Politechnika Warszawska

WYDZIAŁ MECHANICZNY
ENERGETYKI I LOTNICTWA

Instytut Techniki Lotniczej i Mechaniki Stosowanej

Praca dyplomowa inżynierska

na kierunku Robotyka i Automatyka
w specjalności Robotyka

Cyfrowy Bliźniak czworonożnego robota Meldog

Stanisław Sufin
Numer albumu 310555

promotor
Dr. inż. Łukasz Woliński

Warszawa, 2024

Cyfrowy Bliźniak czworonożnego robota Meldog

Streszczenie pracy

Niniejsza praca dyplomowa realizowana w ramach działalności w Kole Naukowym Robotyków, opisuje proces tworzenia modelu symulacji czworonożnego robota kroczącego Meldog. Do osiągnięcia celów obranych dla projektu, wykorzystano oprogramowanie ROS2 w połączeniu z symulatorem Gazebo Sim.

Praca podzielona jest na osiem rozdziałów:

Rozdział pierwszy stanowi wstęp zawierający cel i zakres pracy, motywację do jej podjęcia, oraz przyjęte założenia projektu.

Drugi rozdział poświęcony jest przeglądowi stanu wiedzy oraz istniejących rozwiązań. Objaśnia on podstawowe założenia i zalety technologii Digital Twin oraz odnosi się do głównej pracy, na której wzorowany jest niniejszy dokument. Ponadto zawiera on dalszy opis używanego oprogramowania oraz specyfiki jego środowiska.

Rozdział trzeci stanowi szczegółowy opis struktury robota. Przedstawione są w nim przyjęte uproszczenia konstrukcji w symulacji, a także opisana jest dokładnie konwencja nazewnictwa poszczególnych członów oraz złącz robota. Rozdział zawiera również schemat kinematyczny robota.

Rozdział czwarty stanowi szczegółowy opis pakietu meldog_description, w szczególności jego folderu description zawierającego całość pliku URDF robota. Opisuje zastosowane rozwiązania oraz objaśnia strukturę pakietu, a także ilustruje sposób uruchamiania symulacji robota w środowisku Gazebo.

Rozdział piąty stanowi krótki opis działania regulatora PID, opartego na platformie ROS2 Control, napisanego na potrzeby projektu. Przedstawia głównie równania określające przyjęte prawo sterowania oraz opisuje proces weryfikacji poprawności jego działania.

Rozdział szósty stanowi opis procesu weryfikacji dokładności odwzorowania robota w symulacji. Zawiera szczegółowy opis pakietów napisanych na potrzeby symulacji pojedynczej jednostki napędowej robota, pakietu związanego z zadawaniem trajektorii dla tejże symulowanej jednostki, a także skryptów Python przeznaczonych do sterowania rzeczywistą jednostką napędową. Na końcu rozdziału przedstawiono przebieg testów na rzeczywistej jednostce oraz analizę uzyskanych wyników.

Rozdział siódmy opisuje proces optymalizacji parametrów modelu, celem poprawienia dokładności odwzorowania rzeczywistych przebiegów dla pojedynczej jednostki napędowej.

W rozdziale ósmym zawarto wnioski i potencjalne drogi przyszłego rozwoju projektu.

Słowa kluczowe: cyfrowy bliźniak, ROS, Gazebo Sim, robot czworonożny

Creating a Digital Twin for the quadruped robot Meldog

Abstract

This Engineering thesis, conducted as part of the author's activity in the Students Association of Robotics, describes the process of creating a simulation model for the quadruped robot Meldog. Robot Operating System 2 (ROS 2) and Gazebo sim are utilized to achieve the goals set for the project.

The thesis is divided into 8 chapters:

Chapter one is an introduction describing the goals set for the project, the scope, as well as the main motivation for undertaking the described topic.

Chapter two is a review of literature and current state of knowledge on the subject. It describes in detail the main idea behind digital twin technology and already existing solutions in the field of quadruped robot simulation. In addition, detailed description of used software is presented along with the specifics of its integration within the project's ecosystem.

Chapter four describes in detail the structure of the ROS2 package meldog_description created for the purpose of simulating the robot. The chapter focuses mainly on the description folder of the package containing the entire URDF file of the robot split into multiple sub-files according to their respected functionality.

Chapter five briefly describes the PID controller written for the project using the ros2_control framework. It presents the main theory of operation and control laws applied in the software.

Chapter six describes the process of verifying the prepared model's accuracy. It contains detailed description of the ROS2 package designed for simulating a single actuator. Furthermore, it contains a detailed description of software written for the purpose of controlling joint motion both in the simulation environment and the real actuators. The achieved motion characteristics are shown on separate graphs and the root mean square error is calculated for position and velocity.

Chapter seven focuses on optimization of the model's physical parameters using a patternsearch function with the goal of improving the accuracy of the simulated motion. A before and after comparison for all trajectories is presented and the percentage of improvement is given.

Chapter eight contains the summary of the project along with conclusions and potential improvements that can be applied to the model in the future.

Keywords: digital twin, ROS, Gazebo Sim, quadruped robot

Spis treści

1. Wstęp

1.1. Cel pracy oraz motywacja

Celem niniejszej pracy dyplomowej jest zaprojektowanie cyfrowego bliźniaka dla czworonożnego robota kroczącego o nazwie Meldog. Model robota ma zapewnić odpowiednią dokładność symulacji ruchu robota, a także w odpowiednim stopniu naśladować istniejące już sposoby sterowania nim. Do odwzorowania rzeczywistej struktury robota w środowisku komputerowym wykorzystano platformę Robot Operating System 2 w połączeniu z symulatorem Gazebo Sim.

Główną motywacją do podjęcia tematu pracy była realna potrzeba posiadania cyfrowego bliźniaka w projekcie Meldoga w sekcji robotów kroczących Koła Naukowego Robotyków. Stworzenie modelu symulacyjnego tegoż robota znacznie uprościłoby dalsze prace w zespole, zwiększając możliwości indywidualnej pracy z robotem każdego z członków zespołu programistów. Możliwość symulacji robota jest również krytyczna dla dalszych planów rozwoju projektu, gdyż docelowo ma on być przeznaczony do trudnych warunków pracy, które nie są łatwe do odwzorowania w rzeczywistości.

1.2. Założenia pracy

Dla projektu omawianego w niniejszej pracy przyjęto następujące założenia:

1. Model robota musi być łatwy do modyfikacji w razie potrzeby wprowadzenia zmian w jego strukturze
2. Struktura projektu powinna być intuicyjna, jego katalogi i poszczególne elementy kodu powinny być właściwie uporządkowane i opisane w komentarzach.
3. Symulacja dla opracowanego modelu powinna być możliwie wydajna i płynna nawet dla złożonego ruchu wszystkich kończyn.
4. Model powinien odwzorowywać swój fizyczny odpowiednik z dostateczną dokładnością.
5. Model powinien opierać się na platformie ROS 2

1.3. Zakres pracy

Zakres czynności wykonanych w ramach niniejszej pracy dyplomowej obejmował:

1. Przygotowanie podstawowego pliku URDF robota bez funkcjonalności związanych z symulacją w Gazebo Sim.
2. Przygotowanie pliku URDF robota z funkcjonalnością rozszerzoną o moduły związane z symulacją.
3. Zaimplementowanie strategii sterowania robotem w symulacji z wykorzystaniem modułu ROS2 Control.
4. Napisanie oraz implementacja autorskiego regulatora PID do sterowania momentem dla poszczególnych stawów robota.
5. Przeprowadzenie identyfikacji pożądanych nastaw regulatora oraz wartości parametrów tłumienia i tarcia w stawach, celem polepszenia dokładności opracowanego modelu.

2. Przegląd stanu wiedzy oraz istniejących rozwiązań

Przed rozpoczęciem prac dokonano dokładnego przeglądu stanu wiedzy o zagadnieniu tworzenia Cyfrowych bliźniaków, a także przeanalizowano istniejące rozwiązania. W tym rozdziale opisane zostanie ogólna problematyka zagadnień technologii Digital Twin. Opisane zostanie również odpowiednie oprogramowanie, z którego zdecydowano się korzystać na potrzeby niniejszej pracy.

2.1. Przegląd literatury

W pierwszej kolejności zostanie opisana technologia Digital Twin, jej problematyka, oraz zakres zastosowań. Opisane zostaną również istniejące rozwiązania z zakresu symulacji i modelowania robotów kroczących, a w szczególności najważniejsza praca naukowa, stanowiąca wzorzec dla niniejszego dokumentu.

2.1.1. Technologia Digital Twin

Testowanie nowego oprogramowania na rzeczywistych robotach może być problematyczne z różnych powodów. W niektórych przypadkach takie podejście może okazać się nieefektywne oraz kosztowne, na przykład, gdy prowadzone testy mają charakter iteracyjny, a ich czas wykonywania jest stosunkowo długi, lub gdy wymagają one całkowitego zatrzymania procesu produkcyjnego.

Innym przypadkiem, gdy tego typu podejście okazuje się nieefektywne, jest sytuacja, gdy wiele osób w zespole, pracujących nad różnym oprogramowaniem, dysponuje tylko jednym fizycznym robotem do testów. Dostępność robota jest znacznie ograniczona, co powoduje spadek wydajności pracy całego zespołu. Co więcej, oprogramowanie może okazać się wadliwe i uszkodzić robota, bądź jego otoczenie.

Rozwiązanie tych oraz wielu innych problemów dostarcza technologia Cyfrowych Bliźniaków [1], która opiera się na tworzeniu symulacji pojedynczych urządzeń lub procesów. Podejście to jest szeroko stosowane w przemyśle oraz wielu innych branżach, w których symulacja i optymalizacja odgrywają kluczową rolę. Szczególnie ważne dla niniejszej pracy jest zagadnienie tworzenia Cyfrowych bliźniaków robotów kroczących, które zostało dokładnie opisane w pracy [2] J. Mazurkiewicz (2024) *„Tworzenie cyfrowego bliźniaka dla robota humanoidalnego w ROS: Modelowanie i Symulacja”*.

2.1.2. Istniejące rozwiązania

Powszechnym podejściem do symulacji układów mechanicznych jest wykorzystanie programu Adams, ze względu na jego bogatą funkcjonalność, oraz możliwość stosowania nawet dla bardzo złożonych mechanizmów. Oprócz wysokiej dokładności obliczeń Adams umożliwia również integrację z oprogramowaniem Matlab i Simulink, w celu wydajnego modelowania złożonych systemów sterowania.

Wykorzystanie programu Adams w połączeniu ze środowiskiem Matlab Simulink do symulacji czworonożnego robota kroczącego opisano szczegółowo w pracy [3].

Adams jest jednak oprogramowaniem stawiającym nacisk na dokładność symulacji nad jej szybkością i nie jest przeważnie wykorzystywany do zastosowań wymagających obliczeń w czasie rzeczywistym. Co więcej, nie pozwala on bezpośrednio na modelowanie czujników, oraz nie ma szerokich możliwości integracji z platformą ROS 2, co czyni go mało praktycznym narzędziem w ujęciu tworzenia cyfrowych bliźniaków.

Innym środowiskiem oferującym duży stopień dokładności symulacji z naciskiem na układy wieloczłonowe jest silnik MuJoCo. W odróżnieniu od Adamsa MuJoCo stawia nacisk na wydajną symulację nawet dla złożonych układów, co czyni go silnym narzędziem do pracy w czasie rzeczywistym. Wykorzystanie MuJoCo do symulacji robota czworonożnego opisano w pracy [4].

Silnik ten nie jest jednak dostosowany do wymagań symulacji systemów robotycznych, gdyż nie umożliwia on symulacji bardziej złożonych czujników, które może w przyszłości posiadać Meldog. MuJoCo zapewnia pewien stopień integracji z platformą ROS 2, jednak szczegóły jej implementacji na ten moment nie są wystarczająco udokumentowane.

Ostatnim, najbardziej przystosowanym do celów projektu podejściem jest wykorzystanie symulatora Gazebo Sim, posiadającego rozległe możliwości integracji z platformą ROS 2, oraz licznymi pakietami służącymi do symulacji układów sterowania. Wykorzystanie Gazebo w połączeniu z ROS 2 do symulacji robota kroczącego opisano w pracy [5] oraz [2].

Szczegóły tego podejścia opisano w następnym podrozdziale, poświęconym pracy opisującej proces tworzenia innego cyfrowego bliźniakowi w kole naukowym.

2.1.3. Cyfrowy bliźniak robota Melman

Głównym wzorcem dla niniejszej pracy dyplomowej jest projekt Cyfrowego Bliźniaka robota humanoidalnego Melman opracowany w Kole Naukowym Robotyków. W niniejszym projekcie planuje się wykorzystanie przetestowanych już w pracy [2] rozwiązań symulacji oraz struktury modelu. Pozwoli to na ujednolicenie konwencji przy projektowaniu oraz obsłudze Cyfrowych Bliźniaków w kole naukowym, co ułatwi dostęp do bliźniaków innym członkom koła.

Praca ta dokładnie opisuje budowę modelu robota kroczącego oraz metody zapewnienia odpowiedniego poziomu odwzorowania jego fizycznego pierwowzoru. Bliźniak Melmana skonstruowany jest przy pomocy platformy ROS2 oraz symulatora Gazebo. Jego rdzeniem jest opisujący strukturę kinematyczną robota plik URDF, odpowiednio podzielony na fragmenty związane z poszczególną funkcjonalnością kodu.

Sterowanie modelem zrealizowane jest poprzez moduł ROS2 Control, który umożliwia obsługę wielu rodzajów regulatorów dla różnych przypadków sterowania robotem. Weryfikacja dokładności modelu odbywa się poprzez zebranie danych z rzeczywistych przebiegów dla układów Dynamixel napędzających złącza Melmana, następnie wykorzystany jest program w środowisku MatLab, który wykorzystując funkcję pattern search, dobiera odpowiednie nastawy regulatorów w symulacji, przyjmując sumę kwadratów błędu między rzeczywistym a symulowanym przebiegiem jako funkcję strat do minimalizacji.

Do osiągnięcia celów założonych w niniejszej pracy zastosowane zostaną opisane wyżej metody. Porównanie ruchów dla całego bliźniaka nie będzie jednak możliwe, gdyż obecnie nie powstał jeszcze jego fizyczny korpus, dlatego analiza ruchu zostanie ograniczona do pojedynczej jednostki napędowej.

2.2. Przegląd wykorzystanego oprogramowania

W następnej części dokumentu przedstawiono najważniejsze elementy oprogramowania, które wybrano do zrealizowania celów projektu. W opisie zawarto ogólną specyfikę oprogramowania, jego najważniejszą funkcjonalność, a także podstawowe przykłady istotne dla zrozumienia zagadnień poruszanych w niniejszej pracy.

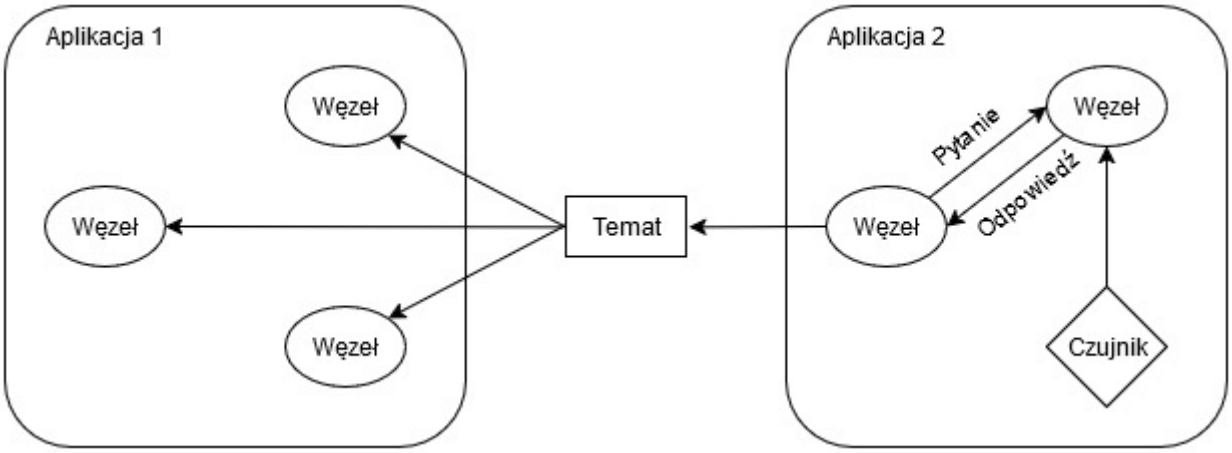
2.2.1. Robot Operating System 2

Oprogramowanie Robot Operating System 2 to platforma middleware zawierająca liczne biblioteki i narzędzia do tworzenia aplikacji dla robotów [6]. Posiada funkcjonalności związaną z symulacją, wspiera operacje czasu rzeczywistego, a także posiada bardzo wydajne mechanizmy komunikacji oparte na Data Distribution Service [7].

Do zalet platformy ROS2 należą:

- **Wysoka modularność:** architektura programu pozwala na łatwe łączenie podstawowych modułów w bardziej złożone i abstrakcyjne struktury.
- **Wydajna komunikacja RT:** ROS2 zapewnia wydajną komunikację z hardware'm w czasie rzeczywistym, co jest szczególnie ważne przy pisaniu oprogramowania dla robotów.
- **Niewrażliwość na język programowania:** Współpracujące ze sobą moduły nie muszą być napisane w tym samym języku. Dzięki temu użytkownik zyskuje dużą dowolność w doborze języka, co może znacznie usprawnić pracę z platformą.
- **Gotowe oprogramowanie dla większości hardware'u:** Większość producentów podzespołów do zastosowań w robotyce udostępnia gotowe oprogramowanie do ROS2 dla swoich produktów.
- **Aktywna społeczność:** Duży dostęp do publicznych bibliotek i gotowych rozwiązań.

Architektura ROS2 opiera się na Węzłach (ang. Nodes) o rozmaitej funkcjonalności, komunikujących się ze sobą nawzajem poprzez Tematy (ang. Topics) oraz Serwisy (ang. services).



Rysunek 1: Przykładowa architektura programu w ROS2

Istotną zaletą ROS2, mającą szczególne znaczenie dla tej pracy jest jego szeroka integracja z narzędziami symulacji oraz wizualizacji robotów. Do najważniejszych z tych narzędzi należą RVIZ2 oraz Gazebo Sim.

Głównym sposobem opisu struktury robotów w ROS2 są pliki URDF, czyli Unified Robot Description Format.

2.2.2. Unified Robot Description Format

Pliki URDF zawierają całość informacji o strukturze robota oraz jego parametrach dynamicznych. Stanowią one bazę do symulacji i analizy stanu robotów z wykorzystaniem oprogramowania ROS2 [8]. Pliki te wykorzystują format XML do opisu struktury robota poprzez odpowiednie klasy złączy oraz członów [9]. Dalej zaprezentowane zostaną przykładowe deklaracje członów oraz złączy, czyli najważniejsze elementy plików URDF.

```
<link name="my_link">
  <inertial>
    <origin xyz="0 0 0.5" rpy="0 0 0"/>
    <mass value="1"/>
    <inertia ixx="100"  ixy="0"  ixz="0" iyy="100" iyz="0" izz="100" />
  </inertial>

  <visual>
    <origin xyz="0 0 0" rpy="0 0 0" />
    <geometry>
      <box size="1 1 1" />
    </geometry>
    <material name="Cyan">
      <color rgba="0 1.0 1.0 1.0"/>
    </material>
  </visual>

  <collision>
    <origin xyz="0 0 0" rpy="0 0 0"/>
    <geometry>
      <cylinder radius="1" length="0.5"/>
    </geometry>
  </collision>
</link>
```

Rysunek 2: Fragment pliku URDF opisujący człon

Źródło: [10]

Powyższy fragment kodu definiuje człon o nazwie my_link. Deklaracja członu dzieli się na poszczególne tagi:

- Tag inertial zawiera parametry dynamiczne członu: jego masę, współrzędne środka ciężkości w lokalnym układzie, oraz elementy tensora bezwładności. Wprowadzone w tym tagu dane są szczególnie ważne dla poprawnej i wiarygodnej symulacji robota.

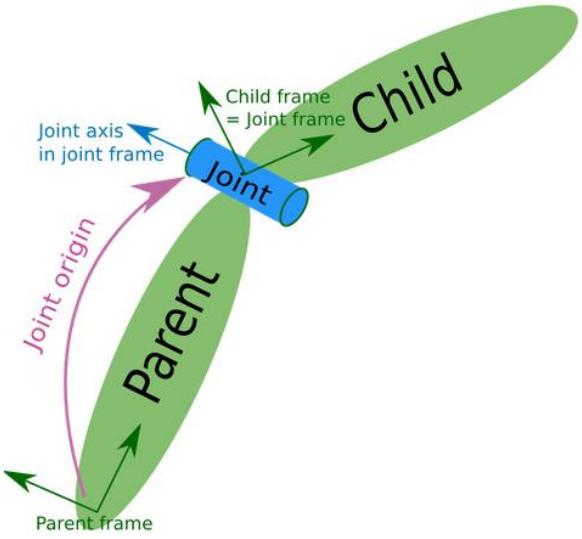
- Tag visual zawiera informacje o wyglądzie członu i ma zastosowanie czysto kosmetyczne. Wygląd może być zadany jako połączenie prostych brył geometrycznych, bądź jako plik formatu STL lub Colada.
- Tag collision zawiera informacje o wyglądzie siatki kolizji dla danego członu. Geometria zadana w tym tagu pełni kluczową rolę podczas symulacji kolizji członu z otoczeniem i może mieć ogromny wpływ na szybkość wykonywanych obliczeń. Należy zadbać, aby dobrana geometria była możliwie uproszczona, jeśli jest to możliwe, do elementarnych form geometrycznych dostępnych w formacie URDF.

```
<joint name="my_joint" type="floating">
   <origin xyz="0 0 1" rpy="0 0 3.1416"/>
   <parent link="link1"/>
   <child link="link2"/>

   <calibration rising="0.0"/>
   <dynamics damping="0.0" friction="0.0"/>
   <limit effort="30" velocity="1.0" lower="-2.2" upper="0.7" />
   <safety_controller k_velocity="10" k_position="15" soft_lower_limit="-2.0" soft_upper_li
mit="0.5" />
 </joint>
```

Rysunek 3: Fragment pliki URDF opisujący złącze

Źródło: [11]

Powyższy fragment kodu deklaruje przykładowe złącze typu „floating” pomiędzy nadrzędnym (parent) członem link1, a członem podrzędnym (child) link2. W ten sposób definiuje się łańcuchy kinematyczne w pliku URDF, gdzie odpowiednio translację oraz obrót do układu współrzędnych członu podrzędnego z układu członu nadrzędnego zadaje się w polu origin. Pozostałe pola klasy dają możliwość dodania wartości takich parametrów jak tłumienie, tarcie, maksymalna siła / moment w złączu, oraz twarde i łagodne ograniczenia zakresów ruchu w parze kinematycznej.



Rysunek 4: Sposób deklarowania złączy w pliku URDF

Źródło: [11]

Należy zaznaczyć, że format URDF nie umożliwia modelowania zamkniętych łańcuchów kinematycznych. Fakt ten okaże się ważny przy projektowaniu modelu czworonoga w dalszych rozdziałach niniejszej pracy.

Nagi format URDF nie obsługuje żadnego rodzaju parametrów, ani deklarowania instrukcji macro. Nie jest również możliwe dołączanie do siebie kilku osobnych plików URDF z poziomu kodu. Sprawia to, że pisany kod ma dużą redundancję i jest mało przejrzysty. Możliwości konfiguracji są również mocno ograniczone.

Aby rozwiązać wymienione wyżej problemy i poszerzyć funkcjonalność formatu, stosuje się rozszerzenie XACRO [12]. Znacznie upraszcza ono pisania plików URDF, dostarczając liczne przydatne funkcje, np. instrukcje warunkowe, parametry, możliwość dołączania innych plików, oraz deklarowania macros.

```
<xacro:property name="the_radius" value="2.1" />
<xacro:property name="the_length" value="4.5" />

<geometry type="cylinder" radius="${the_radius}" length="${the_length}" />
```

Rysunek 5: Przykładowe zastosowanie parametrów xacro

Źródło: [12]

2.2.3. Symulator Gazebo Sim

Program Gazebo Sim to otwartoźródłowe narzędzie symulacji znajdujące szerokie zastosowanie w robotyce, przemyśle, oraz innych branżach stawiających naciska na tworzenie modeli urządzeń oraz procesów produkcyjnych. Swoją funkcjonalność opiera o szereg silników fizycznych: ODE (Open Dynamics Engine), Bullet, oraz Dart [13].

Gazebo Sim umożliwia szczegółową symulację licznych czujników, napędów, oraz systemów sterowania. Dodatkową zaletą tegoż środowiska jest jego szeroka konfigurowalność. Bazowe funkcjonalności można łatwo rozszerzyć o liczne wtyczki (ang. plugins), a dokładność symulacji można zwiększyć dzięki wielu parametrom oferowanym przez silniki fizyczne, w przypadku niniejszej pracy, przez silnik ODE.

Kolejną istotną zaletą środowiska Gazebo jest jego wysoki stopień integracji z platformą ROS2. Paczka ros_gz zapewnia integrację symulatora Gazebo z platformą ROS2, w szczególności:

- Dodaje do Gazebo Sim możliwość przyjmowania danych o strukturze robota w postaci pliku URDF. Jest to kluczowa funkcjonalność dla osiągnięcia celów wyznaczonych w niniejszej pracy.
- Zapewnia jednokierunkowy transport informacji wizyjnych (obrazów) z symulatora do środowiska ROS2. Funkcjonalność ta jest niezbędna w przypadku symulacji różnego rodzaju czujników wizyjnych.
- Zapewnia dwukierunkowy transport informacji, takich jak komunikaty, wiadomości oraz logi pomiędzy modułem Gazebo Transport a platformą ROS2. Niezbędna funkcjonalność dla zapewnienia prawidłowej komunikacji asynchronicznej pomiędzy symulowanymi komponentami robota.

- Dostarcza wielu wygodnych plików wywołania oraz plików wykonywalnych do zastosowania przy współpracy obydwu środowisk. Dzięki tej funkcjonalności możliwe jest uruchamianie symulatora wraz z odpowiednimi modułami ROS2 z poziomu jednego pliku wywołania.
- Zapewnia wtyczki służące do publikowania chmur punktów z Gazebo do odpowiednich tematów w ROS2. Szczególnie ważna funkcjonalność dla czujników LIDAR.

Gazebo Sim umożliwia symulację wyłącznie brył sztywnych. Mechanizmy kolizji sprężystych, w przypadku których następują lokalne odkształcenia członów, symulowane są przy pomocy odpowiednich parametrów silnika ODE. Ta cecha zostanie wykorzystana przy modelowaniu kontaktu stóp robota z podłożem.

Głównym formatem pliku wykorzystywanym w symulatorze Gazebo są pliki SDF (Simulation Description Format), które pozwalają na określenie parametrów symulacji, dostosowanie świata, oraz opisują strukturę symulowanego robota. Podobnie jak pliki URDF, format SDF oparty jest na XML. Dzięki paczce ros_gz symulator Gazebo Sim generuje model robota w oparciu o dostarczony przez użytkownika plik formatu URDF, przy czym informacje, których nie jest w stanie dostarczyć bazowy format URDF, o czujnikach, oraz parametrach symulacji dostarczane są w odpowiednim tagu <Gazebo> wewnątrz przekazanego pliku URDF.

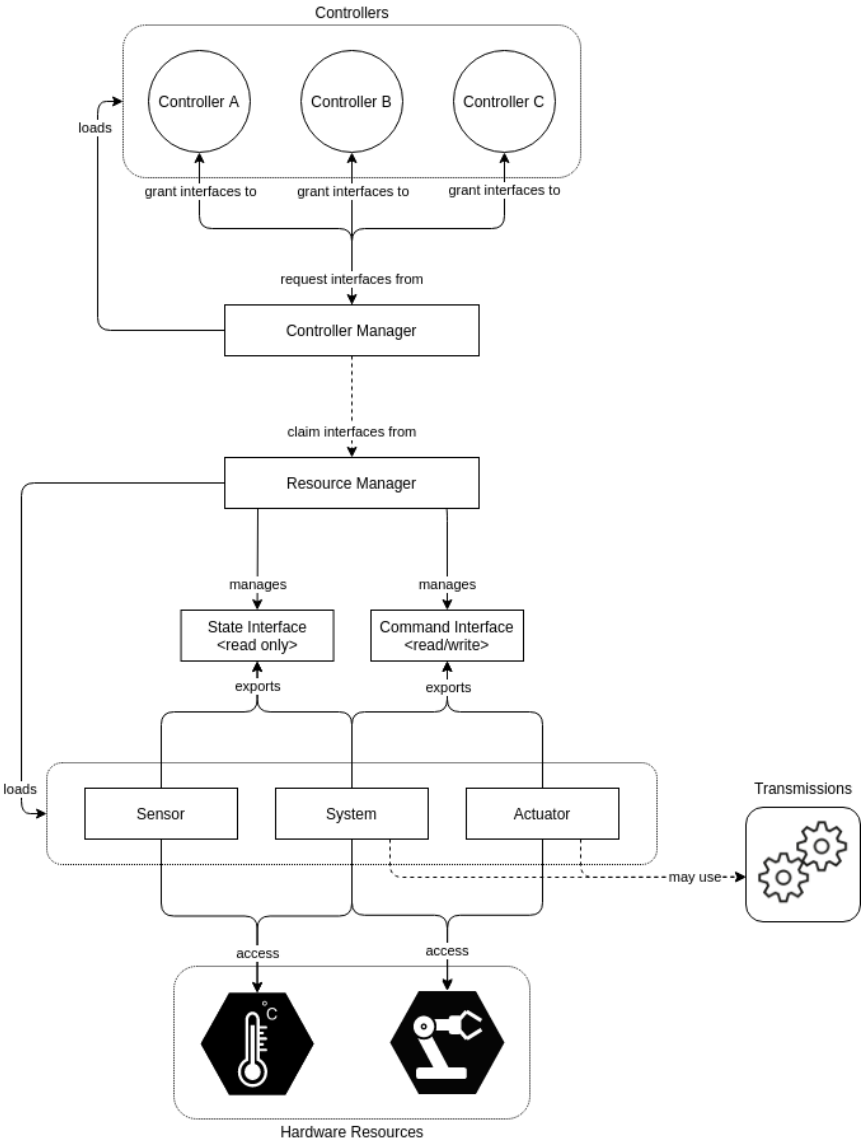
```
<?xml version='1.0' encoding='UTF-8'?>
<!--URDF-->
<robot name='friction_example'>
  <link name='base_link'>
    <inertial>
      <mass value='0.12' />
      <inertia ixx='0.01' ixy='0' ixz='0' iyy='0.01' iyz='0' izz='0.01' />
    </inertial>
    <collision>
      <geometry>
        <sphere radius="2"/>
      </geometry>
    </collision>
    <collision>
      <geometry>
        <cylinder radius="1" length="2"/>
      </geometry>
    </collision>
  </link>
  <gazebo reference='base_link'>
    <mu1>0.25</mu1>
  </gazebo>
</robot>
```

Rysunek 6: Przykład prostego pliku URDF z tagiem <Gazebo> zadającym parametr tarcia dla członu w symulacji

Źródło: [14]

2.2.4. Pakiet ros2_control

Pakiet ros2_control to platforma dostarczająca funkcjonalność związaną ze sterowaniem robotami w czasie rzeczywistym przy użyciu oprogramowania ROS2. Jego architektura oparta jest na kilku warstwach, z których najważniejszą rolę odgrywa Menedżer kontrolerów [15]. Poniżej przedstawiono schemat architektury ros2_control.



Rysunek 7: Architektura platformy ros2_control

Źródło: [15]

Menedżer kontrolerów (ang. Controller Manager) odpowiednio inicjalizuje wybrane przez użytkownika kontrolery z pliku Yaml oraz realizuje główną pętlę regulacji poprzez metodę update(). Zazwyczaj przy konfiguracji kontrolera zadaje się częstotliwość jego pracy, złącza, których zmienne ma on regulować, jego zmienne interfejsu, które otrzymuje z symulowanych, bądź rzeczywistych czujników (np. położenie z enkoderów), oraz nastawy regulatora dla poszczególnych złącz robota. Menedżer kontrolerów jest również głównym punktem komunikacji z użytkownikiem poprzez serwisy ROS2.

Menedżer zasobów (ang. Resource Manager) podlega menedżerowi kontrolerów i jest odpowiedzialny za zarządzanie stanem elementów typu hardware zadeklarowanych wewnątrz tagu <ros2_control> w pliku URDF robota. Mogą być to czujniki, napędy, elementy przełożenia napędu, oraz najczęściej złącza. Wewnątrz głównej pętli regulacji komunikacja z hardwarem realizowana jest przy pomocy metod read() oraz write() menedżera zasobów.

```
<ros2_control name="MultimodalGripper" type="actuator">
  <hardware>
    <plugin>ros2_control_demo_hardware/MultimodalGripper</plugin>
  </hardware>
  <joint name="parallel_fingers">
    <command_interface name="position">
      <param name="min">0</param>
      <param name="max">100</param>
    </command_interface>
    <state_interface name="position"/>
  </joint>
  <gpio name="suction">
    <command_interface name="suction"/>
    <state_interface name="suction"/>    <!-- Needed to know current state of the output -->
  </gpio>
</ros2_control>
```

Rysunek 8: Przykładowy tag <ros2_contol> zawierający deklaracje hardware'u

Źródło: [15]

Biblioteka ros2_control zawiera domyślnie szereg kontrolerów. Poniżej opisano trzy z nich, które znajdą zastosowanie w niniejszej pracy:

- Joint State Broadcaster: Jest to kontroler publikujący aktualne wartości zmiennych interfejsu robota na tematach /joint_states oraz /dynamic_joint_states.
- Joint Trajectory Controller: Jest to kontroler, który jako dane wejściowe przyjmuje zadane wartości zmiennych pozycyjnych i pożądany czas osiągnięcia tychże wartości. Przy użyciu wybranej reguły interpolacji, oblicza on przebiegi dla kolejnych kroków czasowych ruchu.
- PID Controller: Jest to implementacja regulatora PID w ros2_control. Dopuszcza się różne konfiguracje zmiennych stanu i zmiennych regulowanych.

Integrację ze środowiskiem Gazebo Sim dla ros2_control zapewnia paczka gz_ros2_control, która zawiera odpowiednie wtyczki (ang. plugins), umożliwiające korzystanie ze sterowników ros2_control wewnątrz symulacji.

```
<ros2_control name="GazeboSimSystem" type="system">
  <hardware>
    <plugin>gz_ros2_control/GazeboSimSystem</plugin>
  </hardware>
  <joint name="slider_to_cart">
    <command_interface name="effort">
      <param name="min">-1000</param>
      <param name="max">1000</param>
    </command_interface>
    <state_interface name="position">
      <param name="initial_value">1.0</param>
    </state_interface>
    <state_interface name="velocity"/>
    <state_interface name="effort"/>
  </joint>
</ros2_control>
```

Rysunek 9: Przykład tagu <ros2_control> wykorzystującego wtyczkę z gz_ros2_control

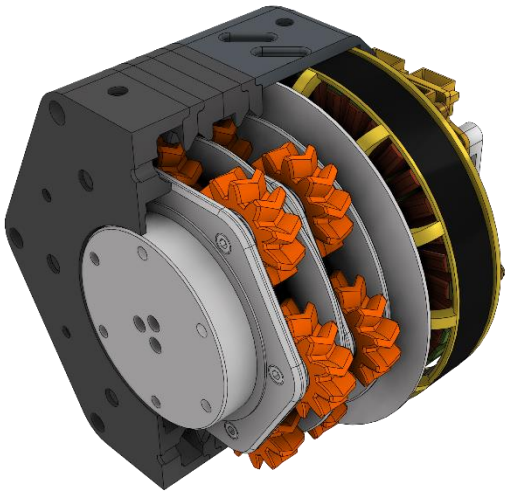
Źródło: [15]

3. Struktura robota Meldog

W tym rozdziale opisano dokładnie ogólną strukturę mechaniczną robota, objaśniając najważniejsze parametry dotyczące zakresów ruchu, prędkości, oraz maksymalnych wartości momentów w złączach. Objaśniono najistotniejsze problemy związane z modelowaniem, którymi obarczony jest niniejszy projekt.

Rysunek 10: Robot Meldog

Robot składa się z 12 członów ruchomych oraz korpusu. Każda z nóg robota posiada trzy stopnie swobody, dwa odpowiedzialne za obrót w kolanie oraz biodrze, jeden za ruch odwodzenia nogi od korpusu. Każdy stopień swobody robota napędzany jest przez osobną jednostkę napędową, której przekrój przedstawiono poniżej.



Rysunek 11: Jednostka napędowa Meldoga

Jednostka wykorzystuje dwustopniową przekładnię planetarną, która przenosi na jej wyjście moment generowany przez silnik BLDC Eaglepower 8308. Obsługa silnika odbywa się poprzez kontroler Moteus r4.11 i jego dedykowane oprogramowanie. Parametry charakteryzujące jednostki napędowe Meldoga przedstawiono w tabeli poniżej:

Tabela 1: Parametry charakteryzujące jednostki napędowe Meldoga

Maksymalny moment chwilowy	34.895 Nm
Maksymalny moment stały	23.928 Nm
Maksymalna prędkość obrotowa	180.542 RPM
Przełożenie przekładni	9.97

W każdej z nóg robota znajdują się dwie jednostki napędzające staw kolana oraz biodra. Jednostki odpowiedzialne za ruch odwodzenia nóg zamocowane są w korpusie. Przeniesienie napędu z wyjścia jednostki na staw biodra odbywa się poprzez dwa koła zębate: jedno napędzające, przymocowane do wyjścia przekładni, drugie napędzane, zamontowane na sztywno do uda robota. Napęd stawu kolana realizowany jest przy pomocy paska zębatego, przenoszącego moment z drugiej jednostki napędowej, zamontowanej w osi obrotu stawu biodrowego. Mechanizm przeniesienia napędu w nodze przedstawiono poniżej:

Rysunek 12: Mechanizm napędu stawów w nodze robota

Przed przystąpieniem do tworzenia plików URDF Meldoga, odpowiednio przeanalizowano jego strukturę w celu wyszczególnienia potencjalnych trudności w projekcie.

Szczególnie problematycznym elementem okazał się być mechanizm napędu stawu kolanowego. Jako, że format URDF nie umożliwia modelowania zamkniętych łańcuchów kinematycznych, wiarygodne zamodelowanie paska zębatego było niemożliwe.

Format URDF oferuje dodatkowy typ złącz „mimic", naśladujących stowarzyszony z nimi staw. Niestety, funkcjonalność ta nie jest wspierana przez część programów, dla których przeznaczony jest cyfrowy bliźniak Meldoga. Zdecydowano się zasymulować napęd przyłożony bezpośrednio w stawie kolanowym. Takie rozwiązanie znacznie uprościło strukturę projektu, nie wprowadzając znacznych niedokładności symulacji.

Mechanizm przełożenia napędu poprzez pasek zębaty wprowadza jednak jedną charakterystyczną zależność do wzajemnego ruch członów w nodze. Przy obrocie członem uda pasek musi obracać się wraz z nim, co powoduje sprzężenie ruchu obrotowego w złączu kolanowym, oraz biodrowym.

W podobny sposób uproszczono mechanizm napędu stawu biodrowego. Zrezygnowano z symulacji ruchu kół zębatych i przyjęto model, w którym napęd w złączu biodrowym przyłożony jest do niego bezpośrednio. Takie rozwiązanie pozwoliło dalej uprościć kod projektu, bez wprowadzania znacznego błędu symulacji.

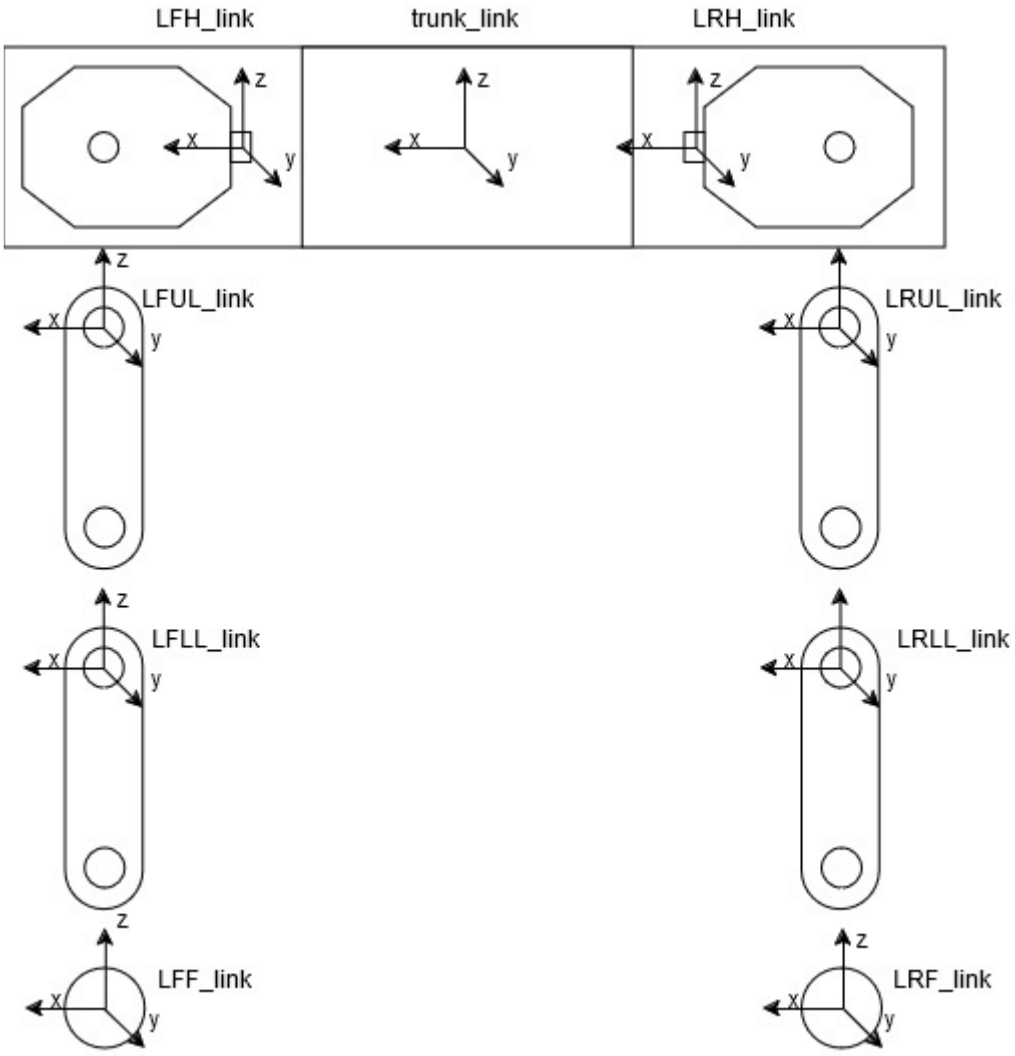
Warto zaznaczyć, że takie rozwiązanie powyższych problemów nie wymaga wprowadzania żadnych zmian w parametrach inercji robota, a ewentualny błąd związany z pomijaniem zwiększenia inercji obracających się elementów jest pomijalnie mały dla części o tak niskich masach.

Dodatkowo zdecydowano się na wyróżnienie stóp robota jako odrębnych członów, by móc łatwiej monitorować ich położenie oraz wygodnie kontrolować trajektorię ich ruchu.

Do zdefiniowania struktury robota w pliku URDF przyjęto następujące założenia:

- Lokalne układy współrzędnych członów zostały umieszczone w złączach.
- Lokalny układ współrzędnych korpusu został umieszczony w jego środku masy, który pokrywał się ze środkiem geometrycznym.
- Obrót tych samych stawów w każdej z nóg robota miał powodować ten sam typ ruchu (dodatni obrót w stawie łączącym biodro z korpusem miał powodować odwodzenie nogi)
- Nazewnictwo poszczególnych członów i złącz w nogach robota uproszczono do form skrótowych.

Poniżej przedstawiono diagram struktury kinematycznej robota z podziałem na człony oraz złącza według wyżej wymienionych zasad. Pary kinematyczne rozłączono w celu poprawienia czytelności rysunku.



Rysunek 13: Diagram struktury Meldoga

Dalej opisano reguły nazewnictwa przyjęte dla opisu modelu:

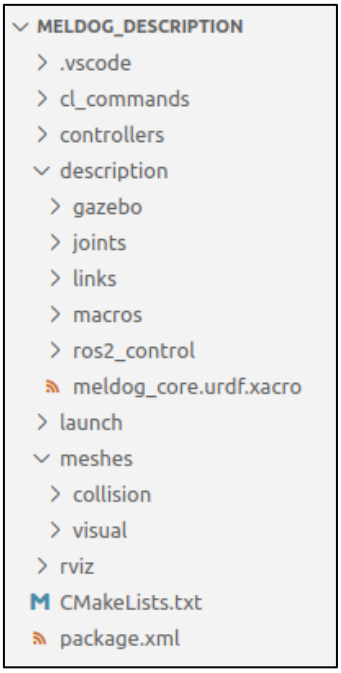
- Pierwsza litera skrótu określa, czy człon lub złącze należy do lewej czy prawej kończyny (L - left lub R - right).
- Druga litera określa, czy człon lub złącze należy do przedniej, czy tylnej kończyny (F - front lub R - rear)
- Trzecia (bądź trzecia i czwarta) litera jest skrótem od rodzaju członu lub złącza. Dla członów wyróżnia się: H – hip (biodro), UL – upper limb (górna kończyna), LL – lower limb (dolna kończyna), F – foot (stopa). Natomiast dla złącz wyróżnia się: T – trunk (staw odwodzący), H – hip (staw biodrowy), K – knee (staw kolanowy), F – foot (łącze stopy i dolnej kończyny robota)

Przykładowo:

- **RFH_joint** – nazwa prawego, przedniego złącza biodrowego
- **LRUL_link** – nazwa lewego, tylnego członu górnej kończyny

4. Pakiet meldog_description

Całość kodu związanego z modelem robota Meldog, oraz jego obsługą została zawarta w pojedynczym pakiecie ROS2 o nazwie meldog_description. W następnych podrozdziałach szczegółowo opisano jego strukturę, oraz proces powstawania.



Rysunek 14: Drzewo katalogów pakietu meldog_description

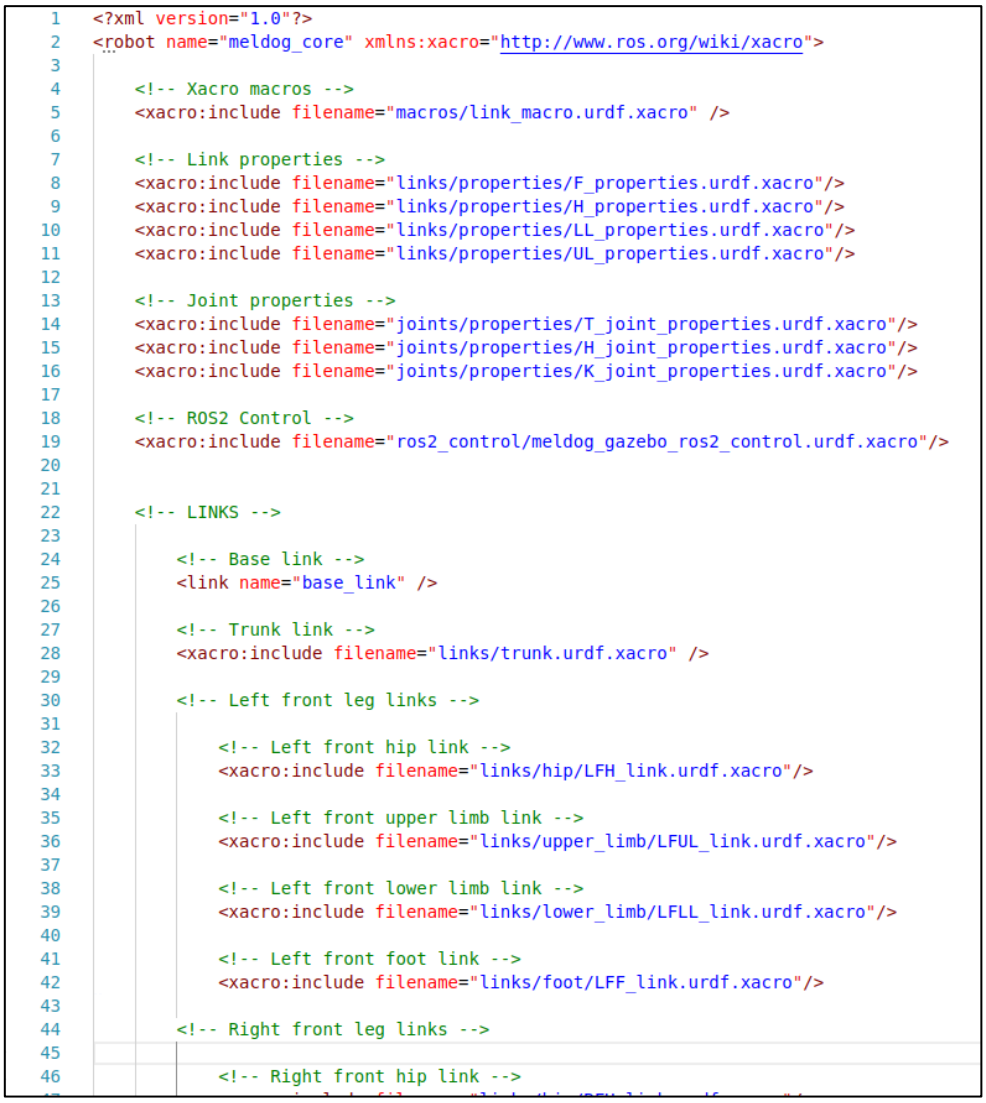
Plik URDF Meldoga w całości zawarto w folderze description, którego strukturę opisano szczegółowo w następnym podrozdziale.

4.1. Struktura folderu description

W celu zwiększenia czytelności kodu, oraz zachowania odpowiedniej organizacji w projekcie, plik URDF Meldoga rozbito na mniejsze pliki oraz katalogi, pełniące odpowiednie role. Dalej opisano poszczególne elementy folderu description.

4.1.1. Główny plik URDF meldog_core.urdf.xacro

Główny URDF robota zawiera odniesienia do wszystkich pomniejszych plików w projekcie. Na początku kodu zawarte są odniesienia do plików związanych z symulatorem Gazebo, pakietem ros2_control, oraz właściwościami złącz i członów. Dalej dołączono deklaracje poszczególnych elementów robota, podzielonych ze względu na to, do której kończyny należą. W celu zachowania przejrzystości kodu, deklaracje każdego z członów i złącz zawarto w osobnych plikach w folderach joints oraz links.



```xml
<?xml version="1.0"?>
<robot name="meldog_core" xmlns:xacro="http://www.ros.org/wiki/xacro">

    <!-- Xacro macros -->
    <xacro:include filename="macros/link_macro.urdf.xacro" />

    <!-- Link properties -->
    <xacro:include filename="links/properties/F_properties.urdf.xacro"/>
    <xacro:include filename="links/properties/H_properties.urdf.xacro"/>
    <xacro:include filename="links/properties/LL_properties.urdf.xacro"/>
    <xacro:include filename="links/properties/UL_properties.urdf.xacro"/>

    <!-- Joint properties -->
    <xacro:include filename="joints/properties/T_joint_properties.urdf.xacro"/>
    <xacro:include filename="joints/properties/H_joint_properties.urdf.xacro"/>
    <xacro:include filename="joints/properties/K_joint_properties.urdf.xacro"/>

    <!-- ROS2 Control -->
    <xacro:include filename="ros2_control/meldog_gazebo_ros2_control.urdf.xacro"/>


    <!-- LINKS -->

        <!-- Base link -->
        <link name="base_link" />

        <!-- Trunk link -->
        <xacro:include filename="links/trunk.urdf.xacro" />

        <!-- Left front leg links -->

            <!-- Left front hip link -->
            <xacro:include filename="links/hip/LFH_link.urdf.xacro"/>

            <!-- Left front upper limb link -->
            <xacro:include filename="links/upper_limb/LFUL_link.urdf.xacro"/>

            <!-- Left front lower limb link -->
            <xacro:include filename="links/lower_limb/LFLL_link.urdf.xacro"/>

            <!-- Left front foot link -->
            <xacro:include filename="links/foot/LFF_link.urdf.xacro"/>

        <!-- Right front leg links -->

            <!-- Right front hip link -->
```

Rysunek 15: Fragment głównego pliku URDF

4.1.2. Deklaracje członów oraz złącz

Aby ujednolicić sposób deklaracji elementów struktury robota, napisano odpowiednie macro dla członów i umieszczono je w odpowiednim folderze o nazwie macros.

Macro link_macro jako argumenty wejściowe przyjmuje nazwę członu, siatki geometrii dla tagu visual, oraz collision w formacie STL, a także wszystkie parametry bezwładnościowe. W późniejszych etapach pracy dodano również argument, który określa, czy deklarowany człon jest stopą robota. Rozróżnienie tego wariantu jest możliwe dzięki wykorzystanej wewnątrz macra instrukcji warunkowej, zapewnionej przez xacro.

```
<xacro:macro name="link_macro" params="name is_foot vis_name coll_name m cm_x cm_y cm_z ixx ixy ixz iyy iyz izz">

    <link name="${name}">
        <visual>
            <geometry>
                <mesh filename="package://meldog_description/meshes/visual/${vis_name}"
                      scale="0.001 0.001 0.001" />
            </geometry>
            <origin xyz="0 0 0" rpy="0 0 0"/>
        </visual>

        <xacro:if value="${is_foot == 1}">
            <collision>
                <geometry>
                    <sphere radius="${foot_radius}" />
                </geometry>
                <origin xyz="0 0 0" rpy="0 0 0" />
            </collision>
        </xacro:if>

        <xacro:if value="${is_foot == 0}">
            <collision>
                <geometry>
                    <mesh filename="package://meldog_description/meshes/collision/${coll_name}"
                        scale="0.001 0.001 0.001" />
                </geometry>
                <origin xyz="0 0 0" rpy="0 0 0" />
            </collision>
        </xacro:if>

        <inertial>
            <mass value="${m}" />
            <origin xyz="${cm_x*0.001} ${cm_y*0.001} ${cm_z*0.001}" rpy="0 0 0" />
            <inertia ixx="${ixx*0.000001}" ixy="${ixy*0.000001}" ixz="${ixz*0.000001}"
                                           iyy="${iyy*0.000001}" iyz="${iyz*0.000001}"
                                                                 izz="${izz*0.000001}" />
        </inertial>

    </link>

</xacro:macro>
```

Rysunek 16: Macro dla deklaracji członu robota

Według przyjętej konwencji lokalne układy współrzędnych wszystkich plików STL robota umieszczono w złączach, zgodnie z rysunkiem Rysunek *13*, dlatego macro nie wymaga definiowania współrzędnych origin deklarowanej geometrii w tagach visual oraz collision. Masę poszczególnych elementów, oraz finalne parametry bezwładnościowe członów wyliczono w programie Inventor, korzystając z danych wyczytanych ze slicera dla części drukowanych, oraz z katalogów producenta dla części kupnych.

```
<xacro:link_macro
    name="LFH_link"
    is_foot="0"
    vis_name="${hip_vis_stl_name}"
    coll_name="${hip_coll_stl_name}"

    m="${hip_mass}"
    cm_x="${hip_cm_x}"
    cm_y="${hip_cm_y}"
    cm_z="${hip_cm_z}"

    ixx="${hip_ixx}" ixy="${hip_ixy}" ixz="${hip_ixz}"
                     iyy="${hip_iyy}" iyz="${hip_iyz}"
                                      izz="${hip_izz}"/>
```

Rysunek 17: Przykładowa deklaracja członu biodra

Wszystkie wyliczone parametry związane z osobnymi członami robota umieszczono w osobnych plikach wewnątrz katalogu properties.

```
<!-- meshes -->
<xacro:property name="lower_limb_vis_stl_name" value="LL_vis_v2.stl" />
<xacro:property name="lower_limb_coll_stl_name" value="lower_limb_collision.stl" />

<!-- Limb dimensions -->
<xacro:property name="lower_limb_length_to_foot" value="0.25" />

<!-- inertial properties -->

    <!-- mass -->
    <xacro:property name="lower_limb_mass" value="0.079"/>

    <!-- center of mass -->
    <xacro:property name="lower_limb_cm_x" value="0.801"/>
    <xacro:property name="lower_limb_cm_y" value="0"/>
    <xacro:property name="lower_limb_cm_z" value="-139.836"/>

    <!-- moments of inertia -->
    <xacro:property name="lower_limb_ixx" value="598.222"/>
    <xacro:property name="lower_limb_ixy" value="0.0"/>
    <xacro:property name="lower_limb_ixz" value="-2.447"/>

    <xacro:property name="lower_limb_iyy" value="603.088"/>
    <xacro:property name="lower_limb_iyz" value="0.0"/>

    <xacro:property name="lower_limb_izz" value="13.771"/>
```

Rysunek 18: Przykładowy plik z parametrami członu dolnej kończyny

Sposób deklaracji złącz w pliku URDF jest na tyle prosty, że zdecydowano się nie definiować osobnego macra dla tego celu. Deklaracje wszystkich złączy robota, podobnie jak w przypadku członów, umieszczono w odrębnym folderze joints. Parametry poszczególnych złącz zawarto w osobnych plikach w folderze properties umieszczonym wewnątrz katalogu złącz.

Strukturę katalogów links oraz joints przedstawiono na rysunku poniżej.

Rysunek 19: Struktura katalogów joints oraz links

4.1.3. Integracja z Gazebo

Wszystkie elementy URDF stowarzyszone z Gazebo umieszczono w odpowiednim folderze dzielącym nazwę z symulatorem. Dwa pliki zawarte w rzeczonym folderze zawierają makra oraz parrametry związane z symulacją fizyczną członów oraz złącz robota.

Plik meldog_gazebo.urdf.xacro zawiera definicje dwóch instrukcji macro związanych z symulacją członów oraz złączy, oraz deklaracje tagów gazebo poprzez odpowiednie macro dla wszystkich elementów robota.

```
<xacro:macro name="gz_link" params="name mu1 mu2 kp kd maxVel dampingFactor">

    <gazebo reference="${name}_link">
        <visual>
            <material>
                <diffuse>0 0 1 1 </diffuse>
            </material>
        </visual>
        <mu1 value="${mu1}"/>
        <mu2 value="${mu2}"/>
        <kp value="${kp}" />
        <kd value="${kd}" />
        <maxVel value="${maxVel}"/>
        <dampingFactor value="${dampingFactor}"/>
    </gazebo>

</xacro:macro>

<xacro:macro name="gz_joint" params="name isd s_stiff s_ref stopErp stopCfm">

    <gazebo reference="${name}_joint">
        <implicitSpringDamper>${isd}</implicitSpringDamper>
        <springStiffness>${s_stiff}</springStiffness>
        <springReference>${s_ref}</springReference>
        <stopErp>${stopErp}</stopErp>
        <stopCfm>${stopCfm}</stopCfm>
    </gazebo>

</xacro:macro>
```

Rysunek 20: Instrukcje macro związane z symulacją członów oraz złączy w Gazebo

Macro gz_link przyjmuje jako argumenty skrótową nazwę członu oraz wartości parametrów związanych z jego symulacją. Znaczenie parametrów objaśniono poniżej [16].

- mu1, mu2 – Są to definiujące wspólczynniki tarcia na głównych kierunkach powierzchni kontaktu używane w silniku fizycznym ODE. W większości przypadków ich wartości są równe sobie.
- kp, kd – Parametry definiujące odpowiednio sztywność oraz tłumienie kontaktowe w silniku fizycznym ODE.
- maxVel – Parametr definiujący maksymalną prędkość ciał możliwą do osiągnięcia poprzez zderzenie.
- dampingFactor – Parametr określający tempo samoistnego wytracania prędkości przez ciało

Dodatkowo macro definiuje kolor członów poprzez tag visual.

Macro gz_joint przyjmuje jako argumenty skrótową nazwę opisywanego złącza oraz wartości parametrów związanych z jego symulacją. Znaczenie parametrów objaśniono poniżej [16].

- implicitSpringDamper – Parametr typu bool określający sposób symulowania tłumienia w złączy w silniku ODE. Określa on, czy silnik ma wykorzystać parasmetry Erp oraz Cfm.
- springStiffness – Parametr określający stałą sprężystości w symulowanym członie
- springReference – Parametr określający położenie równowagi dla członu z symulowaną siłą sprężystości
- stopErp, stopCfm – Parametry określające wartości maksymalne odpowiednio dla zredukowanej siły w złączu (Cfm), oraz dla redukcji błędu symulacji (Erp) .

Warto zaznaczyć, że wartości wyżej wymienionych parametrów złączy nie były znane na etapie tworzenia pliku URDF robota. Ich wyznaczenie zostało opisane w późniejszym rozdziale niniejszej pracy, związanym z minimalizacją błędu odwzorowania rzeczywistego ruchu w złączach.

Parametry xacro dla członów i stawów zadeklarowano w pliku gazebo_properties.urdf.xacro. Celem poprawy dokładności symulacji kontaktu stopy z podłożem, parametry członów rozbito na te używane przez stopy, oraz na te stosowane dla wszystkich innych członów.

Należy zaznaczyć, że identyfikacja parametrów kontaktowych dla członów, a w szczególności dla stóp robota jest zagadnieniem nietrywialnym, które wymaga w przyszłości odpowiednich testów i weryfikacji. Na chwilę obecną parametry kontaktowe tarcia ustawiono na wartości zerowe.

```
<!-- Set to 1 to enable gazebo link physic parameters -->
<xacro:property name="use_gz_links" value="1" />

<!-- Set to 1 to enable gazebo joint physic parameters -->
<xacro:property name="use_gz_joints" value="1" />

<!-- Gazebo joint parameters -->
<xacro:property name="implicitSpringDamper_value" value="0" />
<xacro:property name="springStiffness_value" value="0" />
<xacro:property name="springReference_value" value="0" />
<xacro:property name="implicitSpringDamper_value" value="0" />
<xacro:property name="stopErp_value" value="0" />
<xacro:property name="stopCfm_value" value="0" />

<!-- Gazebo link parameters for all links except feet-->
<xacro:property name="mu1_value" value="0" />
<xacro:property name="mu2_value" value="0" />
<xacro:property name="kp_value" value="0" />
<xacro:property name="kd_value" value="0" />
<xacro:property name="maxVel_value" value="0" />
<xacro:property name="dampingFactor_value" value="0" />

<!-- Gazsebo link parameters for feet -->
<xacro:property name="mu1_value_foot" value="0" />
<xacro:property name="mu2_value_foot" value="0" />
<xacro:property name="kp_value_foot" value="0" />
<xacro:property name="kd_value_foot" value="0" />
<xacro:property name="maxVel_value_foot" value="0" />
<xacro:property name="dampingFactor_value_foot" value="0" />
```

Rysunek 21: Deklaracje parametrów symulacji członów i złączy

Parametry use_gz_links oraz use_gz_joints zostały wykorzystane w instrukcji warunkowej i określają, czy należy użyć opisanych wcześniej parametrów symulacji.

```
<!-- If using gazebo link parameters -->

<xacro:if value="${use_gz_links==1}">

    <!-- Trunk link -->
    <xacro:gz_link name="trunk" mu1="${mu1_value}" mu2="${mu2_value}"
                   kp="${kp_value}" kd="${kd_value}"
                   maxVel="${maxVel_value}" dampingFactor="${dampingFactor_value}" />


    <!-- Left front leg -->
    <xacro:gz_link name="LFH" mu1="${mu1_value}" mu2="${mu2_value}"
                   kp="${kp_value}" kd="${kd_value}"
                   maxVel="${maxVel_value}" dampingFactor="${dampingFactor_value}" />

    <xacro:gz_link name="LFUL" mu1="${mu1_value}" mu2="${mu2_value}"
                   kp="${kp_value}" kd="${kd_value}"
                   maxVel="${maxVel_value}" dampingFactor="${dampingFactor_value}" />

    <xacro:gz_link name="LFLL" mu1="${mu1_value}" mu2="${mu2_value}"
                   kp="${kp_value}" kd="${kd_value}"
                   maxVel="${maxVel_value}" dampingFactor="${dampingFactor_value}" />

    <xacro:gz_link name="LFF" mu1="${mu1_value_foot}" mu2="${mu2_value_foot}"
                   kp="${kp_value_foot}" kd="${kd_value_foot}"
                   maxVel="${maxVel_value_foot}" dampingFactor="${dampingFactor_value_foot}" />

    <!-- Right front leg -->
    <xacro:gz_link name="RFH" mu1="${mu1_value}" mu2="${mu2_value}"
```

Rysunek 22: Deklaracje parametrów symulacji członów

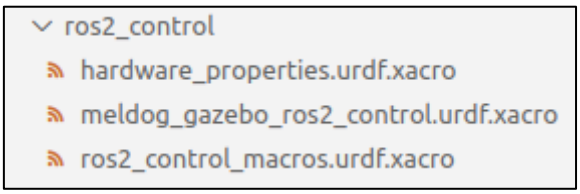
```
<!-- if using gazebo joint parameters -->

<xacro:if value="${use_gz_joints==1}">

    <!-- Left front leg -->
    <xacro:gz_joint name="LFT" isd="${implicitSpringDamper_value}"
                    s_stiff="${springStiffness_value}" s_ref="${springReference_value}"
                    stopErp="${stopErp_value}" stopCfm="${stopCfm_value}" />

    <xacro:gz_joint name="LFH" isd="${implicitSpringDamper_value}"
                    s_stiff="${springStiffness_value}" s_ref="${springReference_value}"
                    stopErp="${stopErp_value}" stopCfm="${stopCfm_value}" />

    <xacro:gz_joint name="LFK" isd="${implicitSpringDamper_value}"
                    s_stiff="${springStiffness_value}" s_ref="${springReference_value}"
                    stopErp="${stopErp_value}" stopCfm="${stopCfm_value}" />

    <!-- Right front leg -->
    <xacro:gz_joint name="RFT" isd="${implicitSpringDamper_value}"
                    s_stiff="${springStiffness_value}" s_ref="${springReference_value}"
                    stopErp="${stopErp_value}" stopCfm="${stopCfm_value}" />
```

Rysunek 23: Deklaracje parametrów symulacji złączy

Celem poprawy czytelności kodu, deklaracje tagów gazebo pogrupowano na podstawie przynależności opisywanego elementu do poszczególnych kończyn. Dodatkowo deklaracje umieszczono wewnątrz dwóch instrukcji warunkowych xacro:if, które na podstawie parametrów use_gz_links oraz use_gz_joints decydują o użyciu parametrów symulacji dla złączy oraz członów.

4.1.4. Elementy pliku URDF związane z ros2_control

Do pliku URDF dodano odpowiednie funkcjonalności związane z pakietem ros2_control, aby umożliwić sterowanie robotem w symulacji. Całość kodu związanego z tymże pakietem umieszczono w podfolderze ros2_control, wewnątrz folderu description.



Rysunek 24: Struktura folderu ros2_control

Kod rozbito na trzy osobne pliki. W pierwszym z nich: hardware_properties zawarto deklaracje parametrów określające maksymalne osiągi jednostki napędowej (Tabela *1*). Parametry te określały, zgodnie z przyjętą konwencją modelowania napędów, maksymalne wartości dla osiąganych momentów oraz prędkości obrotowych w złączach.

```
<?xml version="1.0"?>
<robot xmlns:xacro="http://www.ros.org/wiki/xacro">

    <xacro:property name="motor_max_eff" value="3.5" />
    <xacro:property name="motor_max_vel" value="1800" />
    <xacro:property name="actuator_reduction" value="9.97" />

</robot>
```

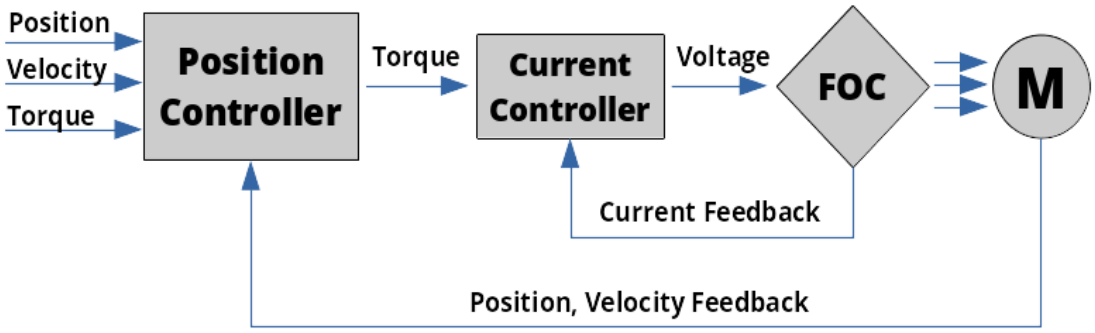
Rysunek 25: Deklaracje parametrów jednostki napędowej

W pliku ros2_control_macros zawarto deklaracje instrukcji macro do tworzenia elementów typu hardware, w tym przypadku złącz oraz elementów przenoszących napęd.

```
<xacro:macro name="ros2_control_joint_macro" params="name pos_min pos_max vel_max eff_max">

    <joint name="${name}_joint">

        <command_interface name="position">
          <param name="min">${pos_min}</param>
          <param name="max">${pos_max}</param>
        </command_interface>

        <command_interface name="velocity" >
          <param name="min">${-vel_max}</param>
          <param name="max">${vel_max}</param>
        </command_interface>

        <command_interface name="effort">
          <param name="min">${-eff_max}</param>
          <param name="max">${eff_max}</param>
        </command_interface>

        <state_interface name="position" />
        <state_interface name="velocity"/>
        <state_interface name="effort"/>

    </joint>

</xacro:macro>
```

Rysunek 26: Instrukcja macro deklarująca element złącza w ros2_control

W rzeczywistych napędach robota zastosowano kontrolory PID sterujące momentem obrotowym silników zgodnie ze schematem przedstawionym poniżej. Szczegóły sterowania rzeczywistymi silnikami przy użyciu odpowiedniego oprogramowania opisano w dalszych rozdziałach niniejszej pracy.



Rysunek 27: Schemat sterowania w rzeczywistych napędach robota

Źródło: [17]

Aby odpowiednio odwzorować taką konfigurację w symulacji, jako zmienne stanu (state_interfaces), pozyskiwane z silników zdefiniowano położenie, prędkość, oraz moment ze złącza. Dla każdego złącza, na potrzeby wstępnych testów oprogramowania, zdefiniowano trzy zmienne sterowane (command_interfaces): pozycję, prędkość, oraz moment, jednak podczas docelowej symulacji wykorzystano jedynie sterowanie momentem.

```
<xacro:macro name="FourBarLinkageTransmission" params="first_joint_name second_joint_name">

    <transmission name="FourBarLinkageTransmission_${first_joint_name}_${second_joint_name}">

        <plugin>transmission_interface/FourBarLinkageTransmission</plugin>
        <actuator name="${first_joint_name}_actuator" role="actuator1">
            <mechanical_reduction>1.0</mechanical_reduction>
        </actuator>
        <actuator name="${second_joint_name}_actuator" role="actuator2">
            <mechanical_reduction>1.0</mechanical_reduction>
        </actuator>
        <joint name="${first_joint_name}_joint" role="joint1">
            <mechanical_reduction>1.0</mechanical_reduction>
            <offset>0.0</offset>
        </joint>
        <joint name="${second_joint_name}_joint" role="joint2">
            <mechanical_reduction>1.0</mechanical_reduction>
            <offset>0.0</offset>
        </joint>

    </transmission>

</xacro:macro>
```

Rysunek 28: Instrukcja macro definiująca przełożenie napędu

Aby odzwierciedlić sprzężenie ruchu między stawami biodrowymi i kolanowymi robota, zdefiniowano macro tworzące element przeniesienia napędu poprzez czworobok przegubowy (ang. four bar linkage). Deklaracja tego elementu dla odpowiednich złączy robota powoduje, że obrót jednego z nich generuje taki sam obrót w drugim [18].

Opisane wyżej makra zastosowano w pliku meldog_gazebo_ros2_control, w którym wewnątrz tagu <ros2_control> zadeklarowano system sterowania robota korzystający z pluginu pochodzącego z pakietu gz_ros2_control, a także umieszczono wszystkie definicje elementów typu hardware.

```
<xacro:include filename="ros2_control_macros.urdf.xacro" />
<xacro:include filename="hardware_properties.urdf.xacro" />
<xacro:include filename="../joints/properties/H_joint_properties.urdf.xacro" />
<xacro:include filename="../joints/properties/K_joint_properties.urdf.xacro" />
<xacro:include filename="../joints/properties/T_joint_properties.urdf.xacro" />

<ros2_control name="Meldog_control" type="system">
    <hardware>
      <plugin>ign_ros2_control/IgnitionSystem</plugin>
    </hardware>

    <!-- joints -->
    <xacro:ros2_control_joint_macro name="LFT" pos_min="${T_lower_limit}" pos_max="${T_upper_limit}" vel_max="${T_velocity}" eff_max="${T_effort}"/>
    <xacro:ros2_control_joint_macro name="LFH" pos_min="${H_lower_limit}" pos_max="${H_upper_limit}" vel_max="${H_velocity}" eff_max="${H_effort}"/>
    <xacro:ros2_control_joint_macro name="LFK" pos_min="${K_lower_limit}" pos_max="${K_upper_limit}" vel_max="${K_velocity}" eff_max="${K_effort}"/>

    <xacro:ros2_control_joint_macro name="RFT" pos_min="${T_lower_limit}" pos_max="${T_upper_limit}" vel_max="${T_velocity}" eff_max="${T_effort}"/>
    <xacro:ros2_control_joint_macro name="RFH" pos_min="${H_lower_limit}" pos_max="${H_upper_limit}" vel_max="${H_velocity}" eff_max="${H_effort}"/>
    <xacro:ros2_control_joint_macro name="RFK" pos_min="${K_lower_limit}" pos_max="${K_upper_limit}" vel_max="${K_velocity}" eff_max="${K_effort}"/>

    <xacro:ros2_control_joint_macro name="LRT" pos_min="${T_lower_limit}" pos_max="${T_upper_limit}" vel_max="${T_velocity}" eff_max="${T_effort}"/>
    <xacro:ros2_control_joint_macro name="LRH" pos_min="${H_lower_limit}" pos_max="${H_upper_limit}" vel_max="${H_velocity}" eff_max="${H_effort}"/>
    <xacro:ros2_control_joint_macro name="LRK" pos_min="${K_lower_limit}" pos_max="${K_upper_limit}" vel_max="${K_velocity}" eff_max="${K_effort}"/>

    <xacro:ros2_control_joint_macro name="RRT" pos_min="${T_lower_limit}" pos_max="${T_upper_limit}" vel_max="${T_velocity}" eff_max="${T_effort}"/>
    <xacro:ros2_control_joint_macro name="RRH" pos_min="${H_lower_limit}" pos_max="${H_upper_limit}" vel_max="${H_velocity}" eff_max="${H_effort}"/>
    <xacro:ros2_control_joint_macro name="RRK" pos_min="${K_lower_limit}" pos_max="${K_upper_limit}" vel_max="${K_velocity}" eff_max="${K_effort}"/>

    <!-- transmission -->
    <xacro:FourBarLinkageTransmission first_joint_name="LFH" second_joint_name="LFK" />
    <xacro:FourBarLinkageTransmission first_joint_name="RFH" second_joint_name="RFK" />
    <xacro:FourBarLinkageTransmission first_joint_name="LRH" second_joint_name="LRK" />
    <xacro:FourBarLinkageTransmission first_joint_name="RRH" second_joint_name="RRK" />

</ros2_control>
```

Rysunek 29: Definicja tagu ros2_control

Oprócz tego, w pliku meldog_gazebo_ros2_control umieszczono wewnątrz tagu <gazebo> deklarację pluginu sterowania, obsługującego sterowniki z architektury ros2_control, a także podano ścieżkę do pliku yaml zawierającego konfiguracją odpowiedniego kontrolera. Dla wstępnych testów oprogramowania zastosowano kontroler joint_trajectory_controller w konfiguracji sterującej pozycją złączy robota.

```
<gazebo>
    <plugin filename="ign_ros2_control-system" name="ign_ros2_control::IgnitionROS2ControlPlugin">
      <!-- Uncomment if using joint trajectory controller -->
      <!-- <robot_param>robot_description</robot_param>
      <robot_param_node>robot_state_publisher</robot_param_node> -->
      <parameters>$(find meldog_description)/controllers/joint_controller.yaml</parameters>
      <!-- <hold_joints>true</hold_joints> -->
    </plugin>
</gazebo>
```

Rysunek 30: Definicja tagu Gazebo z wyborem kontrolera

Pliki konfiguracyjne yaml dla poszczególnych kontrolerów umieszczono w osobnym folderze meldog_controllers. Poniżej przedstawiono przykładowy plik konfiguracyjny dla kontrolera joint_trajectory_controller.

```yaml
controller_manager:
  ros__parameters:
    update_rate: 1000
    use_sim_time: true

    joint_trajectory_controller:
      type: joint_trajectory_controller/JointTrajectoryController

    joint_state_broadcaster:
      type: joint_state_broadcaster/JointStateBroadcaster
```

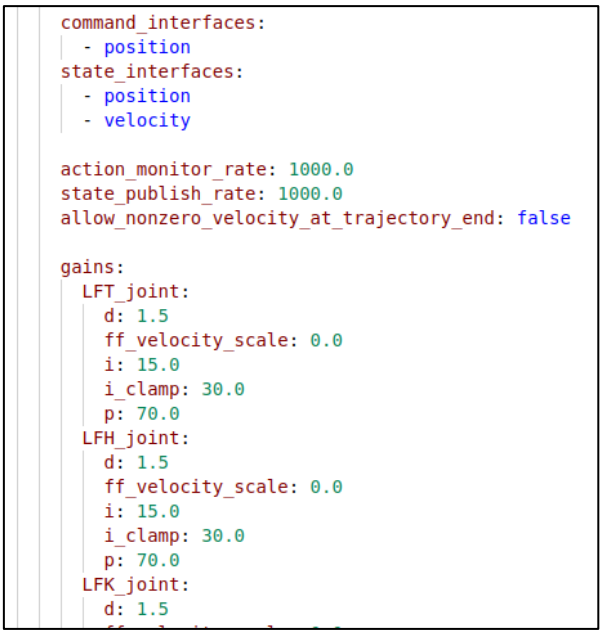
Rysunek 31: Konfiguracja menedżera kontrolerów

Wewnątrz tagu controller_manager zadeklarowane zostały wybrane kontrolery, a także częstotliwość ich pracy, oraz zmienna określająca, czy do ich pracy ma być wykorzystany czas z symulacji. Działanie kontrolerów joint_trajectory_controller oraz joint_state_broadcaster opisano w rozdziale 2.2.4.

```yaml
joint_trajectory_controller:
  ros__parameters:
    joints:
      - LFT_joint
      - LFH_joint
      - LFK_joint
      - RFT_joint
      - RFH_joint
      - RFK_joint
      - LRT_joint
      - LRH_joint
      - LRK_joint
      - RRT_joint
      - RRH_joint
      - RRK_joint

    command_joints:
      - LFT_joint
      - LFH_joint
      - LFK_joint
      - RFT_joint
      - RFH_joint
      - RFK_joint
      - LRT_joint
      - LRH_joint
      - LRK_joint
      - RRT_joint
      - RRH_joint
      - RRK_joint
```

Rysunek 32: Wybór sterowanych złączy

W następnej części pliku skonfigurowano odpowiednio sterownik joint_trajectory_controller, wybierając złącza, oraz określając, które z nich są złączami sterowanymi. W tym przypadku wszystkie wybrane złącza zostały wybrane jako złącza sterowane.



Rysunek 33: Wybór zmiennych oraz ustawienia nastaw kontrolera

W ostatnim fragmencie pliku zadeklarowano zmienne sterowane (command_interfaces), oraz zmienne stanu (state_interfaces), a także ustawiono odpowiednie częstotliwości publikowania tychże zmiennych na odpowiednich tematach ROS2. Dla każdego złącza ustawiono również odpowiednie wartości nastaw kontrolera.

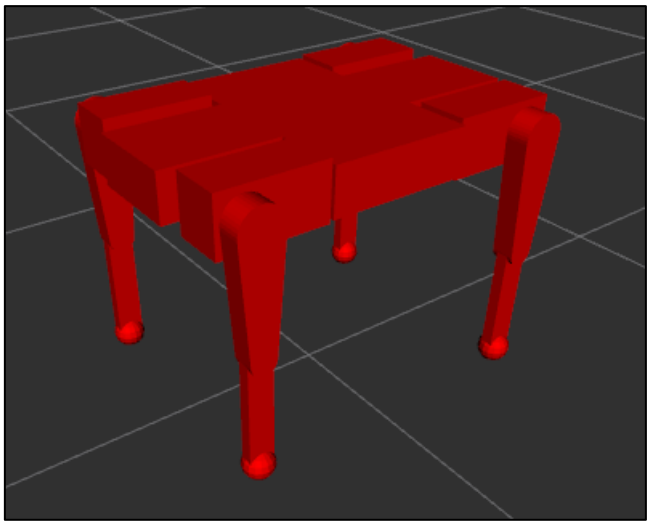
Warto zaznaczyć, że kontroler pracujący w trybie sterowania pozycją, nie wykorzystuje ustawionych nastaw. Potrzebne są one jednyne wtedy, gdy kontroler pracuje w konfiguracji PID, czyli przy sterowaniu momentem.

4.2. Folder Meshes

Wszystkie pliki siatek geometrii STL z pakietu meldog_description umieszczono wewnątrz folderu meshes. Folder podzielono również na dwa podkatalogi dla siatek wizualnych, oraz dla uproszczonych siatek geometrii kolizji.

Wszystkie pliki siatek dla tagów <visual> eksportowano z odpowiednich złożeń w programie Autodesk Inventor. W celu zapewnienia płynności symulacji, modele kończyn robota odpowiednio uproszczono, zachowując jedynie zewnętrzne elementy w złożeniach.

Siatki kolizji robota przygotowano ręcznie w programie Inventor. Starano się zachować możliwie prosty kształt kończyn, wykorzystując jedynie podstawowe bryły geometryczne.

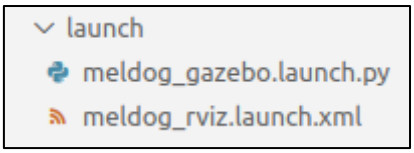


Rysunek 34: Wizualizacja uproszczonych siatek kolizji robota

Układy współrzędnych wszystkich siatek umieszczono w złączach robota, zgodnie z obraną wcześniej konwencją.

4.3. Pliki wywołania, uruchamianie symulacji

Do uruchamiania aplikacji w ROS2 stosuje się pliki wywołania (ang. launch files). Umożliwiają one uruchamianie wielu węzłów, oraz innych funkcjonalności z poziomu jednej komendy terminala.



Rysunek 35: Zawartość folderu launch

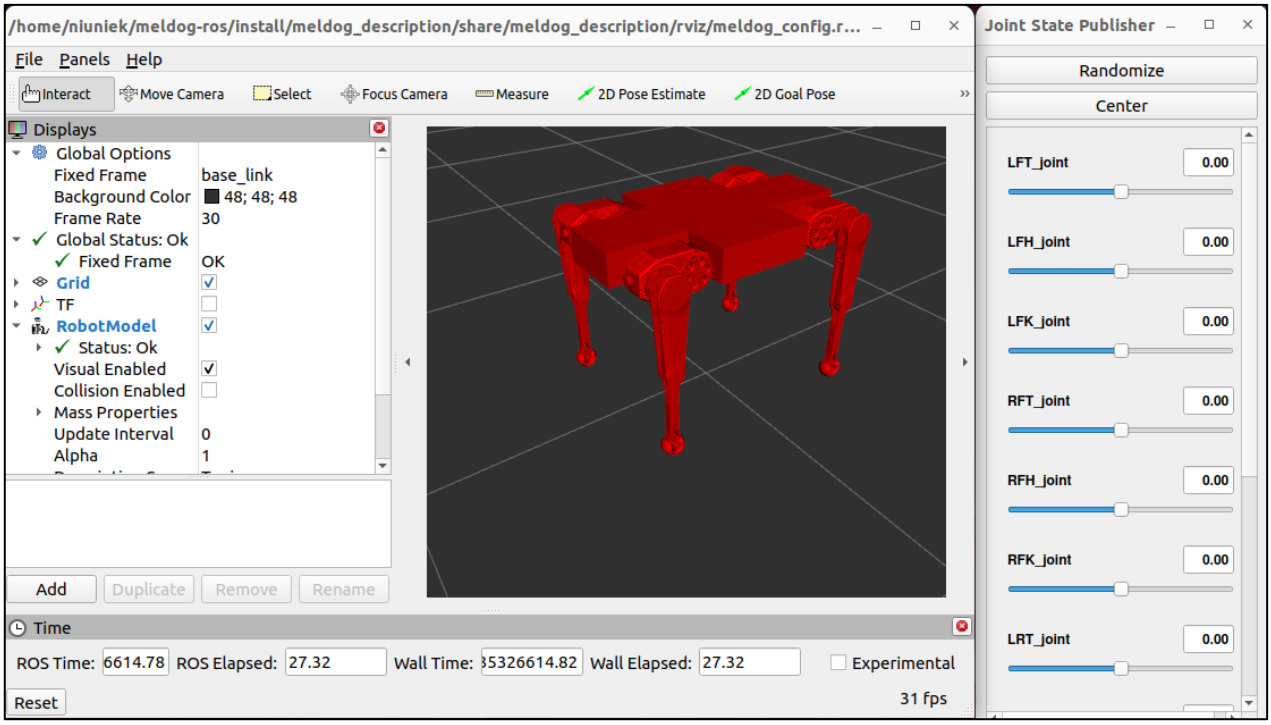
Na potrzeby wizualizacji przy tworzeniu pakietu napisano plik meldog_rviz_launch, który uruchamia jednocześnie następujące węzły:

- **Robot State Publisher –** węzeł publikujący na temacie /robot_description zawartość pliku URDF robota.
- **Joint State Publisher Gui –** węzeł umożliwiający zadawanie położenia w poszczególnych złączach poprzez wygodny interfejs użytkownika.
- **Rviz 2 –** program służący do wizualizacji pliku URDF robota.

```
<launch>

    <let name="urdf_path" value="$(find-pkg-share meldog_description)/description/meldog_core.urdf.xacro" />

    <let name="rviz_config" value="$(find-pkg-share meldog_description)/rviz/meldog_config.rviz" />

    <node pkg="robot_state_publisher" exec="robot_state_publisher">
        <param name="robot_description" value="$(command 'xacro $(var urdf_path)')" />
    </node>

    <node pkg="joint_state_publisher_gui" exec="joint_state_publisher_gui" />

    <node pkg="rviz2" exec="rviz2" output="screen"  args="-d $(var rviz_config)"/>

</launch>
```

Rysunek 36: Plik wywołania wizualizacji robota w Rviz 2

Wizualizację robota w programie Rviz 2 przedstawiono na rysunkach poniżej.



Rysunek 37: Wizualizacja robota w środowisku Rviz 2 oraz interfejs Joint State Publisher

Po zaimplementowaniu całej funkcjonalność związanej z symulatorem Gazebo w pliku URDF robota, napisano odpowiedni plik wywołania do uruchamiania symulacji. Ze względu na jego złożoną strukturę, zdecydowano się napisać go w języku python.

```
def generate_launch_description():

    urdf_path = os.path.join(get_package_share_path('meldog_description'),
                             'description','meldog_core.urdf.xacro')

    robot_description = ParameterValue(Command(['xacro ',urdf_path]),
                                       value_type=str)

    robot_state_publisher_node = Node(
        package="robot_state_publisher",
        executable="robot_state_publisher",
        parameters=[{'robot_description' : robot_description,
                     "use_sim_time" : True}]
    )
```

Rysunek 38: Uruchamianie węzła Robot State Publisher w pliku wywołania symulacji

W pierwszym fragmencie pliku zawarto instrukcje odpowiedzialne za uruchamianie węzła Robot State Publisher, dla którego ustawiono dyrektywę wykorzystania czasu z symulacji. Dodatkowo zadeklarowano parametr dla ścieżki pliku URDF robota.

```
ros_distro = os.environ["ROS_DISTRO"]
physics_engine="" if ros_distro=="humble" else "--physics-engine gz-physics-bullet-featherstone-plugin"

gazebo_resource_path = SetEnvironmentVariable(
    name='GZ_SIM_RESOURCE_PATH',
    value=[str(Path(get_package_share_directory('meldog_description')).parent.resolve())]
)

gazebo = IncludeLaunchDescription(
    PythonLaunchDescriptionSource([os.path.join(
        get_package_share_directory('ros_gz_sim'), 'launch'), '/gz_sim.launch.py']),
        launch_arguments={'gz_args': ['-r -v -v4 empty.sdf'], 'on_exit_shutdown': 'true'}.items()
    )

spawn_entity = Node(package='ros_gz_sim', executable='create',
                    arguments=['-topic', 'robot_description',
                               '-name', 'Meldog'],
                    output='screen')

gz_ros2_bridge = Node(
    package='ros_gz_bridge',
    executable='parameter_bridge',
    arguments=[
        "clock@rosgraph_msgs/msg/Clock[gz.msgs.Clock"
    ]
)
```

Rysunek 39: Deklaracje węzłów odpowiedzialnych za uruchomienie symulatora Gazebo

W dalszej części pliku:

Zawarto fragment kodu odpowiedzialny za ustawienie optymalnego silnika fizyki w zależności od dystrybucji ROS2 użytkownika.

Aby zapewnić prawidłowe wczytywania zasobów przez Gazebo, ustawiono odpowiednio zmienną systemowej GZ_SIM_RESOURCE_PATH, tak aby wskazywała na katalog /share pakietu meldog_description.

Dołączono plik wywołania symulatora Gazebo z załadowanym pustym światem sdf, oraz dodano do kolejki wywołania węzeł create z pakietu ros_gz_sim, odpowiedzialny za wczytanie robota do symulatora poprzez temat /robot_description.

Dołączono węzeł parameter_bridge z pakietu ros_gz_bridge odpowiedzialny za ujednolicenie komunikacji między środowiskami ROS2 i Gazebo Sim.

```
load_joint_state_broadcaster = ExecuteProcess(
    cmd=['ros2', 'control', 'load_controller', '--set-state', 'active',
         'joint_state_broadcaster'],
    output='screen'
)

load_controller = ExecuteProcess(
    cmd=['ros2', 'control', 'load_controller', '--set-state', 'active', 'joint_controller'],
    output='screen'
)

return LaunchDescription([
    RegisterEventHandler(
        event_handler=OnProcessExit(
            target_action=spawn_entity,
            on_exit=[load_joint_state_broadcaster],
        )
    ),

    RegisterEventHandler(
        event_handler=OnProcessExit(
            target_action= load_joint_state_broadcaster,
            on_exit=[load_controller],
        )
    ),
    robot_state_publisher_node,
    gazebo_resource_path,
    gazebo,
    spawn_entity,
    gz_ros2_bridge
```

Rysunek 40: Uruchamianie kontrolerów ros2_control z poziomu pliku wywołania

W ostatniej części pliku dodano instrukcje uruchamiające kontrolery joint_state_broadcaster oraz joint_controller. Dodano również dwie instrukcje RegisterEventHandler, odpowiedzialne za uruchamianie kontrolerów w odpowiedniej kolejności po uprzednim utworzeniu modelu robota w środowisku symulacji.

Uruchomienie symulacji odbywa się z poziomu wiersza poleceń poprzez polecenie „ros2 launch meldog_description meldog_gazebo.launch.py”.

W celu sprawdzenia poprawności oprogramowania przeprowadzono testy z użyciem kontrolera joint_trajectory_controller. Przygotowano odpowiednio komendy ruchu w złączach i zadano je z poziomu wiersza poleceń, najpierw dla kontrolera w konfiguracji bezpośredniego sterowania pozycją w złączach, a następnie dla konfiguracji sterującej momentem z odpowiednio dobranymi nastawami PID.

W pierwszej konfiguracji robot wykonał całość ruchu bez problemu, jednak współczynnik RTF (real time factor) wahał się na poziomie 50%, co nie zgadzało się z założeniem możliwie płynnej symulacji.

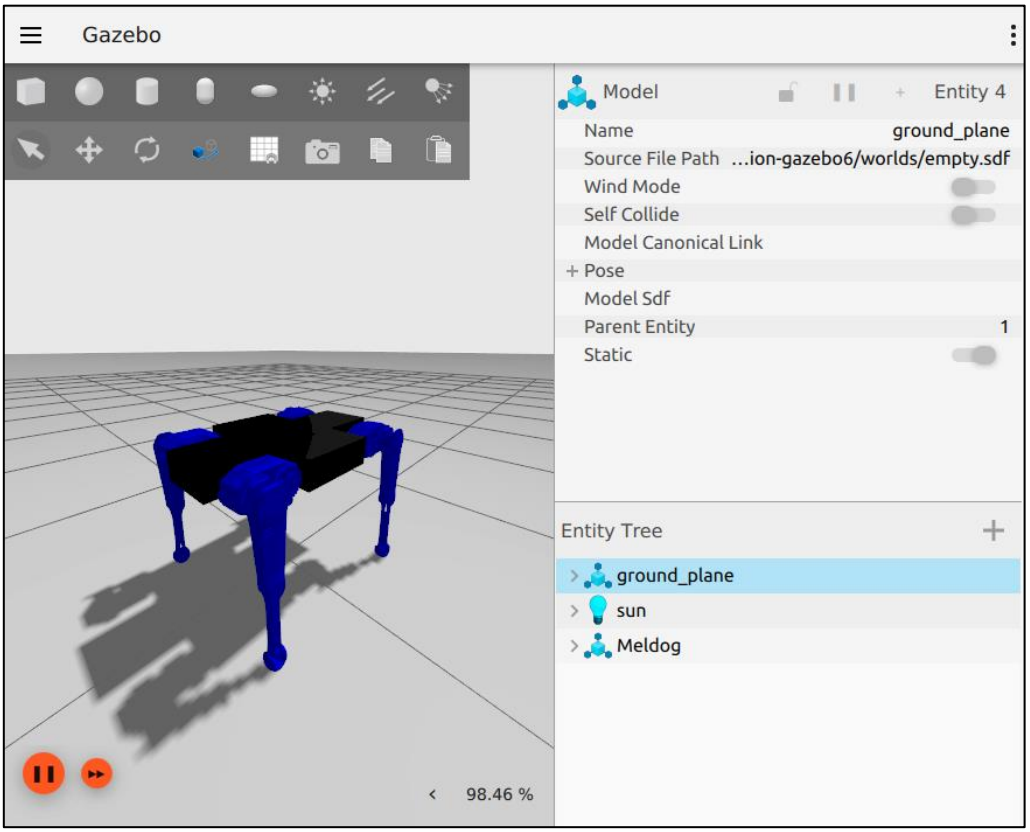
W drugiej konfiguracji kontroler również prawidłowo wykonał zadany ruch, jednak problem niższego współczynnika czasu symulacji pozostał niezmieniony.

Po wnikliwej analizie przebiegu symulacji, odkryto, że problemy z płynnością występowały jedynie podczas styku dolnych kończyn robota z podłożem. Przyczyną problemu okazały się nieoptymalne siatki kolizji dla stóp, które powodowały ogromny wzrost kosztu obliczeń. Po przyjęciu domyślnego elementu sfery z formatu URDF jako siatki kolizji dla stóp, współczynnik czasu symulacji wzrósł do poziomu ponda 98%.

Zauważono również pewien problem podczas kilkukrotnego restartowania symulacji. W pewnych przypadkach ruchy robota po uruchomieniu symulacji stawały się chaotyczne i nieprzewidywalne. Model odlatywał gwałtownie z dużą prędkością w losowym kierunku. Problemem okazały się procesy powiązane z symulacją Gazebo, które pozostawały włączone nawet po zamknięciu klienta symulatora. Rozwiązaniem było wpisanie komendy „killall ruby” do wiersza poleceń.

W trakcie przeprowadzania testów odkryto również problem z kontrolerem PID ze środowiska ros2_control, który zamierzano zaimplementować w docelowym systemie sterowania modelu. Kontroler okazał się działać nieco inaczej niż rzeczywisty PID stosowany w jednostkach napędowych robota. Jego opcje konfiguracyjne nie pozwalały na wierne odwzorowanie funkcjonowania rzeczywistego systemu sterowania Meldoga. Co więcej, jego struktura była niepoprawnie zoptymalizowana i co za tym idzie, nie nadawał się on do pracy z pożądaną częstotliwością.

Zdecydowano się więc na napisanie własnego oprogramowania kontrolera PID do użycia w ros2_control. Jako kryterium oceny pracy kontrolera przyjęto przede wszystkim jego szybkość, oraz możliwie wierne odwzorowanie sposobu pracy regulatora PID z oprogramowania Moteus.



Rysunek 41: Robot w symulacji Gazebo

5. Kontroler PID joint_controller

W celu prawidłowego odwzorowania rzeczywistych mechanizmów sterowania napędami Meldoga, we współpracy z Inż. Bartłomiejem Krajewskim napisano odpowiedni kontroler PID do pracy w środowisku ros2_control [19].

Kontroler odzwierciedla prawo sterowania dla kontrolera Moteus (Rysunek *27*) według odpowiednich zależności:

$$e = \varphi_{des} - \varphi \tag{1}$$

$$e_{vel} = \dot{\varphi_{des}} - \dot{\varphi} \tag{2}$$

,gdzie e – uchyb sterowania, φ_{des} pozycja zadana, φ - aktualna pozycja

$$M = P + I + D + M_{ff} \tag{3}$$

,gdzie M_{ff}- moment feedforward, oraz:

$$P = k_p \cdot e \tag{4}$$

$$I = \int_{t_0}^{t} e \; dt \tag{5}$$

$$D = k_d \cdot e_{vel} \tag{6}$$

,gdzie k_p, k_i, k_d – nastawy regulatora

Wartości aktualnego położenia oraz prędkości pozyskiwane są jako zmienne stanu z tematu /joint_states. Warto zaznaczyć, że aby korzystać z członu różniczkującego, nie trzeba zadać pożądanej prędkości. Kontroler automatycznie wprowadza element tłumienia zależny od wartości k_d.

Kontroler używa odpowiednich metod dyskretyzacji do całkowania błędu położenia.

$$I_k = I_{k-1} + k_i \cdot e_k \cdot dt \tag{7}$$

Kod źródłowy kontrolera został napisany w języku C++. Stworzona została klasa Controller o polach prywatnych określających nastawy regulatora, oraz o metodach wyliczających sterowany moment w kolejnych krokach czasowych.

Do weryfikacji poprawności użytych metod Przygotowano prosty model oscylatora harmonicznego w programie C++. Zadano odpowiednią pozycję i wygenerowano przebiegi trajektorii. Powstałe charakterystyki porównano z regulatorem PID w oprogramowaniu Matlab. Uzyskane przebiegi były identyczne, co pozwoliło stwierdzić poprawność modelu.

Szczegóły implementacji kontrolera w platformie ros2_control nie zostaną opisane, gdyż nie są one istotne dla przedmiotu niniejszej pracy. Warto jednak zaznaczyć, że sposób implementacji pozwolił wiernie odwzorować działanie rzeczywistego kontrolera PID przy jednoczesnym zachowaniu odpowiedniej wydjaności obliczeń, spełniając tym samym wszystkie postawione wymagania.

Wyboru kontrolera dokonuje się z poziomu pliku URDF i konfiguruje się go zgodnie z konwencją opisaną wcześniej w rozdziale 4.1.4.

6. Weryfikacja dokładności opracowanego modelu

Po napisaniu pakietu meldog_description i wstępnym zweryfikowaniu jego poprawności, przystąpiono do strojenia opracowanego modelu, poprzez porównanie uzyskanych w symulacji przebiegów z ich rzeczywistymi odpowiednikami.

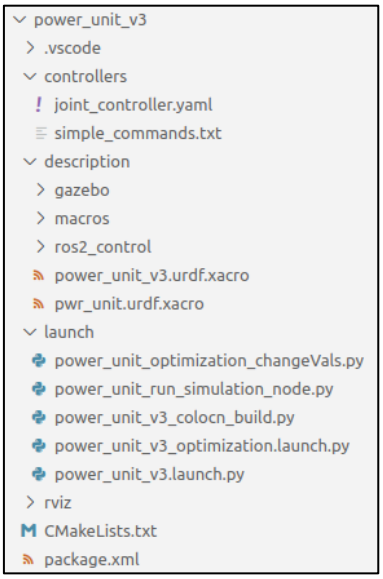
Do tego celu przygotowano osobny pakiet w ROS2 do symulacji jednej jednostki napędowej. Istotne dla dokładności symulacji było właściwe zidentyfikowanie wartości nastaw symulowanego regulatora PID, a także wartości parametrów symulacji określających tarcie oraz tłumienie w złączach.

Na potrzeby testów napisano również pakiet position_nodes, zawierający węzeł zadający porządaną trajektorię dla symulacji pojedynczej jednostki napędowej

Do zadawania przebiegów na rzeczywistej jednostce napędowej przygotowano zestaw skryptów w języku python, korzystających z bibliotyki asyncio

6.1. Pakiet power_unit_v3

Do symulacji pojedynczej jednostki napędowej robota napisano osobny pakiet ROS2 o nazwie power_unit_v3. Struktura pakietu była analogiczna do tej z meldog_description. Kod źródłowy rozbito na pliki związane z konkretną funkcjonalnością.



Rysunek 42: Struktura pakietu power_unit_v3

Aby prawidłowo odwzorować zachowanie rzeczywistej jednostki napędowej przyjęto, że zgodnie z podejściem black box [20], wewnętrzna struktura jednostki nie gra roli w procesie identyfikacji parametrów. Wzięto pod uwagę jedynie sposób oddziaływania jednostki na napędzany człon, analizując charakterystyki pozycji i prędkości napędzanego elementu o znanych parametrach bezwładnościowych.

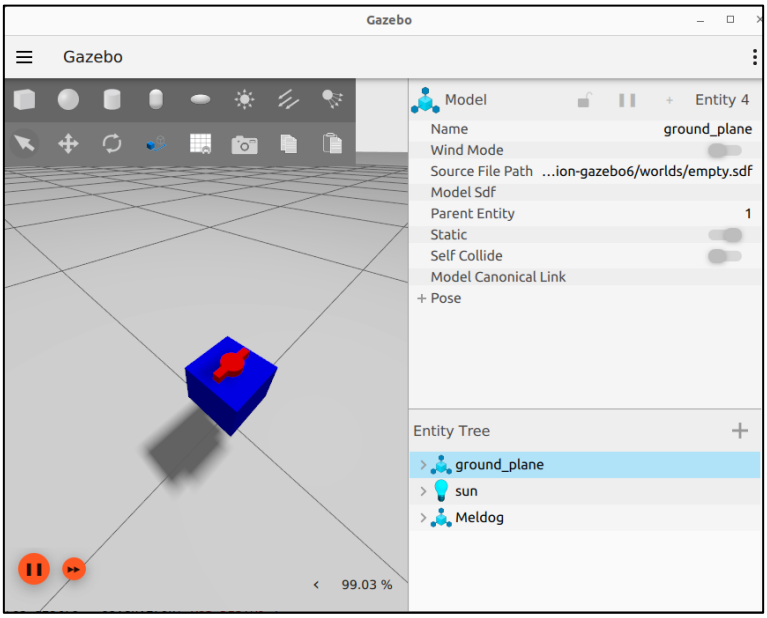
W pliku URDF zamodelowano prostą strukturę: jeden człon stator_link, unieruchomiony poprzez stałe złącze z elementem world, oraz jeden człon napędzany. Jako człon napędzany przyjęto koło zębate napędzające nogę robota, wraz z jego adapterem do osadzania na wyjściu jednostki. Tensor bezwładności element unapędzanego, wartości jego masy, oraz współrzędne jego środka masy

pobrano z programu Inventor. Tym sposobem utrzymano przyjętą wcześniej konwencję uproszczenia napędu w złączach robota.

```
<xacro:include filename="ros2_control/ros2_control.urdf.xacro" />
<xacro:include filename="macros/link_macros.urdf.xacro" />
<xacro:include filename="gazebo/gz_properties.urdf.xacro" />
<xacro:include filename="gazebo/gazebo.urdf.xacro" />
<xacro:property name="body_l" value="0.1" />
<xacro:property name="body_w" value="0.1" />
<xacro:property name="body_h" value="0.086" />
<xacro:property name="shaft_h" value="0.014" />
<xacro:property name="shaft_d" value="0.035" />

<link name="world" />

<xacro:stator_link_macro
length="${body_l}"
width="${body_w}"
height="${body_h}"
mass="10.0"
xc="0" yc="0" zc="${body_h/2.0}"
ixx="${0.17}" ixy="${0.0}" ixz="${0.0}"
iyy="${0.22}" iyz="${0.0}"
izz="${0.25}" />

<xacro:rotor_link_macro
d="${shaft_d}"
length="${shaft_h}"
mass="0.585"
xc="0.0" yc="0.0" zc="11.664"
ixx="56.082" ixy="0.0" ixz="0.0"
iyy="56.082" iyz="0.0"
izz="105.479" />

<joint name="base_joint" type="fixed">
    <parent link="world" />
    <child link="stator_link" />
    <origin xyz="0 0 0.1" rpy="0 0 0" />
</joint>

<joint name="shaft_joint" type="revolute">
    <parent link="stator_link" />
    <child link="rotor_link" />
    <origin xyz="0 0 ${body_h}" rpy="0 0 0" />
    <limit lower="${-1000*pi}" upper="${1000*pi}"
    effort="${motor_max_torque*reduction}" velocity="${motor_max_vel*pi/30/reduction}" />
    <dynamics friction="0.0" damping="0.0" />
    <axis xyz="0 0 1" />
</joint>
```

Rysunek 43: Plik URDF dla jednostki napędowej

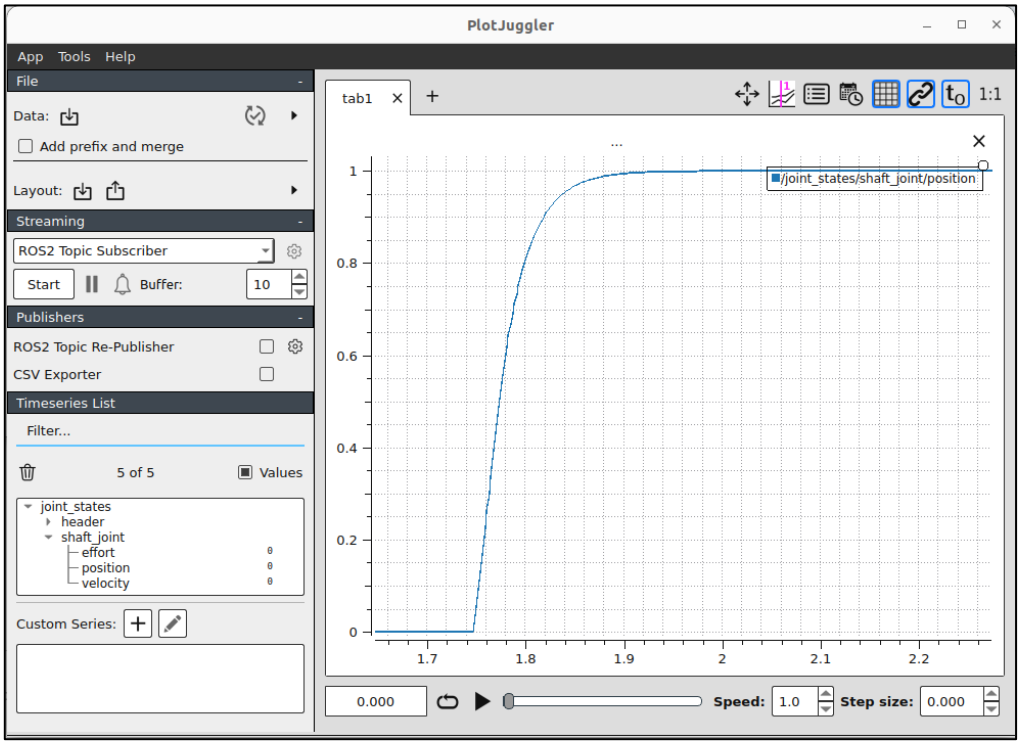
W tagach visual dla obydwu członów wykorzystano domyślne, proste bryły geometryczne z formatu URDF.



Rysunek 44: Jednostka napędowa w środowisku Gazebo

W folderze gazebo pakietu zadeklarowano odpowiednie parametry, opisane wcześniej w rozdziale 4.1.3.

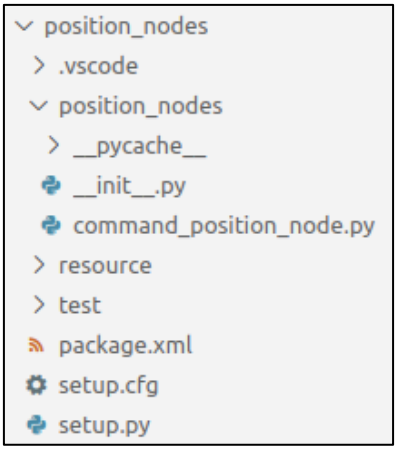
Do sterowania modelem jednostki wykorzystano, opisany w rozdziale 5, kontroler PID joint_controller. Po jego skonfigurowaniu przygotowano proste instrukcje zadające skok jednostkowy położenia i skontrolowano zachowanie kontrolera dla różnych konfiguracji nastaw. Wykresy położenia dla zadanego ruchu sporządzono korzystając z narzędzia PlotJuggler. Na podstawie wyników testów stwierdzono, że kontroler działa poprawnie.



Rysunek 45: Wykres pozycji dla jednostki napędowej przy skoku jednostkowym w konfiguracji PD

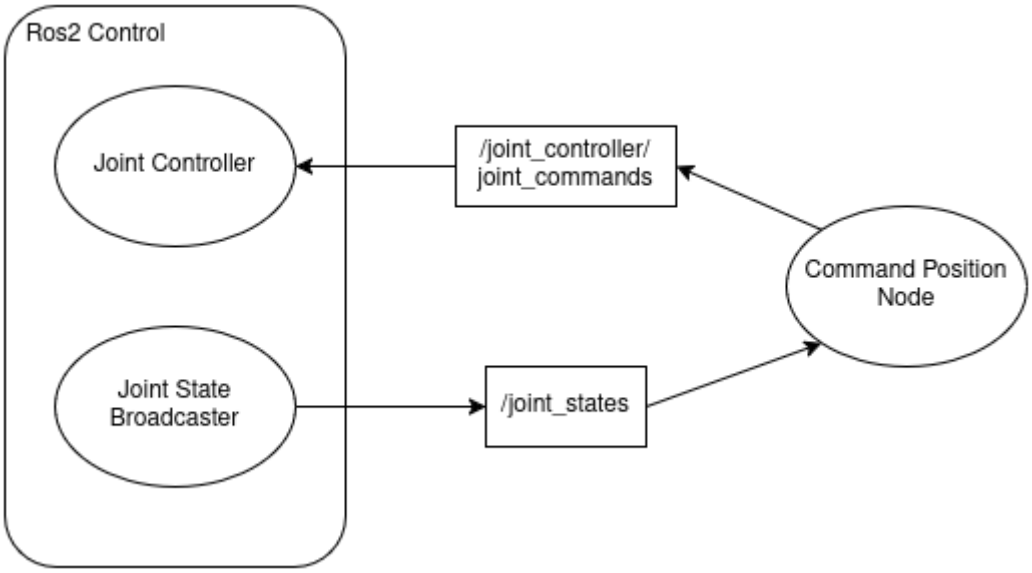
6.2. Pakiet position_nodes

Zadawanie bardziej złożonych trajektorii ruchu wymagało przygotowania osobnego pakietu position_nodes z odpowiednim węzłem command_position_node.



Rysunek 46: Struktura pakietu position_nodes

Pakiet napisano w języku python. Plik command_position_node.py zawiera definicję węzła ROS2, posiadającego subskrybenta (ang. subscriber), pobierającego informacje o aktualnym położeniu i prędkości sterowanego złącza z tematu /joint_states.



Rysunek 47: Schemat komunikacji węzła w środowisku ROS2

Węzeł uruchamiany jest z poziomu terminala, z możliwością wyboru rodzaju przebiegu, oraz zadania wartości charakterystycznych parametrów:

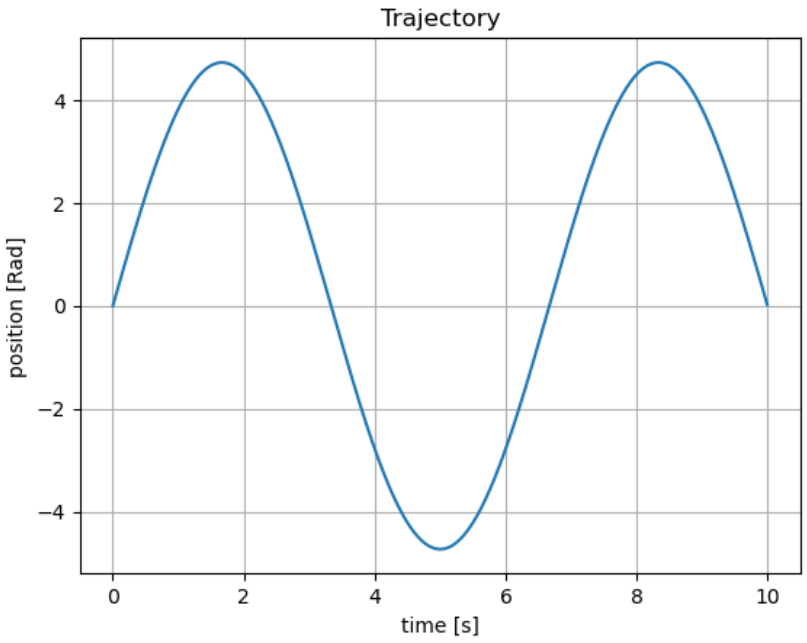
- Częstotliwości pracy
- Maksymalnej amplitudy
- Całkowitego czasu przebiegu
- Czasu wzrastania sygnału oraz czasu szczytu

Węzeł na podstawie wartości zadanych częstotliwości oraz czasu przebiegu tworzy wektory do zapisywania pozycji, prędkości, a także czasu o rozmiarze odpowiadającym ilości kroków czasowych dla całego pomiaru.

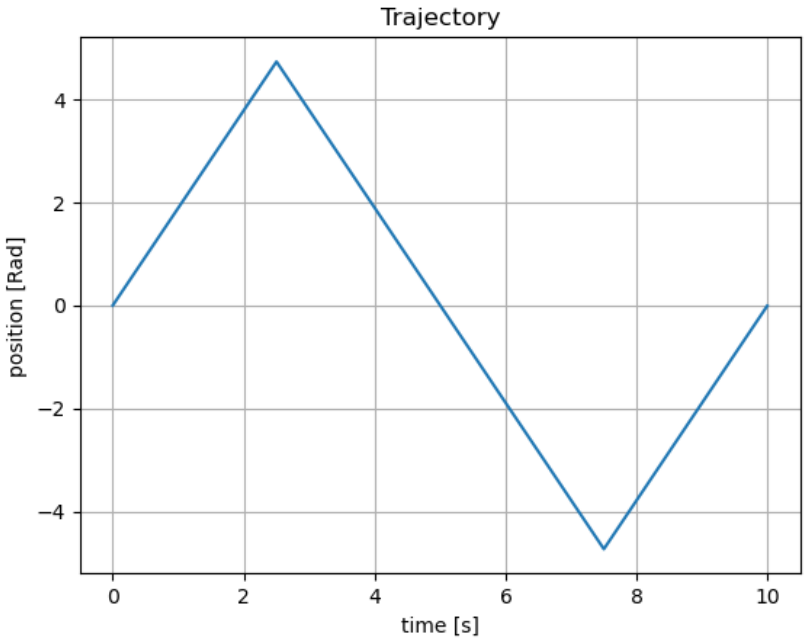
```python
class SinePositionNode(Node):
    def __init__(self):
        super().__init__("sine_position_node")
        self.declare_parameter("motion_type", "sine")
        self.declare_parameter("amplitude", 7.5/9.97*2*math.pi)
        self.declare_parameter("motion_time", 10.0)
        self.declare_parameter("amp_time", 1.)
        self.declare_parameter("peak_time", 0.5)

        self.motion_type_ = self.get_parameter("motion_type").value
        self.amplitude_ = self.get_parameter("amplitude").value
        self.motion_period_ = self.get_parameter("motion_time").value

        self.frequency_ = 200
        if(self.motion_type_=="trapezoid"):
            self.motion_period_ = 4*self.get_parameter("amp_time").value+2*self.get_parameter("peak_time").value

        self.n = int(self.frequency_*self.motion_period_)

        self.time_n_ = np.empty(self.n)
        i = 0

        while(i < self.n):
            self.time_n_[i] = i/self.frequency_
            i+=1

        self.get_logger().info("Initialized time vector!\n")

        self.command_position_ = np.empty(self.n)

        self.joint_position_ = np.empty(self.n)
        self.joint_velocity_ = np.empty(self.n)
```

Rysunek 48: Fragment definicji węzła command_position_node

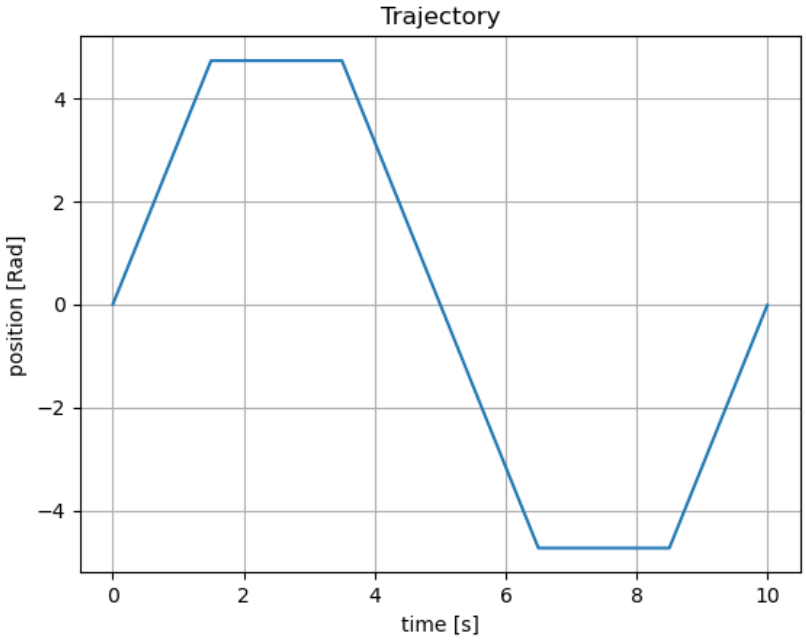
Węzeł po uruchomieniu publikuje z określoną częstotliwością zadaną pozycję na temacie /joint_controller/commands zgodnie z wybraną funkcją przebiegu. Na potrzeby testów przygotowano cztery rodzaje zadanych trajektorii: trapezową, sinusoidalną, skokową, oraz trójkątną.



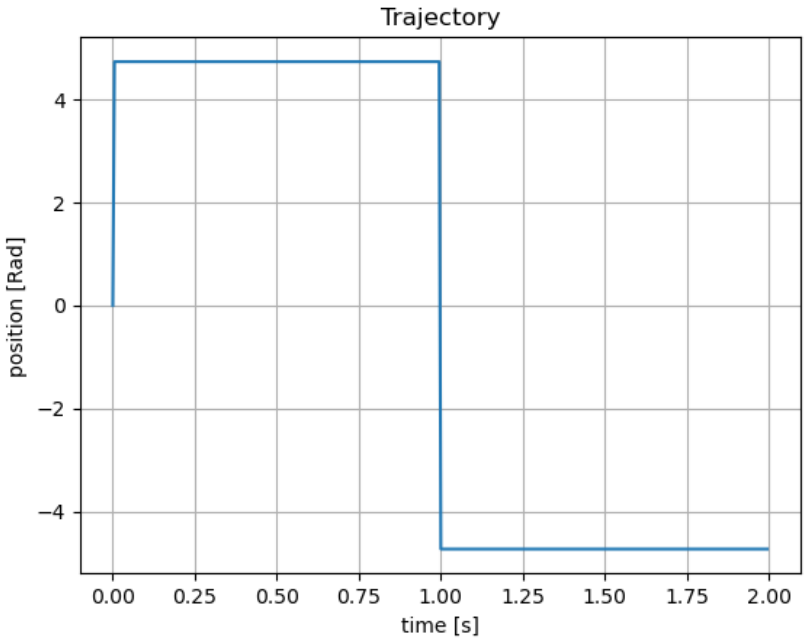
Wykres 1: Przykładowy przebieg sinusoidalny



Wykres 2: Przykładowy przebieg trójkątny



Wykres 3: Przykładowy przebieg trapezowy o czasie narastania 1.5s i czasie szczytu 2s



Wykres 4: Przykładowy przebieg skokowy

```python
def publishPosition(self):
    if(self.i==0):
        self.time_start_ = time.time()
        self.time_vector_[self.i]=0.
        self.command_position_[self.i] = 0.
        self.i+=1
    elif self.i < self.n :
        self.time_vector_[self.i] = time.time()-self.time_start_
        if(self.motion_type_ == "sine"):
            desired_pos = self.amplitude_*math.sin(self.i/self.frequency_)
        elif self.motion_type_ == "square":
            if(self.i<self.n/2):
                desired_pos = self.amplitude_
            else:
                desired_pos = -self.amplitude_
        elif self.motion_type_ == "trapezoid":
            desired_pos = self.trapezoidFunction(self.time_vector_[self.i],self.motion_period_)
        elif self.motion_type_ == "triangle":
            desired_pos = self.triangleFunction(self.time_vector_[self.i],self.motion_period_)
        self.command_position_[self.i] = desired_pos
        self.i+=1

        msg = JointCommand()
        msg.name = ['shaft_joint']
        msg.desired_position = [desired_pos]
        msg.desired_velocity = [0.]
        msg.feedforward_effort = [0.]
        msg.kp_scale = [1.0]
        msg.kd_scale = [1.0]

        self.pos_publisher_.publish(msg)
```

Rysunek 49: Funkcja publikująca zadane położenie według wybranej trajektorii

Po zakończeniu ruchu węzeł zadaje pozycję zerową dla złącza, zapisuje uzyskane przebiegi do pliku .txt w odpowiednim folderze i kończy swoją pracę. Generowane są również wykresy dla pozycji zadanej oraz odpowiedzi złącza w symulacji.

6.3. Skrypty sterowania rzeczywistą jednostką

Do zadawania trajektorii na rzeczywistej jednostce napędowej przygotowano zestaw skryptów python wykorzystujących biblioteki asyncio [21] oraz moteus [17] do asynchronicznej komunikacji z kontrolerem silnika.

Skrypt posiada taką samą funkcjonalność, co węzeł command_position_node. W kolejnych krokach czasowych, nawiązuje on połączenie z silnikiem, zadając odpowiednią wartość położenia, zgodnie z określoną funkcją trajektorii, oraz pobiera aktualne wartości zmiennych stanu, oraz momentu.

Po zakończeniu trajektorii, skrypt kończy swoje działanie i zapisuje przebiegi do pliku .txt w odpowiednim folderze, oraz tworzy wykresy uzyskanych przebiegów.

6.4. Testy na rzeczywistej jednostce napędowej

Po napisaniu niezbędnego oprogramowania przeprowadzono szereg testów na rzeczywistej jednostce napędowej robota, celem porównania symulowanych przebiegów z ich rzeczywistymi odpowiednikami.

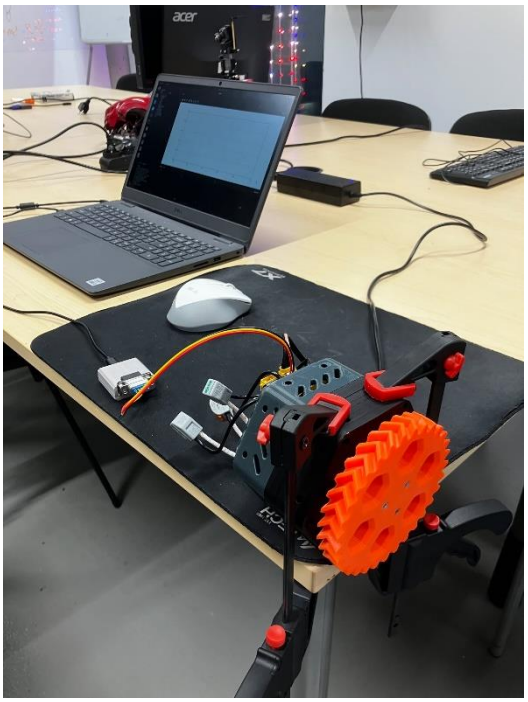
6.4.1. Stanowisko pomiarowe

Stanowisko pomiarowe składało się z:

- Laptopa z przygotowanymi wcześniej skryptami sterowania, oraz zainstalowanymi pakietami oprogramowania Moteus
- Jednostki napędowej Meldoga wyposażonej w kontroler Moteus r4.11
- Dwóch zacisków
- Kabla do komunikacji protokołem CAN-FD do USB
- Zasilacza 24v
- Części napędzanej o znanych parametrach bezwładnościowych

Przed przystąpieniem do testów do jednostki napędowej przymocowano adapter z kołem zębatym. Jednostkę napędową odpowiednio przymocowano do przestrzeni roboczej przy pomocy zacisków.

W pierwszej kolejności laptopa połączono z gniazdem nr. 1 kontrolera poprzez kabel komunikacyjny CAN-FD. Następnie do gniazda zasilania kontrolera podłączono zasilacz i finalnie, po upewnieniu się, że wszystkie elementy stanowiska połączono prawidłowo, włączono zasilacz z listwy.



Rysunek 50: Stanowisko pomiarowe

6.4.2. Badanie rzeczywistych przebiegów położenia

Po przygotowaniu stanowiska przystąpiono do pomiarów. W pierwszej kolejności uruchomiono oprogramowanie t-view i zweryfikowano poprawność konfiguracji sterownika. Zadano próbny ruch do pozycji 10. Silnik wykonał obrót prawidłowo.

Warto zaznaczyć, że oprogramowanie moteus [17] przyjmuje jako położenie wartość obrotów do wykonania, dlatego też w skryptach, oraz innych programach, gdzie przetwarzane dane muszą być przedstawione w radianach należało odpowiednio przeliczyć wartości wyjściowe oraz wejściowe.

Kolejną istotną cechą jednostki napędowej jest umiejscowienie enkodera absolutnego na wale silnika, co w efekcie oznacza, że zliczane są nie obroty wyjścia przekładni, a jej wejścia. Może to powodować pewne rozbieżności pomiędzy rzeczywistymi wartościami obrotu na wyjściu, a tymi wyliczonymi poprzez przemnożenie danych enkodera przez przełożenie przekładni. W takim wypadku znaczący wpływ mogą mieć luzy przekładni.

Testy w symulacji przeprowadzono dla parametrów złącz w Gazebo ustawionych na wartości 0.01. Tarcie oraz tłumienie w złączach wewnątrz tagu <dynamics> ustawiono na wartość zerową.

Podczas pomiarów zadano cztery różne rodzaje trajektorii dla amplitud wychylenia na wejściu przekładni równych 7.5 obrotu, 2.5 obrotu oraz 1 obrotu. Dla charakterystyk skokowych wykonano ruch o mniejszej amplitudzie, aby nie nadwyrężać silnika. Zadano trajektorie:

- Sinusoidalną o czasie ruchu równym 4s i okresie o tej samej wartości
- Trapezową o czasie narastanie równym 1s i czasie szczytu 0.5s
- Trójkątną o czasie ruchu równym 4s i okresie o tej samej wartości
- Skokową o amplitudzie 1.5, 0.75 i 0.25 obrotu

Przed każdym kolejnym pomiarem ustawiano wartość referencyjną położenia zerowego na aktualną pozycję silnika poprzez komendę „python3 -m moteus.moteus_tool --target 1 --zero-offset”. Bez tego kroku silnik usiłował gwałtownie wyzerować swoją pozycję w momencie zadania przebiegu, co skutkowało błędnymi odczytami z enkodera.

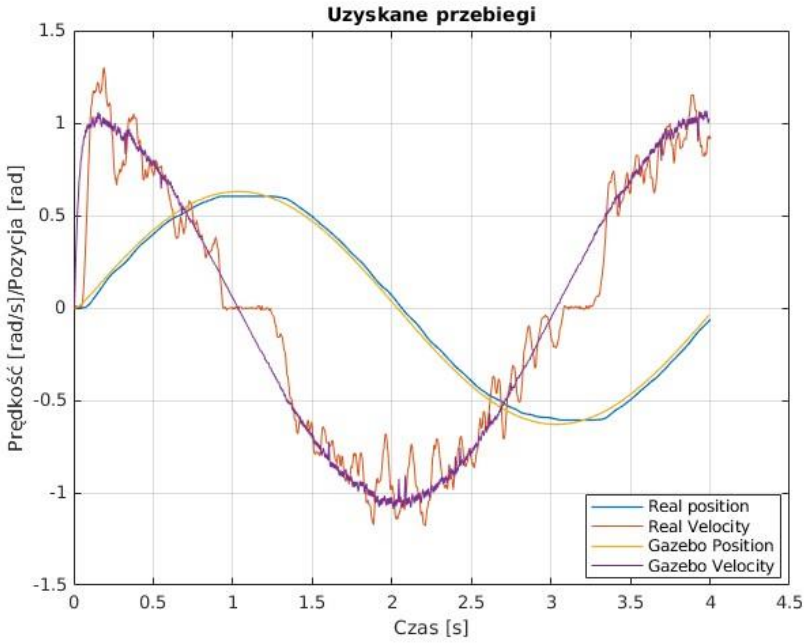
6.4.3. Analiza uzyskanych wyników

Jako kryterium oceny dokładności symulacji przyjęto pierwiastek błędu średniokwadratowego dla położeń i prędkości między wartościami rzeczywistymi a symulowanymi.

$$RMSE = \sqrt{\frac{1}{n} \cdot \sum_{k=1}^{n} (X_k - \overline{X_k})^2} \tag{8}$$

,gdzie $RMSE$ – pierwiastek z błędu średniokwadratowego, n – liczba danych, X_k – wartość symulowana, $\overline{X_k}$ – wartość rzeczywista

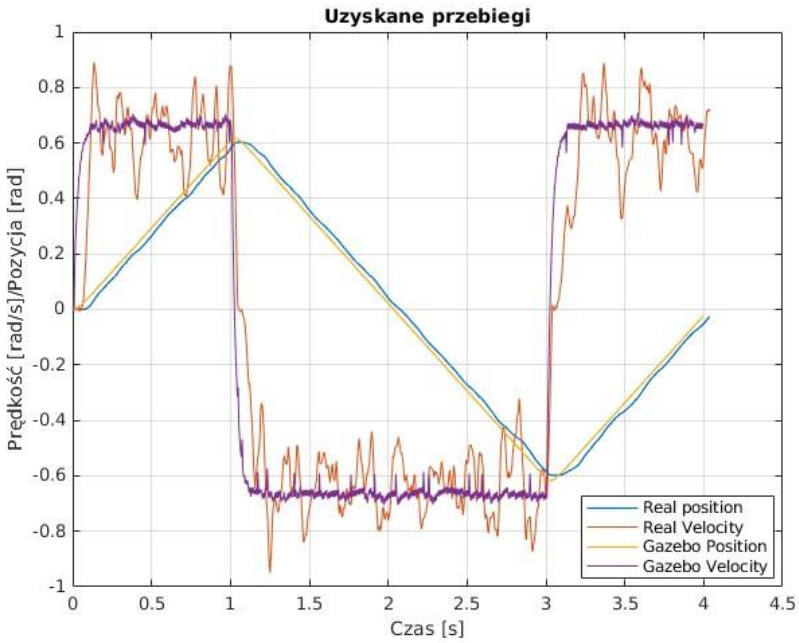
Największy błąd odwzorowania zaobserwowano dla mniejszych zadanych ruchów. Wykresy porównawcze otrzymanych charakterystyk oraz wartości $RMSE$ dla poszczególnych przebiegów przedstawiono poniżej.



Wykres 5: Porównanie uzyskanych trajektorii dla przebiegu sinusoidalnego

$$RMSE_{pos} = 0.023056\, rad$$

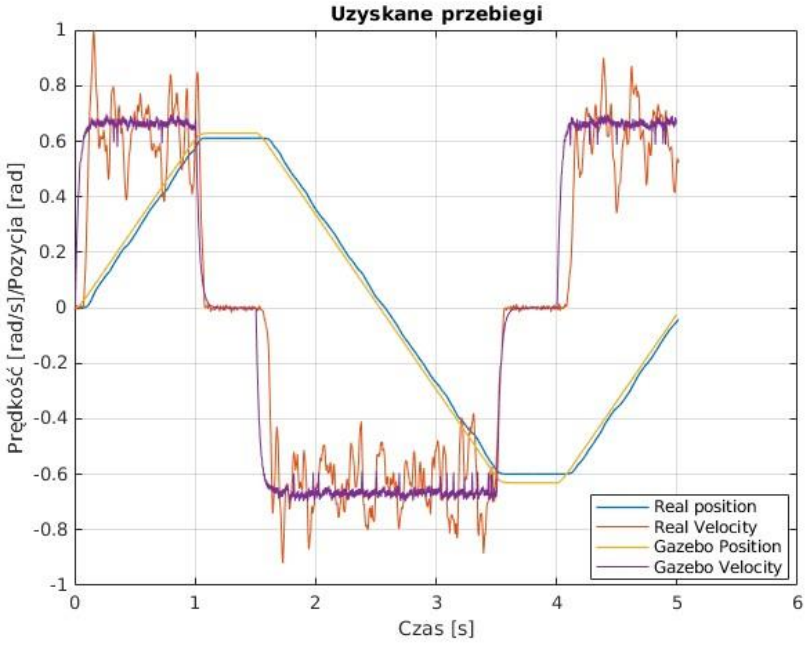
$$RMSE_{vel} = 0.1639\, \frac{rad}{s}$$



Wykres 6: Porównanie uzyskanych trajektorii dla przebiegu trójkątnego

$$RMSE_{pos} = 0.019804\,rad$$

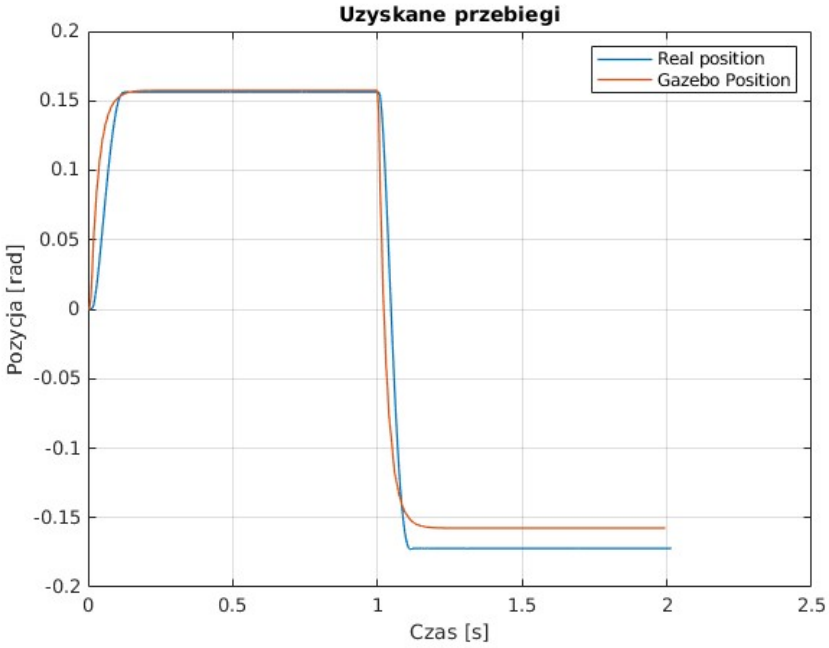
$$RMSE_{vel} = 0.1656\,\frac{rad}{s}$$



Wykres 7: Porównanie uzyskanych trajektorii dla przebiegu trapezowego

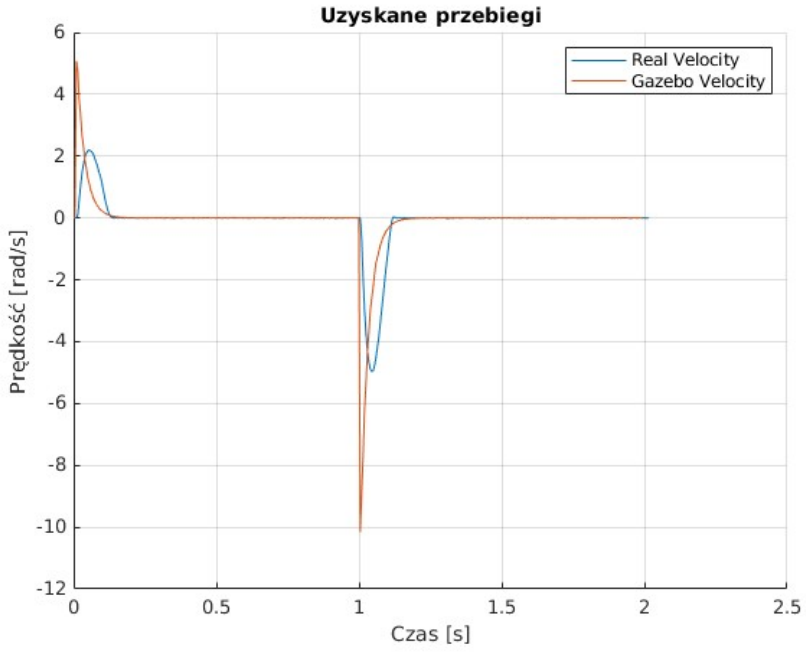
$$RMSE_{pos} = 0.021372\,rad$$

$$RMSE_{vel} = 0.1596\,\frac{rad}{s}$$



Wykres 8: Porównanie położenia dla przebiegu skokowego

$$RMSE_{pos} = 0.020073\, rad$$



Wykres 9: Porównanie prędkości dla przebiegu skokowego

$$RMSE_{vel} = 0.8731\, \frac{rad}{s}$$

Analiza błędu wykazała, że najgorsze odwzorowanie miało miejsce na poziomie prędkości dla przebiegu skokowego, gdzie pierwiastek błędu średniokwadratowego wyniósł 0.873 Rad/s. Dla

pozostałych przebiegów wartości błędu pozostały w okolicy 0.02 radiana dla położenia i 0.16 Rad/s dla prędkości.

Ze względu na to, że w praktyce ruchy robota nie mają charakteru skokowego, zdecydowano się nie brać pod uwagę ostatniego przebiegu w procesie optymalizacji. Nadmierne dopasowanie modelu do tak prostego i niepraktycznego przypadku, nie poprawiłoby błędu odwzorowania, a jedynie poprawiło charakter odpowiedzi skokowej.

Dla przebiegu sinusoidalnego przy małych ruchach widoczny był znaczący wpływ tarcia w przekładni na wierzchołkach charakterystyki. Przy małych zmianach położenia prędkości spadały do zera, co skutkowało zniekształceniem trajektorii. Ze względu na to zdecydowano się użyć przebiegów sinusoidalnych jako wzorca do optymalizacji.

7. Optymalizacja opracowanego modelu

Celem poprawienia dokładności opracowanego modelu, zdecydowano się zastosować iteracyjną metodę doboru parametrów symulacji, wzorowaną na metodzie optymalizacji zastosowanej w pracy [2]. Przygotowano odpowiedni skrypt w programie Matlab, oparty na funkcji patternsearch [22] z Global Optimization Toolbox.

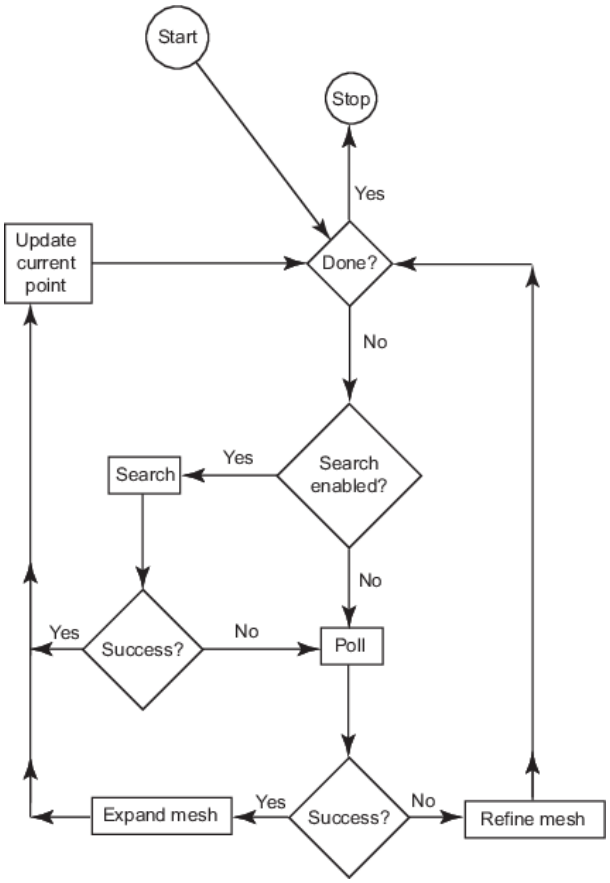
7.1. Funkcja patternsearch

Funkcja patternsearch z pakietu Global Optimization Toolbox jest bezgradientową funkcją optymalizacji numerycznej [20], zgodną z ideą Black Box przyjętą dla opracowanego w niniejszej pracy modelu.

W kolejnych krokach optymalizacji badana jest siatka wokół wartości początkowych w przestrzeni optymalizowanych parametrów. Algorytm eksploracyjnie dobiera kolejne wartości parametrów celem odnalezienie wartości, dla których przyjęta funkcja strat przyjmuje mniejsze wartości.

W przypadku braku zbieżności dla obranego kroku parametrów, siatka optymalizacji jest zagęszczana i proces szukania minimum jest wykonywany powtórnie.

W przypadku pomyślnego odnalezienia jednego lub więcej punktów, dla których funkcja osiąga nowe minimum, eksplorowany jest kierunek, który został odnaleziony jako pierwszy. Punkt ten jest obierany jako aktualne minimum, a wartość kroku parametrów jest zwiększana dwukrotnie [23].



Rysunek 51: Algorytm działania funkcji patternsearch

Źródło: [24]

7.2. Skrypt optymalizacyjny

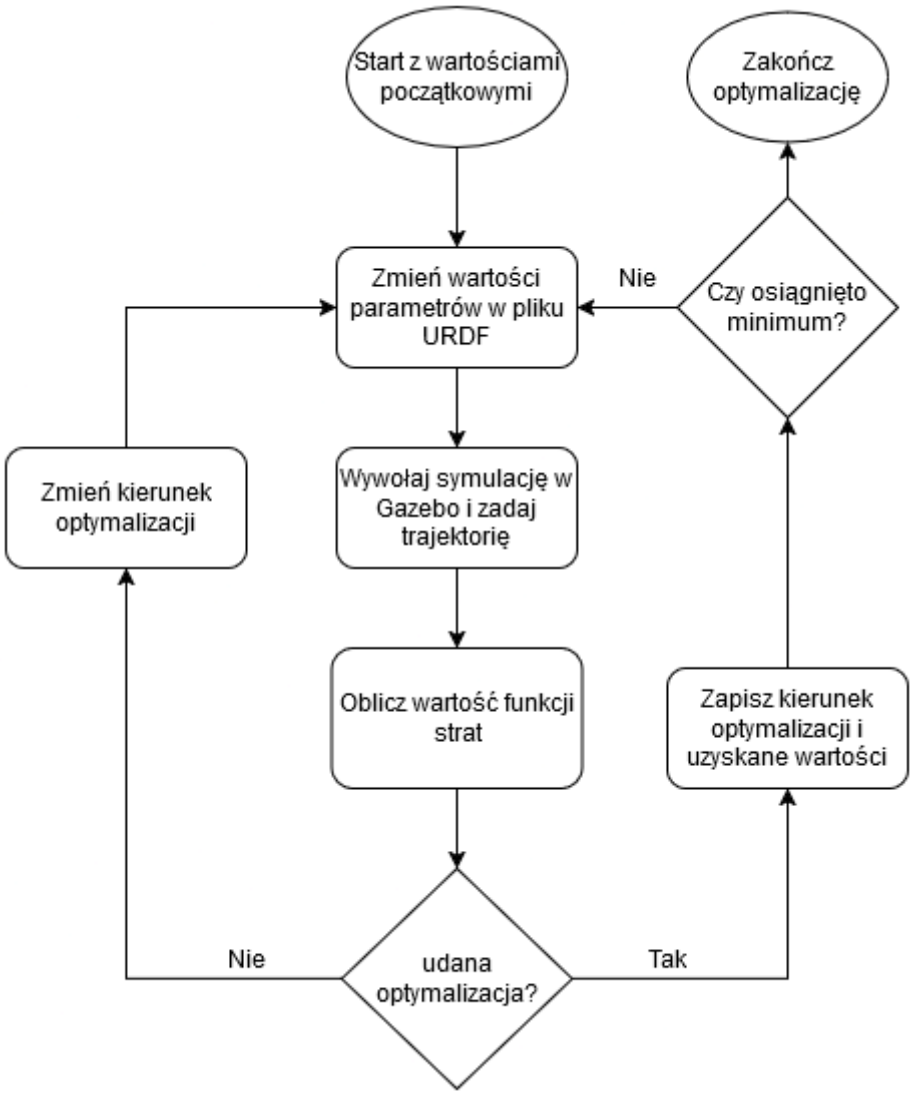
Na potrzeby optymalizacji skorzystano ze skryptu przedstawionego w pracy [2] po odpowiednim zmodyfikowaniu jego struktury zgodnie ze specyfiką modelu opracowanego w niniejszej pracy.

Skrypt iteracyjnie wywołuje symulację zadanej trajektorii, odpowiednio dostrajając wartości poszczególnych parametrów w pliku URDF. Jako funkcję strat przyjęto sumę błędu średniokwadratowego dla położenia oraz prędkości.

$$f_{loss} = 10 \cdot \delta_{pos} + \delta_{vel} \tag{10}$$

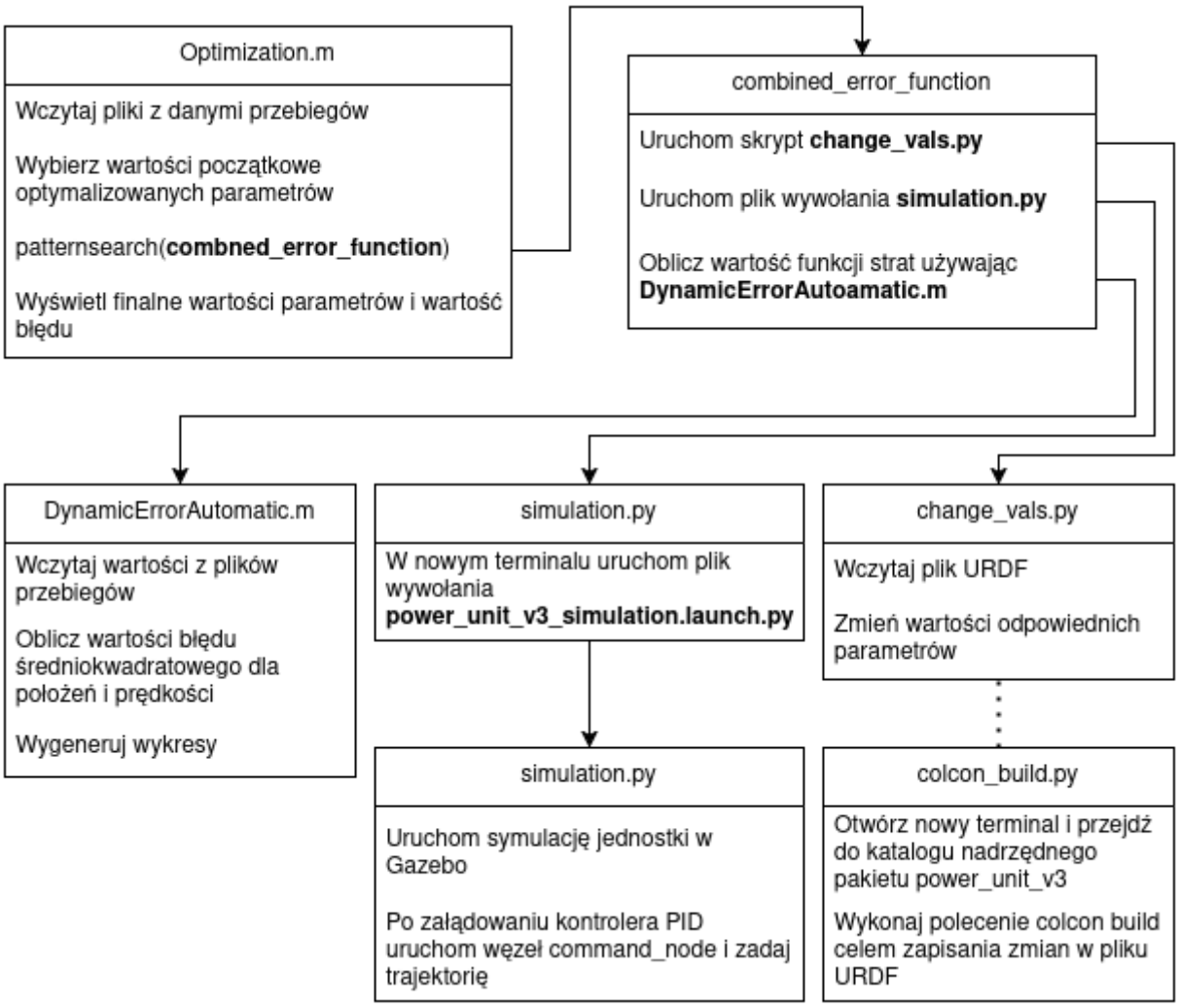
,gdzie δ_{pos} – błąd średniokwadratowy pozycji, δ_{vel} – błąd średniokwadratowy prędkości

Algorytm przygotowanego programu przedstawiono na schemacie poniżej.



Rysunek 52: Algorytm działania skryptu optymalizacyjnego

Do realizacji poszczególnych kroków algorytmu sporządzono lub zmodyfikowano odpowiednie funkcje oraz pomniejsze skrypty w języku Python. Na diagramie poniżej przedstawiono architekturę programu z podziałem na odpowiednie funkcje i skrypty.



Rysunek 53: Schemat struktury skryptu optymalizacyjnego

7.3. Wyniki optymalizacji parametrów modelu

Po zweryfikowaniu poprawności działania skryptu, przystąpiono do optymalizacji parametrów symulacji. Jako wzorzec przyjęto rzeczywisty przebieg dla trajektorii sinusoidalnej (Wykres 5), gdyż występujące dla niego błędy pozwalały zidentyfikować wpływ tarcia przekładni na dokładność odwzorowania zadanej trajektorii.

Wpływ tarcia statycznego był widoczny szczególnie na poziomie prędkości, gdzie przy małych zadanych zmianach położenia, prędkość spadała do wartości zerowych.

Przed przystąpieniem do wyznaczenia optymalnych wartości parametrów, zbadano wpływ poszczególnych z nich z osobna na osiągane trajektorie. Wartość parametru implicitSpringDamper ustawiono na 1, aby używana była odpowiednia metoda numerycznej symulacji tłumienia.

Stwierdzono, że parametry stopErp, oraz stopCfm, będąc jedynie szczegółowymi ustawieniami solvera, nie wpływały w istotny sposób na kształt charakterystyki, dlatego zdecydowano się nie uwzględniać ich w procesie optymalizacji.

Finalnie do programu przekazano następujący wektor parametrów, zawierający parametry tarcia, tłumienia, wartości nastaw regulatora PID, sztywność, oraz pozycję referencyjną dla sztywności:

$$[friction,\ damping,\ p\_value,\ i\_valaue,\ d\_value,\ spring\_stiffness,\ spring\_reference]$$

Podczas wstępnych weryfikacji wpływu parametrów na warunki symulacji wyznaczono również ich maksymalne zakresy. Wartości skrajne ustawiono odpowiednio w wektorach lower_bounds i upper_bounds na:

$$lower\_bounds = [0.0,\ 0.0,\ 0.5,\ 0.0,\ 0.01,\ 0.0,\ 0.1]$$

$$upper\_bounds = [1.5,\ 1.5,\ 4.0,\ 2.0,\ 0.5,\ 1.5,\ 6.28]$$

Wektor wartości początkowych parametrów ustawiono na:

$$initial\_guess = [0.0,\ 0.0,\ 1.7,\ 0.0,\ 0.05,\ 0.0,\ 0.0]$$

Po ustawieniu początkowych i skrajnych wartości optymalizowanych parametrów uruchomiono skrypt power_unit_v3_colcon_build.py, odpowiedzialny za budowanie pakietu symulacji jednostki napędowej po wprowadzeniu przez skrypt zmian w pliku URDF, po czym włączono skrypt optimization.m.

Dla wartości początkowych funkcja strat przyjęła następującą wartość:

$$f_{loss}(initial\_guess) = 25.913$$

Po 78 iteracjach funkcja strat została zminimalizowana do wartości:

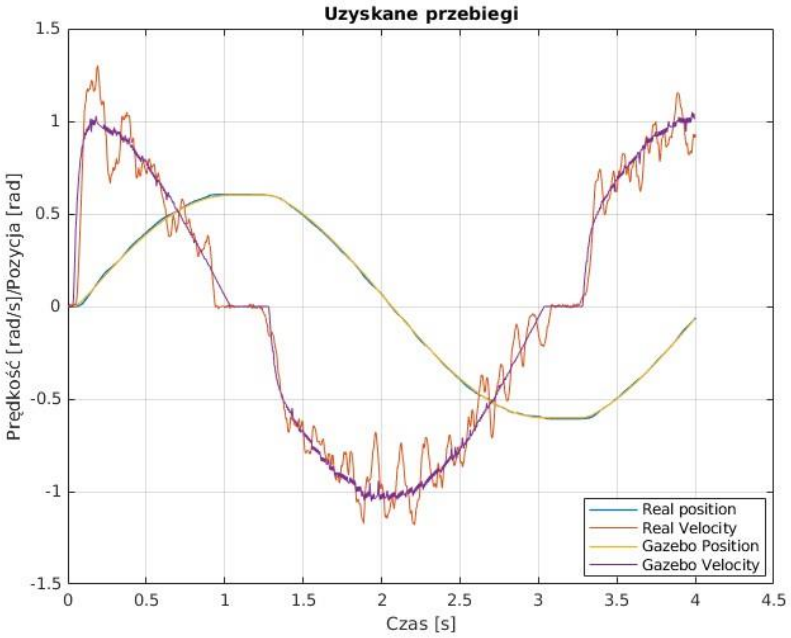
$$f_{loss}(final\_params) = 10.553$$

Wektor zoptymalizowanych parametrów miał postać:

$$optimized\_params = [0.0312,\ 0.0,\ 1.0906,\ 0.0,\ 0.0344,\ 0.0,\ 6.1875]$$

Wartość 6.1875 ostatniego parametru, ustawiającego wartość referencyjną położenia równowagi symulowanej sprężyny w złączu, odrzucono, gdyż ma ona wpływ jedynie wtedy, gdy ustawiona zostanie sztywność tejże sprężyny w parametrze spring_stiffness, którego wartość optymalna wyniosła 0.0.

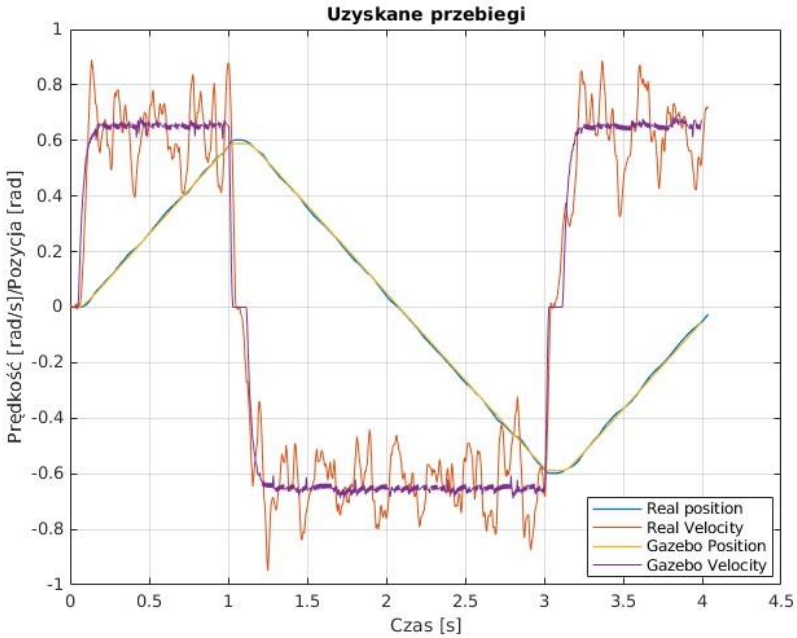
Po odpowiednim ustawieniu parametrów w pliku URDF ponownie wygenerowano wykresy porównawcze, oraz obliczono wartości RMSE zgodnie ze wzorem 8.



Wykres 10: Wynik optymalizacji dla trajektorii sinusoidalnej

$$RMSE_{pos} = 0.008139\,rad$$

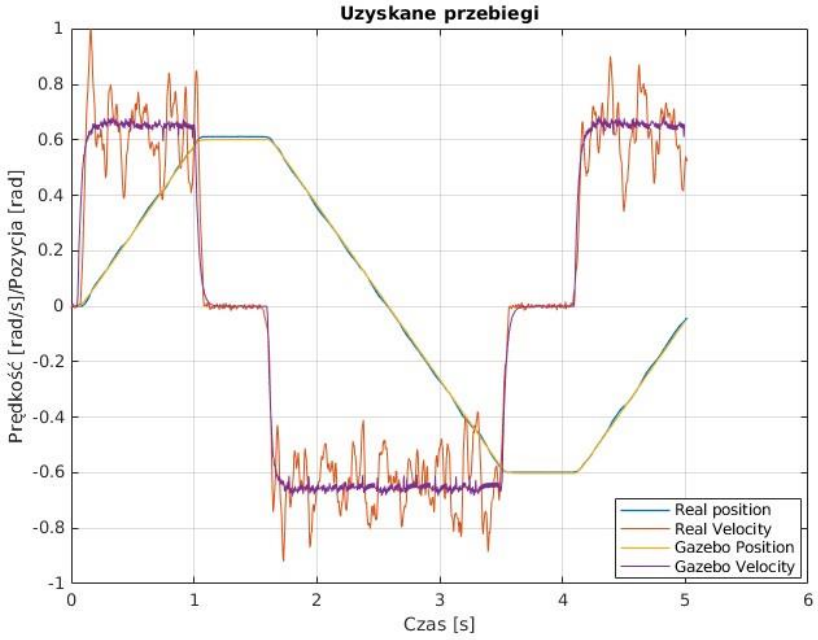
$$RMSE_{vel} = 0.1127\frac{rad}{s}$$



Wykres 11: Wynik optymalizacji dla trajektorii trójkątnej

$$RMSE_{pos} = 0.016943\,rad$$

$$RMSE_{vel} = 0.1245\,\frac{rad}{s}$$



Wykres 12: Wynik optymalizacji dla charakterystyki trapezowej

$$RMSE_{pos} = 0.011473\,rad$$

$$RMSE_{vel} = 0.1096\,\frac{rad}{s}$$

Wartości błędu przed i po optymalizacji dla poszczególnych charakterystyk, a także procentową poprawę odwzorowania przedstawiono w tabelach poniżej.

Tabela 2: Porównanie błędu położenia dla przebiegów po optymalizacji

Przebieg	$\delta_{pos}^{initial}[rad]$	$\delta_{pos}^{optimized}[rad]$	$poprawa[\%]$
Sinusoidalny	0.023056	0.0081394	64.7
Trójkątny	0.019804	0.0169430	14.4
Trapezowy	0.021372	0.0114730	46.3

Tabela 3: Porównanie błędu prędkości dla przebiegów po optymalizacji

Przebieg	$\delta_{vel}^{initial}\left[\frac{rad}{s}\right]$	$\delta_{vel}^{optimized}\left[\frac{rad}{s}\right]$	$poprawa[\%]$
Sinusoidalny	0.16390	0.11271	31.2
Trójkątny	0.16556	0.12454	24.8
Trapezowy	0.15964	0.10960	31.3

Dla wszystkich Trajektorii zaobserwowano znaczący wzrost dokładności symulacji, z czego największą poprawę zaobserwowano na poziomie położeń dla trajektorii sinusoidalnej, gdzie błąd odwzorowania uległ zmniejszeniu o 64.7%.

8. Wnioski i potencjalne drogi rozwoju projektu

Wszystkie cele wyznaczone dla niniejszej pracy został osiągnięte. Udało się opracować model robota w symulacji, w odpowiednim stopniu odwzorowujący jego rzeczywistą strukturę oraz sposób sterowania.

Kod opracowany na potrzeby projektu spełnił kryterium czytelności, oraz został opatrzony w odpowiednie komentarze, dokumentujące jego działanie. Pakiety oraz katalogi przygotowane zostały w zorganizowany sposób, a ich zawartość intuicyjnie podzielono na podfoldery, pomniejsze pliki oraz instrukcje macro, gwarantując modularność i zapewniając prostotę wprowadzania ewentualnych zmian w strukturze projektu.

Dzięki uproszczonym modelom siatek kolizji, oraz autorskiemu regulatorowi PID napisanemu we współpracy z zespołem programistów Koła Naukowego Robotyków, udało się zapewnić odpowiednią wydajność symulacji, dla której współczynnik czasu rzeczywistego RTF utrzymany został na poziomie >95%.

Dzięki przeprowadzonej optymalizacji parametrów symulowanych jednostek napędowych, udało się zwiększyć dokładność odwzorowania rzeczywistych napędów robota. Błędy dla wszystkich zadanych trajektorii spełniły kryteria dokładności przyjęte dla modelu.

Aby zapewnić najlepszy poziom odwzorowania zachowania rzeczywistych napędów robota, w przyszłości wymagane jest powtórzenie procesu identyfikacji parametrów dla każdej jednostki napędowej z osobna.

Dokładność symulacji można również poprawić drogą weryfikacji danych masowych poszczególnych elementów konstrukcji robota. Każdą część z osobna należy zważyć, a tensory bezwładności wyliczyć nie dla pierwotnych modelów pochodzących z programu Inventor, lecz dla geometrii wygenerowanej przez slicer.

W przyszłości należy również przeprowadzić identyfikację parametrów kontaktu stopy z podłożem drogą odpowiednich testów na pojedynczej nodze, bądź całym robocie.

Po ukończeniu konstrukcji rzeczywistego robota należy powtórzyć testy zadawanych trajektorii dla wielu sterowanych złączy. Istotnym aspektem jest określenie, czy obecna konfiguracja robota, w której dane zwrotne o położeniu w złączach wyliczane są jedynie z enkodera na poziomie silnika, pozwala na niezawodne sterowanie, oraz wiarogodną symulację.

9. Załączniki

W systemie APD dołączono poniższe załączniki, będące efektem prac nad projektem omawianym w niniejszej pracy dyplomowej.

1. meldog_description – folder zawierający kod źródłowy pakietu ROS 2, służącego do symulacji czworonoga.
2. power_unit_v3 – folder zawierający kod źródłowy pakietu ROS 2, służącego do symulacji pojedynczej jednostki napędowej.
3. position_nodes – folder zawierający kod źródłowy pakietu ROS 2, służącego do zadawania trajektorii w symulacji.
4. control_scripts – folder zawierający skrypty Python, służące do zadawania trajektorii na rzeczywistej jednostce napędowej.
5. power_unit_optimization – folder zawierający skrypt optymalizacyjny.

Powyższe kody są również dostępne na repozytorium Koła Naukowego Robotyków pod adresami:

- `https://github.com/KNR-PW/LRT_meldog_ros`
- `https://github.com/KNR-PW/LRT_one_power_unit_identification`

10. Bibliografia

Literatura

[2] Mazurkiewicz J. *Tworzenie cyfrowego bliźniaka dla robota humanoidalnego w ROS: Modelowanie i Symulacja*. Warszawa: Politechnika Warszawska 2024.

Artykuły

[1] „Digital Twin”. W: (). dostęp 10.01.2025. URL: https://en.wikipedia.org/wiki/Digital_twin

[3] Shi Y, Shilin L, Mingqiu G, Yuan Y, Dan X, i Xiang L. *"Structural Design, Simulation and Experiment of Quadruped Robot"* W: Applied Sciences 11, no. 22: 10705. doi: 10.3390/app112210705

[4] Teixeira de Paula D, Paciencia Godoy E, i Becerra-Vargas M. *“Dynamic Modeling and Simulation of a Torque-Controlled Spatial Quadruped Robot”.* W: Robotica 42, no. 8 (2024): 2761–80. doi: 10.1017/S0263574724001097

[5] Chang X, An H, Ma H. *“Modeling and base parameters identification of legged robots.”* W: Robotica. 2022;40(3):747-761. doi:10.1017/S0263574721000783

[20] *„Derivative-free optimization”* W: (). dostęp 12.01.2025. URL: https://en.wikipedia.org/wiki/Derivative-free_optimization

Źródła internetowe

[7] *ROS2 For Beginners (ROS Foxy, Humble - 2025)*. dostęp 03.01.2025. URL: https://www.udemy.com/course/ros2-for-beginners/

[8] *ROS2 for Beginners Level 2 – TF | URDF | Rviz | Gazebo*. dostęp 03.01.2025. URL: https://www.udemy.com/course/ros2-tf-urdf-rviz-gazebo/

[13] *Repozytorium gz-sim.* dostęp 12.01.2025. URL: https://github.com/gazebosim/gz-sim

[19] *Repozytorium KNR – LRT_joint_controller_ros2.* dostęp 13.01.2025. URL: https://github.com/KNR-PW/LRT_joint_controller_ros2

Dokumentacje

[6] *ROS2 Humble Documentation.* dostęp 10.01.2025. URL: https://docs.ros.org/en/humble/index.html

[9] *Dokumentacja formatu URDF.* dostęp 12.01.2025. URL: http://wiki.ros.org/urdf/XML

[10] *Dokumentacja formatu URDF – element link.* dostęp 12.01.2025. URL: http://wiki.ros.org/urdf/XML/link

[11] *Dokumentacja formatu URDF – element joint.* dostęp 12.01.2025. URL: http://wiki.ros.org/urdf/XML/joint

[12] *Dokumentacja ROS: xacro.* dostęp 12.01.2025. URL: http://wiki.ros.org/xacro

[14] *Dokumentacja SDFormat – tag gazebo.* dostęp 12.01.2025. URL: http://sdformat.org/tutorials?tut=sdformat_urdf_extensions&cat=specification

[15] *Dokumentacja ros2_control dla ROS Humble.* dostęp 12.01.2025. URL: https://control.ros.org/humble/index.html

[16] *Dokumentacja Gazebo – zastosowanie w URDF.* dostęp 12.01.2025. URL: https://classic.gazebosim.org/tutorials/?tut=ros_urdf

[17] *Dokumentacja oprogramowania Moteus.* dostęp 12.01.2025. URL: https://github.com/mjbots/moteus/blob/main/docs/reference.md

[18] *Dokumentacja ROS – Four Bar Linkage.* dostęp 12.01.2025. URL: http://docs.ros.org/en/jade/api/transmission_interface/html/c++/classtransmission__interface_1_1FourBarLinkageTransmission.html

[21] *Dokumentacja biblioteki asyncio.* dostęp 12.01.2025. URL: https://docs.python.org/3/library/asyncio.html

[22] *Dokumentacja funkcji patternsearch.* dostęp 12.01.2025. URL: https://www.mathworks.com/help/gads/patternsearch.html

[23] *How Pattern Search Polling Works.* dostęp 12.01.2025. URL: https://www.mathworks.com/help/gads/how-pattern-search-polling-works.html

[24] *Patternsearch: searching and polling.* dostęp 10.01.2025. URL: https://la.mathworks.com/help/gads/searching-and-polling.html

Spis ilustracji

Rysunki

Wykresy

Spis tabel