



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, 9 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	9
1.4 Czujnik laserowy	11
1.4.1 Zasada działania	11
1.4.2 Komunikacja	12
1.4.3 Podstawowe cechy	12
1.5 Składniki systemu	13
1.5.1 Model 3D	14
1.5.2 Sterownik silników	15
1.5.3 Sterownik czujników	16
1.5.4 Program sterujący	16
1.6 Technologie	17
1.6.1 Gazebo	17
1.6.2 V-Rep	18
1.6.3 ROS	19
1.6.4 Narzędzia	20
1.7 Plan pracy	21
1.8 Istniejące implementacje	22
2 Model platformy	24
2.1 Sposób zapisu w formacie SDF	25
2.2 Model kinematyczny	26
2.2.1 Problemy implementacji	26
2.2.2 Komunikacja	27
2.2.3 Zachowanie	27
2.3 Model dynamiczny	27
2.3.1 Jak największe zbliżenie do oryginału	28
2.3.2 Resetowanie pozycji koła	29

2.3.3	Zmiana osi rolki	30
2.3.4	Modyfikacja kierunków i wartości wektorów tarcia	30
2.3.5	Komunikacja	33
2.3.6	Rozszerzenie modelu	33
2.4	Porównanie modeli	34
3	Model czujnika laserowego	36
3.1	Obliczenia symulatora	36
3.2	Różnice między czujnikiem, a modelem	37
3.3	Komunikacja	37
3.4	Model w Gazebo	38
3.4.1	Połączenie modeli	38
3.4.2	Mechanika ramek	39
3.5	Błędy	40
3.5.1	Błąd gruby	40
3.5.2	Błąd systematyczny	41
3.5.3	Błąd pomiarowy	41

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.

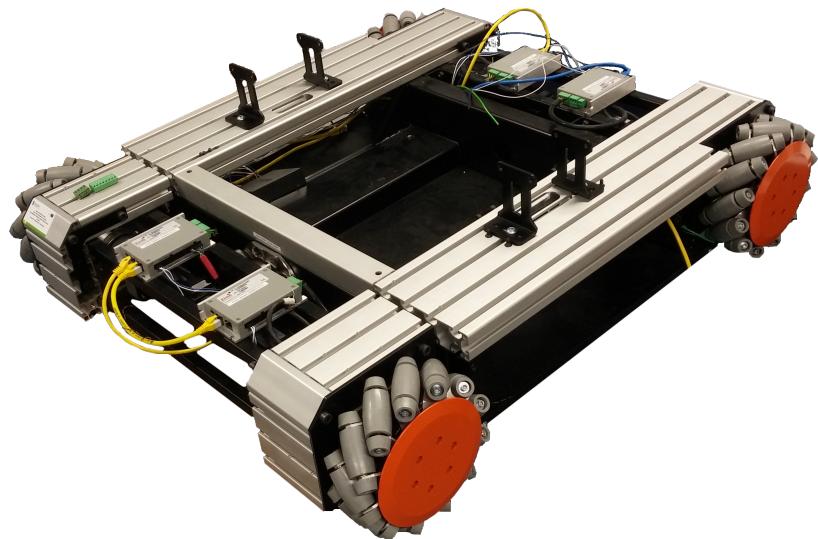
Celem jest stworzenie modelu, który będzie reagował na siły podobnie do rzeczywistego robota i był sterowany tak samo, jak rzeczywisty robot. To spowoduje, że możliwe będzie stworzenie jednego wspólnego programu sterującego do użycia zarówno w symulacji, jak i rzeczywistym robocie.

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.

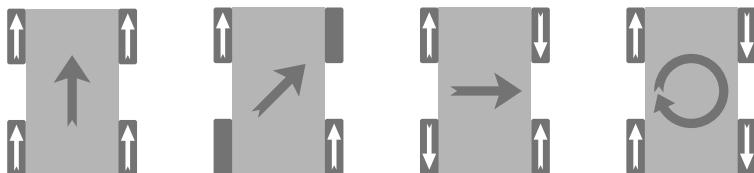
Jest to duża, prostokątna baza dookólna poruszająca się na czterech kołach szwedzkich, patrz fotografia 1.1. Koła są stałe, parami przytwierdzone do dwóch osi. Każde koło jest sterowane osobno przez podłączony bezpośrednio serwomotor, zatem może mieć prędkość i kierunek niezależny od pozostałych kół,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

Jest to najpopularniejsza budowa dookólnych platform mobilnych, mająca zastosowanie także w innych robotach, jak na przykład Kuka Youbot 1.2. Pomimo, że robot o trzech kołach szwedzkich i prostszej budowie ma taką samą ilość stopni swobody, to jego stabilność jest gorsza od czterokołowych wersji [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 niezależnie od kąta obrotu robota, patrz rysunek 1.3. Za ich pomocą da się także obracać bazę stojąc w miejscu, lub w trakcie ruchu po prostej. Na przykład, jeśli obracać tylko przeciwnymi kołami po przekątnej, system zacznie się poruszać po skosie bez zmiany kąta obrotu. A jeśli do tego dodamy 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 obiekt przy stałej prędkości kół zawsze jest okręgiem, możemy uznać prostą za okrąg o nieskończonym promieniu, a punkt za okrąg o zerowym. Wynika to z tego, że każdy obiekt, który ma jednostajną prędkość o kierunku w lokalnym układzie współrzędnych, oraz prędkość kątową będzie się poruszał po takiej krzywej.



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

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 dużą równowagę. 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.



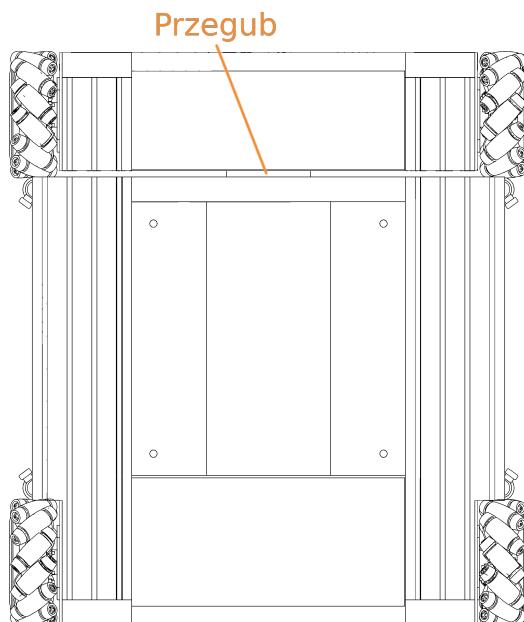
Rysunek 1.4: Robot manipulacyjny Velma.

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 podłoża na ruch bazy, aby każde koło dociskało do podłoża z taką samą siłą, jak po drugiej stronie osi. Bez tego zawiasu nierówny teren uniemożliwiałby sprawne sterowanie platformą na skutek niedeterministycznego tarcia kół tej samej osi, powodując nieplanowany skręt. Niedeterministyczne tarcie kół jest niewykrywalne w bezpośredni sposób, więc należy je wyeliminować na przykład za pomocą takiego przegubu.

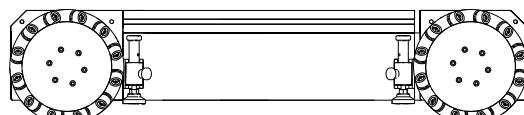
Środkowe kółko nie są rozmieszczone na wierzchołkach kwadratu. Jest 4 cm różnicy między szerokością, a długością. Szerokość jest większa, co można

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, a w związku z tym różne prędkości, co w efekcie może powodować niedeterministyczny ruch. Należy także wziąć tutaj pod uwagę nieprzewidywalne opory, 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.



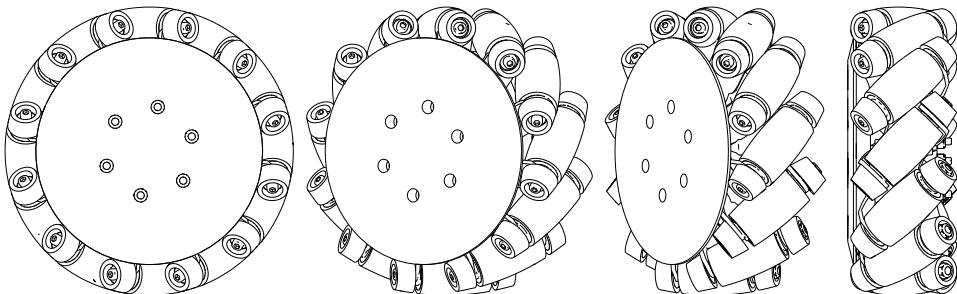
Rysunek 1.7: Platforma mobilna — widok z tyłu.

Platforma posiada 3 stopnie swobody. Pierwszym i drugim jest ruch bez obrotu w osiach X i Y. Trzecim stopniem jest dowolny obrót po 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. Ich osie ustawione są w ten sposób, że osie rolek dwóch kół z tej samej strony robota przecinają się pod kątem prostym. Innymi słowy, robot ma identycznie ustawione koła na przeciwnie wierzchołkach, i razem ustawione są w kształt litery X patrząc na nie z góry. Warto pamiętać, że 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. Pierwszym jest obrót całego koła wzdłuż osi. Drugim są rotacje pojedynczych rolek, a trzecim poślizg obrotowy w miejscu styku rolki z podłożem [1].

Na podstawie rysunku 1.8 widać, że krzywizna rolki jest tak ustawiona, aby punkt kontaktu rolki z podłożem płynnie przechodził na następną rolkę w trakcie obrotu. 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. Pojedyncza rolka zawiera się w paraboloidzie, wzory

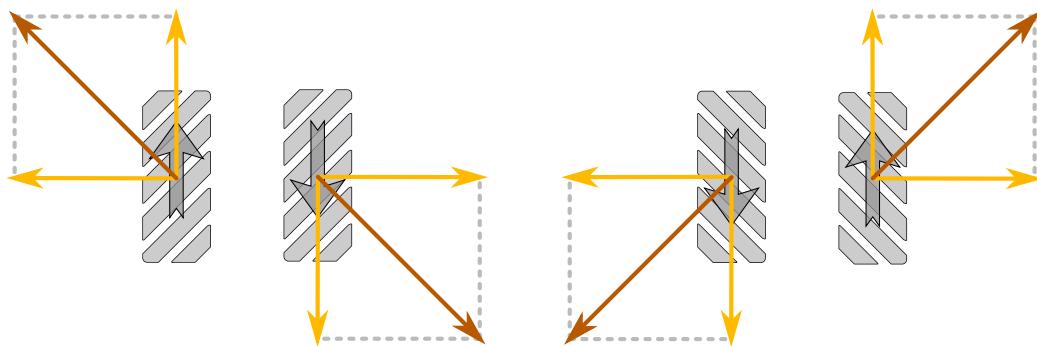
opisujące kształt rolki są złożone. Zazwyczaj przybliża się ją torusem w celu uproszczenia produkcji [4].

Istnieją także inne budowy kół, złożone z wielu małych rolek tak, aby w każdym momencie kilka rolek dotykało podłoża. Można także złożyć kilka powyższych kół obok siebie w jednym kole. 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, z jest jedynym popularnie używanym [3].

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

1.3.1 Działanie

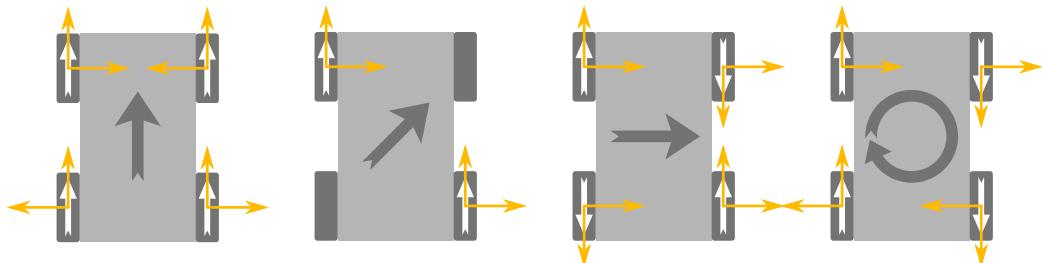
Standardowe koło, używając tarcia, przekształca prędkość kątową na liniową w płaszczyźnie obrotu. Specjalne koło Mecanum ma dodatkowy wektor równoległy do osi obrotu, zatem prędkość wypadkowa jest obrócona o 45° , zależnie od typu koła (prawoskrętne lub lewoskrętne) i kierunku obrotu.



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

Ustawiając te koła w odpowiedni, opisany wcześniej na obrazku 1.5, sposób można zmusić wektory kół do odpowiedniego znoszenia się składowych, a w efekcie pozwolić robotowi na poruszanie się w kierunkach nieosiągalnych dla pojazdów o standardowych kołach. A zatem, można nałożyć te składowe

na wcześniejszy rysunek 1.3 i dokładnie się przyjrzeć, dlaczego koła popchają przy danym obrocie platformę w danym kierunku.



Rysunek 1.10: Wektory składowe i wypadkowe piasty koła widzianego z góry.

W pierwszym przypadku, wektory o kierunku prostopadłym do 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 w przód.

Drugi przypadek jest podobny, lecz dwa koła nie obracają się. Nie jest to ruch pasywny, gdyż taki wprowadzałby nieprzewidywane poślizgi, a aktywne hamowanie. Wtedy wektory nie mają się jak znosić i platforma wykonuje ruch pod kątem 45° do płaszczyzny symetrii.

Zasada ruchu w bok jest bardzo podobna do ruchu w przód. Tutaj również wektory znoszą się parami, jednak tym razem na prawych kołach i lewych.

Alternatywnie, bardzo łatwo nadać prędkość kątową podstawie przy odpowiednim ustaleniu prędkości kół.

Warto nadmienić, że w idealnym przypadku rolki nie obracają się, gdy wypadkowy wektor prędkości koła jest prostopadły do osi koła. 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. I na przykład, przy ruchu w przód rolki koła praktycznie się nie obracają, lecz przy ruchu w bok biorą aktywny udział. Może mieć to wpływ na zużywanie się rolek, nie tylko z punktu widzenia ilości obrotów danej rolki na pokonanym dystansie, ale także sposobu w jaki wymuszany jest jej ruch. Rolki zawsze będą nieznacznie się obracać nieco szarpanym ruchem w obie strony, ze względu na poślizgi od innych kół, niejednostajne tarcie piast wszystkich innych rolek, czy różnice terenu. Zatem przejazd przykładowego odcinka przodem lub bokiem, będzie w różnym stopniu i w różny sposób zużywał elementy wykonawcze. To, jaki styl jazdy opłaca się zastosować, aby zminimalizować zużywanie się elementów jest dużą, odrębną dziedziną nauki.

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. Pośrodku urządzenia znajduje się obrotowe lusterko, zwrócone pod kątem 45° do osi obrotu. Równolegle do osi znajduje się laser, który emitem pulsacyjną wiązkę podczerwonego promienia co pewien okres. 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 zaawansowany algorytm we wbudowanym mikrokontrolerze ustala kąt i odległość wykrytego obiektu. Odpowiada także za usunięcie szumu i ewentualnych odbić promienia. Komunikacja odbywa się za pomocą różnych interfejsów sieciowych, zazwyczaj w architekturze master-slave.

1.4.2 Komunikacja

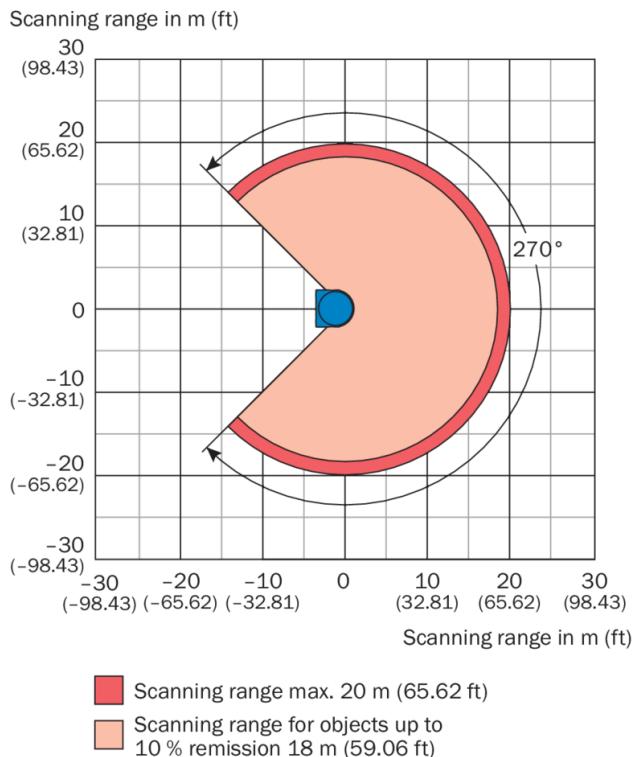
Wysyłając do czujnika odpowiedni ciąg bajtów, można ustawić jego tryb działania, odpytać o zebrane dane, czy wykryć konfigurację i stan.

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 dane opisujące sam pomiar takie jak czas, początkowy kąt pomiaru, tryb itp. Dokładne dane i cechy czujnika dostępne są na stronie producenta [12].

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. To powoduje, że nie może mierzyć w kącie pełnym i powstaje martwa strefa. Przedstawia to dobrze grafika producenta.



Rysunek 1.12: Wykres producenta dotyczący zasięgu czujnika.

Urządzenie jest w stanie komunikować się za pomocą portu szeregowego

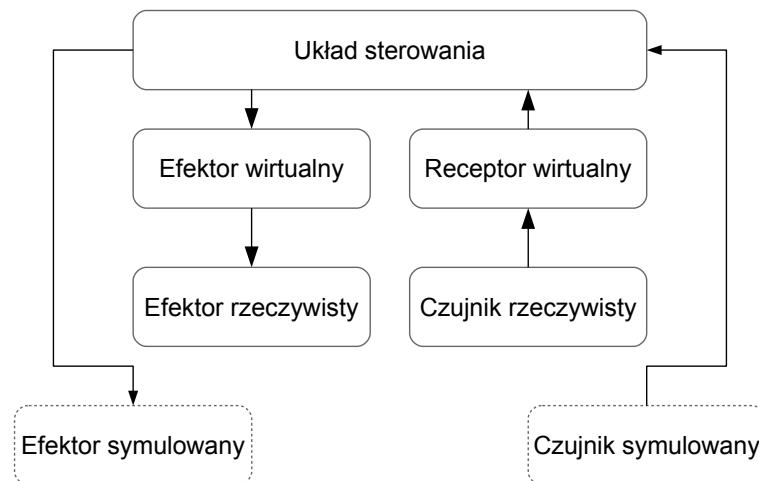
Cecha	Wartość
Kąt pracy	270°
Długość fali światła lasera	905 nm (podczerwień)
Częstotliwość skanowania	25 Hz / 50 Hz
Maksymalna odległość obiektu	≈ 20 m
Rozdzielcość kątowa	0,25° / 0,5°
Systematyczny błąd pomiarowy	±0,03 m
Przypadkowy błąd odległości	0,012 m

Tablica 1.1: Podstawowe cechy czujnika laserowego.

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 tę samą komunikację między składnikami nie tracąc kompatybilności między sobą. 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, jak na rysunku 1.13. Agent upostaciowany składa się wilku modułów komunikujących się ze sobą za pomocą różnych systemów.

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.

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

Zamodelowany efektor symulowany również przyjmuje te same sygnały do układu sterowania, lecz nie zamienia ich na sygnały sterujące, a wywołuje odpowiednie funkcje maszyny symulacyjnej nadające siły i prędkości obiektem 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 kamera Kinect, w której to zachodzi odczytanie obrazu z kilku kamer. Następnie obraz przesyłany jest do komputera w którym sterowniki interpretują dane usuwając błędy, tworzą mapę głębokości, wykrywają szkielety i sylwetki osób. Te dane mogą być wykorzystane łatwo w grach i programach sterujących.

Modelowanie receptora, tak jak w przypadku efektora polega na wygenerowaniu odpowiednich danych używając odpowiednich funkcji w przestrzeni wirtualnej. Mogą one polegać na puszczeniu promieni 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.

1.5.1 Model 3D

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

Model składa się z elementów odwzorowujących rzeczywiste części składowe bazy mobilnej. Elementy posiadają takie cechy, jak pozycja w modelu,

masa, moment bezwładności, kształt, materiał fizyczny i wygląd. Dodatkowo należy uwzględnić więzy tych składowych w postaci symulowanych przegubów. W przypadku tej bazy istnieje jeden typ więzów o jednym stopniu swobody (zawias). 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 podanym obiektom w podobny sposób, jak ma to miejsce w rzeczywistości.

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

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

1.5.2 Sterownik silników

Program sterujący generuje abstrakcyjne dane, na przykład liczbę zapisaną binarnie. Przykładowy silnik fizyczny nie jest w stanie działać na ich podstawie, on potrzebuje odpowiedniego napięcia na wejściu. Do tłumaczenia jednych danych na drugie potrzebny jest sterownik niskopoziomowy. Najczęściej implementowany jest w formie mikrokontrolera, lub podobnego systemu wbudowanego.

Jego zadanie to odczytanie danych podanych przez program sterujący i na przykład generowanie na ich podstawie 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 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 przez producenta tak samo, jak sterowniki efektorów.

Symulując ten element budujemy program generujący dane na podstawie aktualnego stanu maszyny do symulacji w sposób, w jaki działa czujnik w rzeczywistości. Na przykład dla czujnika laserowego wypuszczamy setki promieni i obliczamy ich punkty przecięcia się z wirtualnymi modelami. Możemy renderować obraz, aby symulować kamerę.

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

1.5.4 Program sterujący

Cześć odpowiedzialna za logikę aplikacji. Tutaj obliczane jest sterowanie na podstawie dostarczonych odczytów z czujników. Zazwyczaj wykorzystuje się tu dużą ilość bibliotek dostarczających 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, niż sterowniki ze względu na duże zapotrzebowania na moc obliczeniową. Jeśli robot komunikuje się z użytkownikiem, lub zwraca dane, 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 działa na rzeczywistym robocie, czy symulacji wirtualnej, zawsze powinien móc komunikować się i otrzymywać dane w tym samym formacie. W idealnym świecie program nie powinien mieć możliwości stwierdzić, czy steruje symulacją, czy oryginałem.

1.6 Technologie

Symulator daje użytkownikowi do dyspozycji odpowiednią maszynę symulacyjną odpowiedzialną 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, oraz wszystko to, co potrzebne do jak najwierszszego odtworzenia zachowania rzeczywistego obiektu.

Na rynku jest wiele różnych maszyn zarówno do symulacji w czasie rzeczywistym, jak i do wyznaczania pozycji obiektów po długich obliczeniach. Jedne z technologii są otwartoźródłowe, inne nie. Mogą używać tylko procesora, lub też być wspomagane przez kartę graficzną. Niektóre prócz zderzeń obiektów potrafią także symulować rozpływ cieczy, dymy, płotna, ciała sprężyste i strukturę wewnętrzną obiektów, lecz te funkcjonalności nie są potrzebne dla naszej symulacji.

1.6.1 Gazebo

Program do pobrania z [8]. Ten symulator graficzny jest dość prosty w obsłudze, skupia się na symulowaniu podanych danych, a mniej na możliwości ich łatwego przygotowania. Zazwyczaj używany w trybie wsadowym, uruchamiany z argumentami z linii poleceń i plikiem *world* opisującym symulację. Plik ten zawiera nazwy i ścieżki innych umieszczanych 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, jak Blender.

Symulator oprócz tego ma wbudowany edytor modeli w którym możemy składać odpowiednie obiekty od razu w przestrzeni trójwymiarowej. Edytor budynków pozwala na stawianie wirtualnych ścian, korytarzy, drzwi i ogólnego otoczenia w którym roboty mogą pracować i być symulowane. Jakość wykonania tych składników pozostawia wiele do życzenia, brak jest tak podstawowych funkcji, jak cofanie ruchu. Dlatego lepiej jest definiować model we

wczytywanym pliku tekstowym. Również tworząc modele poza edytorem w ten sposób mamy nad nimi pełną kontrolę, a parametry można ustawiać z dowolną dokładnością.

Gazebo przyjmuje modele w specjalnym formacie SDF. Jest to ustandaryzowany, zdefiniowany zewnętrznie format do opisywania składników robotów i czujników. Dzięki temu taki plik może być użyty także gdzie indziej, pod warunkiem przestrzegania standardu. Składnia jest zwykłym plikiem XML, co znaczy, że może być on tworzony na każdym edytorze tekstowym.

Wtyczka do sterowania modelem jest skompilowaną biblioteką dołączaną na starcie programu. Tworzy się ją w C++ jako klasę dziedziczącą po abstrakcyjnej klasie dostarczonej przez Gazebo. Dzięki temu może się komunikować z innymi systemami poprzez dowolne mechanizmy, nawet systemowe, jak gniazda, czy pamięć współdzielona. Jednak Gazebo dostarcza także swój własny mechanizm kolejek wiadomości, który sprawdza się w jednolitej komunikacji z innymi programami.

Program jest wspierany na systemie Ubuntu ale bez problemu można go także skompilować pod inne systemy. Interfejs jest dopracowany i przestrzega wielu ustawień systemowych, jak DPI. Uruchamianie jest proste i nie wymaga dodatkowych ustawień, uruchamiania skryptów inicjalizujących, tworzenia odpowiednich katalogów, czy definiowania zmiennych systemowych. Standardowo, jak inne programy tworzy ukryty katalog w katalogu domowym użytkownika, gdzie znajdują się wszystkie modele i logi. Czasami trzeba używać tego katalogu, aby umieścić tam swoje modele, które Gazebo będzie automatycznie umieszczał w symulacji.

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 wersjami ROSa, nie można użyć przestarzałej wersji Gazebo z nowszym ROSEm i odwrotnie. Symulator można zainstalować osobno, lub jako jeden z pakietów ROSa.

1.6.2 V-Rep

Program do pobrania z [9]. Duże i skomplikowane środowisko reklamujące się wieloma zaawansowanymi mechanizmami i funkcjami. Pomimo otwartego kodu, użycie komercyjne jest płatne. Dla zastosowań akademickich program jest rozdawany bez opłat. Bogaty interfejs graficzny zakłada budowę i symulację wszystkiego w tym jednym programie.

Używa dwóch z maszyn symulacyjnych, 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 oglądanie modelu bez posiadania całego programu i importowania modelu do symulacji. Brak przenośności, czy wsparcia systemu kontroli wersji dla takich zbiorów bajtów także jest problemem.

Pisanie wtyczek najczęściej odbywa się w C. Są też jednak dostępne inne języki skryptowe, jak Lua, Matlab, Java itp. Komunikacja z innymi programami odbywa się poprzez specjalne dodatki do środowiska. API pozwala nam stworzyć mały, wbudowany interfejs graficzny do sterowania symulacją poprzez przyciski i suwaki.

Ze strony producenta pobieramy 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.

1.6.3 ROS

Platforma programistyczna do pobrania z [10]. ROS jest skrótem od *Robot Operating System*, lecz jego nazwa jest bardzo myląca. Nie jest to żaden 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 itp.

Twórcy zachęcają, aby uzupełniać brakujące moduły swoimi własnymi, a potem dzielić się nimi z resztą programistów, aby każdy mógł skorzystać, jeśli ma podobny problem.

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. Nie jest to symulator, gdyż sam nie generuje żadnych danych, a jedynie prezentuje gotowe.

Działanie systemu jest oparte o pakiety. Każdy pakiet jest katalogiem zawierającym w sobie pliki opisujące jego parametry i skrypty używane w komplikacji. Pakietem może być wszystko, od modelu do programu pomocniczego. Pakiety mogą być zależne od siebie, ale nigdy nie wskazują nawzajem swoich bezpośrednich ścieżek.

Na przykład, jeden pakiet wymaga pliku nagłówkowego generowanego przez komplikację drugiego pakietu, ale załącza go w kodzie tak, jakby był

systemowy. Globalny skrypt komplikacji ROSa dba o odpowiednie podawanie ścieżek, kolejność komplikacji programów i załączanie nazw.

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.

Instalacja programu na systemie operacyjnym jest dużym problemem. Z wyjątkiem odpowiednich wersji Ubuntu nie ma łatwego sposobu na wgranie go do innych systemów. Na przeszkodzie stoją błędy komplikacji dla nowszych wersji kompilatorów i inne problemy w czasie wykonywania, jak naruszenie ochrony pamięci. Instalacja alternatywnych pakietów i ręczna komplikacja niektórych części nie działa we wszystkich przypadkach. Głównym problemem jest niezgodność wersji zewnętrznych bibliotek.

Rozwiązaniem tego problemu jest instalacja tej platformy programistycznej na maszynie wirtualnej, lub na systemie uruchamianym z dysku zewnętrznego. Takie rozwiązanie także daje dostęp do najnowszej wersji długiego wsparcia ROS *Kinetic Kame* z końca maja 2016 roku.

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 całościowych plików. Użycie niektórych funkcji ROS wymaga uruchomionego demona serwera w tle.

Ogólnie instalacja i używanie ROS na systemie zostawia dużo różnorodnych plików w katalogu domowym, dlatego lepiej jest trzymać ją z dala od codziennego systemu operacyjnego, na maszynie wirtualnej, lub dysku. Z drugiej jednak strony wirtualizacja systemu operacyjnego z ROS bardzo ogranicza dostępną moc obliczeniową potrzebną takim programom w dużych ilościach.

1.6.4 Narzędzia

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

Gazebo będzie użyty do symulacji z jego domyślną maszyną symulacji fizyki

ODE.

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

Atom jest popularnym uniwersalnym edytorem tekstowym. Dzięki rozszerzaniu przez wtyczki dobrze się sprawdza przy obróbce plików XML i skryptów. Będzie użyty do konstrukcji modelu.

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 kompilowanego 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 szybki sposób. Uniwersalne narzędzie pomagające w różnych miejscach.

Git jest narzędziem kontroli wersji, które jest wskazane w każdym projekcie informatycznym. Kod należy dla bezpieczeństwa umieszczać także 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 silą na elementy i patrząc, czy reagują w spodziewany sposób.

3. Zapisanie wtyczki sterującej w Gazebo odczytującej odpowiednie dane z zewnętrz i wywołującą funkcje maszyny symulacyjnej, aby modyfikować ruch modelu. Na tym poziomie można dobudować zamiennik programu sterującego jedynie do podawania prostych wartości bez odczytywania pomiarów i sterowania.
4. 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 podłoż ze zmiennym współczynnikiem tarcia.
7. Programy pomocnicze zbierające i wyświetlające dane, interfejs graficzny.
8. Program sterujący w ROS. Największy i najbardziej skomplikowany element, na szczęście wspólny dla obu bytów — wirtualnego i rzeczywistego. Zazwyczaj 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 już wcześniejsze modele jeżdżących robotów na kołach szwedzkich. Można z nich brać przykład i sugerować się źródłami kodu i modeli.

Kuka Youbot jest popularnym robotem wielokierunkowym. Jego modele są domyślnie dostępne zarówno w Gazebo, jak i w V-Rep. Tylko w przypadku V-Rep 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 naszego modelu, czy nie zachowuje się nadzwyczaj dziwnie w pierwszych fazach projektu.

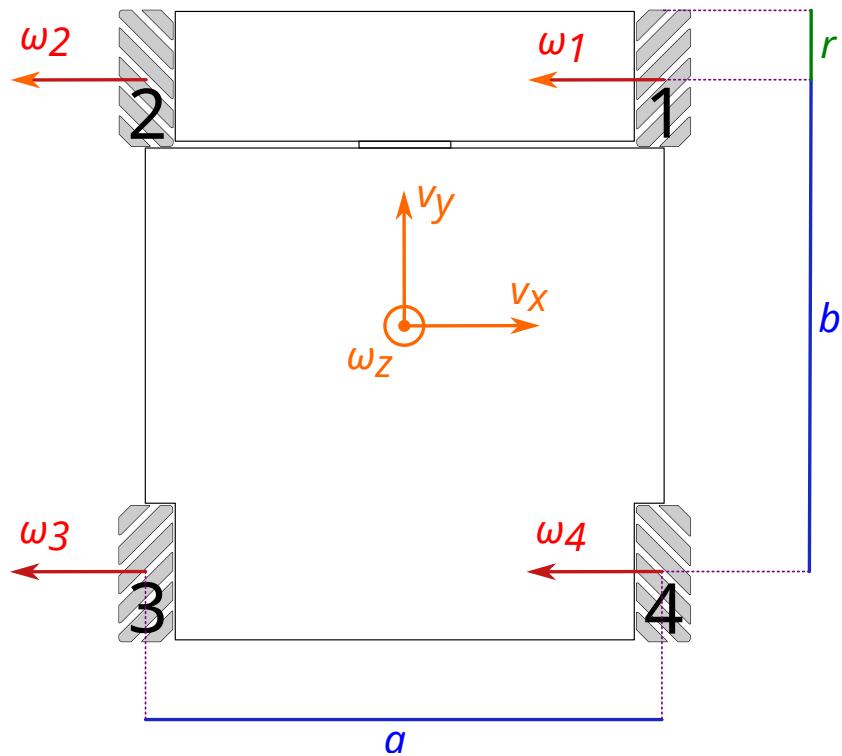
Ze względu na niezwykle zaawansowany obiekt kół i kształt rolek, prawdopodobnie trzeba będzie uprościć model poprzez zamianę niektórych składowych i dodanie sztucznych więzów. Całociągowy model może być zbyt skomplikowany, aby maszyny symulacji mogły go obliczać w czasie rzeczywistym. Taki model także jest znacznie trudniej poprawnie wymodelować ze względu na liczne tarcia i poslizgi 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 [11].

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 obiektami 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. 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 obiektami, 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 `omnivelo_msgs/Vels`. Zawiera cztery wartości zmienoprzecinkowe 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 `omnivelo_msgs/Vels` z zadanymi prędkościami kół.

2.2.3 Zachowanie

Platforma ignoruje kompletne otoczenie poruszając się przez inne obiekty. Po nadaniu stałych prędkości kół następuje ruch po okręgach 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 `omnivelo`, 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. Powodowałby 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 jedną z podstawowych brył, 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 naturalnie podskakiwałby przy ruchu zwiększąc 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 pozwalającą 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 kulę 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.

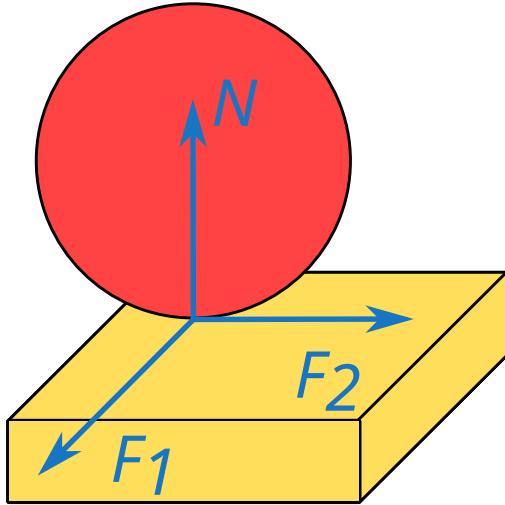
Implementacja tego rozwiązania niestety nie stworzyła działającego modelu. Wykonywanie kodu w różnych momentach symulacji nie wpływało na efekt. Platforma poprawnie poruszała się do przodu i do tyłu. Pierwszy problem pojawiał się gdy w czasie ruchu na bok, prędkość malała, aby zmienić zwrot pomimo niezmiennej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



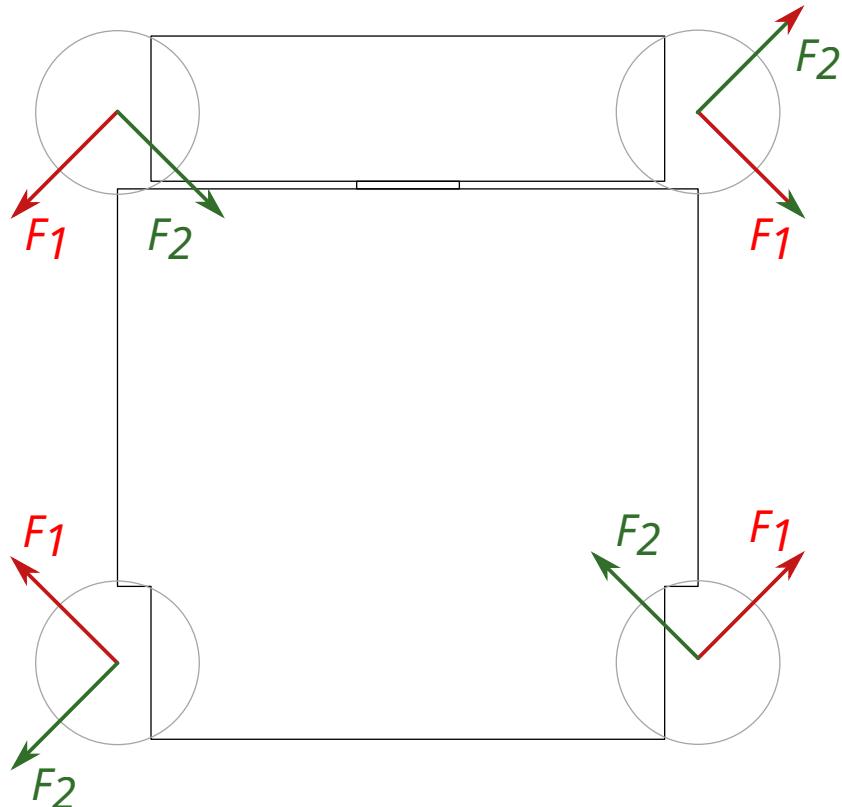
Rysunek 2.2: Wektory punktu kolizji.

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 siłę 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 prostopadle do kierunku jazdy, należy dynamicznie zmieniać współczynnik tarcia dla wektora F_1 , lub F_2 w tym kierunku. Współczynnik tarcia prostopadły 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 [13].

W naszym modelu modyfikujemy kierunek wektora F_1 , oraz współczynniki tarcia w obu kierunkach, aby przybliżyć zachowanie się rolki. Ponieważ wektory F_1 i F_2 są określone w globalnym układzie współrzędnych, w każdej iteracji maszyny symulacji należy obrócić je względem aktualnej pozycji bazy. W doskonałym przypadku rolka obraca się całkowicie bez tarcia, a ruch wzdłuż jej osi jest niemożliwy. Można więc ustawić zerowy współczynnik tar-

cia w kierunku prostopadłym do osi, oraz nieskończoność 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 nieskończoność, a w F_2 zerowe.

Niestety w rzeczywistości rolki wykonane są ze śliskiego plastiku, który pozwala na mały ruch wzdłuż 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. Podłożę 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 podłożą 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 nagłówkiem 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 wzduż wektorów F_1 i F_2 .
- Odbiór wywołań ustawiających masę 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 zmienoprzecinkowych, 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.

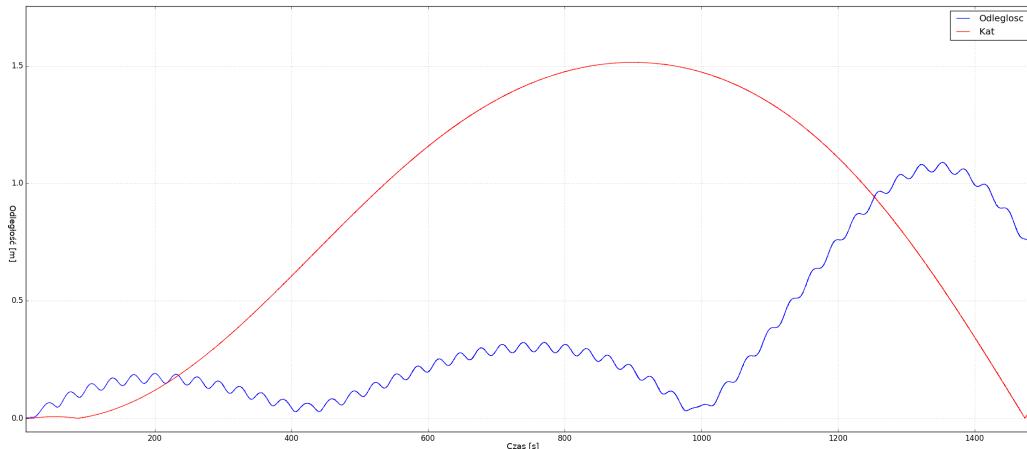
Nadanie prędkości modelom w przestrzeni wirtualnej polega na wywołaniu odpowiedniej funkcji maszyny symulującej fizykę. Można zadać sobie pytanie, jak zachowa się model, jeśli dla niektórych kół nie zmieniać prędkości po każdym odebraniu pakietu.

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 zmienoprzecinkowe zawarte w ROSowych pakietach na dane zrozumiałe przez sterownik silnika, które zazwyczaj są liczbami stałoprzecinkowymi, nie mogącymi przedstawić 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 wynikałoby 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 współczynnikom 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 zauważać, 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**.

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 element w kształcie walca, odpowiedzialny 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.

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. Jednym sposobem na rozwiązywanie 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.

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 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

Punkt ramki	Nazwa punktu
Stałý środek mapy	<code>map</code>
Środek platformy	<code>omnivelman</code>
Emiter prawego lasera	<code>monokl_r_heart</code>
Emiter lewego lasera	<code>monokl_l_heart</code>

Tablica 3.1: Nazwy identyfikatorów ramek, używanych w symulatorze.

Nazwa	Punkt względny	Punkt danych
Pozycja i rotacja platformy	<code>map</code>	<code>omnivelman</code>
Pozycja i rotacja prawego czujnika	<code>map</code>	<code>monokl_r_heart</code>
Pozycja i rotacja lewego czujnika	<code>map</code>	<code>monokl_l_heart</code>

Tablica 3.2: Ramki wysyłane do demona ROS.

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 (nadowanej przez inny sterownik). Stałą, ponieważ czujniki nie zmieniają swojej pozycji na platformie, są przytwierdzone na stałe.

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 wprowa-

dzenie 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 obarwione 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 standar-dowego zostały wprowadzone do standardu języka C++ w 2011 roku.

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] Strona internetowa symulatora Gazebo.
<http://gazebosim.org>
- [9] Strona internetowa symulatora V-Rep.
<http://coppeliarobotics.com>

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