez jednostkę inercyjną oraz jej model.

### 6.3.1 Akcelerometr

Ten czujnik zmierzył przyspieszenia modelu w trakcie testu 6.2.1.



Rysunek 6.16: Porównanie odczytów pomiaru przyspieszenia liniowego jednostki inercyjnej, jej modelu i filtru redukującego szum.

W trakcie testu następowały kolejne przyspieszenia i opóźnienia bazy w zadanych kierunkach, to widać jako impulsy na wykresie 6.16. Na początku platforma zaczęła poruszać się w kierunku -X i w tą stronę zostało nadane pierwsze przyspieszenie. Nie jest znana jego dokładna wielkość, gdyż zadany kierunek prędkości zmienił się w ciągu jednej wiadomości.

Drugi impuls został wygenerowany w trakcie pierwszego skrętu bazy. Zadziałało przyspieszenie zmieniające kierunek prędkości liniowej platformy o  $90^\circ$ , co oznacza że impuls jest obrócony o  $45^\circ$  względem układu współrzędnych. Impuls wystąpił w kierunku (X,-Y). Jest to wypadkowa przyspieszenia bazy w kierunku -Y i opóźnienia w kierunku poprzedniego kierunku jazdy, czyli -X.

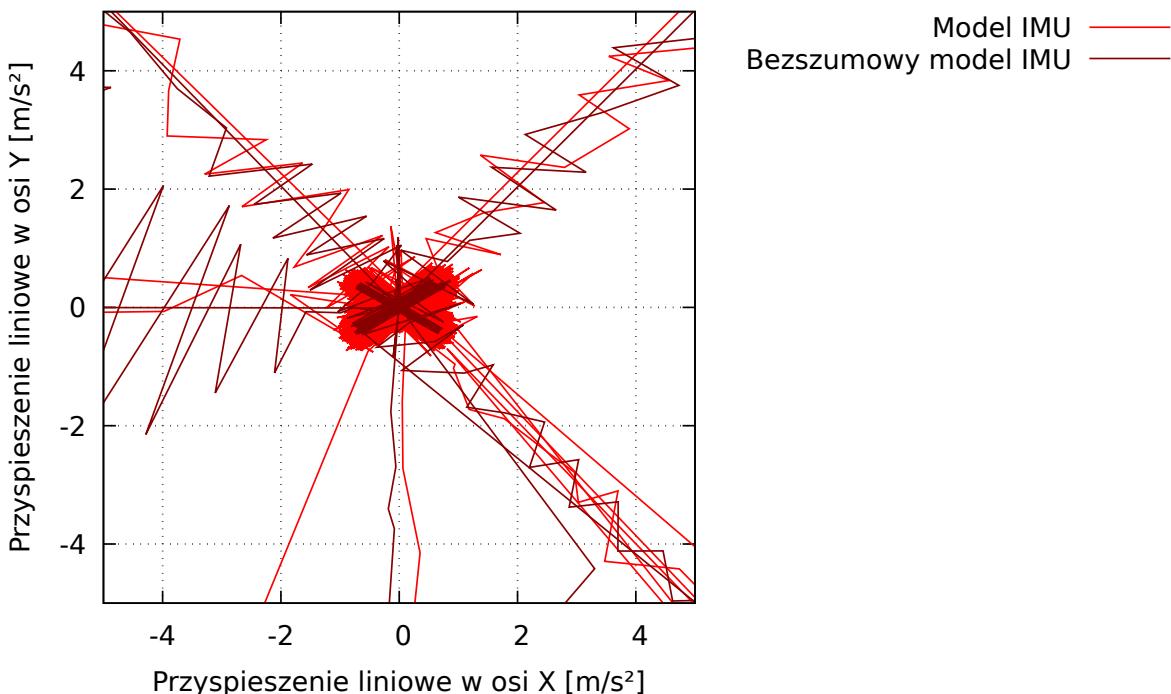
Podobna zmiana prędkości nastąpiła jeszcze dwa razy, generując kolejne, symetryczne impulsy.

Na koniec platforma zatrzymała się, generując opóźnienie w kierunku Y, co także widać na wykresie w postaci przyspieszenia w przeciwnym kierunku.

Porównując dane wygenerowane przez jednostkę inercyjną nie widać zależności, jednak zastosowanie prostego uśrednienia pakietem opisany w sekcji 4.11 powoduje, że impulsy wygenerowane przez robota również są widoczne. Każda uśredniona wartość jest średnią z 40 poprzednich.

Po przefiltrowaniu danych z robota, można zaobserwować jak każdy impuls ma kształt półksiężyca, na początku każdej zmiany prędkości opóźnienie ma kierunek równoległy do wektora prędkości, zatrzymując robota, następnie wolniejsze przyspieszenie nadaje nowy kierunek, a potem następuje reakcja korpusu i gaszące drgania w nowym kierunku. Zamodelowanie tej własności mogłoby okazać się bardzo trudne, prawdopodobnie należałyby nadać dodatkowe ogniva do robota i połączyć je więzami sprężynowymi.

Szum, jakimi są obarczone zamodelowane dane spowodowany jest potrzebą różniczkowania prędkości liniowej modelu, obliczanej dyskretnie przez maszynę do symulacji fizyki. Wartości mas i momentów bezwładności ogniw modelu mocno wpływają na kształt impulsów, korelację osi i szum na wykresie. Nie jest technologicznie możliwe zamodelowanie jednostki inercyjnej pozbawionej szumu ze względu na błędy numeryczne, algorytmy użyte w maszynie symulującej fizykę i system operacyjny na którym pracuje symulator.

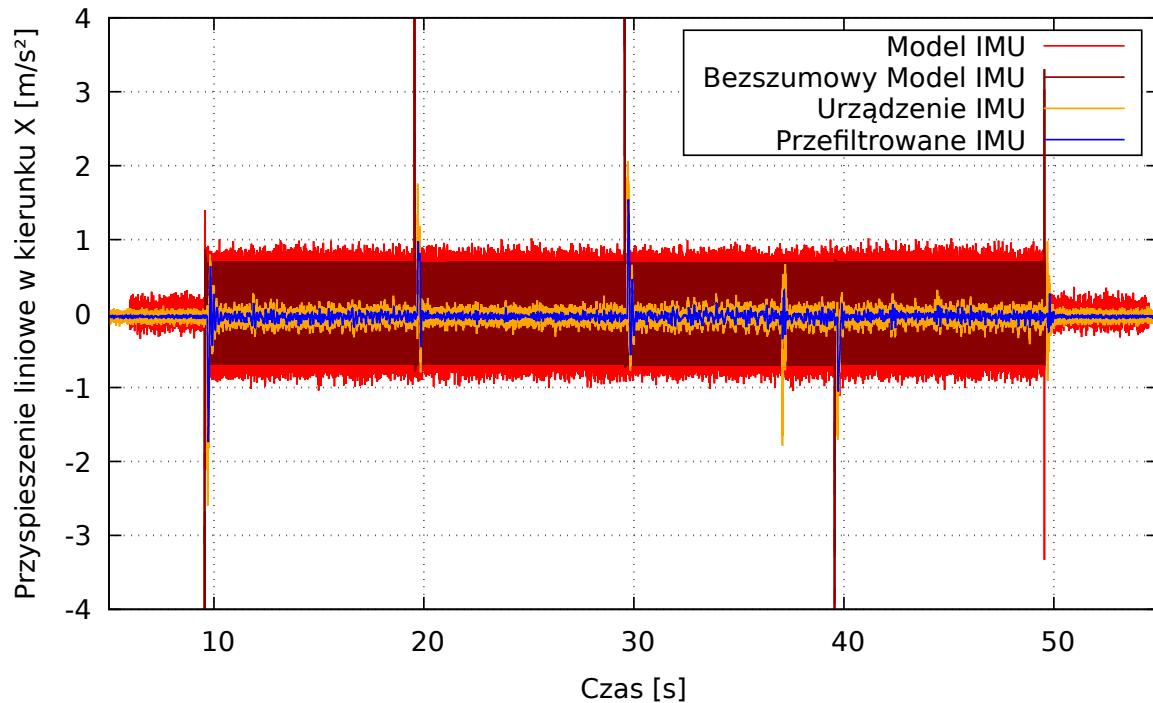


Rysunek 6.17: Porównanie odczytów pomiaru przyspieszenia liniowego modeli jednostki inercyjnej z dodanym szumem i bez.

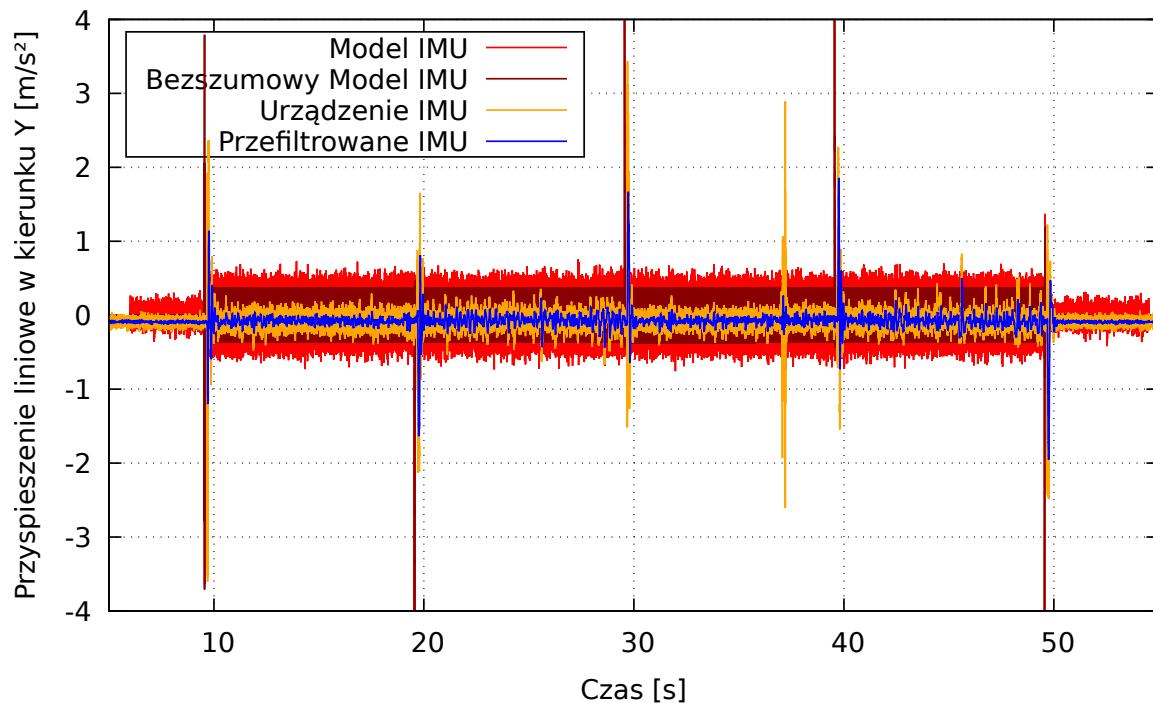
Na wykresie 6.16 widać szum generowany przez jednostkę inercji w trakcie ruchu jednostajnego. Jest on mniejszy od niedokładności generowanych przez model. Jednakże, w trakcie działania przyspieszenia, model jednostki inercyjnej wygenerował większy impuls, niż urządzenie. Do wykrycia kierunków impulsów nie potrzebna jest filtracja danych, ale taka filtracja okazała się niezbędna w przypadku danych zebranych z jednostki inercyjnej.

Aby przybliżyć zachowanie się modelu jednostki inercyjnej w trakcie ruchu z prędkością jednostajną, do odczytów urządzenia, dodano dodatkowy szum o rozkładzie normalnym. Jego parametry określone są w tabeli 5.4. Na rysunku 6.17 przedstawiono porównanie generowanych przez model danych z szumem i bez. Można zauważyć silną korelację modelu jednostki inercyjnej w zależności od kierunku prędkości liniowej robota. Korelacja jest także silnie uzależniona od parametrów modelu dynamiki. Występują bazowe drgania w dwóch kierunkach. Niestety, amplituda bazowych drgań w modelowanym czujniku jest większa niż amplituda szumu rzeczywistej jednostki, w dodatku występuje korelacja

wzdłuż dwóch prostych. Widać to na wykresach przebiegu czasowego 6.19 i 6.18.



Rysunek 6.18: Porównanie składowej X pomiarów przyspieszenia liniowego jednostki inercyjnej, jej modelu i filtra redukującego szum.



Rysunek 6.19: Porównanie składowej Y pomiarów przyspieszenia liniowego jednostki inercyjnej, jej modelu i filtra redukującego szum.

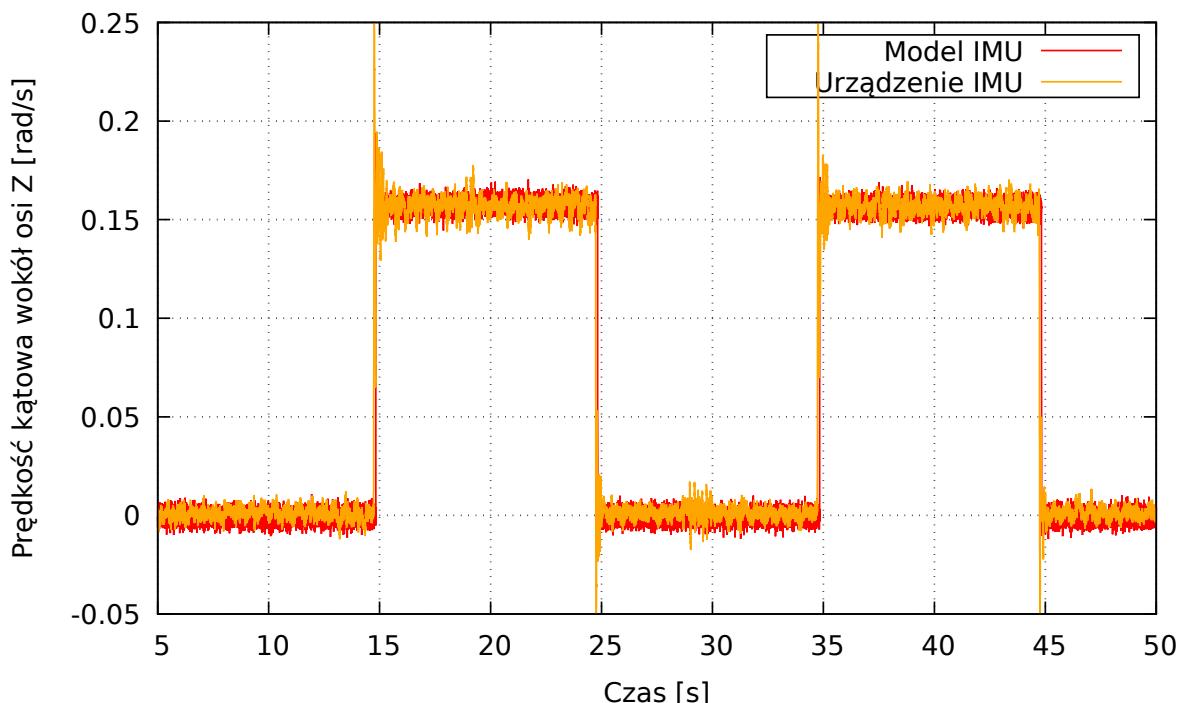
Aby użyć tego czujnika do obliczania pozycji, trzeba wpierw użyć bardziej zaawansowanego algo-

rytmu odszumiającego i wyprowadzić eksperymentalnie zależności pomiędzy czujnikami osi (macierz kowariancji).

Na wykresach 6.18 i 6.19 widać jak w około 15 sekundzie ruchu na platformę zadziałała nagła zmiana prędkości liniowej. Wykres przebiegu trasy nie wskazuje, jakoby zdarzyła się tam nagła zmiana prędkości.

### 6.3.2 Żyroskop

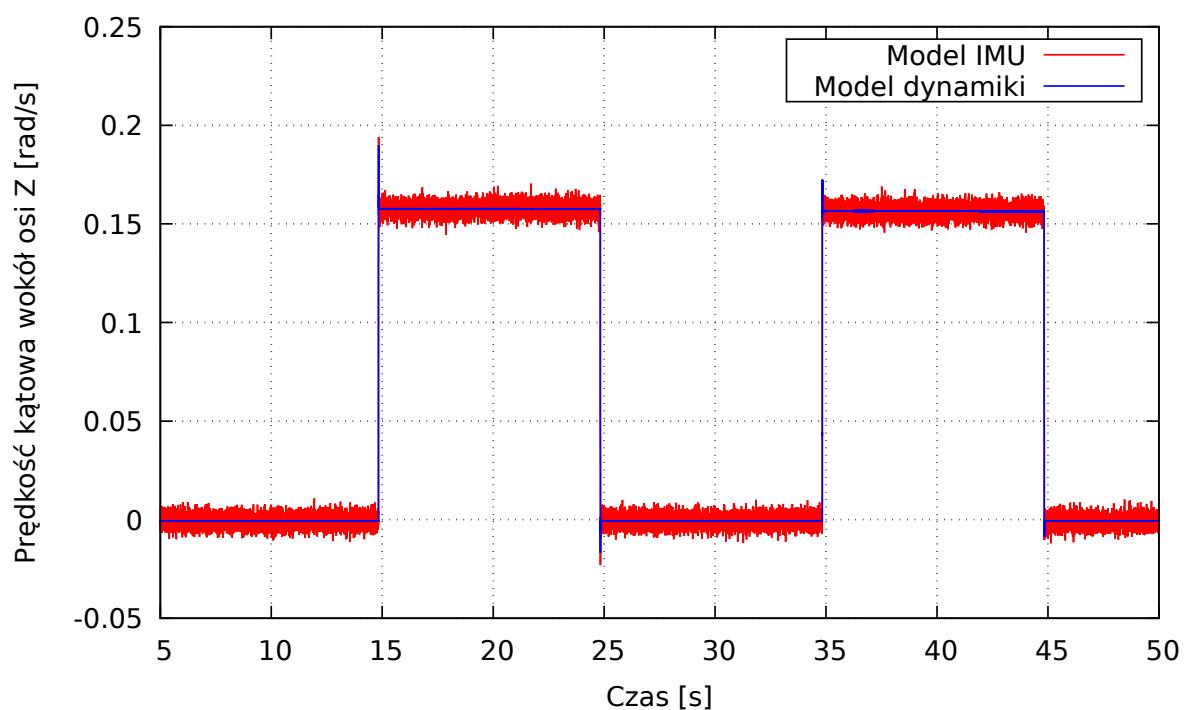
Czujnik prędkości kątowej korzysta z żyroskopu i zwraca prędkość kątową we wszystkich trzech osiach. Ponieważ jednak platforma porusza się po płaskim terenie, wymagany jest jedynie czujnik mierzący obrót wokół osi Z, czyli w góre. Drugą osią wykresu może być zatem czas nadania pakietu. Te dane zostały zebrane w trakcie testu 6.2.2.



Rysunek 6.20: Porównanie prędkości kątowej wokół osi Z, wygenerowanej przez jednostkę inercyjną i jej model.

Ponieważ maszyna do symulacji fizyki wewnętrznie posiada informację o prędkościach obiektów, model tego czujnika po prostu zwraca gotowe dane. To powoduje, że praktycznie pozabawiony jest szumu, co można zauważyć na wykresie 6.21. Aby zatem polepszyć jakość symulacji, dodano sztuczny szum do generowanych danych, zgodnie z tabelą 5.4. W ten sposób wykresy są do siebie bardziej zbliżone.

Jednak niektórych własności czujnika nie da się zamodelować tak łatwo. Po pierwsze, szum czujnika nie do końca ma rozkład normalny. Z wykresu można zobaczyć, że dane prawdopodobnie korelują z częstotliwościami obrotu kół. Dodatkowo, nagła zmiana prędkości kątowej powoduje zauważalny skok na wykresie, co w symulowanych danych jest znacznie mniej widoczne. Może być to spowodowane drganiem robota przy nagłych zmianach prędkości kątowej, co nie jest uwzględnione w modelu.



Rysunek 6.21: Porównanie danych wygenerowanych przez model jednostki inercyjnej i prędkości kątowej wokół osi Z modelu.

## Rozdział 7

# Podsumowanie

Stworzono modele platform oraz czujników, a także system wielu innych pakietów usprawniających testowanie i sterowanie robotem.

Eksperymenty przeprowadzone na modelu i platformie pokazały, że błędy lokalizacji w modelu mogą być nawet większe, niż w rzeczywistym robocie. Z punktu widzenia testowania programu symulacyjnego jest to przydatne, gdyż program sterujący, przetestowany na symulatorze, na pewno poprawnie określi sterowanie dla robota o mniejszych błędach ruchu. Model kinematyki posiada bardzo wiele parametrów działania, nie tylko w kwestii mas ogniw i ich momentów bezwładności, ale także w kwestii obsługi maszyny symulacyjnej fizyki. Istnieje kilka sposobów na nadawanie prędkości kątowej kołom. Aby znaleźć najodpowiedniejszy, należy przeprowadzić czasochłonne badania nad działaniem każdego z nich, również biorąc pod uwagę sposób działania maszyny do symulacji. Sama maszyna może być modyfikowana w celu lepszego zamodelowania bazy (gdyż ma otwarty kod), lub zastąpiona inną maszyną do symulacji fizyki.

Testy pokazały również, jak wiele różnych i czasami nielogicznych zachowań występuje w modelach i robocie. Sama poprawna interpretacja wszystkich tych cech wykresów wymaga dogłębnego zbadania działania robota i maszyny do symulacji fizyki.

Ponieważ model posiada uproszczone modele kół Mecanum, niektóre ich cechy (jak na przykład opór obrotowy rolek) nie mogą być zamodelowane lub też istnieje jakiś nietypowy sposób (na przykład nieznaczna zmiana kierunku wektora siły tarcia) na zamodelowanie takich własności.

Jednostka inercyjna wykrywa własności robota, nieistniejące w modelu, na przykład drgania powstałe przy nagłej zmianie prędkości platformy. Próby wprowadzania takich własności do modelu mogą nie być możliwe lub też wymagać kolejnych badań, na przykład w kwestii dodania sprężystości do niektórych więzów. Istnieje także minimalny szum, zależny w dużym stopniu od ustawień parametrów modelu.

Model skanera laserowego jest bardzo prosty w działaniu, zatem jego ewentualny rozwój nie będzie aż tak skomplikowany, jak innych czujników.

Pakiet pomocnicze i skrypty uruchamiające okazały się przydatne w sterowaniu platformą, jednak niektóre były nadto skomplikowane i niepotrzebne. Pakiet rozdzielania sterowania platformy może być zastąpiony przez kilkukrotne wywołanie wbudowanego pakietu. Program do ręcznego sterowania jest nadto skomplikowany a i tak nie był używany do sterowania robotem, ani do przeprowadzania testów z rozdziału 6. Stanowią jednak one dobre wspomaganie do późniejszego rozwijania modelu w celu odpowiedniego ustawienia parametrów, aby jak najdokładniej przybliżyć jej działanie do platformy.

# Bibliografia

- [1] P. Muir, C. Neuman, "Kinematic modeling for feedback control of an omnidirectional wheeled mobile robot", Proceedings, IEEE International Conference in Robotics and Automation, Vol 4. pp. 1772-1778, 1987.
- [2] M. O. Tătar, C. Popovici, D. Mândru, I. Ardelean, A. Plesă, "Design and development of an autonomous omni-directional mobile robot with Mecanum wheels", Conference: 2014 IEEE International Conference on Automation, Quality and Testing, Robotics, Cluj-Napoca, 2014, pp. 1-6. DOI: 10.1109/AQTR.2014.6857869.
- [3] M. Lamy, "Mechanical development of an automated guided vehicle", Master of Science Thesis MMK 2016:153 MKN 171, KTH Industrial Engineering and Management, Machine Design.
- [4] A. Gfrerrer, "Geometry and kinematics of the Mecanum wheel", Computer Aided Geometric Design, 25(9):784-791.
- [5] L. Xie, C. Scheifele, W. Xu, K. A. Stol, "Heavy-Duty Omni-Directional Mecanum-Wheeled Robot for Autonomous Navigation", 2015 IEEE International Conference on Mechatronics (ICM), Nagoya, 2015, pp. 256-261. DOI: 10.1109/ICMECH.2015.7083984.
- [6] V. Kálmán, "On modeling and control of omnidirectional wheels", PhD. dissertation, Budapest University of Technology and Economics, Department of Control Engineering and Information Technology, Budapest 2013.
- [7] J.B. Song, K.S. Byun, "Design and Control of a Four-Wheeled Omnidirectional Mobile Robot with Steerable Omnidirectional Wheels", Journal of Robotic Systems, 21(4):193-208, Kwiecień 2004.
- [8] I. Doroftei, V. Grosu, V. Spinu, "Omnidirectional Mobile Robot – Design and Implementation", Bioinspiration and Robotics Walking and Climbing Robots, M. K. Habib (Ed.), ISBN: 978-3-902613-15-8, InTech, 2007.
- [9] V. Kálmán, "Omnidirectional Wheel Simulation — a Practical Approach", Acta Technica Jauriensis, 6(2):73-90, 2013.
- [10] Strona internetowa symulatora Gazebo.  
<http://gazebosim.org>
- [11] Strona internetowa symulatora V-Rep.  
<http://coppeliarobotics.com>
- [12] Strona internetowa programowej struktury ramowej *Robot Operating System*.  
<http://www.ros.org>
- [13] Strona internetowa standardu SDF.  
<http://sdformat.org/spec>

- [14] Strona internetowa SICK, producenta czujników laserowych.  
<https://www.sick.com/de/en/detection-and-ranging-solutions/2d-lidar-sensors/lms1xx/lms100-10000/p/p109841>
- [15] Strona internetowa sprzedawcy czujnika inercji.  
<https://www.digikey.com/product-detail/en/analog-devices-inc/ADIS16460AMLZ/ADIS16460AMLZ-ND/5957823>
- [16] Dokumentacja ODE z wyjaśnieniem działania kolizji.  
[http://ode-wiki.org/wiki/index.php?title=Manual:\\_Joint\\_Types\\_and\\_Functions#Contact](http://ode-wiki.org/wiki/index.php?title=Manual:_Joint_Types_and_Functions#Contact)
- [17] Strona internetowa producenta kół Mecanum, użytych w platformie.  
<http://www.andymark.com/8in-Mecanum-HD-Set-p/am-2118.htm>
- [18] Strona internetowa robota Kuka YouBot.  
<http://www.youbot-store.com>