



Politechnika Warszawska  
Wydział Elektroniki i  
Technik Informacyjnych

WYDZIAŁ ELEKTRONIKI  
I TECHNIK INFORMACYJNYCH

Drugie sprawozdanie z Pracowni Dyplomowej  
Inżynierskiej I — PDI2

Informatyka

Tytuł:

Symulacja dookólnej bazy mobilnej

Autor:

Radosław Świątkiewicz

Opiekun naukowy:  
dr hab. inż. Wojciech Szynkiewicz

Warszawa, 13 stycznia 2017

## **Streszczenie**

Ta praca opisuje projektowanie i budowę środowiska symulacyjnego dla wielokierunkowej platformy mobilnej poruszającej się za pomocą kół szwedzkich. Platforma i robot, którego ma wozić są własnością wydziału Elektroniki i Technik Informacyjnych na Politechnice Warszawskiej. Celem jest przygotowanie jak najdokładniejszej kopii oryginału, aby użyć jej zewnętrznym w programie sterującym bez jego modyfikacji.

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

<b>1 Wstęp</b>	<b>2</b>
1.1 Cel . . . . .	2
1.2 Wielokierunkowa platforma mobilna . . . . .	3
1.3 Koła szwedzkie . . . . .	5
1.4 Składniki systemu . . . . .	6
1.4.1 Model 3D . . . . .	7
1.4.2 Sterownik silników . . . . .	7
1.4.3 Sterownik czujników . . . . .	8
1.4.4 Program sterujący . . . . .	8
1.5 Technologie . . . . .	9
1.5.1 Gazebo . . . . .	9
1.5.2 V-Rep . . . . .	10
1.5.3 ROS . . . . .	10
1.5.4 Narzędzia . . . . .	11
1.6 Plan pracy . . . . .	12
1.7 Istniejące implementacje . . . . .	13
<b>2 Model</b>	<b>14</b>
2.1 Sposób zapisu w formacie SDF . . . . .	15
2.2 Model kinematyczny . . . . .	15
2.2.1 Problemy implementacji . . . . .	16
2.2.2 Komunikacja . . . . .	16
2.2.3 Zachowanie . . . . .	17
2.3 Model dynamiczny . . . . .	17
2.3.1 Jak największe zbliżenie do oryginału . . . . .	17
2.3.2 Resetowanie pozycji koła . . . . .	18
2.3.3 Zmiana osi rolki . . . . .	18
2.4 Podsumowanie . . . . .	19

# Rozdział 1

## Wstęp

### 1.1 Cel

Celem tej pracy inżynierskiej jest budowa środowiska robota mobilnego w przestrzeni wirtualnej. Za zadanie jest stworzyć oprogramowanie modelu 3D, oraz modelu dynamiki wielokierunkowej platformy mobilnej z kołami szwedzkimi. Wymaga się, aby model i obsługujące go oprogramowanie było dokładną kopią prawdziwego robota, dzięki czemu zachowanie symulacji będzie jak najbardziej zbliżone do zachowania fizycznego obiektu. Opisywana platforma będzie używana jako baza wielokierunkowa do przemieszczania dwuramiennego robota manipulującego Velma.

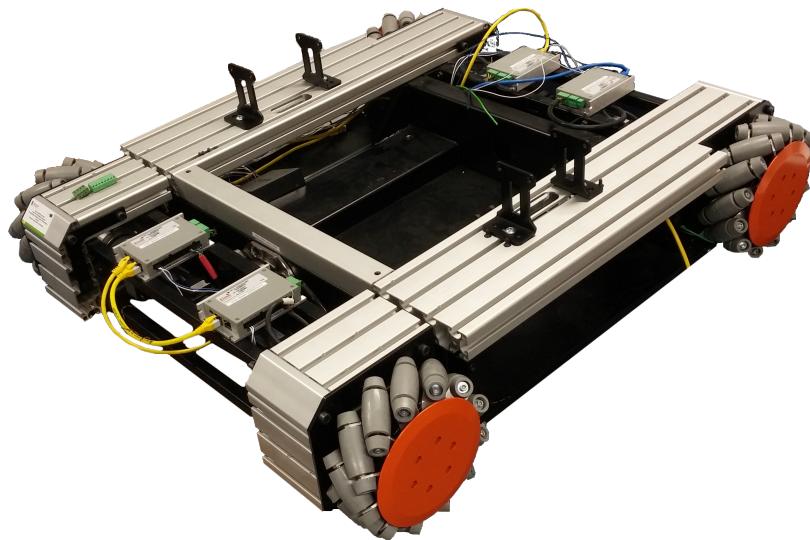
Należy tak opisać model, aby reagował na siły podobnie do rzeczywistej wersji i przyjmował to samo sterowanie z zewnątrz, co rzeczywisty obiekt. To spowoduje, że możliwe będzie stworzenie jednego wspólnego programu sterującego do użycia zarówno w wirtualnej wersji, jak i fizycznej.

Testowanie oprogramowania na prawdziwym obiekcie może prowadzić do jego uszkodzeń, dlatego wpierw trzeba się upewnić o poprawności projektowanych rozwiązań na bezpiecznej kopii wirtualnej. Rzeczywistość nie pozwala także na przeprowadzanie zaawansowanych scenariuszy środowiska testowego. Szybciej i taniej jest stworzyć wirtualne środowisko testowe, niż fizyczne, w dodatku porażka sterowników przy symulacji nie wpływa na zniszczenie robota w rzeczywistości. Dopiero przy osiągnięciu satysfakcjonującej jakości sterowania w symulacji wirtualnej można zastosować algorytmy sterowania do oryginalnego obiektu bez ryzyka uszkodzeń urządzenia.

Należy także móc obsługiwać czujniki, za pomocą których oprogramowanie orientuje się w przestrzeni i generuje sterowanie. Wirtualizacja czujników polega na generowaniu danych na podstawie symulacji. W celu przybliżenia wyjścia takiego programu do rzeczywistego urządzenia, do generowanych

danych zwykle dodaje się szum, oraz błędy.

## 1.2 Wielokierunkowa platforma mobilna



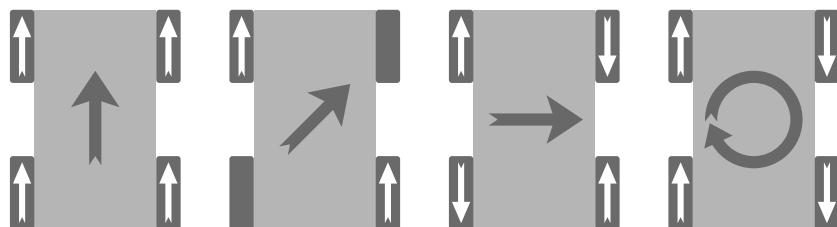
Rysunek 1.1: Fotografia platformy z perspektywy.

Jest to duża, prostokątna baza dookoła poruszająca się na czterech kołach szwedzkich. Koła są stałe, parami przyczepione do dwóch osi. Każde koło jest sterowane osobno przez podłączony bezpośrednio serwomotor, zatem może mieć prędkość i kierunek niezależny od pozostałych kół i kierunku poruszania się robota, oraz jego obrotu. Każdy z serwomotorów ma także wbudowany enkoder umożliwiający pomiar rzeczywistego kąta obrotu koła.



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 wyżej platformie.

Odpowiedni obrót kół względem bazy pozwala na jej ruch w dowolnym kierunku niezależnie od kąta obrotu robota. Za ich pomocą da się także obracać bazą 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.

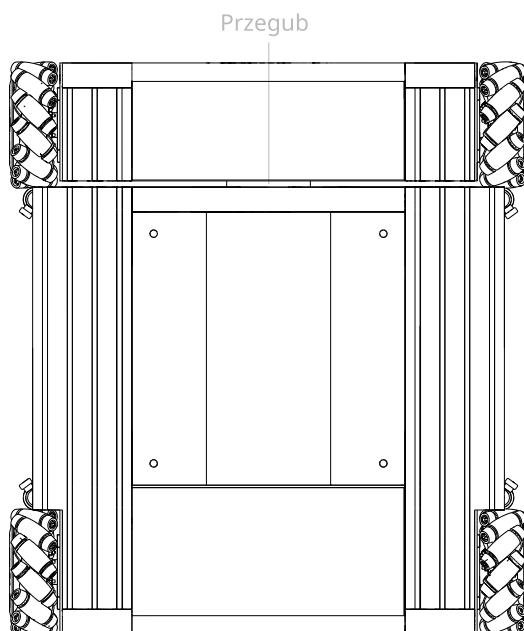


Rysunek 1.3: Podstawowe ruchy, jakie może wykonywać robot o napędzie wielokierunkowym.

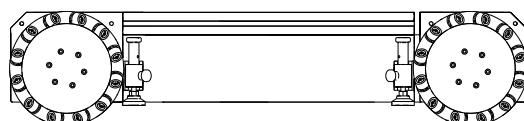
Podstawa ma za zadanie transportować wydziałowego robota manipulującego Velma tworząc razem uniwersalny manipulator mobilny. Velma to wysoki i bardzo ciężki robot wyposażony w dwa chwytki na ramionach o

licznych przegubach. Taka budowa wymaga szerokiej podstawy, aby zachować wystarczająco 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.

Platforma jest niesymetrycznie podzielona w poprzek na dwie niezależne części. Przegub o jednym stopniu swobody (tzw. zawias) jest jedynym łącznikiem pomiędzy tymi dwoma fragmentami. Zadaniem tego przegubu jest niwelować niedoskonałości terenu, 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 nierównego 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.



Rysunek 1.4: Platforma mobilna widziana od góry. Przegub zawiasowy łączy dwie części. Należy zwrócić uwagę na nieprawidłowy układ kół.



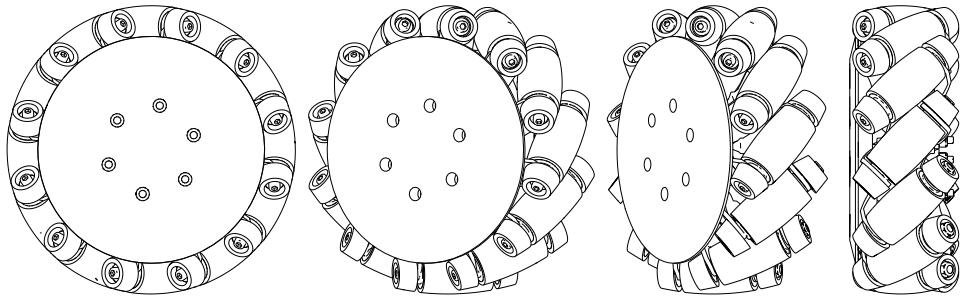
Rysunek 1.5: Platforma mobilna widziana od prawej strony.



Rysunek 1.6: Platforma mobilna widziana od tyłu.

### 1.3 Koła szwedzkie

Koła szwedzkie, zwane także kołami mecanum, to specjalne koła z dodatkowymi rolkami na obwodzie ustawionymi pod kątem  $45^\circ$  do osi koła. Rolki są pasywne i obracają się niezależnie od siebie. Każde koło posiada 12 takich rolek. 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.



Rysunek 1.7: Dokładny 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].

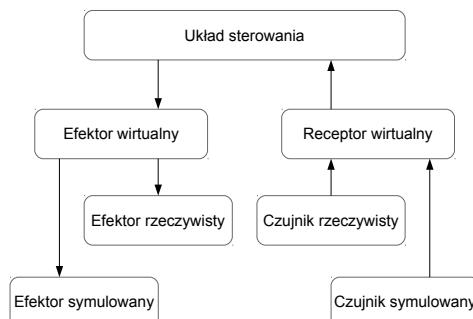
Na podstawie rysunku widać, że krzywizna rolki jest tak ustawiona, aby punkt kontaktu rolki z podłożem płynnie przechodził na następną rolkę w trakcie obrotu. Nie powinno być efektu przeskoku z jednej rolki na drugą, gdyż to wprowadza nierówne tarcie i losowe poślizgi. Koło można wpisać w kulę i każda rolka będzie dotykać jej sfery na całej swojej długości.

## 1.4 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 zamieniać i przepisywać kod źródłowy elementów, używać różnych języków programowania zachowując tę samą komunikację między składnikami i nie tracąc kompatybilności między sobą.

Efektor rzeczywisty, na przykład serwomotor jest sterowany za pomocą efektora wirtualnego, który zamienia wyjście głównego układu sterowania na zrozumiałe dla silnika wartości fizyczne. Przykładowo zmienia podaną liczbę oznaczającą zadaną prędkość na odpowiednie napięcie na wyjściu. W przestrzeni symulacyjnej jedynie wywołuje odpowiednie funkcje maszyny do symulacji.

Podobnie 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, usunięcie błędów wbudowanym układem i zamiana na mapę odległości, szkielety wykrytych osób, ich sylwetki i wiele innych gotowych danych.



Rysunek 1.8: Struktura agenta upustaciowanego.

### 1.4.1 Model 3D

Odpowiednio opisany równaniami fizycznymi powinien mieć zachowanie zbliżone do oryginału w jak największym stopniu. Musi brać pod uwagę masy i momenty bezwładności elementów składowych, a także wszelkie tarcia. Model posiada więzy na ruchome elementy, jak koła i rolki, aby symulować przeguby.

Ta część środowiska oddziałuje bezpośrednio z maszyną do symulacji fizycznej. To kształt, masy i momenty bezwładności brył są argumentami funkcji liczących. Także maszyna manipuluje z powrotem podanymi obiektami nadając im wirtualnie odpowiednie prędkości w czasie.

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.

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

#### 1.4.2 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.

### **1.4.3 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, są 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.

### **1.4.4 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 zapotrzebowanie 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 nie zależy, 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.

## 1.5 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 najwierniejszego odtworzenia zachowania rzeczywistego obiektu.

Na rynku jest wiele różnych maszyn zarówno do symulacji w czasie rzeczywistym, jak i do wyznaczania pozycji obiektów po długich obliczeniach. 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.

### 1.5.1 Gazebo

Program do pobrania z [3]. 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 „offline” w ten sposób mamy nad nimi pełną kontrolę, a parametry można ustawać 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.

### 1.5.2 V-Rep

Program do pobrania z [4]. 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.

### 1.5.3 ROS

Platforma programistyczna do pobrania z [5]. 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 nie koniecznie 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 wykonania, 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 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 wczytywania całościowych plików. Użycie niektórych funkcji ROS wymaga uruchomionego demona serwera w tle.

Ogólnie instalacja i używanie ROS na systemie zostawia dużo różnorodnych plików, 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.

#### 1.5.4 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 ODE. Przełączenie maszyny symulacyjnej wiąże się z rekompilacją programu ze źródła, dlatego zostaną one sprawdzone po przetestowaniu domyślnej. ODE jest szybkie, dobrze napisane i nie ma wielu zaawansowanych niepotrzebnych nam funkcji, jak Bullet.

**ROS** użyty zostanie jako główna platforma programistyczna. Pod łatwą komunikację 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.

**Code::Blocks lub Eclipse** nadają się do pisania kompilowanego kodu wtyczek. Można podłączyć 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** — program kontroli wersji jest podstawowym narzędziem programisty. Kod należy dla bezpieczeństwa umieszczać także w usłudze GitHub.

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

## 1.6 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 działającą siłą na elementy i patrząc, czy reagują w spodziewany sposób.
3. Zapisanie wtyczki sterującej w Gazebo odczytującej odpowiednie dane z zewnątrz i wywołującej funkcje maszyny symulacyjnej, aby modyfikować ruch modelu. Na tym poziomie można dobudować zamiennik programu sterującego jedynie do podawania prostych wartości bez odczytywania pomiarów i sterowania.
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.
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.

## 1.7 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 tego typu. 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 obraca kołami w ten sposób, aby spełnić żądanie. Wersja dla Gazebo jest statycznym modelem, 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.

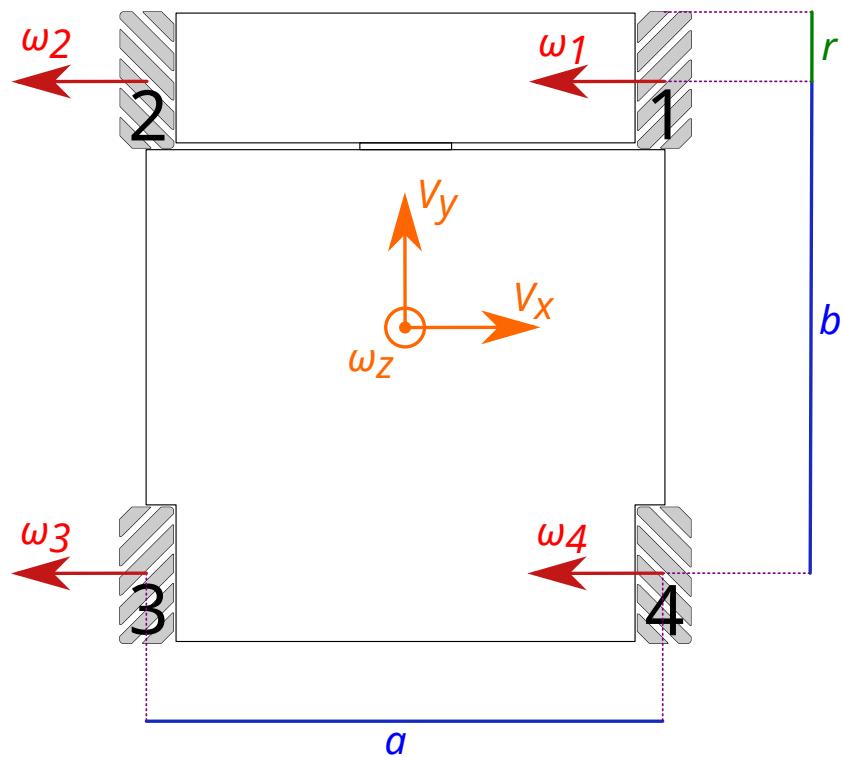
Ze względu na niezwykle zaawansowany obiekt kół i kształt rolek, prawdopodobnie trzeba będzie uprościć model poprzez zamianę niektórych składowych i dodanie sztucznych więzów. Całościowy model może być zbyt skomplikowany, aby maszyny symulacji mogły go obliczać w czasie rzeczywistym. Taki model także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

## Rozdział 2

### Model

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.
$\omega_i$		Prędkość kątowa każdego z kół.
$V_x$		Prędkość transwersalna w osi X.
$V_y$		Prędkość transwersalna w osi Y.
$\omega_z$		Prędkość kątowa w osi Z, wektor skierowany w górę.

## 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 zamiennik 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 wiezy i względne pozycje. Standard jest dobrze opisany na ich stronie internetowej [6].

Na szcycie wszystkich elementów znajduje się `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 do innego pliku i określić ścieżkę.

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 równolegle typy `joint` określające 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 środka ciężkości platformy w układzie współrzędnych 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 standarze, zatem został stworzony własny typ `pseudovelma/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/Pose` z aktualną pozycją i rotacją platformy.
- Odbiór wiadomości własnego typu `pseudovelma/Vels` z zadanimi 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 trójkątnej. 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. Zwiększając jej gęstość zwięksamy jakość symulacji, ale narażamy się na olbrzymi skok ilości obliczeń. Kalkulowanie kolizji z siatką jest najdroższe z możliwych.

### 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  $45^\circ$ , 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  $45^\circ$  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 podłożu. 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 możliwe.

Bardzo skomplikowany sposób działania kół skłania do szukania innych rozwiązań.

### 2.3.3 Zmiana osi rolki

Poprzedni przypadek można zmodyfikować poprzez wyznaczanie nowej osi przegubu łączącego wirtualne koło z rolką.

Oś wewnętrzną podłączone jest do koła wirtualnego i obraca się razem z nim. W każdym cyklu należy zamienić obróconą już nieco oś na kierunek pier-

wotny 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 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.

## 2.4 Podsumowanie

W kolejnych przygotowaniach do pracy udało się przetestować różne sposoby modelowania bazy, znaleźć najlepszy działający i stworzyć modele. Została zdobyta wiedza na temat problemów obsługi programu Gazebo, jakie funkcjonalności nie są zaimplementowane i jakie nie działają. Nie jest to nigdzie dobrze opisane, więc błędy zdarzały się w zupełnie nieprzewidzianych okolicznościach.

Należało nauczyć się działania architektury systemu ROS, aby poznać działanie jego składowych i usług standardowych programów. Udało się uzyskać standardowe pakiety systemu ROS kompilowane przez jego systemy, komunikujące się w standardowy sposób i przenośne na inne komputery.

Należy teraz stworzyć identyczny model w programie V-Rep, a dokładniej zmodyfikować istniejący Kuka Youbot pod kątem rozmiarów bazy. Trzeba go podłączyć do systemu ROS w taki sam sposób, jak poprzednie, aby dało się nim równie łatwo sterować.

Do zrobienia jest system ułatwiający manualne sterowanie bazą za pomocą interfejsu graficznego i zbierający informacje do porównania z ruchami prawdziwego robota. Należy stworzyć także elementy pomocnicze, jak interaktywny model podłogi ze zmiennym współczynnikiem tarcia. Interfejs graficzny będzie wtyczką w QT tworzącą kolejny panel w ROSowym programie *rqt*.

Na koniec używając wystawionego interfejsu metodą prób i błędów należy znaleźć takie ustawienia (np. masa, tarcie), które produkują rezultat najbardziej zbliżony do działania rzeczywistego modelu.

# 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] Strona internetowa symulatora Gazebo. <http://gazebosim.org>
- [4] Strona internetowa symulatora V-Rep. <http://coppeliarobotics.com>
- [5] Strona internetowa platformy programistycznej *Robot Operating System*.  
<http://www.ros.org>
- [6] Strona internetowa standardu SDF. <http://sdformat.org/spec>