



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, 15 stycznia 2018

## **Streszczenie**

Ta praca opisuje projektowanie i budowę 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 ma służyć do wstępnych badań algorytmów planowania ruchu i sterowania robotem mobilnym.

Rozpatrzone są tutaj wymagania i problemy przy tworzeniu każdego ze składników środowiska. Na system składają się wirtualne efektory i receptorzy obsługujące odpowiednią maszynę symulacyjną, a także komponenty wspomagające simulację.

# Spis treści

<b>1 Wstęp</b>	<b>4</b>
1.1 Cel . . . . .	4
1.2 Składniki systemu . . . . .	5
1.2.1 Model 3D . . . . .	6
1.2.2 Sterownik silników . . . . .	7
1.2.3 Sterownik czujników . . . . .	8
1.2.4 Program sterujący . . . . .	9
1.3 Plan pracy . . . . .	9
1.4 Istniejące implementacje . . . . .	10
1.5 Podział tej pracy na sekcje . . . . .	11
<b>2 Narzędzia</b>	<b>12</b>
2.0.1 ROS . . . . .	12
2.0.2 Gazebo . . . . .	14
2.0.3 V-Rep . . . . .	15
2.0.4 Narzędzia . . . . .	16
<b>3 Baza</b>	<b>18</b>
3.1 Dookólna platforma mobilna . . . . .	18
3.2 Koła szwedzkie . . . . .	22
3.2.1 Działanie platformy . . . . .	24
3.3 Czujnik laserowy . . . . .	26
3.3.1 Zasada działania . . . . .	26
3.3.2 Komunikacja . . . . .	27
3.3.3 Podstawowe cechy . . . . .	27
3.4 Czujnik inercji . . . . .	29
<b>4 Model platformy</b>	<b>30</b>
4.1 Sposób zapisu w formacie SDF . . . . .	31
4.2 Model kinematyczny . . . . .	33
4.2.1 Problemy implementacji . . . . .	34

4.2.2	Komunikacja . . . . .	34
4.2.3	Zachowanie . . . . .	35
4.3	Model dynamiczny . . . . .	35
4.3.1	Jak największe zbliżenie do oryginału . . . . .	36
4.3.2	Resetowanie pozycji koła . . . . .	36
4.3.3	Zmiana osi rolki . . . . .	38
4.3.4	Modyfikacja kierunków i wartości wektorów tarcia . . . . .	38
4.3.5	Komunikacja . . . . .	41
4.3.6	Rozszerzenie modelu . . . . .	42
4.4	Model czujnika laserowego . . . . .	42
4.4.1	Obliczenia symulatora . . . . .	43
4.4.2	Różnice między czujnikiem, a modelem . . . . .	43
4.4.3	Komunikacja . . . . .	44
4.4.4	Model w Gazebo . . . . .	45
4.4.5	Błędy . . . . .	48
4.5	Model czujnika inercji . . . . .	50
<b>5</b>	<b>Komponenty systemu</b>	<b>52</b>
5.1	Manualne sterowanie . . . . .	54
5.1.1	Program . . . . .	54
5.1.2	Komunikacja . . . . .	55
5.1.3	Tryby działania . . . . .	55
5.2	Generator sterowania . . . . .	58
5.3	Wyłuskanie struktury wiadomości . . . . .	59
5.4	Podłoga o zmiennym współczynniku tarcia . . . . .	60
5.5	Algorytm usuwania szumu z danych modelu czujnika inercji . . . . .	60
5.6	Obserwator symulacji . . . . .	61
5.7	Model kinematyki odwrotnej . . . . .	61
5.8	Mapa z symulacją . . . . .	62
5.9	Rozdzielacz pakietów . . . . .	62
5.10	Prosty program sterujący . . . . .	63
5.11	Struktury pakietów wiadomości . . . . .	63
5.12	Zewnętrzne pakiety . . . . .	63
5.12.1	Rysownik wykresów . . . . .	63
5.12.2	Wizualizer pomiarów . . . . .	64
5.12.3	Zbieranie danych . . . . .	64
<b>6</b>	<b>Testy systemu</b>	<b>65</b>
6.1	Porównanie modeli dynamiki i kinematyki . . . . .	65
6.1.1	Ruch po kwadracie . . . . .	66
6.1.2	Ruch po „rozecie” . . . . .	67

6.1.3	Powtarzalność testów . . . . .	68
6.2	Enkodery . . . . .	70
6.2.1	Ciągłe nadawanie prędkości kół . . . . .	71
6.2.2	Impulsowe nadawanie prędkości kół . . . . .	71
6.3	Czujnik inercji . . . . .	72
6.3.1	Czujnik prędkości kątowej . . . . .	73
6.3.2	Czujnik przyspieszenia liniowego . . . . .	74

# Rozdział 1

## Wstęp

### 1.1 Cel

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 był sterowany tak samo, jak rzeczywisty robot. To spowoduje, że możliwe będzie stworzenie jednego wspólnego programu sterującego, do użycia zarówno w symulacji, jak i rzeczywistym robocie.

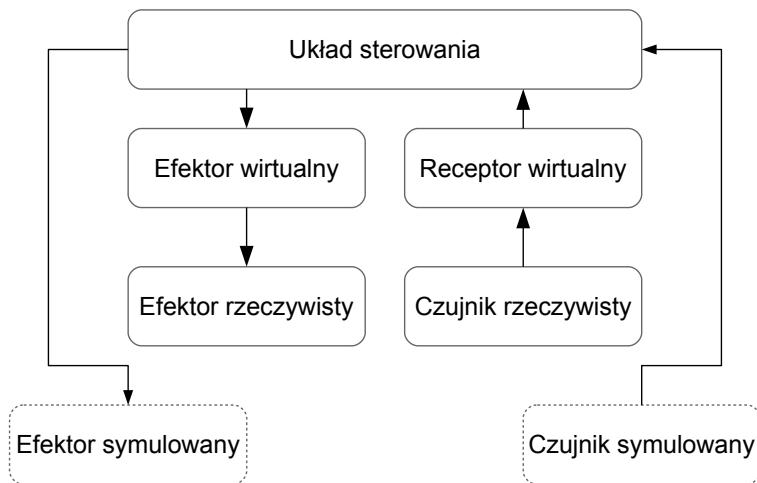
Testowanie oprogramowania sterującego na rzeczywistym obiekcie może prowadzić do jego uszkodzeń, dlatego wpierw należy się upewnić o poprawności projektowanych rozwiązań na bezpiecznym modelu wirtualnym. Rzeczywistość nie pozwala także na skomplikowane scenariusze testów, które w rzeczywistości mogłyby być niemożliwe do wykonania lub koszty jego wykonania byłyby zbyt wysokie. Szybciej i taniej jest stworzyć symulacyjne środowisko testowe, niż fizyczne, w dodatku błąd sterowników przy symulacji nie grozi zniszczeniem rzeczywistego robota. Dopiero przy osiągnięciu satysfakcjonującej jakości sterowania w symulacji wirtualnej, 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 Składniki systemu

Środowisko symulacyjne składa się z kilku odrębnych modułów, które komunikują się ze sobą poprzez specjalne interfejsy, wykorzystujące kolejki wiadomości. Taka implementacja komunikacji pozwala zmieniać i reimplementować poszczególne elementy i używać różnych języków programowania, zachowując jednolitą komunikację między składnikami i nie tracąc na kompatybilności między komponentami. Możliwe jest także przesyłanie wiadomości przez sieć, co pozwala na rozproszenie systemu.



Rysunek 1.1: Struktura agenta upustociowanego.

Można to przedstawić za pomocą zapisu agentowego, rysunek 1.1. 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 serwomotor, 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, 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 fotografii 3.4), 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 pozycje i prędkości 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 dokładnej pozycji, rotacji i prędkości w kartezjańskim układzie współrzędnych. Czujniki typu GPS, lub żyroskopy nie generują tak dokładnych pomiarów.

### 1.2.1 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 elementów składowych, a także wszystkie tarcia. Model obejmuje więzy na ruchome elementy, takie jak koła i rolki, aby umożliwić symulację przegubów.

Model składa się z elementów, odwzorowujących rzeczywiste części składowe bazy mobilnej. Elementy posiadają 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 wiezów o jednym stopniu swobody (zawias), używany przy połączeniu przedniej i tylnej części platformy, oraz jako piasty kół i rolek. Można także uznać, że więzy bez stopni swobody używane są do trwałego połączenia czujników z platformą i transportowanym robotem. Więzy mogą oddziaływać siłą na elementy do których są podłączone, symulując silniki.

Elementy składowe 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 w podobny sposób, jak ma to miejsce w rzeczywistości.

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 ozdoby, można wykorzystać istniejący model CAD do stworzenia siatki trójwymiarowej i nadania symulowanemu obiekowi wyglądu zbliżonego do fizycznego robota.

### 1.2.2 Sterownik silników

Program sterujący generuje abstrakcyjne dane, na przykład liczbę zmienno-przecinkową, zapisaną binarnie. Przykładowy silnik fizyczny nie jest w stanie działać na podstawie takich danych, do pracy potrzebuje odpowiedniego napięcia na wejściu, ale interfejs serwomotoru na przykład przyjmuje bardziej abstrakcyjne dane w formie liczby stałoprzecinkowej, jako pole w odebranej ramce. Do tłumaczenia jednych danych na drugie, potrzebny jest sterownik niskopoziomowy. Najczęściej implementowany jest w formie mikrokontrolera, lub podobnego systemu wbudowanego.

Jego zadanie to odczytanie danych, podanych przez program sterujący i na przykład generowanie na ich podstawie odpowiedniego przebiegu PWM,

lub obsługa przetwornika cyfrowo-analogowego. Do innych zadań może należeć kontrola, czy żądana wartość nie uszkodzi urządzenia. Zazwyczaj sterownik może komunikować się z powrotem z resztą systemu, aby zgłaszać ewentualne awarie.

Taki program i powiązany z nim układ elektroniczny są najczęściej dostarczone przez producenta robota i nieznane użytkownikowi. Dodatkowo, tworzy to kolejną warstwę abstrakcyjną dla sterownika głównego, który nie musi zważyć na generowanie różnych danych dla różnych modeli tych samych efektorów.

W środowisku wirtualnym należy stworzyć moduł o podobnym działaniu. Powinien przyjmować dane w dokładnie takim samym formacie, jak opisany wyżej układ, aby był łatwo wymienialny na sterownik fizycznego urządzenia bez ingerencji w główny program sterujący. Zamiast zamieniać odczytane dane na analogowe wartości, on wywołuje odpowiednie funkcje maszyny symulacyjnej, aby wywołać taki sam efekt, co na rzeczywistym efektorze, lecz w wirtualnej przestrzeni symulacji. Jako argumenty podaje parametry fizyczne symulowanego obiektu, oraz przyłożone siły.

### 1.2.3 Sterownik czujników

Implementowany podobnie do sterownika silników, ma za zadanie konwertować surowe i obarczone błędami dane z czujników, na format zrozumiały dla programu sterującego. W tym miejscu usuwa się błędy grube, niweluje stałe na podstawie kalibracji, wygładza szum i interpretuje dane, aby pozyskać wymagane przez wyższe warstwy informacje.

Przykładowo, czujniki laserowe zwracają jedynie ciąg pomiarów, ale to do tego programu należy interpretacja wykrytych kształtów, łączenie punktów i obróbka do formatu zrozumiałego dla wyższych podzespołów. Większość zaawansowanych receptorów posiada owe układy cyfrowe i programy wbudowane w urządzenie. Dostarczone są przez producenta tak samo, jak sterowniki efektorów.

Aby zasymulować ten element, należy zbudować program generujący dane na podstawie aktualnego stanu maszyny do symulacji, w sposób w jaki działa czujnik w rzeczywistości. Na przykład, dla czujnika laserowego, silnik symulacji fizycznej emitemuje odpowiednią ilość promieni i oblicza ich punkty przecięcia się z wirtualnymi modelami. Renderowanie obrazu pozwala na symulację kamery.

Ponieważ dane fizyczne nigdy nie są idealne, w celu przybliżenia wyjścia wirtualnego czujnika do oryginału, dodaje się szum o odpowiednim rozkładzie i błędy.

#### **1.2.4 Program sterujący**

W programie sterującym obliczane jest sterowanie, na podstawie dostarczonych odczytów z czujników. Zazwyczaj wykorzystuje się tutaj także zewnętrzne biblioteki, dostarczające zaawansowane algorytmy. Ich zadania mogą polegać na budowie wewnętrznej mapy, wyznaczaniu ścieżki, omijaniu przeszkód, odwrotnej kinematyce i tym podobnych.

Taki program zwykle działa na mocniejszych układach logicznych, niż sterowniki, ze względu na duże zapotrzebowanie na moc obliczeniową i nie-deterministyczny czas obliczeń. Jeśli robot komunikuje się z użytkownikiem, to zachodzi to w tym module.

Programy sterujące mogą być implementowane w językach wysokopoziomowych, nawet skryptowych, gdyż wymagania czasowe nie są rygorystyczne. Co więcej, często się zdarza, że odpowiednie składowe programu bazują na różnych technologiach.

Środowisko symulacyjne powinno zapewnić pełną abstrakcję komunikacji tego modułu. Oznacza to, że niezależnie, czy program steruje rzeczywistym robotem, czy symulacją wirtualną, zawsze powinien móc komunikować się i otrzymywać dane w tym samym formacie. W idealnym przypadku program nie powinien mieć możliwości stwierdzić, czy steruje symulacją, czy fizycznym urządzeniem.

### **1.3 Plan pracy**

1. Należy stworzyć model platformy do uruchomienia w symulatorze, zachowując wszystkie rozmiary i momenty rzeczywistej wersji. Bryły składowe modelu muszą przypominać kształtem części z których składa się robot, należy im także ustawić parametry fizyczne, jak masę, moment bezwładności, materiał itp.
2. Zamodelowanie wszystkich więzów na koła, rolki i przegub, aby maszyna symulacyjna poprawnie symulowała obiekt. Taki model powinien na tym stanie poprawnie reagować na wirtualne siły, lecz jego efektory nie będą jeszcze aktywne. Można go prosto побieżnie przetestować działając siłą na elementy i patrząc, czy reagują w spodziewany sposób.
3. Zapisanie wtyczki sterującej modelem, odczytującą odpowiednie dane z zewnątrz i wywołującą 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.

4. Stworzenie modelu kinematycznego, aby przetestować, czy model dynamiczny zachowuje się w miarę poprawnie.
5. Zaprogramowanie wtyczki symulującej podstawowe czujniki, aby generowały dane z enkoderów, oraz innych urządzeń, dodawały błędy pomiarowe, a następnie przekształcały dane na format zrozumiały dla programu sterującego. Czujniki nie muszą być istniejące, mogą generować dane, jak pozycja i rotacja, bardzo trudne do uzyskania rzeczywistymi czujnikami.
6. Wystawienie do zmiany w czasie rzeczywistym masy, momentu bezwładności, współczynników tarcia, aby pozwolić na proste testowanie działania systemu z różnymi współczynnikami.
7. Elementy pomagające w symulacji, jak model kinematyki odwrotnej, sterowany funkcją matematyczną, i podłoż ze zmiennym współczynnikiem tarcia.
8. Programy pomocnicze, zbierające i wyświetlające dane, interfejs graficzny do prostego sterowania robotem.
9. Model czujnika laserowego, powinien zbierać dane i przekazywać je dalej. Musi posiadać także określone kształty.
10. Model czujnika inercji, może być w pełni zaimplementowany jako program.
11. Uproszczony program sterujący, aby zbadać, czy system działa poprawnie na tyle, aby rozwinąć go w końcowy projekt.
12. Program sterujący w ROS. Największy i najbardziej skomplikowany element, wspólny dla obu bytów — wirtualnego i rzeczywistego. Zwykle nie jest to praca jednego człowieka, a jego rozwój nie ustaje przez długi czas. Ten program dostarczy funkcji, aby wyższy sterownik robota mógł użyć tego modułu do sterowania jazdą i odczytywania danych.

## 1.4 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, które są dobrymi kandydatami do użycia w projekcie. Tylko w

przypadku V-Repa, istnieje wstępny sterownik do którego da się wysyłać odpowiednie wartości kierunku, a on nadaje takie prędkości kołom, aby poruszać modelem w zadanym kierunku. Wersja dla Gazebo jest statycznym obiektem z błędnie ustanowionymi przegubami, jego efektory nie są zaimplementowane.

Dodatkowo, V-Rep posiada wbudowane dwa inne pojazdy o napędach kół Mecanum i czujnikach laserowych. Zewnętrzne modele także pomogą przy wstępnej weryfikacji zachowania się budowanego tutaj modelu, czy nie zachowuje się nadzwyczaj dziwnie w pierwszych fazach projektu.

Ze względu na niezwykle zaawansowany obiekt kół i kształt rolek, ważne jest aby uprościć model, poprzez zamianę niektórych składowych i dodanie sztucznych więzów. Całościowy model może być zbyt skomplikowany, aby maszyny symulacji mogły go obliczać w czasie rzeczywistym. Dokładny model także jest znacznie trudniej poprawnie wymodelować, ze względu na liczne tarcia i poślizgi rolek. Proponowane uproszczenia modeli opisane są w sekcji 4.3.

## 1.5 Podział tej pracy na sekcje

Kolejne rozdziały kolejno opisują różne aspekty pracy.

**Wstęp** Ten rozdział, zawiera ogólny opis pracy i sposób jej wykonania.

**Narzędzia** Opis narzędzi programistycznych, użytych przy budowie modeli i testowaniu.

**Baza** Techniczny opis rzeczywistej platformy i czujników, oraz mechaniki stojącej za zasadą ich działania.

**Model platformy** Implementacje modeli opisanych w poprzednim rozdziale, problemy i niedoskonałości z nimi związane.

**Komponenty systemu** Opis poszczególnych dodatkowych składników systemu, używanych w symulacji, testowaniu, wizualizacji i wspomagających tworzenie.

**Testy systemu** Testy różnych składników systemu, wykresy, interpretacja i podsumowanie.

## Rozdział 2

# Narzędzia

W tym rozdziale opisane są narzędzia użyte do wykonania zadania.

Środowisko symulacji składa się z maszyny symulującej fizykę, odpowiedzialnej za obliczenia fizyczne, a także API do obsługi całej symulacji. Zawansowana maszyna symulacyjna powinna dobrze obsługiwać 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.

Na rynku jest wiele różnych maszyn, zarówno do symulacji w czasie rzeczywistym, jak i do wyznaczania pozycji obiektów po długich obliczeniach. Istnieją technologie otwartoźródłowe, inne są własnościowe. Mogą używać tylko procesora, lub też być wspomagane przez kartę graficzną (na przykład *PhysiX*). Niektóre maszyny symulują, prócz zderzeń obiektów, także rozpływ cieczy, dymy, płotna, 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.

### 2.0.1 ROS

ROS jest skrótem od *Robot Operating System*, lecz jego nazwa jest myląca. Nie jest to system operacyjny, lecz obszerna platforma programistyczna (*framework*), zawierająca odpowiednie biblioteki i narzędzia do tworzenia programów sterujących. ROS stara się w łatwy sposób dostarczyć wszystko, co potrzebne do budowy logiki aplikacji sterowania. Są tu algorytmy wyznaczania tras, budowy map, manipulowania robotycznymi ramionami itp. Platforma programistyczna do pobrania jest z [12].

Twórcy zachęcają, aby uzupełniać brakujące moduły swoimi własnymi, a potem dzielić się nimi z resztą programistów. Na ich stronie internetowej

znajduje się obszerna baza danych różnych komponentów, do używania we własnych systemach.

Programy dla ROS pisze się w C++, lub Pythonie i integruje z robotem za pomocą kilku gotowych struktur kolejek wiadomości. Platforma ta także posiada moduły do wizualizacji odbieranych danych w formie graficznej.

Działanie systemu jest oparte o pakiety. Każdy taki komponent jest katalogiem zawierającym w sobie pliki opisujące jego parametry i skrypty CMake, używane do komplikacji. Pakiet może być programem wykonywalnym, danymi, definicjami, lub zestawem plików. W symulacji opisywanej platformy, modele są plikami i programami, łączonymi razem we wspólnym pakiecie, ładowanym do pakietu programu wykonywalnego. Jest to dokładnie opisane w rozdziale 5. Pakiety mogą być zależne od siebie, ale nigdy nie wskazują nawzajem swoich bezpośrednich ścieżek.

Globalny skrypt komplikacji ROSa dba o odpowiednie podawanie ścieżek, kolejność komplikacji programów i załączanie nazw. Na przykład, jeśli pakiet wymaga pliku nagłówkowego, generowanego przez komplikację innego pakietu, załącza go w kodzie tak, jakby był systemowy. Skrypt CMake zadba o wywołanie kompilatora z odpowiednimi argumentami.

Komunikacja między programami odbywa się w sposób ciągły przez kolejki wiadomości, lub pojedyncze asynchroniczne wywołania, zwracające wynik. Program może nadawać strumień wiadomości, ale niekoniecznie musi istnieć w tym czasie odbiornik. Można buforować wiadomości, podglądać strumienie, tworzyć wykresy z danych, podłączać nadajnik do kilku odbiorników, podglądać graf zależności itp. Do wszystkiego służy bogaty zestaw komend i wbudowanych narzędzi.

Instalacja programu na systemie operacyjnym jest złożona. Z wyjątkiem odpowiednich wersji Ubuntu, nie ma łatwego sposobu na instalację go na innych systemach. Na przeszkodzie stoją błędy komplikacji dla nowszych wersji kompilatorów, zależności od dokładnych wersji zewnętrznych bibliotek i inne problemy w czasie wykonywania, jak naruszenie ochrony pamięci. Instalacja alternatywnych pakietów i ręczna komplikacja niektórych części nie działa we wszystkich przypadkach.

Rozwiązaniem tego problemu jest instalacja tej platformy programistycznej na maszynie wirtualnej, lub na systemie uruchamianym z dysku zewnętrznego. Najnowszą wersję 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. Główny symulator fizyki, najważniejszy program, jest w tej samej

wersji w obu dystrybucjach.

Uruchomienie platformy programistycznej na systemie wymaga wielu dodatkowych komend inicjalizujących, a także dopisywania do tworzonych projektów licznych plików konfiguracyjnych za pomocą dostarczonych skryptów. Używanie modułów z linii poleceń wymaga ustawienia kilku zmiennych systemowych, poprzez wczytywanie skryptów. Użycie niektórych funkcji ROS wymaga uruchomionego demona serwera w tle.

Ogólnie instalacja i używanie ROS na systemie zostawia dużo różnorodnych plików w katalogu domowym, co może nie być wskazane na codziennym systemie operacyjnym. Z drugiej jednak strony, wirtualizacja systemu operacyjnego z ROS bardzo ogranicza dostępną moc obliczeniową, potrzebną takim programom w dużych ilościach.

## 2.0.2 Gazebo

Program do pobrania z [10]. Ten symulator graficzny jest dość prosty w obsłudze, skupia się na symulowaniu podanych danych, a nie na możliwości ich łatwego przygotowania. To znaczy, działa na podstawie manualnie napisanych plików konfiguracyjnych. Zazwyczaj używany w trybie wsadowym, uruchamiany z argumentami z linii poleceń i plikiem `world`, opisującym simulację. Plik ten zawiera nazwy i ścieżki umieszczanych w symulacji modeli i wtyczek. Z tego powodu interfejs graficzny jest dość ubogi.

Program przeprowadza symulację podanych modeli, używając jednego z czterech popularnych maszyn symulacyjnych: ODE, Bullet, Simbody lub DART. Wszystkie te projekty są wolnym oprogramowaniem i używane są także w innych programach, na przykład w edytorze Blender.

Symulator oprócz tego ma wbudowany edytor modeli, w którym można składać i ustawiać odpowiednie obiekty razem w przestrzeni trójwymiarowej i generować powyższy plik. Edytor budynków pozwala na stawianie wirtualnych ścian, korytarzy, drzwi i ogólnego otoczenia w którym roboty mogą pracować i być symulowane. Funkcjonalność tych edytorów jest bardzo ograniczona, brak jest tak podstawowych funkcji, jak cofanie ruchu. Dlatego lepiej jest zdefiniować model we wczytywanym pliku tekstowym. Również tworząc modele poza edytorem, posiada się nad nimi większą kontrolę, a parametry obiektów da się ustawać z dowolną dokładnością.

Gazebo przyjmuje modele w specjalnym formacie SDF. Jest to ustandaryzowany, zdefiniowany zewnętrznie 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 standardowym XML, co znaczy, że może być tworzona na każdym 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 ze wszystkich funkcjonalności systemu operacyjnego, jak na przykład komunikacja za pomocą pamięci współdzielonej. Gazebo dostarcza także swój własny mechanizm kolejek wiadomości, który sprawdza się w jednolitej komunikacji z zewnętrznymi programami, korzystającymi z bibliotek dostarczonych przez Gazebo, jednak jest niezależny od podobnej mechaniki ROSa.

Program jest w pełni wspierany na dystrybucji GNU/Linuksa Ubuntu ale bez problemu można go także skompilować pod inne systemy. Interfejs jest dopracowany i przestrzega wielu ustawień systemowych, na przykład takich jak DPI, lecz nie korzysta z dedykowanych bibliotek do tworzenia interfejsów typu Qt, lub GTK. Uruchamianie programu jest proste i nie wymaga dodatkowych ustawień, wywoływanie skryptów inicjalizujących, tworzenia odpowiednich katalogów, czy definiowania zmiennych systemowych. Podobnie jak inne programy, tworzy ukryty katalog w katalogu domowym użytkownika, gdzie składuje wszystkie modele i logi.

Gazebo może także być składnikiem systemu ROS, kod źródłowy jest dzielony w ramach wspólnej organizacji. Chociaż różne osoby odpowiadają za rozwój tych oprogramowań, kolejne wersje Gazebo są powiązane z kolejnymi wersjami ROSa, nie można użyć przestarzałej wersji Gazebo z nowszym ROSem i odwrotnie. Symulator można zainstalować osobno, lub jako jeden z pakietów ROSa. Jednakże, ze względu na chęć zachowania wysokiej kompatybilności pakietów ROSa, to nienajnowsza wersja Gazebo jest dostarczana z ROSem.

### 2.0.3 V-Rep

Program do pobrania z [11]. Duże i skomplikowane środowisko, reklamujące się wieloma zaawansowanymi mechanizmami i funkcjami. Pomimo otwartego kodu, użycie komercyjne jest płatne. Dla zastosowań akademickich program jest rozdawany bez opłat. Bogaty interfejs graficzny zakłada budowę i symulację wszystkiego w tym jednym programie.

Używa dwóch z maszyn symulacyjnych Gazebo, czyli ODE i Bullet, oraz dodatkowo Vortex i Newton. Z tej czwórki tylko Vortex ma zamknięty kod.

Głównym mankamentem programu jest zapisywanie utworzonych w systemie modeli. Program tworzy drzewiastą strukturę modelu, w pliku binarnym 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 Lua. Są też jednak dostępne inne języki, jak C, Matlab, Java itp. Komunikacja z innymi programami odbywa się poprzez specjalne wtyczki do środowiska. API pozwala stworzyć mały, wbudowany 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.

#### 2.0.4 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.

**Gazebo** będzie użyty do symulacji z jego domyślną maszyną symulacji fizyki ODE.

**ROS** użyty zostanie jako główna platforma programistyczna. Pod łatwą komunikację z jego modułami należy budować sterowniki wirtualne.

**CMake** to popularny i polecaný 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. Razem z nim użyty zostanie debugger GDB.

**KDevelop** nadaje się do pisania komplikowanego kodu wtyczek. Można połączyć je pod komendę `make` i korzystać z mechanizmów interpretacji błędnych linii kodu, graficznego debugowania i podobnych.

**Bash** będący bardzo popularnym językiem skryptowym nadaje się do automatyzacji pracy i uruchamiania testu w kontrolowany i prosty sposób. Uniwersalne narzędzie pomagające w wielu miejscach.

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

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

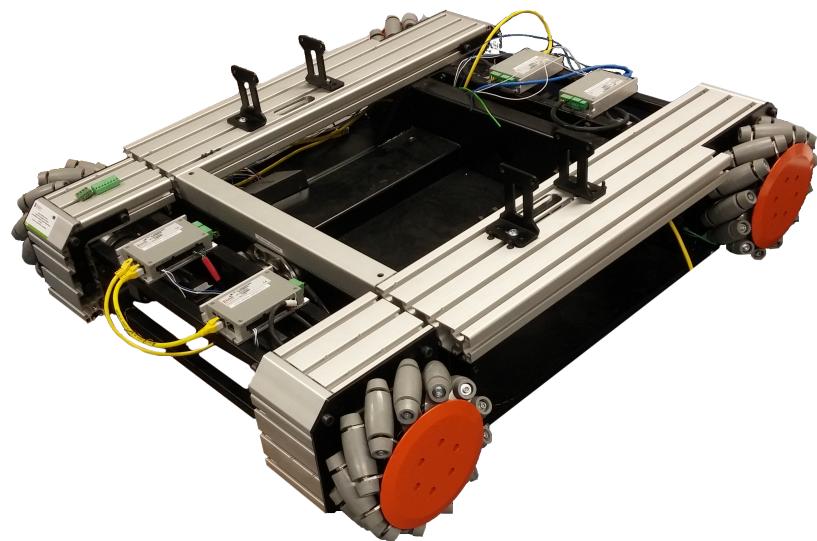
**Dia** to graficzny edytor do tworzenia diagramów UML.

**Virtualbox/Qemu** do ewentualnej wirtualizacji systemu operacyjnego z ROS, lub uruchomienie osobnego systemu z dysku zewnętrznego.

# Rozdział 3

## Baza

### 3.1 Dookólna platforma mobilna



Rysunek 3.1: Dookólna baza mobilna na kołach szwedzkich.

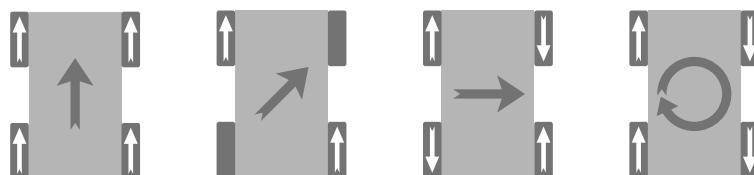
Jest to duża, prostokątna baza dookólna, poruszająca się na czterech kołach szwedzkich, patrz fotografia 3.1. Koła są stałe, parami przytwierdzone do dwóch osi. Każde koło jest sterowane osobno przez podłączony bezpośrednio serwomotor, zatem może mieć prędkość i kierunek niezależny od prędkości pozostałych kół, kierunku poruszania się robota, oraz jego obrotu. Każdy z serwomotorów ma także wbudowany enkoder. Sterownik enkodera zwraca aktualny kąt i prędkość obrotu.

Jest to najpopularniejsza budowa dookółnych platform mobilnych, mająca zastosowanie także w innych robotach, jak na przykład Kuka Youbot 3.2. Istnieją także roboty o trzech kołach szwedzkich, w których to koła rozstawione są promieniście pod kątami  $120^\circ$ . Pomimo prostszej budowy i takiej samej ilości stopni swobody, co czterokołowa wersja, stabilność takiego rozwiązania jest gorsza od zastosowanej tutaj budowy [7]. Ponieważ jest to robot transportowy, to stabilność odgrywa tu ważną rolę i czterokołowa budowa jest wskazana.



Rysunek 3.2: Przykład innej platformy wielokierunkowej na podstawie fragmentu komercyjnego robota Kuka Youbot. Należy zwrócić uwagę na charakterystyczne ustawienie kół, identyczne jak w opisywanej platformie 3.1.

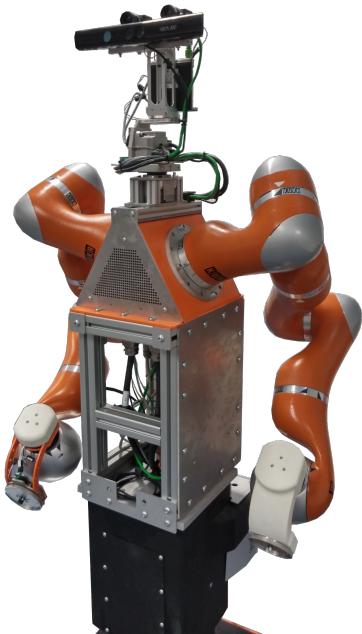
Odpowiedni obrót kół względem bazy, pozwala na jej ruch w dowolnym kierunku, niezależnym od kąta obrotu robota, patrz rysunek 3.3. Jest możliwe także obracać bazę, gdy ta porusza się w dowolnym kierunku, bądź stoi w miejscu.



Rysunek 3.3: Podstawowe ruchy, jakie może wykonywać robot o napędzie wielokierunkowym.

Przykładowo, poruszając tylko przeciwnymi kołami po przekątnej, system będzie mógł poruszać po skosie, bez zmiany kąta obrotu. 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 prosto do kierunku jazdy. Trasa po której porusza się robot, przy stałej prędkości kół, zawsze jest okręgiem, 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 prędkość kątową, będzie się poruszał po takiej krzywej.

Podstawa ma za zadanie transportować robota manipulującego Velma, tworząc razem manipulator mobilny. Velma to wysoki i bardzo ciężki robot, wyposażony w dwa chwytki na ramionach o wielu przegubach, patrz fotografia 3.4. Taka budowa wymaga szerokiej podstawy, aby zachować bezpieczną równowagę całości. Jeżdżąc na tej podstawie, robot może się przemieszczać i obracać w dowolnym kierunku, aby uzyskać lepszy dostęp do manipulowanych przedmiotów. Dodatkowe czujniki laserowe umieszczone tuż nad podstawą odpowiadają za wykrywanie kolizji i lokalizację.



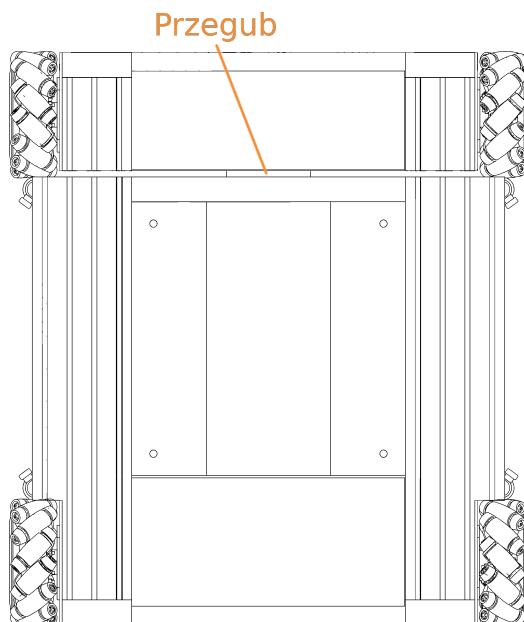
Rysunek 3.4: Robot manipulacyjny Velma.

Platforma jest niesymetrycznie podzielona na dwie niezależne części, przed-

nią i tylną, w sposób pokazany na rysunku 3.5. Przegub o jednym stopniu swobody (tzw. zawias) jest jedynym łącznikiem pomiędzy tymi dwoma fragmentami. Zadaniem tego przegubu jest zmniejszanie wpływu nierówności podłoża na ruch bazy, aby każde koło dociskało do podłoża z taką samą siłą, jak po drugiej stronie osi. Bez tego zawiasu nierówny teren uniemożliwiałby sprawne sterowanie platformą na skutek niedeterministycznego tarcia kół tej samej osi, powodując nieplanowany skręt. Niedeterministyczne tarcie kół jest niewykrywalne w bezpośredni sposob, jak to zostało opisane w [8].

Platforma nie jest idealnym kwadratem, jest 4 cm różnicy między szerokością, a długością robota. Także środki kół nie są ustawione na wierzchołkach tej figury geometrycznej. Szerokość jest większa, co można zobaczyć porównując widok z prawej strony 3.6 z widokiem z tyłu 3.7. Dokładne wymiary są podane na rysunku 4.1 i tabeli 4.1.

Platforma podatna jest na losowy ruch przy rozpoczętym jazdzie i hamowaniu. Jest to spowodowane tym, że asymetria rolek będzie nadawać kołom różne siły oporu, a w związku z tym różne prędkości, co w efekcie może powodować niedeterministyczny ruch. Należy także wziąć tutaj pod uwagę inne cechy budowy kół, jak nierówne tarcie rolek o powierzchnię [6].



Rysunek 3.5: Platforma mobilna — widok od góry. Przegub zawiasowy łączy dwie części.



Rysunek 3.6: Platforma mobilna — widok z prawej strony.



Rysunek 3.7: Platforma mobilna — widok z tyłu.

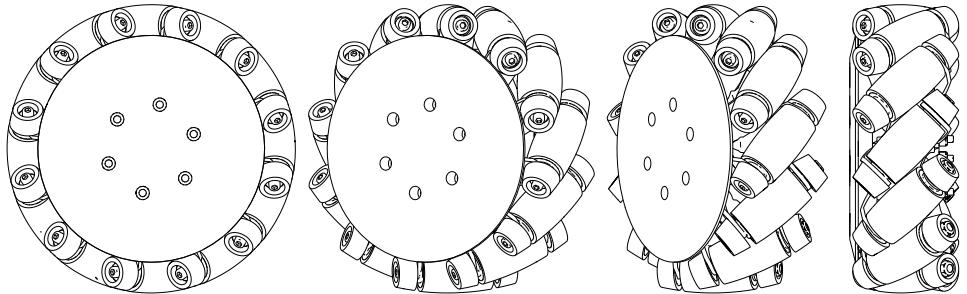
Platforma posiada 3 stopnie swobody.

- Ruch bez obrotu równolegle do osi X.
- Ruch bez obrotu równolegle do osi Y.
- Obrót w płaszczyźnie podłoża.

## 3.2 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 3.8. W platformie ich osie ustawione są w ten sposób, że osie najwyższych, lub najniższych, rolek dwóch kół z tej samej strony robota 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. Warto pamiętać, iż oś aktualnie dolnej rolki jest prostopadła do osi górnej rolki.

Istnieje również odwrotna odmiana ustawienia kół, w której rolki tworzą literę  $O$ , czyli oś przednia jest zamieniona z tylną, lub jakby cała platforma była odwrócona do góry nogami. Ten drugi sposób także pozwala na ruch wielokierunkowy, ale nie jest tak często stosowany [3].



Rysunek 3.8: Widok 12 rolkowego koła szwedzkiego opisywanej platformy wielokierunkowej.

Każde koło ma 3 stopnie swobody [1], tak samo jak cała platforma.

- Obrót koła w osi.
- Rotacje pojedynczych rolek.
- Poślizg obrotowy w miejscu styku rolki z podłożem.

Na podstawie rysunku 3.8 widać, że krzywizna rolki jest tak ustawiona, aby punkt kontaktu rolki z podłożem w czasie obrotu płynnie przechodził na następną rolkę. Celem jest utrzymanie równej odległości osi od płaszczyzny podłożu. Nie powinno być efektu przeskoku z jednej rolki na drugą, gdyż to wprowadza nierówne tarcie, losowe poślizgi i nadmierne zużycie elementów wykonawczych. Kształt pojedynczej rolki zawiera się w paraboloidzie, wzory opisujące kształt rolki są złożone. Zazwyczaj przybliża się taką rolkę wycinkiem torusa, w celu uproszczenia produkcji [4].

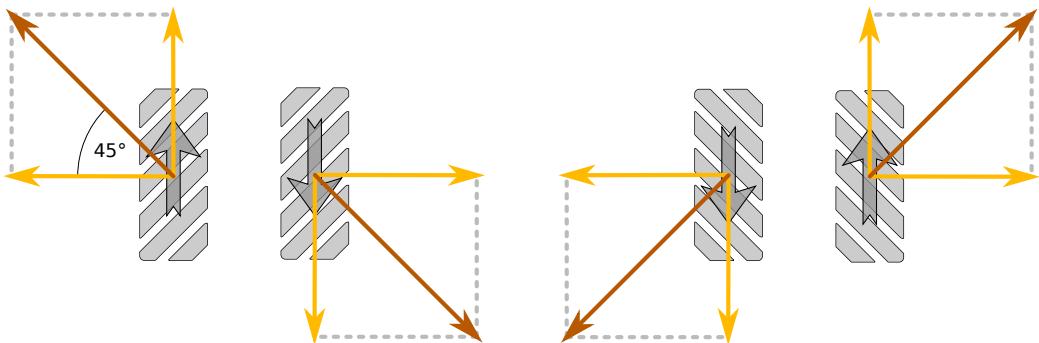
Istnieją także inne budowy kół, złożone z wielu małych rolek, tak aby w każdym momencie więcej jak jedna rolka dotykała podłożu. Można także złożyć kilka powyższych kół obok siebie w jedno koło. Przydatne jest to dla robotów transportujących duże masy, gdyż zmniejsza to obciążenie pojedynczych rolek.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

Podstawowym problemem technologicznym koła jest nie tylko skomplikowana budowa, ale także ślizganie się rolek po powierzchni. Odległość osi od płaszczyzny nieznacznie zmienia się przy przenoszeniu ciężaru z rolki na rolkę, co przy dużych prędkościach powoduje drgania i jeszcze większe błędy pomiarów. Środkowy przegub zmniejsza ich przenoszenie na drugą część platformy. Poślizg kół powoduje, że enkodery nie mogą być jedynymi czujnikami służącymi do wyznaczania pozycji bazy, gdyż są zbyt mało dokładne [5].

Dodano więc dwa czujniki laserowe, opisane dokładniej w sekcji 3.3, aby program sterujący nie bazował jedynie na odometrii przy wyznaczaniu sterowania. Istnieje także czujnik inercji, wykrywający zmianę rotacji i prędkość kątową robota.

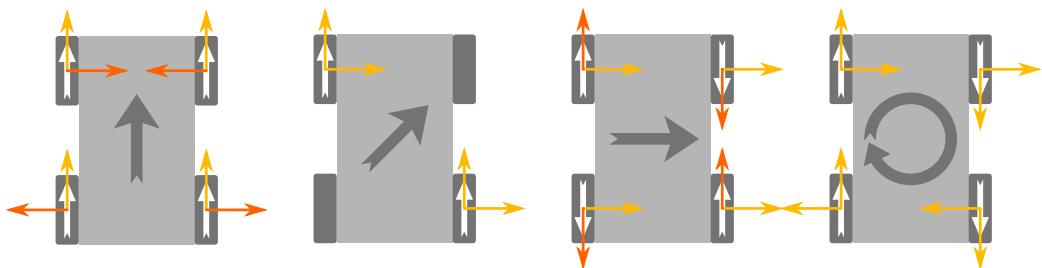
### 3.2.1 Działanie platformy

Standardowe koło, używając tarcia, przekształca prędkość kątową na liniową w płaszczyźnie obrotu. Specjalne koło Mecanum ma dodatkowy wektor, równoległy do osi obrotu, zatem prędkość wypadkowa jest obrócona o  $45^\circ$  w stosunku do wektora prędkości standardowego koła, zależnie od typu koła (prawoskrętnie lub lewoskrętnie) i kierunku obrotu.



Rysunek 3.9: Wektory składowe i wypadkowe koła widzianego z góry.

Ustawiając te koła w odpowiedni, opisany wcześniej na obrazku 3.5, sposób, można wywołać odpowiednie znoszenie się składowych prędkości, a w efekcie pozwolić robotowi na poruszanie się w kierunkach nieosiągalnych dla pojazdów o standardowych kołach. Warto nałożyć te składowe na wcześniejszy rysunek 3.3, aby dokładnie zobaczyć, dlaczego koła nadają platformie daną prędkość wypadkową przy odpowiednim obrocie kół.



Rysunek 3.10: Ruchy platformy widzianej z góry, z nałożonymi składowymi wektorów prędkości. Ciemniejszym kolorem zaznaczono znoszące się składowe.

Warto rozpatrzyć każdy przypadek. Platforma posiada jedną płaszczyznę symetrii, ze względu na asymetryczny przegub.

1. Wektory o kierunku prostopadłym do pionowej płaszczyzny symetrii urządzenia znoszą się, pomimo że mają przeciwnie zwroty na przedniej i tylnej parze kół. Pozostają jedynie składowe równoległe do płaszczyzny symetrii, które powodują prostoliniowy ruch naprzód.
2. Dwa koła nie obracają się. Nie jest to ruch pasywny, gdyż taki wprowadzałby nieprzewidywane poślizgi, a aktywne hamowanie. Wektory się nie znoszą i platforma wykonuje ruch pod kątem  $45^\circ$  do płaszczyzny symetrii.
3. Ruch podobny jest do przypadku 1. Tutaj również wektory znoszą się parami, jednak tym razem na prawych kołach i lewych. Pozostają składowe prostopadłe do płaszczyzny symetrii.
4. Prędkość kątowa powstaje, gdy wypadkowa kół po jednej stronie platformy znosi się z wypadkową po drugiej stronie.

Warto nadmienić, że gdy wypadkowy wektor prędkości koła jest prostopadły do osi koła, to jest gdy koło porusza się zgodnie z kierunkiem obrotu, w idealnym przypadku rolki nie obracają się. Inaczej mówiąc, rolka będzie się obracać tym mocniej, im bardziej ruch koła wymuszany jest równolegle do osi koła, czy to na skutek znoszenia się wektorów, czy oporu przeszkody.

Przykładowo, przy ruchu naprzód rolki koła się nie obracają, lecz przy ruchu w bok biorą aktywny udział. 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 robota przy jeździe zawsze obracają się szarpanym ruchem w obie strony, ze względu na poślizgi od innych kół, niejednostajne tarcie piast wszystkich

rolek, czy różnice terenu. 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 mechanikę kół.

### 3.3 Czujnik laserowy

Program sterujący platformą nie jest w stanie dokładnie określić pozycji robota, bazując jedynie na odometrii. Potrzebny jest zatem czujnik laserowy. Platforma wyposażona jest w dwa, dwuwymiarowe czujniki typu LiDAR firmy SICK. LiDAR to zbitek wyrazów *light* i *radar*, chociaż skrót może być rozwinięty w różne słowa.



Rysunek 3.11: Czujnik laserowy SICK LMS100-10000.

#### 3.3.1 Zasada działania

Wszystkie czujniki 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 pulsacyjną wiązkę podczerwonego promienia co pewien okres czasu. Emitowanie stałego promienia może być niebezpieczne dla wzroku obsługujących go ludzi. Aktualna pozycja lusterka jest wykrywana przez enkoder. Obok lasera jest czujnik, który bada wysłane przez laser, odbite od lusterka, obiektu i ponownie lusterka, światło.

Na koniec, algorytm we wbudowanym mikrokontrolerze ustala kąt i odległość czujnika od wykrytego obiektu. Odpowiada także za usunięcie szumu i ewentualnych odbić promienia. Komunikacja z urządzeniem może odbywać się za pomocą różnych interfejsów sieciowych, zazwyczaj w architekturze typu master-slave. W przypadku tej platformy na obecny czas jest to Ethernet.

Skośna szyba, będąca wycinkiem powierzchni stożka, zabezpiecza wnętrze przed zanieczyszczeniami, jej kształt niweluje ewentualne odbicia lasera, emitowanego poziomo ze środka. W niektórych czujnikach montuje się także szereg dodatkowych diod podczerwieni na obrębie szyby, skierowanych w górę, lub w dół, oraz czujniki/reflektory z drugiej strony. Pozwala to na wykrycie stopnia zanieczyszczenia szyby, aby powiadomić użytkownika o potrzebie wyczyszczenia urządzenia.

### 3.3.2 Komunikacja

Wysyłając do czujnika odpowiedni ciąg bajtów, można ustawić jego tryb działania, odpytać o zebrane dane, czy wykryć konfigurację i stan.

W przypadku tej platformy, komunikacja odbywa się poprzez interfejsy EtherCAT i Ethernet. EtherCAT to sposób komunikacji urządzeń po kablu Ethernetowym, w trybie *master-slave*, przy zachowaniu sztywnych ram czasowych. *Master* wysyła pakiet do podłączonych szeregowo urządzeń *slave*, które przekazują go przez siebie i w razie potrzeby modyfikują dane w locie.

Program odbierający dane od czujnika komunikuje się bezpośrednio z urządzeniem, które zwraca pakiety zawierające pomiary z ostatniego obrotu czujnika, oraz dodatkowe dane opisujące sam pomiar, takie jak czas, początkowy kąt pomiaru, czy tryb pracy. Dokładne pola w pakiecie i cechy czujnika dostępne są na stronie producenta [14].

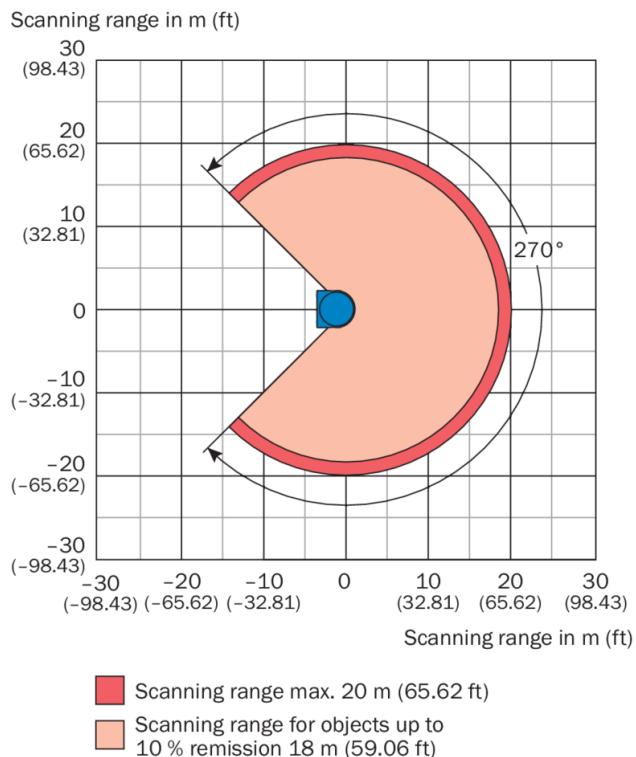
Urządzenie wspiera uwierzytelnianie przez hasło, wgrywanie nowego oprogramowania, ustawienia czasu, oraz zmianę różnych parametrów działania.

### 3.3.3 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 3.12.

Cecha	Wartość
Kąt pracy	270°
Długość fali światła lasera	905 nm (podczerwień)
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 odległości	0,012 m

Tablica 3.1: Podstawowe cechy czujnika laserowego.



Rysunek 3.12: Wykres producenta dotyczący zasięgu czujnika.

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

### 3.4 Czujnik inercji

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

Czujnik użyty w platformie to ADIS16460AMLZ, firmy Analog Devices. Szczegółowa dokumentacja jest dostępna na stronie producenta [15].

Urządzenie ma kształt małej kostki i komunikuje się za pomocąłącza SPI, a co za tym idzie, wymaga zewnętrznego mikrokontrolera, aby móc wysyłać wygenerowane dane do sieci do innych urządzeń.

Czujnik posiada:

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

Ten czujnik, podobnie jak opisany wcześniej LiDAR, również jest podatny na błędy pomiarowe, a co za tym idzie, należy dodać losowe odchylenia do generowanych danych.

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

# Rozdział 4

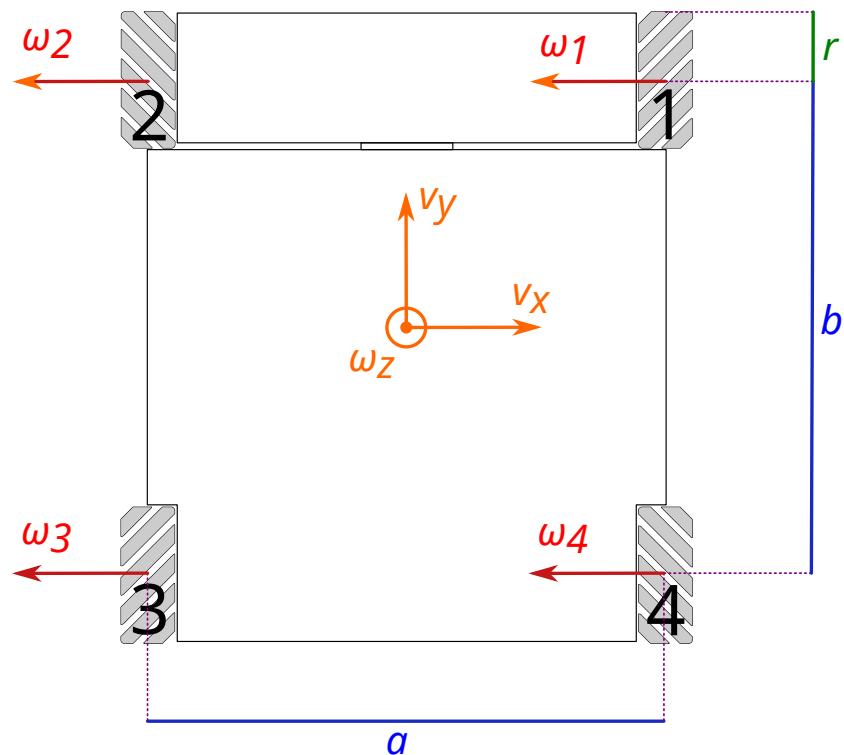
## Model platformy

Pierwszym krokiem do stworzenia modelu dynamicznego jest stworzenie modelu kinematycznego, aby móc porównać z nim tworzony model dynamiczny, oraz fizyczną platformę, w celu weryfikacji poprawności działania. Dzięki temu, można łatwo oszacować, jak bardzo błędy symulacji, oraz błędy niedoskonałości fizycznego modelu odstają od matematycznych wyliczeń.

Dodatkowo, stworzenie modelu platformy kinematycznej pozwala na proste odrzucanie niedziałających implementacji modelu dynamicznego. Model platformy kinematycznej ma także kluczowe zastosowanie w odometrii.

Oznaczenie	Wartość	Opis
$r$	0,1 m	Promień koła w najszerzym miejscu na środku.
$a$	0,76 m	Szerokość platformy między środkami kół tej samej osi.
$b$	0,72 m	Długość platformy między środkami kół tego samego boku.
$\omega_i$		Prędkość kątowa każdego z kół.
$v_x$		Prędkość transwersalna w osi X.
$v_y$		Prędkość transwersalna w osi Y.
$\omega_z$		Prędkość kątowa w osi Z, wektor skierowany w górę.

Tablica 4.1: Opisy i wartości symboli używanych we wzorach i rysunkach.



Rysunek 4.1: Wielkości używane we wzorach.

## 4.1 Sposób zapisu w formacie SDF

*Simulation Description Format* (SDF) jest formatem XML, pozwalającym na określenie elementów 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 wokół robotów, na przykład rozmieszczenie elementów na symulowanej scenie, określania wyglądu i fizycznego zachowania się materiałów itp.

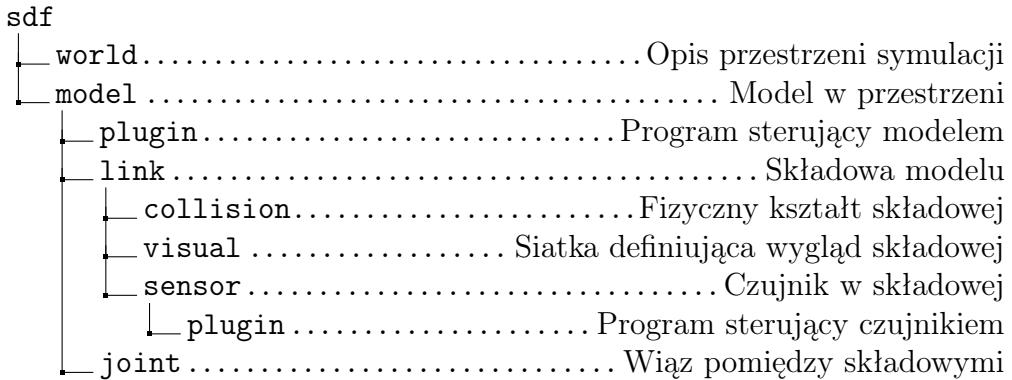
W przeciwnieństwie do poprzednika, zapisującego model w przestrzeni drzewiastej, SDF równolegle określa wszystkie składowe modelu, oraz zależności między nimi jak więzy i względne pozycje. Model składowych robota ma strukturę gwiazdową. Jeden element, `model`, jest nadzędny, wszystkie składowe są logicznie rozmieszczone równolegle jako podelementy. Specjalnie opisane więzy definiują interakcje pomiędzy składowymi. Jako model, standard rozumie nie tylko roboty, ale także obiekty typu przeszkody, źródła światła, elementy animowane i tym podobne. Standard jest dobrze opisany na stronie internetowej [13].

Element typu `world`, równoległy do modeli, zawiera informacje o środowisku symulacji. Dodatkowo można dodać informację o ustawieniach maszyny symulującej fizykę, wyglądzie sceny, wietrze, grawitacji, polu magnetycznym itp.

W każdym z modeli zawiera się nazwa, domyślna pozycja, sposób traktowania przez symulator i wtyczki programów obsługujących zaawansowane zachowanie modelu, opisane w równoległych do składowych elementach, ale jako że wszystkie te mogą wystąpić tylko raz, należy traktować je jako część elementu `model`. Model, lub jego fragment, może być zimportowany z innego pliku, lecz nie zmieni to struktury gwiazdowej, a co za tym idzie, może dojść do utraty informacji. Ten przypadek zachodzi przy modelach czujników laserowych, opisanych w rozdziale 4.4.

Model zawiera w sobie równolegle wszystkie elementy typu `link`, każdy z nich jest osobną, pełną częścią robota, na przykład kołem, fragmentem ramienia chwytaka, kadłubem, czujnikiem. Składa się w sobie informacje o pozycji względem lokalnego środka układu współrzędnych modelu, masie, kształcie, fizycznym kształcie, materiale fizycznym i wyglądzie. Pozwala na dodanie elementów reprezentujących źródła dźwięku, czujniki, baterie itp.

Same elementy zawierają jedynie informacje o swoim początkowym umiejscowieniu w modelu, ale nie o sposobie poruszania się i nałożonych więzach. Do tego potrzebne są, równolegle do elementów `link`, typy `joint` określające typ więzów, osie, współczynniki sprężystości, wytrzymałość, czy moc silników. Każde połączenie określa, między jakimi obiektami się łączy.



Rysunek 4.2: Najważniejsze elementy formatu SDF.

## 4.2 Model kinematyczny

Kinematyka to nauka o ruchach obiektów na podstawie nadanych wektorów prędkości. Pomija ona takie aspekty jak masa, moment bezwładności, czy siły.

Model kinematyczny jest sterowany funkcjami matematycznymi, zamieniającymi prędkość kątową kół platformy na prędkości transwersalne geometrycznego środka platformy w układzie współrzędnych lokalnych, oraz jego prędkość kątową. Symulator pozwala również na całkowanie tego ruchu, aby uzyskać aktualną pozycję platformy.

Funkcje tłumaczące prędkości kół najwygodniej zapisać w postaci macierzystej, podobnie do tego, jak opisano w [2]. Wzór powtarza się w wielu innych pracach naukowych, a jego dokładny kształt zależy od kolejności numerowania kół i interpretacji wymiarów. Dla opisanego tutaj przypadku, (stałe zdefiniowane są w tabeli 4.1, numerowanie kół na rysunku 4.1):

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

Uzyskane wartości należy przemnożyć przez odpowiednie wektory jednostkowe, obrócić względem lokalnego układu współrzędnych dla modelu i zastosować w funkcjach nadających prędkości bryle.

Ponieważ sterowanie pozycją modelu kinematycznego odbywa się wyłącznie poprzez wzory matematyczne, w jego symulacji nie uczestniczy maszyna symulacyjna fizyki. Taki model nie reaguje na kolizje z innymi obiekta mi, nie

reaguje na różnicę terenu i nie używa informacji o współczynnikach tarcia materiałów, tak jak model dynamiczny.

Nazwa kodowa **pseudovelma** odnosi się do tego, że jest to nieprawdziwy ruch sterowany z zewnątrz, a nie prawdziwa symulacja.

### 4.2.1 Problemy implementacji

Gazebo nie ma zaimplementowanego pełnego wsparcia dla standardu SDF. W szczególności nie działa struktura elementów **frame**, odpowiadająca za transformacje obiektów względem innych obiektów. Nie jest to zapisane 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 za pomocą funkcji, nadajemy ją każdej składowej osobno. Każde z kół i dwie części bazy, obracają się zgodnie z zadanymi wartościami, ale ich środki pozostają w miejscu, w którym rozpoczęły symulację, ignorując kompletnie pozycję zdefiniowaną dla elementu rodzica **model**.

Z punktu widzenia symulacji fizycznej ma to sens, gdyż nie można zakładać że składowe modelu są w jakikolwiek sposób podłączone do jego głównej części (która wcale nie musi mieć zdefiniowanego kształtu fizycznego), to wprowadzałoby także nieścisłości w typie połączenia, niektóre elementy powinny być przecież ruchome.

Aby przeciwdziałać temu zjawisku, należy przenieść zawartość elementów **link** do elementu **model** i ustawić je jako **visual** elementu **model**. W ten sposób traktowane są jako część renderowana modelu, a nie osobne składowe. Nie można użyć tutaj więzów statycznych, gdyż te są wykorzystywane przez maszynę symulacyjną fizyki i ignorowane są przy kinematycznym ruchu.

Powstała niedogodność jest taka, że ciężej jest sterować obrotem elementów **visual**, gdyż są zarządzane przez完全nie inny system symulatora, służący do graficznego renderowania sceny. Oczywiście ma to znaczenie jedynie kosmetyczne, gdyż w żaden sposób nie wpływa na ruch modelu bazy.

### 4.2.2 Komunikacja

Komunikacja programu sterującego platformą odbywa się przez wbudowane z ROSa narzędzie *topic*.

Wiadomość zawierająca dane prędkości kół czterokołowego robota nie mieści się w standardzie, zatem został stworzony specjalny typ **omnivelma\_msgs/Vels**.

Ta struktura zawiera cztery wartości zmiennoprzecinkowe podwójnej precyzji, oznaczające prędkości w  $\frac{\text{rad}}{\text{s}}$ .

Program w każdym cyklu symulacji nadaje wiadomości:

- `geometry_msgs/PoseStamped` z aktualną pozycją i rotacją platformy, oraz nagłówkiem z identyfikatorem i czasem nadania pakietu.
- `geometry_msgs/TwistStamped` z aktualną prędkością platformy, oraz identycznym nagłówkiem.
- `nav_msgs/Odometry` z obiema powyższymi danymi i nagłówkiem. Służy przy obliczaniu ruchów platformy na podstawie enkoderów kół.

Ponadto program przyjmuje dane:

- `omnivarma_msgs/Vels` z zadanymi prędkościami kół.

#### 4.2.3 Zachowanie

Platforma ignoruje kompletnie otoczenie, poruszając się przez inne obiekty na scenie. Po nadaniu stałych prędkości kół, następuje ruch po okręgach zgodnie z rysunkiem 3.3.

Program sterujący, co każdą klatkę symulacji (okres zależy od zasobów procesorowych komputera), zwraca aktualną pozycję i rotację, działając jak układ całkujący funkcje ruchu z powyższej macierzy.

### 4.3 Model dynamiczny

Wykorzystując maszynę do symulacji fizyki, można umieścić w niej model określający kształty, masy i zależności pomiędzy składowymi modelu, następnie nadać elementom wirtualne siły i otrzymać przybliżone wyniki do tego, jak zachowywałaby się rzeczywista platforma.

Wszystkie programy i definicje związane z tym modelem noszą nazwę `omnivarma`, co nawiązuje do wielokierunkowości ruchów robota manipulującego, którego podstawa ma transportować.

Robot jest bryłą, na którą składają się następujące części składowe:

- Główna część trzonu.
- Ruchoma, mniejsza część trzonu, z przodu robota.
- 4 koła, 2 podłączone do głównej części, a 2 do przedniej.

- Po 12 rolek na każdym kole.
- Przegub zawiasowy, łączący dwie części podstawy.
- 4 przeguby zawiasowe z silnikami, łączące części bazy z kołami.
- 12 przegubów zawiasowych na każdym kole, łączących koła z rolkami.

Jest to dość skomplikowany obiekt do symulacji, dlatego należy skłonić się do znalezienia sposobu na uproszczenie modelu, w celu zmniejszenia ilości obliczeń symulatora i błędów reprezentacji liczb zmiennoprzecinkowych.

Istnieje wiele podejść do stworzenia odpowiedniego modelu. Każde z nich było proponowane na różnych forach przez osoby symulujące podobne bazy, a także w zawartej bibliografii.

Jednak idealne rozwiązanie nigdy nie zostało znalezione. Działający w tym przypadku sposób także nie został wcześniej sprawdzony.

#### **4.3.1 Jak największe zbliżenie do oryginału**

Wspomniany wyżej sposób jest najbardziej wymagającym obliczeniowo, ale także najprostszym z możliwych. Należy stworzyć elementy składowe systemu i nadać im fizyczny kształt za pomocą odpowiedniej siatki trójkątów. Kształt obiektów może być także ustalony jednym prymitywów, jak sześcian, kula, łamana, walec i płaszczyzna. Takie przybliżenie znacznie przyspiesza obliczenia, gdyż może być specjalnie traktowane przez algorytmy.

Słabym punktem tego rozwiązania jest fakt, że rolki są niestandardowym kształtem, opisany dokładniej w [4], którego dokładność jest bardzo wysoce wymagana dla zmniejszenia niedokładności ruchów. Przybliżenie jej walcem powoduje problemy przy przenoszeniu punktu podparcia na kolejną rolkę, gdyż koło będzie musiało przez chwilę oprzeć się o krawędź. Taki model samoczynnie drgałby przy obrocie koła, zwiększąc tym samym i tak duże niedokładności. Podejście to zostało także zaproponowane w innej pracy naukowej [9].

Przybliżenie rolki siatką jedynie zmniejsza powyższy efekt, gdyż sama siatka zbudowana jest z prostych odcinków. Zwiększając jej gęstość można teoretycznie poprawić jakość symulacji, kosztem olbrzymiego skoku ilości obliczeń, każde obarczone błędami liczbowymi. Kalkulowanie kolizji fizycznej siatek jest najdroższe ze wszystkich obliczeń kolizji w maszynach symulacji fizyki.

#### **4.3.2 Resetowanie pozycji koła**

Ten sposób został użyty w modelach w symulatorze V-Rep.

Polega on na tym, iż koło, podłączone do bazy, posiada przegub zawiasowy, obrócony pod kątem  $45^\circ$  w stosunku do osi koła, tak aby był równoległy do aktualnie dolnej rolki. Do tego przegubu podłączona jest kula reagująca z podłożem, którą to w każdej iteracji symulacji należy zresetować do pozycji wyjściowej, razem z przegubem. Kula jest prymitywem i obliczenia jej kolizji są najmniej wymagające obliczeniowo dla maszyny symulacyjnej.

```
kadłub..... Podstawa bazy
└─ przegub z silnikiem ..... Nadaje moment obrotowy na żądanie
    └─ wirtualne koło..... Obiekt bez kształtu, obraca się jak koło
        └─ siatka ..... Odpowiada za wygląd obiektu koła
            └─ przegub 45° ..... Obrócony pod kątem 45°
                └─ kula kształtu..... Uczestniczy w symulacji fizyki
```

Rysunek 4.3: Zagnieźdzenie obiektów koła z resetowaną symulacją rolki.

Wywołuje to takie działanie, jak gdyby koło w danej chwili mogło obracać się w dwóch kierunkach, tak jak aktualnie najwyższa rolka i tak jak wymusza to na kole symulator silnika mechanicznego. Przez następną klatkę symulacji, model zachowuje się poprawnie, aż obrót głównej osi zaczyna negatywnie wpływać na symulację, zmieniając kąt wewnętrznego przegubu, przez co przestaże być równoległy do dolnej rolki. Zanim jednak ten efekt się nasili, jego rotacja jest przywracana do pozycji początkowej. Ponieważ jest to wywoływane poprzez zmianę rotacji, a nie nadanie momentu obrotowego obiekty, maszyna symulacyjna nie bierze w takim przypadku pod uwagę tarcia kuli o podłoże.

Niestety, nie jest możliwe uzyskanie 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 rotacji i pozycji przegubu, elementu `joint`, co nie jest wystawione do modyfikacji w API.

Bardzo skomplikowany sposób działania kół skłania do szukania innych rozwiązań.

Taka budowa koła ma jeszcze jedną, ważną cechę. Jakość symulacji zależy od jej prędkości. Jest tak, ponieważ im bardziej obciążony jest symulator, tym większy czas pomiędzy kolejnymi klatkami symulacji i pomiędzy kolejnymi resetowaniami pozycji koła. Oznacza to, że druga oś zaczyna wpływać na symulację z nieliniowo rosnącym błędem, aż do kolejnego resetu koła. Przy odpowiednio niskim czasie odświeżania, różnice powodują nieakceptowalny spadek jakości symulacji.

### 4.3.3 Zmiana osi rolki

Poprzedni przypadek można zmodyfikować, poprzez wyznaczanie nowej osi przegubu, łączącego wirtualne koło z kulą symulującą rolkę. Takie rozwiązanie nie wymaga przywracania rotacji obiektów do poprzedniej wartości, a co za tym idzie, może być użyte w symulatorze Gazebo.

Oś wewnętrzna podłączona jest do koła wirtualnego i obraca się razem z nim. W każdym cyklu należy zamienić, obróconą już w tej klatce oś, na kierunek pierwotny względem postawy platformy. Spowoduje to, że kolejna iteracja fizyki będzie mogła obracać kulą reprezentującą rolkę pod odpowiednim kątem.

Należy obliczyć kierunek osi w przestrzeni globalnej, biorąc pod uwagę aktualną pozycję platformy i obrót kół. Obliczenia tego kierunku są złożone.

Potrzeba jest jeszcze wybrania momentu wyznaczania nowego kierunku osi, bowiem maszyna symulacyjna wywołuje wiele różnych funkcji w jednym cyklu symulacyjnym. Zależnie to tego, który moment czasowy się obierze, można spodziewać się różnego zachowania modelu.

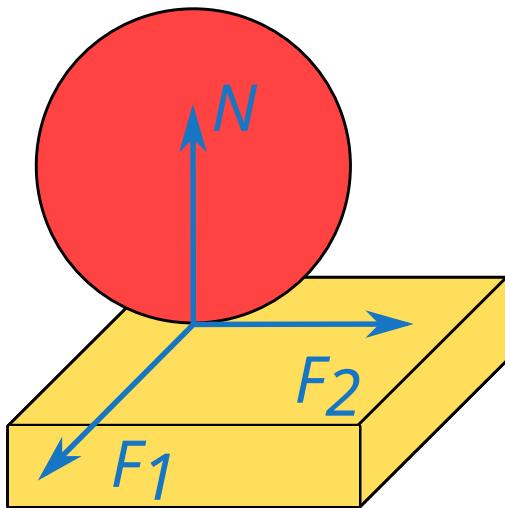
Implementacja tej budowy niestety nie stworzyła działającego rozwiązania. Wykonywanie kodu w różnych momentach symulacji nie wpływało na efekt. Platforma poprawnie poruszała się jedynie do przodu i do tyłu. Pierwszy problem pojawiał się, gdy w czasie ruchu na bok, prędkość malała, aby zmienić zwrot pomimo niezmiennej prędkości kół. Po kilku sekundach problem się powtarzał. Takie zachowanie najczęściej spowodowane jest zachowaniem kierunku osi w lokalnym układzie współrzędnych koła.

Drugim problemem był ruch po krzywej, w której model nieregularnie podskakiwał, w końcu nawet obracając się w pionie. Takie zachowanie oczywiście dalekie jest od oryginału.

Prawdopodobnie problemem było wewnętrzne traktowanie przegubów przez maszynę symulacyjną. Takie nienaturalne zachowanie, jak nagła zmiana osi przegubu, musiała wprowadzać nieprawidłowe wartości do zmiennych stanu, co w rezultacie powodowało tak chaotyczny ruch.

### 4.3.4 Modyfikacja kierunków i wartości wektorów tarcia

Warto tu wytłumaczyć, w jaki sposób maszyny symulacji fizyki interpretują kolizję i dotyk.



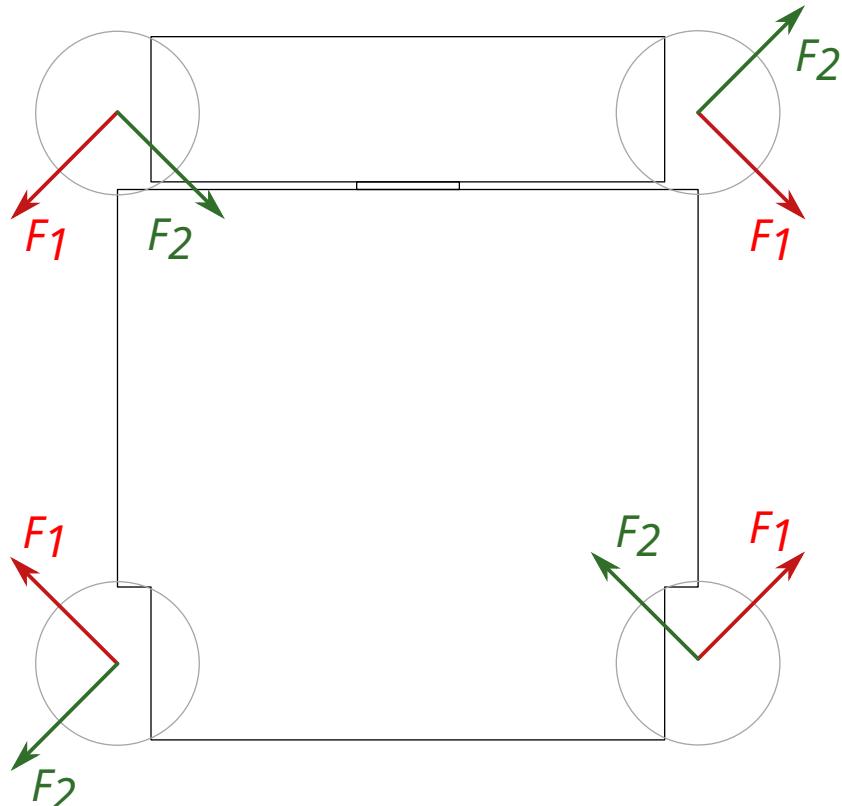
Rysunek 4.4: Wektory punktu kolizji.

Po wykryciu punktu kolizji i wyznaczeniu wektora normalnych  $N$  do dotykających się obiektów, system powinien zadziałać odpowiednimi siłami, aby zatrzymać, lub odbić obiekty od siebie. Dodatkowo, ponieważ prędkości obiektów nie muszą być równoległe do wektora kolizji, należy zasymulować siłę tarcia z odpowiednią dla współczynnika tarcia wartością. Można to uzyskać, nadając obiektom w punkcie kolizji siłę prostopadłą do wektora normalnych, ten wektor może być rozpisany przy pomocy dwóch wektorów jednostkowych  $F_1$  i  $F_2$ . Te wektory zawsze są prostopadłe do wektora normalnych, równoległe do płaszczyzny kolizji.

W normalnej symulacji fizyki nigdy nie potrzeba osobno modyfikować współczynników tarcia i kierunku tych wektorów, gdyż zazwyczaj powierzchnie symulowanych obiektów mają równe 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, prostopadle do kierunku jazdy, należy dynamicznie zmieniać współczynnik tarcia dla wektora  $F_1$ , lub  $F_2$  w kierunku promienia koła. Ten współczynnik tarcia, prostopadły do kierunku jazdy, może być liniowo zależny od prędkości. Spowoduje to, że im większa prędkość samochodu, tym boczna siła odśrodkowa bardziej wpłynie na tor jego jazdy, co ma odzworowanie w rzeczywistości. Więcej informacji można znaleźć na stronie instrukcji maszyny symulacyjnej ODE [16].

W opisywanym tutaj modelu, modyfikuje się wektor  $F_1$ , oraz współczynniki tarcia w obu kierunkach, aby przybliżyć zachowanie się rolki. Ponieważ wektory  $F_1$  i  $F_2$  są określone w lokalnym dla koła układzie współrzędnych,

w każdej iteracji maszyny symulacji należy obrócić ją względem aktualnej pozycji bazy i odwrotności obrotu koła. Idealna rolka obraca się całkowicie bez tarcia, a ruch wzdłuż jej osi jest niemożliwy. Można więc ustawić zerowy współczynnik tarcia w kierunku prostopadłym do osi, oraz nieskończonie duży dla wektora równoległego do osi.



Rysunek 4.5: Kierunki wektorów dla których należy nadać współczynniki tarcia przy symulacji platformy, widok z góry. Tarcie w kierunku  $F_1$  powinno być nieskończone, a w  $F_2$  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, trzeba użyć dużej siły, aby obrócić dowolną z nich, pod naciskiem platformy tarcie jest jeszcze większe. Każda rolka obraca się z innym tarciem wprowadzając kolejne zakłócenia. Podłożę 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 zamontowana jest kula z dynamicznie modyfikowanym tarciem i siatką w kształcie koła do wizualizacji, oraz przegub z motorem łączący odpowiednią część bazy z kołem. To najprostsza budowa modelu (a zatem najszybsza) z poprzednich.

```
kadłub.....Podstawa bazy
└─ przegub z silnikiem.....Nadaje moment obrotowy na żądanie
    └─ kula.....Modyfikowane wektory tarcia
        └─ siatka.....Odpowiada za wygląd obiektu koła
```

Rysunek 4.6: Zagnieźdżenie obiektów koła w strukturze drzewiastej z modyfikowanymi wektorami tarcia. W implementacji Gazebo przegub i kula są zagnieździone równolegle.

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 ten sposób i ma modyfikowalne wektory, lecz nie daje podobnych wyników. Być może jest to spowodowane brakiem odpowiedniej konfiguracji, lub innym wewnętrznym traktowaniem modelu.

#### 4.3.5 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 aktualną pozycją i rotacją 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 rotacją i pozycją, z nagłówkiem. To jest symulator enkoderów wbudowanych w silniki platformy.

Przyjmowane są także dane:

- `omnivelma_msgs/Vels` z zadanymi prędkosciami kół.

- Wywołanie ustawiające współczynniki tarcia wzdłuż wektorów  $F_1$  i  $F_2$ .
- Wywołanie ustawiające masy i momenty obrotowe niektórych elementów składowych konstrukcji.

#### 4.3.6 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 modelom 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 prędkosciami 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 ROSoswych pakietach, na dane zrozumiałe przez sterownik silnika, które zazwyczaj są liczbami stałoprzecinkowymi, program może zachować się nieprzewidywalnie.

### 4.4 Model czujnika laserowego

Ponieważ czujniki 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. Tak, jak model platformy, ten pakiet otrzymał nazwę kodową *monokl*, ponieważ pozwala obserwować otoczenie, jak okular.

#### **4.4.1 Obliczenia symulatora**

Czujnik 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.

Półprosta 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 półprostą 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 renderowanie spowalnia symulację około czterokrotnie. To ze względu na bardzo dużą ich ilość, mogącą przekroczyć 1000 obliczeń kolizji w jednej klatce symulacji.

#### **4.4.2 Różnice między czujnikiem, a modelem**

Półprosta emitowana jest z punktu reprezentującego środek czujnika. Model upraszcza rzeczywisty czujnik (budowa czujnika laserowego została opisana w sekcji 3.3). Uproszczenie to polega na tym, iż nie ma wewnątrz 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 zauważać tym samym, że model czujnika wydaje się funkcjonalnie lepszym, niż rzeczywisty LiDAR. W danej chwili, model emituje promień we wszystkich kierunkach w zakresie jednocześnie, podczas gdy czujnik 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 grupach. Czujnik jest w stanie wysłać pakiet z danymi z ostatniego pomiaru, podczas gdy program modelujący czujnik jest obsługiwany na zasadzie przerwań czasowych po każdej klatce 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 wiadzenia symulacji), odległość pomiaru. Nie tylko jako najdalszy wykryty punkt, ale także i najbliższy. Czujnik może pomijać pomiary przypadające za blisko krawędzi dozwolonego obszaru, gdyż znacznie spada w tych miejscach dokładność pomiaru, lub zwracać 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. Czujnik zmienia swoje błędy, w zależności jak daleko od niego znajduje się obiekt.

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

#### 4.4.3 Komunikacja

Bazując na architekturze opisanej wcześniej na rysunku 1.1, należy tak zbudować system, aby program sterujący mógł się komunikować w identyczny sposób z modelem czujnika, jak i samym czujnikiem. Służą do tego specjalne typy wiadomości ROSa `sensor_msgs/LaserScan`. Program obsługujący model czujnika generuje i wysyła pakiety zawierające:

- Nagłówek z czasem pomiaru, identyfikatorem i ramką 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 czujnika.
- Dane odległości.
- Dane jasności (jeśli czujnik posiada taką funkcjonalność).

Identycznie, program podłączony bezpośrednio do czujnika za pomocą jednego z interfejsów, także powinien generować takie same pakiety i udostępniać je w środowisku ROSa.

#### 4.4.4 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, za pomocą stałego połączenia (elementu `joint`).

Korpus posiada siatkę, reprezentującą uproszczony wygląd urządzenia, a także dwa elementy ustawiające walcowate kształty, odpowiedzialne za kolizje fizyczne. Teoretycznie, lepiej było by, aby model posiadał jeden walec, reprezentujący kształt urządzenia, gdyż to przyspieszyłoby symulację. Jednakże, półproste emitowane ze środka obiektu, również się by z nim zderzały od wewnętrz, a co za tym idzie, nie opuszczaliby modelu czujnika. 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 wiązu. Jak już wcześniej wspomniano, model zawsze ma strukturę gwiazdową i więzy po stronie robota nie mogą wskazywać na element `model` czujnika, a mogą na obiekt korpusu.

Główna część obiektu czujnika, skaner, posiada ozdobną siatkę, 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 jakimkolwiek innymi modelami. Również czujniki nie są w stanie wykryć siebie nawzajem, gdyż zwrócone są do siebie martwymi kątami, a co za tym idzie nie muszą symulować nieprzezroczystych brył dla innych sensorów. W przeciwnym wypadku, element fizycznego kształtu pośrodku urządzenia byłby wymagany.

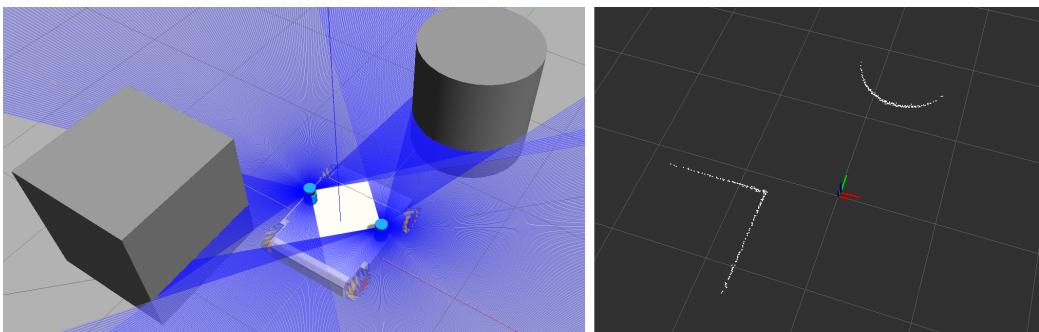
#### Połączenie modeli

Jak wcześniej wspomniano w sekcji 4.1, model SDF ma strukturę gwiazdową. Zagnieżdżenie modeli spowodowałoby, że powstałaby inna struktura, drze-

wiasta. Dlatego też, element `import` nie umieszcza w swoim miejscu całego modelu z innego pliku, a raczej importuje jego składowe 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 składowych importowanego modelu. Element importowanego modelu jest `tracony`, pozostaje jedynie przedrostek nazwy w zaimportowanych 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 gwiazdą i każdy element musi być odpowiednio połączony z pozostałymi, aby dokładnie określić fizykę interakcji. Nie powstają niedopowiedziane sytuacje, w których zachowanie jakichś elementów byłoby nieokreślone.

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



Rysunek 4.7: Zrzut ekranu platformy z Gazebo i wygenerowane dane, obserwowane w Rviz.

## Mechanika ramek

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

Ramka transformacji jest informacją o aktualnej pozycji i rotacji jakiegoś obiektu względem innego. Polega na wysłaniu pakietu typu `geometry_msgs/TransformStamped` prosto do demona ROS. Pakiet zawiera:

- Nagłówek z czasem nadania ramki i identyfikatorem, oraz informacją względem jakiej ramki podane są poniższe dane.
- Nazwa nowej ramki, jaka powstanie po zastosowaniu podanej transformacji do określonej w nagłówku ramki.
- Lokalna pozycja.
- Lokalna rotacja.

Demon ROSa następnie zbiera wszystkie dane ze wszystkich nadających komponentów i oblicza hierarchię transformacji obiektów. Zwraca te dane na pytania od innych komponentów.

Przykładowo, gdyby symulacja robota nie odbywała się w przestrzeni wirtualnej, w maszynie symulacyjnej fizyki, informacja o dokładnym położeniu obiektu składowego w lokalnym układzie współrzędnych wcale nie musiałaby być łatwo dostępna. Ma to szczególné znaczenie dla skomplikowanych mechanizmów, na przykład wielosegmentowego ramienia manipulacyjnego. Obliczenie pozycji i rotacji końcówki ramienia wymagałoby informacji o aktualnych pozycjach i rotacjach wszystkich segmentów. Która część systemu miałaby zajmować się obliczeniami i jaki kod powinien posiadać i gdzie przekazywać te informacje?

Demon ROSa działa tutaj jak trzecia strona, zbierająca dane od przegubów i obliczającą pozycje i rotacje wszystkich punktów. W takim przypadku, każdy segment symulacji mógłby przekazywać swój identyfikator, identyfikator obiektu którym steruje, jego pozycję i rotację 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 ramek jest potrzebna, gdyż pakiet zwierający pomiary z czujnika laserowego nie posiada informacji o aktualnej pozycji samego czujnika w przestrzeni, a jedynie identyfikator ramki czujnika. Pozycja potrzebna jest programowi obliczającemu pozycję z czujników i ewentualnemu wizualizatorowi samych danych.

Symulator platformy zawiera drugi program, który w każdym cyklu symulacji nadaje demonowi ROS pozycje i rotacje środków czujników laserowych, dla uproszczenia względem początku układu współrzędnych, punktu (0,0,0). Program sterujący modelem samej platformy także nadaje ramkę z

Punkt ramki	Nazwa punktu
Stałý środek mapy	map
Środek platformy	omnivelman
Środek platformy kinematycznej	pseudovelma
Emiter prawego lasera	monokl_r_heart
Emiter lewego lasera	monokl_l_heart

Tablica 4.2: Nazwy identyfikatorów ramek, używanych w symulatorze.

Nazwa	Punkt względny	Punkt danych
Pozycja i rotacja platformy	map	omnivelman
Pozycja i rotacja platformy kinematycznej	map	pseudovelma
Pozycja i rotacja prawego czujnika	map	monokl_r_heart
Pozycja i rotacja lewego czujnika	map	monokl_l_heart

Tablica 4.3: Ramki wysyłane do demona ROS.

pozycją i rotacją platformy względem globalnego środka układu współrzędnych. Dokładnie taki sam efekt byłby, gdyby nadawać stałą pozycję i rotację czujników laserowych, ale względem ramki platformy (nadawanej przez inny sterownik). Stałą, ponieważ czujniki nie zmieniają swojej pozycji na platformie, są przytwierdzone na stałe.

#### 4.4.5 Błędy

Jak podano wcześniej w tabelce 3.1, wyróżnione są dwa typy błędów pomiaru, systematyczny i pomiarowy. 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. Nie mniej jednak, to zadanie należy do programu sterującego, więc należy umożliwić mu testowanie tej funkcjonalności poprzez wprowadzenie takich błędów do zasymulowanych odczytów.

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. Czujnik 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 powinien mieć interfejs do ustawienia tej wartości, aby mógł być „skalibrowany” w taki sam sposób, jak faktyczne urządzenie.

### 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 3.1, 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++ w 2011 roku.

Na zrzucie ekranu 4.7 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.

## 4.5 Model czujnika inercji

Ponieważ czujniki tego typu są często stosowane w robotyce, wszystkie komponenty systemu wspierają jego symulację i struktury przekazywanych danych.

- ROS posiada specjalną wiadomość typu `sensor_msgs/Imu`, do przekazywania pomiarów pomiędzy komponentami.
- 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.

Ten czujnik podłączony jest do platformy dynamicznej.

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 kwaternion, 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 wzdułż ich 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 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 w maszynie symulacji. Wtyczka do Gazebo, zapisana podobnie do wtyczki czujnika

laserowego, zbiera w każdym cyklu maszyny symulacyjnej dane i wysyła je za pomocą pakietu ROSa.

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. W sekcji 6.3, sekcji testów, opisane zostały te problemy dokładniej.

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 odszumienia wygenerowanych wartości. Ten komponent jest opisany w sekcji 5.5.

## Rozdział 5

# Komponenty systemu

Aby uruchomić symulację, nie wystarczy uruchomić Gazebo z modelami, należy zadbać także o odpowiednie przekazywanie informacji pomiędzy pakietami. Wskazane jest przetestować modele, czy zachowują się poprawnie w prostych scenariuszach testowych, tak samo, jak testować się będzie program sterujący na modelu. Do tego potrzebne są programy wspomagające, które łączy się w różne konfiguracje, w zależności od scenariusza testowego. Ze względu na niezależność komponentów od siebie, można je także użyć przy komunikacji z rzeczywistym robotem.

Niektóre typy wiadomości mają dopisek **Stamped**, co oznacza że zawierają także nagłówki. Nagłówek posiada trzy pola:

- Numer sekwencyjny, zwiększany przez program wysyłający po każdym pakiecie.
- Czas nadania pakietu, z dokładnością do nanosekund.
- Identyfikator ramki, według której podano dane, ramki zostały opisane dokładniej w sekcji 4.4.4.

Komponenty można podzielić na trzy typy:

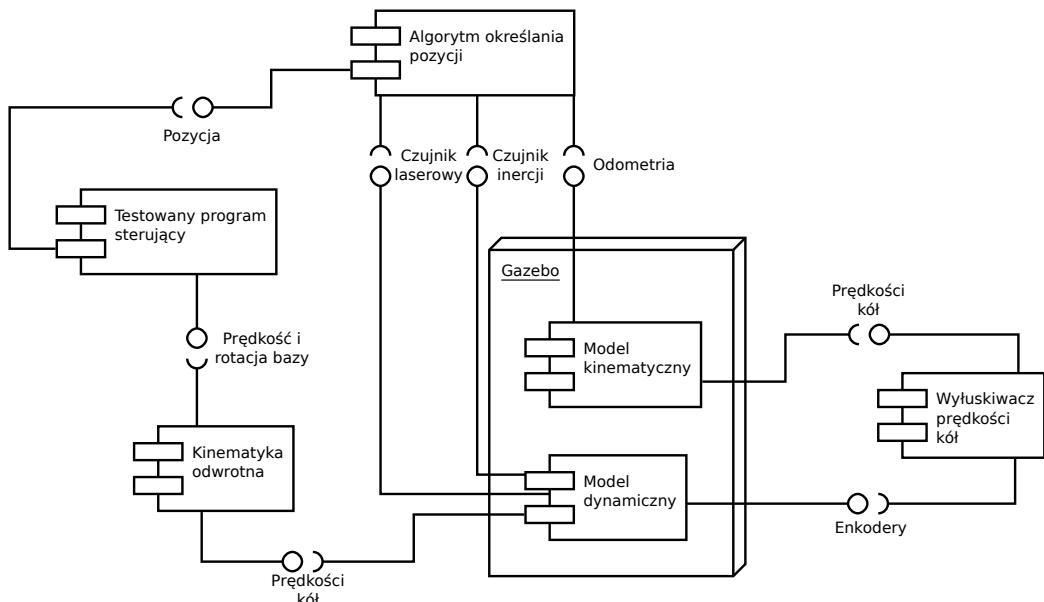
- Generujące pakiety danych.
- Przekazujące i modyfikujące pakiety danych.
- Zbierające pakiety danych.

Poniżej, każdy pakiet opisany jest bardziej szczegółowo, ten dokument nie ma za zadanie być programistyczną dokumentacją komponentów, dlatego nie zagłębia się dokładnie w argumenty, algorytmy i technologie programów.

Typ	Opis
omnivelman_msgs/Encoders	Prędkości i pozycje kół z enkodera.
omnivelman_msgs/Vels	Prędkości kół.
omnivelman_msgs/SetFriction	Nadanie tarcia elementowi modelu.
omnivelman_msgs/SetInertia	Nadanie mas i momentu bezwładności obiektowi.
geometry_msgs/Pose	Pozycja i rotacja obiektu w przestrzeni kartezjańskiej.
geometry_msgs/Twist	Prędkość względna obiektu.
sensor_msgs/LaserScan	Jedno skanowanie LiDARa, wraz z nagłówkiem.
omnivelman_msgs/Relative	Odległość i kąt pomiędzy obiekty.
nav_msgs/Odometry	Transformacja obiektu z macierzą kowariancji.
sensor_msgs/Imu	Dane generowane przez czujnik inercji.

Tablica 5.1: Typy i opisy wiadomości przekazywanych pomiędzy komponentami.

W ostatecznym działaniu symulatora, podłączenie komponentów będzie wyglądać następująco:



Rysunek 5.1: Komunikacja podstawowych komponentów systemu w trakcie testowania programu sterującego.

Ważną rolę odgrywa tutaj algorytm określania pozycji, bazujący na odometrii, czujniku inercji i danych z czujnika laserowego. Odometria jest generowana za pomocą modelu kinematycznego, sterowanego danymi z enkode-

rów modelu dynamicznego. Sam model dynamiczny sterowany jest pośrednio przez program, który generuje zadane prędkości i obrót 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 z czujników laserowych w celu wykrycia przeszkody, nie tylko w celu określenia pozycji. Bardziej zaawansowany program sterujący mógłby generować zadane prędkości kół bezpośrednio.

Komponenty modeli platform i czujnika laserowego, zostały opisane szczegółowo w poprzednich sekcjach.

`omnivelman` Model platformy dynamicznej w sekcji 4.3.

`pseudovelma` Model platformy kinematycznej w sekcji 4.2.

`monokl` Model czujnika laserowego w sekcji 4.4.

## 5.1 Manualne sterowanie

To zaawansowany program do manualnego generowania zadanych prędkości, lub kierunku platformy. Ponieważ jest niezależny od reszty systemu, może być użyty do sterowania rzeczywistym robotem. Pozwala także na wyświetlanie aktualnych prędkości kół, generowanych przez enkodery. Ma nazwę kodową `lalkarz`, ponieważ steruje platformą, tak jak aktor steruje marionetką.

### 5.1.1 Program

Ten komponent jest plikiem wykonywalnym, skompilowanym ze źródeł w C++. 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 mógłaby bez większego problemu pracować z interfejsem tekstowym w terminalu, aby być bardziej przenośna i lżejsza w zasobach, lecz nie mógłaby wykrywać zdarzeń puszczenia klawisza (bez bezpośredniego

czytania z urządzenia `/dev/input/eventX`, do czego są potrzebne prawa rota). 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.

### 5.1.2 Komunikacja

Program potrafi generować dwa typy wiadomości.

Pierwszą są prędkości kół `omnivelm_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 (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 ani model platformy, ani rzeczywisty robot nie są w stanie poruszać się bez informacji, jakimi kołami z jaką prędkością obracać, ten typ wiadomości musi być jeszcze konwertowany przez komponent modelu kinematyki odwrotnej, opisany w sekcji 5.7.

Program opcjonalnie przyjmuje także wiadomość `omnivelm_msgs/Vels`, aby wyświetlić dane enkoderów. Należy zauważyć, że nie przyjmuje całego pakietu `omnivelm_msgs/EncodersStamped`, jaki jest generowany przez model platformy, a jedynie mały ich wycinek, gdyż tylko te informacje jest w stanie wyświetlić i tylko takie potrzebuje. Dzięki temu może być użyty niezależnie od innych komponentów i programów. Jednak może być wymagane użycie dodatkowego komponentu do wyłuskania tej informacji z większego pakietu, patrz 5.3.

### 5.1.3 Tryby działania

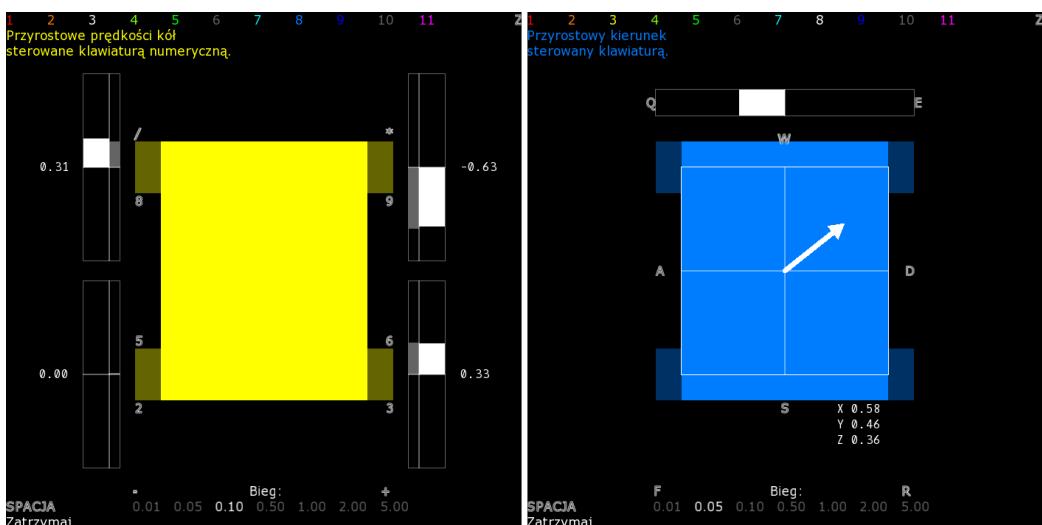
Program posiada 11 trybów działania, w których generuje różne wiadomości w różny sposób. Jedne są bardziej przydatne, inne bardzo proste i służące do ogólnej prezentacji systemu. Globalny mnożnik wyjścia pozwala na łatwe ograniczenie generowanych danych i ustawienia dokładności. Spacja awaryjnie zeruje wszystkie wyjścia.

1. Za pomocą ośmiu klawiszy klawiatury numerycznej, można nadać plat-

formie określone prędkości kół. W tym trybie, koło może albo stać w miejscu, albo obracać się z odpowiednią prędkością, w określonym kierunku. Powoduje to naturalne szarpnięcia i poślizgi platformy. W trakcie braku aktywności użytkownika, platforma stoi w miejscu.

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 4.3.6.
3. Naciśnięcie klawisza płynnie zwiększa, lub zmniejsza prędkość koła. W trakcie braku aktywności użytkownika, platforma porusza się z ustawnionymi prędkosciami kół.
4. Podobnie, co w poprzednim trybie, lecz pozwala na schodkowe ustawienie prędkości kół co  $0,1 \frac{rad}{s}$  (pomnożone przez dokładność). Dzięki temu, możliwe jest w miarę dokładne powtórzenie manualnych testów platformy.
5. Poprzedni tryb, lecz przy ustawieniu prędkości zerowej, generuje nie-liczbę.
6. Sterowanie prędkosciami kół za pomocą gałek kontrolera. Większość kontrolerów posiada dwa, dwuosiowe joysticki, co daje cztery osie, zmieniające się w zakresie  $\langle -1; 1 \rangle$ . Można za ich pomocą bezpośrednio ustawiać prędkości kół, chociaż jest to nieintuicyjne w działaniu.
7. Lokalny kierunek jazdy platformy składa się z dwóch wektorów prędkości liniowej i wektora obrotu. Za pomocą klawiatury, binarnie, można nadać platformie jeden z ośmiu kierunków poruszania się i jeden z dwóch kierunków obrotu. Ponownie, 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 kierunku poruszania się i obrotu platformy. Brak aktywności użytkownika powoduje, że model porusza się zadaną prędkością w zadanym kierunku i z zadanym obrotem.
9. Połączenie dwóch poprzednich trybów, schodkowe sterowanie prędkością platformy. Pozwala na ustawienie prędkości i obrotu platformy z zadaną dokładnością.

10. Sterowanie kierunkiem platformy za pomocą kontrolera. Trzy osie są używane, dwie do nadania prędkości, jedna do nadania obrotu. Jest to prawdopodobnie najczęstszy sposób kontrolowania robotów wielokierunkowych za pomocą kontrolera. Bardzo intuicyjny i używany także przez inne pakiety do manualnego sterowania robotami na kołach Me-canum.
11. Sterowanie za pomocą myszki, najdokładniejsze sterowanie kierunkiem, mniej dokładne obrotem. Kursor myszy wskazuje końcówkę strzałki reprezentującej kierunek ruchu robota, za pomocą kółka, można dodawać, lub odejmować prędkość do obrotu wokół osi. Ponieważ większość myszek ma skokowe obroty kółek, wprowadza to nieznaczne poślizgi. Można także modyfikować obrót klawiaturą w płynnym trybie przyrostowym.



Rysunek 5.2: Zrzuty ekranu dwóch trybów działania programu.

Interfejs 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 współczynnikiem wypełnienia. Obok nich znajdują się szare wskaźniki prędkości, zwrócone przez modele enkoderów. Małe, szare znaki to nazwy klawiszy, używanych w tym trybie do modyfikowania prędkości.

Na prawym obrazku jest tryb generowania kierunku i obrotu. Strzałka wskazuje wektor prędkości, a górny pasek obrót platformy.

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

Wszystkie dane są w jednostkach SI, tzn, efektywna prędkość koła będzie się równać liczbie podanej przy kole, pomnożonej przez aktualny bieg. Dla obrotów to są  $\frac{rad}{s}$ , dla prędkości to  $\frac{m}{s}$ .

## 5.2 Generator sterowania

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

Program `gramofon` 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 w 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. Ta wiadomość powtarza się regularnie.

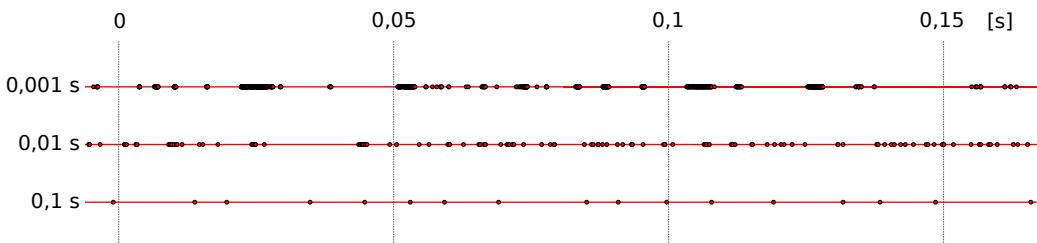
Po upływie czasu  $t$ , program nadaje dodatkową wiadomość z nowymi danymi, oraz przestawia dane generowane przez wywołania pierwszego licznika.

Po wykorzystaniu danych, algorytm nadal generuje wiadomości z zero-wymi prędkościami, 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  $0,2 \frac{m}{s}$ , w kierunku (2,1).
2. Zatrzymanie na czas 0,9 s.
3. Obrót przez czas 10 s, z prędkością kątową  $0,02 \frac{rad}{s}$ .

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

Działa to tak samo, jakby każdą wiadomość wysyłać z losowym opóźnieniem. Gdy nadań jest za dużo (górny wykres), system zaczyna je buforować, a co za tym idzie, nadaje je w grupach o losowej wielkości. Pakiet po nadaniu przechodzi przez wiele procesów, każdy może nadawać losowe opóźnienie dla wiadomości. System operacyjny pod dużym obciążeniem przez inne procesy będzie opóźniał i buforował wiadomości o niższych częstotliwościach.

Aby rozwiązać ten problem, należałyby uruchomić całe środowisko ROSa na systemie operacyjnym czasu rzeczywistego. Jedynym takim systemem, eksperymentalnie wspieranym przez twórców ROSa, jest OpenEmbedded. Jednakże budowa i uruchomienie tego systemu jest bardzo złożone.

### 5.3 Wyłuskanie struktury wiadomości

Każda wiadomość przekazywana pomiędzy komponentami jest zwykle zagnieżdżoną strukturą. Na przykład pakiet typu `geometry_msgs/Twist` składa się z dwóch podstruktur wektorów trójwymiarowych. Jeden odpowiada za prędkość, a drugi za rotację.

Czasami może zdarzyć się, że jakiś komponent potrzebuje jedynie wewnętrznej podstruktury pakietu. Nie powinno mu się zatem przekazywać całej struktury wiadomości, gdyż to powodowałoby niepotrzebne opóźnienia, oraz nie pozwoliłoby zachować niezależności komponentu od innych.

Takie zjawisko występuje przy przekazywaniu informacji o pozycji i prędkości kół, generowanej przez model czujnika enkoderów, do programu manualnego sterowania, opisanego w 5.1.

ROS nie pozwala na automatyczne odbieranie tylko części pakietu, dla tego powstał program o nazwie kodowej `dziadzio`, niczym dziadek do orzechów. Przyjmuje wiadomości typu `omnivelma_msgs/EncodersStamped` i zwraca wiadomości typu `omnivelma_msgs/Vels`, który to typ jest zawarty wewnątrz przyjmowanej struktury. Pozwala to na połączenie modelu czujnika enkoderów z wyświetlaczem prędkości kół w programie do manualnego sterowania.

## 5.4 Podłoga o zmiennym współczynniku tarcia

Symulacja nie składa się jedynie z robota i czujnika, ale także z podłożą, na którym musi się poruszać. Ponieważ podłoże również wpływa na symulację, powinien istnieć sposób na ustawienie jego współczynnika tarcia. Robi się to wewnętrznie w identyczny sposób, jak w przypadku kół platformy, co zostało opisane w sekcji 4.3.4.

W tym przypadku jednak powinno się ustawić identyczne wektory tarć  $F_1$  i  $F_2$ , aby podłoże symulowało równe tarcie we wszystkich kierunkach. Tak samo, jak w przypadku modelu platformy dynamicznej, program nadający podłożu odpowiednie wektory tarcia przyjmuje asynchroniczne wywołanie typu `omnivelma_msgs/SetFriction`, zawierające dwie wartości zmiennoprzecinkowe.

Nazwa kodowa tego programu sterującego to `flooria`, od angielskiej nazwy na podłogę. Program jest uruchamiany jako biblioteka symulatora Gazebo.

## 5.5 Algorytm usuwania szumu z danych modelu czujnika inercji

Jak wcześniej wspomniano, różniczkowanie prędkości w maszynie symulacji fizyki generuje duże błędy.

Ten program uśrednia dane w prosty sposób, licząc średnią z określonej ilości poprzednich pomiarów. Odczytuje nazwę nadajnika i odbiornika wiadomości typu `sensor_msgs/Imu`, oraz wielkość bufora,

Taki algorytm nie sprawdza się jednak, gdy dane są generowane naprzemiennie. Na przykład, jeśli w jednej klatce symulacji maszyna do symulacji fizyki obliczy prawidłową wartość, a w drugiej zwróci zerową, to ten algorytm uśredni te wyniki i zwróci wartość pośrodku.

Jednak stworzenie tego komponentu pozwoliło sprawdzić, czy model czujnika inercji reaguje na ruch platformy z określonymi przyspieszeniami. Po-

zwolił także na odkrycie innych cech generatora. Używany jest w przeprowadzeniu testów w sekcji 6.3. Nazwa kodowa to `odszumiacz`.

## 5.6 Obserwator symulacji

Kolejny program uruchamiany w symulatorze Gazebo, oblicza i zwraca ciąg danych, reprezentujący odległość i kąt pomiędzy platformą dynamiczną i kinematyczną, pakiet `omnivelma_msgs/Relative`. Te dane pozwalają sprawdzić, na ile symulacja fizyczna opóźnia się względem matematycznego modelu.

Ten program nie może być zaimplementowany jako zewnętrzny komponent, 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 pakietów, dodatkowo zachowując pewność że oba pochodzą z tej samej klatki symulacji.

Nazwa kodowa tego programu to `ocznica`.

## 5.7 Model kinematyki odwrotnej

Jest to model kinematyki odwrotnej, alternatywa do modelu kinematycznego, opisanego w sekcji 4.2, jednak działającego bez symulatora i bez możliwości całkowania prędkości (i generowania danych o aktualnej pozycji). Ten komponent przyjmuje zadaną prędkość, kierunek i obrót platformy, a zwraca prędkości kół, które powinny być nadane platformie, aby wywołać taki ruch.

$$\begin{bmatrix} \omega_1 \\ \omega_2 \\ \omega_3 \\ \omega_4 \end{bmatrix} = \frac{1}{r} \begin{bmatrix} 1 & -1 & \frac{a+b}{2} \\ 1 & 1 & -\frac{a+b}{2} \\ 1 & -1 & -\frac{a+b}{2} \\ 1 & 1 & \frac{a+b}{2} \end{bmatrix} \begin{bmatrix} v_y \\ v_x \\ \omega_z \end{bmatrix} \quad (5.1)$$

Stałe, użyte we wzorze zdefiniowane są w tabeli 4.1, a numerowanie kół na rysunku 4.1. Ten wzór, podobnie jak poprzedni, pojawia się w wielu pracach, na przykład [2], dokładny wygląd macierzy zależy od numerowania kół i interpretacji kierunków osi.

Dodatkowo, program pozwala na obrót wektora prędkości o kąt prosty, lub półpełny. Jest to spowodowane tym, że różne komponenty i różne modele przyjmują różną pozycję wyjściową robota. Czasami przed 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.

Nazwa kodowa to **transmutator**, wykonuje transmutację jednego typu prędkości w inny.

## 5.8 Mapa z symulacją

Symulator Gazebo przy uruchomieniu ładuje plik zawierający referencje robotów i ich początkowe pozycje, używane w symulacji. Ten pakiet nie jest programem wykonywalnym, lecz prezentuje informacje dla symulatora o scenie symulacji.

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 5.6.

W tym pliku znajdują się także ustawienia symulacji, jak przyspieszenie grawitacyjne, typ maszyny symulacyjnej fizyki ze współczynnikami, czy ustawienia wirtualnej atmosfery.

Nazwa komponentu **velmaverse** jest zlepkiem słów „universe” i nazwy robota manipulacyjnego.

## 5.9 Rozdzielacz pakietów

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

Ten program wykonywalny pobiera i generuje wiadomości typu **omnivelma\_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.

Nazwa kodowa **widelnica** jest referencją do widelca którego końcówka rozdziela się na kilka części, a także do angielskiej nazwy „fork”, używanej w podobnych przypadkach rozdzielania informacji.

## 5.10 Prosty program sterujący

Jest to uproszczona wersja programu, który docelowo ma być tworzony na podstawie budowanego systemu modeli. Pozwala on sprawdzić, jak dla prostych zasad model będzie się zachowywał.

Program periodycznie wysyła wiadomość typu `geometry_msgs/Twist` o kierunku równoległym do osi współrzędnych. W zależności od danych z czujników laserowych, program zmienia swój stan i obraca kierunek obrotu o 90°. Ten sterownik dla uproszczenia nie generuje poleceń obrotu kątowego, model powinien być zawsze zwrócony w tę samą stronę.

Działanie programu oparte jest na zachowaniu akcja-reakcja. Dane z czujników 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.

Nazwa kodowa `pantofelek` pochodzi z tego, że zachowuje się jak taki pierwotniak.

## 5.11 Struktury pakietów wiadomości

Ten komponent nie jest plikiem wykonywalnym, a definicjami struktur danych, używanych przez wiadomości ROSa w projekcie, jeśli standard nie obejmuje potrzebnego typu wiadomości. Nazwa kodowa i jednocześnie przedrostek wszystkich zawartych typów to `omnivelman_msgs`.

## 5.12 Zewnętrzne pakiety

Istnieje kilka tysięcy różnych pakietów i programów, tworzonych przez społeczność ROSa.

### 5.12.1 Rysownik wykresów

Pakiet `rqt-multiplot` jest wtyczką do większego programu `rqt`. Pozwala na generowanie dwuwymiarowych wykresów, bazując na dwóch dowolnych war-

tościach z odbieranych pakietów, lub czasie. Można porównać różne wykresy na jednym układzie.

W szczególności przy ustawieniach pozycji Y względem X, pobranych z pakietu pozycji, nadawanego przez obie platformy, pozwala narysować trajektorię ruchu platform.

### 5.12.2 Wizualizer pomiarów

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

Pozwala to na przykład umieścić znacznik 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.

### 5.12.3 Zbieranie danych

Każdy `topic` może zostać zapisany do pliku, a następnie odtworzony w ten sam sposób. Wbudowane w ROSa narzędzie `rosbag` pozwala „nagrać” i odtworzyć dane dokładnie w taki sam sposób, w jaki zostały odebrane. Zapisuje to w formie pliku binarnego, wraz z dokładnymi parametrami działania nadajnika wiadomości. Możliwe jest również wydrukowanie danych do pliku tekstowego, aby mógł być wykorzystany przez inne programy jak Gnuplot, Calc (Exel), czy Matlab.

# Rozdział 6

## Testy systemu

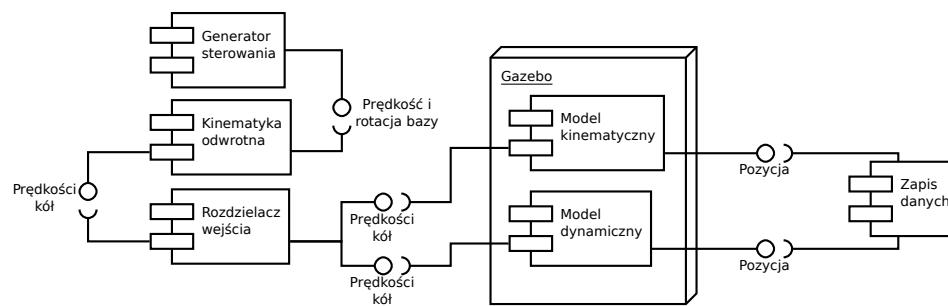
W tym rozdziale przedstawione są różne konfiguracje pakietów, wraz z wykresami ruchów platform, oraz wnioski płynące z tych zachowań. Każdy test wymaga innego podłączenia komponentów do siebie nawzajem.

Wyniki pomiarów zostały zebrane za pomocą ROSowego narzędzia `rosbag`, a następnie wyeksportowane do pliku CSV. Za pomocą programu Gnuplot narysowano wykresy.

### 6.1 Porównanie modeli dynamiki i kinematyki

Posiadając model kinematyczny, którego ruch jest sterowany wzorami, można porównać jego pozycję i rotację z modelem dynamicznym. Należy w tym samym momencie nadać bazom identyczne prędkości kół i zebrać dane dotyczące wzajemnej pozycji. Modele rozpoczynają jazdę z punktu (0,0), początkowo stoją 5 s w miejscu, aż symulator i wszystkie komponenty się załączają.

Te eksperymenty pozwalają sprawdzić, jak model dynamiczny zachowuje się w stosunku do idealnego modelu kinematycznego.

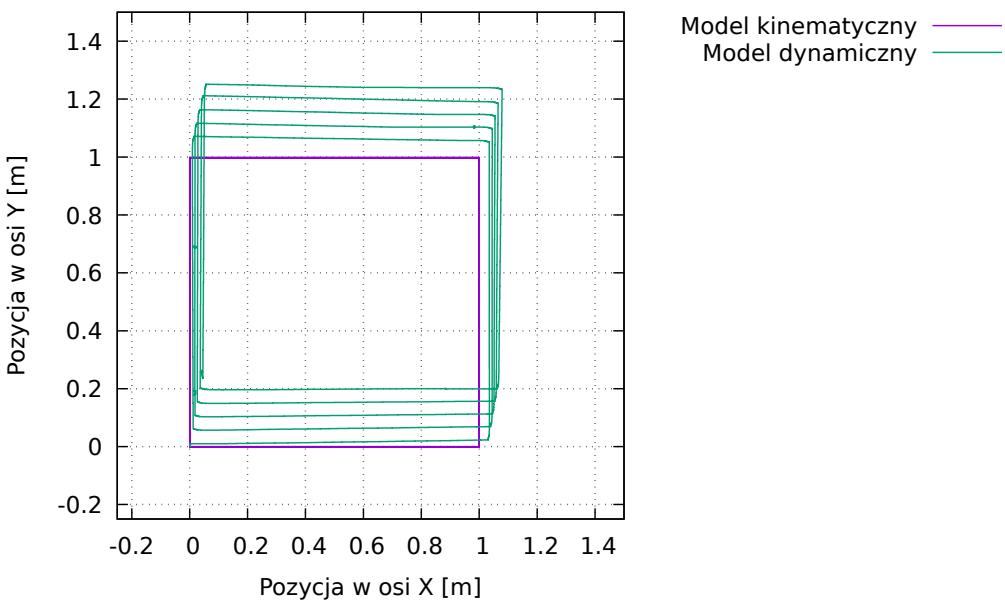


Rysunek 6.1: Połączenie komponentów w celu przetestowania wzajemnego ruchu modelu kinematycznego i dynamicznego.

### 6.1.1 Ruch po kwadracie

Wywołano ruch z prędkością  $0,2 \frac{m}{s}$  po kwadracie o boku 1 m, bez nadawania prędkości kątowej. Modele przejechały ścieżkę pięciokrotnie, po czym zatrzymały się. W kątach kwadratu nie było nadania prędkości zerowej.

Celem tego eksperymentu było sprawdzenie, czy model dynamiczny będzie wykazywał odchylenia w trakcie jazdy po prostej ścieżce i jak będzie reagował na nagłe zmiany kierunku jazdy.



Rysunek 6.2: Ruch modelu kinematycznego i dynamicznego po kwadratowej ścieżce.

Po pierwsze widać, że tuż przed rozpoczęciem jazdy model dynamiczny odsunął się o kilka centymetrów od pozycji początkowej i obrócił nieznacznie. Jest to spowodowane tym, że maszyna do symulacji fizyki działa na liczbach zmiennoprzecinkowych pojedynczej precyzji, zatem nie wszystkie siły działające na obiekty będą się dokładnie równoważyć i może istnieć mały ruch w losowych kierunkach i rotacja.

Następnie, po nadaniu ruchu w bok, platformy jechały po linii prostej. Model dynamiczny wykonał większą trasę do modelu kinematycznego, co nie jest spowodowane poślizgiem. Na takie zachowanie wpływa wiele czynników, zarówno wewnętrzna implementacja maszyny symulacyjnej fizyki, jak i natura użytych algorytmów.

Przy zmianie prędkości widać poślizg przy zatrzymywaniu się z danego kierunku. Kąty trasy są zaokrąglone w jednym kierunku, co pokazuje że

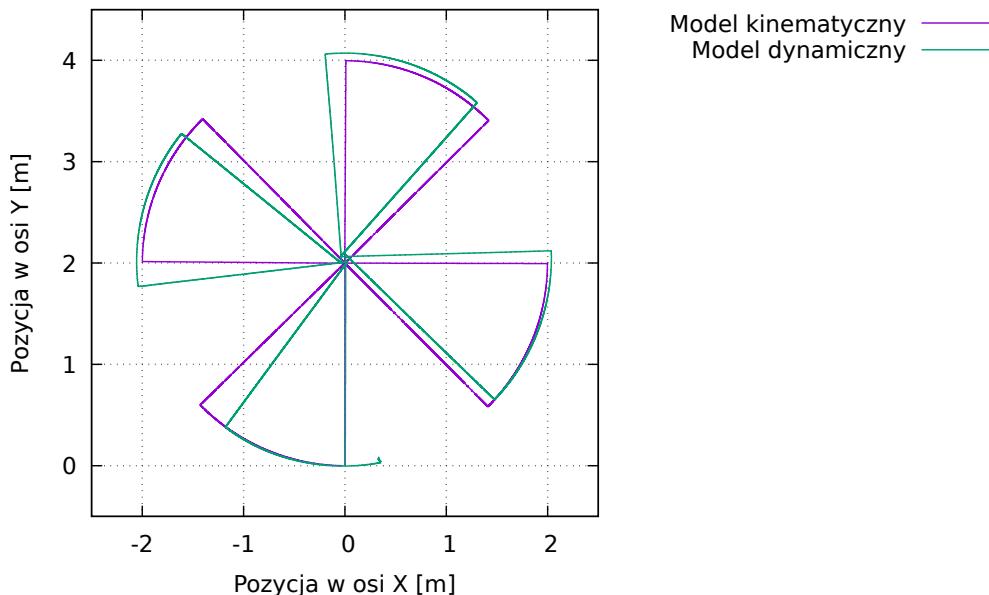
model dynamiczny posiada bezwładność. Gdy otrzymuje nowe prędkości kół, nadal porusza się jeszcze przez pewien czas w poprzednim kierunku. Pośród przy ruszaniu jest mniejszy. To zjawisko nie wpływa istotnie na trasę modelu, gdyż przy testach, w których prędkość zmieniała się wolno i w których nie występowały nagłe hamowania, nadal występowało takie samo zboczenie w stosunku do trasy kinematycznej.

Za każdym obiegem pętli narasta różnica pozycji wynikającej z kinematyki i pozycji obliczonej przez symulator. Po zatrzymaniu się modelu dynamicznego, ponownie porusza się on jeszcze przez pewien czas w losowym kierunku, innym niż na początku, aż do przerwania testu.

### 6.1.2 Ruch po „rozecie”

Drugi, bardziej skomplikowany test składa się na ruchy platform w przód i w tył, oraz obrót wokół punktu.

Modele jechały 2 m w przód z prędkością  $0,25 \frac{m}{s}$ , następnie następował obrót o  $45^\circ$ , po czym odbywała się jazda w tył. Na koniec ruch w bok przy jednoczesnym obrocie i powtórzenie cyklu. Czterokrotne wykonanie tego zamkna trasę.



Rysunek 6.3: Ruch modelu kinematycznego i dynamicznego po rozetowej ścieżce.

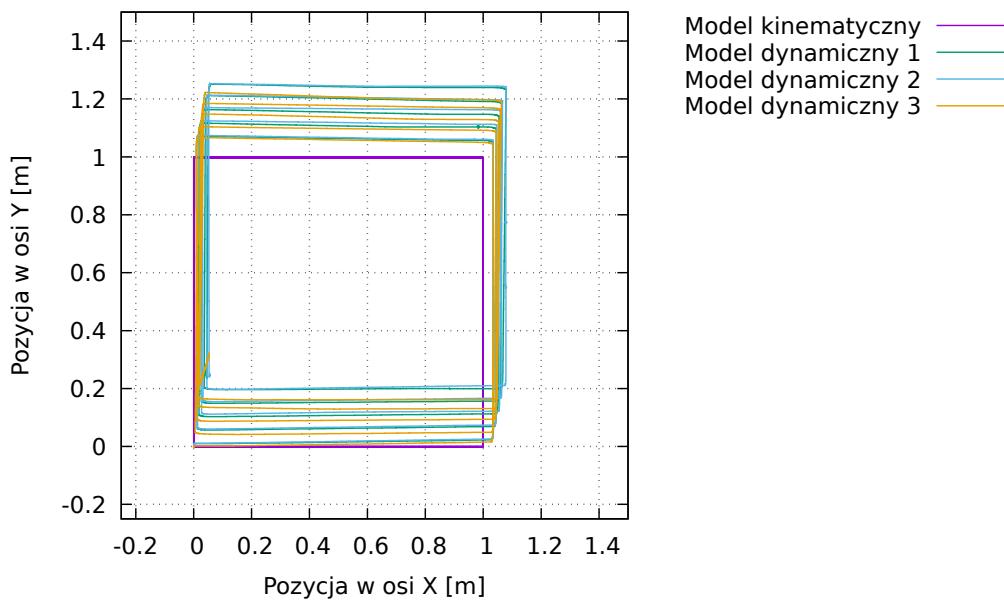
Z wykresu wywnioskować można, oprócz tego co z poprzedniego ekspe-

rymentu, to że model dynamiczny przegania model kinematyczny również w kwestii obrotu. W tym przypadku składowa trasy pionowej (w lokalnym układzie), czyli prędkość w przód i w tył, zniosła się. Obrót i ruch w bok były ciągle narastające, a co za tym idzie, spowodowały przesunięcie się modelu dynamicznego, co widać, porównując ich końcową pozycję.

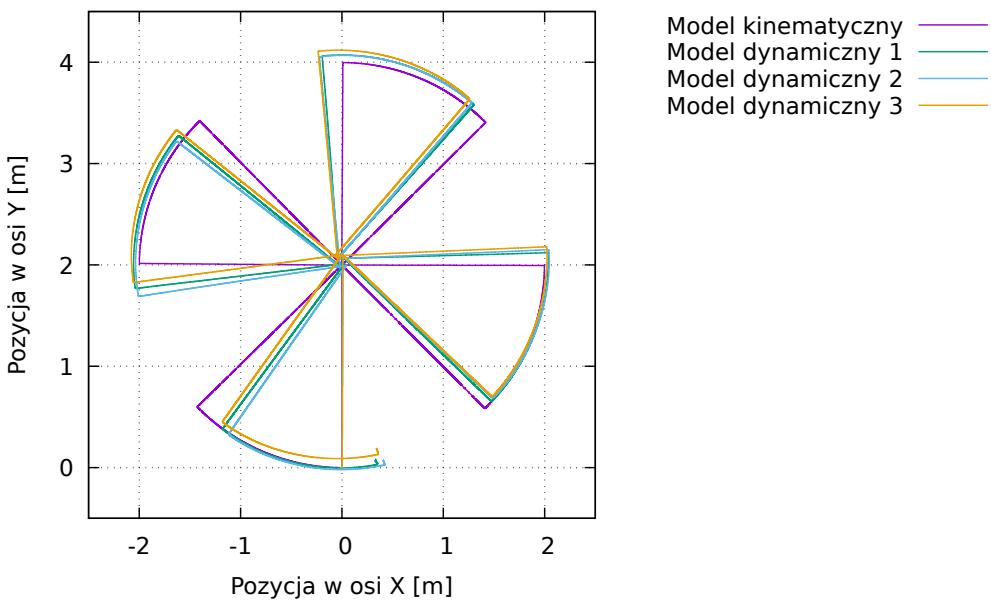
### 6.1.3 Powtarzalność testów

Pomimo, że na środowisko wirtualne w maszynie symulacyjnej fizyki nie działają żadne zjawiska zewnętrzne, nadal jej działanie zależy od wewnętrznych niedoskonałości systemu, na którym działa. W to wchodzą takie rzeczy, jak niedokładności w reprezentacji liczb zmiennoprzecinkowych, czas pomiędzy kolejnymi klatkami symulacji i niedeterministyczne przekazywanie wiadomości przez system operacyjny.

Zatem każdy test będzie generował nieco inne wyniki pozycji modelu zarówno kinematycznego, jak i dynamicznego. Jednak w kinematycznym przypadku różnice są niezauważalne. Przeprowadzono powyższe testy trzykrotnie, aby zbadać jak bardzo każdy z przejazdów modelu różni się od poprzedniego.



Rysunek 6.4: Wielokrotny ruch modelu kinematycznego i dynamicznego po kwadratowej ścieżce.



Rysunek 6.5: Wielokrotny ruch modelu kinematycznego i dynamicznego po rozetowej ścieżce.

Wygląda to tak, że odległość pomiędzy pozycjami modelu dynamicznego w różnych testach zwiększa się w czasie. Różnica narasta wraz z pokonaną odległością. Takie zjawisko jest typowe dla symulacji w których istnieje wiele czynników zewnętrznych, wpływających na symulację.

Aby zmniejszyć te niedoskonałości, należałyby, po pierwsze uruchomić symulację na systemie czasu rzeczywistego, a po drugie zwiększyć dokładność reprezentacji liczb zmiennoprzecinkowych, a co za tym idzie, znacząco zwiększyć obciążenie procesora.

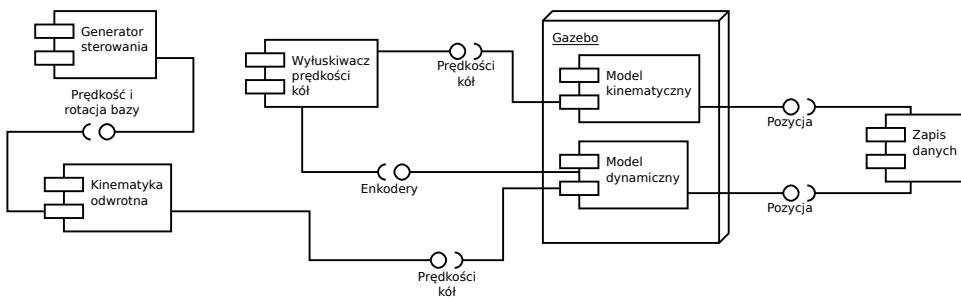
Nie można nadal pozwolić na to, aby symulacja przestała odbywać się w czasie rzeczywistym, gdyż to spowodowałoby opóźnienia w ruchu platform. Ponieważ, na przykład, jeśli program sterujący wywoła przez sekundę ruch z prędkością  $1 \frac{m}{s}$ , aby przejechać dystans 1 m, a obciążony symulator będzie nadawał tą prędkość przez ten czas, to w stosunku od obciążenia, może być to inny czas symulacji. Zatem patrząc z perspektywy modelu, sterowanie platformie nadawane będzie przez krótszy czas, niż zakłada to program sterujący i model przejedzie mniejszą odległość. Należałyby buforować wszystkie otrzymywane wiadomości i stosować je dopiero w czasie symulacji zgodnym z czasem odebrania pakietu. Ale tutaj także jest problem z generowaniem danych, nawet jeśli wygenerowane dane opatrzone będą nagłówkiem z czasem simulatora, to i tak program obierający będzie miał problem z zastosowaniem ich. Dochodzi do problemu synchronizacji programu sterującego i simulatora w

niestałym czasie. Wymagałoby to stosowania marszczenia czasu i podobnych technik.

## 6.2 Enkodery

Model czujnika enkoderów zwraca aktualną prędkość i pozycję kół modelu dynamicznego. W ten sposób jest możliwe dokładniejsze określenie pozycji platformy, niż bazując na modelu kinematycznym. Jednakże, ta metoda także ma swoje ograniczenia, ze względu na poślizgi, zatem program sterujący nie może w pełni bazować na tych czujnikach, a jedynie powinien używać ich pomocniczo.

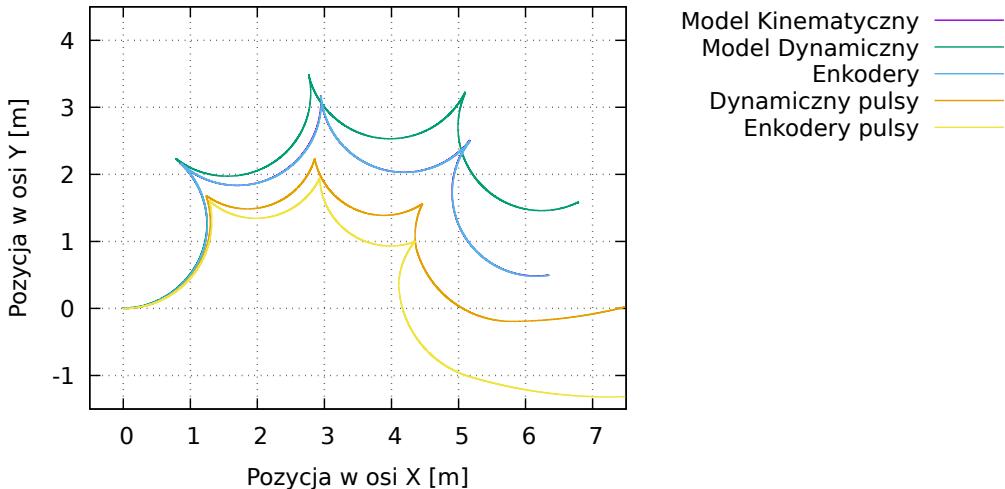
Ten test ma za zadanie sprawdzić, czy gdyby poruszać platformą kinematyczną tak, jak poruszają się koła platformy dynamicznej, to jak duża była by różnica pomiędzy pozycjami platform.



Rysunek 6.6: Połączenie komponentów w celu sprawdzenia poprawności działania enkoderów modelu dynamicznego.

Wyjście enkoderów modelu dynamicznego wyłuskane jest z wiadomości i podane dla modelu kinematycznego, model platformy kinematycznej porusza się tak, jak odbiera to model platformy dynamicznej. Nie ma znaczenia bazowy ruch platformy kinematycznej, jednakże pokazany został tutaj dla porównania i uzyskany w taki sposób, jak dla poprzednich testów.

Przeprowadzono dwa testy, pierwszy standardowy, podobny do powyższych, drugi nadawał prędkości kół jedynie przy zmianie kierunku ruchu modelu dynamicznego. Sterowanie zostało stworzone z myślą o wywołaniu poślizgów platformy, nadana prędkość  $0,5 \frac{m}{s}$  była większa, niż w poprzednich testach.



Rysunek 6.7: Ruch modeli przy ciągłym wysyłaniu pakietów, jedynie na zmiany kierunku i kinematyka bazująca na danych zebranych z enkoderów. Wykres pierwszy i trzeci są na siebie nałożone.

### 6.2.1 Ciągłe nadawanie prędkości kół

Przy ciągłym nadawaniu prędkości kołom, enkodery odczytują praktycznie dokładnie te same wartości, jakie są nadawane. Co za tym idzie, działa to w taki sam sposób, w jaki działałaby platforma kinematyczna w poprzednim teście. Pierwsze trzy wykresy zatem nie różnią się przebiegiem, niż gdyby je wygenerować w poprzednim podłączeniu komponentów.

Przy pierwszej zmianie prędkości platformy, widać różnicę w pozycji modeli, spowodowaną poślizgiem. Duża prędkość kątowa również powoduje poślizg kątowy, przez co różnica pomiędzy pozycjami w tym samym czasie rośnie szybciej, jak w poprzednich testach. Enkodery nie są w stanie wykrywać poślizgu, dlatego ich wykres nie odstaje w momentach zmiany kierunku od wykresu modelu kinematycznego.

### 6.2.2 Impulsowe nadawanie prędkości kół

Jedno wywołanie metody maszyny symulacyjnej fizyki nadaje prędkość kołom. Następnie, z powodu symulowanych oporów, prędkość koła naturalnie spada, aż do ponownego nadania prędkości. To, jak prędkość ruchu spada, widać, gdyż segmenty przebiegu w dwóch ostatnich wykresach nie są łukami,

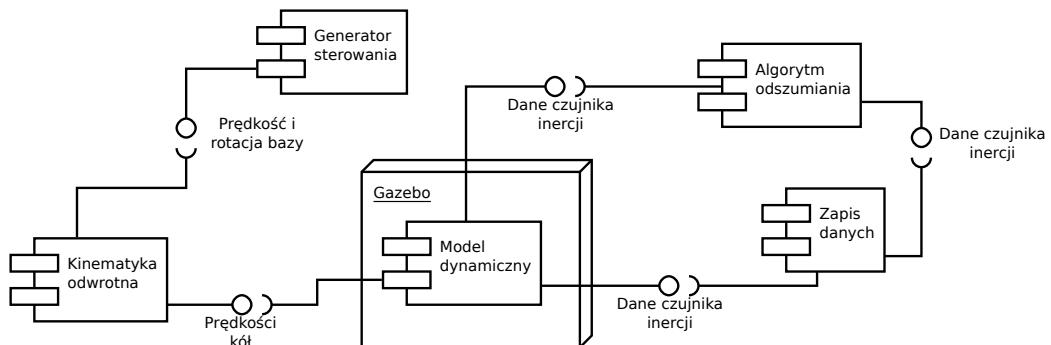
jak ma to miejsce przy ciągłym nadawaniu prędkości. Po nadaniu ostatniej wiadomości, zatrzymującą wszystkie koła, platforma nadal wolno się porusza, pod wpływem bezwładności, a brak oporu rolek w jednym z kierunków, oraz brak oporu obrotu koła pozwalał na ciągły ruch w losowym kierunku.

Co warto tutaj zauważyć, to to że enkodery prawidłowo starają się przekazać zmienną prędkość kół. Inaczej mówiąc, pozycja wynikająca z danych zwróconych z enkoderów jest dokładniejsza, niż wynikająca z wywołanego ruchu modelu kinematycznego w idealnym przypadku. Żółty wykres jest bliżej pomarańczowego, niż fioletowy/niebieski. Tutaj także widać wpływ poślizgu modelu na określanie pozycji.

Zatem model enkoderów może mieć faktyczne zastosowanie w określaniu pozycji platformy. Okazuje się lepszy, niż kinematyka, w przypadku zewnętrznego nadania prędkości platformie, na przykład pod wpływem nagłego zatrzymania. Pod warunkiem, że nie występuje poślizg. Jak już wcześniej wspomniano, platforma jest podatna na ruch w losowym kierunku przy nagłym zatrzymaniu, lecz nie zawsze taki ruch może zostać niewykryty. Poślizg, a nadanie niedeterministycznej siły zewnętrznej, to dwa różne zjawiska.

### 6.3 Czujnik inercji

Test polega na wymuszeniu określonych prędkości i przyspieszeń na modelu bazy i zebraniu wyników.



Rysunek 6.8: Połączenie komponentów dla testu czujnika inercji.

Połączenie nadające sterowanie jest podobne do poprzednich testów. Dodatkowy komponent odszumia dane i zapisuje je, aby można je było wygodnie przedstawić na wykresie. W tym teście nie jest wymagany model kinematyczny.

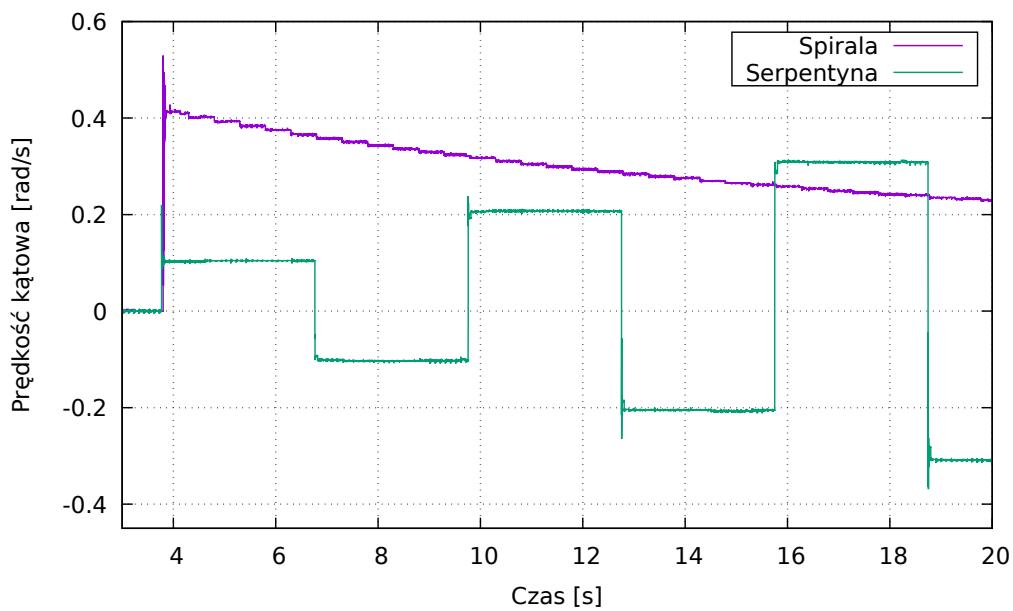
### 6.3.1 Czujnik prędkości kątowej

Ten czujnik korzysta z żyroskopu i zwraca prędkość kątową we wszystkich trzech osiach. Ponieważ jednak platforma porusza się po płaskim terenie, wymagany jest jedynie czujnik w kierunku osi Z, czyli w góre. Drugą osią może być zatem czas nadania pakietu.

Wykonano dwa testy, w pierwszym wymuszono ruch po spirali ze stałą prędkością liniową, a co za tym idzie, z nieliniowo zmieniającą się prędkością kątową platformy. Prędkość platformy aktualizowana była co 0,5 s, co widać w postaci schodków na wykresie. Trasa robota nie pokazywała żadnych nowych zjawisk, które nie zostały już wspomniane w poprzednich testach.

Drugim testem jest ruch po serpentynie. Prędkość liniowa platformy była stała, nadawano prędkości kątowe o rosnącej wartości. Model jeździł po łukach o coraz mniejszych promieniach.

Testy rozpoczęły się w tym samym czasie, lecz program nadający sterowanie celowo oczekiwany po rozpoczęciu kilka sekund.



Rysunek 6.9: Test prędkości kątowej czujnika inercji.

Pierwszy wykres pokazuje, że naturalny szum prędkości kątowej jest niewielki. Należy zatem do modelu tego czujnika celowo dodać szum, podobny do tego, jaki generuje rzeczywiste urządzenie.

Na początku generowany jest szpiculec, przy natychmiastowym nadaniu prędkości kątowej i liniowej platformie. Jest to wewnętrzny szum generowany przez maszynę symulacyjną fizyki, spowodowany dużą zmianą wielu jej

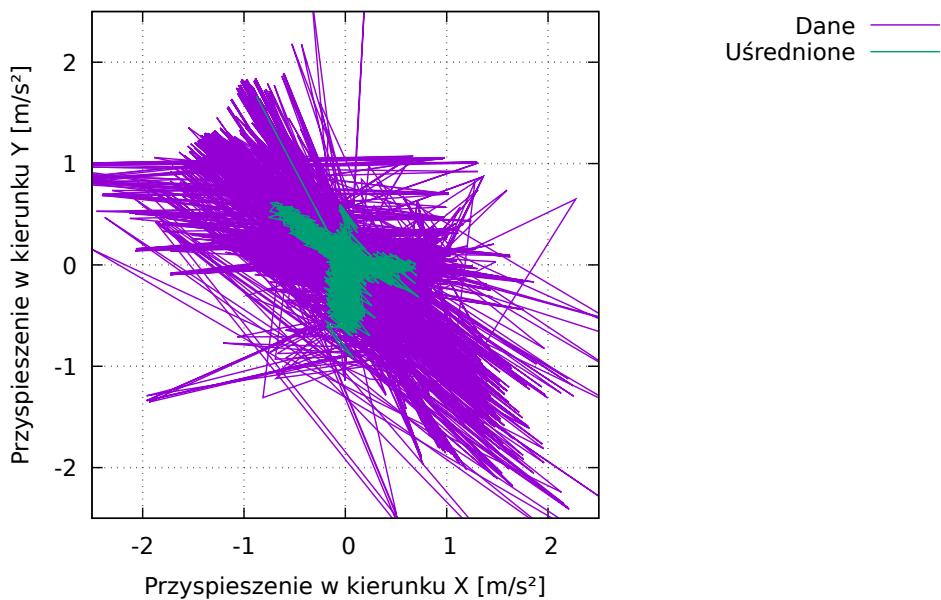
parametrów. Również wszystkie składowe elementy modelu są symulowane oddzielnie, połączone razem za pomocą więzów. Zatem informacja nadająca prędkość platformie przechodzi przez kilka warstw, zanim ustawi odpowiednie prędkości wszystkim składowym systemu.

Widać także niewielką sprężystość modelu, objawia się ona chwilowym zmniejszeniem wartości wielkości tuż po szpikulcu. Nie jest to celowo zasymulowana mechanika, a naturalne zjawisko reakcji na akcję. Jeden element składowy platformy działa na drugi, ale za chwilę drugi element także zaczyna działać na pierwszy. Podobne jest to w działaniu przeregulowania regulatora.

### 6.3.2 Czujnik przyspieszenia liniowego

Ten czujnik jest najtrudniejszy do zamodelowania. Jak wcześniej wspomniano, maszyna do symulacji nie posiada wewnętrznie informacji o przyspieszeniu obiektu, gdyż działa w dyskretnych przedziałach czasowych. Wartość przyspieszenia musi zostać celowo obliczona, co powodować może różne błędy.

W tym teście platformie nadano przyspieszenie  $2 \frac{m}{s^2}$  przez czas 1 s, w kierunku osi X, w wiadomościach co 0,1 s. Następnie platforma poruszała się przez 5 s z prędkością  $0,2 \frac{m}{s}$ . Potem nadano jej jednoczesne opóźnienie w kierunku osi X i przyspieszenie w kierunku Y, o tych samych wartościach, co na początku. Po kolejnych 5 s, platforma zwolniła do zera. Kształt trasy przypominał literę L z zaokrąglonym kątem. Samo przyspieszenie platformy wynika z uśrednienia wysyłanych prędkości w czasie, musi być nadawane dyskretnie, żaden z elementów symulatora nie jest w stanie pracować w ciągłej domenie czasowej.



Rysunek 6.10: Test przyspieszenia czujnika inercji.

Na czystych danych nie widać, jakoby model czujnika inercji w ogóle działał. Jest to widoczne tylko przy natychmiastowej zmianie prędkości platformy, to jest przy teście 6.1.1. Jednak takie teoretycznie nieskończone przyspieszenie nie może być w żaden naukowy sposób zinterpretowane, dlatego należy przeprowadzić testy z kontrolowanym przyspieszeniem.

Po uśrednieniu danych z 10 ostatnich pomiarów, okazuje się, że środek generowanych w tym czasie danych jednak się przesuwał. Co więcej, robił to w poprawnych kierunkach.

Na początku powstał zapis w kierunku dodatnim osi X, reprezentujący przyspieszenie platformy. Następnie program zanotował odchylenie pod kątem  $45^\circ$ , w drugiej kwadrze, co było nałożeniem się opóźnienia w osi X i przyspieszenia w osi Y. Na koniec zatrzymanie się platformy generowało zapis w ujemnym kierunku osi Y.

Jednakże wielkości zanotowanego przyspieszenia nie są poprawne, a mniejsze o ponad połowę. Na szczęście, nie są losowe, gdyż przy teście o mniejszym przyspieszeniu, ich wielkości spadały proporcjonalnie.

Wykres generowany w czasie rzeczywistym sugeruje, jakoby niektóre pakiety przekazywały zerowe przyspieszenie, co w znaczącym stopniu wpływa na ostatecznie generowane dane.

Widac także, że wyraźnie zapisuje się tendencja danych do oscylowania na prostej pod kątem  $45^\circ$ .

# Bibliografia

- [1] P. Muir, C. Neuman “Kinematic modeling for feedback control of an omni-directional wheeled mobile robot”, Proceedings, IEEE International Conference in Robotics and Automation, Vol 4. pp. 1772-1778, 1987.
- [2] Mihai Olimpiu Tătar, Cătălin Popovici, Dan Mândru, Ioan Ardelean, Alin Pleșa “Design and development of an autonomous omni-directional mobile robot with Mecanum wheels”, Conference: 2014 IEEE International Conference on Automation, Quality and Testing, Robotics (AQTR)
- [3] Matthieu LAMY “Mechanical development of an automated guided vehicle”, Master of Science Thesis MMK 2016:153 MKN 171, KTH Industrial Engineering and Management, Machine Design, SE-100 44 Sztokholm
- [4] Anton Gfrerrer “Geometry and kinematics of the Mecanum wheel”, Graz University of Technology, Institute of Geometry, Kopernikusgasse 24, 8010 Graz, Austria, Computer Aided Geometric Design 25(9):784-791
- [5] Li Xie, Christian Scheifele, Weiliang Xu, Karl A. Stol “Heavy-Duty Omni-Directional Mecanum-Wheeled Robot for Autonomous Navigation”, DOI: 10.1109/ICMECH.2015.7083984
- [6] Viktor Kálmán “On modeling and control of omnidirectional wheels”, PhD. dissertation, Budapest University of Technology and Economics, Department of Control Engineering and Information Technology
- [7] Jae-Bok Song, Kyung-Seok 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] Ioan Doroftei, Victor Grosu, Veaceslav Spinu “Omnidirectional Mobile Robot – Design and Implementation”, Bioinspiration and Robotics Walking and Climbing Robots, Maki K. Habib (Ed.), ISBN: 978-3-902613-15-8, InTech, 2007

- [9] Viktor Kálmán “Omnidirectional Wheel Simulation — a Practical Approach”, Budapest University of Technology and Economics, Department of Control Engineering and Information Technology, 2013
- [10] Strona internetowa symulatora Gazebo.  
<http://gazebosim.org>
- [11] Strona internetowa symulatora V-Rep.  
<http://coppeliarobotics.com>
- [12] Strona internetowa platformy programistycznej *Robot Operating System*.  
<http://www.ros.org>
- [13] Strona internetowa standardu SDF.  
<http://sdformat.org/spec>
- [14] Strona internetowa producenta czujników laserowych SICK.  
<https://www.sick.com/de/en/detection-and-ranging-solutions/2d-lidar-sensors/lms1xx/lms100-10000/p/p109841>
- [15] Strona internetowa producenta czujnika inercji.  
<https://www.digikey.com/product-detail/en/analog-devices-inc/ADIS16460AMLZ/ADIS16460AMLZ-ND/5957823>
- [16] Dokumentacja maszyny 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)