



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, 23 stycznia 2018

Streszczenie

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 będzie 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 receptory obsługujące odpowiednią maszynę symulacyjną, a także moduły wspomagające symulację. Do wykonania zadania użyto programowej struktury ramowej ROS i simulatora Gazebo.

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

Abstract

This paper describes creation of simulation environment for omnidirectional platform. Robot, which is powered by four Mecanum wheels, is a mobile base for larger robotic manipulator.

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

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

Spis treści

1 Wstęp	8
1.1 Cel pracy	8
1.2 Zakres pracy	8
1.3 Podział tej pracy na sekcje	10
2 Budowa bazy	11
2.1 Dookólna platforma mobilna	11
2.2 Koła szwedzkie	15
2.2.1 Ruch platformy na kołach	16
2.3 Enkodery	18
2.4 Skaner laserowy	19
2.4.1 Zasada działania	19
2.4.2 Komunikacja	20
2.4.3 Podstawowe cechy	20
2.5 Jednostka inercyjna	20
2.6 Podłączenie urządzenia	22
2.7 Sterowanie urządzeniami	22
2.7.1 Sterownik silników	22
2.7.2 Sterownik czujników	22
2.7.3 Program sterujący	23
3 Środowiska programistyczne	24
3.1 <i>Robot Operating System</i> (ROS)	24
3.2 Gazebo	25
3.3 V-Rep	27
3.4 Pozostałe narzędzia	28
4 Środowisko symulacyjne	29
4.1 Zapis agentowy	31
4.2 Model kinematyczny	32
4.2.1 Zachowanie	33
4.3 Model dynamiczny	33
4.3.1 Zachowanie	33
4.4 Model skanera laserowego	33
4.5 Model jednostki inercyjnej	34
4.6 Model kinematyki odwrotnej	34

4.7	Manualne sterowanie	34
4.8	Generator sterowania	35
4.9	Wyłuskanie struktury wiadomości	35
4.10	Podłoż o zmiennym współczynniku tarcia	35
4.11	Algorytm usuwania szumu z danych jednostki inercyjnej	35
4.12	Obserwator symulacji	36
4.13	Scena z symulacją	36
4.14	Rozdzielacz pakietów	36
4.15	Prosty program sterujący	36
4.16	Struktury pakietów wiadomości	37
4.17	Zewnętrzne pakiety ROSa	37
4.17.1	Rysownik wykresów	37
4.17.2	Wizualizer pomiarów	37
4.17.3	Zbieranie danych	38
5	Implementacja	39
5.1	Istniejące implementacje	39
5.2	Model 3D	39
5.3	Ogólne typy pakietów	40
5.3.1	Program wykonywalny w ROS	41
5.3.2	Wtyczka Gazebo	45
5.4	Mechanika macierzy przekształceń jednorodnych	45
5.5	Instalacja ROSa	47
5.5.1	Tworzenie pakietów	47
5.6	Format SDF	48
5.7	Model kinematyki	48
5.7.1	Komunikacja	49
5.7.2	Problemy implementacji	49
5.8	Model dynamiki	50
5.8.1	Wierność modelu	50
5.8.2	Model koła z przywracaną orientacją	50
5.8.3	Zmiana osi rolki	51
5.8.4	Modyfikacja kierunków i wartości wektorów tarcia	52
5.8.5	Komunikacja	55
5.8.6	Rozszerzenie modelu	55
5.8.7	Obliczenia symulatora	56
5.8.8	Różnice między czujnikiem, a modelem	56
5.8.9	Komunikacja	57
5.8.10	Model w Gazebo	58
5.8.11	Błędy	59
5.8.12	Program	61
5.8.13	Komunikacja	62
5.8.14	Tryby działania	62

6 Testy systemu	68
6.1 Porównanie modeli dynamiki i kinematyki	68
6.1.1 Ruch po kwadracie	68
6.1.2 Ruch po „rozecie”	70
6.1.3 Powtarzalność testów	70
6.2 Enkodery	72
6.2.1 Ciągłe nadawanie prędkości kół	73
6.2.2 Impulsowe nadawanie prędkości kół	74
6.3 Czujnik inercji	74
6.3.1 Czujnik prędkości kątowej	75
6.3.2 Czujnik przyspieszenia liniowego	76
7 Podsumowanie	78

Rozdział 1

Wstęp

1.1 Cel pracy

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

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

Składniki systemu należy wprowadzać po kolej, aby nowe mogły wykorzystywać poprzednie do sprawdzenia ich poprawności działania.

1. Stworzenie obiektu w symulatorze, który będzie sterowany za pomocą wzorów kinematycznych. Jest to całkowanie w modelu kinematycznym.
2. Stworzyć model dynamiczny 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.
3. 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 efektry nie będą jeszcze aktywne. Można go prosto побieżnie przetestować działając siłą na elementy i patrząc, czy reagują w spodziewany sposób.
4. Zapisanie wtyczki sterującej modelem, odczytującej odpowiednie dane z zewnątrz i wywołującej funkcje maszyny symulacyjnej, aby modyfikować ruch modelu. Na tym poziomie można dobudować zamiennik programu sterującego, jedynie do podawania prostych wartości bez odczytywania pomiarów i sterowania. Porównanie z obiektem kinematycznym pozwala sprawdzić, czy model zachowuje się poprawnie i zgodnie z przewidywaniami.
5. Zaprogramowanie wtyczki symulującej czujniki enkoderów, aby generowały dane, bazując na pozycjach i prędkościach kół wirtualnych.
6. Dodanie czujników wirtualnych, nieistniejących w rzeczywistości, to jest pozycja i rotacja.
7. Wystawienie do zmiany w czasie rzeczywistym masy, momentu bezwładności, współczynników tarcia, aby pozwolić na proste testowanie działania systemu z różnymi współczynnikami.
8. Stworzenie dodatkowych programów, pomagających w symulacji i testowaniu, jak model kinematyki odwrotnej, sterowany funkcją matematyczną, i podłoże ze zmiennym współczynnikiem tarcia.
9. Zaprogramowanie programów pomocniczych, zbierających i wyświetlających dane, interfejs graficzny do prostego sterowania robotem.
10. Dodanie modelu czujnika laserowego, powinien zbierać dane i przekazywać je dalej. Musi posiadać także określone kształty.
11. Dodanie modelu czujnika inercji.
12. Stworzenie uproszczonego programu sterującego, bazującego na danych z czujników laserowych, aby zbadać, czy system działa poprawnie na tyle, aby rozwinać go w końcowy projekt.
13. Przeprowadzenie testów, z porównaniem zachowania się rzeczywistej platformy w celu weryfikacji poprawności.

Po stworzeniu symulatora, kolejnym krokiem jest tworzenie głównego programu sterującego, którego testowanie opierać się będzie na powstałym środowisku symulacyjnym. Program jest 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.

1.3 Podział tej pracy na sekcje

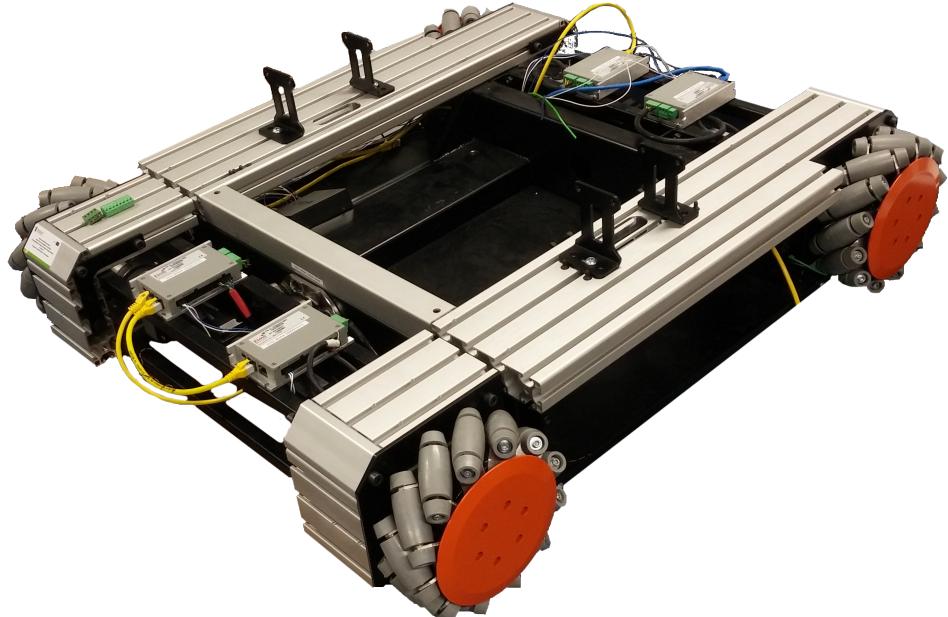
Kolejne rozdziały kolejno opisują różne aspekty pracy.

- 1** Zawiera ogólny opis pracy i sposób jej wykonania.
- 2** Techniczny opis rzeczywistej platformy i czujników, oraz mechaniki stojącej za zasadą ich działania.
- 3** Opis narzędzi programistycznych, użytych przy budowie modeli i testowaniu.
- 4** Implementacje modeli opisanych w poprzednim rozdziale, problemy i niedoskonałości z nimi związane. Opis poszczególnych dodatkowych składników systemu, używanych w simulacji, testowaniu, wizualizacji i wspomagających tworzenie.
- 5** Implementacje poszczególnych komponentów systemu.
- 6** Testy różnych składników systemu, wykresy i interpretacja.
- 7** Krótkie podsumowanie wykonanej pracy.

Rozdział 2

Budowa bazy

2.1 Dookólna platforma mobilna



Rysunek 2.1: Dookólna baza mobilna na kołach szwedzkich.

Jest to duża, prostokątna baza dookólna, poruszająca się na czterech kołach szwedzkich, rysunek 2.1. Koła są stałe, parami przytwierdzone do dwóch osi. Każde koło jest napędzane osobno przez podłączony bezpośrednio serwomotor, zatem może mieć prędkość i kierunek obrotu niezależny od pozostałych kół, kierunku poruszania się robota, oraz jego orientacji. Każdy z serwomotorów ma także wbudowany enkoder. Sterownik enkodera nadaje za pomocą sieci aktualny kąt i prędkość obrotu.

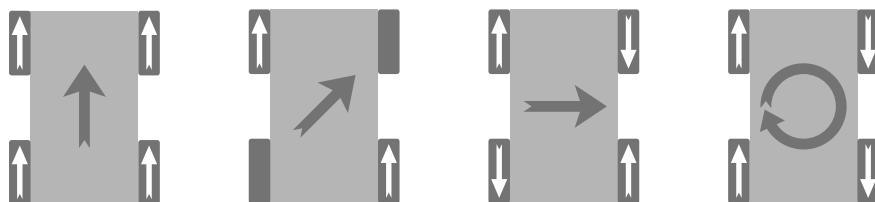
Jest to jeden z najpopularniejszych typów dookólnych platform mobilnych, mających zastosowanie także w innych robotach, jak na przykład Kuka Youbot, rysunek 2.2. Należy zwrócić uwagę na charakterystyczne ustawienie kół, identyczne jak w opisywanej platformie na rysunku 2.1.

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



Rysunek 2.2: Platforma wycofanego już ze sprzedaży robota Kuka Youbot.

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

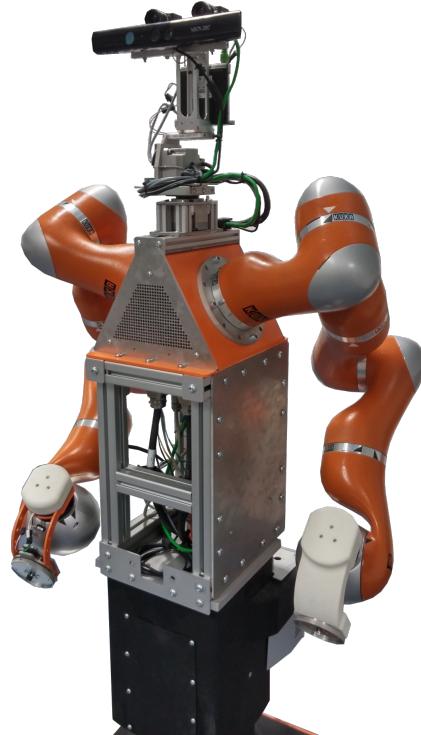


Rysunek 2.3: Podstawowe kierunki ruchu robota o napędzie wielokierunkowym.

Przykładowo, poruszając tylko przeciwnymi kołami po przekątnej, robot będzie mógł poruszać się po skosie, bez zmiany orientacji bazy. A jeśli do tego dodać obrót kół drugiej przekątnej, w odwrotnym kierunku, wtedy pojazd zacznie się poruszać w bok, pomimo faktu, że koła nie są skrętne i nie mogą ustawić się zgodnie z kierunkiem jazdy. Trasa, po której porusza się robot, przy stałej prędkości kół, zawsze jest łukiem okręgu. W szczególnym przypadku można uznać prostą za okrąg o nieskończonym promieniu, a punkt za okrąg o zerowym. Wynika to z faktu, że każdy obiekt, który ma jednostajną prędkość i stały kierunek w lokalnym układzie współrzędnych oraz stałą prędkość kątową,

będzie się poruszał po łuku. Zależność tego promienia okręgu R od prędkości liniowej v i prędkości kątowej ω , wyraża się wzorem $R = \frac{v}{\omega}$.

Baza mobilna będzie podstawą robota manipulującego Velma, tworząc razem manipulator mobilny. Velma to dwuramienny robot, wyposażony w dwa chwytki o wielu przegubach, patrz rysunek 2.4. Taka budowa wymaga szerokiej podstawy, aby zachować bezpieczną równowagę całości. Jeżdżąc na tej bazie, robot może się przemieszczać i obracać w dowolnym kierunku, aby uzyskać lepszy dostęp do manipulowanych obiektów.



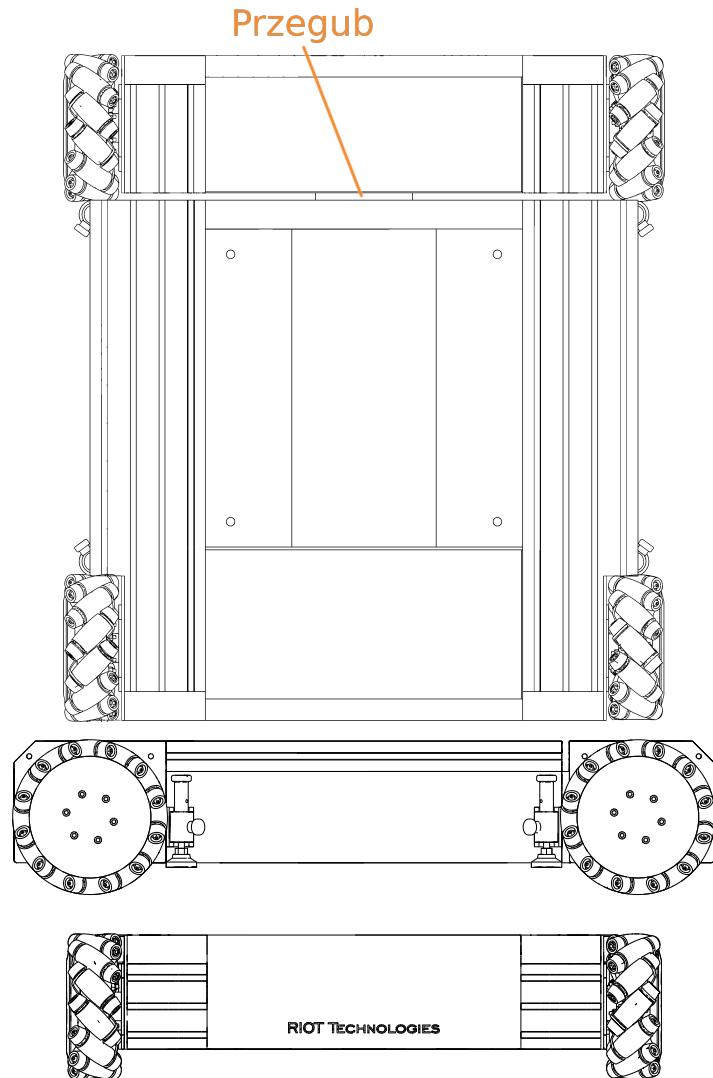
Rysunek 2.4: Robot manipulacyjny Velma.

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

Platforma ma kształt prostokąta, jest 4 cm różnicy między szerokością, a długością robota. Koła są ustawione w wierzchołkach tego prostokąta. Szerokość jest większa, co można zobaczyć porównując widok z prawej strony, z widokiem z tyłu, rysunek 2.5. Dokładne wymiary są podane na rysunku 2.6 i tabeli 2.1.

Platforma może w trakcie hamowania poruszać się innym kierunku, niż tuż przed zatrzymaniem. Jest to spowodowane tym, że konstrukcja rolek powoduje poślizg platformy

w kierunku rolki mającej aktualnie kontakt z podłożem, a ten kierunek zależy od aktualnej orientacji bazy, nie od kierunku w jakim się porusza. Należy także wziąć tutaj pod uwagę inne cechy budowy kół, jak nierówne tarcie poszczególnych rolek o powierzchnię, które może obrócić ten kierunek hamowania w nieprzewidywalny sposób [6].



Rysunek 2.5: Platforma mobilna. Widoki od góry, od prawej strony i od tyłu. Przegub obrotowy łączy dwie części.

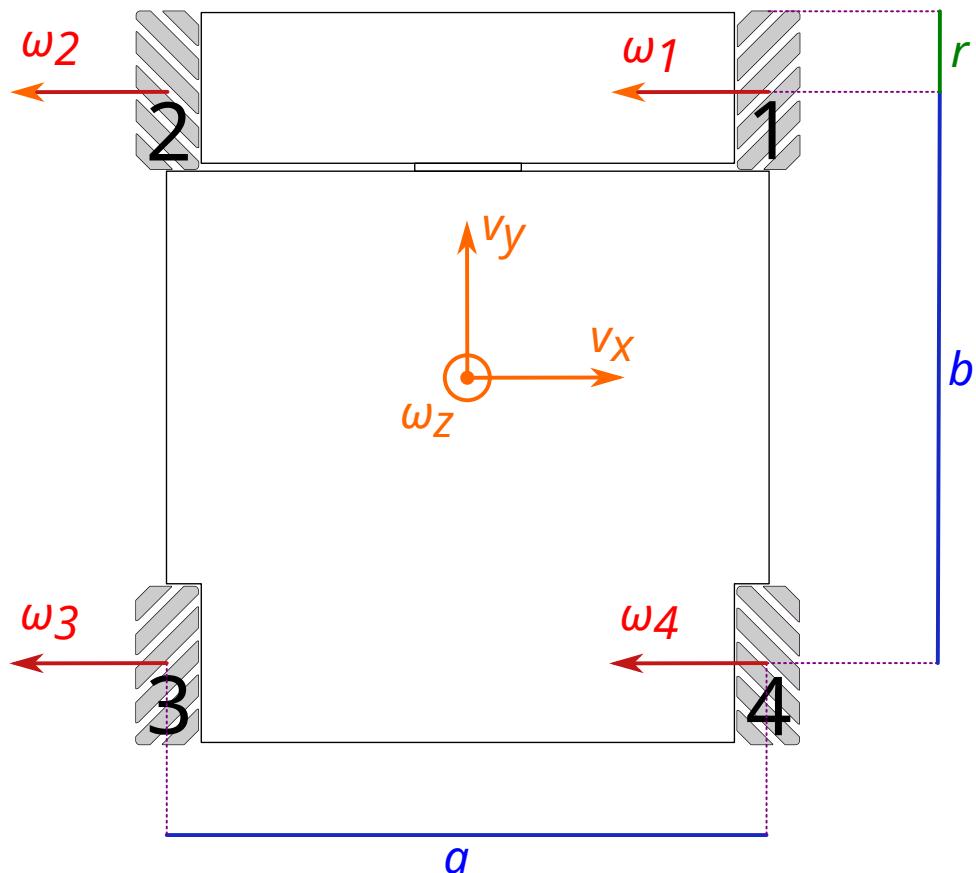
Platforma ma 3 stopnie swobody.

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

Kierunek osi X i Y jest zgodny z ustalonymi dla programów sterujących platformą.

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

Tablica 2.1: Parametry i zmienne modelu.



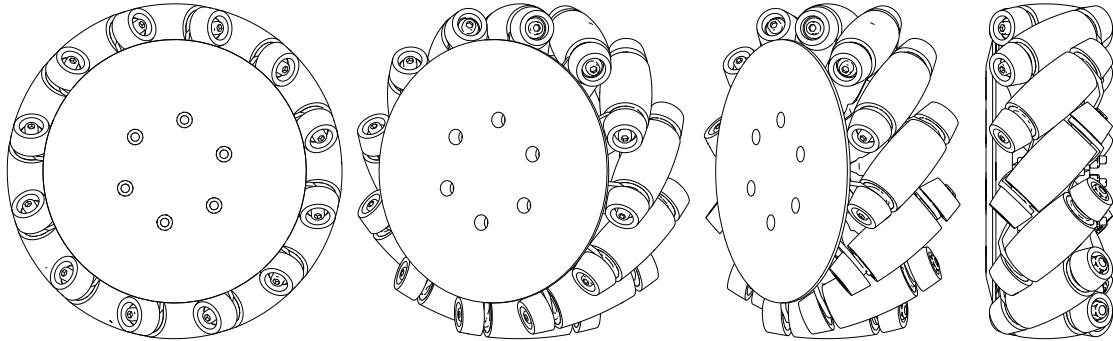
Rysunek 2.6: Wielkości używane we wzorach.

2.2 Koła szwedzkie

Koła szwedzkie, zwane także kołami Mecanum, to specjalne koła z dodatkowymi rolkami na obwodzie, ustawionymi pod kątem 45° do osi koła. Rolki są pasywne i obracają się niezależnie od siebie. Każde koło ma 12 takich rolek, patrz rysunek 2.7. W platformie ich osie ustawione są w ten sposób, że osie najwyższych, lub najniższych, rolek dwóch kół z tej samej strony robota przecinają się pod kątem prostym. Innymi słowy, robot ma identycznie ustawione koła na przeciwnieległych wierzchołkach, i razem ustawione są w

kształt litery X , patrząc na nie z góry. Należy 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. Ten drugi sposób także pozwala na ruch wielokierunkowy, ale nie jest tak często stosowany [3].



Rysunek 2.7: Widok 12 rolkowego koła szwedzkiego platformy wielokierunkowej.

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

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

Na podstawie rysunku 2.7 można zauważyć, że krzywizna rolki jest tak ustawiona, aby punkt kontaktu rolki z podłożem w czasie obrotu koła płynnie przechodził na następną rolkę. Celem jest utrzymanie równej odległości osi obrotu koła 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 jest wycinkiem paraboloidy, wzory opisujące kształt rolki są złożone. Zazwyczaj przybliża się taką rolkę wycinkiem torusa, w celu uproszczenia produkcji [4].

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

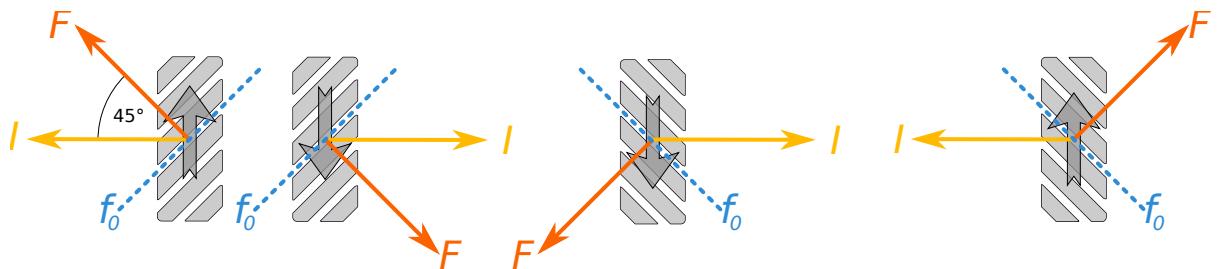
Podstawowym problemem konstrukcji koła jest nie tylko skomplikowana budowa, ale także ślizganie się rolek po powierzchni. Odległość osi obrotu koła od płaszczyzny podłożu nieznacznie zmienia się przy przenoszeniu ciężaru z rolki na rolkę, co przy dużych prędkościach powoduje drgania i jeszcze większe błędy szacowania pozycji.

2.2.1 Ruch platformy na kołach

W zwykłym kole, dzięki tarciu, moment obrotu przekształcany jest na przyspieszenie w kierunku równoległym do podłożu i płaszczyzny koła. Dodatkowo wektory tarcia są

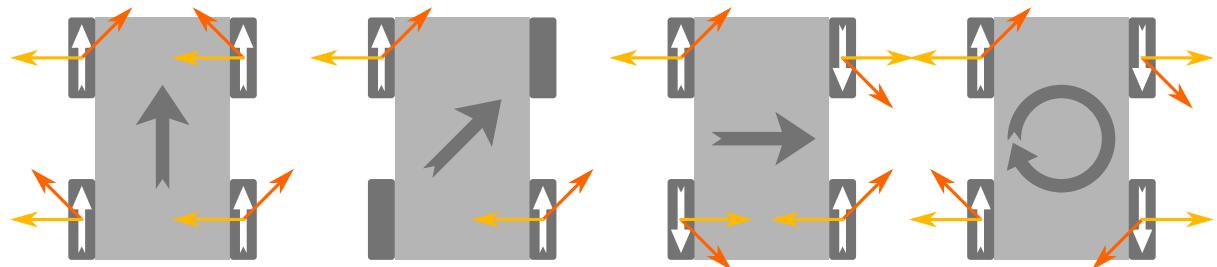
równe we wszystkich kierunkach. To znaczy, koło będzie stawało identyczny opór, niezależnie czy siła zostanie przyłożona wzdłuż osi obrotu koła, czy równolegle do płaszczyzny podłożą i płaszczyzny rolki.

Specjalne koło Mecanum wywołuje tarcie kierunku obróconym o 45° w stosunku do osi obrotu koła, zależnie od typu koła (prawoskrętne lub lewoskrętne). To oznacza, że przy nadaniu momentu siły I , wektor tarcia koła o powierzchnię T będzie obrócony w stosunku do osi obrotu koła o 45° . Z kolei tarcie nada kołu siłę F o przeciwnym zwrocie do wektora tarcia. Tarcie w kierunku f_0 , prostopadłym do F , czyli zgodnie z obrotem najniżej położonej rolki, jest w idealnym przypadku zerowe. Z kolei wypadkowa siła od wszystkich kół nadaje platformie przyspieszenie i prędkość w odpowiednim kierunku.



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

Ustawiając te koła w odpowiedni, pokazany na rysunku 2.6, sposób, można wywołać odpowiednie znoszenie się składowych sił, a w efekcie pozwolić robotowi na poruszanie się w kierunkach nieosiągalnych dla pojazdów o standardowych kołach. Warto nałożyć te wektory na wcześniejszy rysunek 2.3, aby dokładniej zobaczyć, dlaczego koła nadają platformie daną prędkość wypadkową przy odpowiednim obrocie kół.



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

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

1. Składowe wektorów sił o kierunku prostopadłym do pionowej płaszczyzny symetrii urządzenia znoszą się, ponieważ mają przeciwnie zwroty na lewej i prawej 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° do płaszczyzny symetrii. Ruch odbywa się równolegle do kierunku f_0 zatrzymanych kół, zatem nie stawiają one w tym momencie oporu.

3. Ruch podobny jest do przypadku 1. Tutaj również wektory znoszą się parami, jednak tym razem na przednich parach i tylnych. Pozostają składowe prostopadłe do płaszczyzny symetrii, które nadają platformie przyspieszenie w bok.
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.

2.3 Enkodery

Każde koło posiada wbudowany w silnik enkoder. To urządzenie wykrywa aktualny ruch koła i zwraca jego aktualną prędkość i rotację. Korzystając z modelu kinematyki, można obliczyć z tych danych wypadkową prędkość robota, a następnie, za pomocą całkowania, wyznaczyć aktualną pozycję w stosunku do punktu startowego.

W opisywanym robocie, silniki kół mają na tyle dużą moc, że wartość prędkości, wykrytej przez enkodery, bardzo niewiele odstaje od prędkości zadanej. Oznacza to, że teoretycznie, można nadawać kołom bardzo duże momenty sił, a koło rzeczywiście wykona zadaną akcję. Jednakże eksperymenty pokazały, że zasilacz robota może nie móc pracować przy takim poborze prądu i awaryjnie się odłączyć, co jest głównym powodem dla którego należy ograniczać nadawanie zbyt dużych przyspieszeń platformie.

Poślizg kół powoduje jednak, że odometria, bazująca na danych generowanych przez czujniki enkoderów, obarczona jest błędami losowymi i nie może być użyta jako jedyna metoda wyznaczania pozycji względnej w trakcie jazdy robota [5]. Zatem dodano do urządzenia także inne czujniki.

2.4 Skaner laserowy

Dodatkowym czujnikiem, używanym przy wyznaczaniu pozycji platformy, jest skaner laserowy. Platforma wyposażona jest w dwa, dwuwymiarowe czujniki typu LiDAR firmy SICK. LiDAR to połączenie wyrazów *light* i *radar*, chociaż skrót może być rozwinięty w różne słowa.



Rysunek 2.10: Skaner laserowy SICK LMS100-10000.

2.4.1 Zasada działania

Wszystkie skanery tego typu mają bardzo podobną zasadę działania. W środku urządzenia znajduje się obrotowe lusterko, zwrócone pod kątem 45° do osi obrotu. Równolegle do osi jego obrotu znajduje się laser, który emitem 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 jest to Ethernet.

Skośna szyba, będąca wycinkiem powierzchni stożka, zabezpiecza wnętrze przed zanieczyszczeniami, jej kształt niweluje ewentualne odbicia lasera, emitowanego poziomo

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 pomiaru odległości	0,012 m

Tablica 2.2: Podstawowe cechy czujnika laserowego.

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.

2.4.2 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.

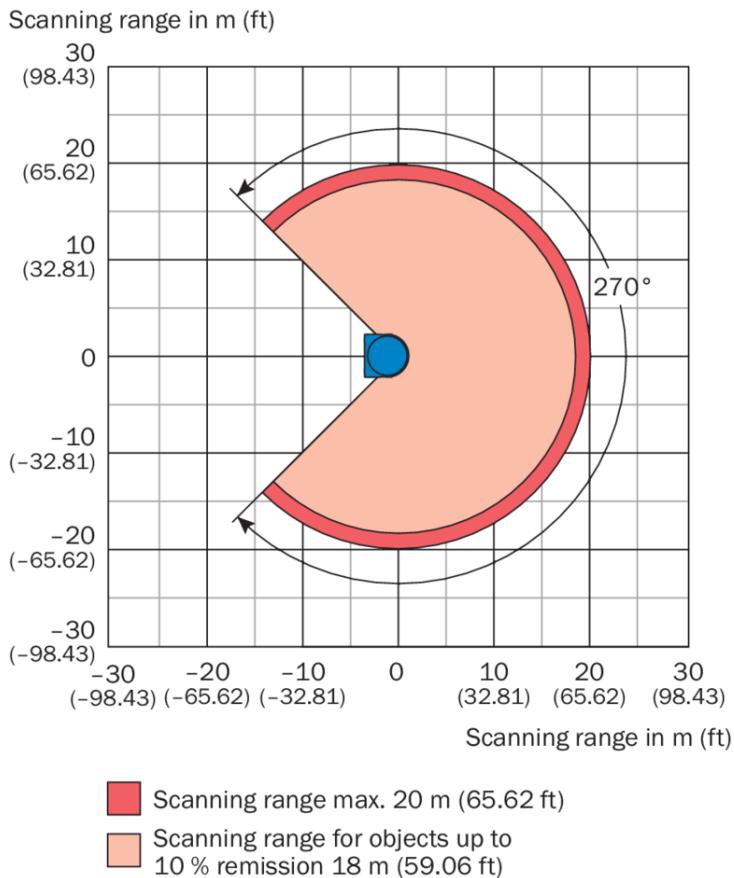
2.4.3 Podstawowe cechy

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 2.11.

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 liczba promieni wymagana jest w symulacji, aby wiernie odwzorować urządzenie.

2.5 Jednostka inercyjna

Ten czujnik to małe urządzenie, posiadające zazwyczaj zestaw wewnętrznych czujników, przydatnych przy określaniu prędkości, rotacji i przyspieszeń modułu. Dodatkowo, wiele



Rysunek 2.11: Wykres producenta dotyczący zasięgu czujnika [14].

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 sprzedawcy [15].

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

Czujnik jest wyposażony w:

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

Błędy pomiarowe akcelerometra są bardzo duże, w stosunku do błędów pomiarowych żyroskopu i skanera laserowego. Aby użyć tych danych w programie, należy zastosować na nich algorytmy usuwające szum i uśredniające wyniki. Rysując dane zebrane bezpośrednio z czujnika na wykresie, można jedynie z małą dokładnością określić kierunek przyspieszenia działającego na platformę, ale nie jego wartość.

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

2.6 Podłączenie urządzenia

Platforma podłączona jest do dedykowanego komputera, który pracuje z systemem operacyjnym Ubuntu i dodatkowymi modułami, które zapewniają pracę w czasie rzeczywistym, aby urządzenie mogło poprawnie współpracować z robotem.

Na tym systemie pracuje program, który przyjmuje komendy z innej sieci, odbierane z innego, biurowego komputera. Są tutaj także uruchomione algorytmy, obliczające pozycję platformy, bazując na odometrii.

2.7 Sterowanie urządzeniami

Efektory robota wymagają podania odpowiedniego sterowania, a czujniki odpowiedniego odbiornika.

2.7.1 Sterownik silników

Program sterujący generuje abstrakcyjne dane, na przykład liczbę zmienoprzecinkową, zapisaną binarnie. Przykładowy silnik fizyczny nie jest w stanie działać na podstawie takich danych, do pracy potrzebuje odpowiedniego napięcia na wejściu. 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.

2.7.2 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, czujnik zwraca jedynie ciąg pomiarów, ale to do tego programu należy połączenie pomiaru z informacją o emisji promienia, na ich podstawie obliczenie odległości od przedmiotu i porównanie z innymi pomiarami w celu usunięcia błędów grubych. 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.

2.7.3 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 zapotrzebowania na moc obliczeniową i niedeterministyczny 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.

Rozdział 3

Środowiska programistyczne

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. Zaawansowana 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, oprócz 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.

3.1 *Robot Operating System (ROS)*

Nazwa tego programu jest myląca. Nie jest to system operacyjny, lecz programowa struktura ramowa (*framework*), zawierająca odpowiednie biblioteki i narzędzia do tworzenia programów sterujących [12]. Są tu algorytmy wyznaczania tras, budowy map, manipulowania robotycznymi ramionami, itp.

Programy w środowisku ROS pisze się w C++ lub Pythonie i integruje z robotem za pomocą kilku gotowych struktur kolejek wiadomości. Ta struktura ramowa zawiera także pakiety do wizualizacji odbieranych danych w formie graficznej.

Działanie systemu jest oparte o pakiety. Każdy taki pakiet jest katalogiem zawierającym w sobie pliki opisujące jego parametry i skrypty CMake, używane do komplikacji. Pakiet może zawierać programy wykonywalne, dane, definicje, lub inne dowolne pliki. W symulacji opisywanej platformy, modele są pakietami, zawierającymi biblioteki ładowane dynamicznie, uruchamiane przez jeszcze inny pakiet symulatora. Pakiety mogą być zależne od siebie, osobno w kwestii komplikacji, jak i uruchomienia.

ROS potrzebuje także działającego demona w tle. Odpowiada on za komunikację i kontroluje stany wszystkich węzłów. Z punktu widzenia konstrukcji systemu, można

porównać go do jądra systemu operacyjnego, a węzły do działających procesów. Dlatego też nazwa *Robot Operating System* nie jest przypadkowa.

Na stronie internetowej ROSa, znajduje się bogata biblioteka pakietów, stworzonych przez inne osoby. Każdy może także umieścić tam swój własny pakiet, aby inni mogli go ściągnąć i wykorzystać w swoich projektach.

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

Używane wbudowane narzędzia z tej struktury ramowej to:

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

catkin System budujący pakiety, działający na skryptach CMake.

roslaunch Program do wykonywania skryptu uruchamiającego pakiet.

rosrun Program do uruchamiania pliku wykonywalnego z pakietu.

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

roscore Demon ROS, zarządzający wszystkimi węzłami.

Dodatkowo, używane funkcjonalności funkcji bibliotecznych:

- Rejestracja nowego węzła w demonie ROS.
- Stworzenie nadajnika strumienia wiadomości.
- Stworzenie odbiornika strumienia wiadomości.
- Powiadomienia.
- Nadawanie macierzy przekształceń jednorodnych.
- Zawieszenie programu.

3.2 Gazebo

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

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

Symulator oprócz tego ma wbudowany edytor modeli, w którym można składać i ustawiać odpowiednie obiekty razem w przestrzeni trójwymiarowej i generować plik opisujący symulację. 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 w pliku tekstowym. Również tworząc modele poza edytorem, posiada się nad nimi większą kontrolę, a parametry składowych da się ustawić z dowolną dokładnością.

Gazebo przyjmuje modele w specjalnym formacie SDF. Jest to standaryzowany, zdefiniowany niezależnie od symulatora format, do opisywania budowy robotów i czujników. Dzięki temu plik SDF może być użyty w innej symulacji, w innym programie, pod warunkiem przestrzegania standardu. Składnia jest zgodna ze standardowym językiem XML, co znaczy, że może być tworzona na dowolnym edytorze tekstowym.

Wtyczka do sterowania modelem jest skompilowaną biblioteką, dołączaną na starcie programu. Tworzy się ją w C++ lub Pythonie, jako klasę dziedziczącą po abstrakcyjnej klasie dostarczonej przez Gazebo. Dzięki temu może korzystać ze wszystkich funkcji 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 symulatora Gazebo, jednak jest niezależny od podobnej mechaniki ROSa. Z punktu widzenia ROSa, programy uruchamiane w Gazebo są jednym węzłem, który posiada wiele strumieni komunikacyjnych, zarówno dostarczonych przez sam symulator, jak i wczytanych wtyczek.

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

Gazebo może także być składnikiem systemu ROS, kod źródłowy jest dzielony w ramach wspólnej organizacji. 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, nie zawsze najnowsza wersja symulatora jest dostarczana razem z najnowszą wersją programowej struktury ramowej.

Gazebo implementuje prawie wszystkie elementy standardu SDF, ale tylko niektóre będą używane. Posiada także kilka narzędzi do wizualizacji wygenerowanych danych, ale i ROS je posiada.

- Symulacja fizyki za pomocą maszyny do symulacji ODE.

- Całkowanie prędkości poprzez umieszczenie obiektu kinematycznego w przestrzeni wirtualnej i nadawanie mu prędkości.
- Możliwość modyfikacji wektorów tarcia.
- Dane o prędkościach i pozycjach wszystkich obiektów na scenie.
- Symulacja skanera laserowego.
- Symulacja jednostki inercyjnej.
- Wizualizacja obiektów za pomocą siatki trójkątów i kolorów.
- Wizualizacja kolizji, kształtów i inercji obiektów.
- Wizualizacja pozycji i rotacji poszczególnych obiektów, ich lokalne układy współrzędnych.

3.3 V-Rep

V-Rep [11], to duże i złożone środowisko, reklamujące się wieloma zaawansowanymi mechanizmami i funkcjami. Pomimo otwartego kodu, użycie komercyjne jest płatne. Dla zastosowań akademickich program jest bezpłatny. Bogaty interfejs graficzny zakłada budowę i symulację wszystkiego w tym jednym programie.

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

Głównym mankamentem programu jest 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 języku Lua. Poprzez komunikację sieciową, są też też dostępne inne języki, jak C, Matlab, Java, itp. Komunikacja z innymi programami odbywa się poprzez specjalne wtyczki do środowiska. API pozwala stworzyć mały, wbudowany interfejs graficzny do sterowania symulacją poprzez przyciski i suwaki.

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

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

3.4 Pozostałe narzędzia

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

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

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

KDevelop jest graficznym edytorem tekstowym i nadaje się do pisania kompilowanego kodu wtyczek. Można podłączyć 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 testów w kontrolowany i prosty sposób. Uniwersalne narzędzie pomagające w wielu miejscach.

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

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

Dia to graficzny edytor do tworzenia diagramów UML.

Rozdział 4

Środowisko symulacyjne

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

Aby uruchomić symulację, nie wystarczy uruchomienie symulatora z modelami, należy zadbać także o odpowiednie przekazywanie informacji do i z symulowanych obiektów. Wskazane jest przetestować modele, czy zachowują się poprawnie w prostych scenariach 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 pakietów od siebie, można ich także użyć przy komunikacji z rzeczywistym robotem.

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

Niektóre typy wiadomości posiadają wbudowany nagłówek, inne istnieją w dwóch wersjach, z nagłówkiem i bez. Dopisek **Stamped** określa istnienie tej dodatkowej informacji. Nagłówek ma trzy pola:

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

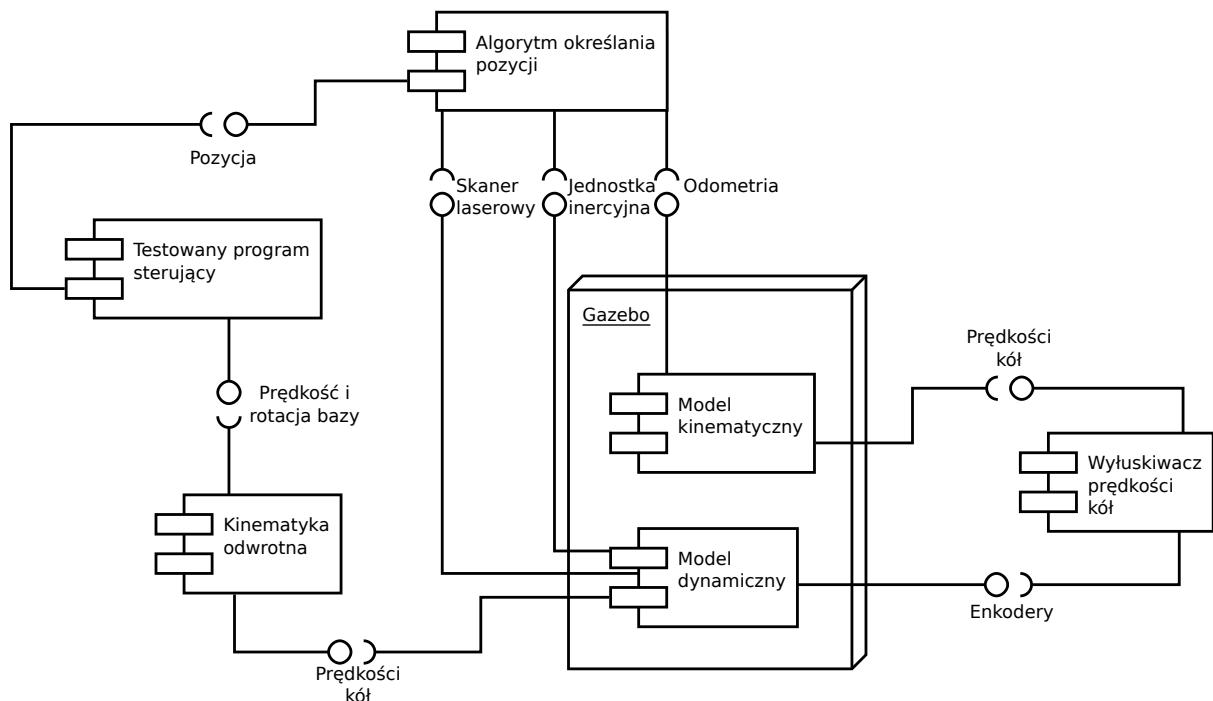
Pakiety można podzielić na trzy typy:

- Generujące dane.
- Przekazujące i modyfikujące dane.
- Zbierające dane.

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

Tablica 4.1: Typy wiadomości przekazywanych pomiędzy pakietami.

Poniżej, każdy pakiet opisany jest bardziej szczegółowo, wraz z jego interfejsem.
W trakcie testowania symulatora, podłączenie pakietów będzie wyglądać następująco:



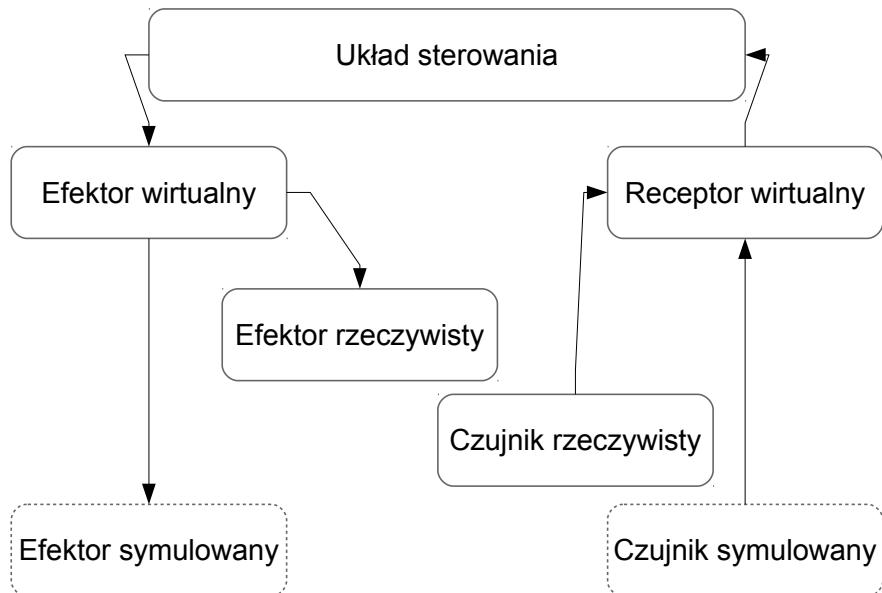
Rysunek 4.1: Komunikacja podstawowych pakietów systemu w trakcie testowania programu sterującego.

Ważną rolę odgrywa tutaj algorytm określania pozycji, bazujący na odometrii, jednostce inercyjnej i danych ze skanera laserowego. Odometria jest generowana za pomocą modelu kinematycznego, sterowanego danymi z enkoderów 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 ze skanerów laserowych w celu wykrycia przeszkody, nie tylko w celu określenia pozycji. Bardziej zaawansowany program sterujący mógłby generować zadane prędkości kół bezpośrednio, nie bazować na modelu kinematyki odwrotnej.

4.1 Zapis agentowy

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



Rysunek 4.2: Struktura agenta upostaciowanego.

Można to przedstawić za pomocą zapisu agentowego, rysunek 4.2. Agent upostaciowany 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 rysunku 2.4), w którym to zachodzi odczytanie obrazu z kilku kamer. Następnie obraz przesyłany jest do komputera, w którym sterowniki interpretują dane, usuwając błędy, tworzą mapę głębokości, wykrywają szkielety i sylwetki osób. Te dane mogą być wykorzystane łatwo w grach i programach sterujących.

Modelowanie receptora, tak jak w przypadku efektora, polega na wygenerowaniu odpowiednich danych, używając odpowiednich funkcji w przestrzeni wirtualnej. Mogą one polegać na emitowaniu półprostych, symulujących laser, lub wręcz renderowaniu obiektów, aby uzyskać obraz z wirtualnej kamery. Receptor symulowany ma pełną wiedzę o symulowanym świecie, dokładne 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.

4.2 Model kinematyczny

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

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

Równania modelu kinematyki najwygodniej przedstawić w postaci macierzowej, podobnej do tego, jak opisano w [2]. Dokładna podstać wzoru zależy od kolejności numerowania kół i interpretacji wymiarów. Dla opisanego tutaj przypadku, (stałe zdefiniowane są w tabeli 2.1, numeracja kół jest pokazane na rysunku 2.6):

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

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

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

4.2.1 Zachowanie

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

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

4.3 Model dynamiczny

Maszyna do symulacji dynamiki używa informacji o kształtach, masach i złączach pomiędzy ogniwami robota. Należy zatem stworzyć obiekt, złożony z modeli ogniw i samych ogniw i umieścić w symulatorze. Potem należy nadać obiektom odpowiednie siły, abytrzymać wyniki przybliżone do tego, jak zachowywałaby się rzeczywista baza mobilna.

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

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

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

4.3.1 Zachowanie

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

4.4 Model skanera laserowego

Symulator generuje odpowiednie dane, emitując promienie w przestrzeni wirtualnej. Następnie maszyna symulacyjna fizyki oblicza kolizje promieni z obiektami na scenie. Jest to operacja bardzo kosztowna obliczeniowo.

Do położień punków dodawany szum o rozkładzie normalnym, aby symulować błędy pomiarowe, powstałe przy odczycie odbicia lasera.

Odczyt z rzeczywistego skanera pokazuje, że sztucznie wygenerowane dane są podobne do danych zebranych przez czujnik. Dodatkowo, wbudowany w czujnik sterownik usuwa błędy grube z pomiarów, zatem rzeczywiste dane nie posiadają ich, więc i nie jest konieczne dodawanie ich do modelu.

4.5 Model jednostki inercyjnej

Rzeczywisty czujnik obarczony jest bardzo dużymi błędami pomiarowymi. Jednakże, jego model także zwraca bardzo zaszumiony obraz.

4.6 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). Całkowanie prędkości kół i tak nie ma większego sensu, ponieważ pozwoliłaby jedynie obliczyć aktualną ich pozycję. Z wyjątkiem porównania tych danych z danymi z enkoderów, nie ma to zastosowania.

Ten pakiet przyjmuje zadaną prędkość, kierunek i obrót platformy, a zwraca prędkości kół, które powinny być nadane platformie, aby wywołać taki ruch. Wzory mogą być przedstawione w postaci macierzowej.

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

Stałe, użyte we wzorze zdefiniowane są w tabeli 2.1, a numerowanie kół na rysunku 2.6. 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 pakiety i różne modele przyjmują różną pozycję wyjściową robota. Czasami przód modelu skierowany jest w dodatnią stronę osi X, a czasami Y. W związku z tym, ta funkcjonalność jest w stanie przekonwertować dane wejściowe dla innego robota tak, aby mogły być użyte do sterowania opisywanym tutaj robotem.

4.7 Manualne sterowanie

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

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

4.8 Generator sterowania

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

Ten pakiet generuje powtarzalne sterowanie, bazując na wczytanym pliku tekstowym. Zwraca zadane prędkości liniowe i prędkość kątową bazy.

Podłączono go do rzeczywistego robota, aby poruszał platformą po kwadracie i obracał wokół osi. Eksperyment przebiegł pomyślnie.

4.9 Wyłuskanie struktury wiadomości

Każda wiadomość przekazywana pomiędzy węzłami jest zwykle zagnieżdzoną strukturą.

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

Takie zjawisko występuje przy przekazywaniu informacji o pozycji i prędkości kół, generowanej przez model czujnika enkoderów, do programu manualnego sterowania, lub przekazywanie danych z enkoderów do modelu kinematycznego.

ROS nie pozwala na automatyczne odbieranie tylko części pakietu, dlatego powstał ten program.

4.10 Podłoże o zmiennym współczynniku tarcia

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

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

4.11 Algorytm usuwania szumu z danych jednostki inercyjnej

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

Ten program uśrednia dane w prosty sposób, licząc średnią z określonej ilości poprzednich pomiarów. 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. Działa to bardzo dobrze przy uśrednianiu szumu przy zerowym przyspieszeniu, w trakcie ruchu robota z prędkością jednostajną.

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

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

4.12 Obserwator symulacji

Model uruchamiany w symulatorze. Oblicza i zwraca statystyki międzymodelowe w przestrzeni symulacji, takie jak odległość i kąt. Pozwala zbadać, jak model dynamiczny zachowuje się w stosunku do modelu kinematycznego, to znaczy, czy pozycja, obliczone przez maszynę symulacyjną fizyki jest zbliżona do pozycji obliczonej równaniami kinematycznymi po całkowaniu.

4.13 Scena 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.

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

4.14 Rozdzielacz pakietów

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

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

4.15 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 dane o zadanej prędkości. W zależności od danych z czujników laserowych, program zmienia swój stan i obraca kierunek obrotu o 90° . Ten

sterownik dla uproszczenia nie generuje polecień obrotu kątowego, sterowany obiekt powinien zachować swoją orientację.

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

4.16 Struktury pakietów wiadomości

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

Dodatkowo zdefiniowane typy wiadomości to:

- Dane prędkości i rotacji kół, zwracane przez model enkoderów.
- Dane o względnej pozycji i rotacji obiektów na scenie.
- Zadane prędkości kół.
- Asynchroniczne wywołanie do ustawienia inercji ogniw modelu.
- Asynchroniczne wywołanie do ustawienia współczynników tarcia.

4.17 Zewnętrzne pakiety ROSa

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

4.17.1 Rysownik wykresów

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

4.17.2 Wizualizer pomiarów

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

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

4.17.3 Zbieranie danych

Każdy strumień wiadomości może zostać zapisany do pliku, a następnie odtworzony w ten sam sposób, w jaki został odebrany. Wbudowane w ROSa narzędzie `rosbag` pozwala „nagrać” i odtworzyć dane. Zapisuje to w formie pliku binarnego, wraz z dokładnymi parametrami działania nadajnika wiadomości, takimi jak nazwy ścieżek, ilość odebranych wiadomości, czas.

Korzystając z pomocniczego narzędzia do zarządzania węzłami, `rostopic`, możliwe jest również wydrukowanie danych do pliku tekstowego w formacie CSV (*Comma Separated Values*), aby mogły być następnie wykorzystane w dowolny sposób przez inne programy jak Gnuplot, Calc (Exel), czy Matlab.

Rozdział 5

Implementacja

W tym rozdziale opisane są szczegóły techniczne zastosowanych rozwiązań.

5.1 Istniejące implementacje

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

Kuka Youbot jest popularnym robotem wielokierunkowym. Jego modele są domyślnie dostępne w różnych symulatorach, między innymi w Gazebo i V-Repie, które są dobrymi kandydatami do użycia w projekcie. Tylko w przypadku V-Rep, istnieje wstępny sterownik do którego da się wysyłać odpowiednie wartości zadanej prędkości liniowej i kątowej, a on nadaje odpowiednie prędkości kołom, aby odpowiednio poruszać modelem. 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.

5.2 Model 3D

Model 3D bazy mobilnej, opisany równaniami matematycznymi, powinien mieć zachowanie zbliżone do oryginału, najbardziej jak to tylko możliwe. Musi uwzględnić 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 więzów o jednym stopniu swobody, 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 odpowiedniej wizualizacji symulacji, można wykorzystać istniejący model CAD do stworzenia siatki trójwymiarowej i nadania symulowanemu obiekowi wyglądu zbliżonego do fizycznego robota. Nie ma to znaczenia dla przebiegu symulacji, gdyż ta część nie bierze udziału w obliczeniach maszyny symulującej fizykę.

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, sterownik 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.

5.3 Ogólne typy pakietów

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

5.3.1 Program wykonywalny w ROS

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

- Inicjalizacja węzła.
- Stworzenie nadajnika strumienia wiadomości.
- Stworzenie odbiornika strumienia wiadomości.
- Zawieszenie procesu.
- Generowanie logów, jak informacje i ostrzeżenia.

Inicjalizacja

Na początku głównej funkcji programu `int main(int argc, char** argv)`, należy podać argumenty wywołania programu do funkcji inicjalizującej ROSa. Ta funkcja odczytuje argumenty dotyczące tej programowej struktury ramowej i modyfikuje zmienne w razie problemów, usuwając niektóre argumenty, aby reszta programu parsowała poprawne dane.

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

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

Następnie program parsuje argumenty wywołania, w zależności od tego, jakie zadanie ma wykonywać.

Stworzenie nadajnika

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

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

Korzystając z obiektu węzła, można zarejestrować, za pomocą powyższej funkcji, nowe nadajniki strumienia wiadomości. Jest to metoda szablonowa, to oznacza, że przy wywołaniu należy podać typ wiadomości, jaką węzeł powinien nadawać. Następne argumenty to nazwa strumienia, wielkość bufora wysłanych wiadomości, oraz opcjonalny argument, decydujący o tym czy ostatnia nadana wiadomość powinna być buforowana i wysłana natychmiast na połączenie się nowego odbiorcy do tego strumienia. Metoda zwraca obiekt nadajnika który to może być użyty do nadawania wiadomości.

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

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

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

Stworzenie odbiornika

W nieco innym sposobie należy stworzyć odbiornik strumienia wiadomości. Po każdej odebranej wiadomości, wywoływana jest podana w argumencie funkcja, która przyjmuje i parsuje wiadomość. Można to porównać do działania przerwań systemowych.

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

Podana metoda, podobnie, jak w nadajniku, jest metodą obiektu węzła. Przyjmuje nazwę strumienia wiadomości, wielkość bufora, wskaźnik na funkcję obsługi oraz opcjonalnie informacje dotyczące połączenia. Te informacje to takie jak typ używanego protokołu (datagramowy, czy połączeniowy), wielkość pakietu sieciowego, itp.

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

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

Wypisywanie logów

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

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

Rysunek 5.1: Makra ROSa, wypisujące powiadomienia.

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

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

Zawieszenie procesu

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

Przykład

Przykładowe użycie powyższych metod przy implementacji prostego filtra wiadomości. Ten kod tworzy nadajnik i odbiornik wiadomości, zawierającej prędkość liniową i kątową, a następnie na odbiór każdej wiadomości, przekazuje dalej jedynie prędkość liniową w płaszczyźnie platformy i prędkość kątową wokół osi Z, skierowanej w górę. Dzięki temu robot nie otrzyma sterowania w niemożliwym do poruszania się kierunku.

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

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

```

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

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

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

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

    //obiekt węzła
    ros::NodeHandle handle;

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

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

    //zawieszenie wykonywania głównego wątku
    ros::spin();
    return 0;
}

```

5.3.2 Wtyczka Gazebo

Wtyczka do symulatora Gazebo jest klasą, dziedziczącą po klasie `ModelPlugin`, lub w przypadku modelu czujnika po `SensorPlugin`. Jest komplikowana do postaci biblioteki i ładowana dynamicznie przy uruchomieniu symulatora. Gazebo, prócz wywołania ewentualnego konstruktora, wywołuje metodę wirtualną

```
void Load(physics::ModelPtr parent, sdf::ElementPtr sdf)
```

która to posiada argument typu sprytny wskaźnik na obiekt modelu, który obsługuje ta wtyczkę, oraz wskaźnik na element pliku SDF, opisujący go.

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

Wywołanie na kroki symulacji

Główną mechaniką używaną u wtyczek jest periodyczne wywołanie kodu co każdy krok symulacji. Pozwala to na przykład nadawać aktualną pozycję modelu. Gazebo obsługuje kilka zdarzeń, wywoływanych na różne części symulacji. Ten, do którego należy się podłączyć to `WorldUpdateBegin`, połączenie się do tego zdarzenia wywołuje następującą metodę:

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

Oznacza to tyle, że do podłączenia można użyć funkторów z biblioteki `boost`, lub lepiej, ze standardu C++. Przykładowe podłączenie do własnej zdefiniowanej metody `OnUpdate` może wyglądać w ten sposób:

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

Należy zatrzymać zwrócony sprytny wskaźnik do połączenia.

5.4 Mechanika macierzy przekształceń jednorodnych

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

Macierz 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 wiadomości i identyfikatorem, oraz informacją względem jakiej pozycji podane są poniższe dane.

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 5.1: Nazwy identyfikatorów ramek, używanych w symulatorze.

Nazwa	Punkt wzgledny	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 5.2: Ramki wysyłane do demona ROS.

- Nazwa nowej pozycji, jaka powstanie po zastosowaniu podanej transformacji do określonej w nagłówku pozycji.
- Lokalne położenie.
- Lokalna orientacja.

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

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

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

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

5.5 Instalacja ROSa

Instalacja programu na systemie operacyjnym jest złożona. Z wyjątkiem odpowiednich wersji Ubuntu, nie ma łatwego sposobu na instalację go na innych systemach. Na przeszkozie 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, niezbędną takim programom w dużych ilościach.

5.5.1 Tworzenie pakietów

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

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

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

W zewnętrznym katalogu uruchamia się skrypt komplikacyjny, który kolejno sprawdza zawartość katalogów i wywołuje ich skrypty. Sprawdza również wszystkie nazwy i wersje. Tworzy dwa osobne katalogi, jeden z wygenerowanymi definicjami, drugi z plikami powstałymi w trakcie komplikacji. Dba o odpowiednie podawanie ścieżek do programów, kolejność komplikacji pakietów i załączanie nazw. Na przykład, jeśli pakiet wymaga pliku nagłówkowego, generowanego przez komplikację innego pakietu, plik ten może być załączony w kodzie tak, jakby był systemowy. CMake zadba o wywołanie kompilatora z odpowiednimi argumentami, aby odszukał wszystkie potrzebne pliki. Ważne tutaj jest zadbanie o odpowiednią kolejność komplikacji, aby nie próbować komplikować pliku, dla którego nagłówki nie zostały jeszcze wygenerowane.

5.6 Format SDF

Simulation Description Format (SDF) [13] 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 w którym poruszają się roboty, na przykład rozmieszczenie elementów na symulowanej scenie, określania wyglądu i fizycznego zachowania się materiałów itp.

W przeciwieństwie 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 nadrzędny, wszystkie składowe są logicznie rozmieszczone równolegle jako jego dzieci. Specjalnie opisane więzy definiują interakcje pomiędzy ogniwami. Jako model, standard rozumie nie tylko roboty, ale także obiekty typu przeszkody, źródła światła, elementy animowane i tym podobne.

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

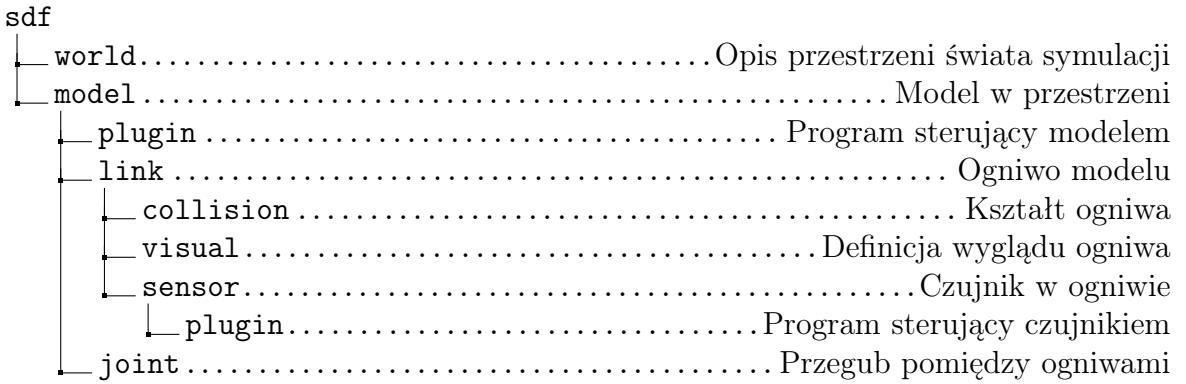
W każdym z modeli zawiera się nazwa, domyślna pozycja, sposób traktowania przez symulator i wtyczki programów obsługujących zaawansowane zachowanie modelu, opisane w równoległych do składowych elementach, ale jako że wszystkie te mogą wystąpić tylko raz, należy traktować je jako część elementu `model`. 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 je 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.

5.7 Model kinematyki

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



Rysunek 5.2: Najważniejsze elementy formatu SDF.

5.7.1 Komunikacja

Komunikacja programu sterującego platformą odbywa się przez kanały komunikacyjne ROSa.

Wiadomość zawierająca dane prędkości kół czterokołowego robota nie mieści się w standardzie, zatem został stworzony specjalny typ `omnivelman_msgs/Vels`. Ta struktura zawiera cztery wartości zmienoprzecinkowe podwójnej precyzji, oznaczające prędkości w $\frac{rad}{s}$.

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:

- `omnivelman_msgs/Vels` z zadanymi prędkościami kół.

5.7.2 Problemy implementacji

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

Oznacza to, że wszystkie elementy typu `link`, będąc dziećmi `model`, nie zachowują swojej pozycji w lokalnym układzie współrzędnych. Powoduje to, że nadając prędkość kątową modelowi, nadajemy ją każdemu ogniwu osobno. Każde z kół i dwie części bazy, obracają się zgodnie z zadanymi wartościami, wokół osi Z, 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 ogniska modelu są w jakikolwiek sposób podłączone do jego głównej części, to wprowadzałoby także nieścisłości w typie przegubów, niektóre elementy powinny być ruchome.

Aby przeciwdziałać temu zjawisku, należy przenieść zawartość elementów `link` do elementu `model` i ustawić je jako `visual` elementu `model`. To znaczy, ustawić je nie jako dane używane przez maszynę symulacyjną fizyki, ale jako dane służące do rysowania obiektów na ekranie. W ten sposób traktowane są jako jedna całość, a nie osobne składowe. Nie można użyć tutaj więzów statycznych, gdyż te są wykorzystywane przez maszynę symulacyjną fizyki i ignorowane przy kinematycznym ruchu.

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

5.8 Model dynamiki

Jest wiele sposobów na modelowanie tego obiektu.

5.8.1 Wierność modelu

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

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

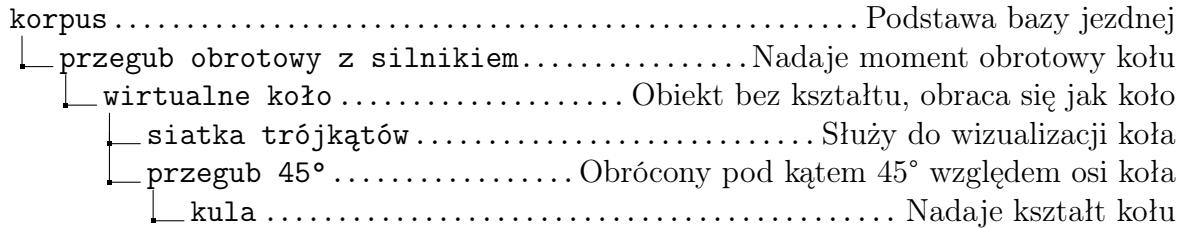
Przybliżenie rolki siatką trójkątów jedynie zmniejsza powyższy efekt, gdyż sama siatka zbudowana jest z prostych krawędzi. Zwiększając jej gęstość można teoretycznie poprawić jakość symulacji, kosztem olbrzymiego skoku ilości obliczeń, każde obarczone błędami numerycznymi. Obliczenia związane z wykryciem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

5.8.2 Model koła z przywracaną orientacją

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

Polega on na tym, iż koło, podłączone do korpusu za pomocą przegubu obrotowego, posiada drugi przegub obrotowy, obrócony pod kątem 45° w stosunku do osi koła (i pierwszego przegubu), tak aby był równolegle do osi obrotu dolnej rolki, mającej kontakt z podłożem. Do tego przegubu jest podłączony obiekt kuli oddziałującej z podłożem, który to w każdej iteracji symulacji należy zresetować do orientacji wyjściowej, razem z

orientacją drugiego przegubu. Zastosowano kulę, gdyż jest prymitywem geometrycznym i obliczenia jej kolizji są najmniej wymagające obliczeniowo dla maszyny symulacyjnej.



Rysunek 5.3: Zagnieżdżenie obiektów koła z resetowaną symulacją rolki.

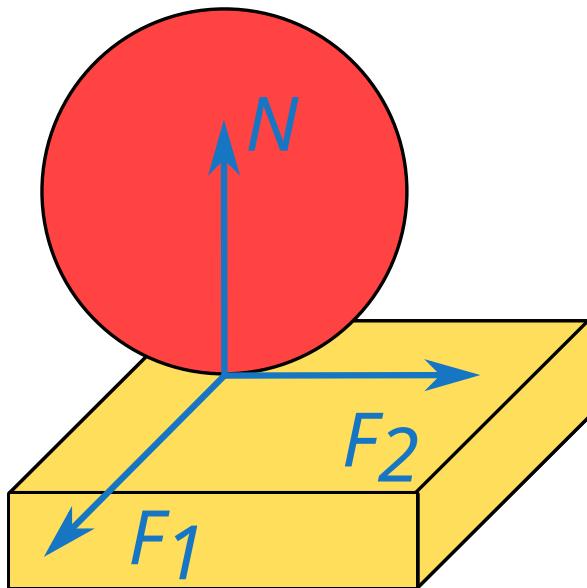
Wywołuje to takie działanie, jak gdyby koło w danej chwili mogło jednocześnie obracać się w dwóch kierunkach, wokół osi obrotu dolnej rolki, oraz wokół osi koła. Przez następny krok symulacji, model zachowuje się poprawnie, aż pod wpływem obrotu koła, druga oś przestaje być równoległa do osi dolnej rolki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

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

Bardzo skomplikowany sposób działania kół skłania do szukania innych rozwiązań.

5.8.3 Modyfikacja kierunków i wartości wektorów tarcia

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

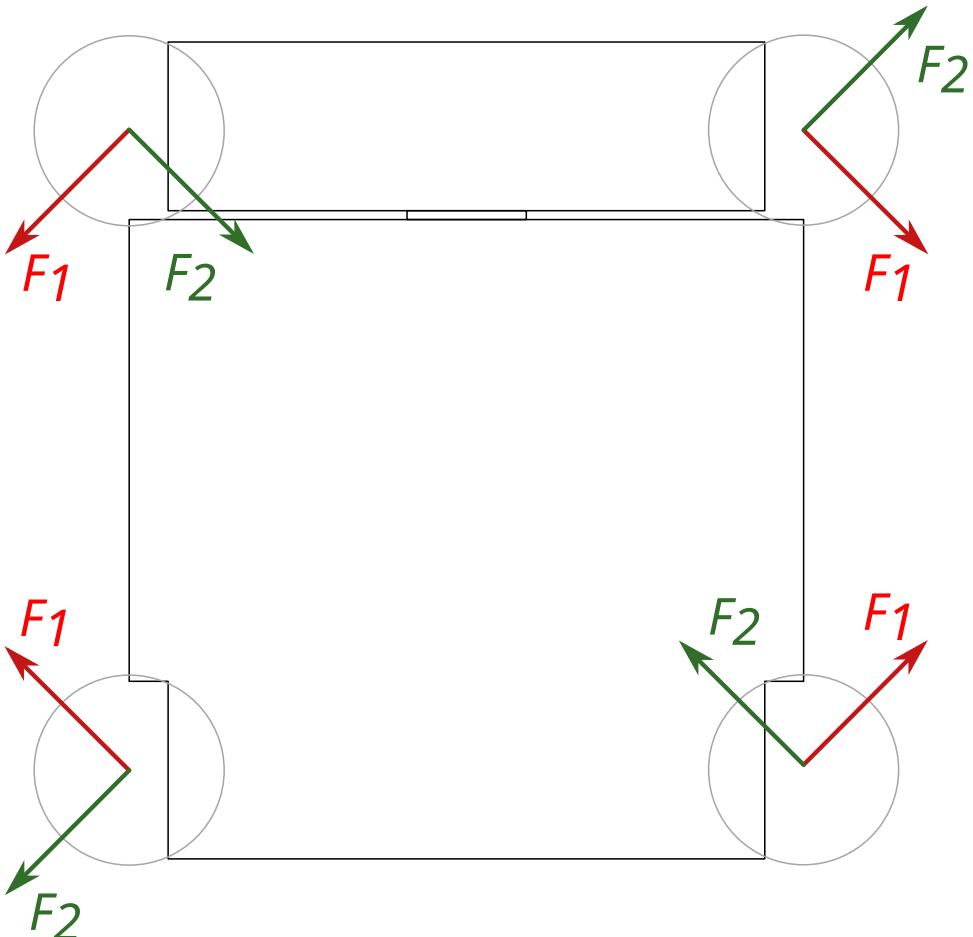


Rysunek 5.4: Wektory punktu kolizji.

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 odwzorowanie 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 je 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 5.5: 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 żądan
    \_ kula ..... Modyfikowane wektory tarcia
      \_ siatka..... Odpowiada za wygląd obiektu koła

```

Rysunek 5.6: Zagnieżdżenie obiektów koła w strukturze drzewiastej z modyfikowanymi wektorami tarcia. W implementacji Gazebo przegub i kula są zagnieżdżone równolegle.

Takie rozwiązanie wiąże się z pewnym ryzykiem. Wymaga, aby symulator używała 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.

5.8.4 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 prędkościami kół.
- Wywołanie ustawiające współczynniki tarcia wzduż wektorów F_1 i F_2 .
- Wywołanie ustawiające masy i momenty obrotowe niektórych elementów składowych konstrukcji.

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

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

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

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.

5.8.6 Obliczenia symulatora

Czujnik laserowy jest bardzo łatwo zasymulować w przestrzeni wirtualnej za pomocą rzu- towania 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 prostopadłościów zawierających obiekt, ale sposób ra- dzenia 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.

5.8.7 Różnice między czujnikiem, a modelem

Półprosta emitowana jest z punktu reprezentującego środek czujnika. Model upraszcza rzeczywisty czujnik (budowa czujnika laserowego została opisana w sekcji 2.4). Uproszczenie to polega na tym, iż nie ma wewnątrz zamodelowanego obiektu żadnego odpowiednika obracającego się lusterka. W rzeczywistym czujniku ponadto jest jeden laser, emitujący脉冲 w określonych odstępach czasu. W modelu warto zatem emitować osobne półproste, dla każdego pulsu lasera.

Można zauważyc tym samym, że model czujnika wydaje się funkcjonalnie lepszym, niż rzeczywisty LiDAR. W danej chwili, model emmituje 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 widzenia 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 zwraca niedokładne dane. Symulator ma całkowitą dowolność w ustawianiu progu, dla którego obcina pomiar.

Podobnie, jak w poprzednim przypadku, symulator posiada niezmienną w odległości dokładność pomiaru. 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.

5.8.8 Komunikacja

Bazując na architekturze opisanej wcześniej na rysunku 4.2, należy tak zbudować system, aby program sterujący mógł się komunikować w identyczny sposób z modelem 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.

5.8.9 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.

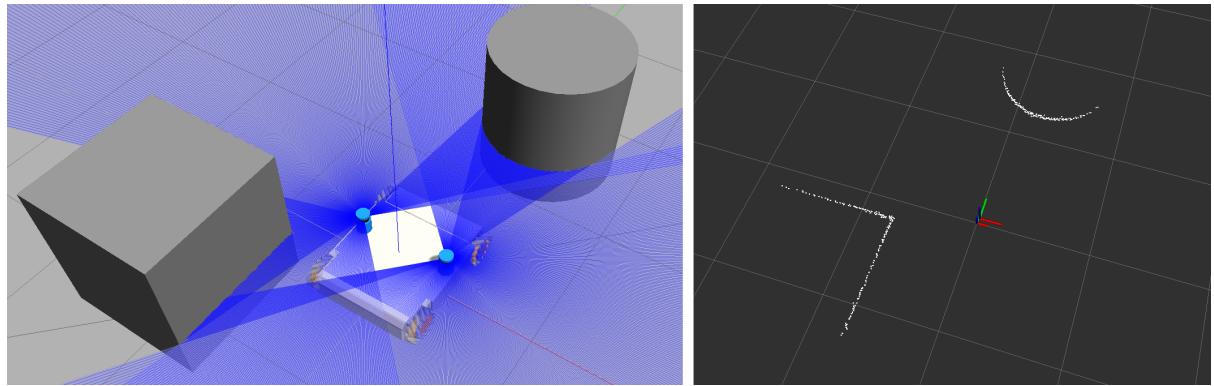
Połączenie modeli

Jak wcześniej wspomniano w sekcji 5.6, model SDF ma strukturę gwiazdową. Zagnieżdżenie modeli spowodowałoby, że powstałyby inna struktura, drzewiasta. 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 zimportowanych składowych. Zatem program sterujący czujnikiem powinien na podstawie tylko nazwy swojego obiektu ustawić przedrostek swojego interfejsu nadawania wiadomości.

Taka mechanika działania wydaje się mało zrozumiała i nieintuicyjna, jednak doskonale dba o zachowanie spójności modelu. Wszystko nadal pozostaje 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 5.7: Zrzut ekranu platformy z Gazebo i wygenerowane dane, obserwowane w Rviz.

Symulator platformy zawiera drugi program, który w każdym cyklu symulacji nadaje demonowi ROS pozycje i rotacje środków czujników laserowych, dla uproszczenia wzgółdem początku układu współrzędnych, punktu (0,0,0). Program sterujący modelem samej platformy także nadaje ramkę z pozycją i rotacją platformy wzgółdem globalnego środka układu współrzędnych. Dokładnie taki sam efekt byłby, gdyby nadawać stałą pozycję i rotację czujników laserowych, ale wzgółdem ramki platformy (nadawanej przez inny sterownik). Stałą, ponieważ czujniki nie zmieniają swojej pozycji na platformie, są przytwierdzone na stałe.

5.8.10 Błędy

Jak podano wcześniej w tabelce 2.2, 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 cięźko 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 2.2, może być bezpośrednio zapisana do elementu odchylenia standardowego, w pliku SDF opisującym czujnik. Wadą takiego rozwiązania jest niemożność modyfikacji tego parametru w trakcie wykonywania programu, gdyż Gazebo nie wystawia API do modyfikacji tej wartości. Aby temu zaradzić, wystarczy obliczać błąd standardowy w programie sterującym i manualnie dodawać go do zwróconej przez symulator tablicy danych. Funkcje do obliczania błędu standardowego zostały wprowadzone do standardu języka C++ w 2011 roku.

Na zrzucie ekranu 5.7 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.

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.

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 wzduł 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 przestrzeń wirtualną, 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 wyprowadzający pozycję obiektów na podstawie nadanych sił korzysta wewnętrznie z wartości prędkości dla każdego obiektu. Zatem kwestią symulacji tego sensora polega na odczytaniu 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 4.11.

Ma nazwę kodową `lalkarz`, ponieważ steruje platformą, tak jak aktor steruje marionetką.

5.8.11 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 mogłaby bez większego problemu pracować z interfejsem tek-

stowym w terminalu, aby być bardziej przenośna i lżejsza w zasobach, lecz nie mogłaby wykrywać zdarzeń puszczenia klawisza (bez bezpośredniego czytania z urządzenia `/dev/input/eventX`, do czego są potrzebne prawa roota). Dodatkowo, interfejs graficzny pozwala na wyświetlenie dokładniejszych wskaźników i elementów wskazujących.

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

5.8.12 Komunikacja

Program potrafi generować dwa typy wiadomości.

Pierwszą są prędkości kół `omnivelma_msgs/Vels`, jakie w danej chwili platforma powinna przyjąć na sterowanie. Pozwala to na dokładne przetestowanie zachowania się modelu platformy. Można także wywołać takie prędkości, które nie powinny być używane przy rzeczywistym sterowaniu, gdyż wprowadzają duże nieścisłości ruchu (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 4.6.

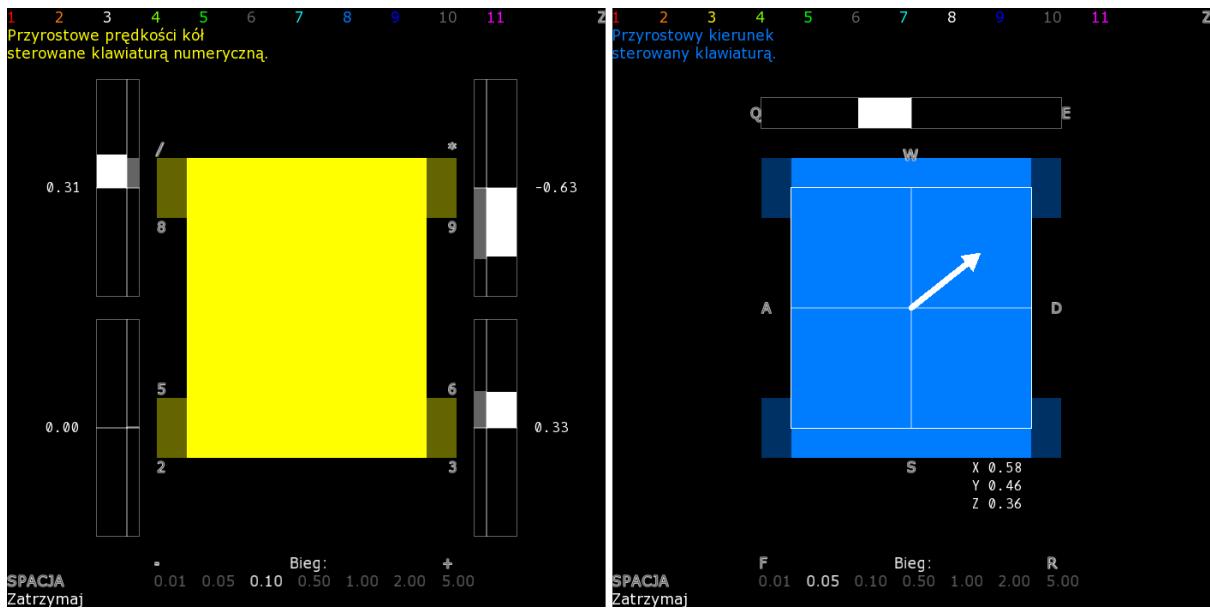
Program opcjonalnie przyjmuje także wiadomość `omnivelma_msgs/Vels`, aby wyświetlić dane enkoderów. Należy zauważyć, że nie przyjmuje całego pakietu `omnivelma_msgs/EncodersStamp`, jaki jest generowany przez model platformy, a jedynie mały ich wycinek, gdyż tylko te informacje jest w stanie wyświetlić 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 4.9.

5.8.13 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 platformie 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 5.8.6.
3. Naciśnięcie klawisza płynnie zwiększa, lub zmniejsza prędkość koła. W trakcie braku aktywności użytkownika, platforma porusza się z ustalonymi prędkościami kół.

4. Podobnie, co w poprzednim trybie, lecz pozwala na schodkowe ustawienie prędkości kół co $0,1 \frac{\text{rad}}{\text{s}}$ (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 prędkościami 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 Mecanum.
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 kólek, wprowadza to nieznaczne poślizgi. Można także modyfikować obrót klawiaturą w płynnym trybie przyrostowym.



Rysunek 5.8: Zrzuty ekranu dwóch trybów działania programu.

Interfejs składa się z listy trybów, wyświetlanego 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 liczbę podanej przy kole, pomnożonej przez aktualny bieg. Dla obrotów to są $\frac{rad}{s}$, dla prędkości to $\frac{m}{s}$.

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.

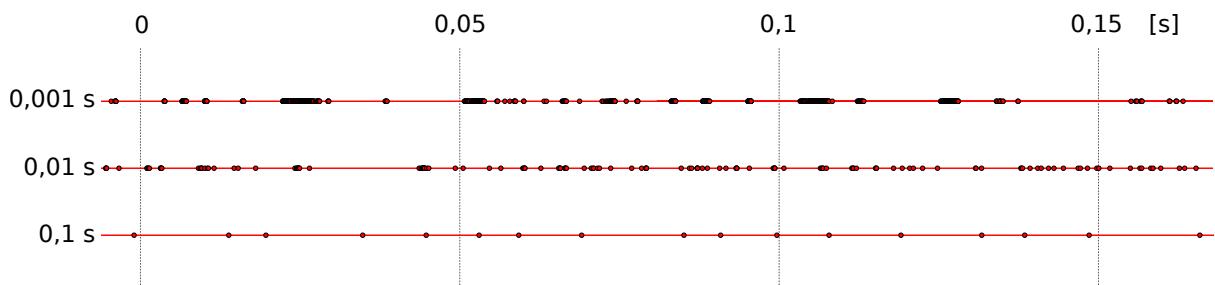
Po upływie czasu t , program nadaje dodatkową wiadomość z nowymi danymi, oraz przestawia dane generowane przez wywołania pierwszego licznika.

Po wykorzystaniu danych, algorytm nadal generuje wiadomości z zerowymi 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 5.9: 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, należało by 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.

Każda wiadomość przekazywana pomiędzy komponentami jest zwykle zagnieżdzoną 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 4.7.

ROS nie pozwala na automatyczne odbieranie tylko części pakietu, dlatego 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.

Robi się to wewnętrznie w identyczny sposób, jak w przypadku kół platformy, co zostało opisane w sekcji 5.8.4.

W tym przypadku jednak powinno się ustawić identyczne wektory tarć F_1 i F_2 , aby podłoż 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 zmienoprzecinkowe.

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

Odczytuje nazwę nadajnika i odbiornika wiadomości typu `sensor_msgs/Imu`, oraz wielkość bufora. Używany jest w przeprowadzeniu testów w sekcji 6.3. Nazwa kodowa to `odszumiacz`.

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 opóżnia 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. Wprowadząby 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`.

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

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

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

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.

Program periodycznie wysyła wiadomość typu `geometry_msgs/Twist` o kierunku równoległy 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.

Nazwa kodowa i jednocześnie przedrostek wszystkich zawartych typów to `omnivelmana_msgs`.

Rozdział 6

Testy systemu

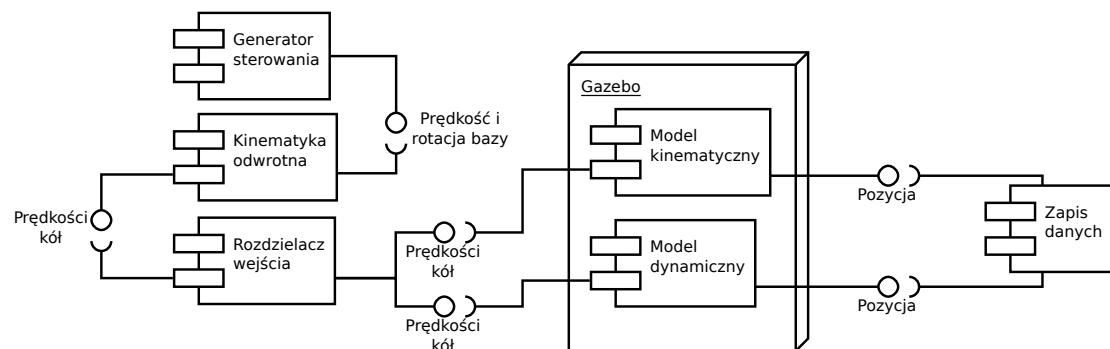
W tym rozdziale przedstawione są różne konfiguracje pakietów, wraz z wykresami ruchów platform, oraz wnioski płynące z tych zachowań. Każdy test wymaga innego podłączenia komponentów do siebie nawzajem.

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 łączą.

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



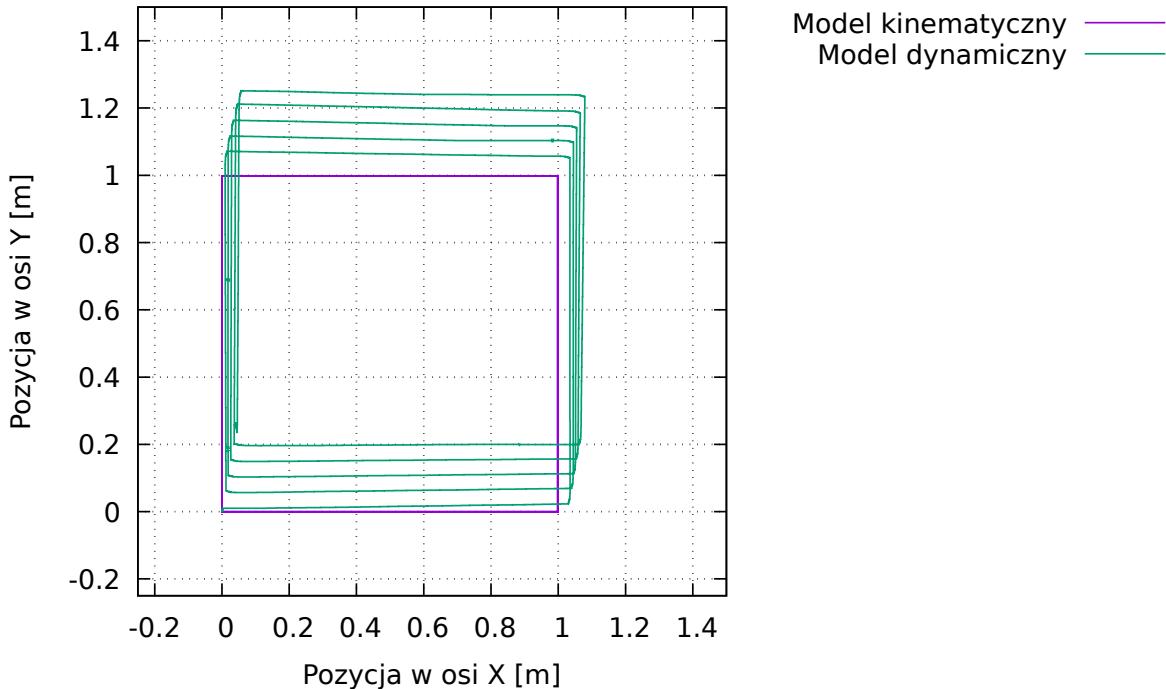
Rysunek 6.1: Połączenie komponentów w celu przetestowania wzajemnego ruchu modelu kinematycznego i dynamicznego.

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 zmienoprzecinkowych pojedynczej precyzji, zatem nie wszystkie siły działające na obiekty będą się dokładnie równoważyć i może istnieć mały ruch w losowych kierunkach i rotacjach.

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.

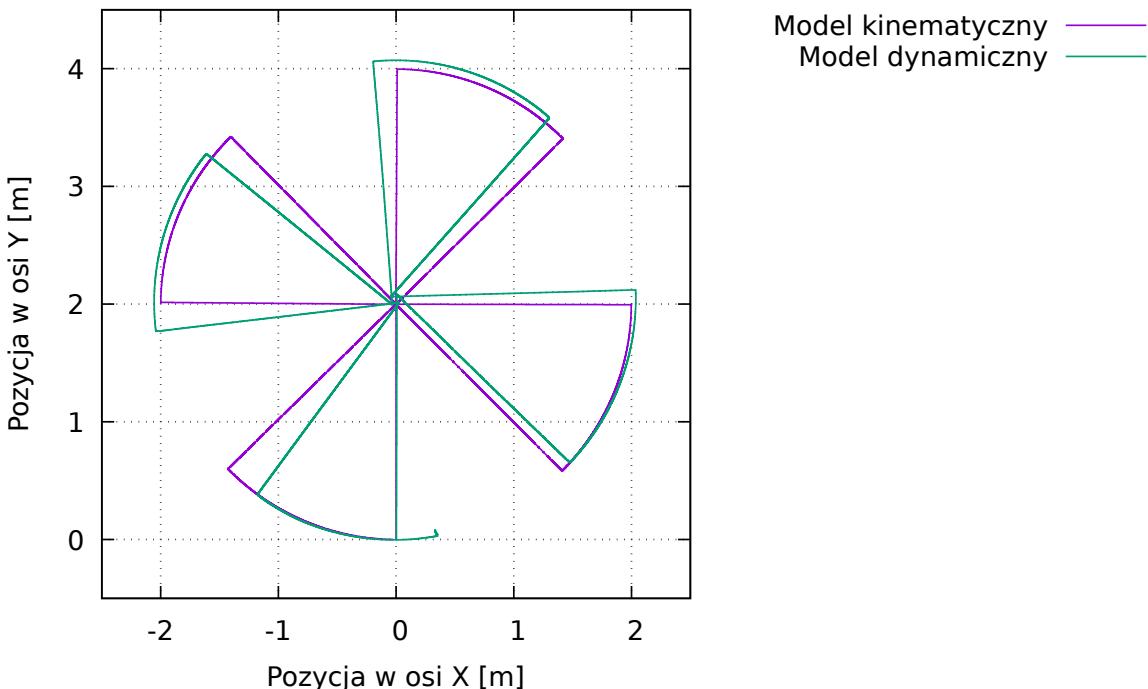
Przy zmianie prędkości widać poślizg przy zatrzymywaniu się z danego kierunku. Kąty trasy są zaokrąglone w jednym kierunku, co pokazuje, że model dynamiczny posiada bezwładność. Gdy otrzymuje nowe prędkości kół, nadal porusza się jeszcze przez pewien czas w poprzednim kierunku. Poślizg przy ruszaniu 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 obiegiem 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° , po czym odbywała się jazda w tył. Na koniec ruch w bok przy jednoczesnym obrocie i powtórzenie cyklu. Czterokrotne wykonanie tego zamyka trasę.



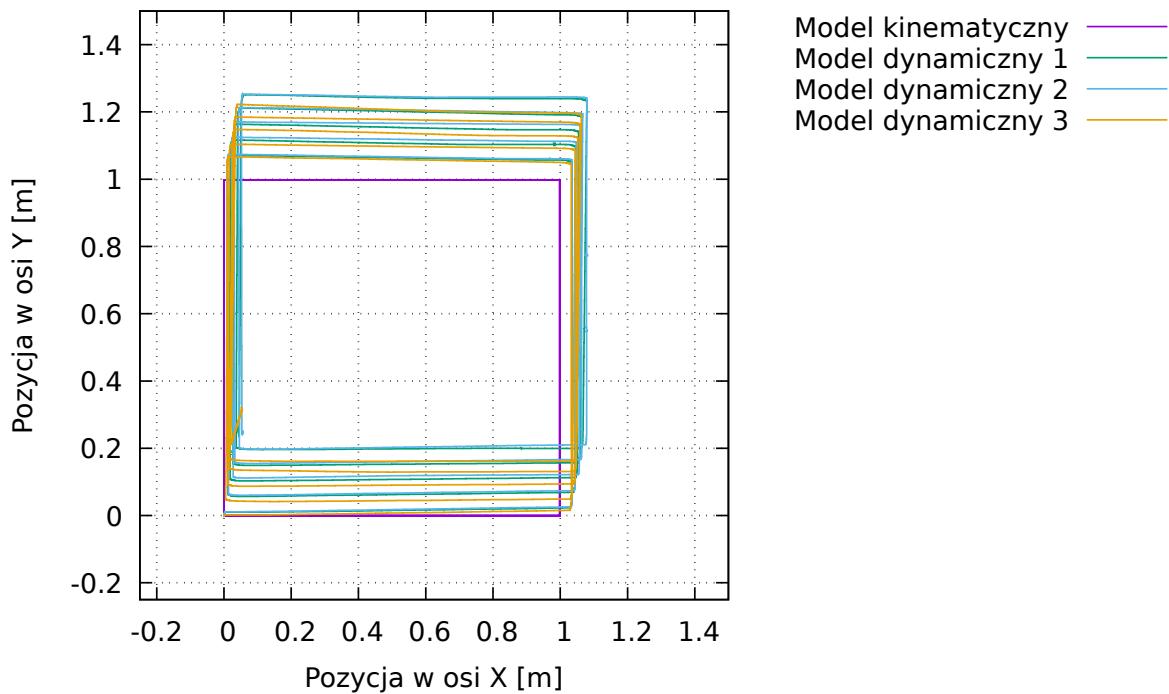
Rysunek 6.3: Ruch modelu kinematycznego i dynamicznego po rozetowej ścieżce.

Z wykresu wywnioskować można, prócz tego co z poprzedniego eksperymentu, 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.

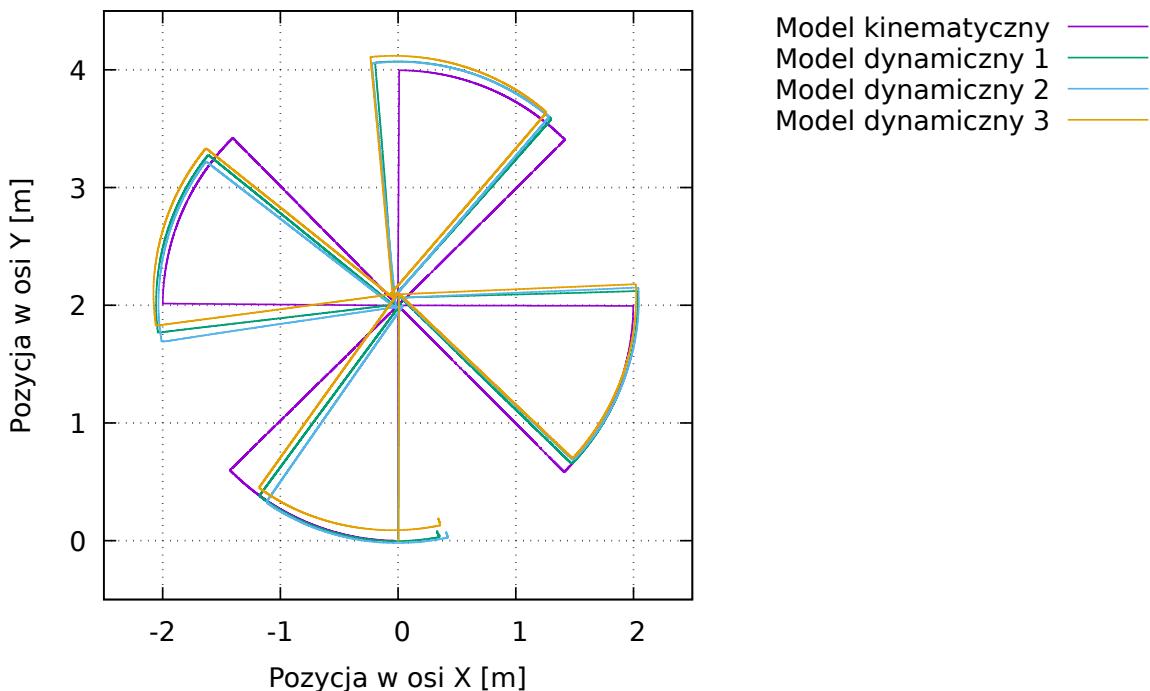
6.1.3 Powtarzalność testów

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 niewauważalne. 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 simulację.

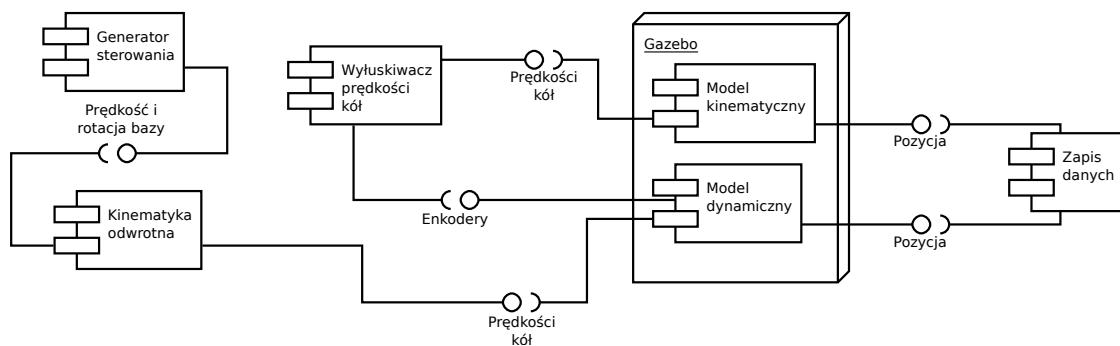
Aby zmniejszyć te niedoskonałości, należałoby, po pierwsze uruchomić symulację na systemie czasu rzeczywistego, a po drugie zwiększyć dokładność reprezentacji liczb zmiennoprzecinkowych, a co za tym idzie, znacząco zwiększyć obciążenie procesora.

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. Należałoby 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 niestandardowym 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.

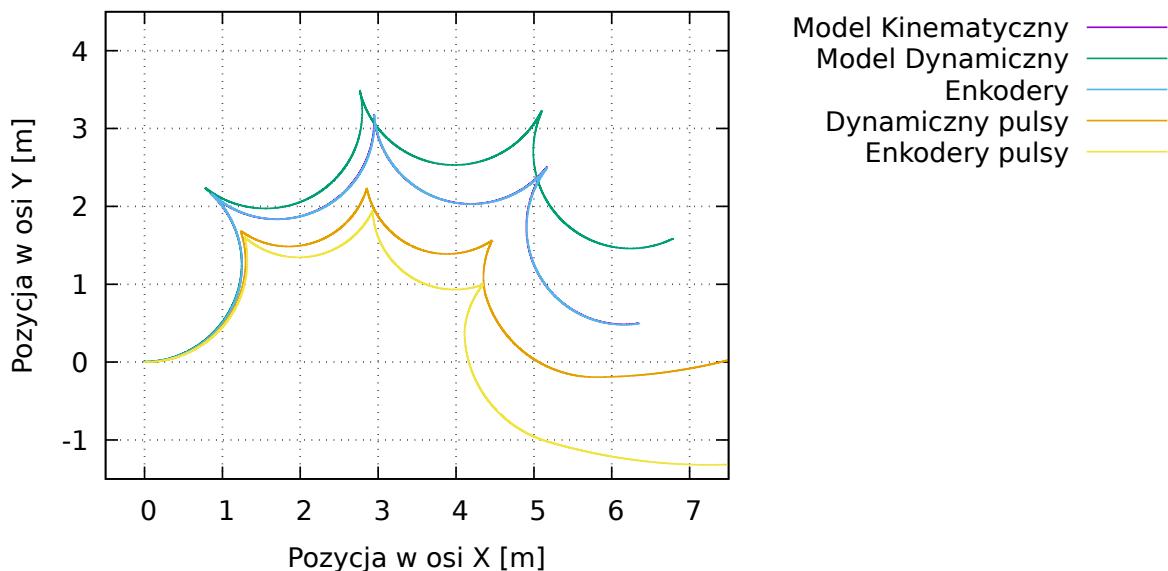


Rysunek 6.6: Połączenie komponentów w celu sprawdzenia poprawności działania enkoderów modelu dynamicznego.

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 połączeniu 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

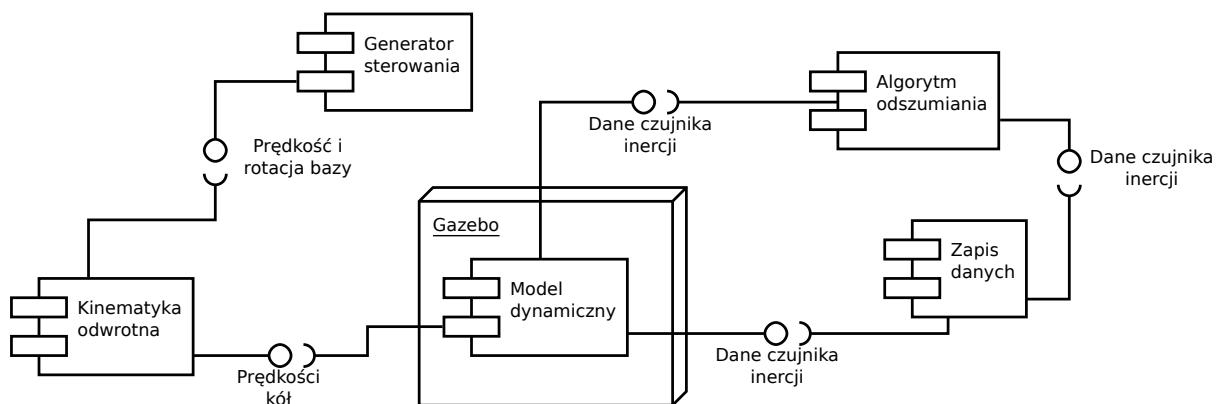
Jedno wywołanie metody maszyny symulacyjnej fizyki nadaje prędkość kołom. Następnie, z powodu symulowanych oporów, prędkość koła naturalnie spada, aż do ponownego nadania prędkości. To, jak prędkość ruchu spada, widać, gdyż segmenty przebiegu w dwóch ostatnich wykresach nie są łukami, jak ma to miejsce przy ciągłym nadawaniu prędkości. Po nadaniu ostatniej wiadomości, zatrzymująccej 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 zebraaniu 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.

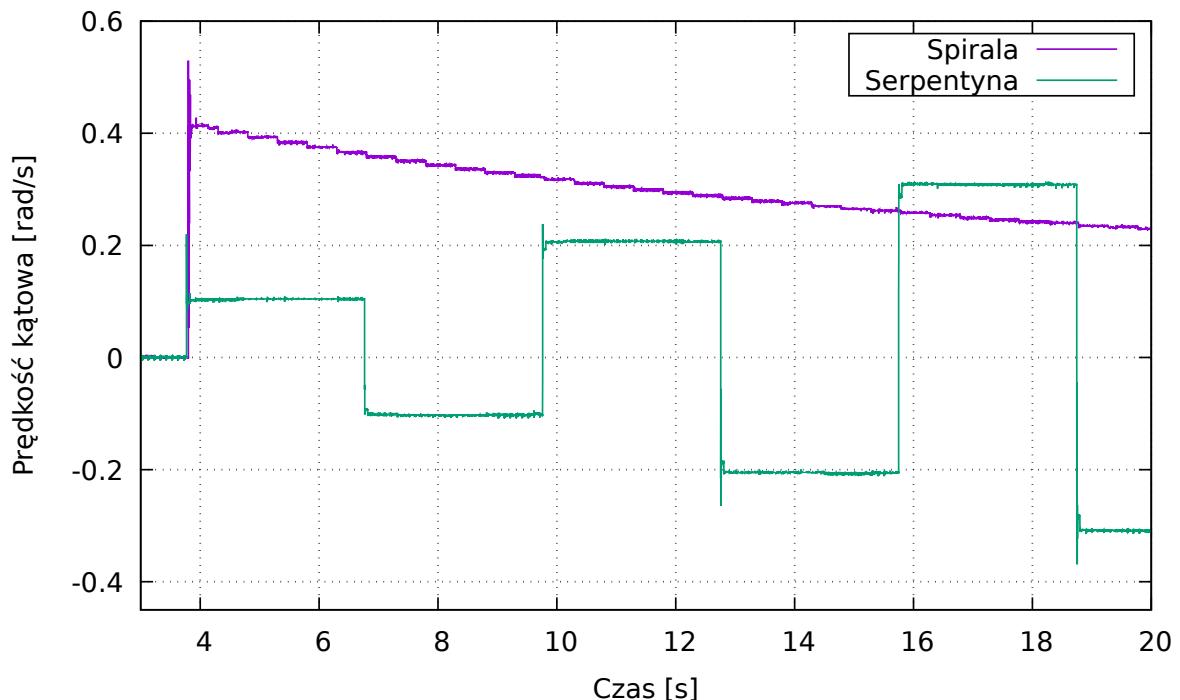
6.3.1 Czujnik prędkości kątowej

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.

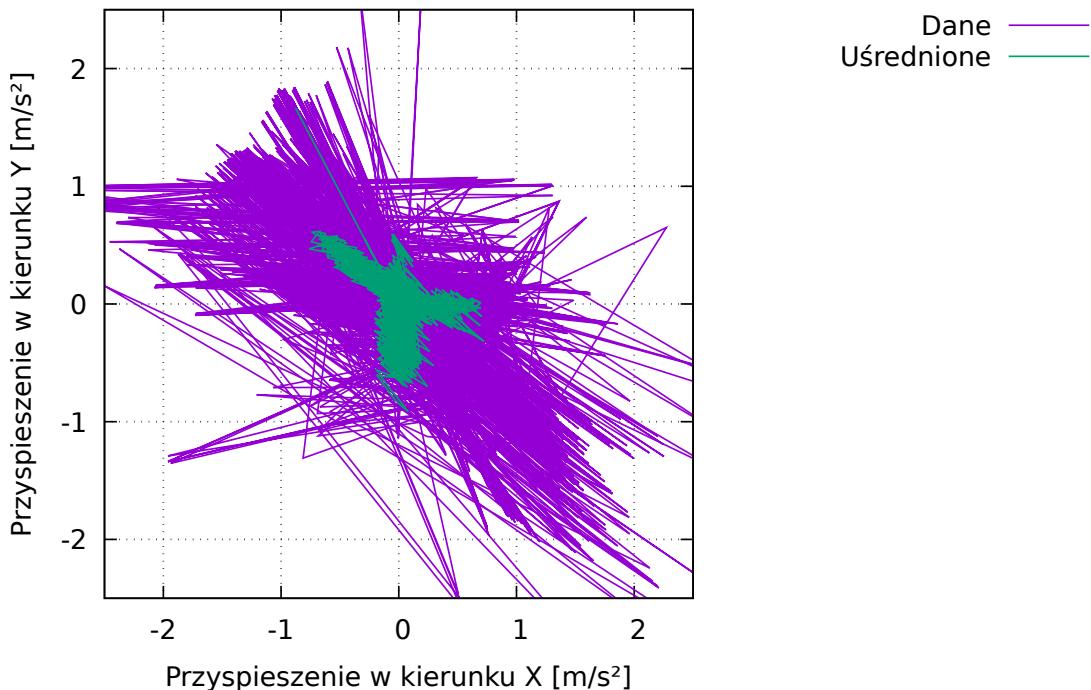
Na początku generowany jest szpiculec, przy natychmiastowym nadaniu prędkości kątowej i liniowej platformie. Jest to wewnętrzny szum generowany przez maszynę symulacyjną fizyki, spowodowany dużą zmianą wielu jej parametrów. Również wszystkie składowe elementy modelu są symulowane oddzielnie, połączone razem za pomocą więzów. Zatem informacja nadająca prędkość platformie przechodzi przez kilka warstw, zanim ustawi odpowiednie prędkości wszystkim składowym systemu.

Widać także niewielką spreżystość 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 simulatora 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° , 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.

Widać także, że wyraźnie zapisuje się tendencja danych do oscylowania na prostej pod kątem 45° .

Rozdział 7

Podsumowanie

Wszystko ładnie pięknie.

Bibliografia

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

- [12] Strona internetowa programowej struktury ramowej *Robot Operating System*.
<http://www.ros.org>
- [13] Strona internetowa standardu SDF.
<http://sdformat.org/spec>
- [14] Strona internetowa SICK, producenta czujników laserowych.
<https://www.sick.com/de/en/detection-and-ranging-solutions/2d-lidar-sensors/lms1xx/lms100-10000/p/p109841>
- [15] Strona internetowa sprzedawcy czujnika inercji.
<https://www.digikey.com/product-detail/en/analog-devices-inc/ADIS16460AMLZ/ADIS16460AMLZ-ND/5957823>
- [16] Dokumentacja ODE z wyjaśnieniem działania kolizji.
http://ode-wiki.org/wiki/index.php?title=Manual:_Joint_Types_and_Functions#Contact