



Politechnika Warszawska
Wydział Elektroniki i
Technik Informacyjnych

WYDZIAŁ ELEKTRONIKI
I TECHNIK INFORMACYJNYCH

Sprawozdanie z Pracowni Dyplomowej
Inżynierskiej I — PDI2

Informatyka

Tytuł:

Symulacja dookólnej bazy mobilnej

Autor:

Radosław Świątkiewicz

Opiekun naukowy:
dr hab. inż. Wojciech Szynkiewicz

Warszawa, 17 grudnia 2017

Streszczenie

Ta praca opisuje projektowanie i budowę środowiska symulacyjnego dla wielokierunkowej platformy mobilnej poruszającej się za pomocą kół szwedzkich.

Platforma jest bazą mobilną dla dwuramiennego robota Velma. Celem pracy jest stworzenie możliwie dokładnego modelu symulacyjnego rzeczywistej bazy mobilnej. Model ten ma służyć do wstępnych badań algorytmów planowania ruchu i sterowania robotem mobilnym.

Rozpatrzone są tutaj wymagania i problemy przy tworzeniu każdego ze składników środowiska. Na system składają się wirtualne efektory i receptory obsługujące odpowiednią maszynę symulacyjną.

Spis treści

1 Wstęp	3
1.1 Cel	3
1.2 Dookólna platforma mobilna	4
1.3 Koła szwedzkie	8
1.3.1 Działanie platformy	9
1.4 Czujnik laserowy	11
1.4.1 Zasada działania	12
1.4.2 Komunikacja	13
1.4.3 Podstawowe cechy	13
1.5 Składniki systemu	14
1.5.1 Model 3D	16
1.5.2 Sterownik silników	17
1.5.3 Sterownik czujników	17
1.5.4 Program sterujący	18
1.6 Technologie	19
1.6.1 ROS	19
1.6.2 Gazebo	21
1.6.3 V-Rep	22
1.6.4 Narzędzia	22
1.7 Plan pracy	23
1.8 Istniejące implementacje	24
2 Model platformy	26
2.1 Sposób zapisu w formacie SDF	27
2.2 Model kinematyczny	28
2.2.1 Problemy implementacji	28
2.2.2 Komunikacja	29
2.2.3 Zachowanie	29
2.3 Model dynamiczny	29
2.3.1 Jak największe zbliżenie do oryginału	30
2.3.2 Resetowanie pozycji koła	31

2.3.3	Zmiana osi rolki	32
2.3.4	Modyfikacja kierunków i wartości wektorów tarcia	33
2.3.5	Komunikacja	35
2.3.6	Rozszerzenie modelu	35
2.4	Porównanie modeli	36
3	Model czujnika laserowego	39
3.1	Obliczenia symulatora	39
3.2	Różnice między czujnikiem, a modelem	40
3.3	Komunikacja	40
3.4	Model w Gazebo	41
3.4.1	Połączenie modeli	41
3.4.2	Mechanika ramek	42
3.5	Błędy	44
3.5.1	Błąd gruby	44
3.5.2	Błąd systematyczny	44
3.5.3	Błąd pomiarowy	45
4	Komponenty systemu	46
4.1	Manualne sterowanie	47
4.1.1	Program	48
4.1.2	Komunikacja	48
4.1.3	Tryby działania	49
4.2	Wyłuskanie struktury wiadomości	51
4.3	Podłoga o zmiennym współczynniku tarcia	52
4.4	Obserwator symulacji	52
4.5	Model kinematyki odwrotnej	52
4.6	Mapa z symulacją	53
4.7	Rozdzielacz pakietów	53
4.8	Prosty program sterujący	54
4.9	Struktury pakietów wiadomości	54
4.10	Zewnętrzne pakiety	54
4.10.1	Rysownik wykresów	54
4.10.2	Wizualizer pomiarów	55

Rozdział 1

Wstęp

1.1 Cel

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

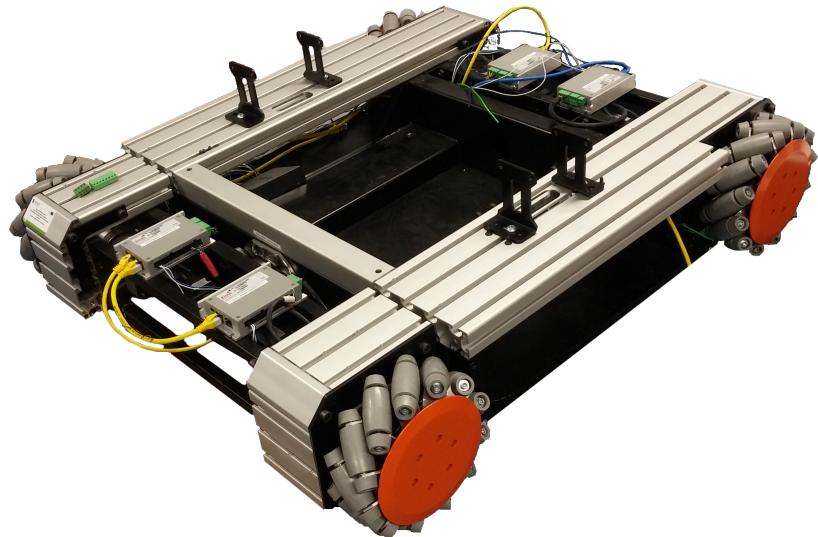
Celem jest stworzenie modelu, który będzie reagował na siły podobnie do rzeczywistego robota i był sterowany tak samo, jak rzeczywisty robot. To spowoduje, że możliwe będzie stworzenie jednego wspólnego programu sterującego do użycia zarówno w symulacji, jak i rzeczywistym robocie.

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

Oprócz modelu bazy jezdnej, środowisko symulacyjne musi również udostępniać modele czujników, w które wyposażony jest robot. Odczyty z symulatorów czujników są następnie wykorzystywane w układzie sterowania

do generacji odpowiednich sygnałów sterujących. W celu możliwie wiernej symulacji działania czujników, do wartości pomiarów dodaje się szum pomiarowy i zakłócenia.

1.2 Dookólna platforma mobilna



Rysunek 1.1: Dookólna baza mobilna na kołach szwedzkich.

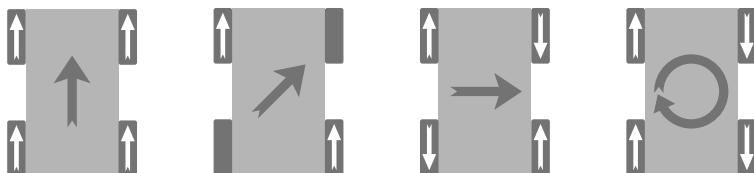
Jest to duża, prostokątna baza dookólna poruszająca się na czterech kołach szwedzkich, patrz fotografia 1.1. Koła są stałe, parami przytwierdzone do dwóch osi. Każde koło jest sterowane osobno przez podłączony bezpośrednio servomotor, zatem może mieć prędkość i kierunek niezależny od prędkości pozostałych kół, kierunku poruszania się robota, oraz jego obrotu. Każdy z servomotorów ma także wbudowany enkoder. Sterownik enkodera zwraca aktualny kąt i prędkość obrotu.

Jest to najpopularniejsza budowa dookólnych platform mobilnych, mająca zastosowanie także w innych robotach, jak na przykład Kuka Youbot 1.2. Istnieją także roboty o trzech kołach szwedzkich, w których to koła rozstawione są promieniście pod kątami 120°. Pomimo prostszej budowy i takiej samej ilości stopni swobody, co czterokołowa wersja, stabilność takiego rozwiązania jest gorsza od zastosowanej tutaj budowy [7]. Ponieważ jest to robot transportowy, to stabilność odgrywa tu ważną rolę i czterokołowa budowa jest wskazana.



Rysunek 1.2: Przykład innej platformy wielokierunkowej na podstawie fragmentu komercyjnego robota Kuka Youbot. Należy zwrócić uwagę na charakterystyczne ustawienie kół, identyczne jak w opisywanej platformie 1.1.

Odpowiedni obrót kół względem bazy, pozwala na jej ruch w dowolnym kierunku, niezależnym od kąta obrotu robota, patrz rysunek 1.3. Jest możliwe także obracać bazę, gdy ta porusza się w dowolnym kierunku, bądź stoi w miejscu.



Rysunek 1.3: Podstawowe ruchy, jakie może wykonywać robot o napędzie wielokierunkowym.

Przykładowo, poruszając tylko przeciwnymi kołami po przekątnej, system będzie mógł poruszać po skosie, bez zmiany kąta obrotu. A jeśli do tego dodać obrót kół drugiej przekątnej, w odwrotnym kierunku, wtedy pojazd zacznie się poruszać w bok, pomimo faktu że koła nie są skrętne i nie mogą ustawić się prosto do kierunku jazdy. Trasa po której porusza się robot, przy stałej prędkości kół, zawsze jest okręgiem, można uznać prostą za okrąg o nieskończonym promieniu, a punkt za okrąg o zerowym. Wynika to z faktu, że każdy obiekt, który ma jednostajną prędkość i stały kierunek w lokalnym

układzie współrzędnych, oraz prędkość kątową, będzie się poruszał po takiej krzywej.

Podstawa ma za zadanie transportować robota manipulującego Velma, tworząc razem manipulator mobilny. Velma to wysoki i bardzo ciężki robot, wyposażony w dwa chwytki na ramionach o wielu przegubach, patrz fotografia 1.4. Taka budowa wymaga szerokiej podstawy, aby zachować bezpieczną równowagę całości. Jeżdżąc na tej podstawie, robot może się przemieszczać i obracać w dowolnym kierunku, aby uzyskać lepszy dostęp do manipulowanych przedmiotów. Dodatkowe czujniki laserowe umieszczone tuż nad postawą odpowiadają za wykrywanie kolizji i lokalizację.



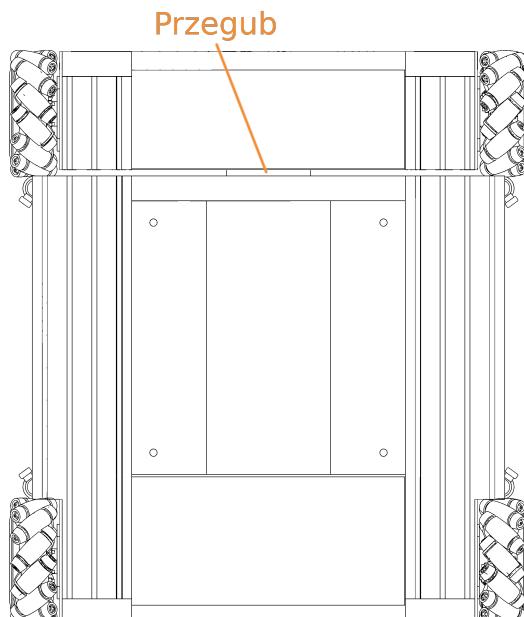
Rysunek 1.4: Robot manipulacyjny Velma.

Platforma jest niesymetrycznie podzielona na dwie niezależne części, przednią i tylną, w sposób pokazany na rysunku 1.5. Przegub o jednym stopniu swobody (tzw. zawias) jest jedynym łącznikiem pomiędzy tymi dwoma fragmentami. Zadaniem tego przegubu jest zmniejszanie wpływu nierówności podłożu na ruch bazy, aby każde koło dociskało do podłożu z taką samą siłą, jak po drugiej stronie osi. Bez tego zawiasu nierówny teren uniemożliwiałby sprawne sterowanie platformą na skutek niedeterministycznego tarcia kół tej samej osi, powodując nieplanowany skręt. Niedeterministyczne tarcie kół jest niewykrywalne w bezpośredni sposób, więc należy je wyeliminować

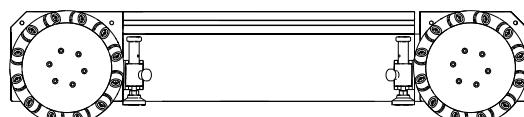
na przykład za pomocą takiego przegubu [8].

Platforma nie jest idealnym kwadratem, jest 4 cm różnicy między szerokością, a długością robota. Także środki kół nie są ustawione na wierzchołkach tej figury geometrycznej. Szerokość jest większa, co można zobaczyć porównując widok z prawej strony 1.6 z widokiem z tyłu 1.7. Dokładne wymiary są podane na rysunku 2.1 i tabeli 2.1.

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



Rysunek 1.5: Platforma mobilna — widok od góry. Przegub zawiasowy łączy dwie części.



Rysunek 1.6: Platforma mobilna — widok z prawej strony.



Rysunek 1.7: Platforma mobilna — widok z tyłu.

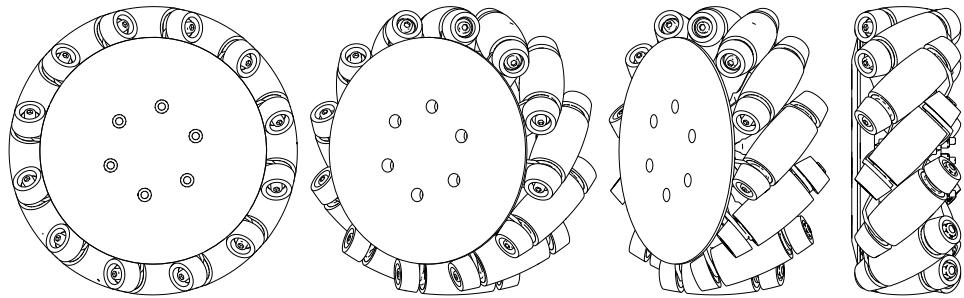
Platforma posiada 3 stopnie swobody.

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

1.3 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 1.8. W platformie ich osie ustawione są w ten sposób, że osie najwyższych, lub najniższych, rolek dwóch kół z tej samej strony robota przecinają się pod kątem prostym. Innymi słowy, robot ma identycznie ustawione koła na przeciwnieństwach wierzchołkach, i razem ustawione są w kształt litery X patrząc na nie z góry. Warto pamiętać, iż oś aktualnie dolnej rolki jest prostopadła do osi górnej rolki.

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



Rysunek 1.8: Widok 12 rolkowego koła szwedzkiego opisywanej platformy wielokierunkowej.

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

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

Na podstawie rysunku 1.8 widać, że krzywizna rolki jest tak ustawiona, aby punkt kontaktu rolki z podłożem w czasie obrotu płynnie przechodził na następną rolkę. Celem jest utrzymanie równej odległości osi od płaszczyzny podłożu. Nie powinno być efektu przeskoku z jednej rolki na drugą, gdyż to wprowadza nierówne tarcie, losowe poślizgi i nadmierne zużycie elementów wykonawczych. Kształt pojedynczej rolki zawiera się w paraboloidzie, wzory opisujące kształt rolki są złożone. Zazwyczaj przybliża się taką rolkę wycinkiem torusa, w celu uproszczenia produkcji [4].

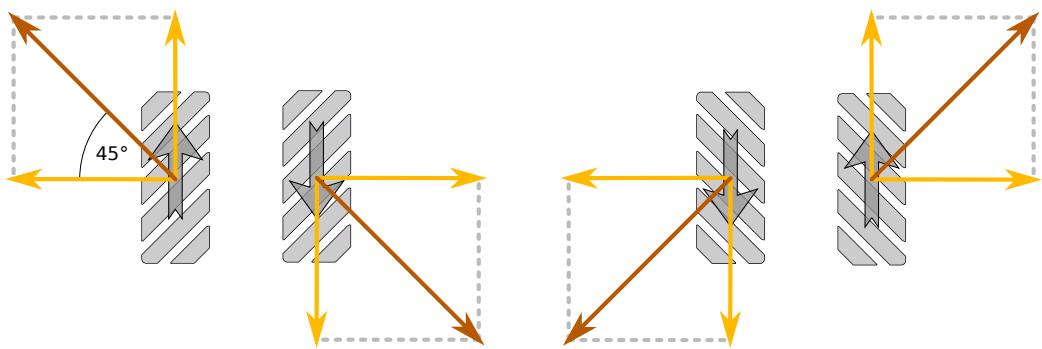
Istnieją także inne budowy kół, złożone z wielu małych rolek, tak aby w każdym momencie więcej jak jedna rolka dотykała podłożu. Można także złożyć kilka powyższych kół obok siebie w jedno koło. Przydatne jest to dla robotów transportujących duże masy, gdyż zmniejsza to obciążenie pojedynczych rolek.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

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

Dodano więc dwa czujniki laserowe, opisane dokładniej w sekcji 1.4, aby program sterujący nie bazował jedynie na odometrii przy wyznaczaniu sterowania.

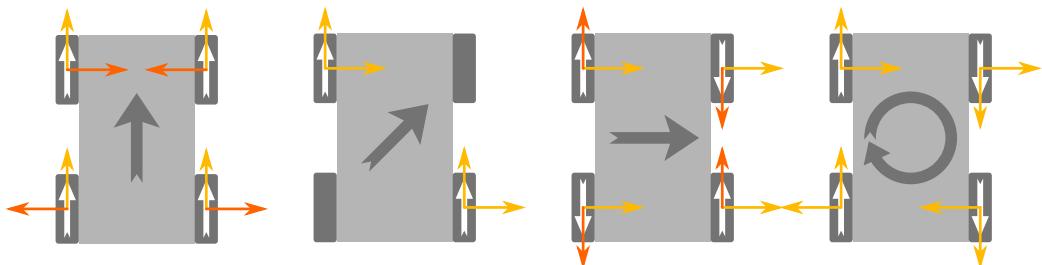
1.3.1 Działanie platformy

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



Rysunek 1.9: Wektory składowe i wypadkowe koła widzianego z góry.

Ustawiając te koła w odpowiedni, opisany wcześniej na obrazku 1.5, sposób, można wywołać odpowiednie znoszenie się składowych prędkości, a w efekcie pozwolić robotowi na poruszanie się w kierunkach nieosiągalnych dla pojazdów o standardowych kołach. Warto nałożyć te składowe na wcześniejszy rysunek 1.3, aby dokładniej zobaczyć, dlaczego koła nadają platformie daną prędkość wypadkową przy odpowiednim obrocie kół.



Rysunek 1.10: Ruchy platformy widzianej z góry, z nałożonymi składowymi wektorów prędkości. Ciemniejszym kolorem zaznaczono znoszące się składowe.

Warto rozpatrzyć każdy przypadek.

1. Wektory o kierunku prostopadłym do pionowej płaszczyzny symetrii urządzenia znoszą się, pomimo że mają przeciwnie zwroty na przedniej i tylnej parze kół. Pozostają jedynie składowe równoległe do płaszczyzny symetrii, które powodują prostoliniowy ruch naprzód.
2. Dwa koła nie obracają się. Nie jest to ruch pasywny, gdyż taki wprowadzałby nieprzewidywane poślizgi, a aktywne hamowanie. Wektory się nie znoszą i platforma wykonuje ruch pod kątem 45° do płaszczyzny symetrii.

3. Ruch podobny jest do przypadku 1. Tutaj również wektory znoszą się parami, jednak tym razem na prawych kołach i lewych. Różnica ruchów polega na większej prędkości obrotu pojedynczych rolek wokół ich osi.
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 prosto-padły 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, jaki styl jazdy opłaca się zastosować, aby zminimalizować zużywanie się elementów jest dużą, odrębną dziedziną nauki. Odpowiednio skomplikowany algorytm sterowania może brać pod uwagę to zachowanie się tych elementów składowych.

1.4 Czujnik laserowy

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



Rysunek 1.11: Czujnik laserowy SICK LMS100-10000.

1.4.1 Zasada działania

Wszystkie czujniki 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 emisuje pulsacyjną wiązkę podczerwonego promienia co pewien okres czasu. 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 odbywa się za pomocą różnych interfejsów sieciowych, zazwyczaj w architekturze typu master-slave.

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

urządzenia.

1.4.2 Komunikacja

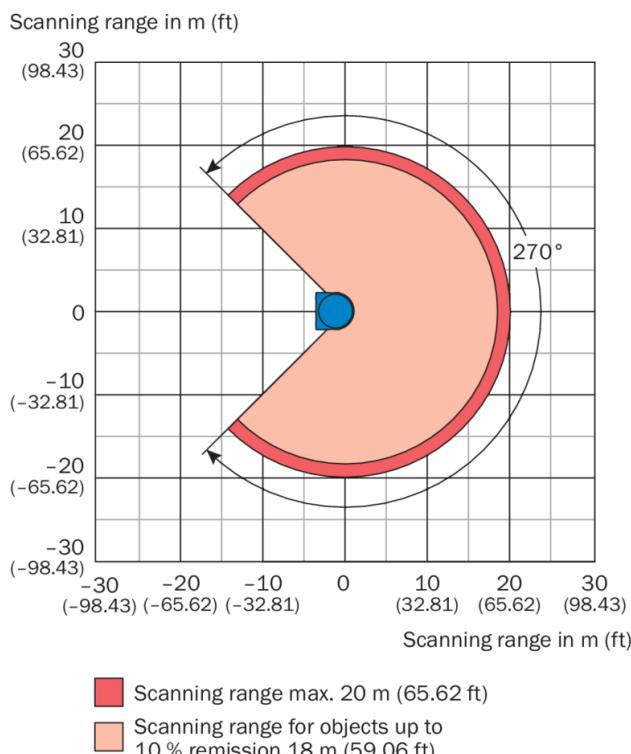
Wysyłając do czujnika odpowiedni ciąg bajtów, można ustawić jego tryb działania, odpytać o zebrane dane, czy wykryć konfigurację i stan.

W przypadku naszej platformy, komunikacja odbywa się poprzez interfejs Ethernetowy. Program komunikujący się bezpośrednio z urządzeniem zwraca pakiety zawierające pomiary z ostatniego obrotu czujnika, oraz dodatkowe dane opisujące sam pomiar, takie jak czas, początkowy kąt pomiaru, tryb itp. Dokładne dane i cechy czujnika dostępne są na stronie producenta [13].

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

1.4.3 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.



Rysunek 1.12: Wykres producenta dotyczący zasięgu czujnika.

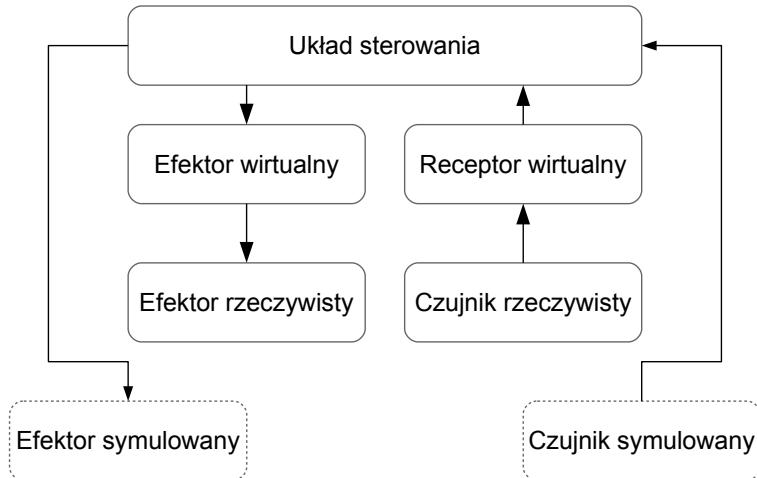
Cecha	Wartość
Kąt pracy	270°
Długość fali światła lasera	905 nm (podczerwień)
Częstotliwości skanowania	25 Hz / 50 Hz
Maksymalna odległość obiektu	≈ 20 m
Rozdzielcość kątowa	0,25° / 0,5°
Systematyczny błąd pomiarowy	±0,03 m
Przypadkowy błąd odległości	0,012 m

Tablica 1.1: Podstawowe cechy czujnika laserowego.

Urządzenie jest w stanie komunikować się za pomocą portu szeregowego RS-232, połączenia Ethernetowego i sieci CAN.

1.5 Składniki systemu

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



Rysunek 1.13: Struktura agenta upustaciowanego.

Można to przedstawić za pomocą zapisu agentowego, rysunek 1.13. Agent upustaciowiony składa się z kilku modułów, komunikujących się ze sobą za pomocą różnych interfejsów.

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

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

Zamodelowany efektor symulowany również przyjmuje te same sygnały do układu sterowania, co efektor rzeczywisty, lecz nie zamienia ich na sygnały sterujące, a wywołuje odpowiednie funkcje maszyny symulacyjnej, nadające siły i prędkości obiektom w przestrzeni wirtualnej.

Receptor wirtualny pobiera surowe dane z czujnika, przekształca na odpowiedni format, usuwa błędy i szum tak, aby program sterujący mógł wykorzystać te dane w prosty sposób. Doskonałym przykładem jest tutaj urządzenie Kinect (widoczne na robocie Velma na fotografii 1.4), w którym to zachodzi odczytanie obrazu z kilku kamer. Następnie obraz przesyłany jest do komputera, w którym sterowniki interpretują dane, usuwając błędy, two-

rzą mapę głębokości, wykrywają szkielety i sylwetki osób. Te dane mogą być wykorzystane łatwo w grach i programach sterujących.

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

1.5.1 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, materiał fizyczny i wygląd. Dodatkowo należy uwzględnić wszystkie więzy, w postaci symulowanych przegubów. W przypadku tej bazy istnieje typ więzów o jednym stopniu swobody (zawias), używany przy połączeniu przedniej i tylnej części platformy, oraz jako piasta kół i rolek. Można także uznać, że więzy bez stopni swobody używane są do trwałego połączenia czujników z platformą i transportowanym robotem. Więzy mogą oddziaływać siłą na elementy do których są podłączone, symulując silniki.

Elementy składowe i symulowane przeguby oddziałują bezpośrednio z maszyną do symulacji fizycznej. To kształt, masy i momenty bezwładności brył są argumentami funkcji liczących. Maszyna symulacyjna oblicza odpowiednie prędkości i nadaje je podanym obiektom w podobny sposób, jak ma to miejsce w rzeczywistości.

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

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

1.5.2 Sterownik silników

Program sterujący generuje abstrakcyjne dane, na przykład liczbę zmienno-przecinkową, zapisaną binarnie. Przykładowy silnik fizyczny nie jest w stanie działać na ich podstawie, sam silnik do pracy potrzebuje odpowiedniego napięcia na wejściu, ale interfejs serwomotoru na przykład przyjmuje bardziej abstrakcyjne dane. 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 odpowiedniej fali PWC, lub obsługa przetwornika cyfrowo-analogowego. Do innych zadań może należeć kontrola, czy żądana wartość nie uszkodzi urządzenia. Zazwyczaj sterownik może komunikować się z powrotem z resztą systemu, aby zgłaszać ewentualne awarie.

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

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

1.5.3 Sterownik czujników

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

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

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

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

1.5.4 Program sterujący

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

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

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

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

1.6 Technologie

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 odzwierciedlenia 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 są otwartoźródłowe, inne są własnościowe. Mogą używać tylko procesora, lub też być wspomagane przez kartę graficzną (na przykład *PhysiX*). Niektóre maszyny symulują, prócz zderzeń obiektów, także rozpływ cieczy, dymy, płotna, ciała sprężyste i strukturę wewnętrzną brył, lecz te funkcjonalności nie są potrzebne dla symulacji opisywanej platformy. Nazywa się je czasami „silnikami symulacji fizyki”, co jest bezpośrednim tłumaczeniem nazwy *physics engine* z języka angielskiego.

1.6.1 ROS

Platforma programistyczna do pobrania z [11]. ROS jest skrótem od *Robot Operating System*, lecz jego nazwa jest myląca. Nie jest to system operacyjny, lecz obszerna platforma programistyczna (*framework*) zawierająca odpowiednie biblioteki i narzędzia do tworzenia programów sterujących. ROS stara się w łatwy sposób dostarczyć wszystko, co potrzebne do budowy logiki aplikacji sterowania. Są tu algorytmy wyznaczania tras, budowy map, manipulowania robotycznymi ramionami itp.

Twórcy zachęcają, aby uzupełniać brakujące moduły swoimi własnymi, a potem dzielić się nimi z resztą programistów. Na ich stronie internetowej znajduje się obszerna baza danych różnych komponentów, do używania we własnych systemach.

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

Działanie systemu jest oparte o pakiety. Każdy pakiet jest katalogiem zawierającym w sobie pliki opisujące jego parametry i skrypty CMake, używane do komplikacji. Pakiet może być programem wykonywalnym, danymi, definicjami, lub zestawem plików. W symulacji opisywanej platformy, modele są plikami i programami, łączonymi razem we wspólnym pakiecie, ładowanym do pakietu programu wykonywalnego. Jest to dokładnie opisane w rozdziale 4. Pakiety mogą być zależne od siebie, ale nigdy nie wskazują nawzajem

swoich bezpośrednich ścieżek.

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

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

Instalacja programu na systemie operacyjnym jest złożona. Z wyjątkiem odpowiednich wersji Ubuntu, nie ma łatwego sposobu na instalację go na innych systemach. Na przeszkodzie stoją błędy komplikacji dla nowszych wersji komplikatorów, 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.

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 dostarczonych 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 dostępna moc obliczeniową, potrzebną takim programom w dużych ilościach.

1.6.2 Gazebo

Program do pobrania z [9]. Ten symulator graficzny jest dość prosty w obsłudze, skupia się na symulowaniu podanych danych, a nie na możliwości ich łatwego przygotowania. To znaczy, działa na podstawie manualnie napisanych plików konfiguracyjnych. Zazwyczaj używany w trybie wsadowym, uruchamiany z argumentami z linii poleceń i plikiem `.world`, opisującym simulację. Plik ten zawiera nazwy i ścieżki umieszczanych w symulacji modeli i wtyczek. Z tego powodu interfejs graficzny jest dość ubogi.

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

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

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

Wtyczka do sterowania modelem jest skompilowaną biblioteką, dołączaną na starcie programu. Tworzy się ją w C++, lub Pythonie, jako klasę dziedziczącą po abstrakcyjnej klasie dostarczonej przez Gazebo. Dzięki temu może korzystać ze wszystkich funkcjonalności systemu operacyjnego, jak na przykład komunikacja za pomocą pamięci współdzielonej. Gazebo dostarcza także swój własny mechanizm kolejek wiadomości, który sprawdza się w jednolitej komunikacji z zewnętrznymi programami, korzystającymi z bibliotek dostarczonych przez Gazebo.

Program jest w pełni wspierany na dystrybucji GNU/Linuksa Ubuntu ale bez problemu można go także skompilować pod inne systemy. Interfejs jest dopracowany i przestrzega wielu ustawień systemowych, na przykład takich jak DPI, lecz nie korzysta z dedykowanych bibliotek do tworzenia interfejsów typu Qt, lub GTK. Uruchamianie programu jest proste i nie wymaga dodat-

kowych ustawień, wywoływania skryptów inicjalizujących, tworzenia odpowiednich katalogów, czy definiowania zmiennych systemowych. Podobnie jak inne programy, tworzy ukryty katalog w katalogu domowym użytkownika, gdzie składuje wszystkie modele i logi.

Gazebo jest składnikiem systemu ROS, kod źródłowy jest dzielony w ramach wspólnej organizacji. Chociaż różne osoby odpowiadają za rozwój tych oprogramowań, kolejne wersje Gazebo są powiązane z kolejnymi wersjami ROSa, nie można użyć przestarzałej wersji Gazebo z nowszym ROSem i odwrotnie. Symulator można zainstalować osobno, lub jako jeden z pakietów ROSa.

1.6.3 V-Rep

Program do pobrania z [10]. Duże i skomplikowane środowisko, reklamujące się wieloma zaawansowanymi mechanizmami i funkcjami. Pomimo otwartego kodu, użycie komercyjne jest płatne. Dla zastosowań akademickich program jest rozdawany bez opłat. Bogaty interfejs graficzny zakłada budowę i symulację wszystkiego w tym jednym programie.

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

Problemem jest także zapisywanie utworzonych w systemie modeli. Program tworzy drzewiastą strukturę modelu, w pliku binarnym własnego formatu, co uniemożliwia edycję i wizualizację modelu bez uruchamiania całego programu i importowania modelu do symulacji. Brak przenośności, czy wsparcia systemu kontroli wersji dla takich nietekstowych plików także jest problemem.

Pisanie wtyczek najczęściej odbywa się w Lua. Są też jednak dostępne inne języki skryptowe, jak C, Matlab, Java itp. Komunikacja z innymi programami odbywa się poprzez specjalne dodatki 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.

1.6.4 Narzędzia

Do tworzenia oprogramowania na systemach Unixowych można użyć dowolnych edytorów, gdyż standardowo wszystko jest potem komplikowane za po-

mocą narzędzi wiersza poleceń i skryptów. Jednak warto sobie ułatwić pracę zaawansowanymi środowiskami graficznymi.

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

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

CMake to popularny i polecaný przez ROS i Gazebo system budowy kodu.

Program tworzy na podstawie swoich plików konfiguracyjnych plik `makefile` do komplikacji źródeł i łączenia bibliotek.

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

KDevelop nadają się do pisania kompilowalnego kodu wtyczek. Można połączyć je pod komendę `make` i korzystać z mechanizmów interpretacji błędnych wierszy, graficznego debugowania i podobnych.

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

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

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

1.7 Plan pracy

1. Należy stworzyć model w SDF zachowując wszystkie rozmiary i momenty rzeczywistej wersji. Bryły składowe modelu muszą przypominać kształtem części z których składa się robot, należy im także ustawić parametry fizyczne, jak masę, moment bezwładności, materiał itp.
2. Zamodelować wszystkie więzy na koła, rolki i przegub, aby maszyna symulacyjna poprawnie symulowała obiekt. Taki model powinien na tym stanie poprawnie reagować na wirtualne siły, lecz jego efektory nie będą jeszcze aktywne. Można go prosto побieżnie przetestować działając silną na elementy i patrząc, czy reagują w spodziewany sposób.

3. Zapisanie wtyczki sterującej w Gazebo odczytującej odpowiednie dane z zewnątrz i wywołujączej funkcje maszyny symulacyjnej, aby modyfikować ruch modelu. Na tym poziomie można dobudować zamiennik programu sterującego jedynie do podawania prostych wartości bez odczytywania pomiarów i sterowania.
4. Zaprogramowanie wtyczki symulującej czujniki, aby generowały dane z enkoderów, oraz innych urządzeń, dodawały błędy pomiarowe, a następnie przekształcały dane na format zrozumiały dla programu sterującego. Czujniki nie muszą być istniejące, mogą generować dane, jak pozycja i rotacja bardzo trudne do uzyskania rzeczywistymi czujnikami.
5. Wystawienie do zmiany w czasie rzeczywistym masy, momentu bezwładności, współczynników tarcia, aby pozwolić na proste testowanie działania systemu z różnymi współczynnikami.
6. Elementy pomagające w symulacji, jak model kinematyczny, sterowany funkcją matematyczną, i podłożę ze zmiennym współczynnikiem tarcia.
7. Programy pomocnicze zbierające i wyświetlające dane, interfejs graficzny do prostego sterowania robotem.
8. Uproszczony program sterujący, aby zbadać, czy system działa poprawnie na tyle, aby rozwinać go w końcowy projekt.
9. Program sterujący w ROS. Największy i najbardziej skomplikowany element, wspólny dla obu bytów — wirtualnego i rzeczywistego. Zwykle nie jest to praca jednego człowieka, a jego rozwój nie ustaje przez długi czas. Ten program dostarczy funkcji, aby wyższy sterownik robota mógł użyć tego modułu do sterowania jazdą i odczytywania danych.

1.8 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 zarówno w Gazebo, jak i w V-Repie. Tylko w przypadku V-Repa, mamy wstępny sterownik do którego wysyłamy odpowiednie wartości kierunku, a on nadaje takie prędkości kołom, aby poruszać się w zadanym kierunku. Wersja dla Gazebo jest statycznym modelem z błędnie ustanowionymi przegubami, jego efektory nie są zaimplementowane.

Te profesjonalne modele także pomogą przy wstępnej weryfikacji zachowania się budowanego tutaj modelu, czy nie zachowuje się nadzwyczaj dziwnie w pierwszych fazach projektu.

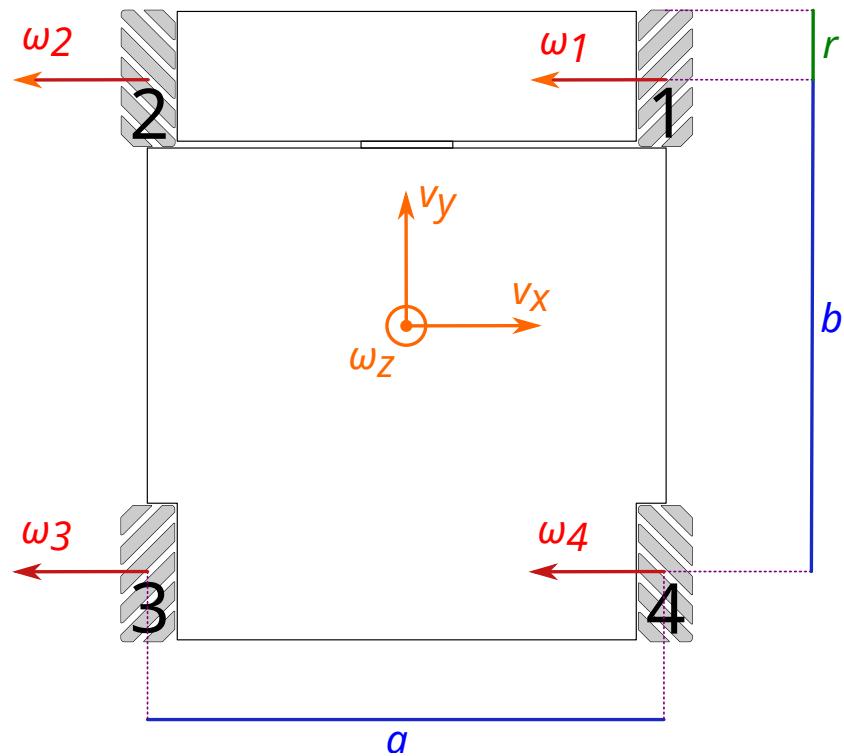
Ze względu na niezwykle zaawansowany obiekt kół i kształt rolek, trzeba będzie uprościć model poprzez zamianę niektórych składowych i dodanie sztucznych więzów. Całościowy model może być zbyt skomplikowany, aby maszyny symulacji mogły go obliczać w czasie rzeczywistym. Taki model także jest znacznie trudniej poprawnie wymodelować, ze względu na liczne tarcia i poślizgi rolek.

Rozdział 2

Model platformy

Należy wpierw stworzyć doskonały model kinematyczny, aby móc porównać z nim stworzony model dynamiczny, oraz fizyczną platformę. Dzięki temu można łatwo oszacować jak bardzo błędy symulacji, oraz błędy niedoskonałości fizycznego modelu odstają od matematycznych wyliczeń.

Dodatkowo stworzenie platformy kinematycznej pozwalało na porównanie szybkie odrzucanie niedziałających implementacji modelu kinematycznego.



Rysunek 2.1: Wielkości używane we wzorach.

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

Tablica 2.1: Opisy i wartości symboli używanych we wzorach i rysunkach.

2.1 Sposób zapisu w formacie SDF

Simulation Description Format (SDF) jest formatem XML pozwalającym na określenie elementów i zależności pomiędzy nimi w przestrzeni trójwymiarowej, w szczególności budowy i rozmieszczenia robotów. Powstał jako zamieńnik URDF ze względu na jego skomplikowaną semantykę i brak możliwości określania ważnych cech, jak rozmieszczenia elementów na symulowanej scenie, określania 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. Standard jest dobrze opisany na ich stronie internetowej [12].

Na szczytce wszystkich elementów znajduje się element `world` zawierający w sobie wszystkie modele na symulowanej scenie. Mogą być to roboty, a także przeszkody, źródła światła, elementy animowane i tym podobne. Dodatkowo można dodać informację o ustawieniach maszyny symulującej fizykę, wyglądzie sceny, wietrza, grawitacji, polu magnetycznym i tym podobnych.

W każdym z modeli oznaczonych tagiem `model` zawiera się nazwa, domyślna pozycja, sposób traktowania przez symulator i wtyczki programów obsługujących zaawansowane zachowanie modelu. Można przenieść zawartość elementu do innego pliku i określić ścieżkę do importu.

Model ma w sobie równolegle wszystkie elementy oznaczone jako `link`, każdy z nich jest osobną, pełną częścią robota, na przykład kołem, fragmentem ramienia chwytaka, trzonem. Zawiera w sobie informacje o pozycji względem innych obiektów, masie, kształcie, kolizjach, materiale fizycznym i wyglądzie. Pozwala na dodanie do siebie źródeł dźwięku, czujników, baterii 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 równoległe do elementów modelu typy `joint` określające

jace typ więzów, osie, współczynniki sprężystości, wytrzymałość, siłę silników. Każde połączenie określa między jakimi obiektyami się łączy.

2.2 Model kinematyczny

Model jest sterowany funkcjami matematycznymi, zamieniającymi prędkość ruchu kół platformy na prędkości geometrycznego środka platformy w układzie współrzędnych lokalnych oraz prędkość kątową. Również następuje tutaj całkowanie tego ruchu, aby uzyskać aktualną pozycję platformy. Te funkcje najwygodniej zapisać w postaci macierzowej tak, jak w [2]. Wzór powtarza się w wielu innych pracach naukowych, a jego dokładny kształt zależy od kolejności numerowania kół i interpretacji wymiarów. Dla naszego przypadku:

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

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

Ponieważ sterowanie pozycją modelu kinematycznego odbywa się wyłącznie poprzez wzory matematyczne, w jego symulacji nie uczestniczy maszyna symulacyjna. Taki model nie reaguje na kolizje z innymi obiektyami, nie reaguje na różnicę terenu i nie używa informacji o współczynnikach tarcia materiałów.

W kodzie nadano mu nazwę **pseudovelma** co odnosi się do tego, że jest to nieprawdziwy ruch sterowany z zewnątrz, a nie prawdziwa symulacja.

2.2.1 Problemy implementacji

Gazebo nie ma zaimplementowanego pełnego wsparcia dla standardu SDF. W szczególności nie działa struktura elementów **frame** odpowiadająca za transformacje obiektów względem innych obiektów. Nie jest to zapisane w dokumentacji, a jedynie zgłoszone od kilku lat w systemie kontroli wersji jako błędy.

Oznacza to, że wszystkie elementy typu **link** będąc dziećmi **model** nie przestrzegają jego pozycji. Powoduje to, że nadając prędkość kątową modelowi za pomocą funkcji nadajemy ją każdemu obiekowi osobno. Każde z kół i dwie części bazy obracają się zgodnie z zadanymi wartościami, ale ich środki

pozostają w miejscu w którym rozpoczęły symulację ignorując kompletnie pozycję w elemencie rodzica `model`.

Aby to naprawić, należy przenieść zawartość elementów `link` i ustawić je jako `visual` tagu `model`. W ten sposób traktowane są jako część renderowana modelu, a nie osobne składowe.

Powstała niedogodność jest taka, że ciężej jest sterować obrotem elementów `visual` i nie można sterować obrotem kół. Oczywiście ma to znaczenie jedynie kosmetyczne, gdyż w żaden sposób nie wpływa na ruch modelu bazy.

2.2.2 Komunikacja

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

Wiadomość o nowych zadanych prędkościach kół nie mieści się w standardzie, zatem został stworzony własny typ `omnivelma_msgs/Vels`. Zawiera cztery wartości zmiennoprzecinkowe podwójnej precyzji. Wartości oznaczają prędkości w rad/s.

Program posiada komunikację:

- Nadawanie w każdym cyklu symulacji wiadomości typu `geometry_msgs/PoseStamped` z aktualną pozycją i rotacją platformy oraz nagłówkiem z identyfikatorem i czasem nadania pakietu.
- Odbiór wiadomości typu `omnivelma_msgs/Vels` z zadanymi prędkościami kół.

2.2.3 Zachowanie

Platforma ignoruje kompletnie otoczenie poruszając się przez inne obiekty. Po nadaniu stałych prędkości kół następuje ruch po okregach zgodnie z przewidywaniami.

Cały czas zwraca aktualną pozycję i rotację, działając jak układ całkujący funkcje ruchu z powyższej macierzy.

2.3 Model dynamiczny

Wykorzystując maszynę do symulacji fizyki możemy umieścić w niej model określający kształty, masy i zależności obiektów, potem nadać elementom wirtualny ruch i otrzymać przybliżone wyniki do tego, jak zachowywałaby się rzeczywista platforma.

Wszystkie elementy związane z tym modelem noszą nazwę **omnivelma**, co nawiązuje do wielokierunkowości ruchów robota manipulującego którego podstawa ma transportować.

Robot jest bryłą na którą składają się następujące części składowe:

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

Jak widać jest dość skomplikowany obiekt do symulacji, dlatego bezpośrednie podejście poprzez budowę modelu jak najdoskonalszego oryginału można wykluczyć. Powodowałaby olbrzymią ilość obliczeń maszyny symulacyjnej, każde obarczone błędem.

Istnieją różne podejścia do stworzenia odpowiedniego modelu. Każde z nich było proponowane na różnorakich forach przez osoby symulujące podobne bazy. Jednak rozwiązanie nigdy nie zostało znalezione. Działający w naszym przypadku sposób także nie został wcześniej sprawdzony.

2.3.1 Jak największe zbliżenie do oryginału

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

Słabym punktem tego rozwiązania jest fakt, że rolki są niestandardowym kształtem, którego dokładność jest bardzo wysoce wymagana. Przybliżenie jej walcem powoduje problemy przy przenoszeniu ciężaru na kolejną rolkę, gdyż koło będzie musiało przez chwilę oprzeć się o krawędź. Taki model samoczynnie podskakiwałby przy ruchu zwiększać tym samym i tak duże niedokładności.

Przybliżenie rolki siatką jedynie zmniejsza powyższy efekt, gdyż sama siatka zbudowana jest z prostych odcinków. Zwiększąc jej gęstość zwiększymy jakość symulacji, ale narażamy się na olbrzymi skok ilości obliczeń. Kalkulowanie kolizji z siatką jest najdroższe z możliwych dla maszyny symulującej fizykę.

Duża liczba przegubów także utrudnia symulację, przede wszystkim, każde kolejne zagnieźdzenie nakłada błędy liczbowe poprzednich przegubów.

2.3.2 Resetowanie pozycji koła

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

Polega on na tym, że koło podłączone do bazy posiada przegub zawiasowy obrócony pod kątem 45° , tak aby był równolegle do dolnej rolki. Do tego przegubu podłączona jest kula reagująca z podłożem, którą to w każdej iteracji symulacji resetujemy do pozycji wyjściowej razem z przegubem.

- Trzon całego robota.
- Przegub zawiasowy z silnikiem, któremu nadajemy odpowiednią prędkość.
- Wirtualne koło łączące ze sobą dwa przeguby. Obraca się tak samo, jak obracało by się rzeczywiste koło.
 - Siatka powodująca ładny wygląd koła i pozwalająca na łatwe stwierdzenie jego rotacji.
 - Wewnętrzny przegub zawiasowy obrócony o 45° w stosunku do osi koła. Pozycja i rotacja tego przegubu jest resetowana do pozycji początkowej względem trzonu po każdej iteracji maszyny symulacji.
 - Kolizja koła w formie kuli, symuluje aktualnie najwyższą rolkę u podłoża. Jego pozycja i rotacja są resetowane po każdej iteracji do pozycji początkowej względem trzonu podstawy.

Wywołuje to takie działanie, jak gdyby na kole istniała jedynie dolna rolka. Przez krótką chwilę model zachowuje się poprawnie, aż obrót drugiej osi zacznie wpływać na symulację. Zanim to jednak nastąpi, jego rotacja jest przywracana do pozycji początkowej. Ponieważ robimy to natychmiast, maszyna symulacyjna nie bierze pod uwagę tarcia i ruchu w takim przypadku.

Niestety nie jest możliwe uzyskanie tego rozwiązania w Gazebo, gdyż struktura drzewiasta obiektów nie jest zaimplementowana. Metody zmieniające pozycję obiektu nie działają. W dodatku potrzebna jest także możliwość ustawiania rotacji i pozycji przegubu, elementu `joint`, co nie jest wystawione

do modyfikacji w API.

Bardzo skomplikowany sposób działania kół skłania do szukania innych rozwiązań.

Jest tutaj także jeszcze jedna, bardzo ważna cecha. Jakość symulacji zależy od jej prędkości. Jest tak ponieważ im bardziej obciążony jest symulator, tym większy czas pomiędzy kolejnymi klatkami symulacji i pomiędzy kolejnymi resetowaniami pozycji koła. Oznacza to, że druga oś zacznie wpływać na symulację z nieliniowo rosnącym błędem aż do kolejnego resetu koła. Przy odpowiednio niskim czasie odświeżania, różnice spowodują nieakceptowalny spadek jakości symulacji. Model zacznie opóźniać się względem fizycznej platformy.

2.3.3 Zmiana osi rolki

Poprzedni przypadek można zmodyfikować poprzez wyznaczanie nowej osi przegubu łączącego wirtualne koło z kulą symulującą rolkę.

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

Należało obliczyć kierunek w przestrzeni globalnej biorąc pod uwagę aktualną pozycję platformy i obrót kół. Fakt, że program wymagał wektora relatywnie do pozycji koła wirtualnego bardzo komplikował obliczenia.

Trzeba było jeszcze wybrać moment wyznaczenia nowego kierunku osi, bowiem maszyna symulacyjna wywołuje wiele różnych funkcji w jednym cyklu symulacyjnym. Można wywołać kod w różnych momentach z teoretycznie różnym efektem.

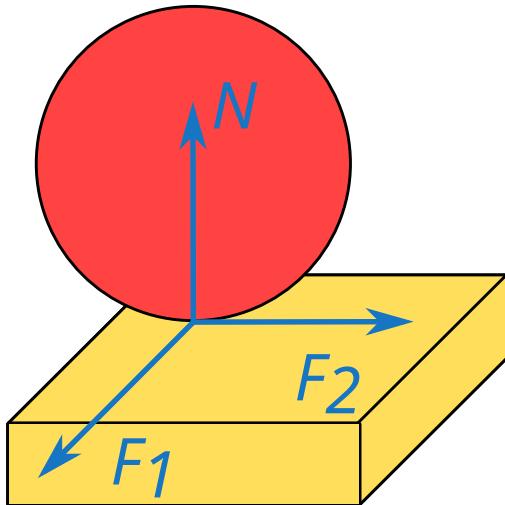
Implementacja tego rozwiązania niestety nie stworzyła działającego modelu. Wykonywanie kodu w różnych momentach symulacji nie wpływało na efekt. Platforma poprawnie poruszała się do przodu i do tyłu. Pierwszy problem pojawił się gdy w czasie ruchu na bok, prędkość malała, aby zmienić zwrot pomimo niezmiennej prędkości kół. Po kilku sekundach problem się powtarzał.

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

Wygląda na to, że problemem było wewnętrzne traktowanie przegubów przez maszynę symulacyjną. Takie nienaturalne zachowanie jak nagła zmiana osi przegubu musiała wprowadzać nieprawdopodobne wartości do zmiennych stanu, co w rezultacie powodowało po pewnym czasie chaotyczny ruch.

2.3.4 Modyfikacja kierunków i wartości wektorów tarcia

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



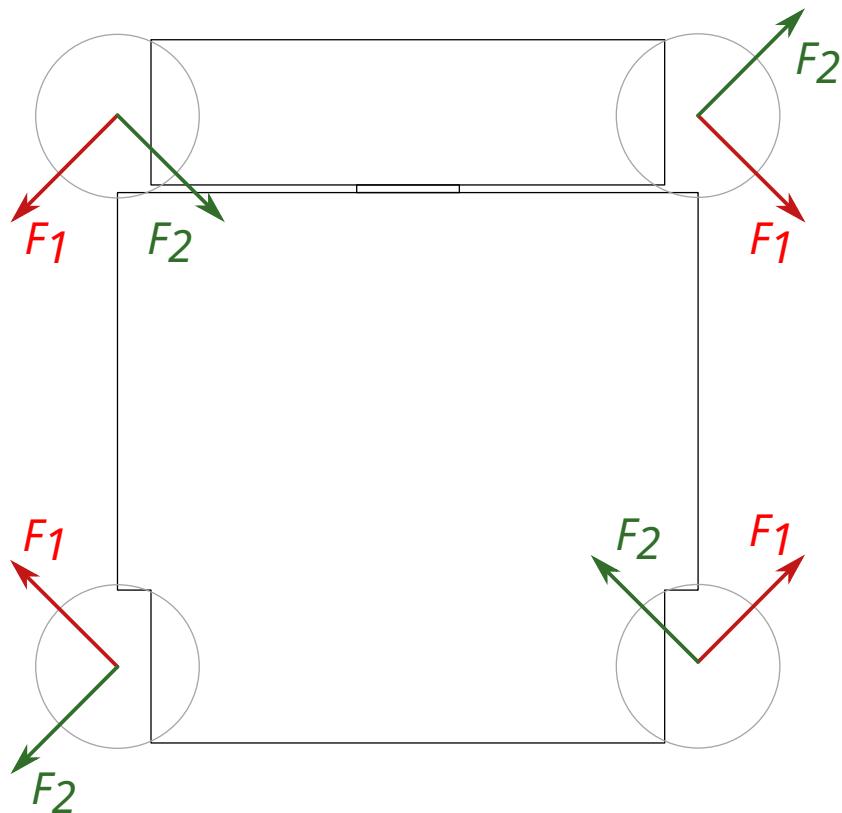
Rysunek 2.2: Wektory punktu kolizji.

Po wykryciu punktu kolizji i wyznaczeniu wektora normalnych N do dotykających się obiektów należy zadziałać odpowiednimi siłami, aby zatrzymać, lub odbić obiekty. 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 punktowi siły prostopadłą do wektora normalnych, ten wektor może być rozpisany przy pomocy dwóch wektorów jednostkowych F_1 i F_2 . Te wektory są prostopadłe do wektora normalnych, równoległe do płaszczyzny kolizji.

W normalnej symulacji nigdy nie potrzeba osobno modyfikować współczynników tarcia wzdłuż tych wektorów, gdyż powierzchnie obiektów mają równe współczynniki tarcia w każdym kierunku. Jednakże modyfikując je statycznie, lub dynamicznie, można uzyskać bardzo ciekawe efekty. Instrukcja podaje przykład w którym, aby zamodelować tarcie opon samochodu prostopadłe do kierunku jazdy, należy dynamicznie zmieniać współczynnik tarcia dla wektora F_1 , lub F_2 w tym kierunku. Współczynnik tarcia prostopadłe do kierunku jazdy może być liniowo zależny od prędkości. Spowoduje to, że im większa prędkość samochodu, tym boczna siła łatwiej zmieni tor jego jazdy, co ma odwzorowanie w rzeczywistości. Więcej informacji można znaleźć na stronie instrukcji maszyny symulacyjnej ODE [14].

W naszym modelu modyfikujemy kierunek wektora F_1 , oraz współczyn-

niki 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. W doskonałym przypadku rolka obraca się całkowicie bez tarcia, a ruch wzduż jej osi jest niemożliwy. Można więc ustawić zerowy współczynnik tarcia w kierunku prostopadłym do osi, oraz nieskończonie duży dla wektora równoległego do osi.



Rysunek 2.3: Kierunki wektorów dla których należy nadać współczynniki tarcia przy symulacji platformy. Tarcie w kierunku F_1 powinno być nieskończonie, a w F_2 zerowe.

Niestety w rzeczywistości rolki wykonane są ze śliskiego plastiku, który pozwala na mały ruch wzduż ich osi. Osie kolek również nie obracają się płynnie, trzeba użyć dużej siły, aby obrócić każdą z nich, pod naciskiem platformy tarcie jest jeszcze większe. Każda rolka w dodatku obraca się z innym tarciem wprowadzając kolejne zakłócenia. Podłożoże 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 modelujemy tylko najniższą, dotykającą podłożu rolkę. Jak wcześniej wspomniano, ma ona bardzo skomplikowany kształt, lecz można przybliżyć całe koło kulą. Mamy zatem w miejscu każdego koła po jednej kuli z dynamicznie modyfikowanym tarciem i ładną siatką w kształcie koła, oraz przegub z motorem łączący odpowiednią część bazy z kołem. To najprostsza budowa modelu (a zatem najszybsza) z poprzednich.

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

2.3.5 Komunikacja

Ze względu na wiele ustawień elementów bazy, należy stworzyć bogaty interfejs.

- Nadawanie w każdym cyklu symulacji wiadomości typu `geometry_msgs/PoseStamped` z aktualną pozycją i rotacją platformy, oraz nagłówka z czasem i identyfikatorem.
- Nadawanie w każdym cyklu prędkości platformy w wiadomości typu `geometry_msgs/TwistStamped` z nagłówkiem.
- Symulator enkodera nadający aktualną pozycję i rotację kół w wiadomości `omnivelm_msgs/EncodersStamped` z nagłówkiem.
- Odbiór wiadomości własnego typu `omnivelm_msgs/Vels` z zadanymi prędkościami kół.
- Odbiór wywołań ustawiających współczynniki tarcia wzdłuż wektorów F_1 i F_2 .
- Odbiór wywołań ustawiających masy i momenty obrotowe niektórych elementów składowych konstrukcji.

2.3.6 Rozszerzenie modelu

Ponieważ komputerowa reprezentacja liczby zmienoprzecinkowej pozwala na zapisanie nie tylko liczbowych wartości, postanowiłem rozszerzyć model

o dodatkową funkcjonalność, wywoływaną wysłaniem do modelu cichej nie-liczby (NaN) w polu zadanej prędkości. Cicha nie-liczba powstaje w procesorze przy przeprowadzaniu nieprawidłowych, acz niekrytycznych obliczeń zmiennoprzecinkowych, na przykład dzielenie przez zero, lub dzielenie nieskończoności przez minus-nieskończoność. Takie operacje nie powodują błędu programu, jedynie wynik w postaci nie-liczby propaguje przez wszystkie pozostałe operacje.

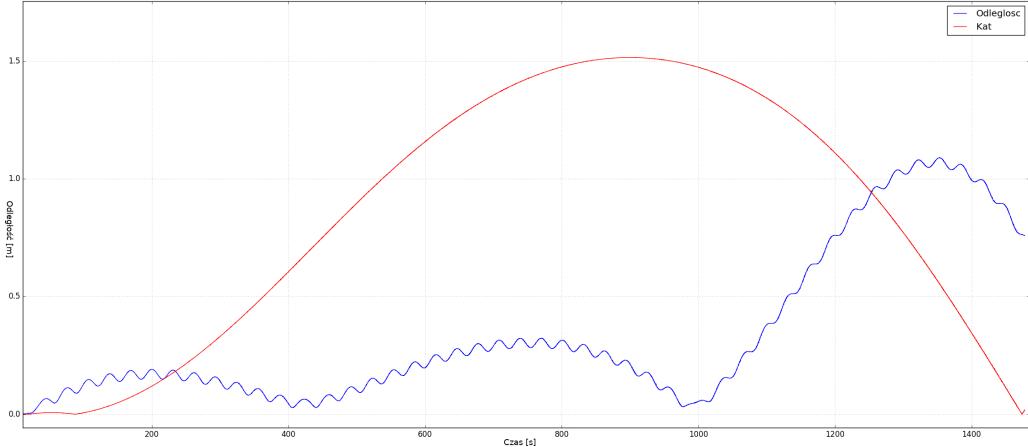
Nadanie prędkości modelom w przestrzeni wirtualnej polega na wywołaniu odpowiedniej funkcji maszyny symulującej fizykę. Można zadać sobie pytanie, jak zachowa się model, jeśli dla niektórych kół nie zmieniać prędkości po każdym odebraniu pakietu.

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

Zwraca nam to uwagę również na potrzebę, aby program do komunikacji z robotem nie skończył się błędem po odebraniu jednej z takich nie-liczb. Ten program przekształca liczby zmiennoprzecinkowe zawarte w ROSowych pakietach na dane zrozumiałe przez sterownik silnika, które zazwyczaj są liczbami stałoprzecinkowymi, nie mogącymi przedstawiać nie-liczb.

2.4 Porównanie modeli

Posiadając model kinematyczny, którego ruch jest sterowany wzorami możemy porównać jego pozycję i rotację z modelem dynamicznym. Należy w tym samym momencie nadać bazom identyczne prędkości kół i zbierać dane dotyczące wzajemnej pozycji. Nadano kołom kolejne prędkości zgodnie z numeracją w rysunku 2.1, $(-2, 1, 0, -3)$.



Rysunek 2.4: Odległość i kąt pomiędzy platformami w czasie.

Takie ustawienie spowodowało ruch po okręgu o okresie ok. 32 s. Modele początkowo posiadały zbliżoną pozycję i rotację, ale w miarę upływu czasu opóźnienia modelu dynamicznego stały się zauważalne. Także kąt między nimi zaczął się zmieniać, lecz w znacznie większym stopniu, niż wynikłoby to z opóźnienia pozycji. Przez cały czas trwania eksperymentu model dynamiczny zrobił kąt pełny w stosunku do kinematycznego, a jego odległość zmieniała się sinusoidalnie. Oba modele wykonały tę samą ilość obiegów.

Po pierwsze widać, że już na samym początku pomiarów odległość nagle wzrosła, co jest spowodowane poślizgiem platformy przy nadaniu kołom prędkości. Model kinematyczny rozpoczyna jazdę natychmiastowo. Dzieje się tak pomimo doskonałym współczynnikiem tarcia, w dodatku masa platformy nie jest dobrana zgodnie z rzeczywistością. Ten efekt może być zmniejszony za pomocą ustawiania współczynnika tarcia podłożu, ale nie jest możliwe jego całkowite wyeliminowanie, gdyż jest to cecha maszyn symulacji fizyki.

Drugą zauważalną cechą wykresu są małe oscylacje o okresie jednego obiegu. Ma to efekt taki sam, jak gdyby obie platformy wystartowały z różnych pozycji, których odległość jest mniejsza od średnicy trasy. To powodowałby, że odległość między ich pozycjami oscylowałaby w właśnie taki sposób. Modele wystartowały z tego samego punktu w tym samym czasie, jednakże środek ciężkości modelu dynamicznego nie pokrywał się ze środkiem platformy względem którego obliczano pozycję. W związku z tym jej ruch odbywa się wokół środka ciężkości masy, ale pozycja jest liczona względem geometrycznego środka. To pokazuje, że należy bardzo dokładnie ustawić masy elementów składowych w stosunku do rzeczywistego modelu, gdyż w przeciwnym wypadku symulacja obarczona będzie właśnie takimi oscylacjami. Co więcej, ruchome elementy transportowanego robota manipulującego będą

miały wpływ na pozycję środka ciężkości i ruch podstawy. Dlatego należy wprowadzić element transportowanej masy, którego środek ciężkości powinien być modyfikowalny.

Trzecią cechą są duże oscylacje, rosnące w czasie. Na razie nie jest dokładnie wiadome, dlaczego powstają. Warto jednak zauważyć, że nie są zbieżne w czasie z kątem obrotu, gdyż jego wartość ma mniejszy okres.

Opóźnienia powstałe na modelu dynamicznym są cechą dokładności maszyny symulacyjnej fizyki i nie da się ich całkowicie wyeliminować. Jednakże ich wartość jest znacznie mniejsza od niedokładności wprowadzonej przez niedoskonałość rzeczywistego obiektu. To oznacza, że model ma rzeczywistą wartość pod kątem pomocy w symulacji robota.

Rozdział 3

Model czujnika laserowego

Istnieje dedykowany obiekt w standardzie SDF dla tego typu czujników. Również Gazebo wspiera wizualnie symulację poprzez możliwość renderowania zasymulowanych impulsów lasera. Tak, jak model platformy, ten pakiet otrzymał nazwę kodową `monokl`, ponieważ pozwala obserwować otoczenie, jak okular.

3.1 Obliczenia symulatora

Czujnik laserowy jest bardzo łatwo zasymulować w przestrzeni wirtualnej za pomocą rzutowania półprostych. Jest to jedna z podstawowych technik renderowania obrazu. Używa się jej także przy obliczaniu symulacji fizycznej i specjalnych wydarzeń związanych, na przykład, z grami komputerowymi.

Półprosta jest emitowana z jakiegoś punktu w jakimś kierunku w przestrzeni trójwymiarowej. Następnie system próbuje znaleźć pierwszy punkt jej kolizji z jakimś symulowanym ciałem fizycznym, posiadającym odpowiedni kolider. Ponieważ zasoby komputera zawsze są ograniczone, symulacja półprostej także musi mieć pewien limit. Zwykle jest on jednak na tyle duży, że z punktu widzenia lokalnych wydarzeń, można uznać tą odległość za nieskończoną.

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

3.2 Różnice między czujnikiem, a modelem

Półprosta emitowana jest z punktu reprezentującego środek czujnika, naturalnie, model upraszcza rzeczywisty czujnik (budowa czujnika laserowego została opisana w sekcji 1.4). Uproszczenie polega na tym, że nie ma w środku żadnego lustra lub obracającej się części. W rzeczywistości w czujniku jest jeden laser, emitujący脉冲 w określonych odstępach czasu. W modelu można zatem przyjąć osobne półproste dla każdego pulsu lasera.

Można zauważać tym samym, że model wydaje się lepszym czujnikiem, niż rzeczywisty LiDAR. W danej chwili model emitemie promień we wszystkich kierunkach jednocześnie, podczas gdy czujnik jednym pulsem może dokonać tylko jednego pomiaru o danym w tej chwili kącie. Jednakże dyskretny sposób symulacji powoduje, że w obu przypadkach dane są podawane w grupach. Czujnik jest w stanie wysłać pakiet z danymi 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 wcale nie różnią się tak bardzo.

Drugą rzeczą, w której czujnik przoduje, jest nieskończona (z punktu widzenia symulacji) odległość pomiaru. Nie tylko jako najdalszy wykryty punkt, ale także i najbliższy. Czujnik nie obcina pomiarów przy określonej odległości, po prostu spada ich jakość, zmniejsza dokładność, zwiększa ilość błędów. Symulator ma całkowitą dowolność w ustawianiu progu, dla którego obcina pomiar.

Podobnie, jak w poprzednim przypadku, symulator zawsze ma taką samą dokładność pomiaru, niezależnie od odległości punktu od czujnika. Czujnik zmienia błędność danych w zależności jak daleko od niego jest obiekt.

W zależności od obciążenia komputera, model czujnika jest podatny na opóźnienia w odczytywaniu stanu. Czujnik zawsze działa z tą samą częstotliwością, a jego program sterujący jest wbudowany w mikrokontroler i spełnia sztywne ramy czasowe.

3.3 Komunikacja

Jednakże, bazując na architekturze opisanej wcześniej na rysunku 1.13, należy tak zbudować system, aby program komunikował się w identyczny sposób z modelem czujnika, jak i samym czujnikiem. Służą do tego specjalne pakiety ROSa, zawierające czas pomiaru, typ i dane. Program obsługujący model czujnika generuje i wysyła pakiety typu `sensor_msgs/LaserScan`.

Identycznie, inny program, podłączony do czujnika za pomocą jednego z

interfejsów, także powinien generować takie same pakiety.

3.4 Model w Gazebo

Tak, jak w modelu platformy, należy stworzyć odpowiedni dokument SDF. Aby umożliwić przenoszenie modelu czujnika na modele innych platform, powinien on być niezależny od implementacji platformy do której będzie przytwierdzony. Dodatkowo, w końcowym modelu istnieć będą dwa takie czujniki.

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 w kształcie walców, odpowiedzialne za kolizję. Odpowiada za przesunięcie samego lasera względem podstawy na której całe urządzenie jest montowane i pozwala na wygodną referencję z innego modelu w celu utworzenia połączenia.

Główna część czujnika 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.

Teoretycznie, lepiej aby model posiadał jeden element odpowiedzialny za kolizje, gdyż to byłoby szybsze do symulacji. Jednakże, półproste emitowane ze środka również się by z nim zderzały, a co za tym idzie, nie opuszczaliby modelu czujnika. Dlatego potrzeba jest, aby w miejscu ich wychodzenia (czarna szybka) nie było żadnego koldera.

3.4.1 Połączenie modeli

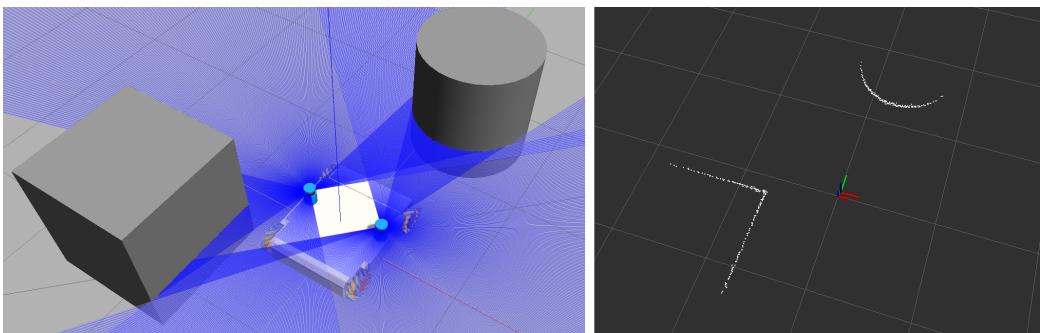
Jak wcześniej wspomniano, model SDF ma strukturę gwiaździstą. Zagnieżdżenie modeli spowodowałoby, że powstaje inna struktura, drzewiasta. Dlatego też, element `import` nie umieszcza w swoim miejscu całego modelu z innego pliku, a raczej importuje jego składowe i umieszcza równolegle do istniejących. To oznacza, że zadbać trzeba także o elementy `joint`, łączące element podstawy platformy z podstawą czujnika, inaczej symulacja widziałaby dwa osobne modele. Więc potrzebna jest także znajomość nazw elementów składowych importowanego modelu.

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 sytuacje w których zachowanie jakichś elementów będzie niezrozumiałe.

Ma to także swoją wadę. Importując dwa, identyczne modele, ich składowe tracą informację o swoich wspólnych rodzicach. Program sterujący czujnikiem nie ma sposoby sprawdzić, którym czujnikiem steruje, a co za tym idzie, jaką nazwę interfejsu wystawić do nadawania wiadomości. Jedynym sposobem na rozwiązanie tego problemu jest nadanie importowanym elementom różnych przedrostków do nazw. I także sterownik musi tylko na tej podstawie określić czujnik, w którym jest.

Alternatywnie, zawsze można stworzyć dwa, osobne modele czujników, albo wszystko umieścić od razu w jednym pliku. Jednak takie rozwiązanie niszczy komponentową budowę środowiska i nie pozwala na użycie składowych w innych modelach.



Rysunek 3.1: Zrzut ekranu platformy z Gazebo i wygenerowane dane, obserwowane w Rviz.

3.4.2 Mechanika ramek

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

Ramka transformacji jest informacją o aktualnej pozycji i rotacji jakiegoś obiektu względem innego. Polega na wysłaniu pakietu typu `geometry_msgs/TransformStamped` prosto do demona ROS. Pakiet zawiera w sobie nagłówek, informujący o czasie i identyfikatorze nadania ramki, i względem jakiego obiektu podawana jest pozycja i rotacja. Następna jest nazwa nowej ramki, jej pozycja i rotacja względem nazwy obiektu podanego w nagłówku. Dowolny program może nadać dowolny pakiet transformacji. Demon ROSa następnie zbiera wszystkie

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 3.1: Nazwy identyfikatorów ramek, używanych w symulatorze.

pozycje i oblicza transformacje dla programów, które pytają się o względne pozycje elementów.

Przykładowo, gdyby symulacja robota nie odbywały się w przestrzeni wirtualnej, w maszynie symulacyjnej fizyki, informacja o dokładnym położeniu obiektu składowego w przestrzeni wcale nie musiałaby być tak łatwo dostępna. To ma szczególne znaczenie dla skomplikowanych mechanizmów typu wielosegmentowe ramię manipulacyjne. Obliczenie pozycji i rotacji końcówki ramienia wymagałoby informacji o aktualnych pozycjach i rotacjach wszystkich innych segmentów. Która strona miałaby zajmować się obliczeniami i kto powinien posiadać i komu przekazywać te informacje?

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

W symulacji platformy wielokierunkowej mechanika ramek jest potrzebna, gdyż pakiet zwierający pomiary z czujnika laserowego nie posiada informacji o aktualnej pozycji samego czujnika, a jedynie identyfikator swojej ramki. Te informacje potrzebne są programowi obliczającemu pozycję z czujników i ewentualnemu wizualizatorowi samych czujników laserowych.

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

Nazwa	Punkt względny	Punkt danych
Pozycja i rotacja platformy	map	omnivelma
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 3.2: Ramki wysyłane do demona ROS.

3.5 Błędy

Jak podano wcześniej w tabelce 1.1, wyróżnione są dwa typy błędów pomiaru, systematyczny i pomiarowy. Dodatkowo istnieje także błąd gruby. Model czujnika powinien uwzględniać te błędy, aby zwracać dane jak najbardziej zbliżone do LiDARa.

3.5.1 Błąd gruby

Najprostszy typ błędu polega na dużych odchyłach niektórych pomiarów od pozostałych wartości. W trakcie przetwarzania odczytu, te punkty powinno się odrzucić. Nie mniej jednak, to zadanie należy do programu sterującego, więc należy umożliwić mu testowanie tej funkcjonalności poprzez wprowadzenie takich błędów do zasymulowanych odczytów.

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

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

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

3.5.2 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 czujnik powinien być skalibrowany przed użyciem właśnie po to, aby wewnętrzny program sterujący mógł obliczyć aktualne zboczenia i skorygować pomiary przed wysłaniem ich dalej. Kalibracja może być nadana na samym urządzeniu, poprzez zmianę jego parametrów działania, inaczej, zmianę współczynników przy korygowaniu pomiarów, przez system

wbudowany, przed wysłaniem. 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.

3.5.3 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 1.1, może być bezpośrednio zapisana do elementu odchylenia standardowego, w pliku SDF opisującym czujnik. Wadą takiego rozwiązania jest niemożność modyfikacji tego parametru w trakcie wykonywania programu, gdyż Gazebo nie wystawia API do modyfikacji tej wartości. Aby temu zaradzić, wystarczy obliczać błąd standardowy w programie sterującym i manualnie dodawać go do zwróconej przez symulator tablicy. Funkcje do obliczania błędu standartowego zostały wprowadzone do standardu języka C++ w 2011 roku.

Rozdział 4

Komponenty systemu

Aby uruchomić system, nie wystarczy uruchomić Gazebo z modelami, należy zadbać także o odpowiednie przekazywanie informacji pomiędzy pakietami. Należy także przetestować modele, czy zachowują się poprawnie w prostych scenariuszach testowych, tak samo, jak testować się będzie program sterujący. Do tego potrzebne są programy wspomagające, które łączy się w różne konfiguracje, w zależności od scenariusza testowego.

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

- Numer sekwencyjny, zwiększany przez program wysyłający za każdym pakietem.
- Czas nadania pakietu, z dokładnością do nanosekund.
- Identyfikator ramki, według której podano dane, ramki zostały opisane dokładniej w 3.4.2.

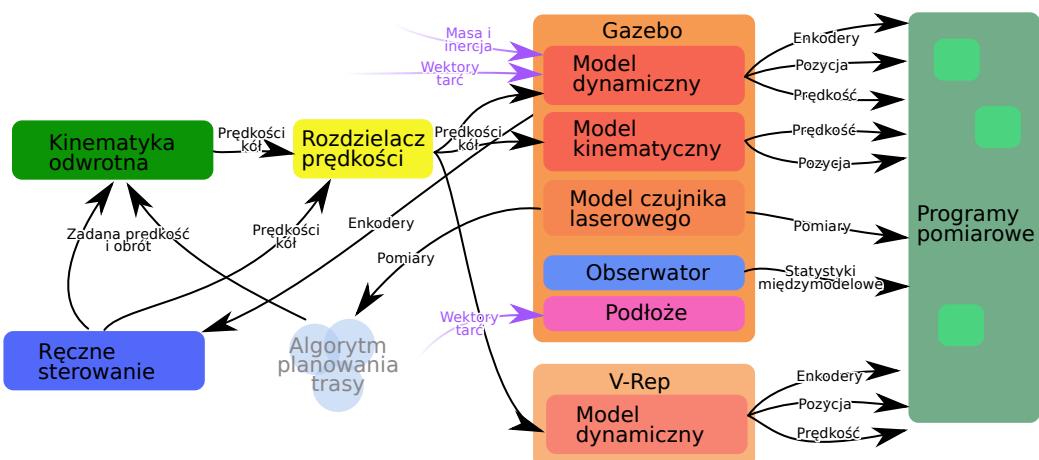
Typ	Opis
<code>omnivelma_msgs/Encoders</code>	Prędkości i pozycje kół z enkodera.
<code>omnivelma_msgs/Vels</code>	Prędkości kół.
<code>omnivelma_msgs/SetFriction</code>	Nadanie tarcia elementowi modelu.
<code>omnivelma_msgs/SetInertia</code>	Nadanie mas i momentu bezwładności obiekowi.
<code>geometry_msgs/Pose</code>	Pozycja i rotacja obiektu w przestrzeni kartezjańskiej.
<code>geometry_msgs/Twist</code>	Prędkość względna jakiegoś obiektu.
<code>sensor_msgs/LaserScan</code>	Jedno skanowanie LiDARa, wraz z nagłówkiem.
<code>omnivelma_msgs/Relative</code>	Odległość i kąt pomiędzy obiektami.

Tablica 4.1: Typy i opisy wiadomości przekazywanych pomiędzy komponentami.

Komponenty można podzielić na trzy typy:

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

Poniżej każdy pakiet opisany jest bardziej szczegółowo, ten dokument nie ma za zadanie być programistyczną dokumentacją komponentów, dlatego nie zagłębia się w dokładne składy przekazywanych struktur i ich nazwy.



Rysunek 4.1: Sekcje i komunikacja podstawowych komponentów systemu.

Komponenty modeli platform zostały opisane szczegółowo w poprzednich sekcjach.

`omnivelman` Model platformy dynamicznej w sekcji 2.3.

`pseudovelma` Model platformy kinematycznej w sekcji 2.2.

`monokl` Model czujnika laserowego w całym rozdziale 3.

4.1 Manualne sterowanie

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

4.1.1 Program

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

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

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

4.1.2 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 nadać swojej prędkości bez informacji, jakimi kołami z jaką prędkością obracać, ten typ wiadomości musi być jeszcze konwertowany przez inne komponenty na zrozumiały format.

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

wyświetlić i tylko takie potrzebuje. Dzięki temu może być użyty niezależnie od innych komponentów i programów, jednak może wymagać dodatkowego komponentu do wyłuskania tej informacji z większego pakietu, patrz 4.2.

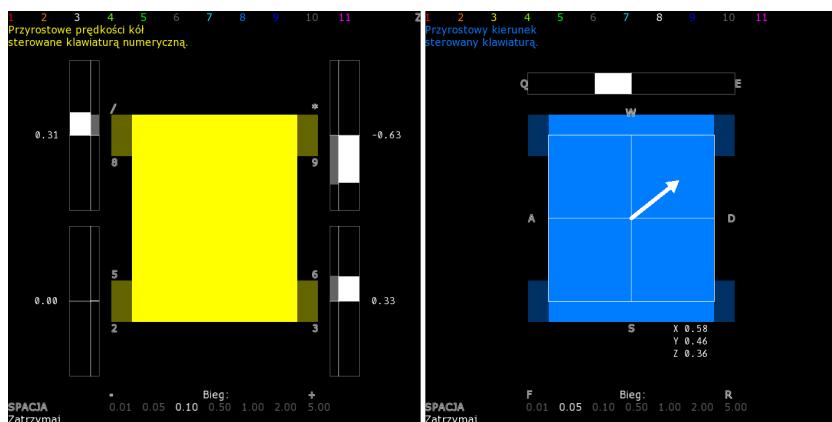
4.1.3 Tryby działania

Program posiada 11 trybów działania, w których generuje różne wiadomości w różny sposób. Jedne są bardziej przydatne, inne bardzo proste i służące do ogólnej prezentacji systemu, albo zabawy. Globalny mnożnik wyjścia pozwala na łatwe ograniczenie generowanych danych. 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 2.3.6.
3. Naciśnięcie klawisza płynnie zwiększa, lub zmniejsza prędkość koła. W trakcie braku aktywności użytkownika, platforma porusza się z ustalonymi prędkościami kół.
4. Podobnie, co poprzednio, lecz pozwala na schodkowe, dokładniejsze ustawienie prędkości kół. Dzięki temu możliwe jest w miarę dokładne powtórzenie manualnych testów platformy.
5. Poprzedni tryb, lecz przy ustawieniu prędkości zerowej, generuje nie-liczby.
6. Sterowanie prędkościami kół za pomocą gałek kontrolera. Większość kontrolerów posiada dwa, dwuosiowe joysticki, co daje cztery osie zmieniające się w zakresie $(-1; 1)$. Można za ich pomocą bezpośrednio ustawiać prędkości kół, chociaż jest to dość 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 z zadaną prędkością w zadanym kierunku i z zadanym obrotem.
9. Połączenie dwóch poprzednich trybów, schodkowe sterowanie prędkością platformy. Pozwala na ustawienie prędkości i kierunku 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 Meccanum.
11. Sterowanie za pomocą myszki, najdokładniejsze sterowanie kierunkiem, mniej dokładne obrotem. Kursor myszy wskazuje końcówkę strzałki reprezentującej kierunek ruchu robota, za pomocą kółka, można dodawać, lub odejmować prędkość do obrotu wokół osi. Ponieważ większość myszek ma skokowe obroty kółek, wprowadza to nieznaczne poślizgi. Można także modyfikować obrót płynnie klawiaturą.



Rysunek 4.2: Zrzuty ekranu dwóch trybów działania programu.

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

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

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

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

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

4.2 Wyłuskanie struktury wiadomości

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

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

Takie zjawisko występuje przy przekazywaniu informacji o pozycji i prędkości kół, generowanej przez model czujnika enkoderów, do programu manualnego sterowania, opisanego w 4.1. Program jest w stanie wygodnie wyświetlić generowane prędkości kół obok zebranych wartości przez enkoder, w celu wizualnego ich porównania.

ROS nie pozwala na automatyczne odbieranie tylko części pakietu, dla tego powstał program o nazwie kodowej `dziadzio`, niczym dziadek do orzechów. Przyjmuje wiadomości typu `omnivelo_msgs/EncodersStamped` i zwraca wiadomości typu `omnivelo_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.

4.3 Podłoga o zmiennym współczynniku tarcia

Symulacja nie składa się jedynie z robota i czujnika, ale także z podłożą na którym musi się poruszać. Ponieważ podłożo również wpływa na symulację, powinien istnieć sposób na ustawienie jego współczynnika tarcia. Robi się to wewnętrznie w identyczny sposób, jak w przypadku kół platformy, co zostało opisane w sekcji 2.3.4.

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

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

4.4 Obserwator symulacji

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

Ten program nie może być zaimplementowany jako zewnętrzny komponent, gdyż wiadomości zawierające pozycje platform będą przychodzić asynchronicznie. Nie da się w takim przypadku obliczyć dokładnych odległości pomiędzy platformami w danej chwili. Wprowadziłoby to także spore opóźnienie, gdyż program musiałby czekać na odbiór obu pakietów, dodatkowo zachować pewność że oba pochodzą z tej samej klatki symulacji.

Nazwa kodowa tego programu to `ocznica`.

4.5 Model kinematyki odwrotnej

Jest to coś na kształt odwrotności modelu kinematycznego, opisanego w sekcji ??, jednak działającego bez symulatora. Ten komponent przyjmuje zadaną prędkość, kierunek i obrót platformy, a zwraca prędkości kół, które powinny być nadane platformie, aby wywołać taki ruch. Nie całkuje tego ruchu i nie różniczuje go, więc nie generuje informacji o aktualnej pozycji modelu platformy.

Dodatkowo, program pozwala na obrót wektora prędkości o zadany kąt prosty, lub półpełny. Jest to spowodowane tym, że różne komponenty i różne modele przyjmują różną pozycję wyjściową robota. Czasami przód skierowany jest w dodatnią stronę osi X, a czasami Y. W związku z tym ta funkcjonalność jest w stanie przekonwertować dane wejściowe dla innego robota.

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

4.6 Mapa z symulacją

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

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 4.4.

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

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

4.7 Rozdzielacz pakietów

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

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.

4.8 Prosty program sterujący

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

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

Działanie programu oparte jest na zachowaniu akcja-reakcja. Dane z czujników laserowych dzielone są, w zależności od kąta pomiaru, na cztery ćwiartki lokalnego układu współrzędnych. Rozpatrywane są tylko te ćwiartki w których kierunku porusza się platforma. Jeśli pomiar wypadnie wystarczająco blisko platformy, kierunek jest obracany w odwrotnym do tej ćwiartki kierunku.

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

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

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

4.9 Struktury pakietów wiadomości

Ten komponent nie jest plikiem wykonywalnym, a definicjami struktur danych, używanych przez wiadomości ROSa, używane w projekcie, jeśli nie zostały zdefiniowane w standardzie. Nazwa kodowa i jednocześnie przedrostek wszystkich zawartych typów wiadomości to `omnivelman_msgs`.

4.10 Zewnętrzne pakiety

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

4.10.1 Rysownik wykresów

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

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

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

4.10.2 Wizualizer pomiarów

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

Pozwala to na przykład umieścić znacznik platformy i chmury punktów odebranych z czujników laserowych. Jest lżejszy na zasobach w działaniu niż Gazebo i pokazuje tylko informacje z odebranych danych, a nie całe środowisko symulacji.

Bibliografia

- [1] P. Muir, C. Neuman “Kinematic modeling for feedback control of an omni-directional wheeled mobile robot”, Proceedings, IEEE International Conference in Robotics and Automation, Vol 4. pp. 1772-1778, 1987.
- [2] Mihai Olimpiu Tătar, Cătălin Popovici, Dan Mândru, Ioan Ardelean, Alin Pleșa “Design and development of an autonomous omni-directional mobile robot with Mecanum wheels”, Conference: 2014 IEEE International Conference on Automation, Quality and Testing, Robotics (AQTR)
- [3] Matthieu LAMY “Mechanical development of an automated guided vehicle”, Master of Science Thesis MMK 2016:153 MKN 171, KTH Industrial Engineering and Management, Machine Design, SE-100 44 Sztokholm
- [4] Anton Gfrerrer “Geometry and kinematics of the Mecanum wheel”, Graz University of Technology, Institute of Geometry, Kopernikusgasse 24, 8010 Graz, Austria, Computer Aided Geometric Design 25(9):784-791
- [5] Li Xie, Christian Scheifele, Weiliang Xu, Karl A. Stol “Heavy-Duty Omni-Directional Mecanum-Wheeled Robot for Autonomous Navigation”, DOI: 10.1109/ICMECH.2015.7083984
- [6] Viktor Kálmán “On modeling and control of omnidirectional wheels”, PhD. dissertation, Budapest University of Technology and Economics, Department of Control Engineering and Information Technology
- [7] Jae-Bok Song, Kyung-Seok Byun “Design and Control of a Four-Wheeled Omnidirectional Mobile Robot with Steerable Omnidirectional Wheels”, Journal of Robotic Systems 21(4):193-208, Kwiecień 2004
- [8] Ioan Doroftei, Victor Grosu, Veaceslav Spinu “Omnidirectional Mobile Robot – Design and Implementation”, Bioinspiration and Robotics Walking and Climbing Robots, Maki K. Habib (Ed.), ISBN: 978-3-902613-15-8, InTech, 2007

- [9] Strona internetowa symulatora Gazebo.
<http://gazebosim.org>
- [10] Strona internetowa symulatora V-Rep.
<http://coppeliarobotics.com>
- [11] Strona internetowa platformy programistycznej *Robot Operating System*.
<http://www.ros.org>
- [12] Strona internetowa standardu SDF.
<http://sdformat.org/spec>
- [13] Strona internetowa producenta czujników laserowych SICK.
<https://www.sick.com/de/en/detection-and-ranging-solutions/2d-lidar-sensors/lms1xx/lms100-10000/p/p109841>
- [14] Dokumentacja maszyny ODE z wyjaśnieniem działania kolizji.
http://ode-wiki.org/wiki/index.php?title=Manual:_Joint_Types_and_Functions#Contact