

Politechnika Warszawska

WYDZIAŁ MECHANICZNY TECHNOLOGICZNY



Instytut Technik Wytwarzania
Zakład Automatyzacji i Obróbki skrawaniem

Praca dyplomowa inżynierska

na kierunku Automatyzacja i Robotyzacja Procesów Produkcyjnych

Zaprojektowanie Symulacji Autonomicznego Robota Mobilnego,
Wykorzystującego Technologie LIDAR jako element systemu
bezpieczeństwa

*Design Simulation of Autonomous Mobile Robot, Using LIDAR
Technologies as an Element of Safety System*

numer pracy według wydziałowej ewidencji prac 110C-ISP-AP/312737/1219473

Michał Aleksander Nowakowski

numer albumu 312737

promotor
dr inż. Mirosław Nejman

konsultacje

—

Warszawa 2024

Streszczenie:

W globalnym kontekście przemysłowym rozwój technologii cyfrowych odgrywa kluczową rolę w rewolucji produkcji. W tej transformacji intralogistyka, czyli logistyka wewnętrzna, staje się istotnym obszarem ze względu na rosnące potrzeby elastyczności i szybkości w cyklach produkcyjnych oraz niedobory siły roboczej. Smart intralogistics oferuje możliwość szybkich i ekonomicznych zmian w przepływie materiałów, umożliwiając dostosowanie procesów produkcyjnych i magazynowych. Jednak elastyczność nie może kompromitować efektywności operacyjnej i kosztowej. W kontekście produkcji i magazynowania, autonomiczne roboty mobilne (AMR) stają się zaawansowaną technologią umożliwiającą wspomnianą elastyczność [1]. Celem niniejszej pracy inżynierskiej, zrealizowanej w Instytucie Technik Wytwarzania, Zakładzie Automatyzacji i Obróbki Skrawaniem w Politechnice Warszawskiej, jest zaprojektowanie symulacji autonomicznego robota mobilnego wykorzystującego technologię *LIDAR* jako integralny element systemu bezpieczeństwa. Praca zawiera krótki opis koncepcji Smart Intralogistics, historię, założenia tego podejścia oraz prezentacje obecnego rynku AMR. Skupia się również na użyteczności symulacji w projektowaniu systemów, szczególnie związanych z bezpieczeństwem, które tworzą funkcjonalność robota *AMR*. Omówione są kwestie oprogramowania wspierającego proces projektowy oraz komercyjnie wykorzystywanej platformy *ROS2* na systemie operacyjnym *Ubuntu*. W dalszym etapie pracy poruszane są kwestie elementów składających się na zaproponowane rozwiązanie takie jak wybór systemu operacyjnego oraz zastosowanie narzędzi informatycznych, m.in. *Docker* i *RViz2*. Opisane są założenia symulacji opartej na *Gazebo* oraz jej interakcja z platformą robota. Wyjaśniona jest struktura nawigacyjna *NAV2* i algorytm lokalizacyjny *SLAM*, zamykając część teoretyczną pracy. Następnie skupiono się na wykorzystaniu tych technologii i algorytmów w autorskim systemie bezpieczeństwa opartym na technologii *LIDAR*. W procesie tworzenia oprogramowania stanowiącego szkielet robota oraz jego parametryzacji wykorzystano repozytorium „*joshnewans/articubot_one*” dostępne na platformie *GitHub*. Algorytmy systemu bezpieczeństwa zostały zaprogramowane w oparciu o strukturę *ROS2*, język programowania *Python* oraz środowisko wizualizacyjne *RViz2*. Zaproponowane rozwiązanie pozwoliło na znaczną redukcję kosztów produkcji robota poprzez integrowany system nawigacji, lokalizacji i wykrywania przeszkód oparty głównie na skanerze *LIDAR*. Wbudowany system bezpieczeństwa stanowi integralną część oprogramowania robota, eliminując potrzebę dodawania wielu komponentów, integracji oprogramowania i potencjalnych punktów awarii. W celu nadzorowania wydajności systemu bezpieczeństwa, użyto skryptu generującego wykresy na podstawie danych dotyczących aktywności nawigacji, odległości od przeszkody, prędkości robota oraz sygnałów wysyłanych przez system bezpieczeństwa. Pozwala

to na interpretację poprawności działania systemu. Praca kończy się prezentacją wyników działania zaproponowanego rozwiązania. Praca inżynierska skupia się na projektowaniu symulacji i implementacji systemu bezpieczeństwa opartego na technologii *LIDAR* dla autonomicznego robota mobilnego, co przyczynia się do zwiększenia elastyczności i redukcji kosztów produkcji w przemyśle.

Słowa kluczowe:

Smart Intralogistics , Python, ROS2, Docker, Ubuntu

Summary:

In the global industrial context, the development of digital technologies is playing a key role in the manufacturing revolution. In this transformation, intralogistics, or internal logistics, is becoming an important area due to the growing need for flexibility and speed in production cycles and labor shortages. Smart intralogistics offers the ability to change material flows quickly and cost-effectively, allowing production and warehouse processes to adapt. However, flexibility must not compromise operational and cost efficiency. In the context of manufacturing and warehousing, autonomous mobile robots (*AMR*) are emerging as an advanced technology to enable the aforementioned flexibility. [1] The purpose of this engineering work, carried out at the Institute of Manufacturing Technology, Department of Automation and Machining, Warsaw University of Technology, is to design a simulation of an autonomous mobile robot using *LIDAR* technology as an integral component of a safety system. The paper includes a brief description of the Smart Intralogistics concept, history, assumptions of the approach and presentations of the current *AMR* market. It also focuses on the usefulness of simulation in the design of systems, particularly those related to safety, that make up the functionality of an *AMR* robot. Software issues supporting the design process and the commercially used *ROS2* platform on the Ubuntu operating system are discussed. The work goes on to address the elements that make up the proposed solution, such as the choice of operating system and the use of IT tools, including docker and rviz2. The assumptions of the Gazebo-based simulation and its interaction with the robot platform are described. The *NAV2* navigation structure and *SLAM* localization algorithm are explained, concluding the theoretical part of the paper. It then focuses on the use of these technologies and algorithms in a proprietary *LIDAR*-based security system. The "*joshnewans/articubot_one*" repository available on the *GitHub* platform was used in the development of the software that forms the robot's framework and its parameterization. The algorithms of the safety system were programmed based on the *ROS2* framework, the *Python* programming language and the Rviz2 visualization environment. The proposed solution significantly reduced the robot's production cost through an integrated navigation, localization and obstacle detection system based mainly on *LIDAR* Scanner. The built-in safety system is an integral part of the robot's software, eliminating the need to add multiple components, software integration and potential points of failure. In order to monitor the performance of the safety system, a script is used that generates graphs based on navigation activity data, distance from the obstacle, robot speed and signals sent by the safety system. This allows interpretation of the correctness of the system's performance. The work ends with a presentation of the results of the proposed solution. The engineering work focuses on the design of simulation and implementation of a safety system based on

LIDAR technology for an autonomous mobile robot, which contributes to increasing flexibility and reducing production costs in industry.

Key words:

Smart Intralogistics , Python, ROS2, Docker, Ubuntu

Politechnika Warszawska

WYDZIAŁ MECHANICZNY TECHNOLOGICZNY



7

1.	Wstęp.....	1
1.1.	Cel i zakres pracy inżynierskiej	1
2.	Koncepcja <i>Smart Intralogistics</i>.....	1
2.1.	Autonomiczne roboty mobilne – czym są i jak działają	2
2.2.	Pierwsze autonomiczne roboty mobilne.....	3
2.3.	Nowoczesna era autonomicznej robotyki mobilnej.....	4
2.4.	Popyt na roboty <i>AMR</i>.....	5
3.	Systemy Bezpieczeństwa robotów AMR.....	5
3.1.	Ograniczenia softwarowe	5
3.2.	Sygnalizacja.....	6
3.3.	Wyznaczanie stref w środowisku	7
4.	Symulacja.....	8
4.1.	Czym jest symulacja	8
4.2.	Współczesne zastosowanie symulacji.....	9
4.3.	Symulacja <i>AMR</i>	14
5.	Opis oprogramowania wykorzystanego w pracy	18
5.1.	<i>Docker</i>	18
5.2.	<i>Ubuntu</i>.....	20
5.3.	<i>ROS2</i>.....	21
5.4.	<i>RViz2</i>.....	25
5.5.	<i>GitHub</i>	26
6.	Zaproponowane rozwiązanie	28
6.1	Założenia symulacji.....	28
6.2	System operacyjny komputera robota: <i>Ubuntu</i>.....	29
6.3	Narzędzie wspomagające rozwój oprogramowania: <i>Docker</i> ..	31
6.4	Platforma <i>ROS2</i> oraz integracja z symulatorem <i>Gazebo</i>.	33
6.4.1	Decyzja wyboru <i>ROS2</i> jako platformy robotycznej	33
6.4.2	Plan Struktury projektu robota w <i>ROS2</i>.....	33
6.5.	Algorytm lokalizacji robota: <i>SLAM</i>	41
6.6.	Zagadnienia nawigacyjne: <i>NAV2</i>	43
6.7	Środowisko wizualizacyjne: <i>RViz2</i>	46

6.8 System bezpieczeństwa oparty o skaner LIDAR	47
7. Implementacja rozwiązania	50
7.1 Przygotowanie systemu komputera.....	50
7.2 Model robota i system bezpieczeństwa.....	53
7.2.1. Zbudowanie środowiska.....	53
7.2.2 Test modelu, wizualizacja i symulacja	58
7.2.3 Biblioteki Lokalizacji i nawigacji.....	63
7.2.4 System Bezpieczeństwa	66
8. Metodyka badań i interpretacji wyników.....	74
8.1 Metoda badawcza.....	74
8.2 System pomiarowy	74
8.3 Pomiary	79
9. Podsumowanie	83
10. Wyniki i wnioski.....	84
11. Bibliografia.....	85

1. Wstęp

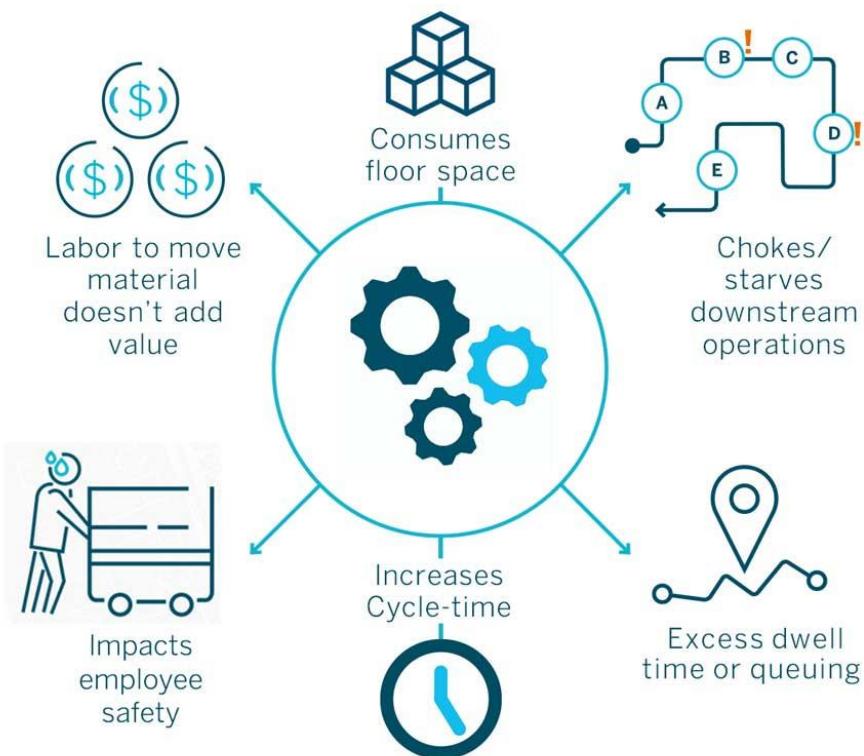
1.1. Cel i zakres pracy inżynierskiej

Celem pracy jest zaprojektowanie symulacji robota AMR, z zaimplementowanym systemem bezpieczeństwa opartym na technologii LIDAR. Za platformę służącą do stworzenia środowiska symulacyjnego wykorzystany został symulator Gazebo. W celu usprawnienia procesu produkcyjnego zastosowano narzędzia informatyczne takie jak *Docker* czy *RViz2*, zaoszczędzając czas związany z nieprzewidywalnymi problemami sprzętowymi / softwarowymi oraz zapewniając możliwość dokładniejszej wizualizacji działania symulowanych elementów. Stworzona platforma robota oparta na strukturze *ROS2* ma za zadanie na podstawie wysyłanych sygnałów sterujących, przemieszczać się w środowisku symulacji. Głównym celem robota, jest zatrzymanie ruchu w przypadku pojawienniu się przeszkody w odległości licznej od bazowego układu współrzędnych robota, określonego przez system bezpieczeństwa. Działanie systemu bezpieczeństwa polega na wstrzymaniu sygnału nawigacyjnego oraz następnej reaktywacji sygnału, po oddaleniu się od przeszkody / przesunięcia. Zagadnienia lokalizacyjne zostały zrealizowane poprzez zastosowanie algorytmu *SLAM*, natomiast zagadnienia nawigacyjne przez strukturę *NAV2*. Gotowe rozwiązanie ma na celu zmniejszyć koszt i zwiększyć prostotę systemu bezpieczeństwa. Projekt wykorzystując wbudowany *LIDAR*, ogranicza się do zaimplementowania skryptów w platformę robota.

2. Koncepcja *Smart Intralogistics*

Koncepcja *Smart Intralogistics*, czyli inteligentnej intralogistyki, odnosi się do zaawansowanych rozwiązań technologicznych stosowanych w zarządzaniu wewnętrznymi procesami logistycznymi w firmach. Głównym celem jest optymalizacja przepływu materiałów, zwiększenie efektywności operacyjnej, redukcja kosztów oraz elastyczne dostosowywanie się do zmieniających się warunków rynkowych [2]. Kluczowymi elementami tej koncepcji są: Automatyzacja, *IoT*(internet of things), Inteligentne oprogramowanie, *Big Data i analityka*, *AI*. Poprzez integrację tych zaawansowanych technologii, koncepcja *Smart Intralogistics* ma na celu stworzenie efektywnego, zautomatyzowanego i intelligentnego środowiska pracy, które pozwala firmom na skuteczne zarządzanie logistiką w dynamicznym otoczeniu biznesowym.

W produkcji, transporcie i magazynowaniu najbardziej dojrzałą technologią, która zapewnia tę możliwość, są autonomiczne roboty mobilne (*AMR*), najbardziej zaawansowany segment pojazdów *AGV* (Automated Guided Vehicles) [2].



Rys. 1 Zakres optymalizowanych obszarów według konceptu Smart Intralogistics [3]

2.1. Autonomiczne roboty mobilne – czym są i jak działają

Autonomiczne Roboty Mobilne (AMR) stanowią innowacyjne jednostki robotyczne, które wykazują zdolność do samodzielnego przemieszczania się w środowisku, nie wymagając stałego nadzoru ze strony ludzkiej. Wyposażone w zaawansowane sensory, wyspecjalizowane algorytmy nawigacyjne oraz precyzyjne systemy sterowania. Posiadają unikalną zdolność podejmowania autonomicznych decyzji oraz efektywnego dostosowywania się do zmiennych warunków otoczenia.



Rys. 2 Przykładowe wersje robotów AMR firmy OMRON [4]

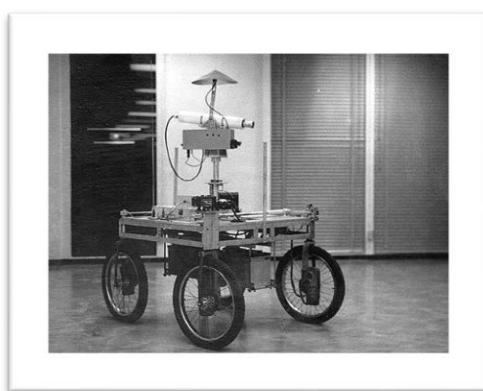
Kluczowym składnikiem funkcjonalności robota są różnorodne sensory, takie jak kamery, skanery laserowe, lidary czy czujniki ultradźwiękowe. Umożliwiają robotowi identyfikację przeszkód oraz śledzenie własnej lokalizacji. Dzięki wykorzystaniu wyspecjalizowanych algorytmów nawigacyjnych, które analizują dane pochodzące z sensorów, robot podejmuje decyzje dotyczące optymalnej trasy poruszania się. W typowych przypadkach system zarządzania zadaniami jest integrowany z systemem WMS (Systemem Zarządzania Magazynem), co pozwala na efektywne i skoordynowane wykonywanie zadań w magazynie.



Rys. 3 Przykładowy skaner LIDAR firmy SICK [5]

2.2. Pierwsze autonomiczne roboty mobilne

Pierwsze autonomiczne roboty mobilne pojawiły się w latach 80. i 90. XX wieku, a ich rozwój był związany z postępem w dziedzinie robotyki i technologii sensorów. Jednym z pionierów w tej dziedzinie był *Hans Moravec*, który w latach 80. eksperymentował z robotami mobilnymi zdolnymi do poruszania się w dynamicznych środowiskach [6].



Rysunek 4 Pierwsze roboty AMR [7]



Rysunek 5 Pierwsze roboty AMR [8]

Jednym z pierwszych komercyjnych zastosowań autonomicznych robotów mobilnych było ich wykorzystanie w zakładach przemysłowych do transportu materiałów i towarów. W miarę rozwoju technologii sensorów, algorytmów nawigacyjnych i systemów sterowania, roboty te zaczęły stawać się coraz bardziej zaawansowane [6].

2.3. Nowoczesna era autonomicznej robotyki mobilnej

W ciągu ostatnich trzech dekad nastąpiła rewolucja w dziedzinie autonomicznych robotów mobilnych. Początkowe rozwiązania, przyczyniły się do otwarcia drogi dla bardziej zaawansowanych AMR, obecnie powszechnie stosowanych w przemyśle i biznesie. Współczesne autonomiczne roboty mobilne mają zdolność do realizacji różnorodnych zadań, przyczyniając się do znaczącego obniżenia kosztów w wielu sektorach gospodarki.

Są w stanie funkcjonować przez długie godziny, co przyczynia się do redukcji obciążenia pracowników w okresach intensywnej produkcji. To umożliwia zoptymalizowanie efektywności w fabrykach i magazynach, eliminując potrzebę zwiększenia zatrudnienia [6].



Rys. 6 Stacja ładowania robotów AMR [9]



Rys. 7 Flota robotów AMR [10]

Oprogramowanie AMR może być aktualizowane na odległość, co umożliwia dostosowanie ich do nowych zadań i warunków pracy poprzez łatwą implementację nowych funkcji i algorytmów. Roboty AMR współpracują we flocie – nadzorem systemem nadzorującym pozycje i potencjalne ścieżki robotów względem punktów oznaczonych jako zrzut lub pobór. Na podstawie algorytmów analizujących całkowity koszt jazdy poszczególnych robotów do zadanej ścieżki, wybierany jest ten robot, dla którego zadanie zostanie wykonane najszybciej. Zależnie od Systemów floty, trasy mogą być planowane z wyprzedzeniem o jedno lub więcej zadań.

2.4. Popyt na roboty AMR

W miarę postępu technologicznego i coraz większego zrozumienia korzyści wynikających z zastosowania AMR, firmy mogą być bardziej skłonne inwestować w te rozwiązania. Zwłaszcza w sektorze logistyki, produkcji, magazynowania i innych obszarach, gdzie automatyzacja procesów może przynieść efektywność, optymalizację oraz oszczędności kosztów.

Service / year	2022	2023	2024	2025	2026
Autonomous inventory robots	718,1	839,8	983,3	1152,9	1353,5
Goods-to-person picking robots	864,5	1011,8	1185,7	1391,4	1634,9
Self-driving forklifts	259,9	306,5	361,9	428	506,7
Unmanned aerial vehicles	125,7	150	179,3	214,4	256,7

Tab. 1 Przewidywana wartość rynku AMR liczona w mln. USD 2022-2026 [11]

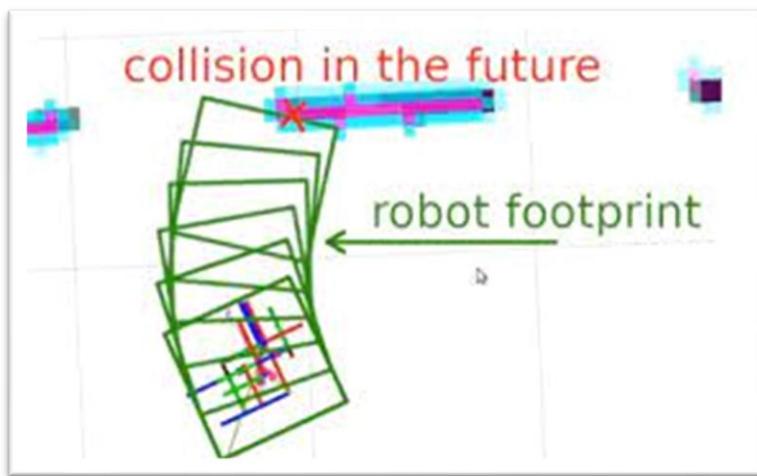
3. Systemy Bezpieczeństwa robotów AMR

3.1. Ograniczenia softwarowe

W sytuacjach bliskich kolizji, zarówno pomiędzy robotami, jak i wózkami widłowymi, bezpieczeństwo pracowników w fabryce staje się priorytetem. Kluczowym elementem jest precyzyjne dostosowanie parametrów prędkości jazdy oraz obrotu, aby robot mógł skutecznie wyhamować w krótkim czasie w przypadku zagrożenia. Istotny wpływ na te parametry ma zarówno masa samego robota, jak i ciężar ładunku, który przewozi.

Parametr maksymalnej prędkości musi być odpowiednio skalowany, uwzględniając nie tylko masę robota, ale również obciążenie, jakie przewozi. To zagwarantuje, iż w przypadku nagłej potrzeby zatrzymania, robot będzie w stanie zrealizować efektywnie zadanie, minimalizując ryzyko kolizji.

Kolejnym kluczowym elementem bezpieczeństwa są tzw. "footprinty". Stanowią one obszar, w którym wszelki ruch robota zostaje automatycznie wstrzymany, jeśli pojawi się w nim obiekt. Definicja footprintu opiera się na odległości mierzonej od układu współrzędnych mocowania skanera do punktów otaczających AMR. W praktyce oznacza to określenie strefy bezpieczeństwa, w której robot reaguje natychmiast na zbliżające się przeszkody, co minimalizuje potencjalne ryzyko kolizji



Rys. 8 Przykładowa konstrukcja footprintu robota [12]

3.2. Sygnalizacja

W niektórych systemach robotów AMR stosuje się niebieskie światło jako dodatkowy mechanizm bezpieczeństwa. Na przykład, gdy robot jest w ruchu lub zbliża się do danej strefy, może emitować niebieskie światło w celu ostrzeżenia otoczenia o swojej pozycji. Jest to szczególnie istotne w miejscach pracy, gdzie roboty współpracują z ludźmi. Rozwiążanie umożliwia zespołom pracowniczym śledzenie ruchu robotów.

Wałąną częścią systemu sygnalizacji są szeregi diod Led, które sygnalizują stan robota, kierunek jazdy, bądź skrętu, co umożliwia prostsze odczytywanie ruchu na hali produkcyjnej, np. Operator wózka widłowego dostaje informację, iż AMR skręca w lewo.

Oprócz sygnalizacji wizualnej, roboty AMR mogą wykorzystywać sygnały dźwiękowe, aby zwrócić uwagę na swój ruch. Specyficzne dźwięki mogą być skonfigurowane w zależności od kierunku, w jakim robot skręca.



Rys. 9 Robot z narzędziem manipulatora wyposażony w światła Blue Light [13]

3.3. Wyznaczanie stref w środowisku

Metoda polega na identyfikacji obszarów, w których roboty mogą operować w specyficzny sposób. Szczególnie przydatna przy pracach wdrożeniowych, gdzie każda fabryka różni się sposobem rozkładu korytarzy oraz maszyn. Zaistniałym problemem jest konieczność dopasowania algorytmów planowania ścieżek przez roboty oraz zasad, jakie na tych ścieżkach obowiązują: np. ruch dwu-stronny, ruch jednostronny. Strefy wyznacza się w lokalizacjach miejsc wyjątkowo niebezpiecznych (potencjalne zderzenie) lub miejsc możliwego wystąpienia tzw. „Deadlocku” – czyli sytuacji w których roboty ustawiają się na ścieżkach w sposób, który blokuje ich wzajemną jazdę.

3.4. Przyciski bezpieczeństwa

Po naciśnięciu przycisku bezpieczeństwa, robot *AMR* zazwyczaj wymaga ręcznego resetowania, co zabezpiecza przed przypadkowym wznowieniem ruchu. Procedura resetowania zapewnia dodatkową warstwę bezpieczeństwa, wymagając potwierdzenia, że sytuacja awaryjna została rozwiązana.

Przyciski bezpieczeństwa są zwykle wyposażone w sygnalizację awaryjną, która informuje otoczenie o aktywacji. Może obejmować migające światło, dźwięk alarmu lub inne środki komunikacji w celu szybkiego spostrzeżenia przez pracowników i zareagowania na sytuację awaryjną.

Operatorzy lub pracownicy obsługujący roboty *AMR* mają bezpośredni dostęp do przycisków bezpieczeństwa, co pozwala im szybko i skutecznie zareagować na sytuacje krytyczne, nawet bez konieczności korzystania z panelu kontrolnego robota.



Rys. 10 Versabot 1500 wersja: przenośnik rolkowy, lokalizacja przycisków bezpieczeństwa [14]

4. Symulacja

4.1. Czym jest symulacja

Symulacja to proces modelowania lub reprezentowania rzeczywistych zjawisk, systemów, lub procesów za pomocą odpowiedniego modelu matematycznego lub komputerowego. Celem symulacji jest badanie, analiza lub zrozumienie zachowania danego systemu w kontrolowanych warunkach, bez konieczności przeprowadzania rzeczywistych eksperymentów.

W przypadku komputerowych programów symulacyjnych mamy do czynienia z dynamicznie rozwijającą się dziedziną nauki umożliwiającą tworzenie statystycznie wiernych modeli badanych problemów oddających ich właściwości dokładnie tak jak byśmy je obserwowali i badali w rzeczywistości.

Przy zastosowaniu symulacji komputerowej weryfikacja różnych scenariuszy 'co jeśli ?' pozwala na szukanie najlepszej odpowiedzi bez ryzyka kosztownych błędów i strat przy prowadzeniu testów na "żywym" organizmie [15].

Symulacje komputerowe można podzielić ze względu na:

przewidywalność zdarzeń

- stochastyczne – korzystają z generatora liczb pseudolosowych lub (bardzo rzadko) losowych (szczególnie popularna jest metoda Monte Carlo).
- deterministyczne – wynik jest powtarzalny i zależy tylko od danych wejściowych i ewentualnych interakcji ze światem zewnętrznym.
- Rozmyte

sposób upływu czasu

- z czasem ciągły – czas zwiększa się stałymi przyrostami, jak w symulacji z czasem dyskretnym, lecz wartości próbek sygnałów są interpolowane dla chwil pośrednich pomiędzy momentami odczytu.
- z czasem dyskretnym – czas zwiększa się stałymi przyrostami, a krok czasowy dobiera się optymalnie ze względu na zasobów zużywanych przez system, jego wydajność i charakter symulowanego obiektu i/lub zjawiska (mikrosekundy w obwodach elektrycznych i miliony lat przy symulacji ewolucji gwiazd).
- symulacja zdarzeń dyskretnych – czas zwiększa się skokowo, ale jego przyrosty są zmienne (ważniejsza jest tu sekwencja zdarzeń niż rzeczywisty lub wirtualny upływ czasu).

formę danych wyjściowych

- statyczne – wynikiem jest zbiór danych, statyczny obraz itp.
- dynamiczne – wynikiem jest proces przebiegający w czasie np. animacja [16].

4.2. Współczesne zastosowanie symulacji

Wizualna Interaktywna Symulacja (VIS)

„To podejście zakłada tworzenie modelu, który reprezentuje zdarzenia zachodzące w konkretnym okresie czasu, uwzględniając ich wzajemne relacje, cechy charakterystyczne oraz niezbędne zasoby. Dzięki takiemu modelowi możliwe jest oddanie pełnej palety zmienności i przypadkowości, które występują w rzeczywistości podczas analizy różnych procesów, na przykład operacji na linii produkcyjnej, ruchu klientów w sklepie, przepływu informacji w firmie czy transportu bagażu na lotnisku. To zaawansowane podejście pozwala na dogłębną analizę realnych scenariuszy, co z kolei umożliwia lepsze zarządzanie oraz optymalizację różnych aspektów działalności” [15].

Vritual Factory – Symulacja wykorzystywana przez Specjalistów intralogistyki AMR. Wykorzystana do optymalizacji procesów, które rzeczywiście będą implementowane w danym obiekcie, co skraca czas wdrożenia i minimalizuje zakłócenia w procesie głównym. W trakcie tych eksperymentów definiowane są topologie procesów, obejmujące punkty końcowe, ścieżki i stacje lądowania, w kontekście potencjalnych nowych wdrożeń intralogistyki opartej na AMR [17].



Rys. 11 Metodyka rozwiązywania problemów przy zastosowaniu programu Virtual Factory [18]

WITNESS Horizon – Wszechstronne Oprogramowanie symulacyjne do prowadzenia nowoczesnych cyfrowych analiz i procesów produkcyjnych i logistycznych oparte na modelowaniu bliźniaka cyfrowego ang. *Digital Twin*.

„*Bliźniaki Cyfrowe* ang. *Digital Twin* to dynamicznie rozwijająca się dziedzina nauki polegająca na tworzeniu statystycznie wiernych wirtualnych cyfrowych modeli badanych obiektów lub procesów oddając dokładnie ich cechy i właściwości.

„*Symulacja komputerowa stworzonego bliźniaka umożliwia prowadzenie wirtualnych testów i weryfikacji setek, tysięcy, a nawet milionów scenariuszy "Co Jeśli?" w celu znalezienia najlepszego rozwiązania bez ponoszenia zbędnych kosztów, ryzyka błędów i strat co ma zawsze miejsce przy prowadzeniu testów na realnym procesie*” [19].



Rys. Logo oprogramowania Witness Horizon 12 [19]

Computer Aided Simulation (CAS)

To dziedzina inżynierii, która wykorzystuje oprogramowanie komputerowe do przeprowadzania symulacji w celu analizy, projektowania i wytwarzania produktów oraz systemów. Pojęcie obejmuje różne systemy CAx, dzięki którym można przeprowadzić:

- symulacje zjawisk podczas projektowania konstrukcji na podstawie metody MES,
- symulacje obróbki w systemach rodzaju CAP(Computer Aid Planning),
- symulacje obróbki w systemach rodzaju CAM(Computer Aided Manufacturing),
- symulacje kinematyki urządzeń wraz z badaniem kolizji,
- symulacje przepływu materiałów podczas wytwarzania,
- Symulacje realizacji zleceń oraz zadań transportowych

Symulator MES - MES to zaawansowana metoda numerycznego rozwiązywania problemów brzegowych. Polega ona na zastosowaniu interpolacji (jedno-, dwu- lub trójwymiarowej) poszukiwanej funkcji, na dyskretnym zbiorze jej węzłów, które powstają w wyniku dyskretyzacji dziedziny jej określoności na tzw. elementy skończone. Często zintegrowana z oprogramowaniem takim jak *Solidworks*, czy *Inventor* w postaci modułów symulacji.



Rysunek 13 Wynik przeprowadzenie analizy opartej o metodę MES w oprogramowaniu Solidworks [20]

Symulator obróbki CAM – Symulator w programie *Mastercam* umożliwia wirtualne przeprowadzenie procesu obróbki, co pozwala operatorom i programistom CNC zobaczyć, jak będzie wyglądał przebieg operacji obróbki na rzeczywistej maszynie CNC przed jej uruchomieniem.



Rysunek 14 Przykład wygenerowanej ścieżki ruchu narzędzia przez program Mastercam [21]

Symulacje układów fizycznych

Symulatory układów fizycznych to narzędzia, które pozwalają na modelowanie i analizę zachowań fizycznych obiektów, systemów czy procesów przy użyciu matematycznych równań i algorytmów. Te symulatory są szeroko stosowane w różnych dziedzinach, takich jak inżynieria, nauka, medycyna czy projektowanie. Poniżej znajduje się ogólny opis symulatorów układów fizycznych.

Modelowanie matematyczne:

- Symulatory układów fizycznych polegają na matematycznych modelach opisujących zachowanie danego systemu.
- Model ten może obejmować równania różniczkowe, równania różnicowe skończone, czy też równania algebraiczne, które reprezentują fizyczne zależności między różnymi parametrami układu.

Interakcja czasu rzeczywistego:

- Wiele symulatorów układów fizycznych umożliwia interakcję w czasie rzeczywistym. To oznacza, że użytkownicy mogą monitorować zmiany w systemie w trakcie jego ewolucji, wprowadzać modyfikacje, obserwować reakcje i analizować wyniki na bieżąco.

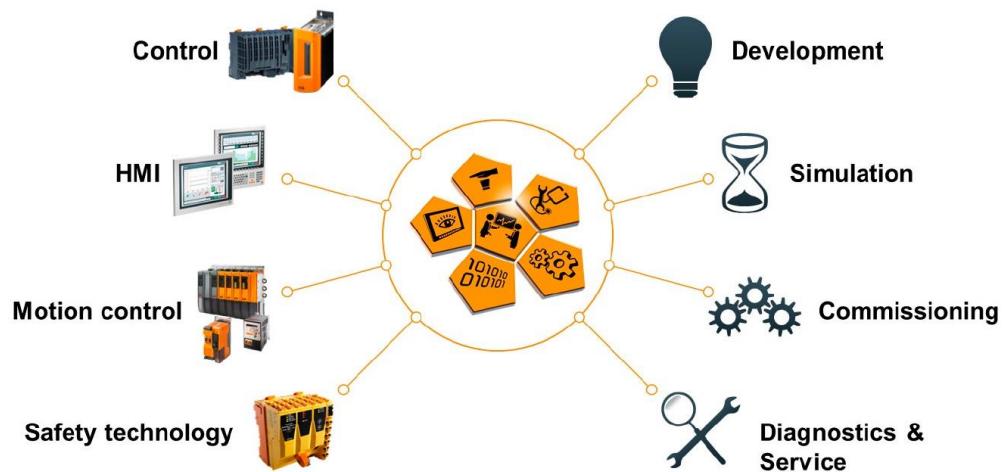
Wizualizacja:

- Symulatory często oferują wizualizację, która pozwala użytkownikowi obserwować, jak elementy układu fizycznego poruszają się, ewoluują, czy oddziałują między sobą. Ta funkcja jest szczególnie ważna w celu zrozumienia skomplikowanych zjawisk fizycznych i analizy wyników symulacji.

Zastosowania w szkoleniach i projektowaniu:

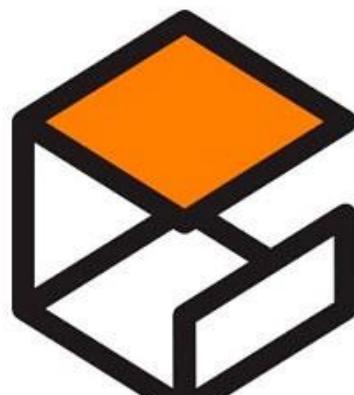
- Symulatory układów fizycznych są szeroko stosowane w celach szkoleniowych, umożliwiając użytkownikom doskonalenie umiejętności w zarządzaniu rzeczywistymi systemami bez konieczności bezpośredniego interweniowania.
- Ponadto, są one używane do projektowania i testowania nowych rozwiązań inżynierijnych przed ich fizyczną implementacją.

AutomationStudio - Automation Studio to oprogramowanie do projektowania obwodów, symulacji i tworzenia dokumentacji projektowej dla systemów zasilania cieczą i projektów elektrycznych. Oprogramowanie wyposażone jest we wszystkie niezbędne moduły od edytorów konfigurowania, programowania, przez rozbudowany symulator, po narzędzia diagnostyczne. W połączeniu umożliwiają efektywną, jednoczesną pracę zespołowi programistów nad wszystkimi elementami maszyny w ramach jednego projektu [22].



Rys. 15 Zakres funkcjonalności programu Automation Studio [22]

Gazebo – Stanowi zestaw bibliotek programistycznych typu *Open Source*, mających na celu usprawnienie procesu tworzenia symulatorów. Oprogramowanie to jest szczególnie cenione przez twórców robotów i projektantów. Jako symulator, Gazebo zdobyło szeroką popularność jako jedno z często preferowanych rozwiązań w obszarze symulacji robotycznej.

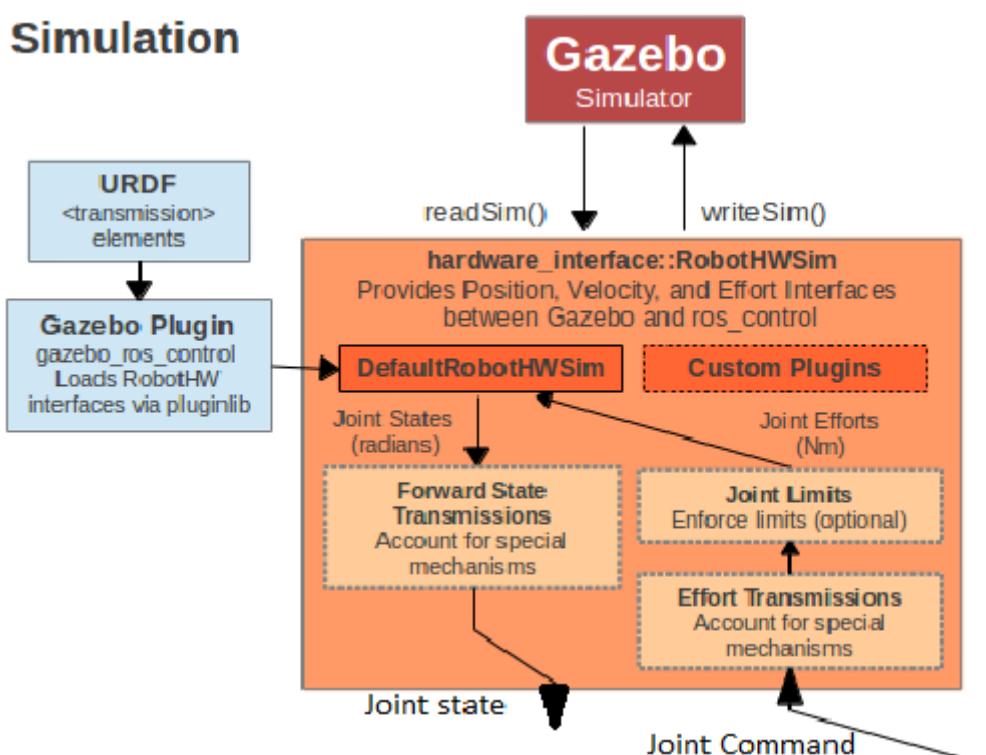


Rys. 16 Logo oprogramowania Gazebo [23]

4.3. Symulacja AMR

Jednym z kluczowych wyzwań w dziedzinie autonomicznej robotyki mobilnej jest skuteczny sposób lokalizacji. Realizacja tego zadania obejmuje konstrukcję modelu układów współrzędnych reprezentujących różne elementy, czujniki oraz charakterystyczne punkty robota. Odniesienia między nimi, oparte na danych ze skanerów, precyzyjnie definiują położenia obiektów w otoczeniu. Dzięki dokładnym informacjom o lokalizacji, nawigacja może efektywnie dobierać optymalną trasę do wyznaczonego celu, wykorzystując skany otoczenia.

Symulator *Gazebo*, z możliwością integracji z platformą *ROS2*, stanowi idealne narzędzie do testowania poprawności stworzonej konfiguracji platformy robota. Dzięki stosowaniu odpowiednich rozszerzeń oraz modułów symulacyjnych dla elementów takich jak lidar czy pozycja kół robota, a także uwzględniając ich fizyczne właściwości, np. tarcie, możliwe jest stworzenie modelu symulacyjnego, który wiernie odwzorowuje realne warunki funkcjonowania [24].



Rys. 17 Zasada interakcji symulacji Gazebo oraz ROS2 [25]

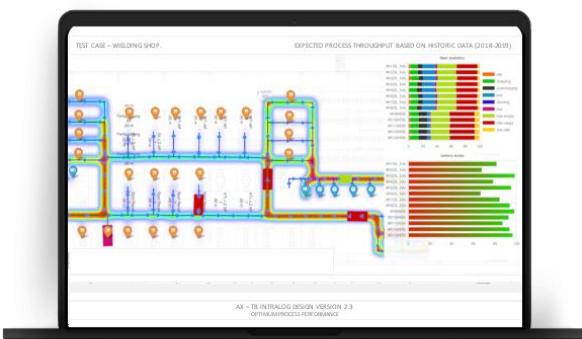
Zostało wybrane jako główne narzędzie symulacyjne służące za środowisko symulacyjne w pracy dyplomowej. Charakteryzuje się szeroką gamą właściwości, które umożliwiają precyzyjne odwzorowanie zachowania robota w rzeczywistym środowisku.

Cechy:

- Dostępne są: kamery głębinowe, *LIDAR*, *IMU*, czujniki kontaktowe, wysokościomierze i magnetometry.
- *DART* to domyślny silnik fizyki w *Gazebo Physics*, zapewniający poziom dokładności przewyższający silniki gier
- Możliwość skorzystania z bibliotek *Gazebo* na systemach *Linux* i *MacOS*.
- Większość bibliotek *Gazebo* oferuje interfejs wtyczek, który obsługuje użycie własnego kodu zintegrowanego z ich strukturą. W szczególności *Gazebo Rendering* i *Gazebo Physics*, które zapewniają wtyczki, potrzebne do zintegrowania dodatkowych silników renderujących i fizycznych.
- Szeroki zasób bibliotek zapewniających płynną integrację z platformą *ROS2*.
- Interfejs graficzny oparty na *QtQuick*: wizualizacja symulacji i zapewniony zestaw przydatnych wtyczek do wizualizacji *Topic*, dostarczania komunikatów oraz kontroli i statystyk świata symulacji [24].

Tematem wysoce interesującym, jednakże daleko wychodzącym poza zakres materiału odpowiedniego dla pracy dyplomowej jest zagadnienie sterowania flotą robotów. Główny komputer czyli tzw. *Menedżer floty* posiadając informację o pozycji każdego aktywnego robota oraz jego obecnym stanie(zajęty, wolny, ładowanie), zadaje robotowi przydział do konkretnego zadania. Algorytmy planujące ścieżki wyliczają optymalną trasę uwzględniając pozycję innych robotów znajdujących się obecnie na hali produkcyjnej, bądź innym centrum logistycznym.

Rozwiązanie firmy Versabox **VERSABOX VIRTUAL FACTORY™** stanowi gotowe rozwiązanie zajmujące się problematyką funkcjonowania floty robotów. Wykorzystując procedurę *run&rate* zastosowane jest do przeprowadzenia symulacji próbnego rozruchu systemu intralogistyki floty opartej na robotach AMR. Ideą produktu jest tworzenie wirtualnego procesu intralogistycznego przeprowadzonego bez fizycznej ingerencji w już działający system transportu wewnętrznego, a nawet przed powstaniem przestrzeni, w której ma być implementowany [17].



Rys. 18 Wygląd okna projektowego programu Virtual Factory [17]

Run at rate (*run@rate, run&rate*) – „najczęściej oznacza próbną serię wyrobów przed uruchomieniem właściwego procesu produkcji masowej” [17].

4.4. Korzyści wynikające z zastosowania symulacji

Wizualna Interaktywna Symulacja (VIS)

Wszechstronne Rozwiązania symulacyjne – Witness Horizon

Symulacja komputerowa tak stworzonego bliźniaka umożliwia prowadzenie wirtualnych testów i weryfikacji scenariuszy w celu znalezienia najlepszego rozwiązania. Brak ponoszenia kosztów, ryzyka błędów i strat w symulacji, co ma zawsze miejsce przy prowadzeniu testów na realnym procesie.

„*Każdy proces charakteryzujący się zmiennością, nieprzewidywalnymi zdarzeniami oraz dużą złożonością interakcji pomiędzy operacjami stanowi idealną okazję do zastosowania symulacji*”.

„*Klasycznych narzędzi takie jak mapy procesów i arkusze kalkulacyjne, najczęściej oparte są na danych uśrednionych, które nie oddają zupełnie prawdziwej natury analizowanych procesów. Analizy prowadzone na podstawie danych uśrednionych generują w najlepszym wypadku średnie wyniki, są przyczyną wielu kosztownych błędów projektowych. Główna korzyść to "spojrzenie w przyszłość" dające możliwość zdefiniowania najlepszych rozwiązań i eliminację strat przed zainwestowaniem dużych pieniędzy na wdrożenie zmian. Celem jest zdefiniowanie od razu optymalnego rozwiązanie z minimalnym ryzykiem błędów nie tracąc czasu na jego ciągłe poprawianie wymagające dodatkowych zasobów i generujące zbędne koszty*” [19].

Specjalistyczne Symulacyjne – Virtual Factory

Procedury *run at rate* przeprowadzane w sposób tradycyjny wymaga zestawienia w fizycznej przestrzeni wszystkich komponentów systemu intralogistycznego. Inaczej nie da się uzyskać wartościowych (sprawdzalnych) wyników. Oznacza to, iż trzeba dysponować kompletem robotów AMR, zainstalować stacje ładowania i miejsca postojowe, systemy poboru i zdawania ładunków. Konieczne jest też dostosowanie ścieżek transportowych (instalacja znaczników, oznakowanie poziome ciągów komunikacyjnych), a często także relokacja urządzeń produkcyjnych czy regałów magazynowych. Należy też przeprowadzić szkolenia dla operatorów innych maszyn transportowych i personelu pracującego w jednej przestrzeni z robotami. Dopiero w pełni zorganizowany system może być miarodajnie testowany.

Przygotowanie rozruchu systemu intralogistycznego w tradycyjny sposób jest czasochłonne i kosztowne – nawet przy założeniu, że większość predykcji znajduje potwierdzenie w praktyce. Gdy pojawiają się błędy, koszty rosną lawinowo. Każda modyfikacja, np. wprowadzenie nowego robota czy zmiana ścieżki, wymusza przeprowadzenie wszystkich procedur od początku.

„Stosując narzędzie VF, można uniknąć żmudnego zestawiania systemu i kosztownych testów, uzyskać wiarygodne wyniki run at rate jeszcze przed wprowadzeniem robotów AMR do hali produkcyjnej czy magazynowej”. [26]

Computer Aided Simulation (CAS)

Symulator MES

Symulacja MES umożliwia przeprowadzenie wirtualnych testów i analiz strukturalnych jeszcze przed faktycznym wykonaniem prototypu. To pozwala na dokładne zrozumienie zachowania struktury pod różnymi warunkami obciążenia, co skutkuje optymalizacją projektu.

Dzięki wczesnym testom i analizom, można uniknąć kosztownych błędów konstrukcyjnych. Możliwość wykrycia potencjalnych problemów i ich naprawa przed przejściem projektu do fazy produkcji jest kluczowa dla efektywności procesu.

Eliminacja konieczności tworzenia wielu prototypów fizycznych dodatkowo przyspiesza cały proces projektowy, umożliwiając szybsze wprowadzanie zmian i doskonalenie projektu już na etapie wirtualnych testów.

Symulator obróbki CAM

Analiza porównawcza półfabrykatu z modelem docelowego wyrobu umożliwia szczegółową ocenę poprawności trasowania narzędzi. Przez przeprowadzenie symulacji obróbki możliwe jest wykrycie potencjalnych błędów w programach CNC, takich jak kolizje narzędzi z materiałem czy elementami maszyny, co skutecznie zapobiega uszkodzeniom maszyn i narzędzi.

Eliminacja kolizji przy użyciu symulacji kinematyki przyczynia się do minimalizacji ryzyka awarii maszyn oraz związanych z nimi przestojów.

Symulacje układów fizycznych

AutomationStudio

Platforma pozwala na symulowanie różnych scenariuszy awaryjnych, co umożliwia sprawdzenie reakcji systemu i opracowanie efektywnych procedur bezpieczeństwa.

Automation Studio umożliwia integrację z innymi systemami, co pozwala na bardziej kompleksowe testowanie systemów w kontekście ich współpracy z różnymi technologiami.

Wczesna symulacja pozwala unikać kosztownych błędów w fazie produkcji czy instalacji. Dzięki temu można zminimalizować ryzyko napraw czy modyfikacji w późniejszych etapach, co przekłada się na ogólną redukcję kosztów projektu.

Automation Studio może być używane do celów szkoleniowych i edukacyjnych, umożliwiając symulację różnych scenariuszy w bezpiecznym środowisku, co jest szczególnie cenne dla osób uczących się obsługi systemów automatyzacji.

Gazebo

Dzięki symulacji w *Gazebo* możliwe jest weryfikowanie koncepcji oraz algorytmów sterowania robota już na etapie projektowania, co umożliwia wykrycie potencjalnych błędów i niedoskonałości.

Platforma ta umożliwia także przetestowanie algorytmów autonomicznych w różnorodnych scenariuszach, co skutkuje doskonaleniem ich działania bez konieczności przeprowadzania fizycznych testów na rzeczywistym robocie.

Dzięki symulacji w *Gazebo*, analiza wpływu różnych rodzajów czujników na funkcjonowanie robota jest prostsza w przeprowadzeniu. To pozwala na optymalizację układu sensorycznego, prowadząc do uzyskania lepszej percepji otoczenia.

Integracja *Gazebo* z ROS (Robot Operating System) ułatwia rozwój i testowanie oprogramowania robota, znacznie upraszczając proces projektowania i doskonalenia systemu robotycznego.

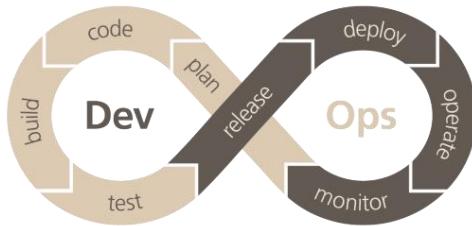
5. Opis oprogramowania wykorzystanego w pracy

5.1. Docker

"It works on my computer!" (Działa na moim komputerze!)

To jedno z powszechnych wyzwań w relacjach między drużyną programistyczną a zespołem testowym w oddziałach rozwoju oprogramowania. Ten komunikat odzwierciedla sytuację, w której programista twierdzi, że aplikacja działa poprawnie na jego lokalnym środowisku programistycznym, ale testerzy napotykają problemy w trakcie testów na innych środowiskach.

Programiści często pracują na własnych komputerach, które mogą różnić się pod względem konfiguracji, systemu operacyjnego, wersji języka programowania czy używanych bibliotek na komputerach testerów. W świetle rozrastających się serwerów i potrzebnych mocy obliczeniowych oraz konieczności szybkiego rozwoju oprogramowania nad którym pracują dziesiątki zespołów programistów, rynek zasugerował nową metodykę łączenia procesu rozwoju i testowania, nazwaną *DevOps*. Metoda opiera się na kontroli wersji i środowiska przy wykorzystaniu takich narzędzi jak między innymi *Docker*.



Rys. 19 Metodyka pracy DevOps [27]

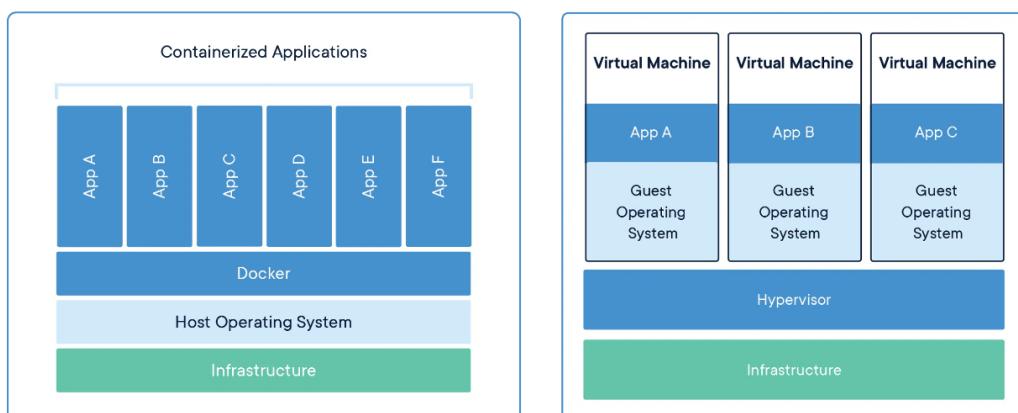
Czym jest Docker

Narzędzie służące do tworzenia ustandaryzowanych paczek rozwijanego oprogramowania, które zawierają wszystkie zależności, aby uruchomić rozwijany program. Metodyka pracowania z *Docker*, polega na korzystaniu przez zespół programistów z tych samych paczek *Docker* przy rozwoju aplikacji. *Kontenery* – bo tak nazywają się te „paczki” oprogramowania, uruchamiane poprzez silnik *Docker engine* zostają zainicjowane z tzw. *Obrazów (skompresowane kontenery)*. Funkcjonują tak samo, ponieważ są oparte na tych samych środowiskach np. *Ubuntu*, które zawierają te same, wszystkie konieczne zależności aby włączyć aplikacje. Takie rozwiązanie dodatkowo wprowadza kolejne ułatwienie – nie ma znaczenia na jakim systemie operacyjnym pracuje programista. System *kontenera* jest niezależny od systemu operacyjnego komputera lokalnego, więc na maszynie z *Windows*, można bez problemu zainicjować *kontener* z linuksem.

Różnica Docker a VM (Virtual Machine)

Maszyny wirtualne to środowiska systemów komputerowych z własnym interfejsem, przydzieloną pamięcią ram, pamięcią masową i przydzieloną mocą obliczeniową.

Kontenery zawierają wyłącznie konieczne zależności systemowe danej architektury (np. Linuks) potrzebne do uruchomienia aplikacji. Zasoby przydzielane są automatycznie z maszyny na której aktywne są procesy *Docker Engine* [28].



Rys. 20 Porównanie zagnieżdżenia VM i Docker na PC [28]

5.2. Ubuntu

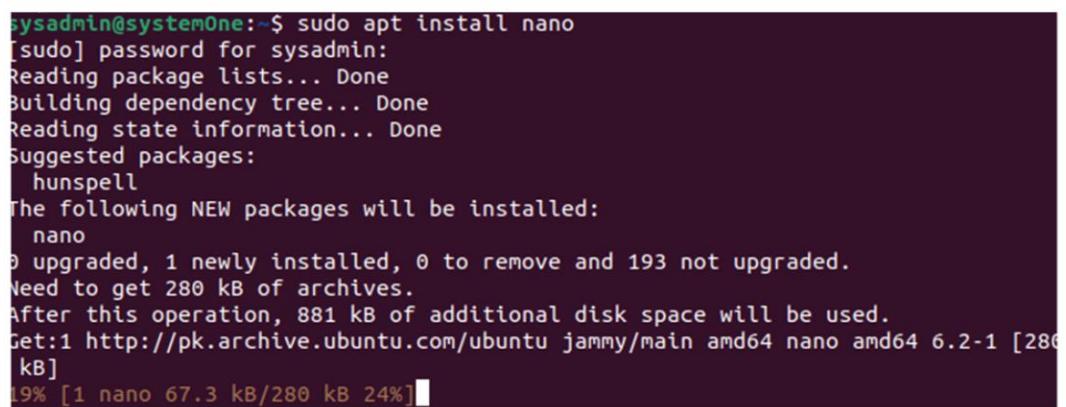
Specyfikacja

Ubuntu jest dystrybucją *Linuxa* opartą o system operacyjny *Debian*. Składa się głównie z oprogramowania typu *Open Source*. Oficjalnie wydane wersje systemu to między innymi: *Desktop*, *Server*, *Core*. System jest rozwijany przez Brytyjską firmę Canonical i społeczność programistów. Najnowsza wersja systemu na styczeń 2023 to 23.10 (*Mantic Minotaur*). Flagowa, stabilna i obecnie wspierana wersja to 22.04 (*Jammy Jellyfish*) [29].

Aktualizacja i pobieranie oprogramowania

Paczki systemu *Ubuntu* bazują na niestabilnym „branchu” (miejsce w narzędziu kontroli wersji oprogramowania, które ulega ciągłym zmianom) *Debian*, który jest aktualizowany co każde 6 miesięcy. *Ubuntu* korzysta z narzędzi do pobierania i zarządzania aktualizacjami: *APT* oraz graficzną nakładką *snap*. Powstanie *snap* wynika z konieczności znajomości pisania komend w terminalu przy korzystaniu z *APT*, jest ona swego rodzaju ułatwieniem dla użytkowników, którzy nie są zaznajomieni z językiem systemu linuks.

Procedura instalowania aplikacji, polega na dodaniu kodu źródłowego aplikacji do repozytorium *PPA* (Personal Package Archive), które następnie należy zbudować i zainstalować poprzez narzędzie *APT* (Advanced Packaging Tool) [29].



```
sysadmin@systemOne:~$ sudo apt install nano
[sudo] password for sysadmin:
Reading package lists... Done
Building dependency tree... Done
Reading state information... Done
Suggested packages:
  hunspell
The following NEW packages will be installed:
  nano
0 upgraded, 1 newly installed, 0 to remove and 193 not upgraded.
Need to get 280 kB of archives.
After this operation, 881 kB of additional disk space will be used.
Get:1 http://pk.archive.ubuntu.com/ubuntu jammy/main amd64 nano amd64 6.2-1 [280 kB]
19% [1 nano 67.3 kB/280 kB 24%]
```

Rys. 21 Przykład instalacji pakietu nano [30]

Wszystko jest edytowalnym plikiem

Charakterystyczną cechą systemu jest szeroka możliwość modyfikacji oprogramowania, sterowników, a nawet plików binarnych. Posiadając uprawnienia admin oraz zainstalowany edytor tekstu np. *nano*, *vim*, *yakuake* itd. możliwa jest modyfikacja każdej części systemu operacyjnego jak i zainstalowanego oprogramowania. Z tego powodu w *Ubuntu* szczególnie ważne jest zagadnienie ustawienia odpowiednich uprawnień dostępu do plików.

Bezpieczeństwo

Domyślnie, system *Ubuntu* zakłada przypisanie uprawnień niskiego poziomu dla instalowanych programów, to znaczy, iż nie mają one dostępu do plików systemu operacyjnego ani plików użytkowników innych niż ten operujący w obecnej sesji. W przypadku konieczności przydzielenia dostępu, przy wpisywaniu komendy uruchamiającej program korzysta się z narzędzia *sudo*, które nadaje oprogramowaniu tymczasowe uprawnienia do wykonywania zadań administracyjnych. Dostęp do profilu admina: *root*, dostępny jest z poziomu terminala. Dostęp zabezpieczony jest wcześniej ustanowionym hasłem [29].

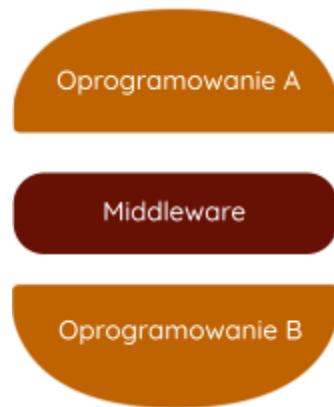
5.3. ROS2

Czym jest ROS2

ROS2 to oprogramowanie typu *middleware*, zbudowane na mechanizmie publikowania/subskrybcji ang. *publish/subscribe*, który wysyła informacje w postaci wiadomości ang. *Message* pomiędzy różnymi procesami.

Podstawą każdego systemu opartego na *ROS2* jest graf *ROS* ang. *ROS graph*. Graf składa z sieci węzłów ang. *Nodes* w systemie *ROS2* oraz połączeń, dzięki którym węzły wysyłają między sobą informację [31].

Middleware - Oprogramowanie umieszczone pomiędzy dwoma/wieloma innymi systemami/aplikacjami, umożliwiające ich komunikację [32].



Rys. 22 Struktura oprogramowania middleware [32]

Platforma *ROS2* to bardzo rozbudowane narzędzie, z którego korzysta większość firm wdrażających / rozwijających / sprzedających systemy robotów autonomicznych np. *Bosh*, *Boston Dynamics*, *MiR*, *Omron*.

Node – Węzeł

To elementy sieci *ROS2*, które korzystają z biblioteki *Client Library*, która stanowi API służące do ich wzajemnej komunikacji. Węzły mogą komunikować się z innymi węzłami w ramach tego samego procesu, w ramach różnych procesów, gdzie węzły obsługują inny proces oraz w ramach różnych komputerów. Są z reguły jednostką obliczeniową grafu ROS, która zajmującą się konkretnym zdaniem.

Węzły mogą publikować tematy ang. *Topic* przesyłając dane do innych węzłów lub subskrybować tematy, aby pobrać dane z innych węzłów. Mogą również pełnić rolę klienta serwisu ang. *Client Service*, który zleca innemu węzłowi wykonanie obliczeń w jego imieniu lub rolę serwera serwisu, aby oferować tą samą funkcjonalność innym węzłom. W przypadku konieczności wykonania długotrwałych obliczeń zamiast klienta/serwera serwisu ang. *Service Client/Server*, używa się klienta/serwera akcji ang. *Action Client/Server*. Zmiana sposobu zachowania węzła w czasie rzeczywistym, może być konfigurowana dzięki dokonywaniu zmian w ich parametrach [31].

Sieć

Węzły łączą się automatycznie przy inicjalizacji.

Kiedy węzeł zostaje zainicjalizowany, powiadamia inne węzły w tej samej domenie *ROS2*. Jeżeli istniejące w domenie węzły wyśle informacje o swoim stanie nawiązywany, a następnie utrwalany jest kanał komunikacyjny między węzłami [31].

Interfejsy

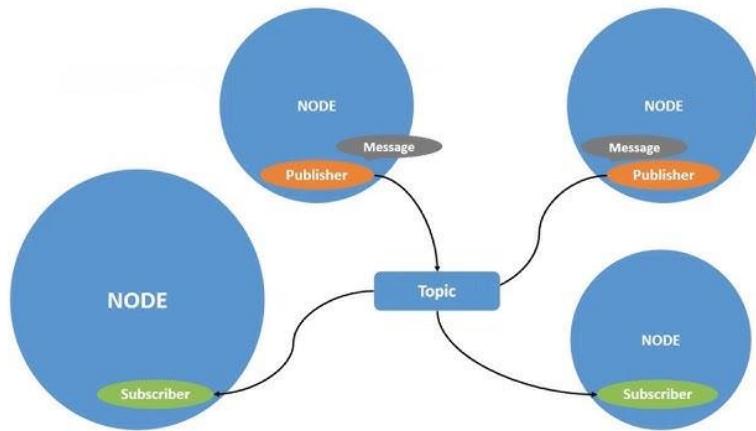
Aplikacje oparte na *ROS2* komunikują się poprzez interfejsy wyżej wspomnianych trzech typów: *Topic*, *Service*, *Action* [31].

Topic

W *ROS2*, *Topic* to sposób komunikacji między węzłami, umożliwiający przesyłanie wiadomości (*Message*) pomiędzy różnymi komponentami systemu. *Topic* jest nazwanym kanałem, za pomocą którego węzły mogą publikować dane (wysyłać wiadomości) lub subskrybować dane (odbierać wiadomości) w czasie rzeczywistym.

Każdy *Topic* jest identyfikowany przez unikalną nazwę, co umożliwia innym węzłom skoncentrowanie się na konkretnych rodzajach danych, które je interesują. Węzły, które chcą przekazywać informacje, publikują na określonym temacie. Węzły zainteresowane danymi subskrybują dany *Topic*, aby otrzymywać i przetwarzać informacje.

Przykładowo, jeśli istnieje *Topic* o nazwie */sensor_data*, węzeł zbierający dane z czujników może publikować na tym *Topic* informacje o odczytach czujników, a inne węzły, takie jak algorytmy przetwarzania danych czy interfejsy użytkownika, mogą subskrybować dany *Topic*, aby odbierać i wykorzystywać te dane [31].



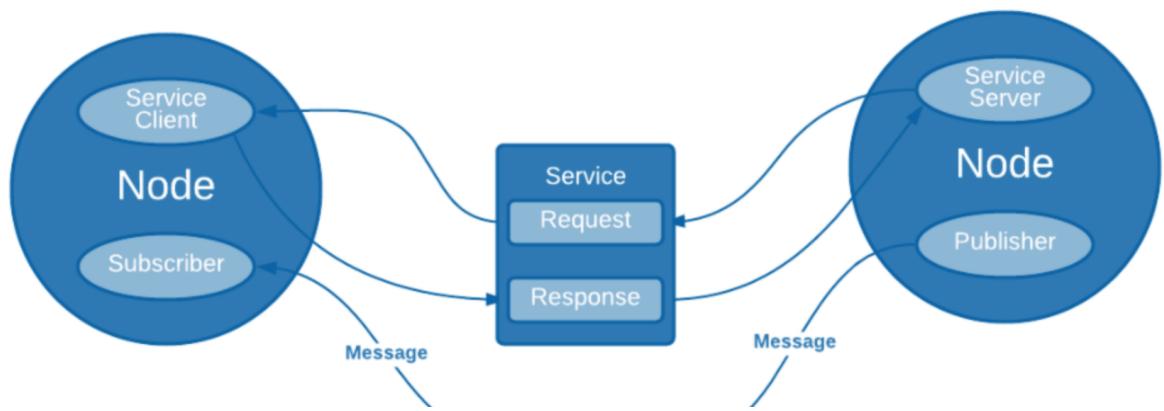
Rys. 23 przykładowa struktura komunikacji między węzłami w sieci ROS2 [33]

Service

Service (usługa) to mechanizm umożliwiający synchroniczną wymianę danych między węzłami. Usługa składa się z dwóch elementów: węzła klienta (*Client Node*), który wysyła żądanie, oraz węzła serwera (*Server Node*), który odpowiada na to żądanie. Usługa umożliwia przekazywanie danych w formie żądań i odpowiedzi, co pozwala na interakcję między różnymi komponentami systemu ROS2 w czasie rzeczywistym.

Węzeł klienta wysyła żądanie do węzła serwera, a następnie oczekuje na odpowiedź. W międzyczasie węzeł serwera przetwarza żądanie i generuje odpowiedź, którą przesyła z powrotem do węzła klienta.

Przykładowo, można stworzyć usługę do dodawania dwóch liczb. Węzeł klienta wysyła żądanie z dwoma liczbami, a węzeł serwera sumuje je i wysyła odpowiedź z wynikiem. Wymiana danych jest synchroniczna, co oznacza w kontekście ROS2 oznacza, iż serwer akcji nie informuje o postępie przetwarzania danych, tylko wysyła gotowy wynik [31].



Rys. 24 Przykład funkcjonowania akcji i serwisów [34]

Action

Action to mechanizm umożliwiający asynchroniczną wymianę danych między węzłami, gdzie jedno żądanie może trwać dłużej, a system informuje o postępie operacji. Akcje są używane

w sytuacjach, w których operacja wymaga pewnego czasu na wykonanie, na przykład przemieszczenie robota do określonej lokalizacji.

W porównaniu do usług (services), które są synchroniczne (czekają na odpowiedź), akcje pozwalają na wykonywanie długotrwałych zadań, a jednocześnie dostarczają informacji o postępie wykonania.

Akcje składają się z trzech głównych części: żądania (*goal*), celu (*result*) i sprzężenia zwrotnego (*feedback*). Węzeł klienta wysyła żądanie do węzła serwera, a następnie może odbierać aktualizacje sprzężenia zwrotnego oraz oczekiwany wynik operacji [31].

Parametryzacja

Parametry w *ROS2* są przypisane do poszczególnych węzłów. Parametry służą do konfigurowania węzłów podczas uruchamiania oraz działania bez wprowadzania zmian w kodzie. Czas życia parametru jest powiązany z czasem życia węzła (choćże węzeł może wdrożyć pewien rodzaj trwałości, aby przeładować wartości po ponownym uruchomieniu) [31].

Komendy

Głównym narzędziem służącym do interakcji z siecią *ROS2* jest polecenie `ros2`, które samo w sobie zawiera różne polecenia podrzędne służące do introspekcji i pracy z węzłami, *Topic*, serwisami i nie tylko.

Przykłady dostępnych poleceń podrzędnych obejmują:

- *action*: Introspekcja/interakcja z działaniami *ROS2*
- *launch*: Uruchom/przeanalizuj plik startowy
- *node*: Introspekcja węzłów *ROS2*
- *param*: Introspekcja/konfiguracja parametrów w węźle
- *run*: Uruchom węzeł *ROS2*
- *topic*: Introspekcja/publikowanie tematów *ROS2* [31]

Plik *Launch*

System *ROS2* zazwyczaj składa się z wielu węzłów działających w wielu różnych procesach (a nawet na różnych komputerach). Chociaż możliwe jest uruchomienie każdego z tych węzłów osobno, dość szybko staje się to kłopotliwe.

System uruchamiania ang. *Launch* w *ROS2* ma na celu zautomatyzowanie uruchamiania wielu węzłów za pomocą jednego polecenia. Plik startowy zapisywany jest w formacie `.py`, `.xml` lub `.yaml`. Po uruchomieniu skryptu, zostaną zainicjalizowane wybrane węzły [31].

Budowanie plików ROS2

Pierwszym krokiem jest pobranie oraz zainstalowanie *ROS2*. Następnie należy stworzyć folder środowiska o wybranej arbitralnej nazwie, w folderze stworzyć podfolder */src*. Znajdując się w podfolderze, korzystając z narzędzia *colcon*, wystarczy wpisać odpowiednią komendę, która inicjuje budowanie środowiska. Tak zbudowane środowisko jest gotowe do procesu rozwoju platformy robota, który zostanie dokładniej opisany w dalszym etapie pracy dotyczący implementacji rozwiązania [31].

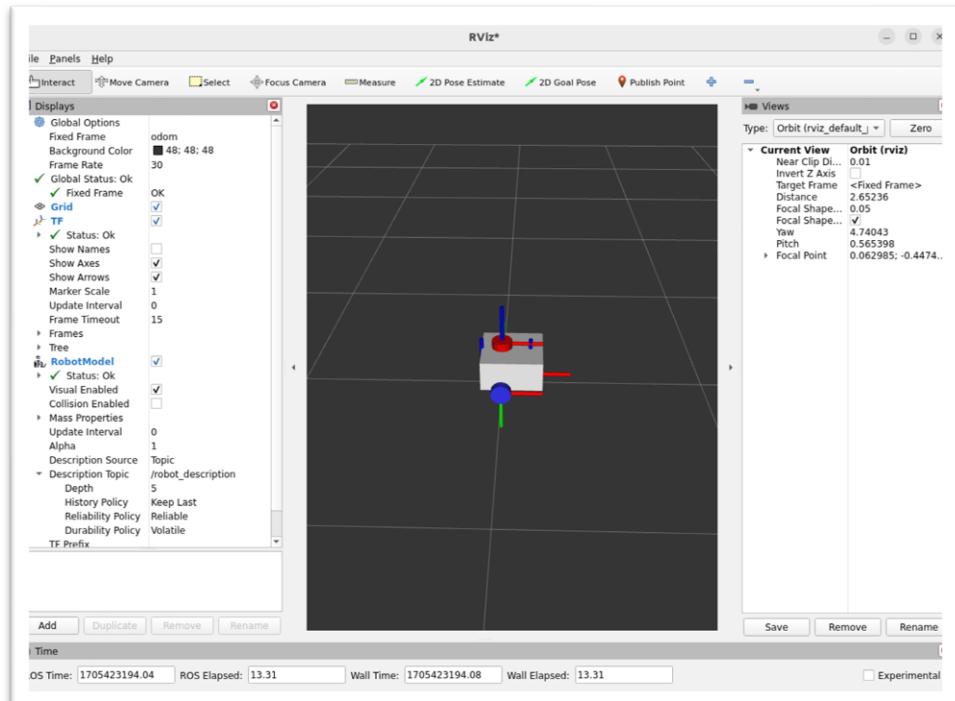
5.4. RViz2

Czym jest *RViz2*

RViz2 to narzędzie wizualizacji dla systemu *ROS2*, które umożliwia użytkownikom interaktywną prezentację danych związanych z robotyką i systemami robotycznymi. Jest następcą oryginalnego narzędzia *RViz2*, które było używane w kontekście *ROS1*.

Integracja *ROS2* i *RViz2*

Pierwszym krokiem do rozpoczęcia pracy z *RViz2* jest pobranie paczki *ROS2* zawierającej wszystkie zależności konieczne do komunikacji. Następnie aktywując program komendą, wyświetla się okno oprogramowania:



Rys. 25 Okno wizualizacyjne modelu zaprojektowanego robota

Układy współrzędnych

Jedną z najważniejszych cech programu jest możliwość sprawdzenia poprawności ustawienia elementów robota, co realizowane jest poprzez zwizualizowanie położen układów współrzędnych skonfigurowanych w pliku URDF (Universal Robotic Description Format) zawartym w folderze środowiska robota. Tematyka konfiguracji, zostanie poruszona w dalszej części pracy dotyczącej implementacji rozwiązania. Wskazanie zakotwiczonego ukł. współrzędnych, czyli opcja *Fixed Frame*, umożliwia zbadanie względnych zmian położenia ukł. współrzędnych, co jest szczególnie przydatne przy badaniu zgodności orientacji w przestrzeni ukł. odniesienia odometrii i kamery lidar w kontekście realizacji algorytmu SLAM.

Tematy (*Topic*)

Klikając w ikonę „Add” użytkownik może określić część robota, którą potrzebuje zwizualizować. Po wybraniu typu wiadomości którą chce przechwycić np. „RobotModel” oraz nazwy Tematu (*Topic*) na której wiadomość(*Message*) jest publikowana, obiekt wyświetla się w programie. Istnieje wiele możliwości parametryzacji wyglądu obiektu np. *mass properties*, czy *collision enabled*. Jeżeli dany Temat (*Topic*) nie istnieje, bądź nie istnieje dana wiadomość(*Message*), program informuje o nie właściwym wykonaniu operacji.

Integracja *RViz2* i *Gazebo*

Programy niezależnie pobierają dane z platformy *ROS2*, dlatego też interakcja z robotem w *Gazebo*, poprzez wysyłanie danych do *ROS2*, prześle wprowadzone zmiany do *RViz2*. Nie istnieje konieczność instalacji dodatkowych paczek – wszystkie dane przetwarzane są przez middleware *ROS2*.

5.5. *GitHub*

Czym jest *GitHub*

GitHub to platforma internetowa służąca do zarządzania projektami opartymi na systemie kontroli wersji Git. Git jest narzędziem służącym do śledzenia zmian w kodzie źródłowym projektu, umożliwiając jednoczesną pracę wielu programistów nad tym samym projektem. *GitHub* dostarcza interfejsu internetowego, który ułatwia współpracę nad projektami, przechowywanie kodu źródłowego oraz zarządzanie nim.

Metodyka pracy z Git

Repozytoria:

- *GitHub* umożliwia tworzenie repozytoriów, czyli miejsc przechowujących kod źródłowy projektu. Każde repozytorium zawiera pełną historię zmian, dzięki czemu można śledzić postęp prac nad projektem.
- Programiści mogą tworzyć oddzielne "gałęzie" (branches) w repozytorium, co pozwala na pracę nad nowymi funkcjami lub poprawkami bez wprowadzania zmian w głównej części projektu. Później te gałęzie mogą zostać „zmergowane” (połączone) z głównym kodem.

Śledzenie Zmian:

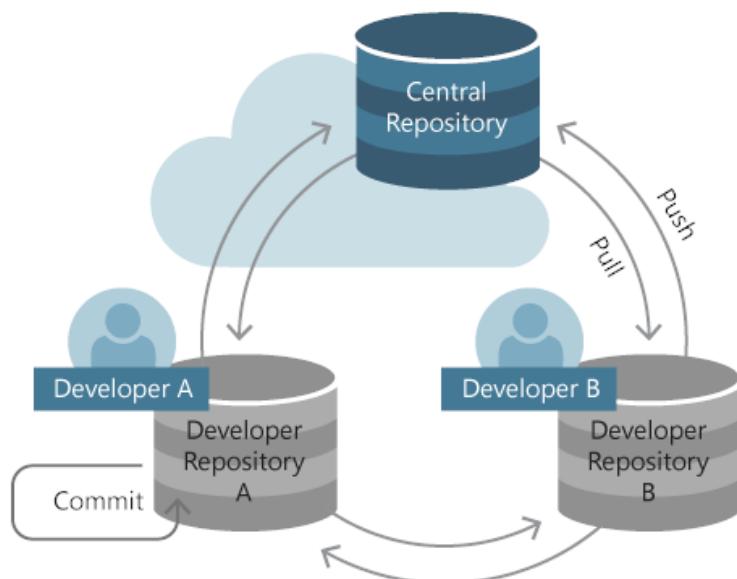
- *GitHub* umożliwia śledzenie zmian w kodzie źródłowym sprawdzanie które linie zostały dodane lub usunięte w danym „commicie” (zmiana w kodzie projektu).

Pull Requests:

- Programiści mogą proponować zmiany w kodzie za pomocą tzw. pull requestów, czyli prośbami widocznymi na platformie *GitHub* z widocznymi zmianami, które pull request wprowadza. Inni członkowie zespołu mogą je przeglądać, komentować i akceptować przed zmergowaniem z głównym kodem.

Bezpieczeństwo:

- Pliki można zabezpieczyć kluczem bezpieczeństwa. Jeżeli programista chce uzyskać dostęp do folderów, musi posiadać odpowiedni klucz prywatny. Jeżeli klucz jest tożsamy z kluczem publicznym ustanowionym przez administratora, weryfikacja uznawana jest za popawną i pliki zostają udostępnione



Rys. 26 Idea działania platformy GitHub [35]

Git w robotyce

W procesie tworzenia oprogramowania dla robotów autonomicznych, często korzysta się z maszyn do rozwoju przed wgraniem systemu na komputer robota np. *Raspberry Pi 4*. Minikomputery nie mają wystarczającej mocy obliczeniowej i odpowiednich komponentów aby przeprowadzić symulacje, dodatkowo pracowanie bezpośrednio na głównym komputerze robota skutkowałoby zaśmiecaniem pamięci niepotrzebnymi plikami. Z tego powodu, najczęściej cały program najpierw tworzy się na komputerze o większych mocach obliczeniowych i lepszych parametrach, które służą do sprawdzania poprawności działania kodu za pomocą symulatorów. Finalnie, poprzez narzędzie kontroli wersji takie jak *GitHub*, przesyła się odpowiednie pliki na maszynę docelową.

6. Zaproponowane rozwiązanie

6.1 Założenia symulacji

Definicja modelu robota:

- Symulacja uwzględnia fizyczne i geometryczne właściwości robota, takie jak wymiary, masa, bezwładność, kształt, tarcie.
- Model 3D robota oparty o format definicji obiektu *URDF* (Unified Robot Description Format).

Oprogramowanie sterujące:

- Implementacja oprogramowania sterującego robotem w środowisku *ROS2*, sygnał sterujący wysyłany z klawiatury oraz nawigacji.

Integracja z ROS2:

- Skonfigurowanie symulacji do współpracy z *ROS2*. Przechwycenie węzłów *ROS2* tworzących platformę robota, przez środowisko symulacyjne.

Modelowanie środowiska:

- Stworzenie środowiska symulacyjnego w *Gazebo*, w którym robot będzie operować. Środowisko zawiera różne przeszkody, pomiędzy którymi robot ma nawigować do wyznaczonego celu w narzędziu *RViz2*.

6.2 System operacyjny komputera robota: *Ubuntu*

Generalne trendy systemów operacyjnych wykorzystywanych w robotyce

W przypadku wyboru dystrybucji systemu operacyjnego, który jest podstawą dla rzeczywistego środowiska robota, które decyduje o sposobie interpretacji informacji, ważnym aspektem jest możliwość jego modyfikacji. Standardowa wersja systemu Windows nie jest stworzona do tego typu projektów – dostęp do kluczowych części architektury jest ograniczony i mało czytelny, dodatkowo nie jest *Open Source*, co powoduje, iż nie ma możliwości pełnego zrozumienia działania wielu elementów. Wiele nakładek graficznych, które są wyjątkowo użyteczne dla użytkownika pracującego na oknach aplikacji, zajmuje niepotrzebne miejsce w pamięci oraz wprowadza dodatkowy zamęt w przypadku przeprowadzania prób zrozumienia działania systemu. Charakter licencji *Open Source* systemu *Linux*, nie ogranicza użytkowników przed tworzeniem różnych dystrybucji przystosowanych do konkretnych zastosowań. Przykłady:

Debian - stabilna i wszechstronna dystrybucja, która jest podstawą dla wielu innych dystrybucji, takich jak *Ubuntu* czy *Linux Mint*. *Debian* to jedna z najstarszych i najbardziej szanowanych dystrybucji *Linux*, znana ze swojego silnego nacisku na stabilność, niezawodność oraz zgodność z filozofią wolnego oprogramowania. W związku z tym *Debian* ma bardzo restrycyjną politykę włączania pakietów do oficjalnych repozytoriów [36].

Ubuntu - Oparte na *Debian*. Używa systemu zarządzania pakietami *APT*. *Ubuntu* to dystrybucja *Linux* stworzona przez firmę Canonical, która od początku stawała na prostotę i łatwość użycia, co przyciągnęło szerokie grono użytkowników. *Ubuntu* oferuje bogate wsparcie dla różnych środowisk graficznych, z których domyślnym jest *GNOME*. W przeszłości korzystano również z własnego środowiska graficznego o nazwie *Unity* [36].



Rys. 27 Logo Ubuntu [37]

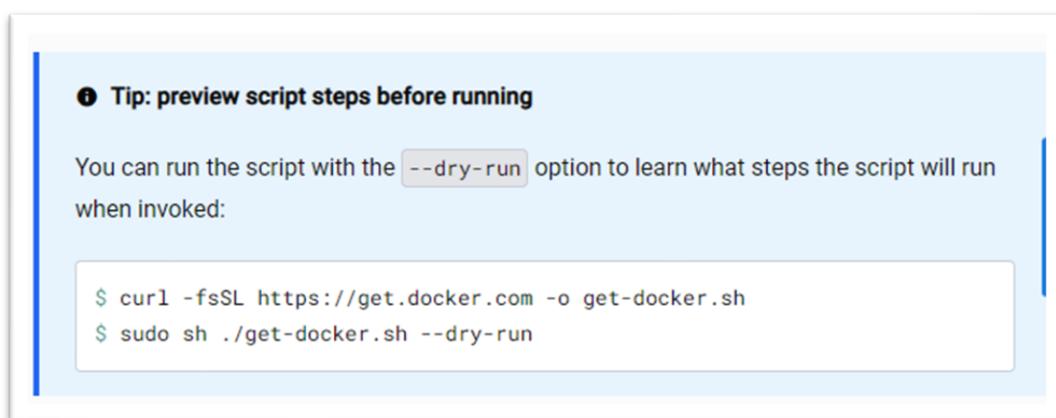
Linux Mint — Dystrybucja oparta na *Ubuntu*, która koncentruje się na prostocie użycia i wygodzie dla początkujących użytkowników. *Linux Mint* oferuje przyjazne środowisko graficzne i szeroką gamę preinstalowanych aplikacji [36].

Wybór *Ubuntu* i przewidziane zastosowanie

W świecie robotyki, najczęściej wybieranym systemem dla maszyn rozwojowych i robotów opartych o komputery industrialne i jest *Ubuntu*, natomiast komputery takie jak *Raspberry pi4* z reguły pracują na *Linux mint*.

Paczki *ROS2* oparte na *Ubuntu* cechują się największą stabilnością i wsparciem aktywnej społeczności. Jest to najlepiej przetestowana i najbardziej rozwinięta dystrybucja w kontekście rozwoju oprogramowania robotów.

Wybór podjęty na podstawie trendów panujących w świecie robotyki oraz posiadaniu doświadczenia pracy ze wspomnianym systemem. Pierwszym etapem pracy z *Ubuntu* jest zainstalowanie narzędzia *Docker* używając pakietów *APT*.



Rys. 28 komendy instalacyjne Docker na systemie Ubuntu [38]

Kolejne etapy pracy związane są z pobieraniem odpowiednich paczek rosowych wewnętrz kontenera oprogramowania *Docker*, którego zależności również będą oparte na systemie plików *Ubuntu*.

```
apt-get update && apt-get install -y nano  
sudo apt install ros-humble-joint-state-publisher-gui -y --no-install-recommends  
sudo apt install ros-humble-slam-toolbox -y --no-install-recommends  
sudo apt install ros-humble-navigation2 ros-humble-bringup ros-humble-turtlebot3*  
sudo apt install ros-humble-twist-mux
```

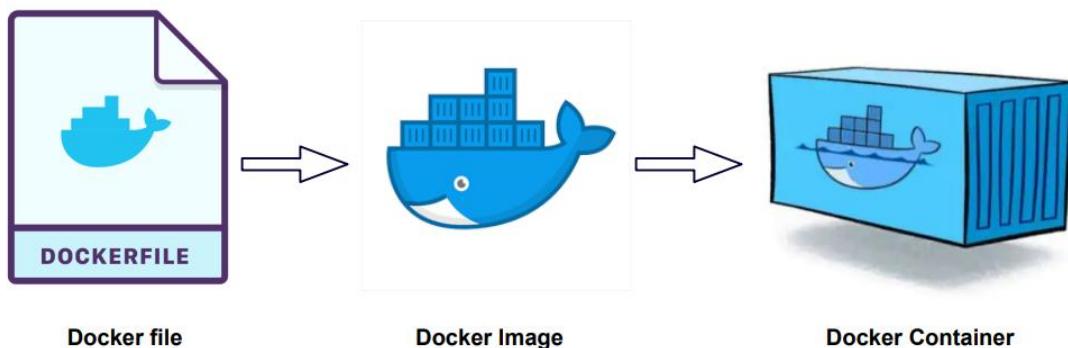
Rys. 29 Wybrane pakiety ROS2 wykorzystane w projekcie robota

6.3 Narzędzie wspomagające rozwój oprogramowania: *Docker*

Dlaczego wykorzystano narzędzie do konteneryzacji?

Proces rozwoju robota wymaga instalacji wielu zależności, aby platforma *ROS2* współpracowała prawidłowo z systemem operacyjnym. W przypadku instalowaniu oprogramowania bezpośrednio na systemie, jeżeli gdzieś popełniono błąd, należy poświęcić nie małą ilość czasu, aby znaleźć przyczynę. W przypadku uruchomieniu platformy wewnętrz kontenera:

- Łatwiej przeanalizować przyczynę, instalacja przebiega na zubożonej wersji systemu operacyjnego, który zawiera tylko wymagane zależności.
- Jeżeli błąd poskutkuje spowodowanie niestabilności systemu OS, wystarczy usunąć kontener, w przypadku bezpośredniej instalacji, należy przeinstalowywać cały system.
- W przypadku *Docker*, społeczność zaangażowana w rozwój oprogramowania do konteneryzacji często udostępnia pliki *Dockerfile*, które są podstawą do budowania obrazów, na podstawie których uruchamia się kontenery. Pliki te definiują wszystkie zależności i aplikacje jakie mają znajdować się w obrazie. Takie rozwiązanie pozwala na wykorzystanie już skonfigurowanego środowiska, bez konieczności samodzielnej instalacji [28].



Rys. 30 Proces tworzenia oprogramowania kontenera [39]

Konteneryzacja w robotyce opiera się na dwóch najpopularniejszych systemach: *Docker* i *Kubernetes*

Docker pozwala na łatwe tworzenie, zarządzanie i uruchamianie kontenerów, jest przyjazny dla osób bez wcześniego doświadczenia z tematyką konteneryzacji. Klarowna dokumentacja i aktywna społeczność programistów rozwijających produkt, jest dużym atutem w kontekście szybkości rozwiązywania niejasności związanych z wdrażaniem rozwiązań opartych na oprogramowaniu. Charakter *Open Source* pozwala na wprowadzanie zmian w kodzie *Docker*, dopasowanych do potrzeb projektu. Liczne *API* ułatwiają automatyzację rozwiązań, a *Docker CLI* (Command Line Interface) w szczególności dla systemu *Ubuntu*, nie jest skomplikowany i intuicyjnie reprezentuje pojęcia związane z konteneryzacją.

API (Interfejs Programowania Aplikacji) to zestaw zdefiniowanych reguł i protokołów, które umożliwiają jednej aplikacji komunikację z inną

Kubernetes, z kolei, to system do automatyzacji i zarządzania kontenerami. Zapewnia funkcje takie jak skalowanie, zarządzanie dostępem, monitorowanie i równoważenie obciążenia. *Kubernetes* pozwala na efektywne zarządzanie dużą liczbą kontenerów, co jest kluczowe w przypadku rozbudowanych aplikacji mikroserwisowych, gdzie istnieje wiele mniejszych komponentów działających niezależnie. Kolejną funkcjonalnością konteneryzacji jest replikacja tych samych procesów – Jeżeli komputer serwera ulegnie awarii, funkcje kontenera zostają przenoszone na inną maszynę, gdzie następuje uruchomienie kontenera spełniającego taką samą funkcjonalność, jaką miał przypisany kontener na wadliwej maszynie.

Wybór *Docker* i przewidziane zastosowanie:

W przypadku budowy prostego projektu robota autonomicznego, głównym zadaniem wykorzystywanej technologii konteneryzacji jest ułatwienie procesu instalacji systemu przy wykorzystaniu *Dockerfile* oraz stworzenie warstwy izolacyjnej pomiędzy komputerem przeznaczonym do rozwoju a systemem robota. Założenia narzędzia *Kubernetes* dotyczą współpracy z większą ilością robotów, całymi parkami maszyn, gdzie kluczowa jest organizacja komunikacji między wieloma kontenerami oraz ich replikacja. Dodatkowo struktura *Kubernetes* jest skomplikowana, funkcjonalności programu przystosowane są do tworzenia wielkich systemów zarządzania kontenerami.

Pierwszym etapem pracy z *Docker* jest pobranie gotowego pliku *Dockerfile* twórcy oraz modyfikacja podporządkowana pod potrzeby projektu. W momencie wpisania wszystkich koniecznych danych w plik *Dockerfile*, zostanie zbudowany obraz, na podstawie którego można uruchomić kontener.

```
# Install ROS2
RUN sudo add-apt-repository universe \
    && curl -sSL https://raw.githubusercontent.com/ros/rosdistro/master/ros.key -o /usr/share/keyrings/ros-archive-keyring.gpg \
    && echo "deb [arch=$(dpkg --print-architecture) signed-by=/usr/share/keyrings/ros-archive-keyring.gpg] http://ros.org/debs $(lsb_release -c -s) main" | tee /etc/apt/sources.list.d/ros-latest.list \
    && apt-get update && apt-get install -y --no-install-recommends ros-foxy-ros-base
```

Rys. 31 Kawałek kodu zapisanego w *Dockerfile*

Następnie korzystając z narzędzia *Docker-compose*, zostanie stworzony plik definiujący jakie dodatkowe właściwości paczki *ROS2* mają być zawarte w kontenerze robota. Likwiduje to konieczność każdorazowej przebudowy obrazu, przy wprowadzaniu zmian na kontenerze robota.

```
version: '3.7'
services:
  my-service:
    image: my-custom-image
    network_mode: "host"
    ipc: host
    entrypoint: /bin/sh
    command:
      - -c
      - |
        apt-get update && apt-get install -y nano
        sudo apt install ros-humble-joint-state-publisher-gui -y --no-install-recommends
        sudo apt install ros-humble-slam-toolbox -y --no-install-recommends
```

Rys. 32 Kawałek kodu zapisanego w pliku Docker-compose

6.4 Platforma *ROS2* oraz integracja z symulatorem *Gazebo*.

6.4.1 Decyzja wyboru *ROS2* jako platformy robotycznej

Sposoby integracji systemów robotów mobilnych

Zagadnienie budowy robota mobilnego polega na rozwiązaniu problemu sposobu interpretacji sygnałów wysyłanych między narzędziami sterującymi i czujnikami, a sterownikami napędów. W przypadku budowy prostych robotów mobilnych, wystarczy zastosować system oparty na mikrokontrolerach takich jak *arduino*, gdzie sygnał z czujnika w prosty sposób przekształcany jest na ruch silników. W przypadku kiedy informacja pobierana z czujnika wymaga większych mocy obliczeniowych dla uzyskania pożądanej interpretacji sygnału, problem staje się bardziej skomplikowany. Rozwiązania skomplikowanych systemów robotycznych polegają na korzystaniu z oprogramowania typu *middleware* składającego się z bibliotek realizujących zadanie interpretacji sygnałów i łączenia elementów systemu w jedną spójną platformę. Przykładem tego typu platformy jest *ROS2*, która została wybrana jako oprogramowanie na podstawie którego oparty jest projekt systemu robotycznego pracy dyplomowej.

***ROS2* to jedyna platforma Open Source dostosowana do rozwoju oprogramowania autonomicznych robotów mobilnych**

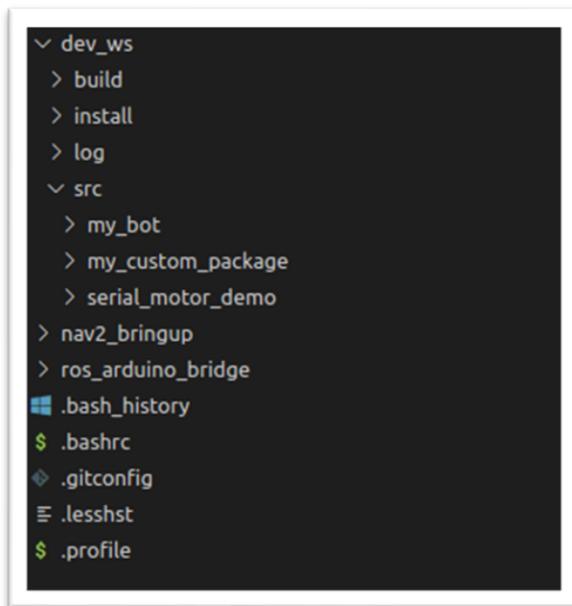
W kontekście autonomicznych robotów mobilnych, jest jedną platformą która umożliwia utworzenie docelowego systemu w dość szybkim tempie. Ogromna ilość projektów dostępnych w internecie, biblioteki umożliwiające integracje większości zaawansowanych urządzeń sensorycznych takich jak kamery czy *LIDAR*. Ciągły rozwój oprogramowania jest obserwowałny poprzez funkcjonalności jakie oferuje względem poziomu integralności z najnowszymi technologiami informatycznymi takimi jak np. *Docker*.

6.4.2 Plan Struktury projektu robota w *ROS2*

Środowisko bazowe projektu

Bazowy projekt Robota – konfiguracje poszczególnych elementów takich jak podwozie, koła i LIDAR, zostały utworzone na podstawie środowiska gotowego robota twórcy ArticulatedRobotics, którego kod jest umieszczony na koncie GitHub: github.com/joshnewans/articubot_one [40].

Pierwszym etapem pracy z systemem ROS2 jest przygotowanie środowiska. Folder składa się z 4 głównych komponentów – *build*, *install*, *log* i *src*. Pozostałe pliki widoczne na zdjęciu nie są konieczne do omówienia, gdyż zawierają dane, które stanowią automatycznie wygenerowane informacje przy budowie środowiska, które nie wpływają bezpośrednio na strukturę kodu.



Rys. 33 Struktura plików projektu

Build – Przechowywanie plików tymczasowych, artefaktów komplikacji i innych generowanych danych podczas procesu budowania projektu.

Install - Przechowywanie zainstalowanych plików i komponentów danego pakietu lub zestawu pakietów.

Log - Przechowywanie plików dziennika generowanych przez różne komponenty działające w systemie ROS2.

Src - Przechowywanie kodu źródłowego związanego z danym pakietem.

Folder my_bot zawiera definicję struktury fizycznej robota oraz pliki skryptów potrzebnych do uruchomienia symulacji robota.

Definicja Fizyki elementów

Posiadając gotowe środowisko, należy określić jak dany robot ma wyglądać, jak części mają być ulokowane względem siebie, jaką pełnią funkcję oraz jakie właściwości fizyczne je charakteryzują. Informacje te zdefiniowano w plikach *URDF* (Unified Robot Description Format), których wzajemna interakcja zostanie zaprojektowana w konkretnej hierarchii:

```

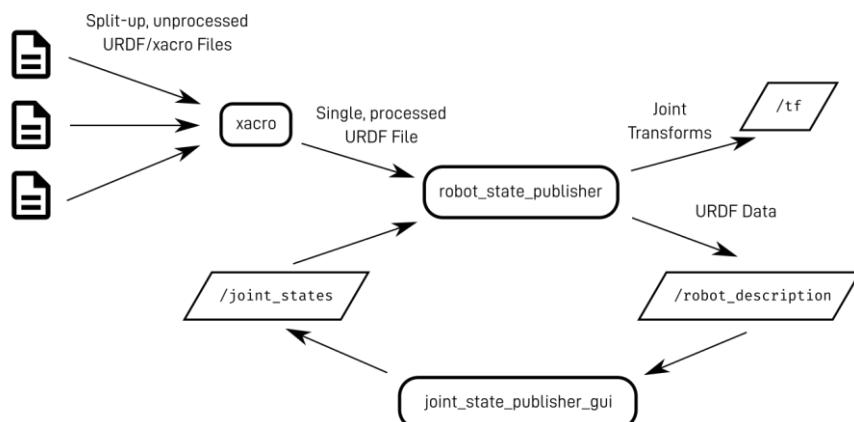
XPLORER ... Nav_Stop_mechanism.py ! mapper_params_online_async.yaml u
OOT [CONTAINER MY-CUSTOM-IMAGE (MY_PROJECT-MY...]
... .vcs.xml
... .ros
... .rviz2
... .sdformat
... .ssh
... .vscode-server
... dev_ws
... > build
... > install
... > log
... > src
... > my_bot
... > config
... > description
... | gazebo_control.xacro

Nav_Stop_mechanism.py
! mapper_params_online_async.yaml u
robot.urdf.xacro
<?xml version="1.0"?>
<robot xmlns:xacro="http://www.ros.org/wiki/xacro" name="...
<!-- Example link -->
<xacro:include filename="robot_core.xacro"/>
<xacro:include filename="inertial_macros.xacro"/>
<xacro:include filename="gazebo_control.xacro"/>
<xacro:include filename="lidar.xacro"/>
</robot>

```

Rys. 34 Główny plik *URDF*

Struktura przesyłu informacji w *ROS2* wymaga, aby istniał jeden plik *URDF*. Linia kodu `<xacro:include filename=<urdf_file>/>` określa kolejne pliki *xacro*, z których ma się składać główny plik *URDF*. W folderze *description* definowane są tzw. przeguby ang. *Joints*, które określają relacje między układami wspł. elementów oraz tzw. połączenia ang. *Link*, które definiują ich istnienie, a w przypadku korzystania z symulacji, również geometrię oraz właściwości fizyczne [41].



Rys. 35 Struktura udziału plików *URDF* w sieci *ROS2* [41]

Informacje wysyłane są do narzędzia *robot_state_publisher*, który udostępnia na *Topic* `/tf` pozycje ukł. wspł. wszystkich elementów oraz opis ich aktywnych parametrów na *Topic*

/robot_description. Joint_state_publisher_gui wychwytuje zmiany w pozycjach elementów i przesyła na Topic /joint_states, który aktualizuje informacje wysyłane do robot_state_publisher o rzeczywistą pozycję robota [41].

Link

Stworzenie *Link* jest tożsame z utworzeniem nowego elementu robota. W przypadku symulacji, należy zdefiniować parametry takie jak *visual*, *collision*, *inertial*.

<visual> – definicja wymiarów robota widocznego w symulacji / wizualizacji. Składa się z parametrów <geometry> (definicja rodzaj geometrii obiektu np. box), <origin> (definicja offsetu środka geometrii), <material> (kolor obiektu).

<collision> – wymiary geometrii przestrzeni, które uwzględniane są podczas obliczeń przebiegu kolizji robota z otoczeniem. Parametry takie same jak w *visual*.

<Inertial> - Definicja fizycznych właściwości elementu. Składa się z parametrów: <mass>(masa elementu), <origin>(centrum masy), <inertia> (Tensor inercji) [41].

```
<link name="chassis">
  <visual>
    <origin xyz="0.15 0 0.075"/>
    <geometry>
      <box size="0.3 0.3 0.15"/>
    </geometry>
    <material name="white"/>
  </visual>
  <collision>
    <origin xyz="0.15 0 0.075"/>
    <geometry>
      <box size="0.3 0.3 0.15"/>
    </geometry>
    <material name="white"/>
  </collision>
  <xacro:inertial_box mass="0.5" x="0.3" y="0.3" z="0.15">
    <origin xyz="0.15 0 0.075" rpy="0 0 0"/>
  </xacro:inertial_box>
</link>
```

Rys. 36 Struktura połączenia ang. Link podwozia robota

Przeguby

Stworzenie przegubu określa, jak dany *Link* ma być zorientowany do innego *Link*. Strukturę zależności przegubów definiuje się względem *base_link*, który najczęściej skojarzony jest z stałą częścią robota np. podwoziem. Każdy przegub jest określany poprzez parametry: *name*, *type*, *parent*, *child*, *origin* [41].

<Name> - nazwa elementu

<type> - typ elementu np. *prismatic*, *revolute*, *continuous*

<parent> i <child> - definiowanie relacji między *Link*, które łączy dany przegub

<origin> - orientacja między *Link* zanim układ zostanie wprowadzony w ruch.

```
<joint name="chassis_joint" type="fixed">
  <parent link="base_link"/>
  <child link="chassis"/>
  <origin xyz="-0.1 0 0"/>
</joint>
```

Rys. 37 Struktura przegubu podwozia robota

W przypadku przegubów inne niż *fixed* czylinieruchome, definiuje się również górne i dolne ograniczenie ruchu, limity prędkości oraz obciążenia.

Znajomość powyższej metodyki modelowania elementów w platformie *ROS2*, zostanie wykorzystana do utworzenia:

- Podwozia
- Dwóch kół
- Koła pomocniczego
- Lidaru

Połączenie modelu robota z symulatorem Gazebo

Procedura polega na zaimplementowaniu gotowej wtyczki do symulacji znajdującej się w bibliotece *ROS2 „libgazebo_ros_diff_drive.so”*. Dotyczy modelu robota osadzonego na dwóch kołach. Aby symulacja działała prawidłowo, wystarczy dostosować parametry znajdujące się pod komentarzem <!— Wheel Information --!> [42].

```

plugin name="diff_drive" filename="libgazebo_ros_diff_drive.so"

    <!-- Wheel Information -->
    <left_joint>left_wheel_joint</left_joint>
    <right_joint>right_wheel_joint</right_joint>
    <wheel_separation>0.35</wheel_separation>
    <wheel_diameter>0.1</wheel_diameter>

    <!-- Limits -->
    <max_wheel_torque>200</max_wheel_torque>
    <max_wheel_acceleration>10.0</max_wheel_acceleration>

    <command_topic>cmd_vel</command_topic>

    <!-- Output -->
    <odometry_frame>odom</odometry_frame>
    <robot_base_frame>base_link</robot_base_frame>
    <publish_odom>true</publish_odom>
    <publish_odom_tf>true</publish_odom_tf>
    <publish_wheel_tf>true</publish_wheel_tf>

```

Rys. 38 konfiguracja wtyczki ROS2 dla symulatora gazebo [42]

Dołączenie symulatora LIDAR

Podobnie jak w przypadku łączeniu elementów modelu, należy skorzystać z gotowej wtyczki symulatora skanera LIDAR znajdującej się w bibliotece „libgazebo_ros_ray_sensor.so”. Należy zdefiniować *Topic* na którym publikowane są sygnały wychodzące ze skanera (~out) i jak ten sygnał ma zostać sparametryzowany (~out:=scan). <scan> zawiera parametry o odległości skanu i zakresu obrotu skanera. Parametr <output_type> określa typ *Topic* (sensor_msgs/LaserScan – czyli chmura punktów) [43].

```

<scan>
    <horizontal>
        <samples>360</samples>
        <min_angle>-3.14</min_angle>
        <max_angle>3.14</max_angle>
    </horizontal>
</scan>
<range>
    <min>0.3</min>
    <max>12</max> <!-- Changed from mix to max -->
</range>
</ray>
<plugin name="laser_controller" filename="libgazebo_ros_ray_sensor.so">
    <ros>
        <argument>~out:=scan</argument>
    </ros>
    <output_type>sensor_msgs/LaserScan</output_type>
    <frame_name>laser_frame</frame_name>

```

Rys. 39 konfiguracja parametrów LIDAR wysłana do Gazebo [43]

Skaner zostanie zdefiniowany jako element, zgodnie z zasadami opisanymi w wątku pracy dyplomowej: definicja fizyki elementów, a nazwa *Link* lidar'u to: laser_frame.

```

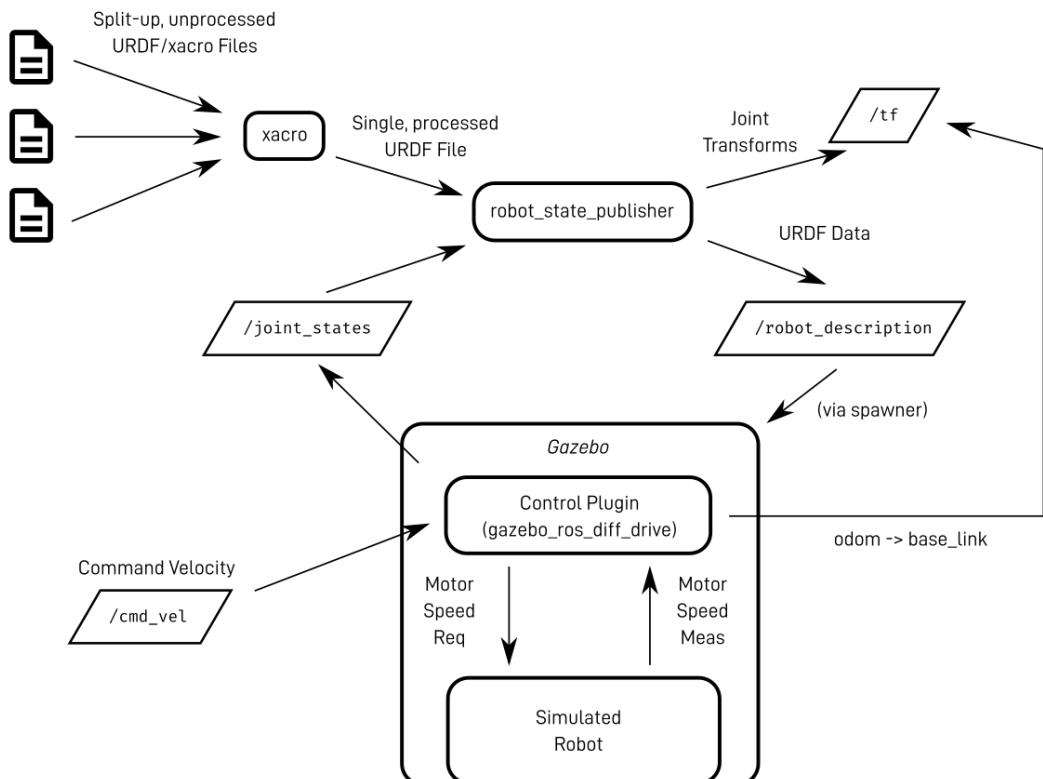
<link name="laser_frame">
  <visual>
    <geometry>
      <cylinder radius="0.05" length="0.04"/>
    </geometry>
    <material name="red"/>
  </visual>
  <collision>
    <geometry>
      <cylinder radius="0.05" length="0.04"/>
    </geometry>
    <material name="blue"/>
  </collision>
  <xacro:inertial_cylinder mass="0.1" length="0.04" radius="0.05">
    <origin xyz="0 0 0" rpy="0 0 0"/>
  </xacro:inertial_cylinder>

```

Rys. 40 konfiguracja przegubu i połączenia skanera [43]

System sterowania

W przypadku korzystania z wtyczki gazebo_ros_diff_drive, informacje na temat zmiany położenia są wysyłane z symulowanego robota na *Topic/joint_states*. Dodatkowo generowane jest nowe położenie – *odom*. Jest to miejsce punktu startowego robota, którego różnica z *Link base_link* zostaje publikowana jako oddzielny *Tf* (joint transform) na *Topic /tf*. Ta transformacja jest kluczowym elementem algorytmu SLAM, którego sposób działania zostanie poruszony w dalszej części pracy. *Topic /cmd_vel* to najważniejsza część poniższego diagramu – to właśnie w to miejsce, zostają wysyłane sygnały opisujące parametry wprawiające robota w ruch [42].



Rys. 41 Schemat komunikacji między ROS2 oraz symulatorem Gazebo [42]

Narzędziem, które posłuży za sterowanie robota jest: *teleop_twist_keyboard*. Po aktywacji narzędzia na ekranie wyświetla się okno informujące o tym, jaki klawisz klawiatury jest odpowiedzialny za jaki ruch. Po wciśnięciu odpowiedniego przycisku na klawiaturze, sygnał zostaje odpowiednio przekształcony i wysłany na *Topic /cmd_vel*.

Nawigacja i lokalizacja

Wyposażenie robota w funkcjonalność lokalizacji polega na zainstalowaniu biblioteki lokalizacyjnej oraz uruchomieniu odpowiedniego skryptu. Dodatkowo ważne jest, aby odpowiednio ustawiać parametry funkcji zależnie od potrzeb. W momencie w którym robot znajduje się w nowej przestrzeni, warto ustawić tryb lokalizowania na budowanie mapy. W przeciwnym wypadku, można skorzystać z ówcześnie zbudowanej poprzez modyfikacje pliku *yaml*:

```
# ROS Parameters
odom_frame: odom
map_frame: map
base_frame: base_footprint
scan_topic: /scan
use_map_saver: true
mode: localization

# if you'd like to immediately start continuing a map at a given pose
# or at the dock, but they are mutually exclusive, if pose is given
# will use pose
map_file_name: /home/my_map_serial
```

Rys. 42 Pliki konfiguracyjne algorytmu SLAM znajdujące się w projekcie

W mode: *localization* lub *map*.

W *map_file_name* należy wpisać ścieżkę do zapisanej mapy.

Funkcjonalność nawigacji dodaje się podobnie, należy zainstalować odpowiednie biblioteki i zoptymalizować parametry nawigacyjne zależnie od potrzeb.

Plik typu *launch* uruchamiający symulację

Standardową procedurą uruchomienia platformy robota, jest aktywacja poszczególnych węzłów za pomocą komendy: ros2 run. W przypadku korzystania z wielu modułów, warto skorzystać z pliku typu *launch*, który może włączyć wszystkie potrzebne węzły jednocześnie. W pracy skorzystano z gotowego rozwiązania twórcy *ArticulatedRobotics*, który włącza bazowy model robota, którego modyfikacje o moduł bezpieczeństwa zostanie opisana w dalszej części pracy.

6.5. Algorytm lokalizacji robota: SLAM

Stosowane Algorytmy lokalizacyjne w robotyce

Zależnie od zastosowania, roboty wyposażone są w różne systemy lokalizacyjne wykorzystujące odpowiednie technologie. Przykładowo, telefony komórkowe zawierają moduły GPS, które zależnie od jakości modelu, wskazują z określona dokładnością na obecną pozycję. Sygnały WiFi znajdują zastosowanie jako nośnik informacji o obecności maszyny w pobliżu obiektu: np. podajnik opon przydziela zadanie poboru opony danemu robotowi. Dopóki robot A jest widoczny w sieci podajnika, żaden inny robot nie może wjechać w strefę otaczającą obiekt. Kiedy robot A pobierze oponę i odjedzie, kolejka zostaje resetowana, sygnał wifi robota A zanika i robot B może podjechać odebrać następną oponę. Kolejnym rodzajem rozwiązań lokalizacyjnych, są systemy wykorzystujące obecną pozycję urządzenia jako informacje niezbędną do planowania ruchu w kierunku zadanego punktu. Najprostszym jest system oparty na GPS. Robot przemieszcza się z punktu A do punktu B, gdzie celem jest uzyskanie jak najmniejszej różnicy odległości, poprzez wykorzystanie prostych algorytmów nawigacyjnych.

Zaawansowane rozwiązania, polegają na połączeniu informacji zeskanowanego otoczenia w połączeniu z danymi pochodzącyymi z innych sensorów np. *odometrii*, które określają położenie robota względem istniejącej już mapy, bądź budują jej przybliżony obraz. Biblioteki ROS2 udostępniają metody: *SLAM* i *AMCL*

AMCL

Algorytm *AMCL* (Adaptive Monte Carlo Localization) to metoda, którą robot używa do znalezienia swojego położenia na zapisanej mapie.

Robot ocenia, jakie miejsca lokalizacji są najbardziej prawdopodobne, bazując na informacjach z czujników, takich jak kamery czy czujniki odległości, które wychwytują pozycje przeszkód. Następnie na podstawie tych danych aproksymuje swoją pozycję.

AMCL dostosowuje swoje przewidywanie do zmian w otoczeniu i błędów w ruchu, dostosowując się do nowych warunków.

Algorytm używa metody próbkowania *Monte Carlo*, czyli tworzenia wielu "próbek" miejsc, w których robot może się znajdować, i przypisywania wag tym, które są bardziej prawdopodobne [44].

SLAM

SLAM (Simultaneous Localization and Mapping) to algorytm używany do jednoczesnego określania pozycji robota i tworzenia mapy otoczenia nawet w przypadku, gdy nie znana jest pozycja początkowa.

Robot porusza się po otoczeniu, zbierając dane z czujników takich jak kamery, lidary, czy inne czujniki odległości.

Czujniki zbierają informacje o otoczeniu, takie jak odległości do ścian, kształty, punkty charakterystyczne itp.

Algorytm stara się określić, lokalizacje w danym czasie, korzystając z danych z czujników oraz informacji o ruchu pobranych z *odometrii*.

Na podstawie zebranych danych, algorytm buduje mapę otoczenia reprezentując ściany, przeszkody, czy inne charakterystyczne punkty.

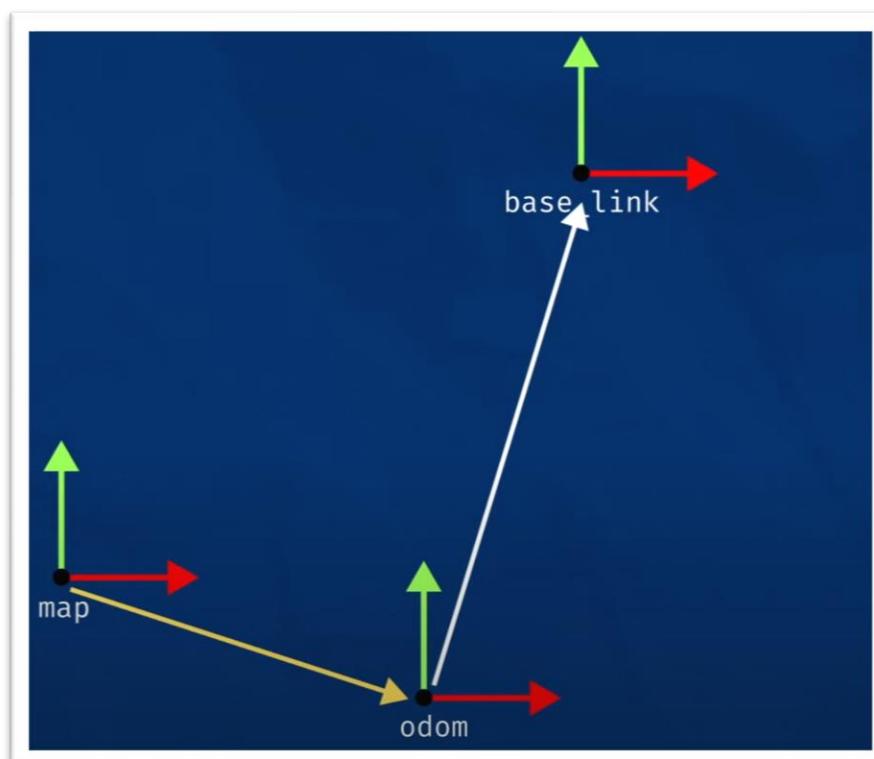
W trakcie ruchu po otoczeniu, jednocześnie aktualizowana jest pozycja robota co umożliwia skuteczną lokalizację nawet w nieznanych wcześniej miejscach.

Algorytm musi radzić sobie z niepewnościami związanymi z pomiarami czujników, błędami *odometrii*, czy zmianami w otoczeniu.

Ze względu na uniwersalność tej metody, zostanie ona wybrana jako metoda lokalizacji robota [44].

SLAM w ROS2

Biblioteka odpowiedzialna za zintegrowanie algorytmu *SLAM* z systemem sieci węzłów robota w *ROS2*, to *slam_toolbox*. Wykorzystuje informacje z trzech *Link*: *base_link*, *map* i *odom*. *Base_link* to układ wspł. podwozia robota, *map* to ukł. współrzędnych początku mapy, a *odom* to ukł. współrzędnych, z którego robot zaczynał ruch. Ukł. wspł. *odom*, ze względu na niedokładne pomiary odometrii, cały czas będzie zmieniał swoje położenie względem ukł. *map* co spowoduje desynchronizację chmury punktów kamery lidar związanej z ukł. *base_link*. Rozwiązaniem jest ciągła transformacja ukł. *map* względem ukł. *odom* [44].



Rys. 43 Transformacje między przegubem *map* i *odom* w *ROS2* podczas realizacji lokalizacji opartej na *SLAM* [44]

Dodatkowo, aktywacja algorytmu *SLAM* tworzy dwa nowe *Topic*: /odom i /map
/odom – zawiera wartość pozycji robota względem ukł. oraz informację o prędkości robota.
/map – zawiera dane siatki punktów budowanej i aktualizowanej mapy .Odnoszą się do ukł. wspł. map [44].

6.6. Zagadnienia nawigacyjne: NAV2

Algorytmy nawigacyjne

Nawigacja w robotyce odnosi się do zdolności robota do poruszania się w środowisku w celu osiągnięcia określonych celów. W zależności od złożoności zadania i charakterystyki otoczenia, nawigacja może obejmować różne aspekty, takie jak lokalizacja, planowanie trasy i unikanie przeszkód.

Lokalizacja:

- Określenie pozycji robota w przestrzeni jest kluczowe dla skutecznej nawigacji. Wykorzystywane są różne techniki, opisane w wątku o lokalizacji w pracy dyplomowej.

Planowanie Trasy:

- Po określeniu aktualnej lokalizacji, robot musi zaplanować trasę do celu. Algorytmy planowania trasy, takie jak A^* , czy Dijkstra pomagają znaleźć optymalną ścieżkę, uwzględniając przeszkody i inne ograniczenia.

Unikanie Przeszkód:

- Roboty muszą posiadać zdolność unikania przeszkód w czasie rzeczywistym. Algorytmy unikania przeszkód, oparte na mapach potencjałów, pozwalają robotowi elastycznie reagować na zmiany w otoczeniu.

Algorytm Dijkstry

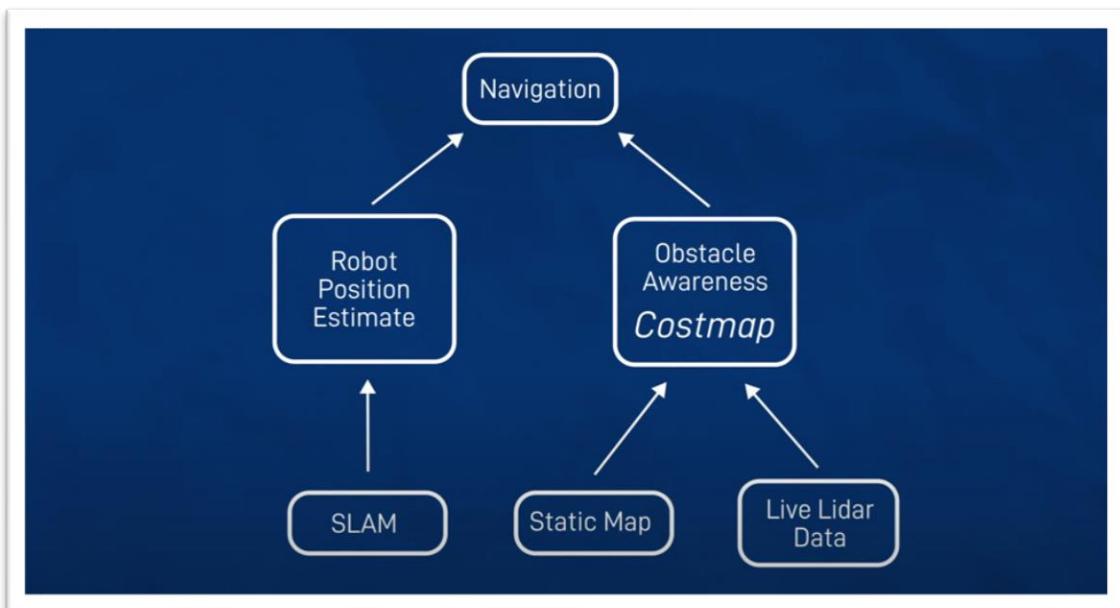
Metoda rozwiązywania problemu najkrótszej ścieżki w grafie. Rozpoczynając od wybranego wierzchołka startowego, iteracyjnie odnajduje najkrótsze ścieżki do wszystkich pozostałych wierzchołków, minimalizując koszt dotarcia do każdego z nich.

Algorytm A^*

zaawansowana technika znajdowania najkrótszej ścieżki, uwzględniająca heurystykę. Korzysta z kombinacji rzeczywistego kosztu dotarcia do danego wierzchołka i funkcji heurystycznej szacującej koszt dojścia do celu. W trakcie iteracyjnego przeszukiwania grafu, algorytm wybiera wierzchołki na podstawie minimalnej wartości funkcji oceny, łącząc efektywność Dijkstry z dodatkową informacją heurystyczną, co umożliwia bardziej intelligentne podejście do przeszukiwania przestrzeni stanów.

Biblioteka NAV2

Platforma ROS2 zawiera jedną obszerną bibliotekę nawigacyjną, dlatego tylko ta zostanie omówiona. NAV2 jest dystrybuowana na licencji Open Source, dodatkowo można znaleźć jej dokładnie opisaną dokumentację na stronie *GitHub*. Głównym algorytmem planowania trasy, czyli obliczania map kosztów jest Dijkstra.

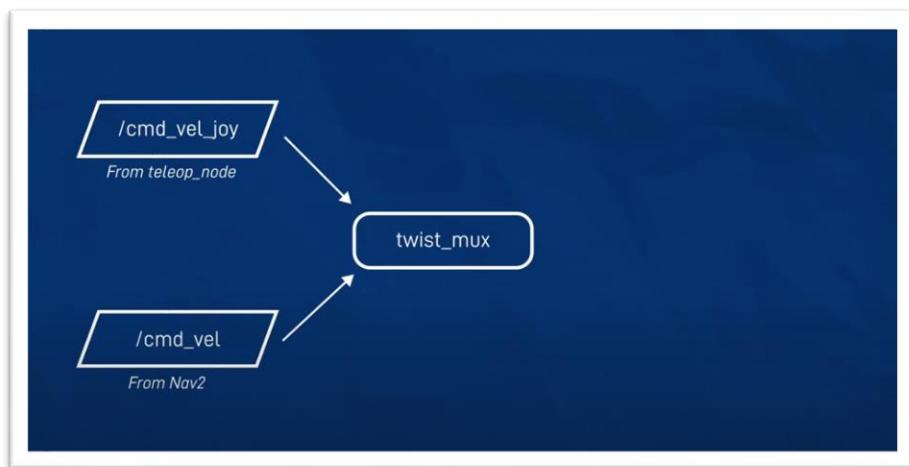


Rys. 44 Zasada działania nawigacji opartej na NAV2 w ROS2 [45]

Pierwszą podstawową informacją konieczną do inicjalizacji nawigacji, jest uzyskanie obecnej pozycji robota. Realizacja zadania przydzielona zostanie algorytmowi *SLAM*, który udostępnia estymacje położenia robota z określoną częstotliwością. Następnie należy zdefiniować, jak interpretować otaczające robota środowisko. Przy omawianiu algorytmu *SLAM*, wskazano iż *Topic /map* zawiera dane o pozycjach aktualnie widocznych elementów wykrytych przez skaner oraz tych, które zostały zarejestrowane w poprzednich chwilach czasu. *Topic /map* połączony zostaje z danymi o najnowszych danych wysyłanych ze skanera LIDAR. Tworzona zostaje mapa kosztów, która połączona z estymowaną pozycją robota, składa się na wykorzystaną w projekcie strukturę nawigacyjną. Sprecyzowanie celu jazdy robota, zostanie nadane przez skrypt systemu bezpieczeństwa – temat omówiony w dalszej części pracy [45].

Priorytety sterowania: klawiatura i nawigacja

Gdy nawigacja jest aktywna, w przypadku załączenia sterowania zadanego z narzędzia `teleop_twist_keyboard`, dochodzi do wzajemnego nałożenia dwóch informacji, co może poskutkować błędem. Aby rozwiązać ten problem, instaluje się dodatkową bibliotekę `twist_mux`, której pliki konfiguracyjne pozwalają określić pierwszeństwo sygnałów [45].



Rys. 45 Schemat przedstawiający idee działania narzędzia `twist_mux`

Parametry nawigacji

NAV2 można sparametryzować, korzystając z załączonych w strukturze plików o rozszerzeniu .yaml. Zależnie od potrzeb projektowych, funkcjonowanie nawigacji jest możliwe do modyfikacji na różnych płaszczyznach, rozpoczynając od zmian prędkości względem zmieniającego się otoczenia, a kończąc na modyfikacji algorytmów wyliczających mapy kosztów.

<code>nav2_params.yaml</code>	<code>U</code>	216	<code>observation_sources: scan</code>
<code>twist_mux.yaml</code>	<code>U</code>	217	<code>scan:</code>
<code>view_bot.rviz</code>		218	<code>topic: /scan</code>
<code>description</code>		219	<code>max_obstacle_height: 2.0</code>
<code>camera.xacro</code>		220	<code>clearing: True</code>
<code>depth_camera.xacro</code>		221	<code>marking: True</code>
<code>face.xacro</code>		222	<code>data_type: "LaserScan"</code>
<code>gazebo_control.xacro</code>		223	<code>static_layer:</code>
<code>inertial_macros.xacro</code>		224	<code>plugin: "nav2_costmap_2d::StaticLayer"</code>
<code>lidar.xacro</code>		225	<code>map_subscribe_transient_local: True</code>
		226	<code>inflation_layer:</code>

Rys. 46 Część plików konfiguracyjnych parametrów nawigacji robota

6.7 Środowisko wizualizacyjne: RViz2

Zastosowanie narzędzia

RViz2 to narzędzie wizualizacyjne używane w *ROS2* do analizy i interakcji z danymi związanymi z robotyką w trakcie działania.

Wizualizacja Danych:

- *RViz2* umożliwia wizualizację różnych rodzajów danych związanych z robotyką, takich jak chmury punktów, mapy, model kinematyczny robota, trasy planowanej nawigacji, czy obrazów z kamer.

Interakcja z Robotem i debugowanie

- Narzędzie to pozwala na interakcję z robotem w czasie rzeczywistym. Można kontrolować roboty, zmieniać parametry działania, czy nawet planować ruchy i monitorować ich wykonanie.
- Jest powszechnie używane do debugowania i testowania algorytmów nawigacyjnych, planowania ruchu, percepji, czy innych modułów robotycznych.

Zostanie zastosowane w projekcie w celach testu: Poprawności modelu robota, poprawności mechanizmu sterującego, funkcjonalności lokalizacji i nawigacji.

Test modelu robota

- Wizualizacja kinematyki i modelu w przestrzeni. Zostaną wykorzystane narzędzia takie jak "RobotModel" lub "RobotState" w celu dodania modelu robota publikowanego na *Topic /robot_description*

Test sterowania

- Jeżeli zadając sygnały sterujące z klawiatury, korzystając z narzędzia *teleop_twist_keyboard* zwizualizowany model będzie wykonywał ruch, test zostanie uznany za udany.

Test lokalizacji

- Wykorzystane zostanie narzędzie "Map" w panelu "Add" do wizualizacji zeskanowanej mapy. Jeżeli chmura punktów nie nakłada się na zarysy przeszkód widocznych na mapie, robot nie lokalizuje się prawidłowo.

Test nawigacji

- Klikając na narzędzie „2D Goal Pose” zostanie określona docelowa pozycja robota. Jeżeli robot zacznie przemieszczać się w kierunku celu oraz zatrzyma się w zamierzonym miejscu, nawigacja działa prawidłowo. Dodatkowo, *RViz2* oferuje wiele narzędzi wizualizacji dynamicznie aktualizującej się mapy kosztów.

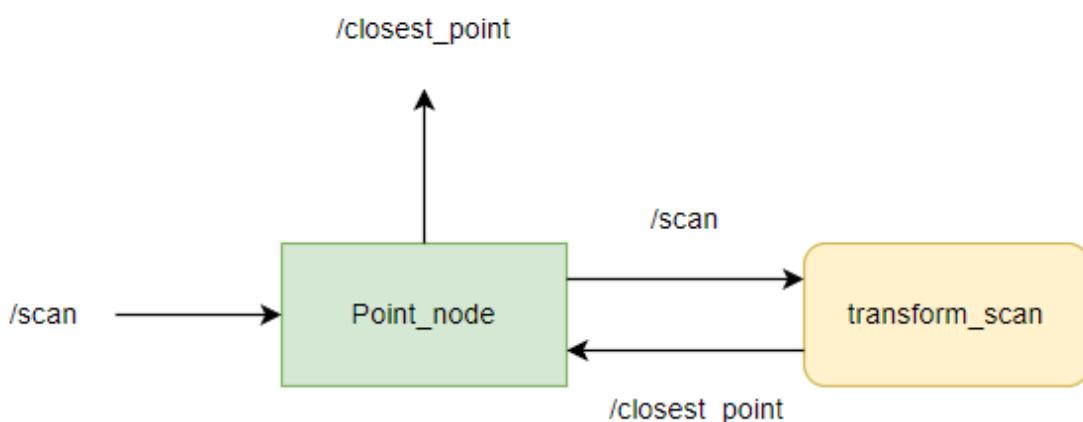
6.8 System bezpieczeństwa oparty o skaner LIDAR

Zasada działania

W dynamicznym środowisku, może nadarzyć się sytuacja, w której nagle w niebezpiecznie bliskiej odległości względem robota pojawi się przeszkoda. Przykładem może być człowiek, który nie spostrzegł przemieszczającego się AMR, gdy chciał przejść na drugą stronę trasy ruchu w fabryce. Sytuację w której robot również powinien się zatrzymać może być np. blokująca trasę paleta, która spadła z mocowania narzędzi. Skaner lidar wysyła dane w postaci chmury punktów na Topic /scan. Posiadając informację o pozycji każdego z punktów, można wybrać jeden z nich, który znajduje się najbliżej. Na podstawie tej informacji można wywnioskować, czy robot jest w stanie bezpiecznie zaplanować trasę dojazdu do celu. Skrypt realizujący składa się z dwóch węzłów. Pierwszy publikuje informację o najbliższym punkcie, drugi decyduje o załączeniu systemu bezpieczeństwa.

Węzeł publikujący najbliższy punkt

Węzeł subskrybuje Topic /scan, który wysyłany jest do funkcji wykonującej operacje transformacji trygonometrycznej każdego punktu z chmury punktów. Odległość liczona jest względem środka modelu robota. Założono iż ukł. wspł skanera jest usadzony w tym samym miejscu i pozycji, co środek modelu. Informacja o tym gdzie znajduje się najbliższy punkt, jest publikowana przez węzeł na Topic /closest_point. Funkcja transformująca dane ze skanera, działa do momentu wyłączenia węzła z sieci ROS2, co przy obecnej konfiguracji, jest równoważne z wyłączeniem robota lub skryptu. Aby aktywować węzeł, należy skorzystać z komendy: ros2 run lub uruchomić skrypt bezpośrednio poprzez interfejs Pythona.



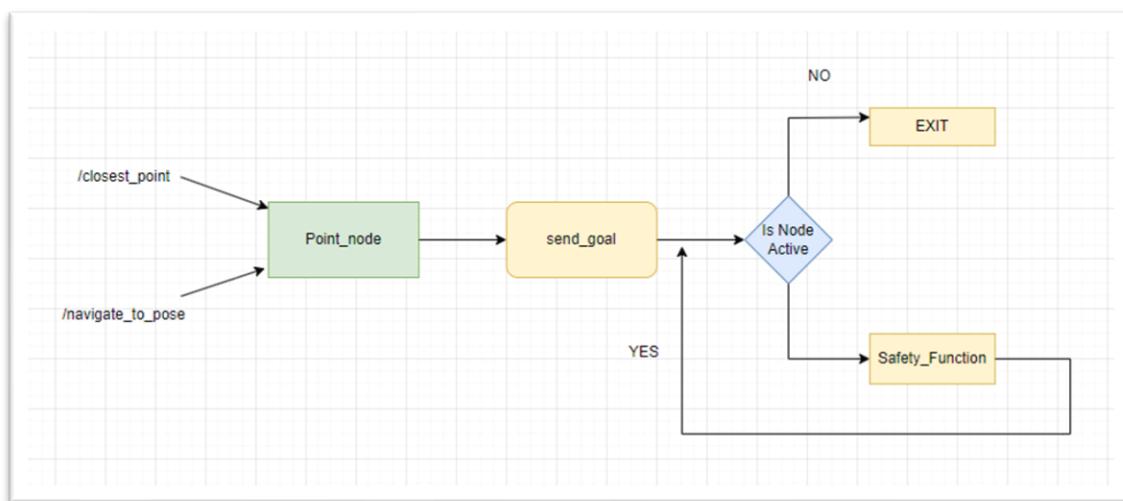
Rys. 47 Schemat ideowy węzła publikującego najbliższy punkt chmury punktów zebranej przez LIDAR

Węzeł kontrolujący aktywację systemu bezpieczeństwa na podstawie klienta akcji

Węzeł obsługujący system bezpieczeństwa, subkrybuje *Topic /goal_pose* i */closest_point*. Pierwszy z wymienionych zawiera informację o zadanym punkcie stanowiącym cel dla nawigacji. Jeżeli cel jest zadany, skrypt publikujący *Topic /closest_point* i kolejny warunek, czyli serwer akcji */navigate_to_pose* jest aktywny, zostaje wywołana funkcja sterująca sygnałem systemu bezpieczeństwa. Pierwsza część systemu sprawdza, czy istnieje punkt, który znajduje się w krytycznej odległości względem robota. Sytuacja podlega sprawdzeniu do momentu, kiedy wartość warunku zmieni się na przeciwną, klient akcji wysyła zapytanie o zatrzymanie nawigacji oraz zapisuje docelową pozycję.

Gdy klient akcji otrzyma odpowiedź o pozytywnym przyjęciu zlecenia, nawigacja przestaje sterować ruchem robota i wywoływana jest kolejna pętla, która sprawdza, czy warunek aktywacji systemu bezpieczeństwa dalej jest spełniony. Jeżeli nie jest, klient akcji ponownie wysyła zapytanie, tym razem oczekując informacji o załączeniu nawigacji.

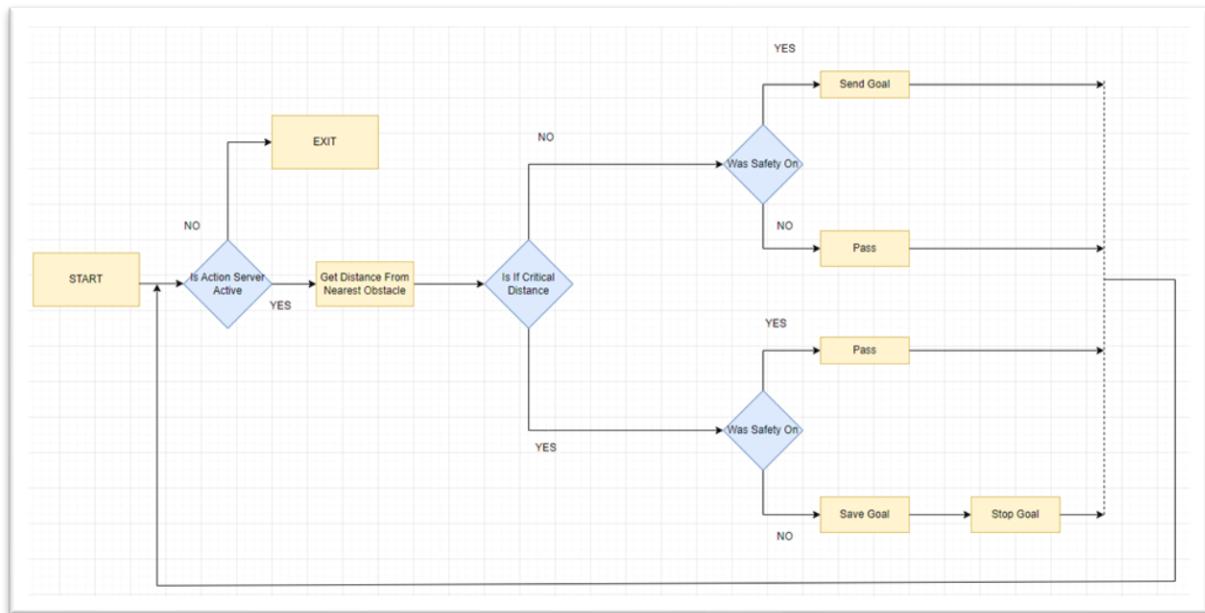
Funkcja zostaje ponownie wywołana w pętli, aż do momentu wyłączenia węzła z sieci ROS2. Aby aktywować węzeł, należy skorzystać z komendy: *ros2 run* lub uruchomić skrypt bezpośrednio poprzez interpreter *Pythona*.



Rys. 48 Schemat ideowy działania systemu bezpieczeństwa obiegowanego przez węzeł *Safety_Node*

Safety_Function

Funkcja sprawdza czy jest aktywny serwer akcji. Jeżeli jest, pobierana zostaje wartość odległości do najbliższego punktu i testowany zostaje warunek załączenia systemu bezpieczeństwa. Po wybraniu odpowiedniego warunku, procedura zostaje wznowiona.



Rys. 49 Schemat ideowy przedstawiający działanie funkcji Safety_Function

Lokalizacja skryptu

Folder środowiska zawierający pliki definiujące strukturę robota zostanie oddzielony od folderu zawierającego skrypty systemu bezpieczeństwa.

Działanie podjęte w celu uniknięcia problemów przy ewentualnej pomyłce wprowadzającej błędy w główną strukturę modelu robota.

Lokalizacja modelu: my_bot.

Lokalizacja systemu bezpieczeństwa: my_custom_package.

7. Implementacja rozwiązania

7.1 Przygotowanie systemu komputera

System operacyjny komputera rozwojowego *Ubuntu*, specyfikacja podana w tabeli.

OS	GPU	CPU	RAM	SSD	PCI
<i>Ubuntu</i> 22.04.3 LTS	GeForce GT 740	i5-11400F	PATRIOT Viper 4 Blackout 16GB TUO	970 EVO Plus 500Gb	Gigabyte H510M K V2

Tab. 2 Parametry komputera rozwojowego

Po zainstalowaniu systemu operacyjnego na komputer, za pomocą instrukcji [46], należy zainstalować system *Ubuntu*. Instrukcja zawarta instalacji *Docker* zawarta jest na stronie internetowej [38].

Po zainstalowanym narzędziu *Docker*, pobrano *Dockerfile* zawierający wszystkie potrzebne zależności konieczne do uruchomienia *ROS2*. *Dockerfile* należy do twórcy *athackst* [47].

Plik *Dockerfile* składa się z następujących części:

Konfiguracja podstawowych parametrów systemowych (z reguły realizowana przy instalacji systemu) takich jak strefa czasowa, język. Instalacja podstawowych narzędzi w pracy z terminaliem. Instalacja oprogramowania *ROS2*:

```
#####
FROM ubuntu:22.04 AS base

ENV DEBIAN_FRONTEND=noninteractive

# Install language
RUN apt-get update && apt-get install \
    locales \
    && locale-gen en_US.UTF-8 \
    && update-locale LC_ALL=en_US.UTF-8 \
    && rm -rf /var/lib/apt/lists/*
ENV LANG en_US.UTF-8
```

Rys. 50 konfiguracja języka

```
# Install ROS2
RUN sudo add-apt-repository universe
&& curl -sSL https://raw.githubusercontent.com/athackst/docker-ros2/main/install.sh | bash
&& echo "deb [arch=$(dpkg --print-architecture)] http://packages.ros.org/ros2/ubuntu $(lsb_release -cs) main" | sudo tee /etc/apt/sources.list.d/ros2.list && apt-get update && apt-get install -y ros-humble-ros-base \
    python3-argcomplete \
    && rm -rf /var/lib/apt/lists/*

ENV ROS_DISTRO=humble
ENVAMENT_PREFIX_PATH=/opt/ros/humble
ENV COLCON_PREFIX_PATH=/opt/ros/humble
ENV LD_LIBRARY_PATH=/opt/ros/humble
```

Rys. 51 instalacja ROS2

Instalacja zależności potrzebnych do uruchomienia *ROS2* oraz oprogramowania *ROS2*. Ustawienie zasad dotyczących uprawnień do plików pomiędzy użytkownikami systemu operacyjnego.

```

ENV DEBIAN_FRONTEND=noninteractive
RUN apt-get update && apt-get install \
    bash-completion \
    build-essential \
    cmake \
    gdb \
    git \
    openssh-client \
    python3-argcomplete \
    python3-pip \
    ros-dev-tools \
    ros-humble-ament-* \

```

Rys. 52 Instalacja zależności systemowych

```

# Create a non-root user
RUN groupadd --gid $USER_GID $USERNAME \
    && useradd -s /bin/bash --uid $USER_ID \
    # Add sudo support for the non-root user
    && apt-get update \
    && apt-get install -y sudo \
    && echo $USERNAME ALL=(root) NOPASSWD:ALL > /etc/sudoers.d/$USERNAME \
    && chmod 0440 /etc/sudoers.d/$USERNAME \
    && rm -rf /var/lib/apt/lists/*

# Set up autocompletion for user
RUN apt-get update && apt-get install \
    && echo "if [ -f /opt/ros/$ROS_DISTRO/share/buil...

```

Rys. 53 Ustawienia uprawnień do plików

Finalnie, pobranie pakietu symulacyjnego Gazebo wraz z wszystkimi sterownikami obsługującymi komunikację z podzespołami komputera. Na końcu pliku *Dockerfile* znajduje się linia `ENTRYPOINT`. Definiuje jakie komendy mają zostać wywołane w terminalu przy inicjalizacji kontenera.

```

# Env vars for the nvidia-container-runtime
ENV NVIDIA_VISIBLE_DEVICES all
ENV NVIDIA_DRIVER_CAPABILITIES graphics,utility
ENV QT_X11_NO_MITSHM 1

#####
# Set up ENTRYPOINT
#####
COPY entrypoint.sh /entrypoint.sh
COPY bashrc /root/.bashrc

ENTRYPOINT ["/bin/bash", "/entrypoint.sh"]

CMD ["tail", "-f", "/dev/null"]

```

Rys. 54 Sterowniki Gazebo

```

> dev > Desktop > my_project > Dockerfile
# Full+Gazebo+image
#####
FROM full AS gazebo

ENV DEBIAN_FRONTEND=noninteractive
# Install gazebo
RUN wget https://packages.osrfoundation.org/gazebo/ubuntu-stable/ \
    && echo "deb [arch=$(dpkg --print-arch)] https://packages.osrfoundation.org/gazebo/ubuntu-stable/ $(lsb_release -cs) main" | tee /etc/apt/sources.list.d/gazebo.list \
    && apt-get update && apt-get install -y \
    ros-humble-gazebo* \
    && rm -rf /var/lib/apt/lists/*
ENV DEBIAN_FRONTEND=

#####
# Full+Gazebo+Nvidia image

```

Rys. 55 Instalacja Gazebo

Do *Dockerfile*, należy dodać narzędzie nano, służące do otwierania i edytowania plików programów. Dodatkowo, na końcu dopisano komendę `CMD [„tail”, „-f”, „/dev/null”]`, która wyłączeniu kontenera po jego starcie.

Plik *Dockerfile* należy zbudować, wpisując w terminal komendę:

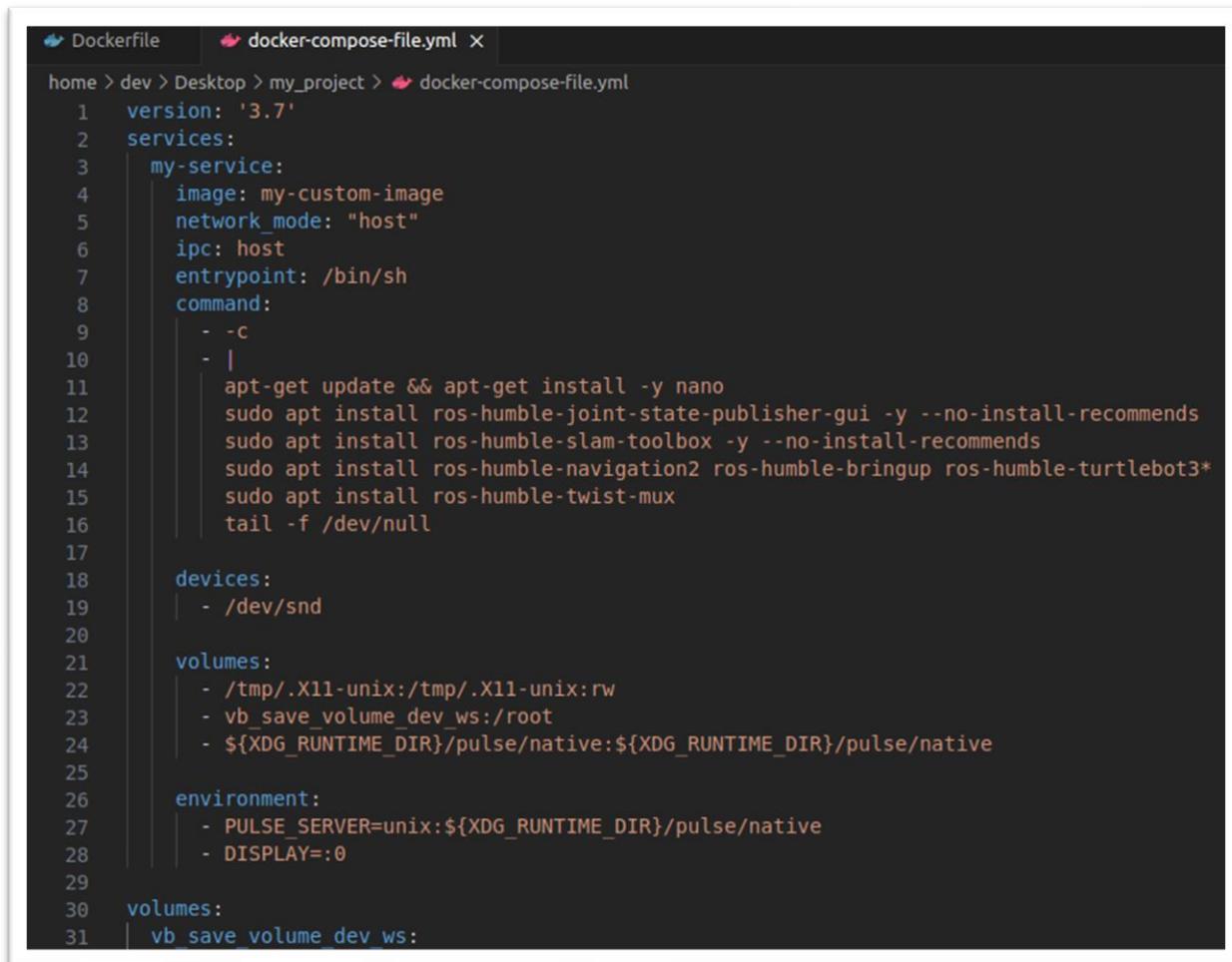
`docker build -t my-custom-image`

System zacznie tworzyć obraz kontenera, który po zbudowaniu będzie dostępny poprzez narzędzia programu *Docker*.

Zbudowany obraz należy przekształcić w kontener. Bezpośrednio na kontenerze będą wprowadzane zmiany, dlatego przekształcanie obrazu za każdym razem jest niepraktyczne, gdyż wymaga każdorazowej ponownej instalacji dodatkowych plików. Rozwiązanie polega na inicjalizacji kontenera za pomocą narzędzia *Docker-compose*.

Docker-compose

Plik narzędzia zakończony jest rozszerzeniem .yaml. Należy zdefiniować nazwę obrazu z którego prowadzony jest rozruch kontenera image: my-custom-image. Dwa kolejne parametry dotyczą ustawienia miejsca uruchomienia graficznych interfejsów aplikacji włączanych w kontenerze, które umiejscowiono na „host” czyli bezpośrednio na maszynie rozwojowej. Możliwe jest zdefiniowanie *entrypoint*, jeżeli nie było zdefiniowane w *Dockerfile*. Parametr *command*: określa jakie komendy mają zostać dodatkowo wpisane w terminal. Są to linie instalujące takie biblioteki ROS2 jak nawigacje, lokalizacje, narzędzia do publikacji / subskrybcji *Topic* oraz sterowania. W *devices*: znajdują się informacje wysyłane z klawiatury do symulacji *Gazebo* włączanej z kontenera.



```
home > dev > Desktop > my_project > docker-compose-file.yml X
Dockerfile docker-compose-file.yml

1 version: '3.7'
2 services:
3   my-service:
4     image: my-custom-image
5     network_mode: "host"
6     ipc: host
7     entrypoint: /bin/sh
8     command:
9       - -c
10      - |
11        apt-get update && apt-get install -y nano
12        sudo apt install ros-humble-joint-state-publisher-gui -y --no-install-recommends
13        sudo apt install ros-humble-slam-toolbox -y --no-install-recommends
14        sudo apt install ros-humble-navigation2 ros-humble-bringup ros-humble-turtlebot3*
15        sudo apt install ros-humble-twist-mux
16        tail -f /dev/null
17
18     devices:
19       - /dev/snd
20
21     volumes:
22       - /tmp/.X11-unix:/tmp/.X11-unix:rw
23       - vb_save_volume_dev_ws:/root
24       - ${XDG_RUNTIME_DIR}/pulse/native:${XDG_RUNTIME_DIR}/pulse/native
25
26     environment:
27       - PULSE_SERVER=unix:${XDG_RUNTIME_DIR}/pulse/native
28       - DISPLAY=:0
29
30   volumes:
31     vb_save_volume_dev_ws:
```

Rys. 56 Plik Docker-compose projektu

Ważną częścią *Docker-compose* jest parametr *volume*. Składa się z dwóch części, elementów mapowania (kopiujących na bieżąco pliki z maszyny „host” na kontener np. /tmp/.X11-unix:/tmp/.X11-unix:r) oraz elementów łączenia ang. *bind*, które określają przestrzeń zapisu (w parametrze *volumes:* vb_save_volume_dev_ws) danych wprowadzanych w folderze (:/root) lub pliku kontenera.

Włączenie kontenera

Wykonanie wszystkich koniecznych opisanych kroków pozwala na aktywację kontenera zawierającego wszystkie konieczne pliki, który w przypadku awarii może być zrestartowany lub zbudowany ponownie.

```
Dev@DevOps:~/Desktop$ cd my_project/
Dev@DevOps:~/Desktop/my_project$ ls
bashrc config docker-compose-file.yml Dockerfile entrypoint.sh
Dev@DevOps:~/Desktop/my_project$ docker-compose -f docker-compose-file.yml up -d
```

Rys. 57 Inicjacja kontenera

Rozpoczynając pracę nad projektem, należy wpisać w terminalu w miejscu ~/Desktop/my_project komendę (Rys. 58). Po wykonaniu komendy, terminal użytkownika podpięty jest pod system kontenera.

```
✓ Container my_project-my-service-1 Started
Dev@DevOps:~/Desktop/my_project$ docker exec -it my_project-my-service-1 /bin/bash
root@DevOps:/# 
```

Rys. 58 Podłączenie kontenera pod terminal

7.2 Model robota i system bezpieczeństwa

7.2.1. Zbudowanie środowiska

Pliki składające się na model robota, oparte są na podstawie repozytorium joshnewans/my_bot, które można znaleźć na stronie: [48]. Należy stworzyć nowy folder w systemie plików OS kontenera:

```
src
root@DevOps:~/dev_ws2# cd src/
root@DevOps:~/dev_ws2/src# 
```

Rys. 59 Środowisko robota w OS

Następnie pobrać folder twórcy ze strony *GitHub*, używając komendy: git clone, uprzednio konfigurując klucz bezpieczeństwa zgodnie z instrukcją [49].

```
root@DevOps:~/dev_ws2/src# git clone https://github.com/joshnewans/my_bot.git
Cloning into 'my_bot'...
remote: Enumerating objects: 16, done.
remote: Counting objects: 100% (4/4), done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 16 (delta 1), reused 1 (delta 1), pack-reused 12
```

Rys. 60 Pobieranie kodu z GitHub

Ostatnim ważnym etapem pracy z plikami środowiska jest zbudowanie, czyli skompilowanie ze źródła systemu robota komendą: colcon build –symlink-install:

```
oot@DevOps:~/dev_ws2# colcon build --symlink-install
tarting >>> my_bot
inshed <<< my_bot [0.89s]

ummary: 1 package finished [1.39s]
oot@DevOps:~/dev_ws2# █
```

Rys. 61 Kompilacja plików

Pliki URDF modelu

Struktura głównego *URDF* została opisana w rozdziale 6.4.2. Nie istniała potrzeba zmiany parametrów poszczególnych elementów w stosunku do modelu gotowego robota, ponieważ główne zagadnienie dotyczy systemu bezpieczeństwa, którego funkcjonowanie nie zależy od wymiarów robota ani jego dynamiki wynikającej z przypisanych fizycznych właściwości.

Base_footprint

```
<joint name="base_footprint_joint" type="fixed">
    <parent link="base_link"/>
    <child link="base_footprint"/>
    <origin xyz="0 0 0" rpy="0 0 0"/>
</joint>

<link name="base_footprint">
</link>
```

Rys. 62 Konfiguracja Link dla struktury lokalizacyjnej

Wyżej przedstawione zdjęcie opisuje konfiguracje dodatkowego *Link* wykorzystywanego przez bibliotekę lokalizacyjną.

Podwozie i koło samonastawne

```
</geometry>
  <material name="white"/>
</visual>
<collision>
  <origin xyz="0.15 0 0.075"/>
  <geometry>
    <box size="0.3 0.3 0.15"/>
  </geometry>
  <material name="white"/>
</collision>
<xacro:inertial_box mass="0.5" x="0.3">
  <origin xyz="0.15 0 0.075" rpy="0 0 0"/>
</xacro:inertial_box>
</link>

<gazebo reference="chassis">
  <material>Gazebo/White</material>
</gazebo>
```

Rys. 63 Część parametrów podwozia

```
</visual>
<collision>
  <geometry>
    <sphere radius="0.05"/>
  </geometry>
  <material name="black"/>
</collision>
<xacro:inertial_sphere mass="0.1">
  <origin xyz="0 0 0" rpy="0 0 0"/>
</xacro:inertial_sphere>
</link>

<gazebo reference="caster_wheel">
  <material>Gazebo/Black</material>
  <mu1 value="0.001"/>
  <mu2 value="0.001"/>
</gazebo>
</robot>
```

Rys. 64 Parametrów koła samonastawnego

W ostatnich linijkach kodu widoczne są ustawienia referencji dotyczących dodatkowych właściwości obiektów. Materiał podwozia zaprogramowany jest na kolor biały, koła na czarny. Dodatkowo między parametrami `<mu1/>` oraz `<mu2/>` ustalone współczynniki tarcia, przypisane do koła samonastawnego w symulacji Gazebo.

Prawe koło napędowe

```
<!-- RIGHT WHEEL LINK -->

<joint name="right_wheel_joint" type="continuous">
  <parent link="base_link"/>
  <child link="right_wheel"/>
  <origin xyz="0 -0.175 0" rpy="-1.5708 0 0"/> <!-- Use the numerical values -->
  <axis xyz="0 0 -1"/>
</joint>

<link name="right_wheel">
  <visual>
    <geometry>
      <cylinder radius="0.05" length="0.04"/>
    </geometry>
    <material name="blue"/>
  </visual>
  <collision>
    <geometry>
      <cylinder radius="0.05" length="0.04"/>
    </geometry>
    <material name="blue"/>
  </collision>
</link>
```

Rys. 65 Przegub i Link prawego koła napędowego

Lewe koło napędowe

```
<joint name="left_wheel_joint" type="continuous">
  <parent link="base_link"/>
  <child link="left_wheel"/>
  <origin xyz="0 0 0.175" rpy="-1.5708 0 0"/> <!-- Use the numerical value of pi -->
  <axis xyz="0 0 1"/>
</joint>

<link name="left_wheel">
  <visual>
    <geometry>
      <cylinder radius="0.05" length="0.04"/>
    </geometry>
    <material name="blue"/>
  </visual>
  <collision>
    <geometry>
      <cylinder radius="0.05" length="0.04"/>
    </geometry>
    <material name="blue"/>
  </collision>
  <xacro:inertial_cylinder mass="0.1" length="0.04" radius="0.05">
    <origin xyz="0 0 0" rpy="0 0 0"/>
  </xacro:inertial_cylinder>
```

Rys. 66 Przegub (joint), Link oraz parametry bezwładności lewego koła

Materiał kół w symulacji został ustawiony na kolor biały.

Bezwładność elementów

```
<xacro:macro name="inertial_cylinder" params="mass length radius *origin">
  <inertial>
    <xacro:insert_block name="origin"/>
    <mass value="${mass} />
    <inertia ixx="${(1/12) * mass * (3*radius*radius + length*length)}" ixy="0.0" ixz="0.0"
          iyy="${(1/12) * mass * (3*radius*radius + length*length)}" iyz="0.0"
          izz="${(1/2) * mass * (radius*radius)}" />
  </inertial>
</xacro:macro>
```

Rys. 67 Definicja bryły inertial_cylinder z pliku inertial_macros.xacro

Nazwa pliku: inertial_macros.xacro. Definiuje bryły, dla których określone są momenty bezwładności. Parametry dla obliczeń właściwości fizycznych momentu przekazywane są poprzez odwołanie się do danej bryły przy tworzeniu elementu (można zaobserwować w przypadku kół i podwozia: <xacro:inertial_cylinder mass="0.1" length="0,04" radius="0.05">).

LIDAR

Opisany w podrozdziale dotyczącym przedstawieniu proponowanego rozwiązania projektowego.

```
<joint name="laser_joint" type="fixed">
  <parent link="chassis"/>
  <child link="laser_frame"/>
  <origin xyz="0.1 0 0.175" rpy="0 0 0"/>
</joint>

<link name="laser_frame">
  <visual>
    <geometry>
      <cylinder radius="0.05" length="0.04"/>
    </geometry>
    <material name="red"/>
  </visual>
  <collision>
    <geometry>
      <cylinder radius="0.05" length="0.04"/>
    </geometry>
    <material name="blue"/>
  </collision>
  <xacro:inertial_cylinder mass="0.1" length="0.04" radius="0.05">
    <origin xyz="0 0 0" rpy="0 0 0"/>
  </xacro:inertial_cylinder>
```

Rys. 68 Przegub i Link LIDAR

Pliki przekazujące sygnały sterujące do Gazebo

Kluczowe informacje to odległości między osiami kół oraz ich promienie, przeguby ukł. wspł kół oraz ograniczenia prędkości jakie robot może osiągnąć.

```
<plugin name="diff_drive" filename="libgazebo_ros_diff_drive.>
  <!-- Wheel Information -->
  <left_joint>left_wheel_joint</left_joint>
  <right_joint>right_wheel_joint</right_joint>
  <wheel_separation>0.35</wheel_separation>
  <wheel_diameter>0.1</wheel_diameter>

  <!-- Limits -->
  <max_wheel_torque>200</max_wheel_torque>
  <max_wheel_acceleration>10.0</max_wheel_acceleration>

  <command_topic>cmd_vel43e</command_topic>

  <!-- Output -->
  <odometry_frame>odom</odometry_frame>
  <robot_base_frame>base_link</robot_base_frame>
  <publish_odom>true</publish_odom>
  <publish_odom_tf>true</publish_odom_tf>
```

Rys. 69 Plik przekazujący sygnały sterujące do Gazebo

Po utworzeniu wszystkich koniecznych plików, należy przebudować robota stosując narzędzie colcon przedstawione na początku podrozdziału. Narzędzie będzie wykorzystywane przy wprowadzaniu każdej zmiany w konstrukcji robota.

7.2.2 Test modelu, wizualizacja i symulacja

Skrypt rsp

Gotowy skrypt pobrany z repozytorium twórcy *Articulated Robotics* [40]. Tworzy ścieżkę do pliku *URDF*, następnie definiuje parametry rozruchowe dla węzła oparte o strukturę w *URDF*, które są przekazywane do pliku wykonującego integracje platformy robota *robot_state_publisher*. Zwraca węzeł odpowiedzialny za inicjalizację projektu modelu robota.

```
import xacro

def generate_launch_description():

    # Check if we're told to use sim time
    use_sim_time = LaunchConfiguration('use_sim_time')

    # Process the URDF file
    pkg_path = os.path.join(get_package_share_directory('my_bot'))
    xacro_file = os.path.join(pkg_path,'description','robot.urdf.xacro')
    robot_description_config = xacro.process_file(xacro_file)

    # Create a robot_state_publisher node
    params = {'robot_description': robot_description_config.toxml(), 'use_si
    node_robot_state_publisher = Node(
        package='robot_state_publisher',
        executable='robot_state_publisher',
        output='screen',
        parameters=[params]
    )

    # Launch!
    return LaunchDescription([
        DeclareLaunchArgument(
            'use_sim_time',
            default_value='false',
            description='Use sim time if true'),
        node_robot_state_publisher
    ])
```

Rys. 70 Skrypt zwracający węzeł generujący platformę robota

Skrypt launch

Gotowy skrypt pobrany z repozytorium twórcy *Articulated Robotics* [40]. Zawiera definicje trzech obiektów: *rsp* (reprezentuje plik definiujący główny węzeł platformy robota), *Gazebo* (włącza symulację *Gazebo* korzystając z plików zawartych w bibliotece *gazebo_ros* pobranej podczas budowy kontenera) oraz *spawn_entity* (obiekt łączący pliki głównego węzła robota z symulacją *Gazebo*, inicjalizuje robota w symulacji).

```
rsp = IncludeLaunchDescription(  
    PythonLaunchDescriptionSource([os.path.join(  
        get_package_share_directory(package_name), 'launch', 'rsp.launch.py'  
    )]), launch_arguments={'use_sim_time': 'true'}.items()  
  
# Include the Gazebo launch file, provided by the gazebo_ros package  
gazebo = IncludeLaunchDescription(  
    PythonLaunchDescriptionSource([os.path.join(  
        get_package_share_directory('gazebo_ros'), 'launch', 'gazebo.launch.py')])  
  
# Run the spawner node from the gazebo_ros package. The entity name doesn't really matter  
spawn_entity = Node(package='gazebo_ros', executable='spawn_entity.py',  
    arguments=['-topic', 'robot_description',  
              '-entity', 'my_bot'],  
    output='screen')
```

Rys. 71 Główny plik *launch*

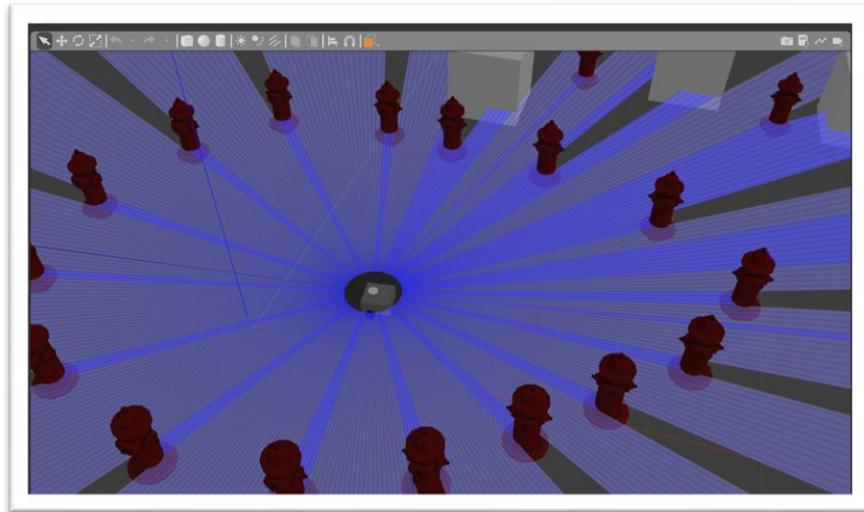
Test poprawności modelu

Pierwszą czynnością jest uruchomienie skryptu *launch*: *launch_sim.launch.py*

```
oot@DevOps:~# source dev_ws/install/setup.bash  
oot@DevOps:# ros2 launch my_bot launch_sim.launch.py world:=./src/my_bot/worlds/o  
stacles.world  
[INFO] [launch]: All log files can be found below /root/.ros/log/2024-01-23-18-06-2  
-804490-DevOps-797  
[INFO] [launch]: Default logging verbosity is set to INFO
```

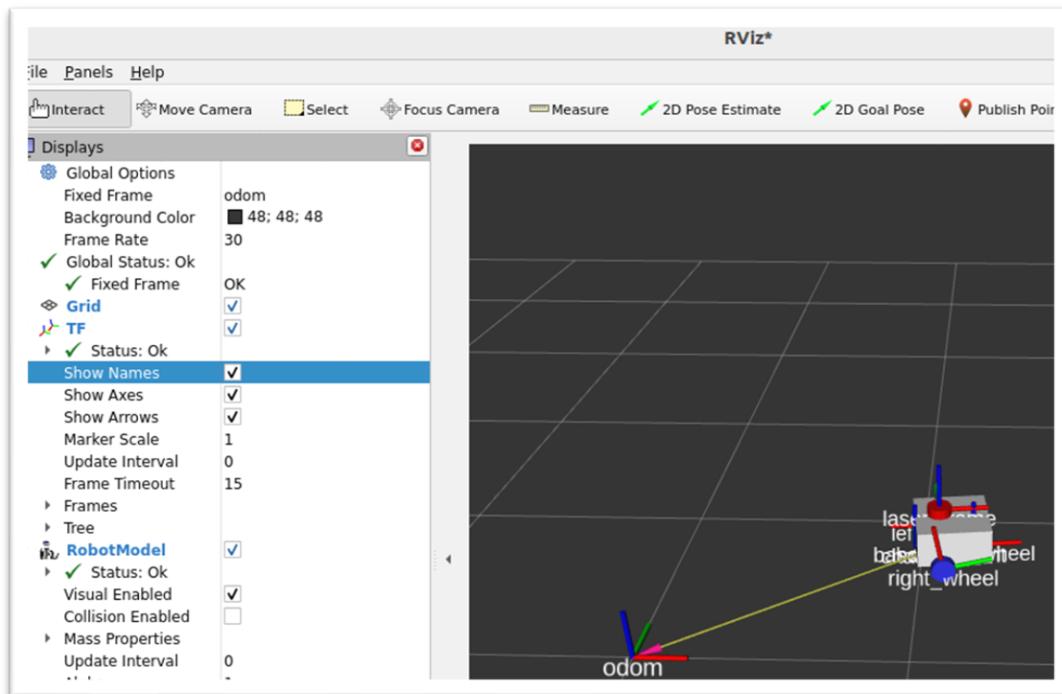
Rys. 72 Uruchomienie platformy robota

Parametr *world* wskazuję na wcześniej stworzone środowisko symulatora *Gazebo* zawierające różne przeszkody. Powinna wyświetlić się symulacja robota. Niebieskie linie oznaczają zakres widoczności skanera *LIDAR*.



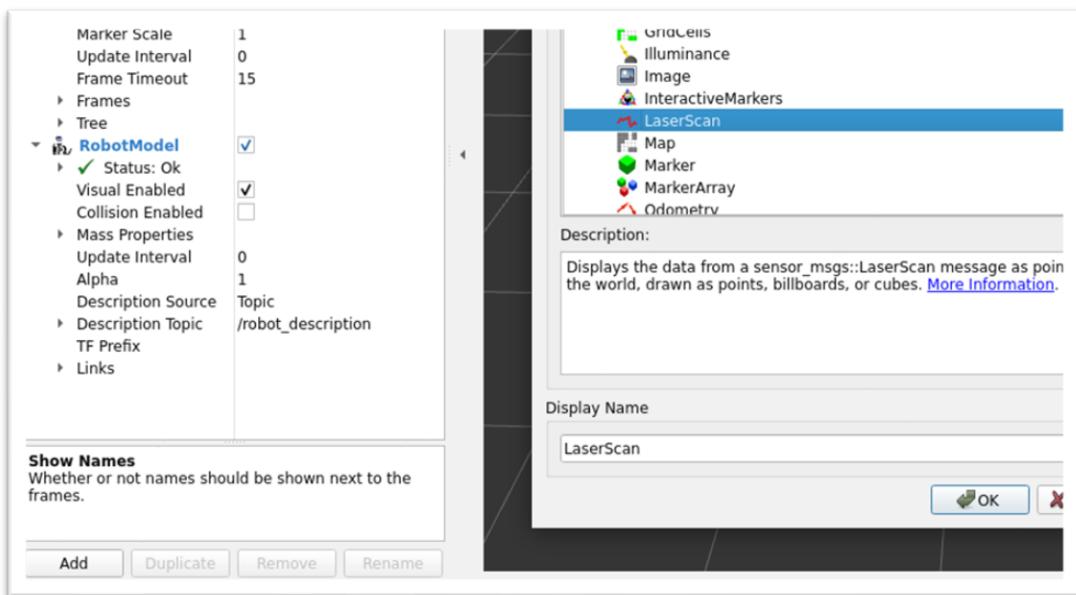
Rys. 73 Środowisko w symulacji Gazebo z widocznymi skanami skanera

Następnie należy włączyć program wizualizujący robota, aby sprawdzić poprawność ustawienia ukł. wspł oraz potwierdzić poprawność skanów generowanych przez *LIDAR* w symulacji. Uruchomienie wizualizatora, czyli programu *RViz2*, polega na włączeniu kolejnego terminala w kontenerze i wpisaniu linii kodu: *rviz2*. Pojawi się okno programu. Należy zaznaczyć opcję *Show Names*, aby nazwy ukł. wspł ukazały się.



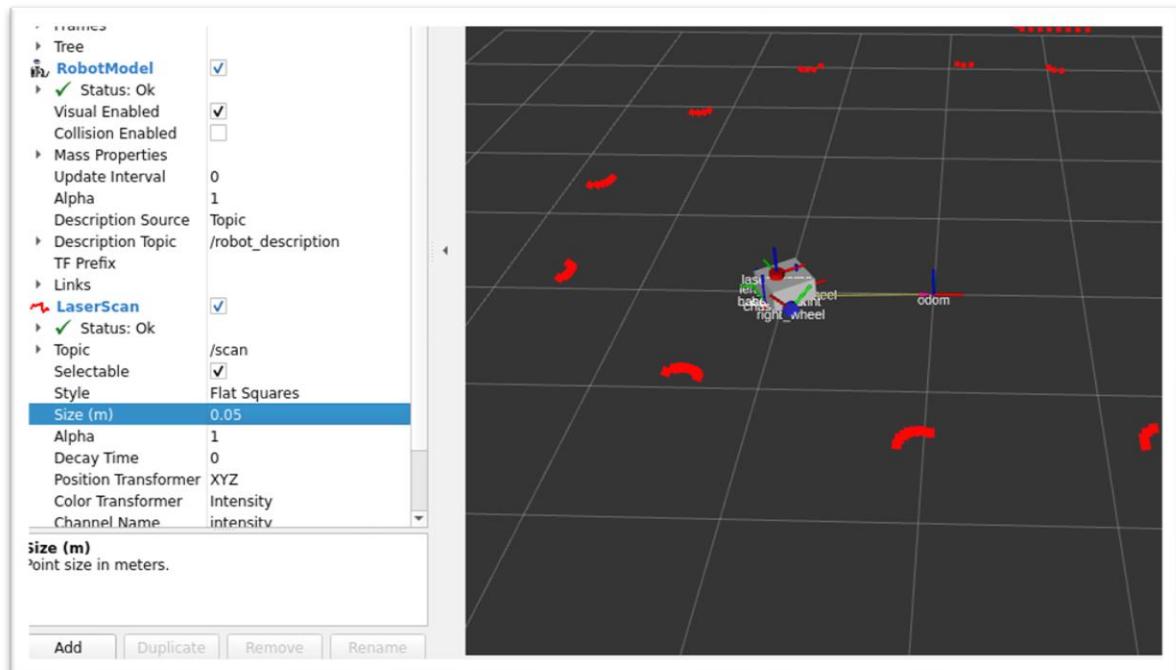
Rys. 74 RViz2 Ustawienie widoczności nazw ukł. wspł

Po sprawdzeniu poprawności położenia ukł. wspł, należy sprawdzić wygląd chmury punktów generowanej przez skaner. Klikając w przycisk „Add” należy dodać LaserScan. W drzewku po lewej stronie pojawi się folder LaserScan.



Rys. 75 Wizualizacja skanera (1)

Należy wybrać *Topic*: /scan oraz zwiększyć wymiar punktów, aby były lepiej widoczne np. na 0.05. Powinny pojawić się skany przechwycone z LIDAR. Procedura testu modelu i skanera platformy robota, przebiegła pomyślnie.



Rys. 76 Wizualizacja skanera (2)

Sterowanie - Wizualizacja w programie *RViz2* i symulacja w *Gazebo*

Realizacja zagadnienia sterowania oparta jest o narzędziu *teleop_twist_keyboard*. Po wpisaniu i zatwierdzeniu komendy, pojawi się okno, które informuje jakie przyciski na klawiaturze odpowiedzialne są za ruch robota. Narzędzie publikuje sygnały ruchu bezpośrednio na *Topic* odpowiedzialny za przekazywanie informacji o ruchu na platformę robota: */cmd_vel*.

```
oot@DevOps:~/dev_ws# ros2 run teleop_twist_keyboard teleop_twist_keyboard
```

Rys. 77 Uruchomienie narzędzia *teleop_twist_keyboard*

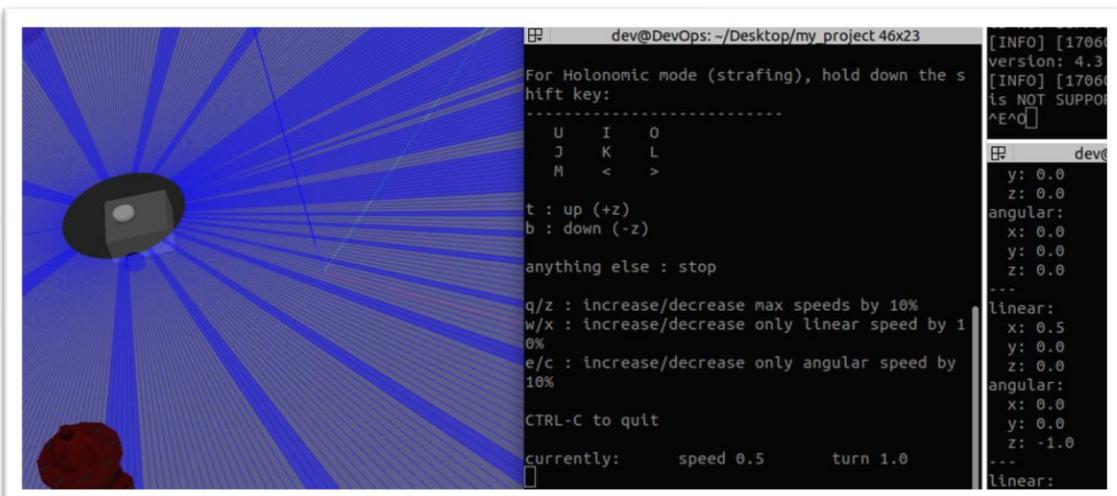
```
oot@DevOps:~/dev_ws# ros2 topic list
clicked_point
clock
cmd_vel
goal_pose
initialpose
joint_states
odom
parameter_events
performance_metrics
robot_description
rosout
scan
tf
tf_static
oot@DevOps:~/dev_ws#
```

Rys. 78 Topic list

```
oot@DevOps:~/dev_ws# ros2 topic echo /cmd_vel
linear:
  x: 0.0
  y: 0.0
  z: 0.0
angular:
  x: 0.0
  y: 0.0
  z: 0.0
```

Rys. 79 Informacje zwrotne z Topic */cmd_vel*

Komenda *ros2 topic list* wyświetla aktywne *Topic*. Aby włączyć podgląd *Topic* w terminalu, należy wpisać: *ros2 topic echo /cmd_vel*



Rys. 80 Sposób korzystania z narzędzia *teleop_twist_keyboard*

Należy przycisnąć lewym przyciskiem myszy na okno z aktywnym narzędziem teleop_twist_keyboard oraz wysyłać kolejne komendy ruchu zgodnie z instrukcją. Zmiana pozycji będzie widoczna w terminalu podsłuchującym *Topic /cmd_vel* (z: -1.0) oraz wizualnie, poprzez obserwacje ruchów robota na symulacji.

7.2.3 Biblioteki Lokalizacji i nawigacji

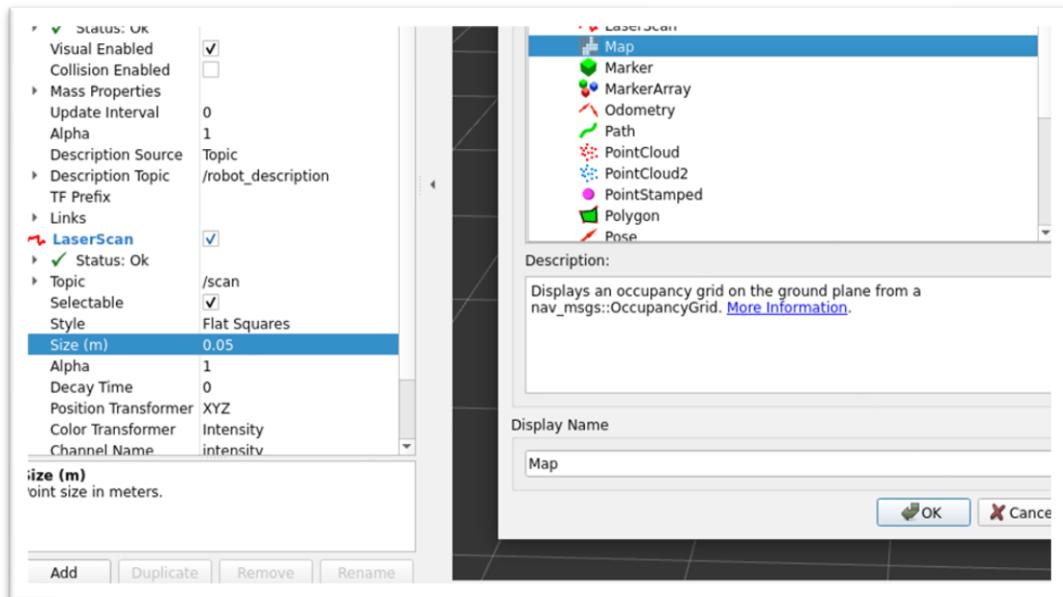
Lokalizacja

Instalacja biblioteki lokalizacji została wykonana podczas inicjalizacji kontenera poprzez narzędzie *Docker-compose*. Włączenie funkcji polega na wpisaniu w terminal komendy:

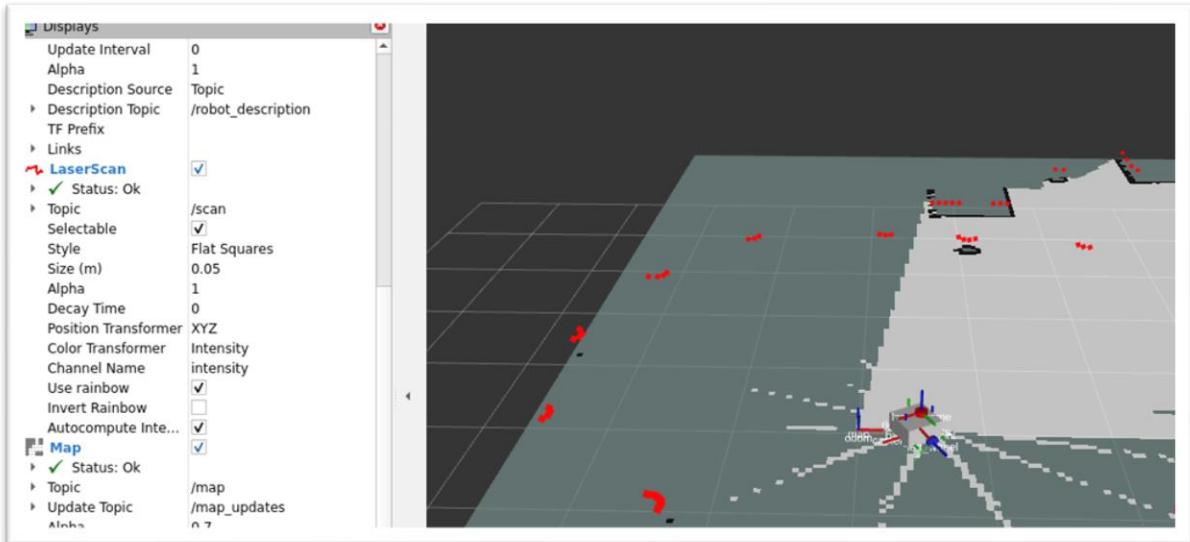
```
root@DevOps:~/dev_ws# ros2 launch slam_toolbox online_async_launch.py slam_params_file:=./src/my_bot/config/mapper_params_online_async.yaml use_sim_time:=true
```

Rys. 81 Włączenie lokalizacji

W *RViz2* należy w drzewie przycisnąć przycisk „Add” i wybrać opcję „map”. W drzewku po lewej stronie wybrać odpowiedni *Topic*, na którym publikowana jest mapa przez narzędzie *SLAM*, jest to */map*. Powinna wyświetlić się mapa.

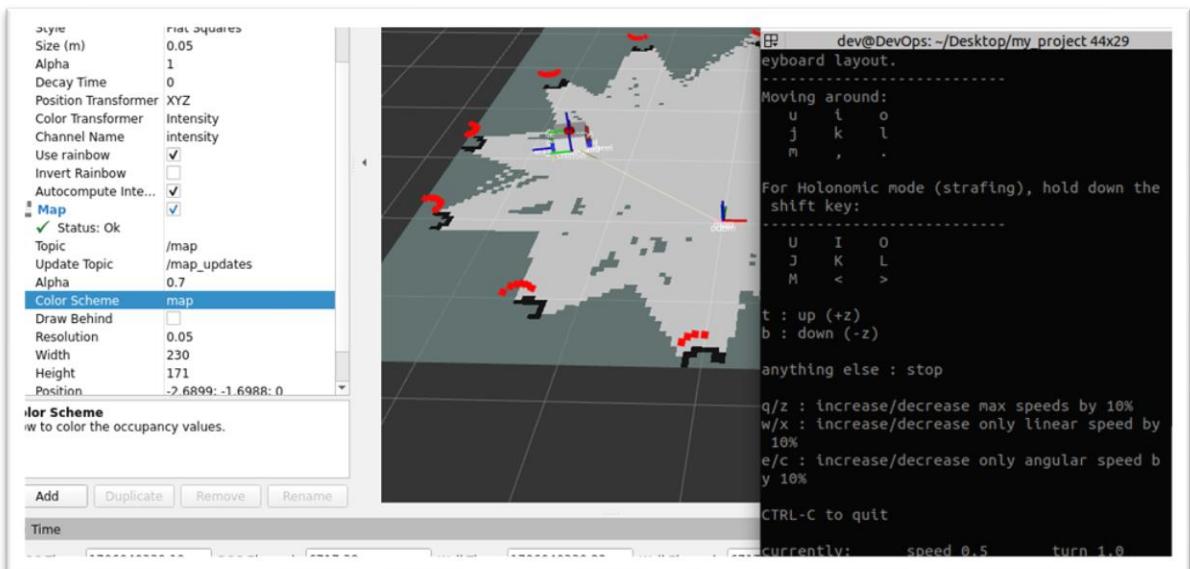


Rys. 82 Dodanie mapy



Rys. 83 Wygląd mapy

Podobnie jak w przypadku testu sprawności systemu sterowania modelu, należy włączyć narzędzie *teleop_twist_keyboard*. W terminalu znajdującym się w prawym dolnym rogu widać, iż *Topic /cmd_vel* przyjmuje wiadomości zmiany pozycji. Ruch robota spowodował zaaktualizowanie się mapy, algorytm SLAM poprawnie analizuje otoczenie i tworzy mapę, co widać wnioskując po skanach nakładających się z zarysami mapy.



Rys. 84 Budowanie mapy

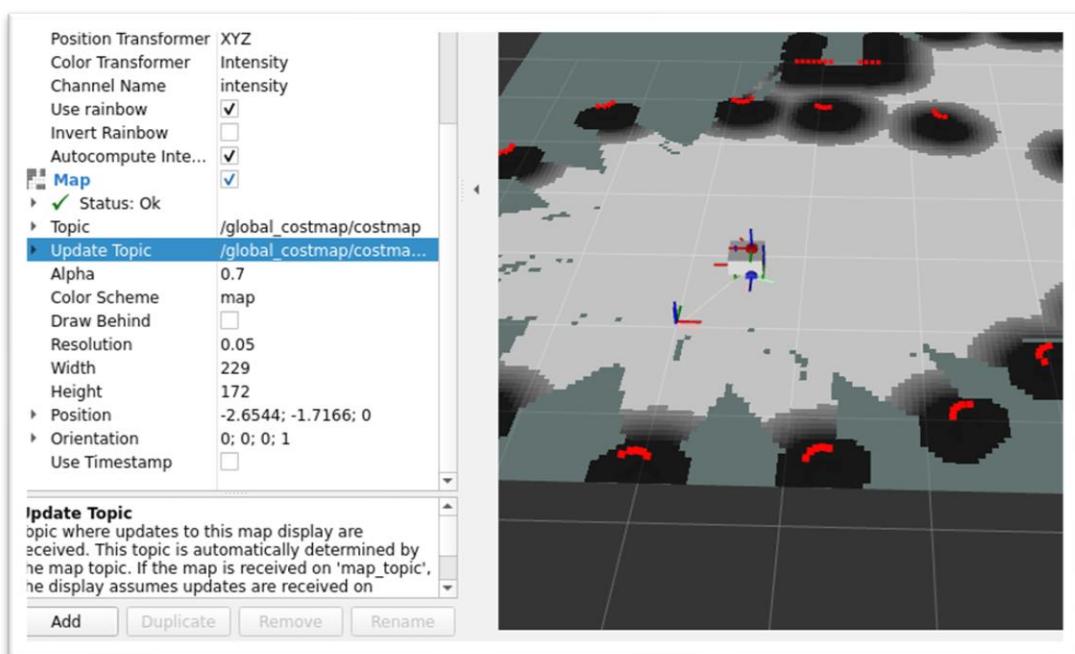
Nawigacja

Instalacja biblioteki nawigacji została wykonana podczas inicjalizacji kontenera poprzez narzędzie *Docker-compose*. Warunkiem działania nawigacji jest aktywny moduł lokalizacyjny. Włączenie funkcji polega na wpisaniu w terminal komendy:

```
root@DevOps:~/dev_ws# ros2 launch nav2_bringup navigation_launch.py use_sim_time:=true
```

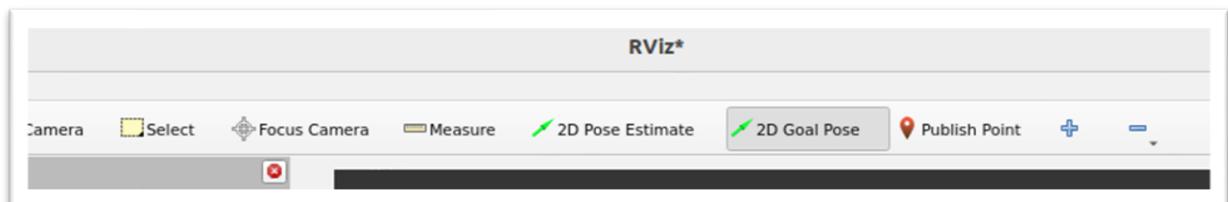
Rys. 85 Uruchomienie nawigacji

Podobnie jak w przypadku lokalizacji znajdującej się w RViz2, należy w drzewie folderów po lewej stronie przycisnąć przycisk „Add” i wybrać opcję „map”. Wybrać odpowiedni Topic, na którym publikowana jest mapa kosztów przez NAV2. Jest to /global_costmap/costmap. Powinna wyświetlić się mapa kosztów.



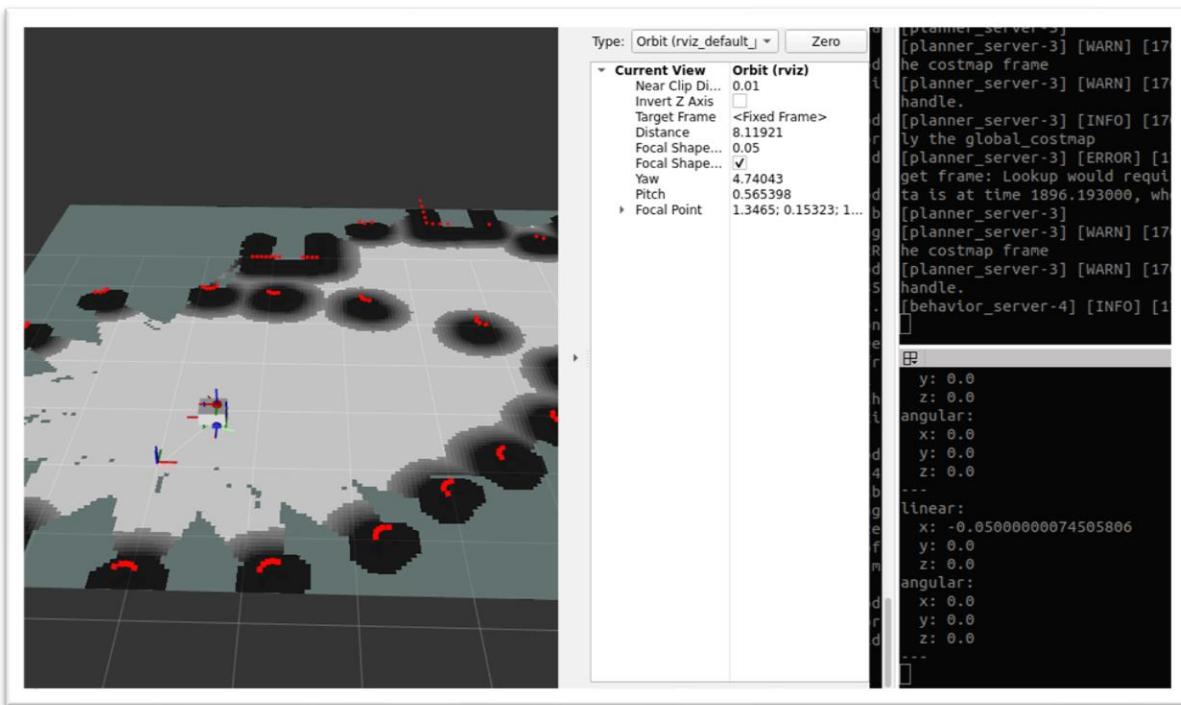
Rys. 86 Mapa kosztów w RViz2

Przetestowanie nawigacji polega na wybraniu przycisku 2D Goal Pose znajdującego się w górnej części okna. Robot powinien zacząć nawigować do wybranego elementu



Rys. 87 Przycisk 2D Goal Pose

W prawym dolnym terminalu wyświetlają się wiadomości z Topic/cmd_vel. Robot aktywnie przemieszcza się do zdefiniowanego celu, co stanowi o działaniu nawigacji.



Rys. 88 Samoczynna jazda robota – widoczna zmiana prędkość w kierunku linear: x

Struktura modelu robota, nawigacji i lokalizacji została zaimplementowana. Ostatnią częścią projektu, jest zaprogramowanie systemu bezpieczeństwa.

7.2.4 System Bezpieczeństwa

Pliki systemu bezpieczeństwa zostały zorganizowane w dwóch oddzielnych skryptach – pierwszy realizuje publikacje *Topic /closest_point*, drugi tworzy węzeł odpowiedzialny za wykrywanie warunków aktywacji systemu bezpieczeństwa. Skrypty zostały zlokalizowane w oddzielnym środowisku o nazwie *my_custom_package* zbudowanym, według wspomnianych w pracy metod. Nadawanie parametrów punktu do którego nawigacja ma zmierzać, realizowane jest wewnątrz skryptu, gdzie można modyfikować wartość wspł. jego położenia. Podobnie z parametrem zawierającym dane o odległości do najbliższego punktu, który łączyła system.

Skryptu Point_Checker_3

Schemat skryptu znajduje się: (Rys. 47)

W głównej funkcji: stworzenie obiektu, który dziedziczy z klasy węzła: *LaserRot()*. Nadanie częstotliwości pętli 10 hz. Dopóki węzeł istnieje tzn. skrypt nie zostanie przerwany, wywołaj na obiekcie metodę *transform_scan*. Jeżeli skrypt zostanie przerwany lub węzeł napotka poważny błąd, wyczyść pamięć i bezpieczne wyłącz węzeł.

```

def main(args=None):
    rclpy.init(args=args)
    lr = LaserRot()
    rate = lr.node.create_rate(10)

    while rclpy.ok():
        lr.transform_scan()
        rclpy.spin_once(lr.node)

    node.destroy_node()
    rclpy.shutdown()

```

Rys. 89 Main programu Point_Checker_3

Przy inicjalizacji obiektu, zdefiniowany jest typ *Topic*, na który będzie wysyłała informacją, nazwa węzła to „Point_node”. Zostaje zasubskrybowany *Topic* /scan, z którego dane cały czas są pobierane poprzez funkcję laser_callback. Obiekt closestP obsługuje funkcjonalność publikowania wiadomości na *Topic* /closest_point.

```

class LaserRot(object):
    def __init__(self):
        msg = String()
        msg.data = 'Hello, ROS 2!'
        self.laser = LaserScan()

        self.node = rclpy.create_node("Point_node")

        self.laserS = self.node.create_subscription(LaserScan, "/scan", self.laser_callback, 10)
        self.closestP = self.node.create_publisher(PointStamped, "/closest_point", 1)
        self.node.get_logger().info('Publishing: "%s"' % msg.data)

    def laser_callback(self, msg):
        self.laser = msg

```

Rys. 90 __init__ węzła inicjowanego jako obiekt

Parametr laser przechowuje wszystkie wartości odległości skanów pobranych dla przesunięcia skanera o laser.angle_increment w zakresie 360 stopni. Lista laser.ranges jest sprawdzana w pętli dla każdej wartości, gdzie na podstawie znajomości zmiany kąta względem położenia startowego (self.angle_min) oraz zmiany kąta (i*laser.angle_increment) obliczana jest współrzędna x oraz y współrzędnych punktu dzięki przekształceniom trygonometrycznym. Warunek sprawdzający czy dany skan jest najbliższy to if laser[i] < shortest_laser. Po sprawdzeniu wszystkich wartości z listy laser, współrzędne najbliższego punktu są publikowane.

```

laser = self.laser.ranges

shortest_laser = 10000
point = PointStamped()
for i in range(len(laser)):
    if laser[i] < shortest_laser:
        angle = self.laser.angle_min + i * self.laser.angle_increment
        x = laser[i] * cos(angle)
        shortest_laser = laser[i]
        point.point.x = x
        point.point.y = shortest_laser * sin(angle)

point.header.frame_id = 'laser_frame' # Set the frame ID

self.closestP.publish(point)

```

Rys. 91 Algorytm wyliczający najbliższy punkt względem robota

Skrypt Nav_Stop_Mechanism

Schemat skryptu znajduje się: (Rys. 48) oraz (Rys. 49)

W głównej funkcji: stworzenie obiektu, który dziedziczy z ogólnej klasy dla węzła: SafetyNode(). Nadanie częstotliwości pętli węzła 10 hz. Dopóki węzeł istnieje w pętli wykonuje się wywołanie głównej funkcji bezpieczeństwa safety_controll pod warunkiem, jeżeli został ustawiony atrybut arg_start. Zawiera on wartość True, jeżeli funkcja callback(rys43243) wykona się 300 razy, jest to zabezpieczenie przed problemami związanymi z brakiem uzyskiwania wartości położenia punktu pozyskiwanego z Topic /closest_point. Skrypt pobiera dane szybciej niż następuje ustabilizowanie połączenia. Jeżeli węzeł/skrypt zostanie wyłączono, pamięć zostaje bezpiecznie zwolniona.

```

SF = SafetyNode()
rate = SF.create_rate(10)
run_once = False
while rclpy.ok():
    if SF.callback_active == True:
        if run_once == False:
            SF.send_goal()
            run_once = True
            SF.arg_start = True
        else:
            pass

    if SF.arg_start == True:
        SF.safety_controll()
        print("safety_work")
    else:
        pass
    rclpy.spin_once(SF)

```

Rys. 92 Main Nav_Stop_mechanism

NAVIGATION_X_POSITION i NAVIGATION_Y_POSITION definiują wspł. punktu celu nawigacji. THRESHOLD_DISTANCE to dystans najbliższego punktu załączającego system bezpieczeństwa. Następuje subskrypcja *Topic closest_point* oraz definicja atrybutów w których przechowywane są dane o znajdującym się najbliżej robota punkcie.

```
NAVIGATION_X_POSITION = 3.0
NAVIGATION_Y_POSITION = 4.0
THRESHOLD_DISTANCE = 1.0

class SafetyNode(Node):

    def __init__(self):
        super().__init__('safety_node')

        # Parametry /closest_point parameters

        self.x = 0
        self.y = 0
        self.callback_active = False
        self.callback_iter = 0

        self.closest_point_subscription = self.create_subscription(
            PointStamped,
            '/closest_point',
            self.closest_point_callback,
            10
```

Rys. 93 Parametryzacja skryptu

THRESHOLD_DISTANCE zawiera parametr krytycznej odległości punktu względem robota, arg_start kontroluje pętlę funkcji bezpieczeństwa w funkcji main skryptu. Goal_handle przechowuje status nawigacji i zarządza jej funkcjonalnością, klient akcji nawiązuje kontakt z serwerem akcji ulokowanym w /navigate_to_pose. First_goal_msg zostanie zdefiniowany jako obiekt zawierający informację o pierwszym punkcie celu nawigacji.

```
self.threshold_distance = THRESHOLD_DISTANCE

self.arg_start = False

self.goal_msg = None

self.safety = False

self.goal_handle = None

self.action_client = ActionClient(self, NavigateToPose, '/navigate_to_pose')

self.first_goal_msg = None
```

Rys. 94 innne funkcji

closest_point_callback nasłuchiwał najnowszych wartości współrzędnych pozyskiwanych z *Topic /closest_point*.

```

def closest_point_callback(self, msg: PointStamped):

    # Pobranie informacji o położeniu punktu publikowanego na /closest_point
    self.x = msg.point.x
    self.y = msg.point.y

    # Wykonanie w petli dodawania do 300, opóźnienie rozruchu funkcji safety, az atrybuty funkcji
    self.callback_iter = self.callback_iter + 1
    if self.callback_iter == 300:
        self.callback_active = True

```

Rys. 95 Konstrukcja `closest_point_callback`

Funkcja `send_goal` czeka na odpowiedź serwera akcji. Ustawione są parametry celu nawigacji, zostają wysłane do serwera akcji. Jeżeli serwer przyjmie zadanie, zwrócone przez serwer parametry zostają zawarte w obiekcie `goal_result`, który zostanie wykorzystany przy włączaniu / wyłączaniu nawigacji.

```

def send_goal(self):
    """
    Funkcja odpowiedzialna za przeslanie pierwszego zlecenia jazdy do celu.
    Konieczne do prawidłowego funkcjonowania systemu safety
    """

    self.action_client.wait_for_server()
    self.first_goal_msg = NavigateToPose.Goal()
    self.first_goal_msg.pose.header.frame_id = 'map'
    self.first_goal_msg.pose.pose.position.x = NAVIGATION_X_POSITION
    self.first_goal_msg.pose.pose.position.y = NAVIGATION_Y_POSITION

    send_goal_future = self.action_client.send_goal_async(self.first_goal_msg)
    rclpy.spin_until_future_complete(self, send_goal_future)
    self.goal_handle = send_goal_future.result()

    if send_goal_future.result() is not None:
        print('Goal sent successfully!')
    else:
        print('Failed to send the goal.')

```

Rys. 96 Zapisywanie celu nawigacji

Blok `Is Node active` schematu (Rys. 48) jest zawarty w funkcji systemu bezpieczeństwa.

Kod początku funkcji `safety_controll` sprawdza, czy istnieje serwer akcji. Jeżeli nie istnieje, zostaje wyświetlony komunikat i skrypt wyłącza się.

```

def safety_controll(self):
    server_exists = self.action_client.wait_for_server(timeout_sec=0.5)

    stop_war = None

    if server_exists:
        pass
    else:
        print("NavigateToPose action server does not exist or is not available, safety cannot operate.")

```

Rys. 97 Warunek istnienia serwera akcji

Algorytm wyznaczania najbliższego punktu polega na wykorzystaniu twierdzenia pitagorasa, gdzie wynikiem jest odległość w metrach. Następnie sprawdzane jest, czy w obecnej chwili spełniony jest warunek zatrzymania (stop_war). Istnieją 4 możliwości: 1) System bezpieczeństwa nie działał i należy załączyć, ponieważ stop_war = True 2) System bezpieczeństwa nie działał i nie należy nic robić, ponieważ stop_war = False 3) System bezpieczeństwa działał i należy wyłączyć, ponieważ stop_war = False 4) System bezpieczeństwa działał i nie należy nic robić, ponieważ stop_war = True.

```

distance_to_robot = (self.x ** 2 + self.y ** 2) ** 0.5

if distance_to_robot <= self.threshold_distance:
    stop_war = True
elif distance_to_robot > self.threshold_distance:
    stop_war = False
else: ...
print(stop_war)

if self.safety == True and stop_war == True: ...
elif self.safety == False and stop_war == True: ...

elif self.safety == False and stop_war == False: ...

elif self.safety == True and stop_war == False: ...
else: ...

```

Rys. 98 Konstrukcja funkcji warunkowej

Przy warunku 3) należy wysłać poprzez klienta akcji zapisane zadanie nawigacyjne w goal_msg, poprzez funkcję send_goal_async. Dla przypadku 4) nie dzieje się nic.

```

if self.safety == True and stop_war == True:
    pass

elif self.safety == True and stop_war == False:
    print("SENT GOAL")
    time.sleep(3)
    if self.goal_msg == None:
        raise("Error when start, goal_msg is None, when it should be set! ")
    send_goal_future = self.action_client.send_goal_async(self.goal_msg)
    rclpy.spin_until_future_complete(self, send_goal_future)
    self.goal_handle = send_goal_future.result()
    if send_goal_future.result() is not None:
        print('Goal sent successfully!')
    else:
        print('Failed to send the goal.')

```

Rys. 99 Przypadki 3) oraz 4) funkcji warunkowej

W sytuacji 1) zanim sygnał nawigacji zostanie przerwany (goal_handle.cancel_goal_async()), należy zapisać parametry celu w goal_msg.pose.position.x oraz goal_msg.pose.pose.position.y. W przypadku 2) nie dzieje się nic.

```

        elif self.safety == False and stop_war == True:
            print("STOPPED_GOAL")
            self.goal_msg = NavigateToPose.Goal()
            self.goal_msg.pose.header.frame_id = 'map'
            self.goal_msg.pose.pose.position.x = NAVIGATION_X_POSITION
            self.goal_msg.pose.pose.position.y = NAVIGATION_Y_POSITION

            future = self.goal_handle.cancel_goal_async()
            rclpy.spin_until_future_complete(self, future)
            if future.result() is not None:
                print('Cancel all goals request sent successfully')
            else:
                print('Failed to send cancel all goals request')

            self.safety = True

        elif self.safety == False and stop_war == False:
            pass

        self.safety = False

```

Rys. 100 Przypadki 1) oraz 2) funkcji warunkowej

Warunkiem załączeń skryptów są aktywne moduły nawigacji i lokalizacji. Aby system działał poprawnie, należy najpierw opublikować w sieci ROS2 najbliższy punkt. Zadanie jest obsługiwane przez skrypt:

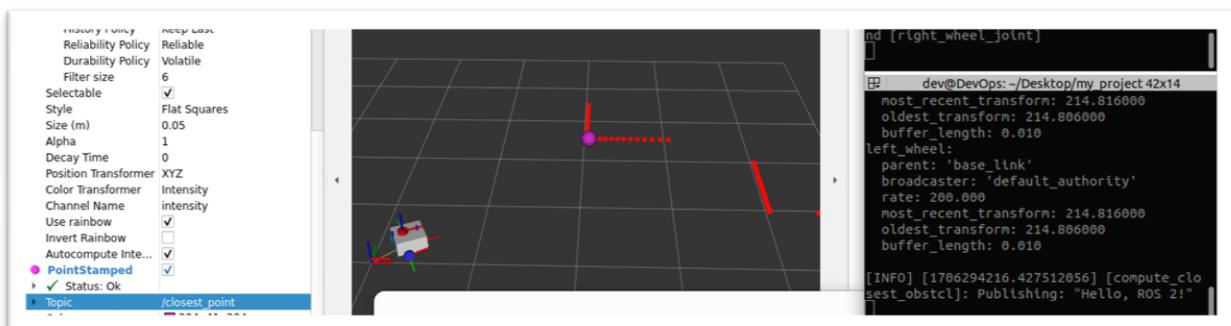
```

root@DevOps:~/dev_ws/src/my_custom_package
/scripts# python3 Point_Checker_3.py
2024-01-26 18:26:54.105 * [TCP TRANSPORT] ...

```

Rys. 101 Komenda załączająca skrypt

Znajdując się RViz2, w drzewku należy dodać Topic PointStamped i wskazać na miejsce w którym wysyłane są informacje publikowane przez skrypt:



Rys. 102 Topic /closest_point w Rviz2

Jeżeli fioletowa kropka wyświetla najbliższy punkt i terminal zwraca informacje jak na (Rys. 102) to procedura przebiegła prawidłowo. Następnie należy uruchomić system bezpieczeństwa:

```
root@DevOps:~/dev_ws/src/my_custom_package  
/scripts# python3 Nav_Stop_mechanism.py
```

Rys. 103 Uruchomienie systemu bezpieczeństwa

W terminalu w którym uruchomiono skrypt, powinny wyświetlać się komunikaty: False – jeżeli system nie powinien być aktywny czyli wszystkie przeszkody znajdują się odpowiednio daleko i True – jeżeli przeszkody są za blisko. Komunikat safety_work wskazuje na wykonanie pętli systemu bezpieczeństwa.

```
[start_costmap], max/min laser range setting (20.0 m) exceeds the capabilities of the used Lidar (12.0 m)  
[async_slam_toolbox_node-1] Registering sensor: [Custom Described Lidar]  
[1706312481.13114029] [global_costmap.global_costmap]: StaticLayer: Resizing costmap to 228 X 171 at 0.050000 m/pix  
dev@DevOps: ~/Desktop/my_project/package/python3 Nroot@DevOps:~/dev_ws/src/my_custom_package/thonroot@DevOps:~/dev_ws/src/my_custom_package/root@DevOps:~/dev_ws/src/my_custom_package/root@DevOps:~/dev_ws/src/my_custom_package/~dev_ws/src/my_custom_package/scripts# python3 Nav_Stop_mechanism.py
```

Terminal	Output
dev@DevOps: ~ 40x11	True safety_work True safety_work True safety_work True safety_work True safety_work

Rys. 104 Terminal aktywacji skryptu systemu bezpieczeństwa

Posiadając gotowy system, należy przejść do pomiarów prezentujących prawidłowe działanie funkcjonalności.

8. Metodyka badań i interpretacji wyników

8.1 Metoda badawcza

Wykresy pomiarowe zostaną zbudowane na podstawie czterech informacji publikowanych na *Topic*: /bool_topic (system bezpieczeństwa), /cmd_vel (sygnał zadania ruchu z klawiatury), /cmd_vel_nav (sygnał zadania ruchu z nawigacji) oraz /closest_point (odległość do najbliższego elementu). Każda z wartości opartej na *Topic*, będzie funkcją zależną od tego samego czasu. Zostaną wykonane 3 pomiary trwające 50 sekund każdy: Dojazd do punktu x=3.0 (m), y=4.0 (m), odległość krytyczna 1 (m), dojazd do punktu x=5.0 (m), y=2 (m), odległość krytyczna 1 (m), dojazd do punktu x=5 (m), y=2 (m), odległość krytyczna 0,5 (m)

8.2 System pomiarowy

Zmiany w skrypcie Nav_stop_mechanism.py

W celu pobieraniu danych o stanie systemu bezpieczeństwa, dodano funkcjonalność publikującą *Topic* /bool_topic, którego informacje wysyłane są do sieci ROS2 co 0.05 sekundy. Realizacja zadania polega na zaimplementowaniu metody timer_callback, która wykrywa stan systemu. Jeżeli system bezpieczeństwa jest aktywny wysyłana jest wartość „True”, jeżeli nie „False”.

```
self.action_client = ActionClient(self, NavigateToPose, '/navigate_to_pose')
self.first_goal_msg = None
self.publisher_bool = self.create_publisher(String, '/bool_topic', 10)
self.timer_period = 0.05 # seconds
self.timer = self.create_timer(self.timer_period, self.timer_callback)

def timer_callback(self):
    msg = String()
    if self.safety == False:
        msg.data = 'False'
        print(msg)
        self.publisher_bool.publish(msg)
    elif self.safety == True:
        msg.data = 'True'
        self.publisher_bool.publish(msg)
```

Rys. 105 funkcja timer_callback

Skrypt pomiarowy

Informacja pobierana z *Topic* zapisywana jest jako 0 albo 1, to znaczy brak lub istnienie sygnału pochodzącego z danego źródła. Dane zapisywane są w plikach csv. W przypadku *Topic* mierzącego odległość najbliższego od punktu, dane zwracane są w metrach. Pomiar każdej wielkości rozpoczyna się w tym samym czasie, dane pobierane co sekundę. Uruchomienie skryptu realizowane poprzez wpisanie komendy:

```
root@DevOps:~/dev_ws/src/my_custom_package/scripts# python3 Data_interceptor.py
```

Rys. 106 Uruchomienie skryptu pobierającego dane pomiarowe

Main skryptu pomiarowego

Standardowa pętla utrzymująca aktywność węzła. Jeżeli czas TIME_LIMIT + 1 przekroczy zadaną wartość zdefiniowaną w pierwszych liniach kodu, następuje koniec pomiarów.

```
data_subscriber = DataSubscriber()

while rclpy.ok():
    rclpy.spin_once(data_subscriber)
    if data_subscriber.time == TIME_LIMIT + 1:
        break
    # Destroy the node explicitly
    # (optional - otherwise it will be done automatically
    # when the garbage collector destroys the node object)
    data_subscriber.destroy_node()
rclpy.shutdown()
```

Rys. 107 Main skryptu pomiarowego

Subskrybcje

Cztery subskrybcje, na rysunku pokazana konstrukcja każdej z nich. Różnica to nazwa funkcji *callback* i *Topic*.

```
self.subscription_one = self.create_subscription(
    String,
    '/bool_topic',
    self.callback_one,
    10)
```

Rys. 108 Subskrybcja

Standardowa funkcja callback

Jeżeli zwracana jest wartość True lub False, zapis danych w zwykłej zmiennej. Jeżeli wartości w postaci współrzędnych, w atrybutach obiektu.

```
def callback_one(self, msg: String):
    self.callback_one_msg = msg
    #CALL_FUN 2
def callback_two(self, msg: Twist):
    self.callback_two_msg.x = msg.linear.x
    self.callback_two_msg.y = msg.angular.z
```

Rys. 109 Standardowa funkcja callback

Timer

Definiuje co jaki czas ma zostać wywołana kolejna funkcja callback, która dodatkowo odpowiednio interpretuje i przesyła dane do pliku .csv.

```
self.timer_two = self.create_timer(1.0, self.timer_two_callback)
# TIM 3
```

Rys. 110 Timer

Interpretacja i przesyłanie do csv – Callback dla SafetyInfo

Ze względu na zawartość wiadomości „callback_one_msg” należało zastosować filtr oparty o regex, który zapisuje w zmiennej result wszystko co znajduje się między apostrofami. Ustalono interpretacje: Jeżeli wartości zmiennej result to True, przypisz 1 dla zmiennej message i wyslij pliki do SafetyInfo.csv. Analogicznie dla wartości False zawartej w zmiennej result – przypisanie 0 i wysłanie do csv.

```
def timer_one_callback(self):
    self.time += 1
    message = str(self.callback_one_msg)
    match = re.search(r'"(.*)"', message)
    if match:
        result = match.group(1)
    else:
        result = None
    self.callback_one_msg_timer = result
    print(result)
    print(type(result))

    if result == "False" or result == None:
        message = "0"
        print(message)
    else:
        message = "1"
        print(message)
    self.df_bool_topic = self.dataframe_merge(message, self.df_bool_topic)
    self.df_bool_topic.to_csv('SafetyInfo.csv', index=False)
```

Rys. 111 Wysyłanie danych do SafetyInfo.csv

Interpretacja i przesyłanie do csv – Callback dla CmdVel

Gdy nawigacja działa, wysyła sygnały również na *Topic /cmd_vel*. Dlatego aktywność jazdy nie wywołanej przez NAV2, tylko przez operatora, definiowana jest w następujący sposób: Jeżeli */cmd_vel* zwraca jakiekolwiek wartości ruchu w osi x lub z oraz nie jest atywna nawigacja (*self.cmd_message == 0*), operator steruje robotem. W przeciwnym wypadku wartość message równa jest 0. Wartość zmiennej message zapisywana jest do pliku *CmdVel.csv*.

```
def timer_two_callback(self):
    if ((self.callback_two_msg.x != 0 or self.callback_two_msg.y != 0) and self.cmd_message == "0"):
        message = "1"
        print(message)
    else:
        message = "0"
        print(message)
    self.df_cmd_vel = self.dataframe_merge(message, self.df_cmd_vel)
    self.df_cmd_vel.to_csv('CmdVel.csv', index=False)
```

Rys. 112 Wysyłanie danych do *CmdVel.csv*

Interpretacja i przesyłanie do csv – Callback dla NavVel

Struktura taka sama jak dla plików zapisywanych do *CmdVel*, z tą różnicą, iż */cmd_nav_vel* działa tylko wtedy gdy nawigacja jest aktywna. Wartość zmiennej *self.cmd_message* zapisywane w *NavVel.csv*.

```
def timer_three_callback(self):
    if self.callback_three_msg.x != 0 or self.callback_three_msg.y != 0:
        self.cmd_message = "1"
        print(self.cmd_message)
    else:
        self.cmd_message = "0"
        print(self.cmd_message)
    self.df_cmd_vel_nav = self.dataframe_merge(self.cmd_message, self.df_cmd_vel_nav)
    self.df_cmd_vel_nav.to_csv('NavVel.csv', index=False)
```

Rys. 113 Wysyłanie danych do *NavVel.csv*

Interpretacja i przesyłanie do csv – Callback dla Distance

Obliczenie odległości najbliższego punktu z twierdzenia pitagorasa, wynik zapisany z dokładnością czterech miejsc po przecinku, wartość zapisana w pliku Distance.csv.

```
def timer_four_callback(self):
    distance_to_robot = round((self.callback_four_msg.x ** 2 + self.callback_four_msg.y ** 2) ** 0.5,4)
    self.df_closest_point = self.dataframe_merge(distance_to_robot, self.df_closest_point)
    self.df_closest_point.to_csv('Distance.csv', index=False)
    print(distance_to_robot)
    print(self.time)
    print(self.df_closest_point)
```

Rys. 114 Wysyłanie danych do Distance.csv

Przesyłanie plików do excel

Wszystkie pliki wejściowe wykresów .csv wygenerowane w środowisku robota zostały wgrane do programu excel za pomocą narzędzia „Z pliku tekstowego/CSV”.

A	B	C	D	E	F
value	time				
0	1				
0	2				
0	3				
0	4				
0	5				
0	6				
0	7				

Rys. 115 Dane wgrane do Excel

CmdVel.csv	Data_interceptor.py
Distance.csv	Nav_Start.py
Nav_Stop_mechanism_3.py	Nav_Stop_mechanism_te...
Nav_Stop_mechanism.py	Nav_Stop_mechanism2.p...
Nav_Stop_mechanism2.py	Nav_Stop.py
NavVel.csv	Point_Checker_2.py
Point_Checker_3_test.py	Point_Checker_3.py
Point_Checker_3.py	Point_in_space.py
SafetyInfo.csv	

Rys. 116 Dane csv

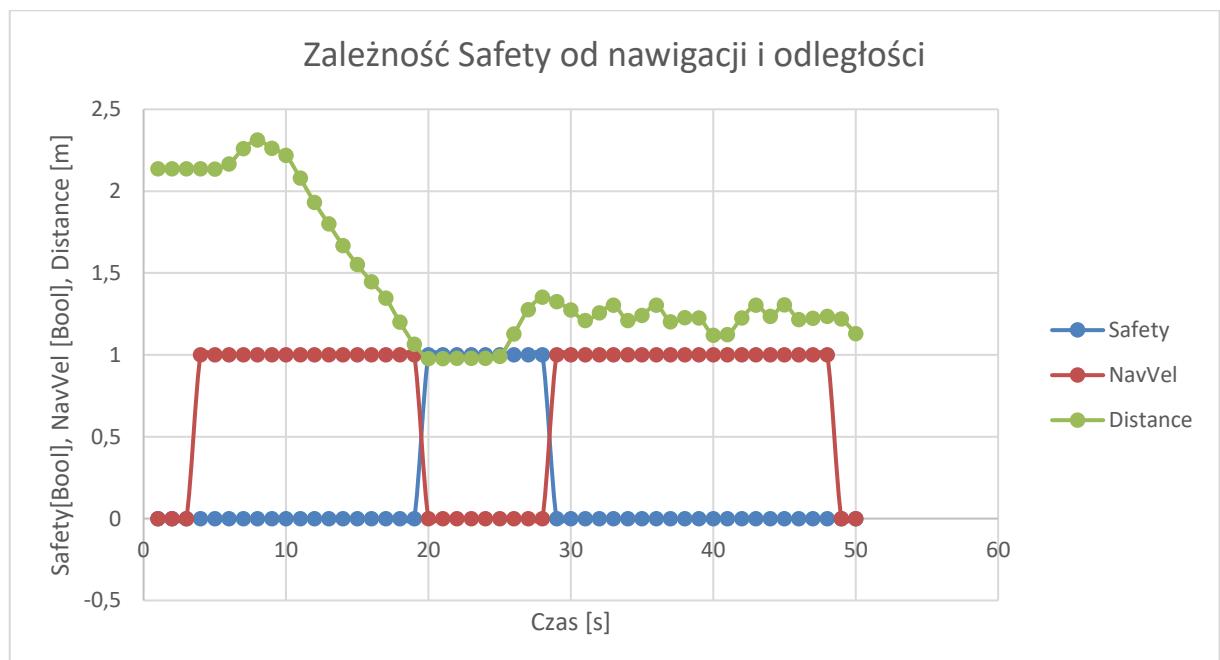
8.3 Pomiary

Metoda pomiarowa

Badanie oparte na dwóch wykresach. Pierwszy reprezentuje zależność działania systemu bezpieczeństwa od odległości do najbliższego punktu, gdzie sygnał nawigacji potwierdza, iż system rzeczywiście działa. Reaktywacja systemu, wymaga oddalenia robota od najbliższego punktu za pomocą ruchu zleconego przez operatora. Drugi wykres wskazuje na momenty, w których operator wymusza ruch robota.

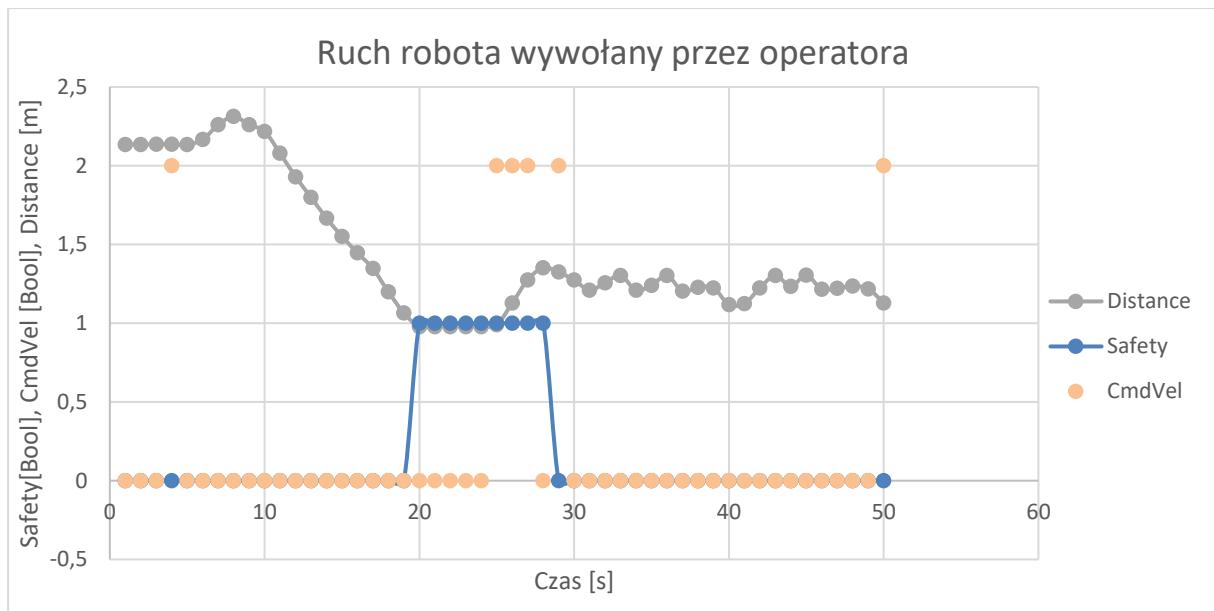
Pomiar 1

$x = 3 \text{ [m]}$, $y = 4 \text{ [m]}$, Odl. Krytyczna = 1 [m], czas rejestracji = 50 [s]



Rys. 117 Pomiar 1 - Zależność Safety od nawigacji i odległości

W chwilach 0-5 następuje załączenie systemu. Od sekundy 6 do 19 widoczna jest aktywność nawigacji oraz malejący charakter dystansu robota do najbliższego elementu. W momencie osiągnięcia odległości krytycznej w 20 sekundzie, załącza się system bezpieczeństwa, odległość do najbliższego punktu jest stała do momentu ruchu robota zleconego przez operatora w 25 sekundzie (CmdVel Rys.118). Po 4 sekundach od momentu nie spełnienia warunku krytycznej odległości, nawigacja załącza się, system bezpieczeństwa wyłącza się, robot nawiguje dalej do celu.

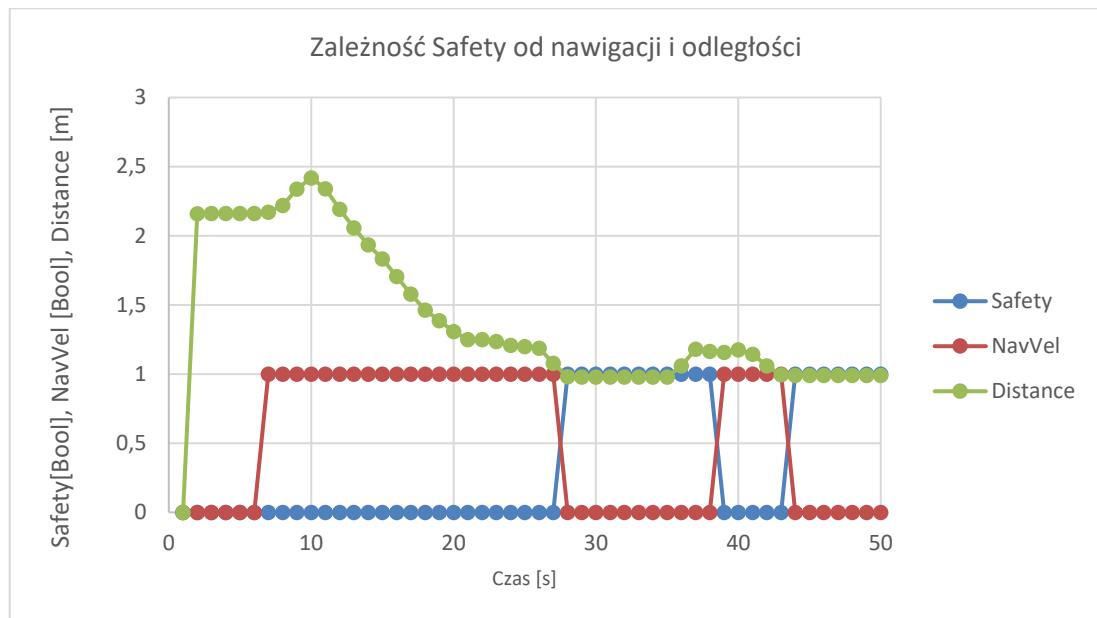


Rys. 118 Pomiar 1 - Ruch robota wywołany przez operatora

Nagle skoki CmdVel spowodowane są opóźnieniami pętli w algorytmie. *Topic* CmdVel naturalnie jest aktywny razem z NavVel, jednakże na potrzeby klarownego ukazania wyników, skrypt zakłada, że definiuje on tylko ruch spowodowany przez operatora. Z tego powodu zanim Skrypt bezpieczeństwa zostanie poprawnie załączony, CmdVel jest rejestrowany nawet wtedy, gdy działa nawigacja.

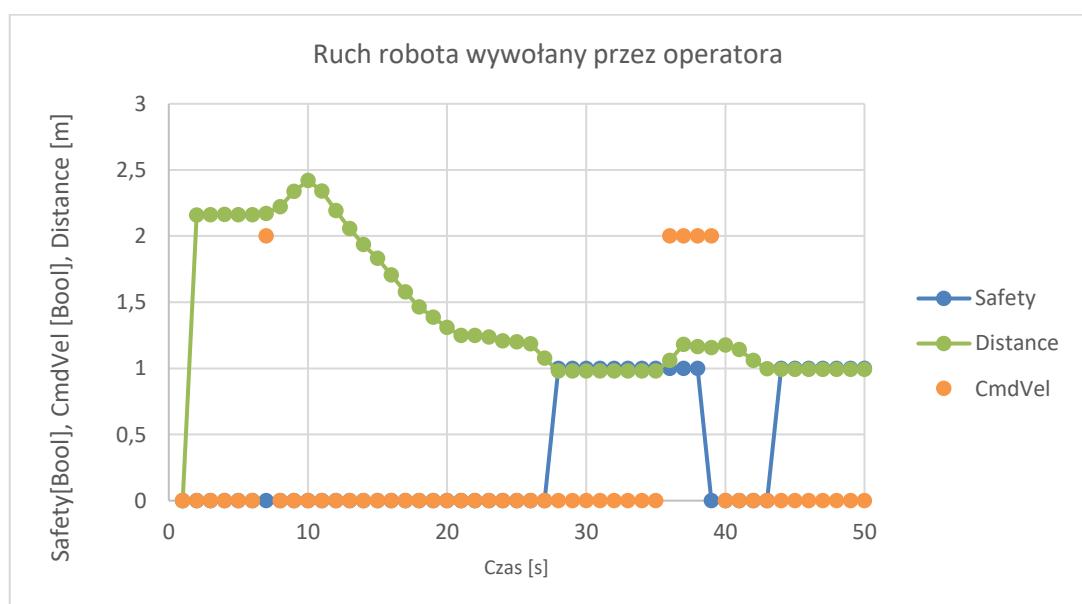
Pomiar 2

$x = 5 \text{ [m]}$, $y = 2 \text{ [m]}$, Odl. Krytyczna = 1 [m], czas rejestracji = 50 [s]



Rys. 119 Pomiar 2 – Zależność Safety od nawigacji i odległości

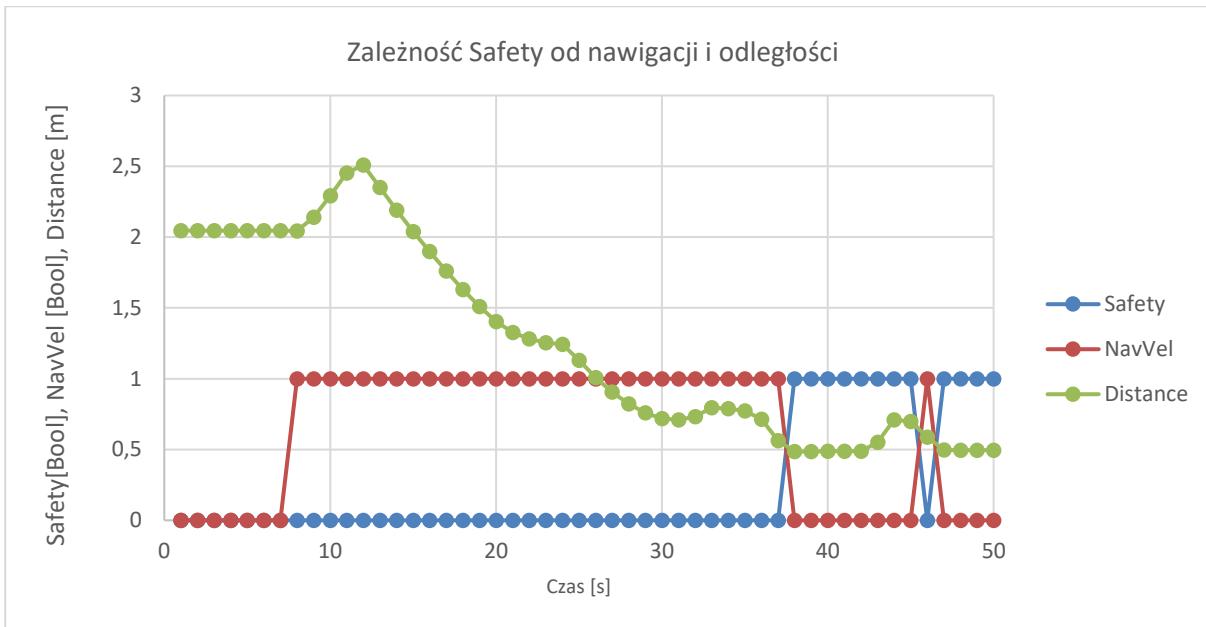
W chwilach 0-7 skrypt bezpieczeństwa nie jest uruchomiony. Po aktywacji, robota zaczyna nawigować do momentu 28 sekundy, gdzie system bezpieczeństwa załącza się w wyniku zbliżenia się do najbliższego punktu (Distance = 1m). W sekundzie 36 operator odjeżdża robotem, po czasie czterech sekund system bezpieczeństwa zostaje wyłączony nawigacja załącza się. W 44 sekundzie robot ponownie osiąga krytyczną odległość, system safety pozostaje aktywny do końca badania.



Rys. 120 Pomiar 2 - Ruch robota wywołany przez operatora

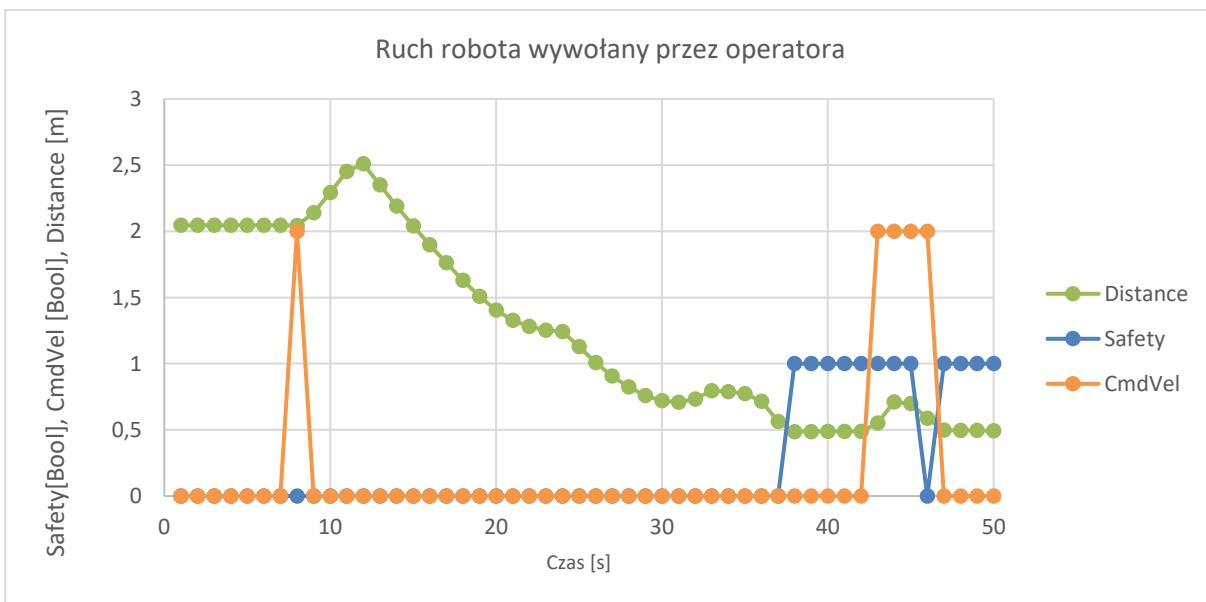
Pomiar 3

$x = 5 \text{ [m]}$, $y = 2 \text{ [m]}$, Odl. Krytyczna = 0.5 [m], czas rejestracji = 50 [s]



Rysunek 121 Pomiar 3 - Zależność Safety od nawigacji i odległości

Badanie różni się od pozostałych definicją odległości krytycznej. W 8 sekundzie zaczyna nawigować, w sekundzie 38 system bezpieczeństwa zostaje załączony w wyniku dojazdu do miejsca odległego od najbliższego punktu o 0,5 m. W 43 sekundzie operator odjeżdża robotem, jednakże na tak małą odległość, iż po sekundzie działania punkt położenia robota ponownie znajduje się w położeniu krytycznej odległości i system bezpieczeństwa zostaje załączony.



Rys. 122 Pomiar 3 - Ruch robota wywołany przez operatora

9. Podsumowanie

Dzięki dodaniu funkcjonalności systemu bezpieczeństwa opartego na skanerze LIDAR, robot z właściwą dokładnością reaguje na pojawienie się przeszkód w krytycznie bliskiej odległości. Prosta możliwość parametryzacji systemu, czyli definiowanie krytycznej odległości, umożliwia przystosowanie systemu bezpieczeństwa robota do różnych zadań, które charakteryzowane są przez potrzeby funkcjonującego w fabryce procesu. Fakt bezpośredniej integracji z platformą *ROS2* znacznie upraszcza problematykę dalszego rozwoju istniejącego rozwiązania, gdzie programista ma pełną kontrolę nad sposobem działania, który można zmieniać zależnie od potrzeb klienta. Na podstawie zebranych danych można stwierdzić, iż system odpowiednio szybko reaguje na zaistnienie warunków załączenia. Wykresy ze względu na rozdzielcość pobranych danych wskazują na sekundowe opóźnienie, w rzeczywistości wyniki są mniejsze, a optymalizacja zagadnienia jest wyłącznie kwestią dalszego rozwoju oprogramowania.

Środowisko symulacyjne jest idealnym rozwiązaniem w kontekście prowadzenia badań nad nowymi zagadnieniami powiązanymi z lokalizacją czy nawigacją. Oprogramowanie konstruowane w celach wdrożenia algorytmów interpretujących sygnały ze skanerów takich jak LIDAR z pewnością może być rozwijane za pomocą programów symulacyjnych, gdyż z dużą dokładnością przedstawiają ich rzeczywiste działanie. Kolejnym atutem projektu jest jego potencjał dydaktyczny – struktura oprogramowania wykorzystana w symulacji, nie różni się niczym oprócz wtyczek symulacyjnych od kodu funkcjonującego na rzeczywistych robotach. Platforma *ROS2* jest szeroko stosowana w przemyśle do projektowania systemów robotów.

Proces rozwoju przebiegał sprawnie, dzięki zastosowaniu narzędzia do konteneryzacji *Docker*. W przypadkach napotkania sytuacji zacięcia się symulacji, błędów w systemie lub generalnie trudnych do zlokalizowania źródeł awarii, wystarczy uruchomić zapisaną wcześniej wersję tego samego systemu z innego kontenera. Oprogramowanie wizualizacyjne *RViz2* usprawnia prostotę i szybkość procesu weryfikacji poprawności działania napisanego kodu.

Podsumowując: System platformy robota oparty na platformie *ROS2*, to rozwiązanie pozwalające na skorzystanie z wielu udogodnień ułatwiających proces rozwoju projektu, między innymi takich jak nowoczesne narzędzia informatyczne *Docker*. Wykorzystanie Symulacji nie tylko pozwala na przeprowadzenie testów algorytmów przy niskich kosztach i braku ryzyka zepsucia prawdziwego robota, ale również posiada potencjał dydaktyczny. Zaletą zaprojektowanego systemu bezpieczeństwa jest: prostota, wysoka elastyczność w ramach wprowadzania zmian w funkcjonalności i niski koszt implementacji.

10. Wyniki i wnioski

- Informacje pozyskane z wykresów prezentują dane, które wskazują na opóźnienia czasowe między załączeniem różnych części systemu, takich jak ruch zlecony przez nawigacje czy system bezpieczeństwa. Z racji rozdzielczości wykresu trudno zinterpretować wartości opóźnień, ale jest to zagadnienie jak najbardziej kluczowe w kontekście dalszych prac rozwojowych nad systemem.
- Podczas prac nad projektem napotkano na różne przeszkody takie jak zawieszenia systemu, czy generalnie błędy których charakter był trudny do wyjaśnienia. Narzędzie *Docker* pozwoliło na szybkie przywrócenie działających wersji poprzez włączanie uprzednio zbudowanych kontenerów. Natura systemu kontenera zawierającego tylko potrzebne zależności konieczne do rozruchu robota, z pewnością zapobiegła dodatkowym awariom, które mogły mieć miejsce w przypadku pracy z systemem *Ubuntu* przeznaczonym do prac biurowych.
- Modułowa postać platformy *ROS2* pozwala na kontrolowanie aktywnych funkcjonalności. Szczególnie przydatne przy analizowaniu awarii systemu, gdzie w przypadku wyłapania błędu poprzez analizę przechwytywanych informacji ukazujących się w terminalu danego modułu, programista może łatwo zidentyfikować przyczynę i wyłączyć funkcjonalność.
- Ważną częścią programowania platformy robota jest zagadnienie optymalnego wykorzystywania pamięci RAM. Niepoprawna implementacja algorytmów może prowadzić do przepełniania pamięci danymi, które są bezużytecznymi artefaktami z wykonanych procesów. W przypadku zaprojektowanego systemu jest to problem, który należy rozwiązać w dalszych pracach rozwojowych, gdyż program liniowo z czasem zajmuje rosnące zasoby pamięci RAM.
- Struktura Publikacji / Subskrypcji platformy *ROS2*, umożliwia w dość intuicyjny sposób konstruować system komunikacyjny pomiędzy elementami robota. Dodatkowo przydatna w przypadku tworzenia danych statystycznych, w sposób zaprezentowany w pracy dyplomowej

11. Bibliografia

- [1] Versabox S.A., „versabox.eu,” 2022. [Online]. Available: <https://versabox.eu/pl/>.
- [2] Versabox S.A., „versabox.eu,” 2022. [Online]. Available: <https://versabox.eu/smart-intralogistics/>.
- [3] Aethon, „aethon.com,” 2018. [Online]. Available: <https://aethon.com/intralogistics-is-the-secret-to-a-smart-supply-chain/>.
- [4] Omron, „Omron-ap.com,” 2023. [Online]. Available: <https://www.omron-ap.com/solutions/robotic-solutions/>.
- [5] plc-trade, „plc-trade.com,” 2024. [Online]. Available: <https://plc-trade.com/mpn/s30a-7011ca/>.
- [6] D. Goodwin, „control.com,” 9 Wrzesień 2020. [Online]. Available: <https://control.com/technical-articles/the-evolution-of-autonomous-mobile-robots/#:~:text=William%20Grey%20Walter%20developed%20the,to%20simulate%20two%20connected%20neurons>.
- [7] A. Sheth, „medium.com,” 2017. [Online]. Available: <https://medium.com/bloomberg/history-of-machine-learning-7c9dc67857a5>.
- [8] Holloway, „holloway.com,” 2 Listopad 2022. [Online]. Available: <https://www.holloway.com/g/making-things-think/sections/a-brief-history-of-ai>.
- [9] Haiyn, „dreamstime.com,” [Online]. Available: <https://www.dreamstime.com/autonomous-mobile-robots-charging-modern-warehouse-autonomous-mobile-robots-charging-modern-warehouse-warehouse-automation-image161315324>.
- [10] K. Searles, „roboticsandautomationmagazine.co.uk,” [Online]. Available: <https://www.roboticsandautomationmagazine.co.uk/news/amrs/geek-and-engineer-create-automated-intelligent-cold-chain-port-warehouse.html>.
- [11] AGVnetwork, „www.agvnetwork.com,” 2021. [Online]. Available: <https://www.agvnetwork.com/amr-market-size>.
- [12] O. N. LLC, „linkedin.com,” 2024. [Online]. Available: https://www.linkedin.com/posts/open-nav_nav2-nav2-ros-activity-7127429063945728000-RXIN/.
- [13] Wellwit Robotics, „ozrobotics.com,” [Online]. Available: <https://ozrobotics.com/shop/composite-collaboration-amr-with-robotic-arm-and-gripper/>.

- [14] Versabox S.A., „versabox.eu,” [Online]. Available: <https://versabox.eu/pl/product/roller-module/>.
- [15] Amc Sp. z o.o., „amc.waw.pl,” 2024. [Online]. Available: <https://www.amc.waw.pl/symulacja-witness/33-zarzpdzanie-procesem/466-opis-programu-witness>.
- [16] Wikipedia, „pl.wikipedia.org,” [Online]. Available: https://pl.wikipedia.org/wiki/Symulacja_komputerowa.
- [17] Versabox S.A., „versabox.eu,” 2022. [Online]. Available: <https://versabox.eu/pl/product/virtual-factory/>.
- [18] Greatmc, „greatmc.pl,” [Online]. Available: <https://greatmc.pl/portfolio-items/012-versabox/>.
- [19] Amc Sp z o.o., „amc.waw.pl,” 2024. [Online]. Available: [https://www.amc.waw.pl/bli%C5%BAAniak-cyfrowy/wprowadzenie-dobli%C5%BAAniak%C3%B3w-cyfrowych#:~:text=Wizualna%20Interaktywna%20Symulacja%20\(VIS\)%20za,powi%C4%85za%C5%84%C2%20charakterystyk%20i%C2%20potrzebnych%20zasob%C3%B3w](https://www.amc.waw.pl/bli%C5%BAAniak-cyfrowy/wprowadzenie-dobli%C5%BAAniak%C3%B3w-cyfrowych#:~:text=Wizualna%20Interaktywna%20Symulacja%20(VIS)%20za,powi%C4%85za%C5%84%C2%20charakterystyk%20i%C2%20potrzebnych%20zasob%C3%B3w).
- [20] premiumolutions, „premiumolutions.pl,” 2024. [Online]. Available: <https://premiumolutions.pl/program/solidworks-simulation/>.
- [21] Mastercam, „mastercam.pl,” 2024. [Online]. Available: <https://mastercam.pl/>.
- [22] P. Zadroga, „iautomatyka.pl,” 2018. [Online]. Available: <https://iautomatyka.pl/co-to-jest-automation-studio-recenzja-oprogramowania-od-br-automation/>.
- [23] iridalabs, „iridalabs,” [Online]. Available: <https://iridalabs.com/blog/synthetic-training-data-tools-computer-vision-ai/>.
- [24] Gazebo, „gazebosim.org,” [Online]. Available: <https://gazebosim.org/about>.
- [25] D. Chikurtev, „researchgate.net,” [Online]. Available: https://www.researchgate.net/figure/Principal-Robot-Simulation-in-Gazebo_fig1_348312677.
- [26] P. Oziębło, „versabox.eu,” 2021. [Online]. Available: <https://versabox.eu/pl/run-at-rate-czyli-wirtualny-test-wdrozenia-amr>.
- [27] M. Dragan, „ubs.com,” 2020. [Online]. Available: <https://www.ubs.com/global/en/our-firm/what-we-do/technology/2020/next-level-software-development-on-devcloud.html>.
- [28] docker.com, „docker.com,” [Online]. Available: <https://www.docker.com/resources/what-container/>.

- [29] Wikipedia, „en.wikipedia,” [Online]. Available: <https://en.wikipedia.org/wiki/Ubuntu>.
- [30] K. Buzdar, „linuxhint.com,” [Online]. Available: <https://linuxhint.com/install-nano-ubuntu-22-04/>.
- [31] O. Robotics, „docs.ros.org,” [Online]. Available: <https://docs.ros.org/en/humble/Concepts/Basic.html>.
- [32] polskagrupa.it, „polskagrupa.it,” [Online]. Available: <https://polskagrupa.it/2021/05/03/middleware-co-to-jest-i-jak-wplywa-na-innowacyjnosc/>.
- [33] BotBuilder, „youtube.com,” 2021. [Online]. Available: https://www.youtube.com/watch?v=VDtB-TkcX7k&ab_channel=BotBuilder.
- [34] theconstructsim, „www.theconstructsim.com,” [Online]. Available: https://www.theconstructsim.com/robotigniteacademy_learnros/ros-courses-library/ros2-basics-python/.
- [35] learn.microsoft, „learn.microsoft.com,” [Online]. Available: <https://learn.microsoft.com/en-us/devops/develop/git/set-up-a-git-repository>.
- [36] javappa, „javappa.com,” [Online]. Available: <https://javappa.com/linux-dla-programisty-java/dystrybucje-linuxa>.
- [37] theurbanpenguin.com, „theurbanpenguin.com,” [Online]. Available: <https://www.theurbanpenguin.com/calculate-boot-time-ubuntu-16-04/>.
- [38] docs.docker, „docs.docker.com,” [Online]. Available: <https://docs.docker.com/engine/install/ubuntu/>.
- [39] Z. U. Abideen, „linkedin.com,” [Online]. Available: <https://www.linkedin.com/pulse/docker-zain-ul-abideen/>.
- [40] J. Newans, „github.com,” [Online]. Available: https://github.com/joshnewans/articubot_one/tree/782fe052eb881ffe7f92f7f3363dfef65eec6fb6.
- [41] A. Robotics, „articulatedrobotics.xyz,” [Online]. Available: <https://articulatedrobotics.xyz/mobile-robot-2-concept-urdf/>.
- [42] J. Newans, „articulatedrobotics.xyz,” [Online]. Available: <https://articulatedrobotics.xyz/mobile-robot-3-concept-gazebo/>.
- [43] J. Newans, „articulatedrobotics.xyz,” [Online]. Available: <https://articulatedrobotics.xyz/mobile-robot-8-lidar/>.
- [44] J. Newans, „Easy SLAM with ROS using slam_toolbox,” [Online]. Available:

https://www.youtube.com/watch?v=ZaiA3hWaRzE&list=PLunhqkrRNhYAffV8JDiFOatQXuU-NnxT&index=17&ab_channel=ArticulatedRobotics.

- [45] J. Newans, „Making robot navigation easy with Nav2 and ROS!,” [Online]. Available:
https://www.youtube.com/watch?v=jkoGkAd0GYk&list=PLunhqkrRNhYAffV8JDiFOatQXuU-NnxT&index=18&ab_channel=ArticulatedRobotics.
- [46] Ubuntu, „<https://ubuntu.com>,” [Online]. Available:
<https://ubuntu.com/tutorials/install-ubuntu-desktop#1-overview>.
- [47] athackst, „github.com/athackst,” [Online]. Available:
<https://github.com/athackst/dockerfiles/blob/main/ros2/galactic.Dockerfile>.
- [48] JoshNewans, „github.com/joshnewans/my_bot,” [Online]. Available:
https://github.com/joshnewans/my_bot.
- [49][Github, „docs.github.com,” [Online]. Available:
<https://docs.github.com/en/authentication/connecting-to-github-with-ssh>.