Politechnika Warszawska
WYDZIAŁ MECHANICZNY
ENERGETYKI I LOTNICTWA

Instytut Techniki Lotniczej i Mechaniki Stosowanej

na kierunku Robotyka i Automatyka
w specjalności Robotyka

promotor
dr Andrzej Chmielniak

Warszawa, 2022

Spis treści

Streszczenie pracy

Pierwszym celem pracy dyplomowej jest przeprowadzenie analiz i eksperymentów sprawdzających możliwości stworzenia oprogramowania autopilota do pojazdów naziemnych. Następnym celem tej pracy były analiza metod procesowania danych z czujników oraz zapoznanie się z rodzajami filtracji danych.

Rozdziały do 11 włącznie są rozdziałami teoretycznymi. Od rozdziału 12 rozpoczyna się opis implementacji oprogramowania autopilota.

Zakres pracy dyplomowej obejmuje zagadnienia związane z robotyką oraz informatyką. W zakres robotyki wchodzi obsługa czujników, filtracja danych, konwersja danych z czujników, dobór układu współrzędnych czy też teoria napędów. W zakres informatyki wchodzi architektura oprogramowania, programowanie równoległe oraz komunikacja międzyprocesowa.

Dokonano przeglądu dostępnych filtrów. Przeprowadzono eksperymenty i analizy danych otrzymanych podczas procesów filtrowania różnymi metodami filtrowania danych. Przeanalizowano w ten sposób wyniki dla danych o położeniu oraz orientacji robota. Dokonano również analizy myślowej na temat doboru niezbędnych podzespołów (takich jak na przykład napęd robota), które wymagane są przez oprogramowanie autopilota. Na końcu poddano analizie różnice pomiędzy robotem obecnym w symulacji a rzeczywistą maszyną.

Zaprojektowano oprogramowanie autopilota pozwalającego na wykonywanie zautomatyzowanych misji. Wykorzystano w tym celu ROS jako platformę programistyczną. Do stworzenia tego rozwiązania wykorzystano takie języki jak C++ oraz Python.

Na podstawie wyników analiz i eksperymentów wywnioskowano, że stworzone oprogramowanie jest w stanie sprostać postawionemu celowi z pewnymi uproszczeniami. Stwierdzono również, że filtr Kalmana jest najlepszym kandydatem na filtrowanie pozycji oraz orientacji robota. Stwierdzono dodatkowo, że istniejące oprogramowanie można i należałoby je rozwinąć.

Słowa kluczowe: autopilot, GPS, Kalman, filtrowanie, ROS

Abstract

The first purpose of this thesis is to conduct analyses and experiments which check the possibilities of autopilot software creation for ground vehicles. The next purpose is an analysis of data processing methods from sensors and getting familiar with data filtration methods.

Chapters up to and including 11 are theoretical chapters. The description begins in Chapter 12 autopilot software implementation.

The scope of the thesis includes issues related to robotics and computer science. The robotic scope includes sensors service, data filtration, data conversion from sensors, coordinate frame selection, and drive theory. The computer science scope includes software architecture, parallel programming, and interprocess communication.

Made a review of available filters. Conducted experiments and analyses of acquired data during filtration processes by various methods of data filtering. In this way analyzed results for data about the robot's location and orientation. Made also mental analyses on the subject of necessary components (such as robot drive), which are required by autopilot software. In the end, the difference between the simulated robot and the real one was analyzed.

The autopilot software was designed to allow the execution of automated missions. For this purpose, ROS was used as a development platform. To create this languages such as C++ and Python were used.

Based on the analyzes results and experiments it was concluded that creating software can meet the goal with certain simplifications. Also found that the Kalman filter is the best filter for position and orientation filtration. Additionally, it was discovered that existing software should be upgraded.

Keywords: autopilot, GPS, Kalman, filtering, ROS

1 Wprowadzenie

1.1 Przedstawienie rozwiązania

Rozwiązanie, które przedstawiam w tej pracy dyplomowej, to oprogramowanie autopilota do sterowania robotami naziemnymi. Sterowanie rozumiemy tutaj jako wyznaczanie trajektorii skomponowanej z odcinków, które łączą punkty na mapie. Punkty są przekazywane w pliku misji. Ogólny pomysł jest taki, aby robot wyposażony w to oprogramowanie był w stanie pokonywać zadaną trajektorię w sposób autonomiczny. Punkty są podane w postaci współrzędnych geograficznych tzn. szerokości i długości geograficznej.

1.2 Motywacja

Motywacją do stworzenia tego oprogramowania była chęć nauczenia się:

- modelowania robotów w symulacjach,
- filtrowania danych pomiarowych,
- projektowania oprogramowania złożonego z wielu procesów.

Na rynku są autopiloty dla bezzałogowych pojazdów latających, ale istnieje deficyt autopilotów dla robotów naziemnych. Dodatkowo takie oprogramowanie ma dużo innych zastosowań i może pomóc zautomatyzować zachowania wielu pojazdów.

1.3 Cel

Celem pracy było stworzenie oprogramowania, które nada robotowi naziemnemu pewne cechy autonomiczności. Głównym celem było osiągnięcie tego, by użytkownik nie musiał manualnie sterować pojazdem, tę czynność często można zautomatyzować. Zwiększona efektywność oraz produktywność takiego pojazdu w zastosowaniach komercyjnych mogłaby być zapewniona przez wyposażenie pojazdu w algorytmy, które obliczają trasę samodzielnie, wykorzystując informacje o terenie, przeszkodach oraz sieci istniejących dróg. Zaletą takiego rozwiązania byłoby zmniejszenie wydatków firmy używającej takiego oprogramowania w swoich pojazdach.

1.4 Możliwe zastosowania

Robot wyposażony w takie oprogramowanie może zostać wykorzystany do patrolowania zadanej trasy. Można nakazać takiemu robotowi poruszać się cyklicznie według zadanych trajektorii według zadanego harmonogramu. Można również wyposażyć większe pojazdy w to oprogramowanie i zaplanować im przewożenie ładunków na otwartym terenie. Takiego oprogramowania można użyć wszędzie tam, gdzie widoczność jest ograniczona, i mamy pewność, że teren nie zawiera przeszkód, na przykład nocne patrole na lotniskach oraz ich pasach startowych.

Zastosowanie tego oprogramowania znajdzie się również w transport ładunków wzdłuż kolei (na przykład przy jej budowie bądź modernizacji) na większe odległości. Przy sprzężeniu z czujnikami laserowymi lub kamerami można wyposażyć takie oprogramowanie w możliwość detekcji i omijania przeszkód.

Oprogramowanie może mieć również zastosowanie w transporcie towarów pomiędzy poszczególnymi ich poszczególnymi odbiorcami. Za pomocą takiego oprogramowania można również zautomatyzować sprzęt rolniczy. Tutaj takie oprogramowanie będzie musiało współpracować również z kamerami oraz oprogramowaniem osprzętu rolniczego.

1.5 Funkcje autopilota

Autopilot ma za zadanie przyjąć plik z misją i ją wyegzekwować. Ma on za zadanie sterować prędkością kątową i liniową robota podczas procedury realizacji misji tak, by osiągał on kolejne punkty, które znajdują się w pliku misji.

1.6 Wymagany sprzęt

Autopilot wykorzystuje szereg czujników w celu wyznaczenia aktualnej orientacji oraz położenia robota mobilnego. Do określenia aktualnego położenia wykorzystany został odbiornik GPS. Do aktualnej orientacji robota zostały wykorzystane IMU oraz magnetometr, które dostarczają absolutną orientację (względem globalnego układu współrzędnych) wyrażoną w radianach. Robot taki powinien mieć na pokładzie również odpowiedni osprzęt do komunikacji bezprzewodowej.

Dodatkowo sam robot musi być wyposażony w odpowiednie silniki, układ napędowy oraz zasilanie. Silniki i układ napędowy należy dobrać zależnie od zastosowania, gdyż inne wymagania będą stawiane na przykład na placu budowy, inne podczas transportu towarów po lokalnych i utwardzonych drogach oraz całkiem inne, jeśli robot będzie wykorzystywany w rolnictwie.

2 Opis metod programowania

2.1 Języki programowania

Python i C++ to dwa języki programowania ogólnego przeznaczenia, które są najczęściej wykorzystywane w robotyce. Python posłużył w pracy jako język, w którym odbywa się implementacja wstępnego rozwiązania, spełniającego wszystkie jego podstawowe funkcjonalności. Z drugiej strony język C++ zapewnia statyczne typowanie, co chroni przed przypadkowym niedopasowaniem typów. Język C++ sprawdza się w przypadku, gdy implementowane rozwiązanie musi charakteryzować się większą wydajnością względem Pythona. Wynika to z faktu, że jest to język kompilowany, a także oferuje niskopoziomowy dostęp do pamięci. Z kolei Python jest językiem, który umożliwia szybsze prototypowanie i pozwala na szybkie tworzenie aplikacji do rozwiązywania złożonych zadań w krótszym czasie. Dodatkowo Python jest językiem interpretowanym, który posiada duży zbiór bibliotek, dzięki czemu nadaje się do różnych celów, takich jak uczenie maszynowe, tworzenie stron internetowych i skryptów. Tak więc zarówno Python, jak i C++ mają swoje zalety i wady, gdy są używane w programowaniu robotyki i mogą być używane w zależności od sytuacji.

2.2 Przegląd architektur oprogramowania

2.2.1 Oprogramowanie zorientowane na mikroserwisy

Oprogramowanie zorientowane na mikroserwisy definiuje się jako takie, które zamiast jednego procesu (tzw. monolitu) wykorzystuje kilka małych procesów [28], gdzie każdy inny jest odpowiedzialny za inne zadanie. Procesy komunikują wykorzystując mechanizmy komunikacji międzyprocesowej. Jest to spowodowane tym, że wygodne jest przekształcenie kombinacji adresu IP oraz portu na nazwę, co ułatwia komunikację i pozwala na automatyczne uniknięcie kolizji.

Zalety takiego podejścia [28]:

- Ułatwia ono opracowywanie oprogramowania w zespole. Każda osoba pracuje nad jednym procesem.
- Umożliwia mieszanie technologii i języków programowania.
- Pozwala na wytworzenie oprogramowania maksymalnie prostego, czytelnego i dobrze przetestowanego.
- W razie potrzeby zmiany implementacji procesu wystarczy tylko go podmienić na inny z pozostawieniem takiego samego interfejsu.
- Zmiany w jednym procesie nie mają żadnego wpływu na inny proces.
- Ułatwia zintegrowanie się z innymi istniejącymi systemami.
- Pozwala zminimalizować użycie wielowątkowości, co zmniejsza liczbę błędów.

Wady takiego podejścia:

- Takie podejście w sposób geometryczny zwiększa liczbę połączeń pomiędzy komponentami systemu, zwiększa poziom skomplikowania systemu.
- Zwiększa liczbę kombinacji przypadków przy testowaniu oprogramowania.
- Następuje zwiększenie zużycia pamięci.

2.2.2 Oprogramowanie monolityczne

Architektura monolityczna jest typem oprogramowania, gdzie cały kod źródłowy uruchomiony jest jako jeden proces [56]. Takie podejście ma zastosowanie w małych projektach oraz prototypowaniu większych projektów. Podejście to sprawia, że wszystkie usługi i zasoby są osadzone w jednym miejscu. Ta architektura charakteryzuje się brakiem komunikacji międzyprocesowej.

Zalety takiego podejścia:

- Takie oprogramowanie jest bardzo łatwe do implementacji.
- Scentralizowane oprogramowanie charakteryzuje się większą wydajnością. Jest to spowodowane brakiem komunikacji międzyprocesowej.
- Oprogramowanie napisane w ten sposób jest łatwe do testowania.
- Łatwiejsze jest szukanie błędów, gdyż debugger może dotrzeć do konkretnego miejsca bezpośrednio, co jest niemożliwe w przypadku mikroserwisów.

Wady takiego podejścia:

- Wykonywanie różnych akcji równolegle wymaga zastosowania wielowątkowości.
- Rozwiązanie napisane w taki sposób jest trudne w skalowaniu.
- Błędy obecne w jednym module mają wpływ na inne moduły.
- Uzależnienie się od wąskiej grupy technologii i języków programowania.
- Mała zmiana w kodzie źródłowym wymaga ponownego wdrożenia całej aplikacji na docelową maszynę.

3 Środowisko ROS

3.1 Opis środowiska ROS

Roboting Operating System to zestaw pakietów oprogramowania, które stosuje się w robotyce [22]. Wszystkie jego pakiety są całkowicie otwartoźródłowe. Pakiety tego oprogramowania często są abstrakcyjnymi interfejsami pośredniczącymi pomiędzy sprzętem a kodem wysokopoziomowym. ROS nie jest systemem operacyjnym, mimo że tworzy własne środowisko w systemie operacyjnym, jest tylko zestawem pakietów i bibliotek.

Powstał on z myślą o tym, by oferować rozwiązania wielu problemów spotykanych w robotyce w postaci gotowych dobrze przetestowanych pakietów. Skraca to czas implementacji rozwiązania z zakresu robotyki. Istnieje duża liczba pakietów, które potrafią same obsłużyć sprzęt, wystawić lub przyjmować dane czytelne i zrozumiałe dla człowieka.

Oferuje on takie użyteczne pakiety jak na przykład:

- Pakiet **message_filters** służy do filtrowania wiadomości. Pozwala on na szeregowanie filtrów, tworząc zestaw filtrów.
- Pakiet **laser_geometry** służy do obsługi skanów z laserowych czujników odległości.
- Pakiet **urdf** pozwala budować roboty w symulacji. Jest to rozbudowany pakiet, który zawiera wystarczającą ilość możliwości, by zbudować symulację dowolnego robota mobilnego.
- Pakiet **image_transport** pozwala w wydajny sposób transportować zdjęcia za pomocą protokołu TCP/IP.
- Pakiet **dynamixel_sdk** służy do sterowania serwomechanizmami. Zapewnia on interfejs przystosowany do środowiska ROS, który w wygodny sposób pozwala na sterowanie wspomnianymi silnikami.

ROS działa w architekturze mikroserwisowej [28]. Uruchomione procesy mogą być przedstawione w postaci grafu. Oprogramowanie wytwarza się w postaci węzłów (ang. *nodes*), które komunikują się między sobą.

Komunikacja przebiega za pomocą wykorzystania modelu TCP/IP [36] oraz gniazd opisanych w rozdziale 4.1. Adres IP maszyny oraz numery portów są konwertowane na nazwę węzła, dzięki czemu użytkownik nie musi zarządzać samodzielnie architekturą sieciową na swojej maszynie. ROS pozwala również na komunikację pomiędzy kilkoma maszynami, wystarczy wystawić główny węzeł (roscore) na jednej maszynie i przekazać informację do drugiej, gdzie on się znajduje, podając adres IP i numer portu.

Komunikacja odbywa się na trzy główne sposoby:

- tematów (ang. *topics*),
- serwisów,
- akcji.

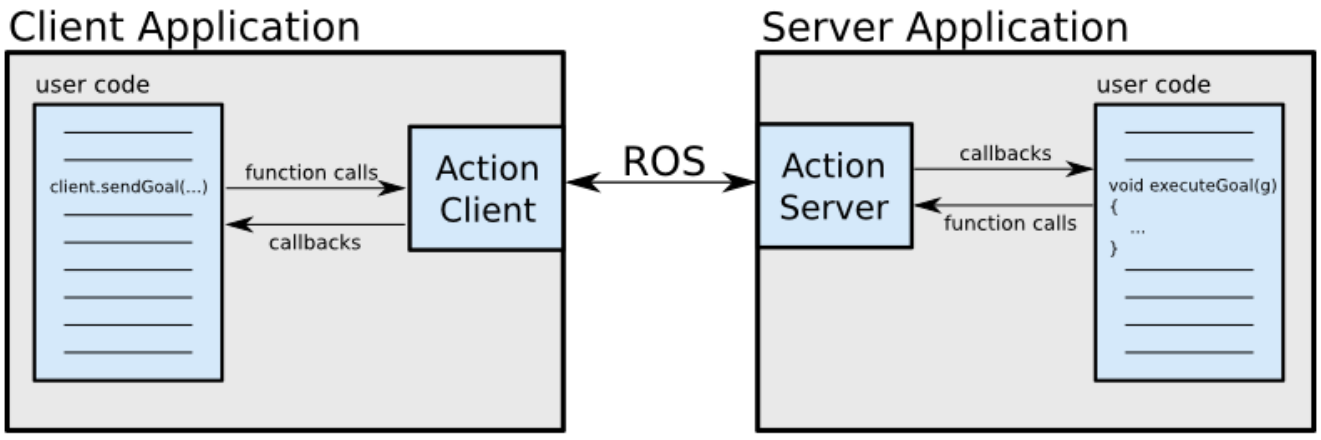
Tematy (ang. *topics*) [26] są najprostszą metodą komunikacji. Służą do wymiany wiadomości. Działają na podstawie modelu publisher – subscriber, gdzie jedne węzły nasłuchują danego tematu, a inne publikują na niego wiadomości. Jest to kierunkowa metoda komunikacji. Każdy temat ma swoją nazwę, po której można go znaleźć oraz swój format wiadomości np. int32.

Serwisy [25] działają na zasadzie zapytania oraz odpowiedzi. Jest to synchroniczny sposób komunikacji pomiędzy węzłami, gdzie jeden wysyła zapytanie i żąda również odpowiedzi. Przykładowo możemy zażądać od sterownika silnika krokowego o obrót 5^o w prawo, a w odpowiedzi dostaniemy ilość czasu, jaką zajął ten obrót w sekundach. Serwisy również mają swoje typy oraz nazwy tak samo, jak tematy.

Akcje [24] są najbardziej rozbudowanym sposobem komunikacji pomiędzy węzłami. Są podobne do serwisów. Są asynchroniczne, dzięki czemu węzeł nie będzie skupiony na oczekiwaniu na rezultat tak jak w przypadku serwisów, zatem będzie mógł wykonywać inne zadania w trakcie trwania akcji.

By rozpocząć akcję, trzeba do węzła wysłać tzw. cel (ang. *goal*) do serwera akcji umieszczonego w węźle. W międzyczasie otrzymujemy informację o postępie danej akcji (ang. *feedback*). Na końcu otrzymujemy rezultat tej akcji. Można akcję przerwać w dowolnym momencie. Wyobraźmy sobie robota wyposażonego w skaner laserowy. I chcemy, by skanował nam pokój przez 5 minut. Przykładem celu może być wysłanie do robota celu w postaci czasu skanowania (5 minut). Dane o postępie mogą być wysyłane, co sekundę w postaci aktualnej mapy otoczenia, a rezultatem może być to, czy akcja przebiegła pomyślnie.

Poniższy schemat przedstawia jak przebiega komunikacja w trakcie wykonywania akcji.



Rysunek 1: Schemat działania klienta oraz serwera akcji [24]

ROS zawiera predefiniowane zestawy najczęściej wykorzystywanych typów [34] wiadomości (np. Float32, String, LaserScan), serwisów oraz akcji. Formaty wiadomości mogą być również definiowane przez użytkownika w plikach .msg, .srv oraz .action.

Poniżej przedstawiono przykłady formatów wiadomości.

```
uint32 x
int32 y
```

Listing 1: Przykładowy plik wiadomości tematu .msg

```
int64 first
int64 second
---
int64 sum
```

Listing 2: Przykładowy plik wiadomości serwisu .srv

```
#goal
uint64 count
---
#result
uint64 result
---
#feedback
uint64 current_number
```

Listing 3: Przykładowy plik akcji .action

Jeśli chodzi o publikację wiadomości, to ROS charakteryzuje się dość krótkimi opóźnieniami. Oprogramowanie wykorzystujące ROS'a i jego pakiety może być napisane w kilku językach programowania, najpopularniejsze to C++ oraz Python.

3.2 Opis wymagań dla środowiska ROS

ROS wymaga systemu operacyjnego z rodziny Linux, a szczególnie pochodnych Debiana. Można go uruchomić na komputerach z procesorem x86, x64 lub ARM. Czyli można również go uruchomić na komputerze jednopłytkowym Rapsberry Pi, co idealnie nadaje się do zastosowania w małych robotach mobilnych.

3.3 ROS w porównaniu do systemu RTOS

Systemy RTOS[1] z reguły wymagają od swoich aplikacji, by uwzględniały reguły pracy w czasie rzeczywistym [27]. Systemy RTOS są zwykle niskopoziomowe tzw. *bare metal*. W tych systemach ramy czasowe wykonania danej operacji są ściśle określone, niedopuszczalne jest przekroczenie zadeklarowanej wielkości opóźnień.

ROS, jako że działa pod systemem linuksowym nie zapewnia wykonania operacji w czasie rzeczywistym, nawet nie gwarantuje wykonania żadnej operacji w zadanym czasie. A to dlatego, że systemy RTOS wykonują dane operacje bezpośrednio na procesorze, nie ma pomiędzy nimi systemu operacyjnego, który zarządza procesami, pamięcią, buforami danych. Mimo że opóźnienia pomiędzy wysłaniem a odebraniem wiadomości są bardzo małe, to jednak istnieją i należy to wziąć pod uwagę podczas implementacji oprogramowania.

Zastosowanie RTOS-a ma pewną znaczącą wadę. Oprogramowanie rozwiązania bazującego na RTOS wymaga bardzo dużo czasu i testowania oraz sprzętu, czyli generuje wysokie koszty.

[1] Real Time Operating System

4 Metody komunikacji międzyprocesowej

4.1 Wstęp

Komunikacja międzyprocesowa jest krytycznym składnikiem każdego systemu operacyjnego. Komunikacja między procesami może odbywać się w obrębie jednej maszyny lub pomiędzy wieloma maszynami, umożliwiając przesyłanie i synchronizację informacji [53].

Komunikacja między procesami jest niezbędna dla wydajnych i bezpiecznych systemów. Bez niej procesy nie byłyby w stanie komunikować się ze sobą i koordynować swoich działań, co prowadziłoby do spowolnienia operacji. Komunikacja międzyprocesowa powinna być wdrażana w sposób efektywny, aby zmaksymalizować jej korzyści i zminimalizować potencjalne zagrożenia z nią związane. Również pozwala ona na podział większego zadania między kilka procesów w celu jego szybszego wykonania.

Komunikowanie procesów może się odbywać za pomocą różnych mechanizmów, takich jak gniazda, pamięć współdzielona, potoki nazwane lub nienazwane czy protokół D-Bus. Jest ona stosowana jest na serwerach, systemach rozproszonych, systemach wbudowanych oraz sieciach przemysłowych.

4.2 Gniazda

Dzięki zastosowaniu protokołu TCP lub UDP istnieje możliwość komunikacji międzyprocesowej opartej na portach oraz adresach IP. Tutaj istnieje możliwość komunikowania się procesów pomiędzy różnymi maszynami połączonymi w jedną sieć. Ten rodzaj komunikacji jest dwukierunkowy i opiera się na architekturze klient – serwer [8].

Architektura klient – serwer polega na istnieniu dwóch procesów zwanych klientem oraz serwerem. Serwer oferuje pewne usługi lub zasoby. Klient wykonuje żądania pewnych usług lub zasobów. Zasoby należy rozumieć jako dane binarne lub tekstowe.

Gniazda można zdefiniować jako punkty końcowe połączenia sieciowego. Są identyfikowane jako kombinacja adresu IP oraz numer portu. Gniazdo może być połączone w danym czasie tylko i wyłącznie jednym z gniazdem [8]. Gniazda są w stanie wykonać poniższe podstawowe operacje [8]:

- utworzenie połączenia,
- wysłanie danych,
- odbiór danych,
- zamknięcie połączenia.

Na listingach 4 oraz 5 przedstawiono najprostsze przykładowe implementacje serwera oraz klienta korzystających z gniazd. Komunikacja odbywa się w obrębie jednej maszyny, zatem jako adres IP została przypisana wartość *localhost*. Klient wysyła krótką wiadomość, w odpowiedzi dostaje jej kopię i następnie zamyka połączenie z serwerem. Serwer nasłuchuje na odpowiednim porcie i czeka na połączenie. Po utworzeniu połączenia odbiera wiadomości wysłane przez klienta. Serwer odbiera dowolną wiadomość i zwraca ją w odpowiedzi do klienta.

```python
import socket

HOST = "localhost"
PORT = 5555

with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
    s.bind((HOST, PORT))
    s.listen()
    conn, addr = s.accept()
    with conn:
        while True:
            data = conn.recv(128) # 128 - rozmiar bufora danych
            if not data:
                break
            conn.sendall(data)
```

Listing 4: Przykładowa implementacja serwera za pomocą gniazd w języku Python

```python
import socket

HOST = "localhost"
PORT = 5555

with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
    s.connect((HOST, PORT))
    s.sendall(b"Message from robot")
    data = s.recv(128) # 128 - rozmiar bufora danych
```

Listing 5: Przykładowa implementacja klienta za pomocą gniazd w języku Python

4.3 Pamięć współdzielona

Model komunikacji między procesami na bazie pamięci współdzielonej polega na współdzieleniu pewnego obszaru pamięci operacyjnej przez kilka procesów [53]. Procesy mogą odczytywać lub zapisywać dane do tego obszaru pamięci. Synchronizacja w tym przypadku pomiędzy procesami odbywa się za pomocą semaforów [53]. Na listingach 6 oraz 7 zaprezentowano komunikację pomiędzy procesami za pomocą pamięci współdzielonej w języku C. Przykłady zostały napisane na podstawie dokumentacji systemu Linux [54]. Wykorzystano biblioteki *sys/shm.h* oraz *sys/ipc.h*, które dostarczają poniższe funkcje.

- Funkcja **ftok(const char *pathname, int proj_id)** przyjmuje ścieżkę do istniejącego pliku oraz 8 bitowe ziarno do generowania klucza. Zwraca unikalny klucz używany przez inne funkcje.
- Funkcja **shmget(key_t key, size_t size, int shmflg)** przyjmuje unikalny klucz wygenerowany przez **ftok()**, rozmiar obszaru pamięci wyrażony w bajtach oraz sposób dostępu do pamięci. Funkcja zwraca identyfikator pamięci współdzielonej.
- Funkcja **shmat(int shmid ,void *shmaddr ,int shmflg)** pozwala na podłączenie się procesu do pamięci współdzielonej. Zwraca wskaźnik na początek pamięci współdzielonej.
- Funkcja **shmdt(void *shmaddr)** pozwala na odłączenie się procesu od fragmentu pamięci współdzielonej.

```c
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>

int main()
{
  key_t unique_key = ftok("file",65);
  int shmid = shmget(unique_key,128,0666|IPC_CREAT);
  char *mem = (char*) shmat(shmid, NULL, 0);
  sprintf(mem, "Message from robot");
  shmdt(mem);
  return 0;
}
```

Listing 6: Przykładowa implementacja procesu zapisującego dane do pamięci współdzielonej w języku C

```c
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>

int main()
{
    key_t unique_key = ftok("file",65);
    int shmid = shmget(unique_key,128,0666|IPC_CREAT);
    char *mem = (char*) shmat(shmid, NULL, 0);
    printf("Data: %s",mem);
    shmdt(mem);
    return 0;
}
```

Listing 7: Przykładowa implementacja procesu odczytującego dane z pamięci współdzielonej w języku C

4.4 Potoki nazwane

Następnym typem komunikacji są potoki (ang. *pipes*). Ten rodzaj komunikacji jest szeregowy oraz jednokierunkowy [53]. Potoki one zwykle widoczne jako pliki. Mają one ograniczoną i z góry znaną pojemność, która pozwala na zapis danych bez ich utraty w określonych granicach. Potoki nazwane pozwalają na komunikację niespokrewnionych z sobą procesów [55]. Na listingach 8 oraz 9 podano przykład komunikacji między procesami, używając potoków nazwanych. By móc operować na tych potokach należy posłużyć się następującymi funkcjami:

- Funkcja **mkfifo(const char *pathname, mode_t mode)** tworzy potok nazwany na odpowiednim pliku. Drugi argument ustawia tryb dostępu.
- Funkcja **open(const char *pathname, int flags)** otwiera plik wraz z odpowiednimi uprawnieniami.
- Funkcja **read(int fd, void *buf, size_t count)** czyta zadaną ilość bajtów z dane potoku i zapisuje je do bufora.
- Funkcja **write(int fd, const void *buf, size_t count)** zapisuje zadaną ilość bajtów do potoku z danego bufora.

```c
#include <fcntl.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>

int main()
{
    char* file_name = "file";
    mkfifo(file_name, 0666);
    int fd = open(file_name, O_WRONLY);
    char* msg = "Message from robot";
    write(fd, msg, strlen(msg)+1);
    close(fd);
    return 0;
}
```

Listing 8: Przykładowa implementacja procesu zapisującego dane do potoku nazwanego w języku C

```c
#include <fcntl.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>

int main()
{
    char* file_name = "file";
    mkfifo(file_name, 0666);
    int buff_size = 80;
    char buffer[buff_size];
    int fd = open(file_name,O_RDONLY);
    while (1)
    {
        read(fd, buffer, buff_size);
        printf("Incoming msg: %s", buffer);
    }
    close(fd);
    return 0;
}
```

Listing 9: Przykładowa implementacja procesu odczytującego dane z potoku nazwanego w języku C

4.5 Semafory

Czasami istnieje potrzeba, by kilka procesów współdzieliło pewien zasób. By chronić ten krytyczny zasób wykorzystuje się semafory. Semafory mogą być przydatne przy synchronizacji procesów potomnych, które operują na wspólnym fragmencie kodu (sekcji krytycznej). Synchronizacja może być również wymagana przy wypisywaniu informacji na standardowe wyjście. Nie można pozwolić, by kilka procesów przekazywało informacje na standardowe wyjście w tym samym czasie. Semafory umożliwiają na korzystanie z danego fragmentu kodu ograniczonej liczbie procesów. Na listingu 10 pokazano przykład synchronizacji z procesem potomnym za pomocą semafora.

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <semaphore.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int main() {
    int semid, pid;
    key_t key;
    struct sembuf sem_lock = { 0, -1, SEM_UNDO };
    struct sembuf sem_unlock = { 0, 1, SEM_UNDO };

    // Tworzenie semafora
    key = ftok(".", 'S');
    semid = semget(key, 1, 0666 | IPC_CREAT);

    // Ustawianie warto Żci pocz tkowej semafora na 1
    semctl(semid, 0, SETVAL, 1);

    // Tworzenie nowego procesu
    pid = fork();

    semop(semid, &sem_lock, 1); // Blokowanie semafora
    printf("Bardzo dlugi tekst");
    semop(semid, &sem_unlock, 1); // Odblokowanie semafora
    return 0;
}
```

Listing 10: Przykładowa implementacja synchronizacji z procesem potomnym w języku C

4.6 Spotkania

Spotkania są metodami synchronizacji międzyprocesowej. Polegają na zawieszeniu jednego procesu w oczekiwaniu na rezultat od drugiego procesu. Pierwszy proces będzie zawieszony dopóki drugi proces nie wykona swego zadania. Po wykonaniu zadania przez drugi proces, oba wznawiają swoje działanie.

4.7 Przesył i kodowanie danych

Podczas przesyłania danych pomiędzy procesami często istnieje potrzeba przesłania całych obiektów, czy też kolekcji obiektów. Żeby takie obiekty nadawały się do przesyłu, należy je zakodować do strumienia bajtowego lub tekstowego. Proces odbierający te dane musi je zdekodować tzn. przekształcić strumień danych binarnych bądź tekstowych do obiektu, który może być umieszczony w pamięci operacyjnej. Przykładowo by w języku Python dokonać wspomnianych wcześniej przekształceń, to należy użyć pakietu *pickle*.

5 Mechanizmy synchronizacji międzywątkowej

5.1 Wstęp

Mówiąc o komunikacji międzyprocesowej w rozdziale 4 nie można nie wspomnieć o komunikacji operacji pomiędzy wątkami. Wątki pozwalają na wykonywanie kilku operacji jednocześnie w ramach jednego procesu. Przetwarzanie współbieżne pozwala na zwiększenie wydajności oprogramowania poprzez rozdzielenie złożonego obliczeniowo zadania na kilka wątków. Przykładowo pozwala to na odciążenie głównego wątku programu poprzez delegowanie operacji wejścia lub wyjścia do innych wątków. Często w taki sposób zaimplementowana jest komunikacja międzyprocesowa, za którą odpowiadają osobne wątki w ramach jednego procesu. W tym rozdziale opisano kilka nowoczesnych metod komunikacji i synchronizacji pomiędzy wątkami. Wszystkie przykłady kodów źródłowych zostały napisane w języku C++.

5.2 Wyścigi do danych

Jest to sytuacja, gdy następuje próba modyfikacji danych zaburzająca niezmiennik. Przykładem może być modyfikacja obiektu operującego na wskaźnikach w tym samym czasie przez dwa wątki, co może spowodować nieprawidłowe ustawienie się wskaźników i zaburzenie jego struktury.

5.3 Muteksy

Muteksy są obiektami, które służą do synchronizacji współbieżnego dostępu do danych z wielu wątków [9]. Służą one do ochrony tzw. sekcji krytycznej. Sekcję krytyczną definiuje się jako fragment kodu, w którym jednoczesny dostęp do danych z poziomu wielu wątków może spowodować niedeterministyczne zachowanie się programu. Obiekt muteksu jest współdzielony przez kilka wątków. Tylko jeden wątek może w tym samym czasie posiadać blokadę na tym muteksie. Na listingu 11 pokazano przykład użycia muteksów.

```
#include <chrono>
#include <thread>
#include <mutex>

int number = 0;
std::mutex mutex;

void function(int id) {
    mutex.lock();
    std::this_thread::sleep_for(std::chrono::seconds(5));
    ++number;
    mutex.unlock();
}

int main(){
    std::thread t1(function, 0);
    std::thread t2(function, 1);
    t1.join();
    t2.join();
}
```

Listing 11: Przykład użycia muteksów w języku C++

W operowaniu na muteksach często poawia się problem zakleszczenia. Zbiór wątków jest zakleszczony wtedy, gdy wszystkie wątki z tego zbioru oczekują na wznowienie działania jednego lub więcej wątków z tego zbioru.

By uchronić się przed zakleszczeniem należy stosować się do poniższych zaleceń [9]:

- Należy unikać zagnieżdżonych blokad. Należy unikać żądania blokady, jeżeli dany wątek dysponuje już jakąś blokadą.

- Należy stosować blokady w tej samej kolejności w tym samym wątku. Jeśli operacja wymaga zablokowania kilku muteksów, to należy zablokować je jednocześnie (np. za pomocą funkcji **std::lock**). W przypadku jakichkolwiek magazynów danych (list, tablic itd...) należy umożliwić operację na tych strukturach tylko jednemu wątkowi, co sprowadza się do tego, by używać dużo małych magazynów danych.

- Należy unikać wykonywania kodu użytkownika w czasie utrzymywania blokady. Nie wiadomo, czy kod użytkownika wywołuje jakieś blokady.

5.4 Semafory

Semafor również chroni kod sekcji krytycznej. Jednak w odróżnieniu od muteksu, pozwala on więcej niż jeden jednoczesny dostęp do danej sekcji. Obiekt semafora posiada w sobie wewnętrzny licznik, który jeśli dotrze do zera, to próba jego dekrementacji sprawi, że wątek zostaje zablokowany do czasu zwiększenia się wartości licznika. Na listingu 12 pokazano działanie semaforów.

```
#include <thread>
#include <chrono>
#include <semaphore>

std::counting_semaphore semaphore{2};
int data = 0;

void function()
{
    semaphore.acquire();
    ++data;
    semaphore.release();
}

int main()
{
    semaphore.acquire();
    semaphore.acquire();
    std::thread thread(function);
    std::this_thread::sleep_for(std::chrono::seconds(5));
    semaphore.release();
    semaphore.release();
    thread.join();
}
```

Listing 12: Przykład użycia semaforów w języku C++

5.5 Zmienne warunkowe

Czasami istnieje potrzeba, by jeden wątek mógł powiadomić inny o jakimś zdarzeniu. W tym celu są stosowane zmienne warunkowe. Zmienna warunkowa jest powiązana z pewnym zdarzeniem lub warunkiem oraz co najmniej jednym wątkiem, który czeka na spełnienie tego warunku. Wątek, który odkrywa, że warunek jest spełniony, może powiadomić pozostałe wątki oczekujące na tę zmienną warunkową, aby je obudzić i umożliwić im dalsze przetwarzanie. Zmienne warunkowe do działania wymagają muteksu. Przykład działania tego mechanizmu pokazano na listingu 13.

```
#include <string>
#include <thread>
#include <mutex>
#include <condition_variable>

std::mutex mutex;
std::condition_variable cv;
std::string data{"start"};
bool is_processed = false;

void function()
{
    std::this_thread::sleep_for(std::chrono::seconds(5));
    data =  "koniec";
    is_processed = true;
    lock.unlock();
    cv.notify_one();
}

int main()
{
    std::thread thread(function);
    data = "start";
    {
        std::unique_lock lock(mutex);
        cv.wait(lock, []{return is_processed;});
    }
    thread.join();
}
```

Listing 13: Przykład użycia zmiennych warunkowych w języku C++

5.6 Przyszłości i obietnice

Przyszłości są obiektami zwracanymi przez zadania, które są uruchomione w sposób asynchroniczny [9]. Przyszłości służą do komunikacji między wątkami. Główny wątek może okresowo sprawdzać, czy zadanie w tle się wykonało i jednocześnie wykonywać inne zadania. Obiekt przyszłości jest jedynym egzemplarzem odwołującym się do zdarzenia. Gdy wywołanie zadania zostanie uruchomione, to wspomniane wywołanie zwraca natychmiastowo obiekt przyszłości. Obiekt ten zawiera w sobie przyszły wynik wykonania pewnej funkcji. Gdy funkcja skończy swoje wykonywanie, to jej wynik będzie zapisany do obiektu przyszłości i będzie gotowy do odbioru. Zwracanie wartości przez zadania wykonywane w tle. Na listingu 14 pokazano użycie przyszłości. Obietnice są związane z przyszłościami. Pozwalają one na ustawienie danych w przyszłości w osobnym wątku. Wątek oczekujący może wstrzymać działanie w oczekiwaniu na przyszłość, natomiast wątek udostępniający dane może użyć obiektu obietnicy do ustawienia powiązanej wartości, tak aby odpowiednia przyszłość przeszła w stan gotowości.

```
#include <thread>
#include <future>
#include <chrono>
#include <iostream>

void function(std::promise<int> promise)
{
    std::this_thread::sleep_for(std::chrono::seconds(15));
    promise.set_value(555);// Powiadom watek glowny o
                           //gotowosci i ustaw wartosc
}

int main()
{
    std::promise<int> promise;
    std::future<int> future = accumulate_promise.get_future();
    std::thread thread(function, std::move(accumulate_promise));
    std::this_thread::sleep_for(std::chrono::seconds(5));
    auto status = future.wait_for(2s); // poczekaj przez 2 sekundy i sprawdz co
    sie dzieje
    std::cout << status + '\n';
    int result = future.get(); //poczekaj do konca na wynik i go zwroc
    thread.join();
}
```

Listing 14: Przykład użycia przyszłości w języku C++

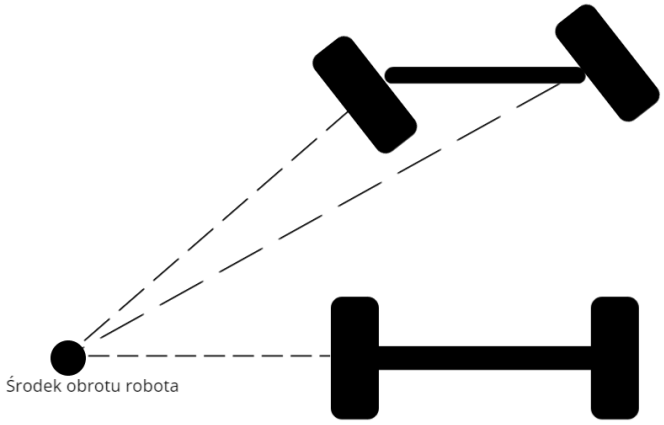
6 Napędy w robotach mobilnych

6.1 Wstęp

Wybór napędu do robota mobilnego jest kwestią kluczową. Istnieją różnego rodzaje napędów, które różnią się skomplikowaniem w budowie oraz oferowanymi możliwościami. W tym rozdziale został dokonany przegląd wybranych napędów.

6.2 Napęd Ackermana

Sterowanie orientacją [31] odbywa się za pomocą zmiany orientacji dwóch przednich kół. Napęd Ackermana sprawdza się przy pokonywaniu trajektorii o torze w kształcie krzywej, gdyż skręt może odbyć się w trakcie jazdy bez zmiany prędkości kątowych żadnego z kół. Z wykorzystaniem tego napędu nie da się zmienić orientacji robota w miejscu, gdyż dwa pozostałe koła nie są przystosowane do tego, by móc wykonać skręt. W napędzie Ackermana osie obrotu kół muszą się przecinać w jednym punkcie. Zastosowanie tego rodzaju napędu odbiera możliwość precyzyjnego obrotu robota. Kierunek jazdy robota jest określony pośrodku tylnej osi. Robot wykorzystujący napęd Ackermana podczas skrętu porusza się po łuku pewnego okręgu. Promień tego okręgu nazywa się promieniem skrętu.

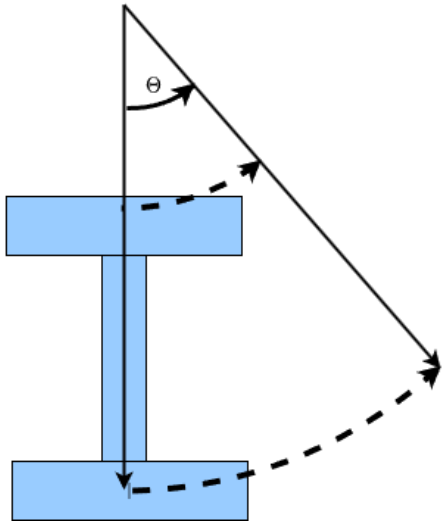


Rysunek 2: Schemat działania napędu Ackermana

6.3 Napęd różnicowy

Ten rodzaj sterowania [16] jest najczęściej spotykanym w robotyce, jest to spowodowane jego prostotą konstrukcji. Dzięki zastosowaniu tego rodzaju napędu robot uzyskuje pełną zwrotność. Sterowanie różnicowe zakłada to, że koła robota są sterowane niezależnie po obu przeciwnych stronach jego platformy mobilnej. Tutaj skręt może się odbyć tylko i wyłącznie za pomocą zmiany prędkości kątowej jednego z kół lub zmiany prędkości obrotowych par kół z lewej, lub prawej strony. Ten napęd pozwala na obrót w miejscu.

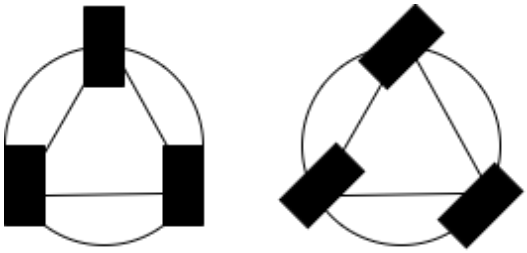
Zakładając, że jest się w posiadaniu robota wyposażonego w dwa osobne napędzane koła, oraz jedno koło samonastawne, to jazdę w przód lub w tył można wymusić poprzez obrót obu kół w tym samym kierunku. Obrót robota w miejscu wykonuje się poprzez obrót kół w przeciwnych kierunkach z tym samym modułem prędkości. Obrót robota z przemieszczeniem (jazda po łuku) wykonuje się poprzez obrót kół w przeciwnych kierunkach z różnym modułem prędkości.



Rysunek 3: Schemat działania napędu różnicowego

6.4 Napęd synchroniczny

Na rynku jest dostępny jeszcze napęd zwany synchronicznym. Jest on wyposażony w 3 lub 4 koła, które są napędzane przez jeden silnik [47], gdzie drugi silnik służy do zmiany orientacji kół wokół osi pionowej. Jako że wszystkie koła są obracane przez jeden silnik, to ich obrót zawsze będzie jednakowy dla wszystkich kół. By robot mógł się poruszać prostoliniowo, to wspomniane koła muszą być zawsze równolegle ustawione względem siebie nawzajem. Wadą tego napędu to jest to, że nadaje się on tylko do wykorzystania w pomieszczeniach.



Rysunek 4: Schemat działania napędu synchronicznego

6.5 Przegląd implementacji napędów robota

ROS zapewnia szereg sterowników napędów, które przejmują sterowanie napędami robota [17] i symulują zachowanie prawdziwego napędu robota. Symulowane napędy wyprowadzają interfejs prędkościowy, który pozwala nam sterować prędkością kątową oraz liniową. Pakiet, który zawiera w sobie różne sterowniki, nazywa się **ros_controllers**.

Wszystkie te symulatory wyprowadzają interfejs sterujący w postaci dwóch wektorów. Pierwszy z nich odpowiada za wymuszenie prędkościowe liniowe robota (tzn. wymuszenie wzdłuż osi x jego własnego układu współrzędnych). Drugi z nich odpowiada za wymuszenie prędkościowe kątowe robota (tzn. wymuszenie wzdłuż osi pionowej własnego układu współrzędnych).

```
geometry_msgs/Vector3 linear
geometry_msgs/Vector3 angular
```

Listing 15: Format wiadomości wymuszenia prędkościowego

```
#geometry_msgs/Vector3
float64 x
float64 y
float64 z
```

Listing 16: Format wiadomości definiującej wektor

Wszystkie sterowniki posiadają kilka wspólnych konfigurowalnych parametrów. Są to między innymi:

- Parametr **publish_rate** pozwala ustawić częstotliwość publikacji danych z odometrii.
- Paramtetr **linear/x/max_velocity** pozwala ustawić maksymalną prędkość liniową robota.
- Parametr **angular/z/max_velocity** pozwala ustawić maksymalną prędkość kątową robota wokół jego własnej osi obrotu.
- Parametr **wheel_separation** oznacza rozstaw kół wzdłuż osi x.
- Parametr **wheel_radius** oznacza promień koła.
- Parametr **publish_cmd** oznacza nazwę kanału do publikowania wymuszenia prędkościowego.

Istnieją trzy szczególne symulatory napędów robota.

- **ackermann_steering_controller**,
- **diff_drive_controller**,
- **skid_steer_drive_controller**

Symulator **ackermann_steering_controller** symuluje napęd Ackermana [18]. Komenda prędkości jest rozdzielona. Wymuszenia liniowe oraz kątowe są generowane przez osobne pary kół.

Symulator **diff_drive_controller** odpowiada za symulację napędu różnicowego [16]. Wymuszenia liniowe, jak i kątowe jest przeliczane na prędkość obrotową kół robota.

Symulator **skid_steer_drive_controller** jest ulepszoną wersją symulatora z napędem różnicowym [20]. Symuluje on dodatkowo poślizgi, co poprawia stopień odwzorowania rzeczywistości w symulacji. Częstotliwość występowania poślizgów jest konfigurowana we właściwościach sterownika.

7 Symulator Gazebo

7.1 Opis symulatora Gazebo

Gazebo jest symulatorem służącym do symulowania robotów oraz ich otoczenia [29]. Można w nim zaprojektować model robota oraz otaczające go środowisko wraz z prawami fizyki, rządzącymi zasymulowanymi obiektami. Pozwala on na testowanie zaprojektowanych przez inżynierów algorytmów, nie wykorzystując przy tym często kosztownego sprzętu.

Symulator Gazebo posiada pełną integrację z ROS-em. Wszystkie składowe elementy symulatora, jego stan oraz procesy zachodzące w nim są publikowane na odpowiednie tematy. Na symulator również można oddziaływać z zewnątrz za pomocą serwisów oraz tematów.

Dzięki niemu można przykładowo uniknąć uszkodzeń sprzętu w przypadku błędu implementacji algorytmu. Przyspiesza on również testowanie rozwiązań, gdyż ponowne uruchomienie sekwencji odbywa się bez resetowania ustawień sprzętu lub bez jego przenoszenia do pozycji początkowej. Zapewnia on również możliwość symulacji różnych czujników z możliwością wprowadzenia do nich dowolnego modelu zakłóceń.

7.2 Sposób opisu modelu robota za pomocą URDF

Model robota opisuje się w formacie URDF (ang. *Unified Robot Description Format*), by potem ten opis w postaci pliku XML został przetworzony przez parser[2] URDF na model robota [30]. Potem model ten może być osadzony w symulatorze Gazebo, mając możliwość interakcji z środowiskiem. Możliwość konstrukcji modelu robota jest możliwa dzięki pakietowi **urdf**.

```
<robot name="cube">
    <link name="base_link">
        <visual>
            <geometry>
                <box size="1 2 3"/>
            </geometry>
        </visual>
        <collision>
            <geometry>
                <box size="1 2 3"/>
            </geometry>
        </collision>
    </link>
</robot>
```

Listing 17: Przykładowy plik URDF z definicją prostego elementu

[2]Algorytm najpierw wykonujący analizę składniową danych w celu określenia ich zgodności z określonym językiem, a następnie przetwarzający te dane na obiekty w zaalokowane pamięci

7.3 Język makr XACRO

Istnieje również język makr zgodny z URDF zwany XACRO [32]. Język ten pozwala na definiowanie generycznych elementów oraz ich właściwości pozwalając później na ich konkretyzację w celu stworzenia docelowych elementów robota. Ten język makr służy do przyspieszenia definiowania modelu robota.

Na listingu 18 podano przykład wzorca prostopadłościanu opisanego za pomocą XACRO. Zostały stworzone trzy konkretne modele tego wzorca.

```
<xacro:macro name="cube" params="x y z">
   <link name="base_link">
       <visual>
           <geometry>
               <box size="${x} ${y} ${z}"/>
           </geometry>
       </visual>
       <collision>
           <geometry>
               <box size="${x} ${y} ${z}"/>
           </geometry>
       </collision>
   </link>
/xacro:macro>

xacro:cube x="1" y="1" z="1"/>
xacro:cube x="2" y="4" z="0.4"/>
xacro:cube x="1" y="5" z="6"/>
```

Listing 18: Przykładowy plik XACRO z definicją szablonu prostego elementu

7.4 Elementy dostępne przy budowie modelu robota

URDF zawiera swoją specyfikację XML [33], czyli zbiór podstawowych obiektów, z których można zbudować model robota.

Niezbędne elementy, które pozwolą nam opisać model robota są następujące.

- Element typu **robot** określa korpus całego robota.
- Element typu **link** określa element robota.
- Element typu **joint** określa więzy kinematyczne.
- Element typu **xacro:macro** określa szablon elementu robota.
- Element typu **gazebo-plugin** dodaje do modelu funkcjonalność dodatku oferowanego przez Gazebo.

8 Czujniki stosowane w robotyce

8.1 Wstęp

Czujniki są bardzo ważne dla robotów, ponieważ pozwalają im na orientację w danym środowisku. Czujniki umożliwiają robotom pobieranie danych z otoczenia, takich jak odległość od innych obiektów, położenie czy też informacje o pogodzie. Na podstawie tych informacji roboty są w stanie podejmować decyzje i wykonywać odpowiednie działania.

8.2 IMU

IMU (ang. *Inertial Measurement Unit*) jest urządzeniem, które składa się z żyroskopu oraz akcelerometru [37]. Służy ono do nawigacji bezwładnościowej (inercyjnej). W celu określenia orientacji obiektu mierzy się jego prędkość kątową i działające na niego przyspieszenia. [5]

8.2.1 Akcelerometr

Akcelerometr służy do mierzenia przyspieszeń liniowych [11]. Pozwala to (za pomocą całkowania) na obliczenie wektora prędkości obiektu. Ma on możliwość dokonania pomiaru w dwóch lub trzech kierunkach. Wielokierunkowe akcelerometry składają się z kilku prostych akcelerometrów, które mierzą przyspieszenie w jednym kierunku. Dzięki uzyskanym wskazaniom możliwe jest uzyskanie kąta odchylenia akcelerometru od pionu. Zwykle akcelerometry wskazują przyspieszenie w jednostkach, które są iloczynem przyspieszenia grawitacyjnego, np. $2g$. Akcelerometry mierzą absolutne przyspieszenie, oznacza to, że jeśli na akcelerometr nie działa żaden wektor przyspieszenia liniowego skierowanego w kierunku przyspieszenia ziemskiego, to wskaże on przyspieszenie równe $1g$.

8.2.2 Żyroskop

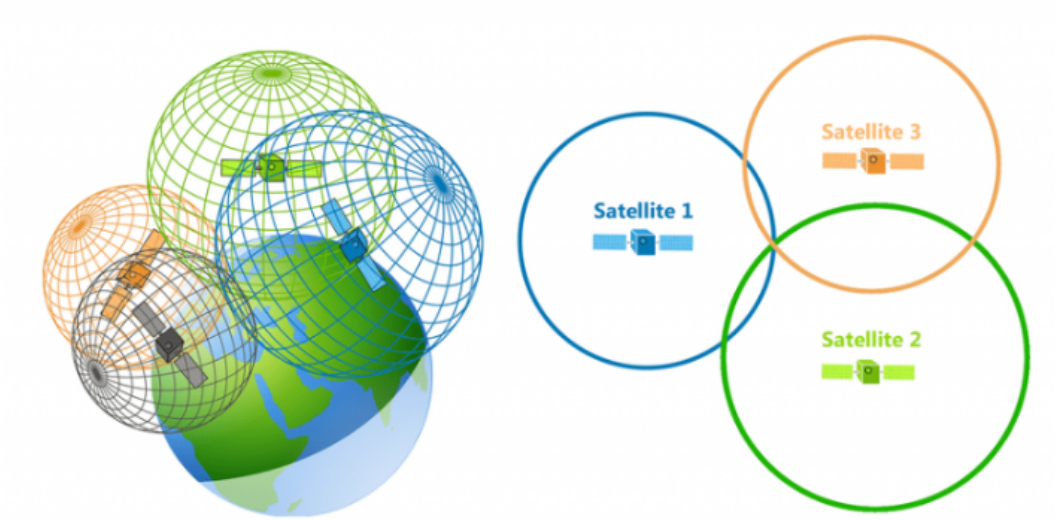
Żyroskopy są urządzeniami służącymi do pomiaru bieżącej prędkości kątowej danego obiektu [10]. Żyroskopy mogą być użyte jako urządzenia samodzielne lub wchodzić w skład bardziej skomplikowanych urządzeń, takich jak np. IMU lub AHRS [41]. Na rynku dostępnych jest kilka implementacji żyroskopów, między innymi takie jak żyroskopy mechaniczne, żyroskopy optyczne żyroskopy RLG [6] oraz MEMS [10]. W układach elektronicznych stosuje się najczęściej żyroskopy typu MEMS, które działają na zasadzie detekcji prędkości kątowej za pomocą elementu mechanicznego doznającego wibracji. Takie żyroskopy nie muszą być wyposażone w elementy wymagające łożyskowania. Ta właściwość pozwala na miniaturyzację takiego żyroskopu i zastosowanie go w układach elektronicznych. Żyroskop pozwala na zmierzenie prędkości kątowej obiektu względem jego dowolnej osi. Pozwala on również na obliczenie względnej orientacji (względem początkowego położenia) danego obiektu. Operacja określania względnej orientacji również odbywa się za pomocą całkowania.

8.3 Odbiornik GPS

Odbiorniki systemu GPS znalazły szerokie zastosowanie w sektorach militarnym, nawigacyjnym oraz automotive. Odbiornik współpracuje z globalnym systemem lokalizacji satelitarnej [4]. Nawigacji GPS używa się również na otwartych przestrzeniach oraz w miastach. Lokalizacji satelitarnej można użyć do lokalizacji obiektów, nawigacji, mierzenia prędkości oraz mierzenia czasu

[43]. Nawigacja satelitarna działa na zasadzie współpracy trzech elementów: satelitów, stacji kontrolnych oraz odbiorników GPS. Satelity obecne na orbicie wokół Ziemi wysyłają sygnał radiowy w sposób ciągły w jej kierunku. Sygnały te przebywają całą swoją trasę z prędkością bliską światła [43]. Sygnały są generowane przez układy nadawcze satelitów. [43]. Odbiornik odbiera nadchodzący sygnał radiowy z kilku widocznych satelitów i oblicza odległość od każdego z nich z osobna. Odległość odbiornika od satelity wyraża się poprzez iloczyn prędkości fali radiowej oraz różnicy czasu między wysłaniem fali radiowej przez satelitę a odebraniem jej przez odbiornik.

Mając obliczone odległości od poszczególnych satelitów to dzięki trilateracji (metody określania tego, jak daleko dany punkt znajduje się od innych trzech punktów) można obliczyć szerokość geograficzną, długość geograficzną, wysokość oraz czas [44]. Trilateracja w procesowaniu GPS oznacza wyznaczenie przecięcia się jak największej ilości sfer (sfery są zbudowane z promieni, które są odległościami między odbiornikiem GPS a poszczególnymi satelitami) w pobliżu jednego punktu i na tej podstawie estymację położenia odbiornika i tym samym obiektu.



Rysunek 5: Wizualiacja algorytmu trilateracji [49]

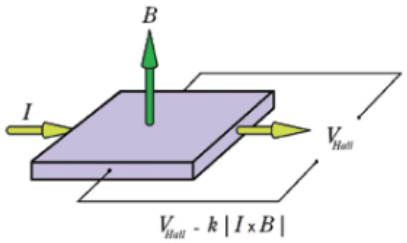
Odbiorniki GPS podają swoją lokalizację w formacie NMEA [42]. Protokół NMEA jest ogólnym protokołem stosowanym w nawigacji satelitarnej oraz w komunikacji między elektronicznym osprzętem obecnym na statkach. Protokół ten definiuje kilka komend, lecz najważniejszą jest $GPGGA, która zawiera w sobie pozycję geograficzną obiektu. Przykład takiej wiadomości podano poniżej:

```
$GPGGA,055415.000,3804.8000,S,17617.4000,E,2,06,1.8,69.5,M,26.2,M,0.8,0000*54
```

Szerokość geograficzna jest podana na drugim miejscu, a długość geograficzną na trzecim miejscu. Po odpowiednich konwersjach otrzymanej wiadomości istnieje możliwość poznania lokalizacji odbiornika wyrażonej we współrzędnych geograficznych. Minimalna liczba satelitów pozwalająca na określenie pozycji geograficznej wynosi 4 satelity. Większa liczba satelitów zwiększa precyzję oraz zmniejsza tendencję do nagłego przeskakiwania pozycji na mapie. Dokładność określenia położenia zależy od liczby widocznych satelitów.

8.4 Magnetometr

Jest to czujnik służący do pomiaru wektora (moduł i kierunek) pola magnetycznego [15]. Magnetometry znajdują zastosowanie w takich sektorach jak wojskowość oraz nawigacja. Są też osprzętem często obecnym na telefonach oraz wszystkich urządzeniach, które muszą być mobilne np. drony. Zależnie od rodzaju magnetometr może być dwuosiowy lub trójosiowy. Na rynku dostępnych jest kilka różnych implementacji magnetometrów, są to między innymi magnetometry korzystające z efektu Halla, magnetometry GMR, magnetometry MTJ oraz AMR [45]. Wybór danego rodzaju magnetometru podyktowany jest zwykle jego ceną oraz jakością wskazań. Najpopularniejszym magnetometrem jest ten, który wykorzystuje efekt Halla. Gdy obiekt jest położony w silnym polu magnetycznym prostopadłym do jego płaszczyzny 6, wtedy na obwodzie przyłączonym do tego obiektu wykrywane jest napięcie, zwane napięciem Halla. Kompas skonstruowany na bazie magnetometru pozwala w prosty sposób określić bezwzględną orientację obiektu względem północy.



Rysunek 6: Wizualizacja działania magnetometru opartego o czujnik Halla [50]

8.5 Enkoder

Enkoder jest czujnikiem, który jest zamontowany na wale obrotowym koła lub innego mechanizmu. Za pomocą enkodera istnieje możliwość zmierzenia liczby impulsów w danym czasie, co pozwala obliczyć przebyty dystans w zadanym czasie. Enkodery inkrementalne [51] generują serię impulsów od momentu włączenia zasilania. Zwykle wykorzystują tarczę osadzoną na wale, w której wykonane są otwory. Przetwarzają one obrót tarczy na dwukanałowy sygnał kwadraturowy [51].

9 Zakłócenia przy współpracy z czujnikami

9.1 Ogólnie o zakłóceniach

W każdym urządzeniu pomiarowym lub czujniku występują zakłócenia, które zniekształcają otrzymany sygnał. Zakłócenia te wprowadzają szum w nasz pomiar i przez to jego wartość może on przyjąć wartość całkowicie odmienną od oczekiwanej. Pomiary obarczone zakłóceniami można wyrazić za pomocą wzoru 1.

$$y(t) = x(t) + \epsilon(t) \tag{1}$$

gdzie,
$y(t)$ – otrzymany pomiar,
$x(t)$ – rzeczywista wartość,
$\epsilon(t)$ – wartość szumu

Istnieją metody do eliminacji szumów. Szumy można zlikwidować za pomocą filtrów, które są opisane w rozdziale 9.7.

9.2 Zakłócenia IMU

Ogólnie zakłócenia IMU można podzielić na dwie kategorie [1]:

- stochastyczne,
- deterministyczne.

Deterministyczne są spowodowane głównie wadami produkcyjnymi tych urządzeń. Błędy deterministyczne są zwykle stałe podczas całego czasu pracy urządzenia. Błędy stochastyczne są błędami o naturze losowej. Można je reprezentować jako zmienne losowe o pewnym rozkładzie losowym. Elementy mechaniczne oraz elektryczne w IMU zużywają się również z czasem. Zużycie powoduje narastanie błędów.

9.3 Zakłócenia magnetometru

Magnetometry są głównie osadzane w układach scalonych. Głównym źródłem zakłóceń są pola magnetyczne od cewek silników napędowych, które zakłócają odczyt pola magnetycznego Ziemi. Kolejnym źródłem zakłóceń są szumy wywołane przez prąd przepływający przez obwód [2]. Prąd ten generuje pole magnetyczne wokół przewodów. Dodatkowo materiały ferromagnetyczne (np. w szkielecie robota będącymi stopami żelaza) wprowadzają stały błąd pomiaru.

9.4 Zakłócenia czujników

Również sam pomiar jest obarczony błędem. Czujnik po zebraniu sygnału analogowego przetwarza go na sygnał cyfrowy. Takie przetworzenie, czyli kwantyzacja zmienia dane pomiarowe (dokładniej mówiąc, klasyfikuje je do najbliższej pasującej wartości, która może być reprezentowana przez liczbę z odpowiednią rozdzielczością).

9.5 Precyzja systemu GPS oraz opis zakłóceń

System GPS jest złożony, co oznacza, że w porównaniu do wymienionych już czujników posiada on więcej źródeł błędów i zakłóceń. Jakość odczytów i intensywność pojawiania się błędów jest zależna od:

- liczby widocznych satelitów,
- wpływu tzw. rozmycia pozycji (ang. *Dilution of Position*),
- błędów sygnału GPS.

Precyzja wskazań GPS znacząco różni się w zależności od liczby widocznych satelitów [43]. Im więcej widocznych satelitów, tym lepsze przybliżenie aktualnej pozycji odbiornika. Rozmycie precyzji jest miarą otrzymania rozstawienia satelitów zmniejszającego liczbę błędów wynikających z aktualnej konfiguracji widocznych satelitów na niebie. Błędy systemu GPS mogą wynikać z błędów zegara odbiornika, które są spowodowane jego małą rozdzielczością w porównaniu do zegarów atomowych. Tutaj błąd ma raczej naturę wynikającą z kwantyzacji pomiarów. Kolejnym czynnikiem wprowadzającym błędy są szumy pochodzące z fal radiowych o podobnych częstotliwościach. Tę samą klasę błędu można uzyskać z powodu odbijania się fal radiowych o tej samej częstotliwości od przeszkód blisko odbiornika. Zakłócenia mogą być również generowane poprzez celowe zakłócanie sygnału GPS.

9.6 Zakłócenia odometrii

Odometria charakteryzuje się obecnością błędów systematycznych oraz niesystematycznych. Systematyczne błędy są jednym z największych wyzwań dla precyzyjnej kontroli pozycji. Różne źródła mogą przyczyniać się do ich powstawania, takie jak nierówność średnic kół. Również rozstaw kół, który jest inny od nominalnego oraz niewspółosiowość kół może być przyczyną błędów systematycznych. Dodatkowo, skończona rozdzielczość enkodera również może przyczynić się do powstawania tych błędów. Błędy niesystematyczne mają charakter losowy i powstają na wskutek poślizgów kół oraz nierówności podłoża.. Obie grupy źródeł zakłóceń sprawiają, że pomiary odometrii należy korygować.

9.7 Model zakłóceń w Gazebo

By najwierniej oddać rzeczywistość, Gazebo wprowadza do każdego symulowanego czujnika szum [13]. Parametry tego szumu da się ustawić w pliku konfiguracyjnym. Zwykle typ tego szumu jest szumem o rozkładzie normalnym. Ten sposób symulowania szumów oddaje w wiarygodny sposób prawdziwe zakłócenia. Gęstość prawdopodobieństwa zmiennej z jest dana wzorem 2.

$$\rho_G(z) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(z-\mu)^2}{2\sigma^2}} \tag{2}$$

gdzie,
z – zmienna losowa,
μ – średnia arytmetyczna,
σ – odchylenie standardowe

W Gazebo istnieją dwa rodzaje konfiguracji szumów w pliku konfiguracyjnym [20]:

- W pliku konfiguracyjnym można podać wartość odchylenia standardowego σ oraz wartość średnią μ.
- W pliku konfiguracyjnym można podać tylko wartość odchylenia standardowego σ. Przykładem tutaj jest IMU, które w pliku konfiguracyjnym przyjmuje tylko wartość odchylenia standardowego [52].

10 Filtry

10.1 Opis działania filtrów

Podczas odbierania sygnałów z czujników, zwykle towarzyszy im zniekształcenie spowodowane przez szum [35]. Sygnał nadający się do użycia można otrzymać wtedy, gdy szum zostanie zredukowany. Szum jest często zmienny w czasie. Filtry są algorytmami lub urządzeniami, które wygładzają surowy sygnał. Nie usuwają one szumu, lecz zmniejszają jego wpływ.

10.2 Filtr średniej ruchomej

Jest to filtr FIR mający wszystkie współczynniki równe sobie [35]. Właściwość ta zaprezentowana jest we wzorze 3, gdzie L jest długością filtra, czyli szerokość okna pomiarowego.

$$h_i = \frac{1}{L} \tag{3}$$

Długość filtra L jest liczbą próbek zebranych wstecz od ostatniego pomiaru (n najnowszych próbek). Filtr średniej ruchomej jest podany wzorem 4, gdzie $x(i)$ są kolejnymi ostatnimi pomiarami, a $y(n)$ jest otrzymaną przefiltrowaną wartością.

$$y(n) = \frac{1}{L}(x(n) + x(n-1) + ... + x(n-L+1)) \tag{4}$$

10.3 Jednowymiarowy filtr Kalmana

Filtr Kalmana jest algorytmem szacującym, który jest stosowany do przetwarzania danych z czujników, które generują dane z błędem. Głównym celem filtru Kalmana jest obliczenie najlepszej estymacji dla stanu systemu, na podstawie dostępnych danych pomiarowych i modelu matematycznego opisującego system [38]. Filtr Kalmana ma szerokie zastosowanie w wielu dziedzinach, ponieważ jest on skuteczny w radzeniu sobie z problemami związanymi z brakiem danych, zakłóceniami i szumami. Jest szeroko stosowany w robotyce do bezwładnościowego pozycjonowania robota, śledzenia obiektów i nawigacji. Jest on także elastyczny i może być stosowany do różnych rodzajów problemów, takich jak filtracja danych ciągłych lub dyskretnych, z modelami linearnymi lub nieliniowymi. Jednowymiarowy filtr Kalmana [38] jest filtrem wykorzystującym w swoich obliczeniach:

- stan systemu oraz jego niepewność,
- przychodzące pomiary oraz ich niepewności,
- estymację nowego stanu oraz niepewność tej estymacji.

Poniżej przedstawiono algebraiczne równania filtru Kalmana dla układów liniowych. Filtr ten składa się z pięciu równań. Ta forma filtru ma zastosowanie do obliczeń, gdzie estymacji poddawana jest wartość skalarna.

$$\hat{x}_{n,n} = \hat{x}_{n,n-1} + K_n(z_N - \hat{x}_{n,n-1}) \tag{5}$$

$$\hat{x}_{n+1,n} = \hat{x}_{n,n} + \Delta t\hat{\dot{x}}_{n,n} \tag{6}$$

$$K_n = \frac{p_{n,n-1}}{p_{n,n-1} + r_n} \tag{7}$$

$$p_{n,n} = (1 - K_n)p_{n,n-1} \tag{8}$$

$$p_{n+1,n} = p_{n,n} + q_n \tag{9}$$

gdzie,
$\hat{x}_{n,n-1}$ – poprzednia estymacja stanu,
$\hat{x}_{n,n}$ – estymacja aktualnego stanu,
$\hat{x}_{n+1,n}$ – prognoza estymacji stanu(lub po prostu wstępna estymacja stanu),
$p_{n,n-1}$ – niepewność poprzedniej estymacji stanu,
$p_{n,n}$ – aktualna niepewność estymacji stanu,
$p_{n+1,n}$ – prognoza niepewności estymacji stanu,
$\hat{\dot{x}}_{n,n}$ – pochodna estymacji aktualnego stanu,
K_n – wzmocnienie Kalmana,
z_N – pomiar stanu,
Δt – różnica czasu pomiędzy przeskokami okna pomiarowego,
q_n – szum procesowy,
r_n – niepewność pomiaru

Poniżej przedstawiono nazwy równań filtru Kalmana w odpowiadającej im kolejności.

- Wzór 5 zwany jest równaniem aktualizacji.
- Wzór 6 nazywa się równaniem prognozy stanu.
- Wzór 7 nazywa się estymacją wzmocnienia Kalmana.
- Wzór 8 nazywa się równaniem aktualizacji niepewności estymacji stanu.
- Wzór 9 nazywa się prognozą niepewności estymacji stanu.

Podczas obliczeń wykorzystuje się niepewność estymacji p_n oraz niepewność pomiaru r_n. Niepewność estymacji zmienia się wraz z każdą iteracją filtru, a niepewność pomiaru jest z góry zadana i stała przez cały czas (np. niepewność pomiaru woltomierza). Wielkość $z_N - \hat{x}_{n,n-1}$ jest czasami zwana innowacją. Wzmocnienie Kalmana jest najważniejszym elementem i należy je należy rozumieć jako:

$$K_n = \frac{Niepewnosc\, estymacji}{Niepewnosc\, estymacji + niepewnosc\, pomiaru} \tag{10}$$

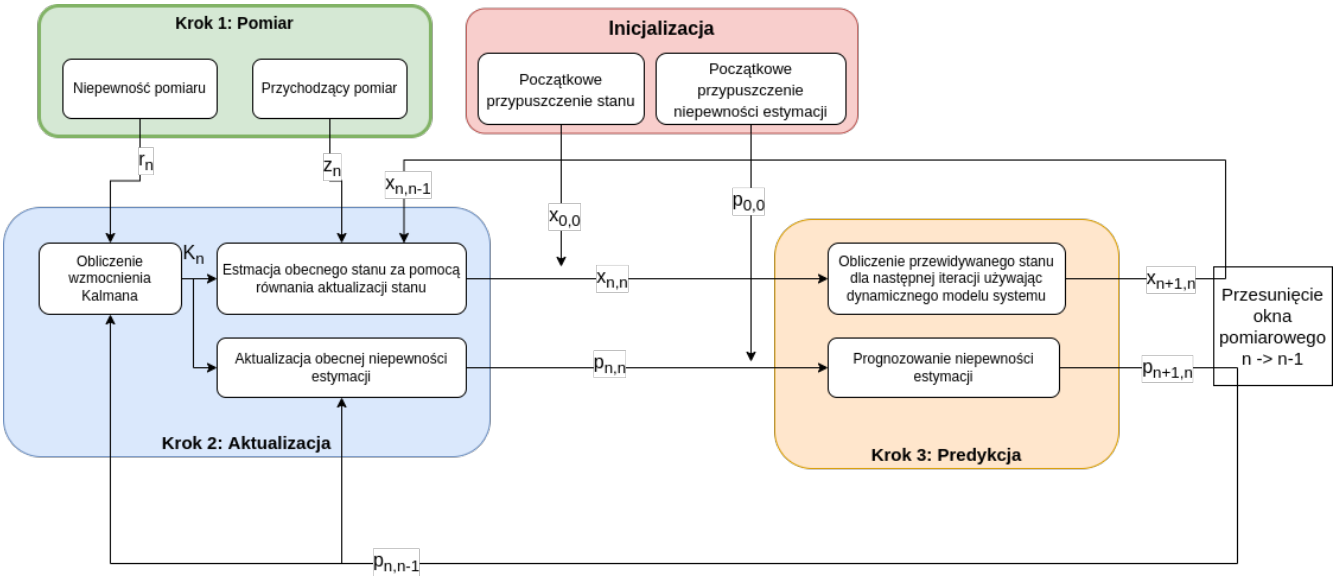
Wartość wzmocnienia Kalmana zawiera się w przedziale $0 < K_n < 1$. Jeśli jego wartości są bliskie zeru, to zbieżność niepewności estymacji do zera jest bardzo powolna. Jednak jeśli wzmocnienie Kalmana jest bliskie 1, to niepewność estymacji szybko zbiega do 0.

Losowy szum procesowy q_n jest wartością fluktuacji wartości prawdziwej (przykładowo prawdziwa wartość napięcia czasami może być równa 15V lub 14.998V). Szum procesowy wzmaga błędy estymacji. Tę wartość zwykle zakłada się odgórnie i jest stała podczas obliczeń. Tę wartość należy dobrać eksperymentalnie, gdyż należy ją dopasować do fluktuacji systemu. Nieodpowiednie jej dopasowanie prowadzi do zaniku skuteczności filtracji lub nienadążania wyjścia filtru za sygnałem oryginalnym.

Działanie filtru opiera się na czterech fazach:

- inicjalizacji,
- pomiaru,
- aktualizacji stanu,
- predykcji stanu.

Podczas inicjalizacji zakładane są odgórnie estymacja stanu początkowego oraz niepewność tej estymacji. Dowolność doboru tych wartości jest duża, gdyż filtr Kalmana i tak całkiem sprawnie zbiegnie do takiej estymacji, gdzie wartości estymowane będą bliskie wartościom rzeczywistym. Etap inicjalizacji odbywa się tylko raz (np. przy włączeniu urządzenia). Etap pomiaru nie jest tak naprawdę właściwym etapem tego filtru, lecz tylko obsługą czujników i odebrania z nich danych. Etap aktualizacji polega na obliczeniu wzmocnienia Kalmana K_n, estymacji aktualnego stanu $\hat{x}_{n,n}$ oraz obliczeniu niepewności tej estymacji $p_{n,n}$. Etap predykcji polega obliczeniu prognoz estymacji następnego stanu $\hat{x}_{n+1,n}$ oraz niepewności tej estymacji $p_{n+1,n}$. Na rysunku 7 został zaprezentowany algorytm filtru Kalmana w formie graficznej wraz z podanymi wejściami i wyjściami do i z filtru.



Rysunek 7: Schemat algorytmu filtru Kalmana

10.4 Fuzja wskazań z czujników

Fuzja wskazań (ang. *sensor fusion*) z czujników polega na łączeniu informacji pochodzących z różnych czujników, tak aby uzyskać dokładniejsze i kompletne dane *sensor fusion*. Jeśli istnieje możliwość skorzystania z kilku czujników, które mierzą tę samą wielkość, to można wykorzystać ich pomiary w celu otrzymania wartości bardziej zbliżonej do wartości rzeczywistej. Jest to sposób filtracji danych, który wykorzystuje inne filtry (np. filtr Kalmana). Można do tego wykorzystać jednowymiarowy, wielowymiarowy filtr Kalmana lub Rozszerzony Filtr Kalmana (EKF) [7]. Wielowymiarowy filtr Kalmana opisano w rozdziale 10.5

Taki sposób filtracji ma taką zaletę, że pozwala to na łagodzenie błędów wynikających z działania poszczególnych czujników [7], a także zakładając również, że zbierane są dane z czujników A i B. Zakładając również, że te czujniki są czujnikami działającymi na innych zasadach. Zastosowanie czujników różnego typu pozwala na wzajemne zmniejszanie szumów z różnych źródeł (np. pirometr i termometr rtęciowy mają inne źródła szumów, inny rozkład błędów pomiaru w czasie). Mając rozkład błędów w czasie dla kilku czujników, to wykorzystanie fuzji tych czujników jest w stanie zapewnić wynikowy rozkład błędów, który jest uśredniony. Jeśli na rozkładzie błędów jednego z czujników w danym miejscu jest obecny pik, a na innym rozkładzie w tym miejscu błąd w porównaniu z tym pikiem jest dużo mniejszy, to ten pik zostanie wygładzony przez uśrednienie rozkładu błędów w tym miejscu.

Nawet jeśli pomiary z dwóch czujników są z sobą skorelowane (np. odczyty z dwóch takich samych modeli odbiornika GPS, które są umieszczone prawie w tym samym miejscu) to i tak ich fuzja zapewni pomiar bliżej wartości prawdziwej.

10.5 Wielowymiarowy filtr Kalmana

Czasami jednak filtr musi mieć za zadanie dostarczyć na wyjściu wektor pewnych wartości. Wtedy istnieje potrzeba zastosowania równań filtru Kalmana w postaci macierzowej [23]. W takim wypadku można filtrować jednocześnie kilka wielkości oraz ich pochodne. Gdy mierzonych wartości jest bardzo dużo, wtedy ta metoda jest wydajniejsza numerycznie niż zastosowanie równań algebraicznych dla każdej mierzonej wartości z osobna.

Należy jednak tutaj rozszerzyć pojęcie niepewności estymacji i zacząć je nazywać je wariancją. Dotychczas opisana była wariancja w jednym wymiarze. Natomiast mając do czynienia z kilkoma zmiennymi, należy wiedzieć, że może te zmienne łączyć jakiś stopień zależności (wyrażony jako skalar). Zależność określa tę się terminem kowariancja [39] (wariancja odnosiła się do jednej zmiennej). Dla dwóch zmiennych losowych **x** oraz **y** zebranych w ilości **n** próbek kowariancję dla można wyrazić jako:

$$\sigma(x,y) = \frac{1}{n-1}\sum_{i=1}^{n}(x_i - \overline{x})(y_i - \overline{y}) \tag{11}$$

Przy operowaniu na wektorze zmiennych losowych **X** należy się posługiwać macierzami kowariancji (macierz kwadratowa), które są zdefiniowane jak poniżej.

$$\boldsymbol{P} = \frac{1}{n-1}\sum_{i=1}^{n}(X_i - \overline{X})(X_i - \overline{X})^T \tag{12}$$

$$\boldsymbol{P} \in R^{d\times d} \tag{13}$$

gdzie,
n - liczba próbek (macierz kowariancji zmienia się z każdą iteracją algorytmu0),
d - liczba zmiennych losowych (dla x,y,z $d = 3$)

Na przykład macierz kowariancji dla zmiennych **x** oraz **y** wygląda następująco:

$$\boldsymbol{P} = \begin{bmatrix} \sigma(x,x) & \sigma(x,y) \\ \sigma(y,x) & \sigma(y,y) \end{bmatrix} \tag{14}$$

Poniżej zaprezentowano filtr Kalmana w formie macierzowej.

$$\hat{\boldsymbol{x}}_{n+1,n} = \boldsymbol{F}\hat{x}_{n,n} + \boldsymbol{G}u_n \tag{15}$$

$$\hat{\boldsymbol{x}}_{n,n} = \hat{\boldsymbol{x}}_{n,n-1} + \boldsymbol{K}_n(\boldsymbol{z}_N - \boldsymbol{H}\hat{\boldsymbol{x}}_{n,n-1}) \tag{16}$$

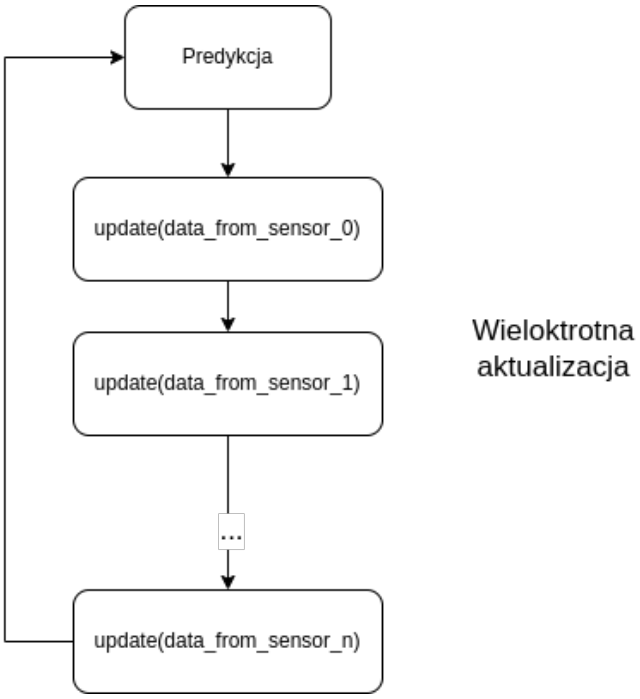
$$\boldsymbol{K}_n = \boldsymbol{P}_{n,n-1}\boldsymbol{H}^T(\boldsymbol{H}\boldsymbol{P}_{n,n-1}\boldsymbol{H}^T + \boldsymbol{R}_n)^{-1} \tag{17}$$

$$\boldsymbol{P}_{n,n} = (\boldsymbol{I} - \boldsymbol{K}_n\boldsymbol{H})\boldsymbol{P}_{n,n-1}(\boldsymbol{I} - \boldsymbol{K}_n\boldsymbol{H})^T + \boldsymbol{K}_n\boldsymbol{R}_n\boldsymbol{K}_n^T \tag{18}$$

$$\boldsymbol{P}_{n+1,n} = \boldsymbol{F}\boldsymbol{P}_{n,n}\boldsymbol{F}^T + \boldsymbol{Q} \tag{19}$$

gdzie,
n_x – długość wektora stanu,
n_u – długość wektora wejścia,
n_z – długość wektora z danymi pomiarowymi
$\hat{\boldsymbol{x}}_{n+1,n}$ – estymowany wektor stanu w kroku czasowym $n+1$,
$\hat{\boldsymbol{x}}_{n,n}$ – estymowany wektor stanu w kroku czasowym n,
$\hat{\boldsymbol{x}}_{n,n-1}$ – estymowany wektor stanu w kroku czasowym $n-1$,
$\boldsymbol{u}_n$ – wektor wejścia,
$\boldsymbol{z}_N$ – wektor z danymi pomiarowymi,
$\boldsymbol{F}$ – macierz tranzycji stanu o wymiarze $n_x \times n_x$,
$\boldsymbol{G}$ – macierz tranzycji wejścia o wymiarze $n_x \times n_u$,
$\boldsymbol{P}_{n+1,n}$ – macierz kowariancji (niepewności estymacji) następnego (prognozy) stanu, jej wymiar to $n_x \times n_x$,
$\boldsymbol{P}_{n,n}$ – macierz kowariancji (niepewności estymacji) aktualnego stanu, jej wymiar to $n_x \times n_x$,
$\boldsymbol{P}_{n,n-1}$ – wcześniejsza estymacja dla poprzedniego stanu (jak obliczano tę macierz, ten stan był wtedy aktualnym stanem) $n_x \times n_x$,
$\boldsymbol{Q}$ – macierz szumu procesowego, jej wymiar to $n_x \times n_x$ i może być ona niezależna(wyrazy niezerowe tylko na diagonali) lub zależna,
$\boldsymbol{H}$ – macierz obserwowalności, jest stała i decyduje o tym, które elementy wektora z pomiarami są brane pod uwagę i z jaką skalą. Jej wymiar to $n_z \times n_x$,
$\boldsymbol{R}_n$ – macierz niepewności pomiaru (kowariancji szumu pomiarowego), jest stała i jej wymiar to $n_x \times n_x$,
$\boldsymbol{K}_n$ – macierz wzmocnienia Kalmana, jej wymiar to $n_x \times n_z$.

Wielowymiarowy filtr Kalmana skonstruowany w postaci macierzowej pozwala na wykorzystanie fuzji czujników w wygodny sposób. Mając pomiary z tej samej chwili czasowej z kilku źródeł, to aby otrzymać ostateczną przefiltrowaną wielkość, należy dla każdego odczytu zaaplikować etap aktualizacji. Odbywa się to w bardzo prosty sposób, którego wizualizację pokazano na rysunku 8.



Rysunek 8: Fuzja czujników z zastosowaniem macierzowego filtru Kalmana

10.6 Filtr Sawickiego-Golaya

Jest to jeden z najpopularniejszych filtrów. Filtr Sawickiego-Golaya [35] dopasowuje wielomian n-tego rzędu do wybranych odcinków sygnału (okna). Ten filtr stosuje tutaj metodę najmniejszych kwadratów. Podczas procesu filtracji filtr ten oblicza nowe dopasowanie wielomianowe za każdym razem, gdy okno pomiarowe przesunie się, czyli zostanie dokonany nowy pomiar. Filtry te mają większą złożoność obliczeniową niż filtry średniej ruchomej, ale dobrze się dopasowują do sygnału oryginalnego.

11 Metody transformacji danych z wybranych czujników

11.1 Transformacja układu współrzędnych geograficznych do kartezjańskich

11.1.1 Opis współrzędnych geograficznych

Współrzędne te są wyrażone w postaci dwóch liczb, szerokości (ang. lattitude) oraz długości (ang. longitude) geograficznych. Obie te współrzędne wyraża się miarą kątową (stopnie) od środka układu współrzędnych geograficznych. Początkiem układu współrzędnych geograficznych jest przecięcie się południka zerowego z równikiem.

Współrzędne te związane są następującymi symbolami:

- ϕ – szerokość geograficzna,
- λ – długość geograficzna.

Orientacja środka układu współrzędnych geograficznych wygląda następująco. Wartości dodatnie dla szerokości geograficznej znajdują się na północy, a dla długości geograficznej na wschodzie od środka układu. Ujemne natomiast na południe i zachód od środka układu.

Szerokość geograficzna jest kątem pomiędzy półprostą poprowadzoną od środka kuli ziemskiej, która przechodzi przez punkt na jej powierzchni, a płaszczyzną równika. Zawiera się w przedziale od $-90°$ do $90°$. Długość geograficzna jest kątem pomiędzy półprostą poprowadzoną od środka kuli ziemskiej, która przechodzi przez punkt na jej powierzchni, a płaszczyzną południka zerowego. Zawiera się w przedziale od $-180°$ do $180°$.

11.1.2 Opis algorytmu transformacji

Współrzędne geograficzne są wygodne przy opisywaniu położenia danego miejsca na globie. Jednakże często istnieje potrzeba skorzystania ze współrzędnych kartezjańskich (x, y), by dany algorytm mógł wykonywać obliczenia na wektorach i wyznaczać trajektorie. Wykorzystamy tutaj algorytm konwersji zwany *equirectangular projection* [14].

Metoda ta polega na przybliżeniu pewnego fragmentu terenu za pomocą płaszczyzny, gdzie błąd przeliczenia współrzędnych geograficznych na kartezjańskie (wynikający z tego, że Ziemia jest okrągła) będzie znikomy.

Transformację współrzędnych geograficznych na kartezjańskie wyraża się wzorami 20 oraz 21. Współrzędne po konwersji wyrażone są w metrach [14]. Długość geograficzna referencyjna jest taką, gdzie skala transformacji będzie 1:1.

$$x = R(\lambda - \lambda_0)cos(\phi_1) \tag{20}$$

$$y = R(\phi - \phi_0) \tag{21}$$

gdzie,
λ – długość geograficzna danej pozycji,
ϕ – szerokość geograficzna danej pozycji,
λ_0 – długość geograficzna wybranego środka układu współrzędnych,
ϕ_0 – szerokość geograficzna wybranego środka układu współrzędnych,
ϕ_1 – długość geograficzna referencyjna,
R – promień Ziemi wyrażony w metrach

11.2 Transformacja danych z magnetometru na orientację obiektu

Dane z magnetometru dostarczają nam podstawowe wskazania na temat aktualnej orientacji danego obiektu. By uzyskać pomiary aktualnej orientacji wyrażonej w radianach, to należy się posłużyć do tego wzorami 22 [15]:

$$\phi_R = \begin{cases} y > 0, & \frac{\pi}{2} - \operatorname{arc\,tg}(\frac{H_x}{H_y}) \\ y < 0, & \frac{3\pi}{4} - \operatorname{arc\,tg}(\frac{H_x}{H_y}) \end{cases} \tag{22}$$

gdzie,
ϕ_R – orientacja obiektu,
H_y – składowa y wektora pola magnetycznego,
H_x – składowa x wektora pola magnetycznego

11.3 Odometria

Znajomość pozycji pojazdu ma fundamentalne znaczenie dla jego prawidłowego działania. Odometria jest metodą na obliczenie dystansu przebytego przez pojazdu, względem jego pozycji początkowej mierząc obroty kół dzięki zastosowaniu enkoderów obrotowych [40]. Jest to bardzo prosta metoda na poznanie aktualnej pozycji pojazdu, lecz jest ona zwykle obarczona błędem ze względu na dryft pozycji spowodowany poślizgiem kół [46]. Błędy odometrii akumulują się z czasem, zatem cechuje się odpowiednią dokładnością tylko przez pewien czas jazdy pojazdu. Na rynku obecne są zastosowania odometrii kołowej oraz odometrii wizualnej VO [46]. Odometria kołowa jest najprostszą oraz najbardziej rozpowszechnioną metodą na mierzenie dystansu pojazdu względem jego pozycji początkowej. Odometria wizualna pozwala na estymację pozycji dzięki bieżącej analizie tylko i wyłącznie strumienia zdjęć uzyskanego z jednej lub wielu kamer [46]. Idea polega na badaniu przemieszczenia się poszczególnych pikseli pomiędzy dwoma klatkami i na tej podstawie może być wyznaczona przebyta odległość. Metoda ta jest dobrym balansem pomiędzy dokładnością estymacji pozycji a ceną [46]. Odometria wizualna jest odporna na poślizgi kół oraz cechy przebywanego terenu. Dzięki zastosowaniu odometrii jesteśmy w stanie określić aktualną liniową oraz kątową prędkość pojazdu, a po scałkowaniu, jego pozycję i orientację. Wszystkie metody odometrii znajdą zastosowanie w pomieszczeniach oraz innych zamkniętych przestrzeniach, gdzie przykładowo dostęp do sygnału GPS jest niemożliwy. Odczyty z odometrii mogą być zintegrowane przykładowo z odczytami z GPS po to, aby zwiększyć precyzję pomiarów. Odometria stosowana jest głównie w branży robotycznej oraz automotive. Poniższe wzory opisują jak transformować dane z odometrii kołowej do współrzędnych *(x,y)*.

$$x_L = \frac{n_L}{N} \tag{23}$$

$$x_R = \frac{n_R}{N} \tag{24}$$

$$d = \frac{x_L + x_R}{2} \tag{25}$$

$$d\theta = \operatorname{arc\,sin} \frac{x_R - x_L}{L} \tag{26}$$

$$dx = \cos d\theta \cdot d \tag{27}$$

$$dy = \sin d\theta \cdot d \tag{28}$$

gdzie,
N – rozdzielczość enkoderów $[\frac{impuls}{m}]$,
L – odległość między środkiem prawej i lewej opony$m$
n_L – liczba impulsów zebrana przez enkoder lewego koła w czasie dt,
n_R – liczba impulsów zebrana przez enkoder prawego koła w czasie dt,
x_L – dystans przebyty przez lewe koło w czasie dt,
x_R – dystans przebyty przez prawe koło w czasie dt,
d – dystans przebyty przez pojazdu w czasie dt,
$d\theta$ – kąt przebyty przez pojazdu w czasie dt,
dx – rzut dystansu przebytego przez pojazdu na oś x w czasie dt,
$d\theta$ – rzut dystansu przebytego przez pojazdu na oś y w czasie dt

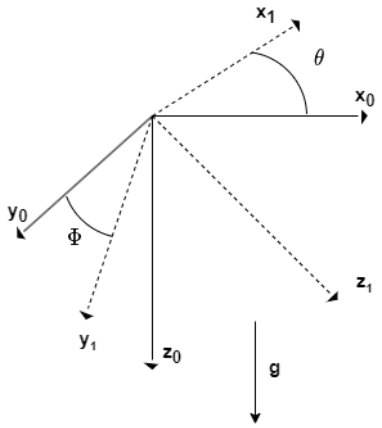
11.4 Określanie kątów nachylenia obiektu za pomocą wskazań akcelerometru

W tym podrozdziale opisano sposób wyznaczania kątów obrotu obiektu między osiami układu zerowego oraz wtórnego. Do tego celu można wykorzystać wskazania akcelerometru, które dostarczają informacje na temat składowych przyspieszenia liniowego działającego na obiekt. Kąt pomiędzy osiami x wspomnianych układów oznaczono jako θ. Kąt pomiędzy osiami y oznaczono jako Φ. Znajomość kątów θ oraz Φ jest szczególnie istotna, dla systemów pokładowych maszyn latających (np. drony). By uzyskać informacje o wartości wspomnianych kątów, należy wykorzystać informacje o wektorze przyspieszenia liniowego działającego na obiekt [57]. Wzory 29 oraz 30 pozwalają na wyznaczenie wspomnianych kątów [57]. Na rysunku 9 przedstawiono wizualizację kątów Φ oraz θ.

$$\Phi = \operatorname{arc\,tg}(a_y/a_z) \tag{29}$$

$$\theta = \operatorname{arc\,sin}(a_x/g) \tag{30}$$

gdzie,
a_x, a_y, a_z – składowe wektora przyspieszenia,
g – przyspieszenie grawitacyjne



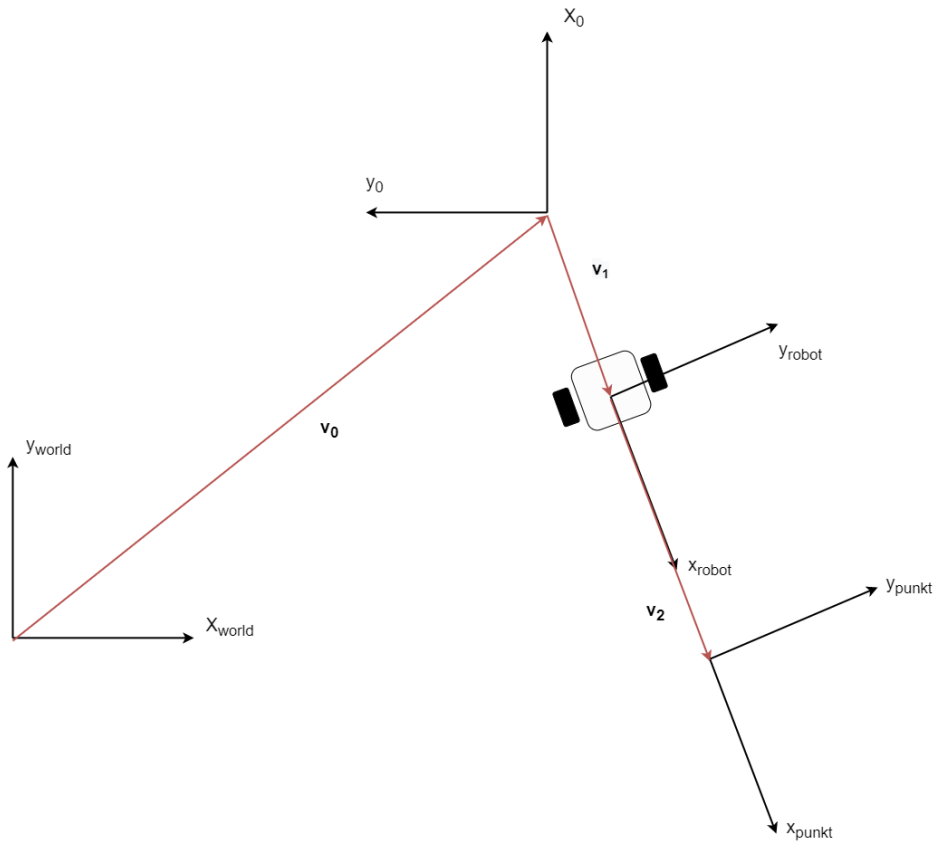
Rysunek 9: Wizualizacja kątów nachylenia

12 Model robota

12.1 Opis układów współrzędnych

Cały system jest wyposażony w 4 układy odniesienia. To według nich dokonywane są obliczenia podczas wykonywania się algorytmów.

- Układ globalny zerowy znajdujący się na przecięciu równika oraz południka zerowego.
- Układ lokalny zerowy znajdujący się w miejscu rozpoczęcia misji.
- Układ robota.
- Układ punktu, do którego robot podąża.



Rysunek 10: Układy współrzędnych w systemie

Wszystkie układy są układami prawoskrętnymi. Oś Z nie była prezentowana na schemacie, by utrzymać jego prostotę i klarowność. Układ zerowy jest układem zerowym globalnym. Jest to jedyny układ, który jest nieruchomy przez cały czas. Jego obie współrzędne geograficzne są równe zeru.

Układ lokalny zerowy jest układem, którego współrzędne geograficzne są równe współrzędnym, pod jakimi robot znajdzie się podczas rozpoczynania misji. Wektor wodzący od układu początku globalnego, do początku tego układu wynosi v_0. Jest on obrócony względem układu globalnego o 90^o. Zapewnia to zgodność orientacji symulacji oraz prawdziwego robota.

Układ robota jest związany z modelem robota. Robot zawsze skierowany wzdłuż swojej osi x. Wektor wodzący od układu początku lokalnego zerowego, do początku tego układu wynosi v_1. Układ docelowego punktu jest związany z aktualnym celem robota. Początek tego układu znajduje się tam, gdzie są współrzędne geograficzne docelowego punktu. Jego oś X leży na osi X układu robota, kąt między nimi wynosi 0. Wektor wodzący od układu początku układu robota, do początku tego układu określony jest jako v_2. Jeśli robot porusza się do punktu, to równanie przedstawiające podróż do punktu przedstawia się następująco.

$$v_0^{(world)} = [x_0 \;\; y_0 \;\; 0]^T \tag{31}$$

$$v_1^{(1)} = v_{robot}^{world} - v_0^{(world)} \tag{32}$$

Wektor v_2 jest wektorem, który jest różnicą trasy pomiędzy układem robota oraz punktu

$$v_2^{(robot)} = r_{punkt}^{world} - r_{robot}^{world} \tag{33}$$

Układy robota i zerowe są obrócone między sobą o stały kąt $\phi_{0,robot}$

$$R_{robot}^{0}(\phi_{0,robot}) = const \tag{34}$$

Układy robota oraz punktu mają tę samą orientację.

$$R_{punkt}^{robot}(t) = I_{3x3} \tag{35}$$

Układy globalny zerowy oraz lokalny zerowy są obrócone względem siebie o 90^o

$$R_{punkt}^{robot}(\frac{\pi}{2}) = const \tag{36}$$

Współrzędne x_0, y_0, współrzędne robota r_{robot}^{world} oraz punktu docelowego r_{punkt}^{world} są odczytywane z GPS i znane na początku obliczeń.

12.2 Opis modelu robota wykorzystanego w symulacji autopilota

W symulacji użyto modelu czterokołowego robota mobilnego. Model składa się z 5 elementów: korpusu głównego oraz czterech kół. Koła obracają się wokół swoich własnych osi **y**. Jest to prosty model, wystarcza do przeprowadzenia testów oprogramowania autopilota. Współczynnik tarcia pomiędzy kołami a nawierzchnią ustawiono na 0,8. Maksymalna prędkość liniowa to $1\frac{m}{s}$, a prędkość kątowa $1\frac{rad}{s}$. Masa całego robota w symulacji wynosi 15 kg.

13 Zaimplementowane składowe algorytmy autopilota

13.1 Algorytmy konwersji danych z czujników

13.1.1 Odczytywanie pozycji robota

W skład oprogramowania autopilota wchodzi węzeł odpowiedzialny za transformację współrzędnych geograficznych na współrzędne kartezjańskie. Odczytuje on z odpowiedniego kanału (patrz rozdział 13) pozycję geograficzną robota i wykorzystuje on algorytm odwzorowania walcowego równoodległościowego, by otrzymać współrzędne kartezjańskie wyrażone w metrach. Wspomniany algorytm został opisany w rozdziale 11.1.1. By uprościć obliczenia, to długość geograficzną referencyjną przyjęto jako długość geograficzną w środku mapy ϕ_0, czyli $\phi_1 = \phi_0$.

13.1.2 Odczytywanie orientacji robota

W skład oprogramowania autopilota również wchodzi węzeł odpowiedzialny za transformację danych z magnetometru na aktualną orientację robota. Jako podstawy użyto algorytmu konwersji przedstawionego w rozdziale 11.1. Znaczenie symboli w tym rozdziale jest takie samo jak w rozdziale 11.1. Postać algorytmu opisanego w rozdziale 11.1 jest niepraktyczna w dalszych obliczeniach. By ujednolicić zapis użyto funkcji $atan2$.

$$\phi_R = \text{arc tg}\, 2(H_x, H_y,) \tag{37}$$

Ta funkcja zwraca wartości z zakresu $(-\pi,\pi)$. Jednak wygodniej było by korzystać z wartości z zakresu $(0,2\pi)$. Dzięki wykorzystaniu dzielenia z resztą przez 2π, poniższa funkcja spełnia wspomniane oczekiwania.

$$\phi_R = \quad \text{mod}\ (\text{arc tg}\, 2(H_x, H_y,) + 2\pi, 2\pi) \tag{38}$$

Węzeł odczytuje surowe dane z magnetometru i wykorzystując zaimplementowany w Pythonie wzór 38 publikuje aktualną orientację robota wyrażoną w radianach.

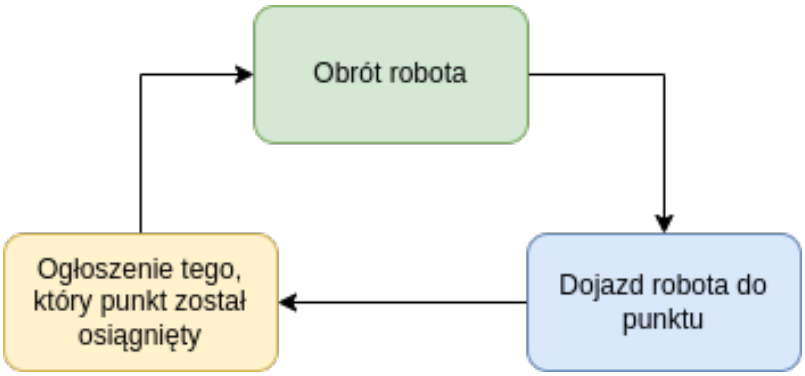
13.2 Algorytm wykonywania misji

Algorytm wykonywania misji wykorzystuje algorytm skrętu oraz dojazdu do punktu wraz z wymaganą dokładnością, by zrealizować misję przesłaną w pliku. Ostatecznie doprowadza on robota do ostatniej lokalizacji zapisanej w pliku. Na wejściu przyjmuje on lokalizację pliku z misją, odczytuje go i wykonuję tę misję. Na rysunkach 11 oraz 12 znajdują się graficzne przedstawienia tego algorytmu.

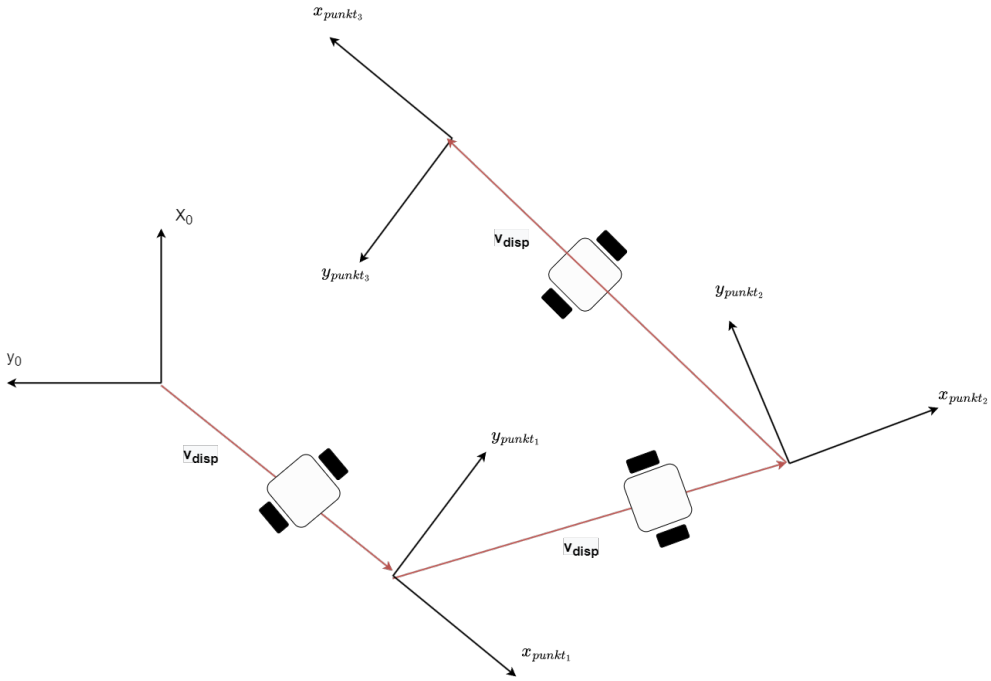
13.3 Algorytm skrętu wraz z wymaganą dokładnością

Zadaniem tego algorytmu jest obrócenie robota, tak by jego oś $\mathbf{x}$ pokryła się z kierunkiem wektora przemieszczenia pomiędzy aktualnym punktem a punktem następnym z wymaganą dokładnością podaną w stopniach.

Najpierw następuje obliczenie wspomnianego wektora przemieszczenia v_{disp} z punktu aktualnego do punktu następnego. Znając aktualną orientację robota oraz v_{disp} następuje obliczenie całkowitego wymaganego kąta potrzebnego do obrotu, oraz kierunku tego obrotu. Opisany algorytm opiera się na wzorach 39 oraz 40.



Rysunek 11: Schemat algorytmu wykonywania misji



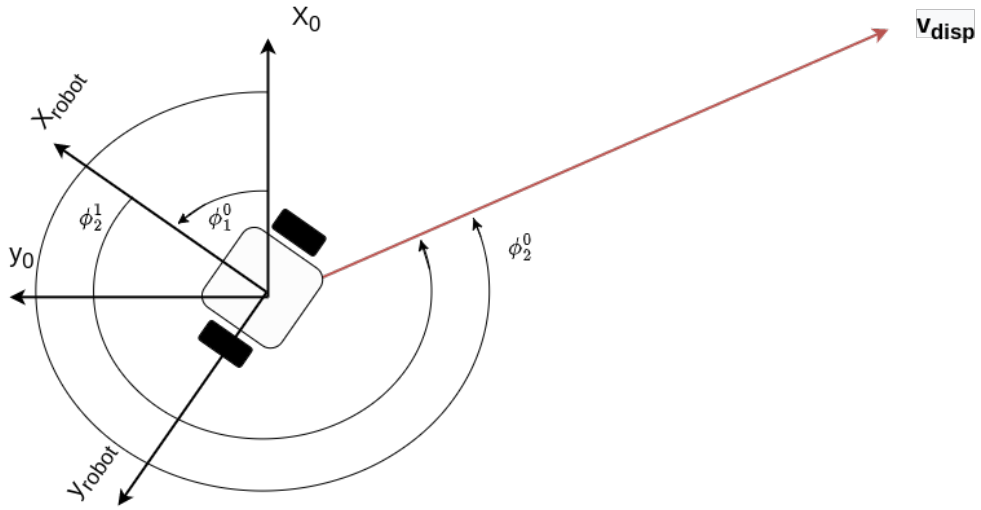
Rysunek 12: Schemat podróży robota

$$\phi_2^1 = \phi_2^0 - \phi_1^0 \tag{39}$$

$$\phi_2^0 = \operatorname{arc\,tg} 2(v_{disp_y}, v_{disp_x}) \tag{40}$$

Wartość ϕ_1^0 odczytuje się z kompasu. Jednak jest obecny tutaj obrót z pewną dokładnością co do kąta. Jest to problematyczne do zrealizowania, gdyż błąd 2-3^o jest dużym błędem, często niepozwalającym na prawidłowe dotarcie do celu. Rozwiązanie tego problemu przedstawiono w rozdziale 17.2.

Obrót jest sterowany parametrami i te parametry są podane w pliku misji obok współrzędnych docelowego punktu. Wszystkie parametry użyte w pliku misji zostały przedstawione w rozdziale 13.5. Obrót robotem odbywa się za pomocą wymuszenia prędkościowego. Zadaje się pewną prędkość kątową, by robot znalazł się w danej orientacji absolutnej. Obrót robota składa się z trzech faz, które przedstawiono również na wykresie 14



Rysunek 13: Schemat algorytmu skrętu robota

- stopniowego rozwijania prędkości kątowej robota,
- obrotu robota ze stałą prędkością kątową,
- stopniowej redukcji prędkości kątowej robota.

Rysunek 14: Schemat rozłożenia wymuszenia prędkościowego w czasie dla skrętu robota

gdzie:
$\phi_{traveled}$ – moduł przebytego już kąta,
$abs(\phi_2^1)$ – moduł całkowitego brakującego kąta,
$\frac{\phi_{traveled}}{abs(\phi_2^1)}$ – postęp wykonywania się algorytmu,
ω_{max} – maksymalna prędkość kątowa,
K_2 – współczynnik oznaczający postęp, przy którym robot przestaje zwiększać swoją prędkość kątową,
K_3 – współczynnik oznaczający postęp, przy którym robot zaczyna redukować swoją prędkość kątową.

Maksymalna prędkość kątowa ω_{max} jest wyrażona wzorem 41.

$$\omega_{max} = K_1 \cdot abs(\phi_2^1) \tag{41}$$

gdzie,
K_1 – współczynnik oznaczający ograniczający maksymalną prędkość kątową

Sterowanie prędkością kątową w fazach redukcji i rozwijania prędkości kątowej odbywa się za pomocą pętli sprzężenia zwrotnego. Ta pętla jest skwantyzowana, to znaczy, że wymuszenie nie jest nadawane w sposób ciągły, tylko dopiero po osiągnięciu danego postępu w obrocie. Wielkość ziarnistości kwantyzacji N podaje się również w pliku misji. Ta liczba oznacza liczbę zmian prędkości podczas wykonywania się faz stopniowego rozwijania prędkości kątowej robota oraz stopniowej redukcji prędkości kątowej robota.

Sterowanie brakującym kątem odbywa się jednak w sposób ciągły. Algorytm jest zaprojektowany tak, że robot zawsze wykonuje obrót mniejszy niż π, czyli w celu obrotu do punktu wybiera bliższą trasę kątową.

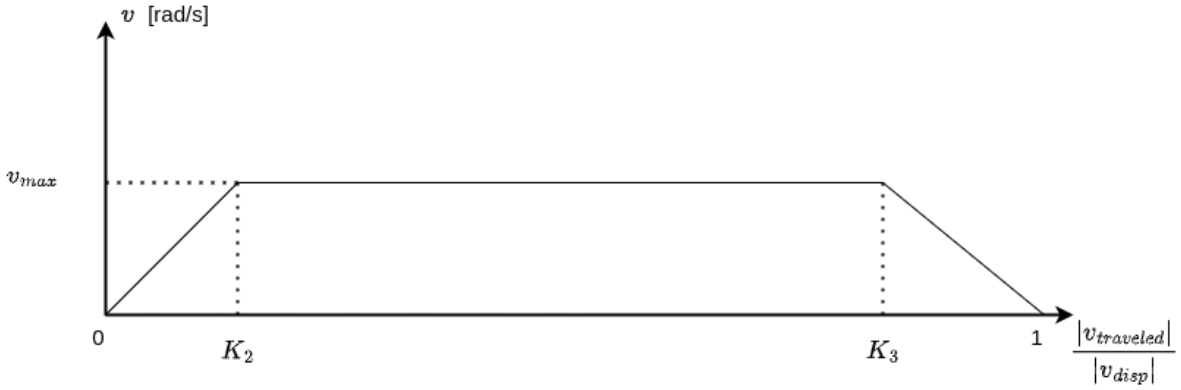
13.4 Algorytm dojazdu do punktu wraz z wymaganą dokładnością

Zadaniem tego algorytmu jest przemieszczenie robota, tak by jego położenie pokryło się z punktem docelowym z wymaganą dokładnością podaną w metrach. Najpierw następuje obliczenie wektora przemieszczenia v_{disp} (patrz rysunek 13) z aktualnego punktu do następnego punktu. Jednak obecny jest tutaj dojazd do punktu z pewną dokładnością co do pewnego błędu wyrażonego w metrach. Jest to pewne wyzywanie, gdyż już po starcie robot jest obarczony pewnym błędem orientacji. Dodatkowo skala problemu jest zwiększana przez niedokładność odbiornika GPS. Dlatego punkty należy rozsiewać gęsto lub zmniejszać wymaganą dokładność.

Dojazd do punktu jest sterowany pewnymi parametrami i te parametry są podane w pliku misji obok współrzędnych docelowego punktu. Wszystkie parametry w pliku misji zostały przedstawione w rozdziale 13.5. Ruch translacyjny robota odbywa się za pomocą wymuszenia prędkościowego liniowego. Dojazd robota do punktu składa się z trzech faz, które zostały również przedstawione na wykresie 15:

- stopniowego rozwijania prędkości liniowej robota,
- jazdy robota z stałą prędkością liniową,
- stopniowej redukcji prędkości liniowej robota.

gdzie,
$v_{traveled}$ – przebyty dystans
$|v_{disp}|$ – odległość między punktami
$\frac{|v_{traveled}|}{|v_{disp}|}$ – postęp wykonywania się algorytmu
v_{max} – maksymalna prędkość liniowa
K_2 – współczynnik oznaczający postęp, przy którym robot przestaje zwiększać swoją prędkość liniową
K_3 – współczynnik oznaczający postęp, przy którym robot zaczyna redukować swoją prędkość liniową.



Rysunek 15: Schemat rozłożenia wymuszenia prędkościowego w czasie dla algorytmu dojazdu do punktu

Sterowanie prędkością liniową w fazach stopniowego rozwijania prędkości liniowej robota i stopniowej redukcji prędkości liniowej robota odbywa się za pomocą pętli sprzężenia zwrotnego. Ta pętla jest skwantyzowana, to znaczy, że wymuszenie nie jest nadawane w sposób ciągły, tylko dopiero po osiągnięciu danego postępu w obrocie. Wielkość ziarnistości kwantyzacji N podaje się również w pliku misji. Ta liczba oznacza liczbę zmian prędkości podczas wykonywania się faz redukcji oraz rozwijania prędkości liniowej. Sterowanie brakującą odległością odbywa się jednak w sposób ciągły. Maksymalna prędkość liniowa v_{max} jest wyrażona wzorem 42.

$$v_{max} = K_1 \cdot |v_{disp}| \tag{42}$$

gdzie:
K_1 – współczynnik oznaczający ograniczający maksymalną prędkość liniową

13.5 Opis pliku misji

W pliku misji znajdują się kolejne punkty, do których robot ma dotrzeć wraz z podanymi parametrami trasy takimi jak na przykład dokładność. Współrzędne są podane jako współrzędne geograficzne. Po przygotowaniu zestawu punktów, plik należy następnie załadować do autopilota, który po pomyślnym załadowaniu go zacznie wykonywać misję. Znaczenie parametrów użytych w pliku zostało opisane w rozdziałach 13.3 i 13.4

```
52.4535363,21.9054322,0.1,0.2,0.8,10,0.3,1
52.4538811,21.9039300,0.1,0.2,0.8,10,0.3,1
52.4532222,21.9036366,0.1,0.2,0.8,10,0.3,1
52.4588843,21.9035553,0.1,0.2,0.8,10,0.3,1
52.4576575,21.9035444,0.1,0.2,0.8,10,0.3,1
52.4545334,21.9022263,0.1,0.2,0.8,10,0.3,1
52.4556423,21.9035363,0.1,0.2,0.8,10,0.3,1
52.4536544,21.9034331,0.1,0.2,0.8,10,0.3,1
```

Listing 19: Przykładowy plik misji

Parametry są ułożone w następującej kolejności:

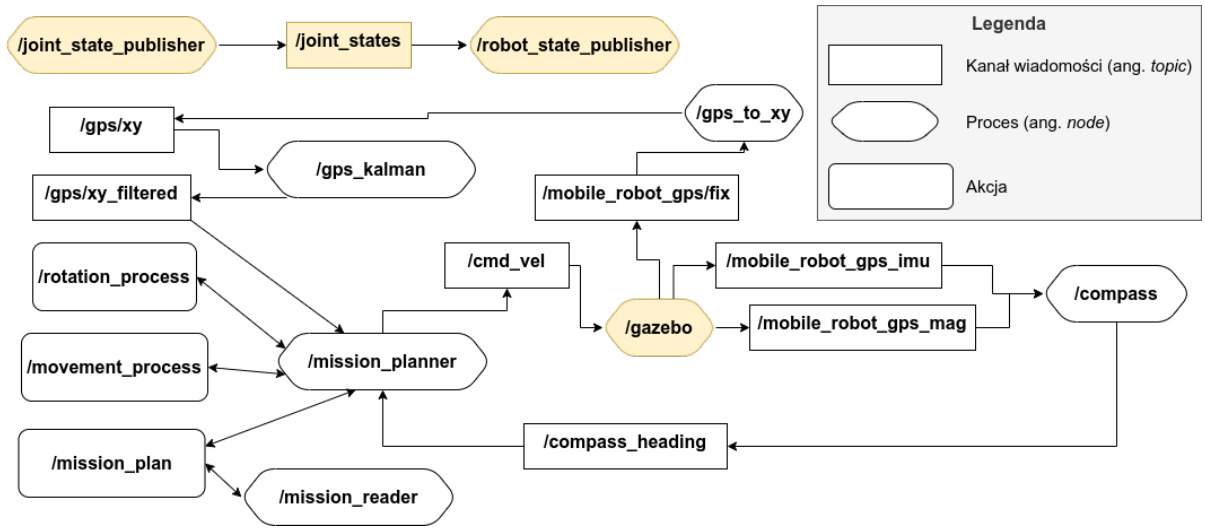
- szerokość geograficzna,
- długość geograficzna,
- współczynnik K1,
- współczynnik K2,
- współczynnik K3,
- stopień kwantyzacji zmian prędkości,
- precyzja dojazdu do punktu wyrażona w metrach,
- precyzja obrotu wyrażona w stopniach.

14 Struktura oprogramowania

14.1 Architektura

W projekcie wykorzystano architekturę zorientowaną na mikroserwisy. Było to podyktowane koniecznością wykorzystania środowiska ROS, które narzuca taki styl architektury. Dodatkowo istniała konieczność przeprowadzenia kilku eksperymentów, zatem niektóre części oprogramowania musiały być co jakiś czas podmieniane na inne.

14.2 Struktura procesów i kanałów wiadomości



Rysunek 16: Struktura procesów i kanałów wiadomości

Na rysunku 16 zaprezentowano strukturę oprogramowania autopilota (niektóre elementy są związane tylko i wyłącznie z samą symulacją). Rysunek został stworzony na podstawie schematu, za którego wygenerowanie jest odpowiedzialny ROS. W legendzie podano oznaczenia rodzajów elementów przedstawionych na schemacie. Elementy zaznaczone na żółto są elementami związanymi z symulacją.

Węzeł **/mission_plannner** jest centrum całego oprogramowania autopilota, które agreguje wszystkie przefiltrowane dane i aplikuje je do algorytmów nawigacyjnych. Komunikuje się z akcjami algorytmów **/movement_process** oraz **/rotation_process** służącymi do skręcania oraz dojazdu do punktu z wymaganą dokładnością. Pośrednio za pomocą wspomnianych algorytmów steruje wymuszeniem prędkościowym pojazdu, umożliwiając sterowanie robotem. Docierają do niego przefiltrowane odczyty GPS z procesu **/gps_kalman** oraz orientacji z kanału **/compass_heading**. Proces **/mission_reader** przyjmuje misję, analizuje jej zawartość, przetwarza tę zawartość i przekazuje wynik tego parsowania[3] jako trasę do węzła **/mission_plannner**. Węzeł **/gps_kalman** pobiera surowe pomiary dotyczące pozycji z kanału **/gps/xy**. Węzeł **/compass** pobiera surowe dane dotyczące odczytów z magnetometru oraz żyroskopu odpowiednio z kanałów **/mobile_robot_gps_mag** oraz **/mobile_robot_gps_imu**. Dane na **/mobile_robot_gps_mag**, **/mobile_robot_gps_imu** oraz **/mobile_robot_gps/fix** są przesyłane bezpośrednio z symulacji.

[3] Analizowania składni danych wejściowych i przetwarzania ich na obiekty w zaalokowane pamięci

Poniżej znajduje się opis wszystkich procesów, kanałów wiadomości oraz akcji:

- Proces **/joint_state_publisher** odpowiedzialny za publikowanie wiadomości o stanie więzów kinematycznych robota na kanał **/joint_states**.
- Kanał **/join_states** wiadomości służący do przesyłu wiadomości na temat więzów kinematycznych robota.
- Proces **/robot_state_publisher** związany z publikowaniem wiadomości o stanie robota. Jest wewnętrznie wykorzystywany przez Gazebo.
- Proces główny **/gazebo** komunikuje się z symulatorem Gazebo, wystawia oraz subskrybuje wszystkie kanały bezpośrednio związane z symulacją.
- Kanał **/mobile_robot_gps_imu** jest tym, na który są wysyłane i odbierane są wiadomości dotyczące surowych wskazań IMU.
- Kanał **/mobile_robot_gps_mag** jest tym, na który wysyłane i odbierane są wiadomości dotyczące surowych wskazań magnetometru.
- Proces **/compass** dokonuje fuzji wskazań magnetometru i IMU za pomocą filtru Kalmana, by otrzymać wskazania orientacji. Obliczoną orientację publikuje na kanał **/compass_heading**.
- Kanał **/compass_heading** jest tym, na który przychodzą wiadomości odnośnie do absolutnej przefiltrowanej orientacji robota.
- Kanał **/mobile_robot_gps/fix** jest tym, przez który przepływają surowe odczyty wskazań odbiornika GPS.
- Proces **/gps_to_xy** konwertuje współrzędne geograficzne do kartezjańskich. Wyniki publikuje na **/gps/xy**.
- Kanał **/gps/xy** służy do przepływu wiadomości dotyczących pozycji robota we współrzędnych kartezjańskich.
- Proces **/gps_kalman** filtruje wskazania GPS za pomocą filtru Kalmana i publikuje przefiltrowane wiadomości na kanał **/gps/xy_filtered**.
- Przez kanał **/gps/xy_filtered** przepływają przefiltrowane wskazania GPS.
- Proces **/mission_reader** przyjmuje plik misji i uruchamia akcję **/mission_plan**.
- Akcja **/mission_plan** nadzoruje wykonywanie całej misji.
- Głównym procesem jest **/mission_plannner**, w którym osadzone są wszystkie algorytmy dotyczące sterowania robotem oraz akcji.
- Akcja **/rotation_process** odpowiedzialna jest za wykonanie obrotu robota do zadanej orientacji.
- Akcja **/movement_process** odpowiedzialna jest za dojazd robota do zadanej pozycji.
- Na kanał **cmd_vel** wysyłane są wymuszenia prędkościowe napędu robota.

14.3 Użyte języki programowania oraz biblioteki

Do implementacji funkcjonalności autopilota wykorzystano języki C++ oraz Python. Do implementacji wszystkich węzłów skorzystano z poniższych bibliotek:

- Pakietu **rospy** dostarczającego narzędzi do tworzenia węzłów oraz subskrypcje i publikowanie tematów w języku Python.
- Pakietu **actionlib** dostarczającego narzędzi do definiowania serwerów oraz klientów akcji.
- Pakietu **roscpp** dostarczającego narzędzi do tworzenia węzłów oraz subskrypcji i publikowania tematów w języku C++.
- Pakietu **std_msgs** dostarczającego definicji standardowych wiadomości dotyczących typów prostych takich jak np. Int32 lub String.
- Pakietu **sensor_msgs** dostarczającego definicji standardowych wiadomości dotyczących wskazań z czujników.
- Pakietu **nav_msgs** dostarczającego definicji standardowych wiadomości dotyczących wskazań odometrii czy GPS.
- Pakietu **geometry_msgs** dostarczającego definicji standardowych wiadomości dotyczących danych geometrycznych (np. definicje punktów lub wektorów).

Przy implementacji węzła, który filtrował dane filtrem Sawickiego-Golaya użyto pakietu **scipy.signal**, który zapewniał gotową implementację tego filtru. Do zaimplementowania wielowymiarowego filtru Kalmana użyto pakietu **numpy** dostarczającego możliwość obliczeń na macierzach.

14.4 Sposób komunikacji międzyprocesowej

Użyto komunikacji międzyprocesowej działającej z wykorzystaniem gniazd. Gniazda opisano w rozdziale 4.1. Wybór taki był spowodowany tym, że ROS narzuca komunikację wykorzystującą gniazda. Dodatkowo ten sposób jest najwydajniejszym ze wszystkich sposobów wymienionych w rozdziale 4. Ta metoda jako jedyna nie wykonuje operacji na plikach, co jest wąskim gardłem pozostałych metod. Odczyt i zapis do pliku są kosztownymi operacjami, zatem nie istnieje potrzeba buforowania danych. Dodatkowo pozwala ona na przesyłanie bardziej skomplikowanych rodzajów danych i zapewnia większy poziom abstrakcji nad przesyłem.

14.5 Tworzenie plików wykonywalnych

ROS narzuca używanie narzędzia **CMake** do budowania aplikacji. Domyślnie generuje ono pliki Makefile, które można potem skompilować narzędziem **make**. Dodatkowo ROS używa nakładki na narzędzie **CMake** zwane **catkin**. Ta nakładka powala w plikach CMakeLists.txt deklarować pliki wiadomości, serwisów i akcji, które mają być poddane kompilacji.

15 Dobór elementów oprogramowania autopilota

15.1 Wybór właściwego sterownika napędu

Napęd który należy dobrać charakteryzuje się następującymi cechami:

- Pozwoli robotowi skręcać w miejscu.
- Pozwoli na uzyskanie wystarczającej dokładności podczas obrotu.

W takim wypadku właściwym wydaje się wybór napędu różnicowego, który pozwoli spełnić powyższe założenia. Taki napęd również uprości budowę robota. Aby jak najlepiej oddać jak najwierniej rzeczywistość oraz zastosować napęd różnicowy wybieramy **skid_steer_drive_controller**. Ta odmiana sterownika napędu różnicowego zapewniona przez wspomniany pakiet dodatkowo wprowadza możliwość poślizgów kół, co przybliża nas do rzeczywistości.

15.2 Pakiety symulujące dane czujniki

W symulatorze Gazebo oraz środowisku ROS istnieje możliwość symulacji wszystkich niezbędnych czujników. Poniższe pakiety zapewniają możliwość definiowania modelu zakłóceń.

- Pakiet **libhector_gazebo_ros_gps** służy do symulacji odbiornika GPS, posiada on interfejs do definiowania postaci symulowanych szumów.
- Pakiet **libhector_gazebo_ros_magnetic** symuluje wskazania magnetometru.
- Pakiet **libgazebo_ros_imu** symuluje wskazania IMU, czyli umożliwia agregację pomiarów z akcelerometru oraz żyroskopu.

15.3 Filtr wygładzający odczyty orientacji robota

Odnosząc się do analiz w rozdziale 17.2 wywnioskowano, że skorzystanie z filtru Kalmana zapewni wygładzenie danych, w sposób pozwalający na bezproblemowe korzystanie z algorytmów autopilota. Filtr ten zapewnia odpowiednią reaktywność względem danych surowych oraz cechuje się brakiem opóźnień. Reaktywność należy rozumieć jako szybkość nadążania za zmianami sygnału oryginalnego.

Metodą kalibracji filtru Kalmana jest dobranie odpowiednich macierzy stałych oraz macierzy startowych, gdzie w przypadku filtrów Sawickiego-Golaya średniej ruchomej należy odpowiednio manipulować rozmiarem okna pomiarowego. W przypadku Kalmana wartości kalibracyjne są wielkościami fizycznymi, zatem metoda kalibracji jest intuicyjna. Kalibracja filtru Sawickiego-Golaya jest trudna i nieintuicyjna, ze względu na potrzebę doboru stopnia wielomianu oraz abstrakcyjnych właściwości filtra.

Jeśli chodzi o złożoność obliczeniową, to filtr Sawickiego-Golaya charakteryzuje się dużą złożonością obliczeniową. Filtr średniej ruchomej posiada wysoką złożoność ze względu na dodatkowe zabiegi opisane w rozdziale 17.2.3. Zastosowany filtr Kalmana ma niską złożoność, gdyż jeden krok składa się tylko z kilku prostych obliczeń. Jeśli chodzi o reaktywność względem danych surowych, to w przypadku filtrów Sawickiego-Golaya jest ona mniejsza względem reszty przetestowanych filtrów. Filtry średniej ruchomej oraz filtr Kalmana ma większą reaktywność. Jeśli chodzi o wzrost jakości wygładzenia względem wzrostu rozmiaru okna pomiarowego, to wzrost ten jest szybszy dla

filtrów średniej ruchomej niż Sawickego-Golaya. Dla filtrów Kalmana szerokość okna pomiarowego jest zawsze stała. By osiągnąć wymagane wygładzenie, to minimalny rozmiar okna pomiarowego dla filtru Sawickiego-Golaya był większy niż 51 próbek. Dla filtru średniej ruchomej wystarczyło 35 próbek. Tylko filtr Kalmana pozwala na fuzję danych z wielu czujników. Zatem ostatecznie został wybrany filtr Kalmana.

15.4 Filtr wygładzający odczyty pozycji robota

Odnosząc się do analiz wykonanych w rozdziale 17.3 wywnioskowano, że w tym przypadku również skorzystanie z filtru Kalmana, ale w postaci macierzowej, będzie odpowiednim rozwiązaniem. Dołożenie pomiarów z odometrii nie wnosi poprawienia jakości odczytów, więc ten etap nie został zastosowany.

16 Odwzorowanie rzeczywistości w symulacji

16.1 Założenia i uproszczenia

Algorytm działa, ale z pewnymi założeniami dotyczącymi samego robota oraz jego otoczenia. Pierwszym założeniem jest to, że teren wokół robota jest idealnie płaski oraz że współczynnik tarcia między kołami robota a nawierzchnią jest stały podczas jazdy. Drugim założeniem jest to, że na trajektorii podróży robota nie ma żadnych przeszkód. Trzecim założeniem jest to, że silniki sterujące napędem robota mają możliwość szybkiego hamowania oraz przyspieszania. Dodatkowym założeniem jest to, że robot ma równomiernie rozłożony ciężar w całej swojej objętości, inaczej mówiąc, w całym korpusie robota jego gęstość jest stała.

16.2 Wymagania względem układu elektronicznego

Robot musi mieć zaprogramowany sterownik napędu, który steruje wymuszeniem prędkościowym. Wymagany jest tutaj napęd różnicowy. Robot musi być wyposażony w odbiornik GPS. By algorytm obliczania orientacji mógł zadziałać, to robot musi być wyposażony w magnetometr oraz w IMU. Komputer pokładowy robota musi mieć możliwość zainstalowania na nim systemu operacyjnego z rodziny Debian oraz mieć wystarczającą ilość pamięci operacyjnej, oraz dyskowej, by móc zainstalować na nim wspomniane środowisko ROS.

16.3 Wymagania względem otoczenia robota

W aktualnej implementacji autopilota robot jest w stanie wykonywać proste misje. Robot nie powinien być umieszczony w budynkach oraz w miejscach znajdujących się pod ziemią. W aktualnej implementacji robot nie będzie w stanie podróżować między gęstymi zabudowaniami ze względu na słaby sygnał GPS, którego błąd odczytu pozycji może przekroczyć odległość między budynkami. Teren taki również nie powinien być zbytnio zalesiony, gdyż w takim otoczeniu również precyzja wskazań odbiornika GPS zmniejszy się do tego poziomu, że robot nie będzie w stanie wykonać misji ze względu na błędy odczytu swojej pozycji. Sam teren powinien być sprawdzony pod tym kątem, czy napęd robota jest w stanie go pokonać. W ogólności należy unikać miejsc, które ograniczają widoczność satelitów. Teren również powinien być płaski oraz nie zawierać przeszkód, które mogłyby się znaleźć na trajektorii robota. Ta sprawa tyczy się również ruchu ulicznego, patrolowany obszar nie powinien mieć kontaktu z lokalnymi drogami oraz zabudowaniami.

17 Eksperymenty

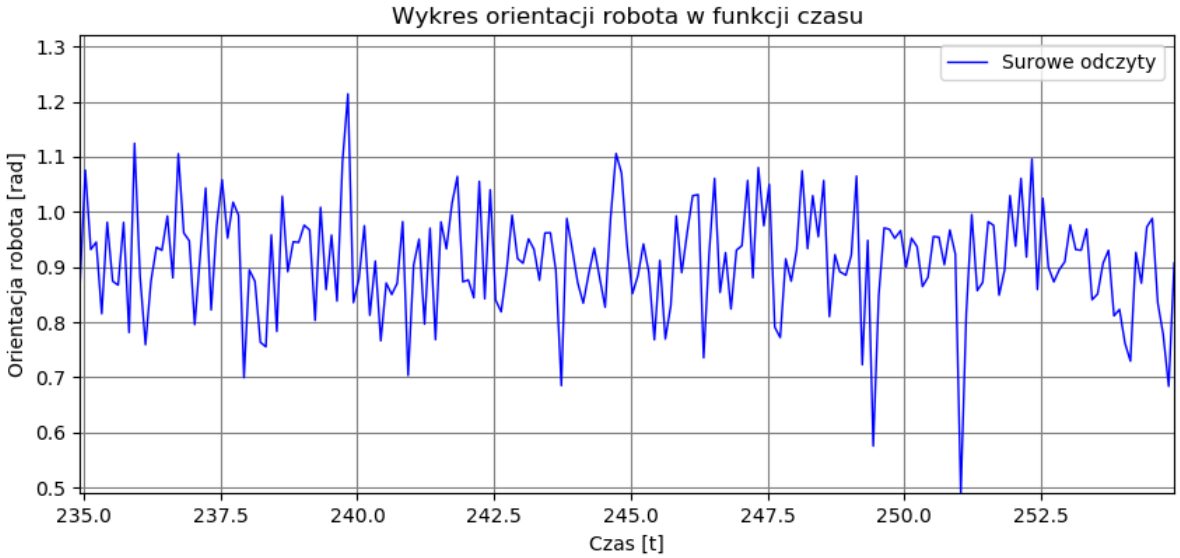
17.1 Wstęp

W tym rozdziale przedstawiono wykonane implementacje algorytmów filtrujących oraz ich analizę wyników. Mają one za zadanie rozwiązać problemy opisane w rozdziałach 17.2.1 oraz 17.3.

17.2 Niedokładność pomiarów bieżącej orientacji robota

17.2.1 Przebieg danych odczytanych z kompasu

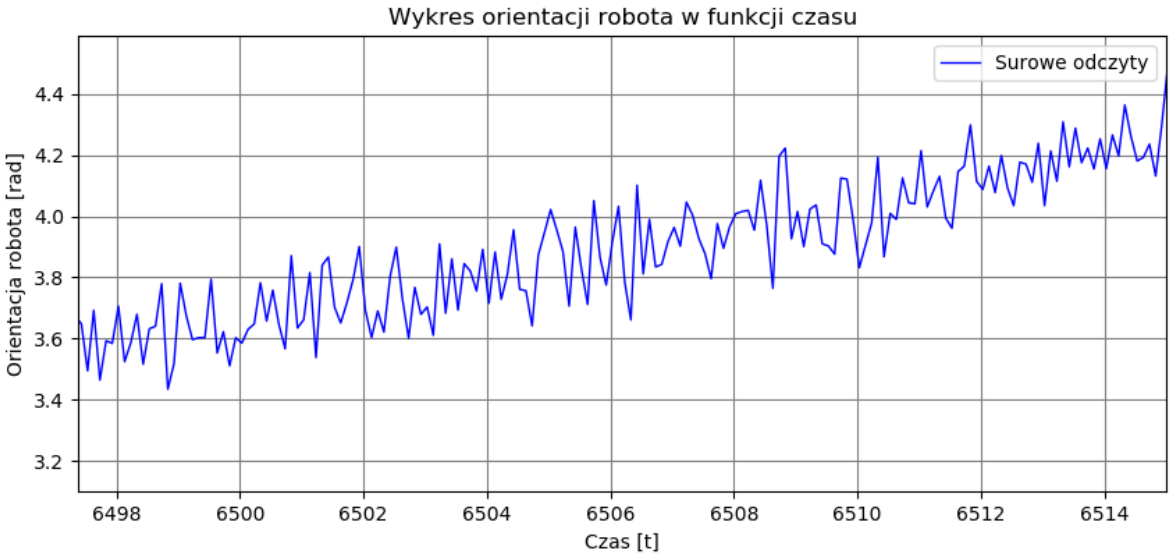
W tym rozdziale została wykonana analiza surowych danych z kompasu. Poniżej przedstawiono przykładowy wykres zależności absolutnej orientacji robota od czasu. Czas jest mierzony od początku uruchomienia symulatora Gazebo. Wykres obecny na rysunku 17 jest pewnym odcinkiem czasowym wyjętym z symulacji.



Rysunek 17: Przykładowy wykres odczytów orientacji robota podczas jego spoczynku

Dane na wykresie charakteryzują się obecnością stabilnego szumu wzbogaconego o piki, które pojawiają się w losowych chwilach. Otrzymane dane charakteryzują się odchyleniem standardowym około 0,2 radiana. To oznacza, że precyzja obrotu wynosi około 23^o. Takie zachowanie stwarza niepożądaną sytuację, która polega na tym, że algorytmy wchodzące w skład oprogramowania autopilota nie są w stanie spełniać swoich zadań, w sposób umożliwiający bezproblemowe działanie autopilota. W rzeczywistości szum może być spowodowany przez czynniki opisane w rozdziałach 9.2 oraz 9.1.

Przyjrzyjmy się teraz danym zebranym podczas powolnego obrotu robota.



Rysunek 18: Przykładowy wykres odczytów orientacji robota podczas jego obrotu

Odczytując surowe dane w kompasu podczas obrotu nie można być do końca pewnym co do orientacji robota w punkcie czasowym t_i. Patrząc na powyższy wykres można zauważyć, że mimo obrotu ze stałą prędkością kątową, otrzymane dane miejscami wskazują na to, że robot wręcz obrócił się na chwilę w przeciwną stronę. Algorytm precyzyjnego obrotu może zakończyć swoje działanie w nieoczekiwanej chwili. Robot może nawet przeskoczyć poszukiwaną wartość kąta i robot może wykonać obrót o dodatkowe 360^o.

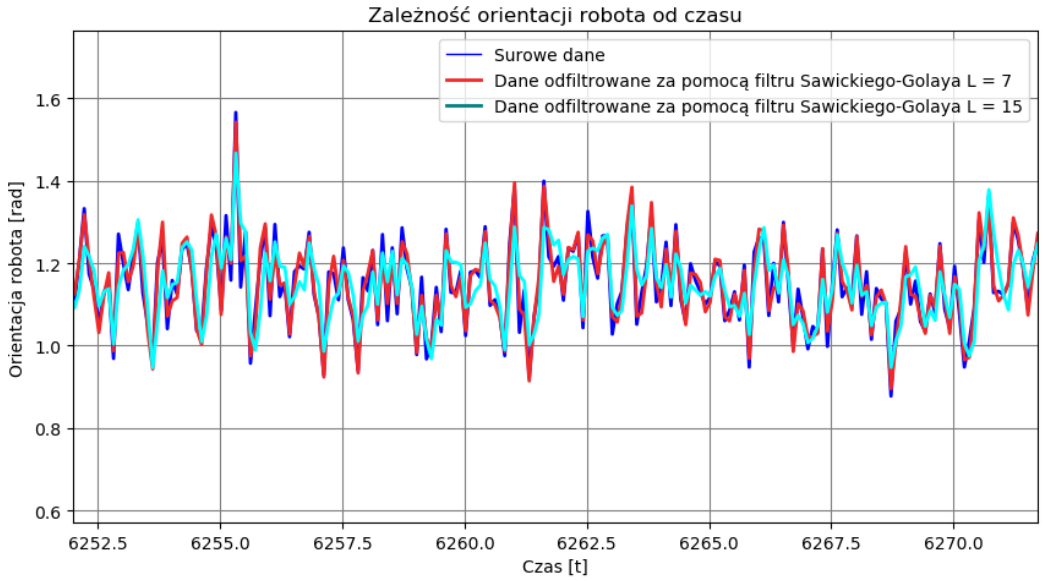
Dane z kompasu należy odfiltrować z pomocą dostępnych filtrów. Inaczej zachowanie zaimplementowanych algorytmów opisanych w rozdziale 13 będzie nieprzewidywalne. Przetestowano w tym celu 3 rodzaje filtrów wygładzających: filtr średniej ruchomej, filtr Sawickiego-Golaya oraz filtr Kalmana. Dokonano analizy wyników z wyjść z poszczególnych filtrów oraz wybrano filtr, którego działanie wprowadza jak najwięcej efektów pozytywnych na działanie autopilota przy czym nie wprowadzą efektów, które spowodują niestabilne zachowanie algorytmów autopilota.

17.2.2 Filtracja odczytów orientacji robota filtrem Sawickiego-Golaya

Wykorzystując bibliotekę **scipy.signal** użyto gotowej implementacji filtru Sawickiego-Golaya. Zaczerpnięto surowe dane z kompasu i przefiltrowano je wspomnianym filtrem. Zastosowano tutaj dopasowanie wielomianowe za pomocą wielomianu 3 stopnia. Pomiary zostały wykonane dla filtrów o następujących długościach:

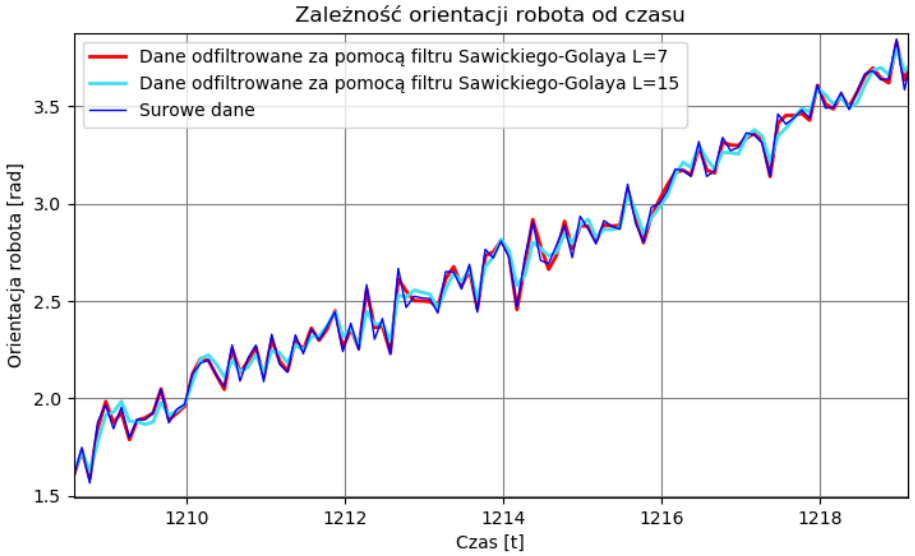
- $L = 7$ próbek,
- $L = 15$ próbek,
- $L = 35$ próbek,
- $L = 51$ próbek.

By wykresy były czytelne oraz analiza była wygodniejsza, to pomiary podzielono na dwie grupy względem liczby próbek. Na jednym wykresie umieszczono pomiary na $L = 7$ i $L = 15$ oraz $L = 35$ i $L = 51$.



Rysunek 19: Wykres odczytów orientacji robota podczas jego spoczynku dla filtru Sawickiego-Golaya, L=7 i L=15

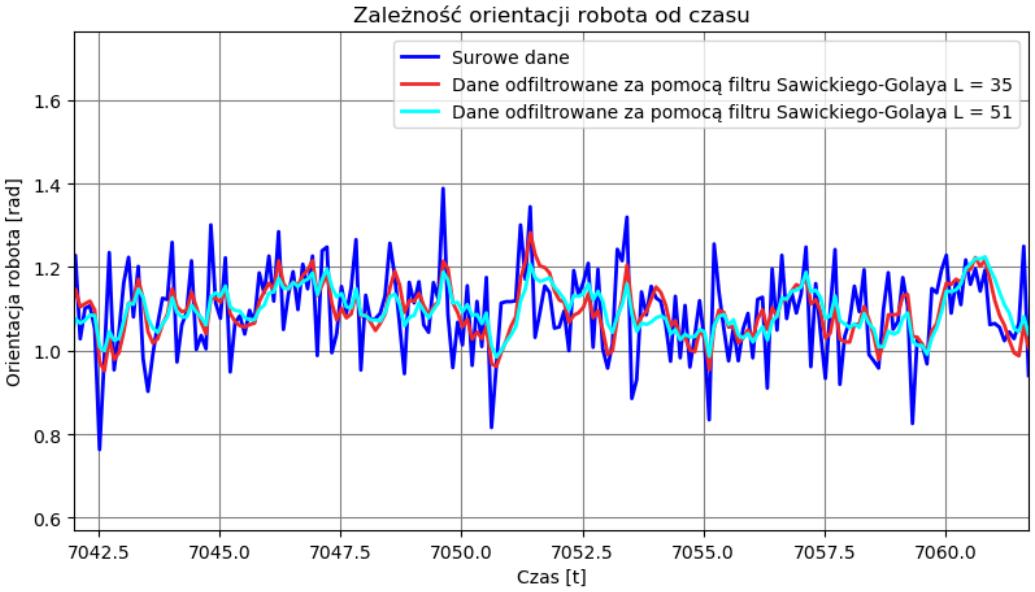
Odnosząc się do wykresu obecnego na rysunku 19 można zauważyć, że filtr sprawdził się niewystarczająco, odfiltrowane dane były prawie takie same jak oryginalne. Trzeba było zastosować powiększenie wykresu, by zobaczyć jakąkolwiek różnicę. Dane otrzymane z wyjścia filtru prawie się pokrywają. Częstotliwość zmian wartości danych z filtra jest niemal taka sama jak danych surowych. Taki wynik z wyjścia filtra spowodowany jest wielomianowym charakterem filtru (jego sposobem dopasowania do danych surowych). Uzyskany wynik jest niezadowalający, jest to spowodowane zbyt wąskim oknem pomiarowym. Użycie tej kombinacji (filtru Sawickiego-Golaya wraz z określonymi długościami okien pomiarowych) uniemożliwia działanie jakiegokolwiek algorytmu ze względu na zbyt duże skoki wartości.



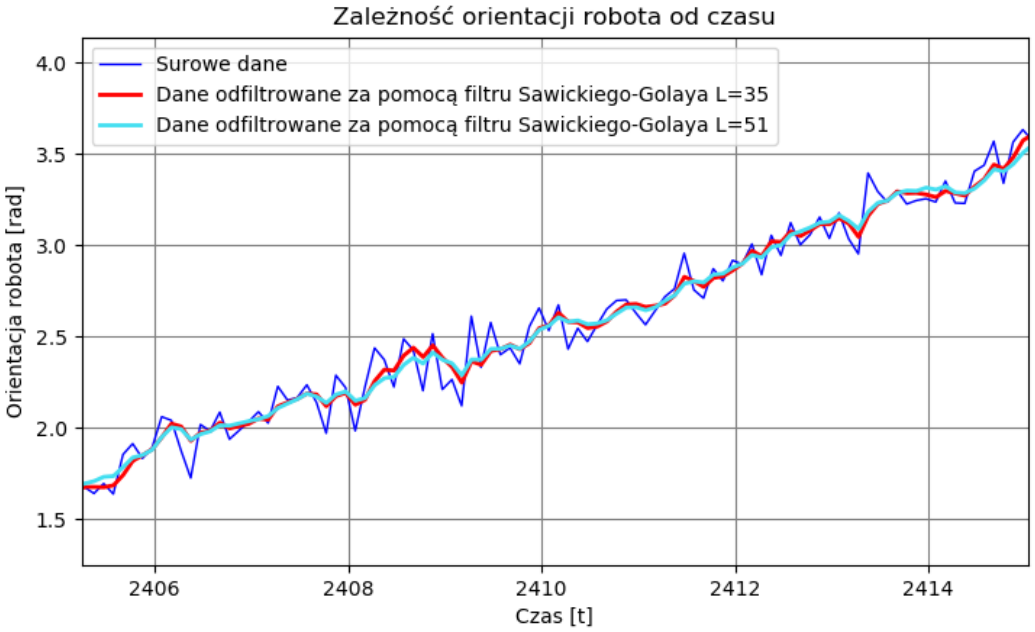
Rysunek 20: Wykres odczytów orientacji robota podczas jego obrotu dla filtru Sawickiego-Golaya, L=7 i L=15

W przypadku wykresu z rysunku 20 można zauważyć, że podczas obrotu odfiltrowywane dane były niemal takie same jak oryginalne. Podczas obrotu dane z filtru charakteryzują się brakiem opóźnienia, co akurat jest pożądaną cechą poszukiwanego filtru. Wyjście z filtra dla $L = 15$ pozwala jedynie pozbyć się ostrych pików z danych surowych.

Ostateczny wniosek co do użycia filtru Sawickiego-Golaya przy długościach okien pomiarowych $L = 7$ oraz $L = 15$ jest taki, że ten filtr w tej kombinacji nie jest w stanie sprostać wymaganiom dotyczących działania autopilota. Zbyt mocno dopasowuje się do oryginalnych danych nie dając satysfakcjonujących wyników.



Rysunek 21: Wykres odczytów orientacji robota podczas jego spoczynku dla filtru Sawickiego-Golaya, L=35 i L=51



Rysunek 22: Wykres odczytów orientacji robota podczas jego obrotu dla filtru Sawickiego-Golaya, L=35 i L=51

Nawiązując do wykresów 21 oraz 22 można zauważyć silniejsze wygładzenie na wyjściu z filtra względem poprzednich konfiguracji. Częstotliwość zmian również uległa redukcji względem filtrów o krótszych oknach pomiarowych. Tutaj odfiltrowane dane wydają się łagodniejsze niż na wykresach 20 lub 19. Nadal można zauważyć brak opóźnienia mimo tego, że okno pomiarowe zostało zwiększone. Można z tego wywnioskować, że modyfikacja długości okna pomiarowego nie wpływa na opóźnienie na wyjściu z tego filtra. Piki zostały wygładzone dla filtrów o długościach $L = 35$ oraz $L = 51$.

Dla wszystkich rozpatrywanych okien pomiarowych, ten filtr cechuje się niewystarczającą skutecznością podczas odfiltrowywania absolutnej orientacji robota. Zwiększenie okna próbek pomiarowych nie zwiększyło znacząco jakości filtracji szumów.

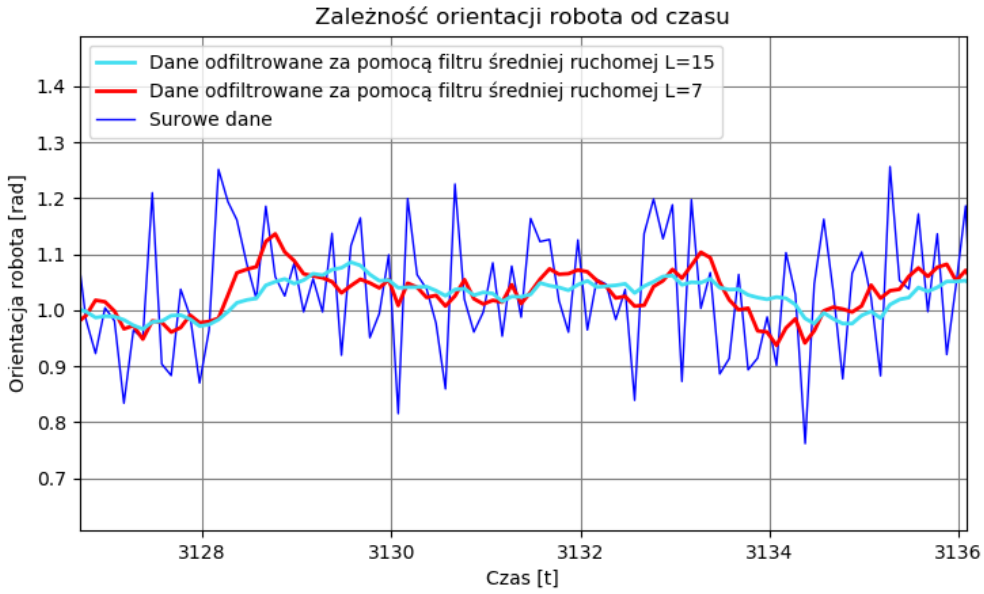
17.2.3 Filtracja odczytów orientacji robota filtrem średniej ruchomej

Filtr średniej ruchomej zaimplementowano według wzorów opisanych w rozdziale 10.1. Implementację wykonano w języku Python. Następnie wykonano pomiary orientacji w spoczynku oraz podczas obrotu jednocześnie odczytując przefiltrowane wartości orientacji.

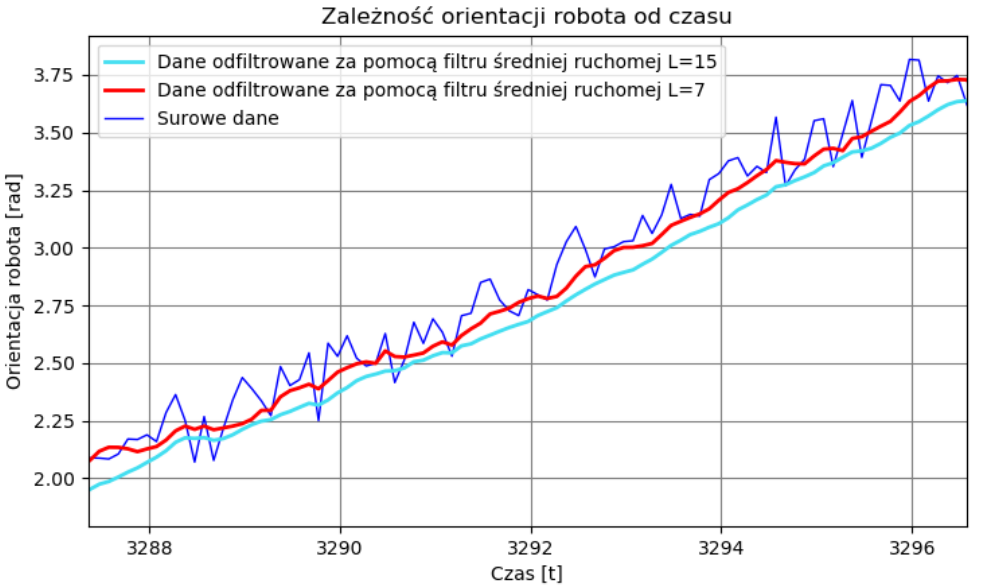
Pomiary zostały wykonane dla filtrów o następujących długościach:

- $L = 7$ próbek,
- $L = 15$ próbek,
- $L = 35$ próbek,
- $L = 51$ próbek.

Aby wykresy były czytelne oraz analiza była wygodniejsza, to pomiary podzielono na dwie grupy względem liczby próbek. Na jednym wykresie umieszczono pomiary na $L = 7$ i $L = 15$ oraz $L = 35$ i $L = 51$.

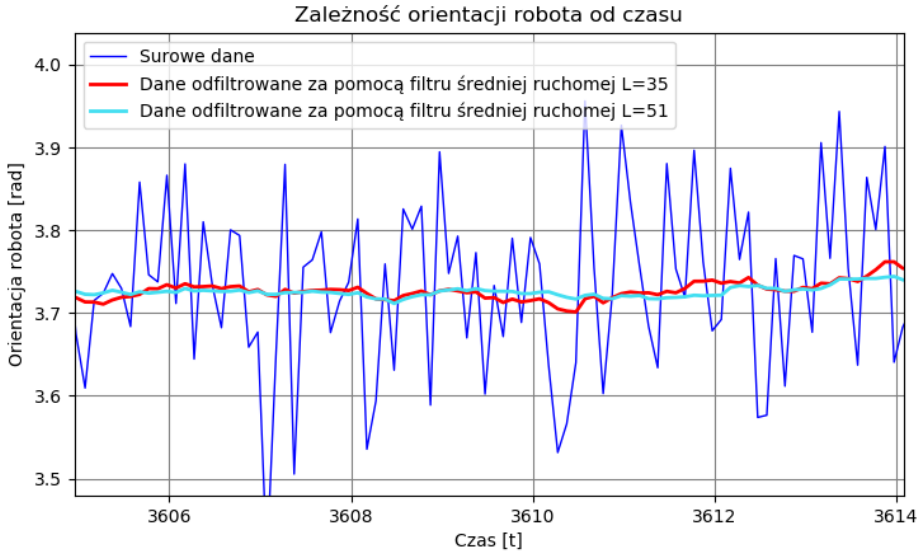


Rysunek 23: Wykres odczytów orientacji robota podczas jego spoczynku dla filtru średniej ruchomej, L=7 i L=15

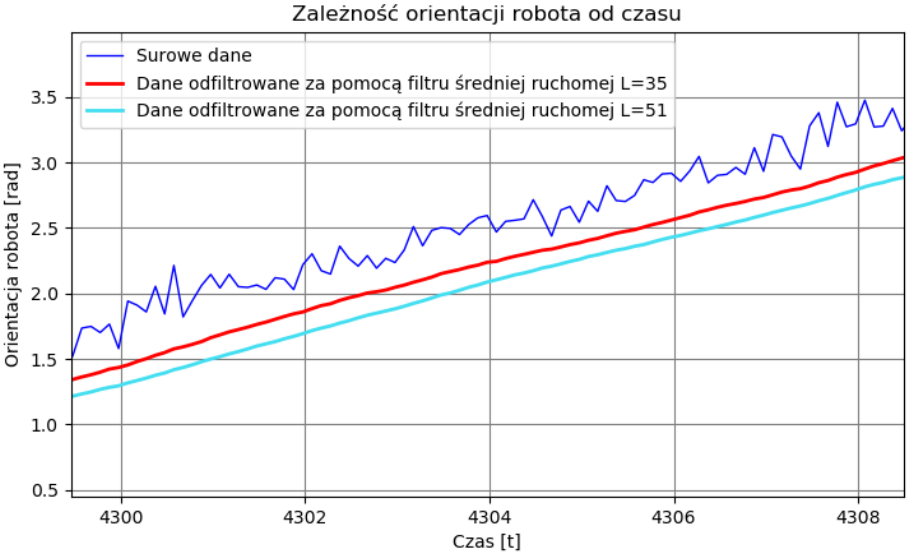


Rysunek 24: Wykres odczytów orientacji robota podczas jego obrotu dla filtru średniej ruchomej, L=7 i L=15

Nawiązując do wykresów 23 oraz 24 można zauważyć, że dane otrzymane na wyjściu z filtra są wygładzone w większym stopniu oraz częstotliwość zmian jest niższa niż przy zastosowaniu filtra Sawickiego-Golaya dla okna pomiarowego o tej samej długości. Podczas obrotu jakość filtracji jest podobna do tej, gdy robot jest nieruchomy.



Rysunek 25: Wykres odczytów orientacji robota podczas jego spoczynku dla filtru średniej ruchomej, L=35 i L=51

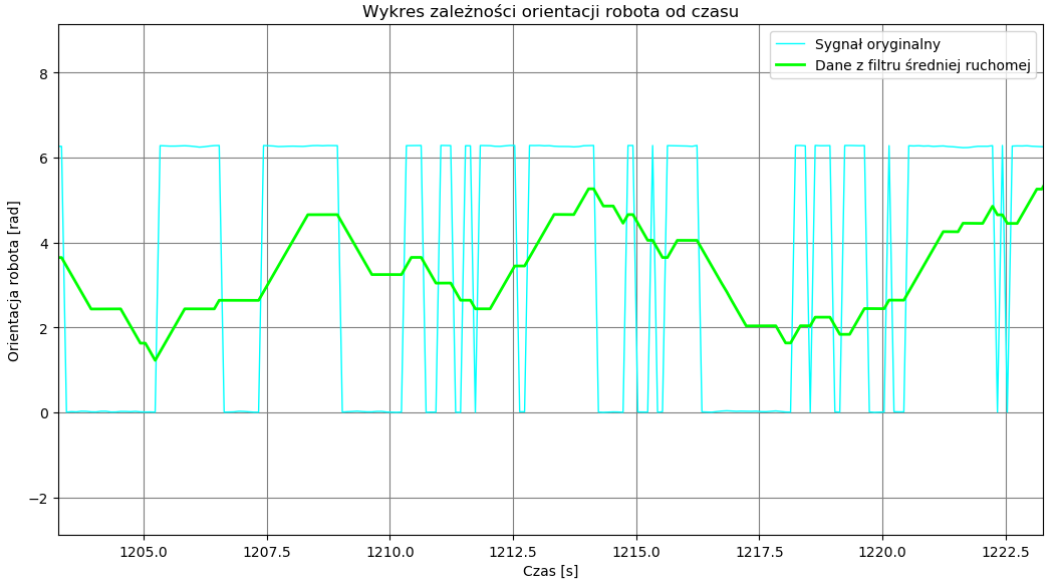


Rysunek 26: Wykres odczytów orientacji robota podczas jego obrotu dla filtru średniej ruchomej, L=35 i L=51

Nawiązując do wykresów 25 oraz 26 tutaj dane zostały wygładzone mocniej aniżeli w poprzedniej próbie oraz ich zachowanie jest stabilne. Dane wyjściu z tak skonfigurowanego filtra charakteryzują się znikomą tendencją do zmian lokalnych. Stosowanie szerszych okien pomiarowych nie przyniesie już znaczącego wzrostu wygładzenia. Zauważono, że wraz ze wzrostem szerokości okna pomiarowego, opóźnienie zwiększa się.

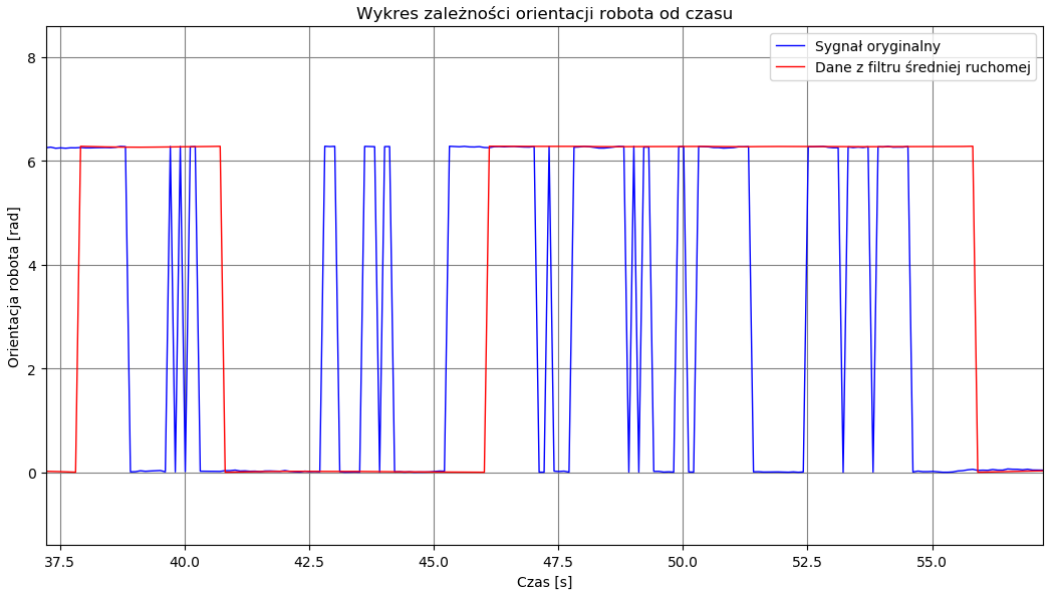
Jednak patrząc na otrzymane wyniki z tego filtra (np. rys. 26), można zauważyć ,że są opóźnione względem danych surowych. Jest to poważny problem, gdyż jest w stanie spowodować wprowadzeniem błędu rzędu kilku stopni. Jest to spowodowane tym, że każda próbka ma tę samą wagę, a nie powinno tak być, gdyż podczas obrotu możliwe są poślizgi kół robota, czyli przez ułamek sekundy prędkość kątowa będzie inna.

Ten filtr jest lepszym rozwiązaniem niż filtr Sawickiego-Golaya, jednak nie jest idealny. Przyrost redukcji szumów, przy jednoczesnym zwiększaniu rozmiaru okna pomiarowego jest większy niż w przypadku filtru Sawickiego-Golaya. Dużą wadą filtru średniej ruchomej jest to, że uśrednia on dane bezkrytycznie, nie stosuje on żadnych wag do poszczególnych próbek danych. Ta cecha ujawnia się gdy robot jest skierowany skierowany na północ, gdzie jego orientacja odczytana z kompasu przeskakuje pomiędzy 0, a 2π. Zachowanie filtru w tej sytuacji nie jest deterministyczne.



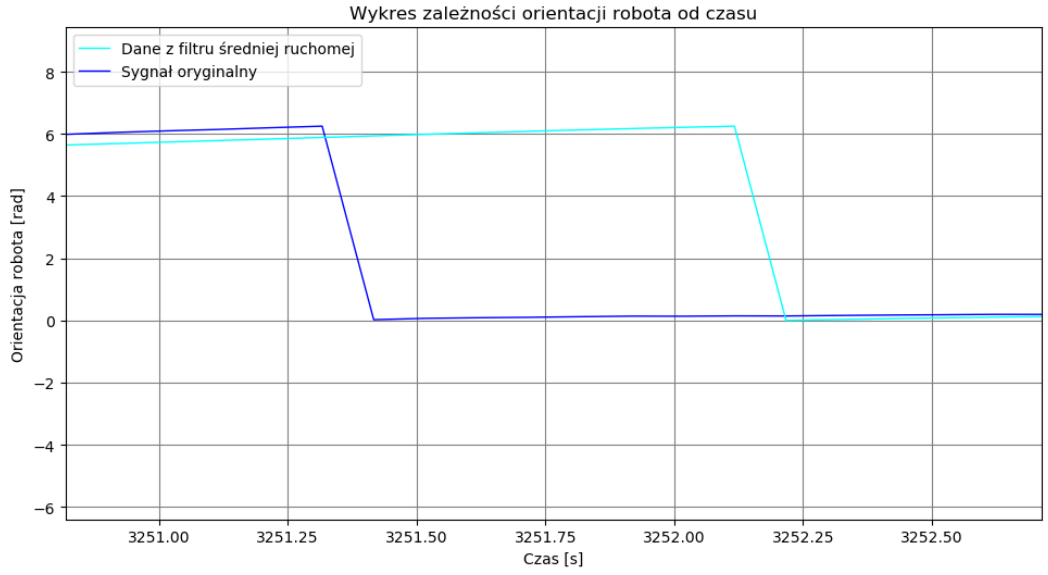
Rysunek 27: Wykres pokazujący niedeterministyczne zachowanie filtru średniej ruchomej

Nawiązując do wykresu 27 można zauważyć, ta właściwość sprawia, że w takiej sytuacji orientacja robota jest losowa. By sobie z tym poradzić, to na wejściu filtra orientacja z kompasu zostaje zamieniona na liczbę zespoloną o module 1, a na wyjściu z powrotem należy ją przekonwertować na orientację wyrażoną w radianach.



Rysunek 28: Wykres pokazujący deterministyczne zachowanie filtru średniej ruchomej po korekcie wejścia

Nawiązując do wykresu 28 zostało pokazane, że problem przeskakującej orientacji robota został rozwiązany. Jednak pozostał jeszcze jeden problem. Gdy robot obraca się szybko, wyjście filtru nie nadąża za sygnałem oryginalnym jak pokazano na rysunku 29.



Rysunek 29: Wykres pokazujący problemy z nadążaniem dla filtru średniej ruchomej po korekcie wejścia

By rozwiązać problem pokazany na rysunku 29 istnieje potrzeba, by filtr średniej ruchomej uczynić dynamicznym. Chodzi o to, że szerokość okna pomiarowego powinna zmieniać się wraz ze zmianą prędkości kątowej robota. Robot ma maksymalną dostępną prędkość kątową ustawioną na $\omega_{max} = 1\frac{rad}{s}$. Należy zauważyć, że dokładność jest potrzebna przy niskich prędkościach, tam, gdzie położenie robota się stabilizuje. Należy więc wyznaczyć wyrażenie pełniące rolę mnożnika maksymalnej długości filtra, by otrzymać aktualną długość filtra. Na podstawie przeprowadzonych prób, podczas których analizowano wymagany stopień zmian długości filtra w czasie, wyznaczono poniższy wzór:

$$L = L_{max}(\frac{\omega_{max} - \omega_{current}}{\omega_{max}})^2 \tag{43}$$

gdzie:
L_{max} - maksymalna długość filtra
ω_{max} - maksymalna prędkość kątowa
$\omega_{current}$ - aktualna prędkość kątowa robota

17.2.4 Filtracja odczytów orientacji robota filtrem Kalmana

Tutaj zastosowano możliwość skorzystania z pomiarów z kilku czujników, które mogą zapewnić pomiary tej samej wielkości. Do odczytu aktualnej orientacji robota wykorzystamy 2 czujniki.

- Wskazania wektora pola magnetycznego otrzymane z magnetometru.
- Wskazania prędkości kątowej robota wokół osi **z** otrzymane z żyroskopu zawartego w IMU.

Istnieje możliwość skorzystania z kilku czujników w celu estymacji orientacji w danym punkcie czasu. W tym celu zaimplementowano rozwiązanie wykorzystujące jednowymiarowy filtr Kalmana oraz fuzję wskazań czujników (ang. *sensor fusion*), po to aby z odczytów z dwóch czujników otrzymać jeden, lecz lepszej jakości. Fuzję czujników opisano w rozdziale 10.3. Jednowymiarowy filtr Kalmana opisano w rozdziale 10.2.

Orientacja robota jest bezwzględna (względem układu zerowego lokalnego przedstawionego w rozdziale 12.1) i jest wyznaczana w radianach. Zakres wartości dla orientacji robota wynosi $0; 2\pi$. Model jest liniowy i zmiana jego stanu jest również liniowa. Dane, jakie podlegają pomiarom to prędkość kątowa uzyskana z żyroskopu oraz wskazania magnetometru, zatem etap predykcji następnego stanu składa się ze wzorów jak poniżej. Te wzory są konkretyzacją wzorów opisanych w rozdziale 10.2 i zostały użyte to implementacji filtru.

$$\phi_{n+1,n} = \phi_{n,n} + \Delta t \cdot \phi_{gyro} \tag{44}$$

$$p_{n+1,n} = p_{n,n} + q_n \tag{45}$$

gdzie:
$\phi_{n,n}$ - aktualna estymacja orientacji,
$\phi_{n+1,n}$ - prognoza estymacji orientacji,
ϕ_{gyro} - odczyt prędkości kątowej z żyroskopu,
$p_{n+1,n}$ - prognoza niepewności estymacji orientacji,
$p_{n,n}$ - aktualna niepewność estymacji orientacji,
q_n - szum procesowy.

Następnie zostały przedstawione równania aktualizacji stanu

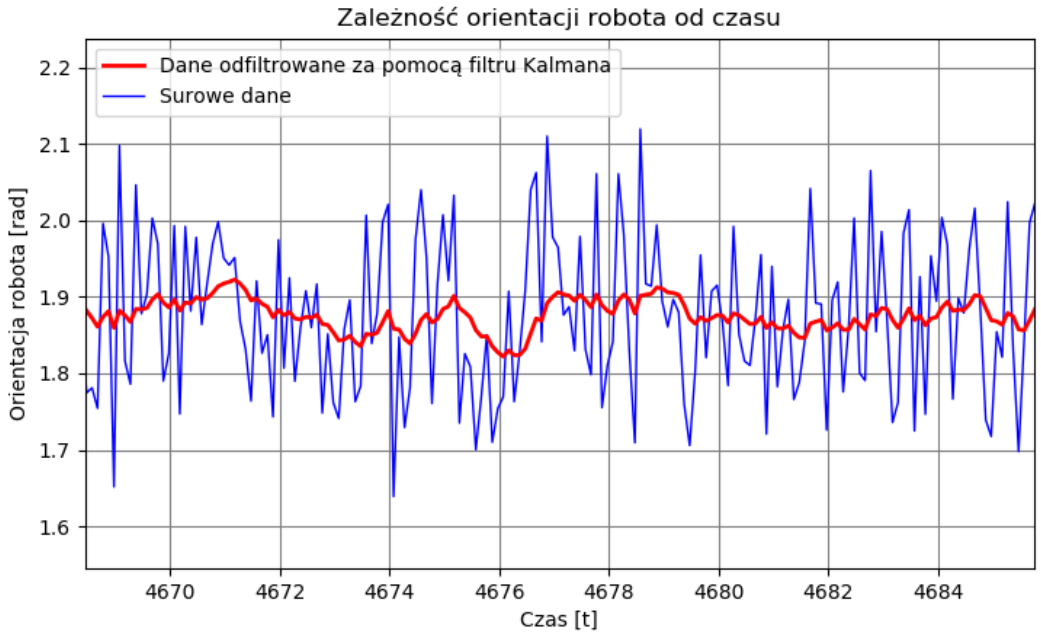
$$\hat{\phi}_{n,n} = \hat{\phi}_{n,n-1} + K_n(\phi_N - \hat{\phi}_{n,n-1}) \tag{46}$$

$$K_n = \frac{p_{n,n-1}}{p_{n,n-1} + r_n} \tag{47}$$

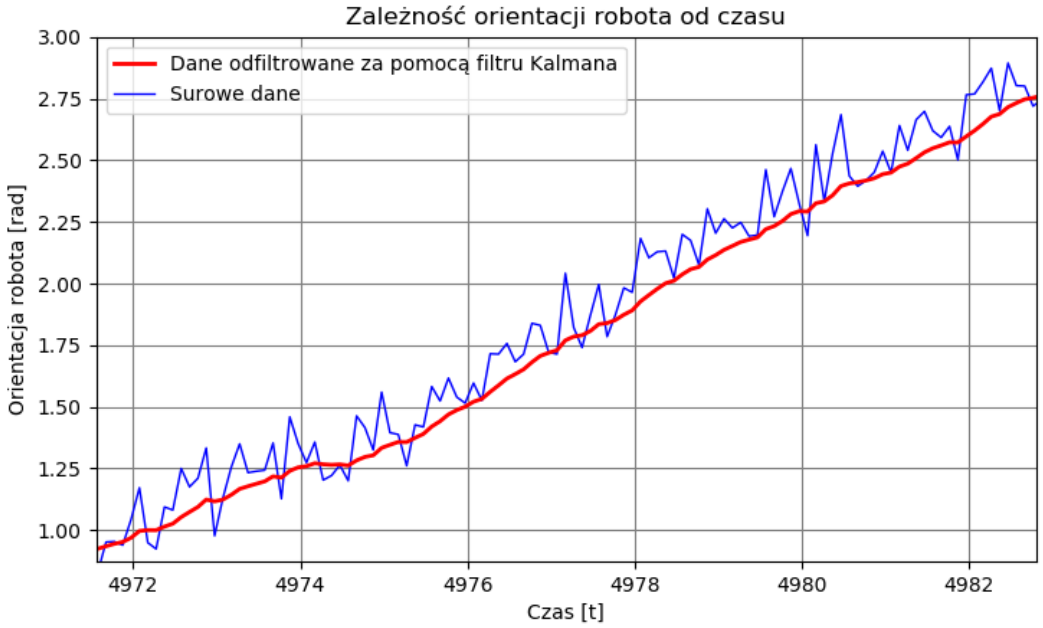
$$p_{n,n} = (1 - K_n)p_{n,n-1} \tag{48}$$

gdzie:
$\phi_{n,n-1}$ - poprzednia estymacja orientacji,
ϕ_N - przychodzący pomiar orientacji z magnetometru,
r_n - niepewność pomiaru.

Przyjęto wartość szumu procesowego q_n jako 0.01 rad. Niepewność pomiaru r_n przyjęto jako 1 rad. Po uruchomieniu robota, ten algorytm jest uruchomiony przez cały czas, a jego wywoływanie się wyzwalane jest przychodzącymi pomiarami z żyroskopu i magnetometru. Ten model nie jest idealny, gdyż nie uwzględnia przyspieszenia kątowego, ale jest to wystarczające.



Rysunek 30: Wykres odczytów orientacji robota podczas spoczynku dla filtru Kalmana



Rysunek 31: Wykres odczytów orientacji robota podczas obrotu dla filtru Kalmana

Nawiązując do wykresów 30 oraz 31 można zauważyć, że otrzymano stopień wygładzenia porównywalny do wyników otrzymanych z filtru średniej ruchomej dla okna pomiarowego o długości $L = 35$. Wyniki te charakteryzuje niska częstotliwość zmian, ale trendy globalnie są zgodne z danymi surowymi. Filtr Kalmana sprawdził się w sposób wystarczający oraz nie wprowadza żadnych opóźnień względem danych surowych.

17.3 Niedokładność wskazań odbiornika GPS

Aby poradzić sobie z problemem niedokładności wskazań GPS, zaimplementowano wielowymiarowy filtr Kalmana na podstawie wzorów przedstawionych w rozdziale 10.5. Mierzone są tutaj tylko dwie współrzędne **x** oraz **y** (po transformacji). Zatem wektor stanu będzie wyglądać następująco:

$$\boldsymbol{X} = \begin{bmatrix} x \\ y \end{bmatrix}$$

Stan początkowy będzie przyjęty jak poniżej. Poniższe macierze są zerowe. Przyjęto je w taki sposób dlatego, że nie znano docelowych wartości ich elementów przed rozpoczęciem pomiaru.

$$\boldsymbol{X} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

$$\boldsymbol{P} = \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix}$$

Macierz przejścia nie modyfikuje danych wejściowych, zatem jest macierzą jednostkową o wymiarze 2x2 ze względu na obecność dwóch filtrowanych zmiennych.

$$\boldsymbol{F} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

Zdefiniowano macierze, które są stałe i z góry znane podczas obliczeń. Poniżej zdefiniowano macierz obserwowalności. Obserwacji podlega pomiar wartości współrzędnych kartezjańskich x oraz y z konwertera GPS (po transformacji).

$$\boldsymbol{H} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

Macierz wejścia oraz wektor wejścia są zerowe. Jest to spowodowane brakiem wejścia do układu, które wpływa w sposób deterministyczny na wskazania GPS.

$$\boldsymbol{G} = \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix}$$

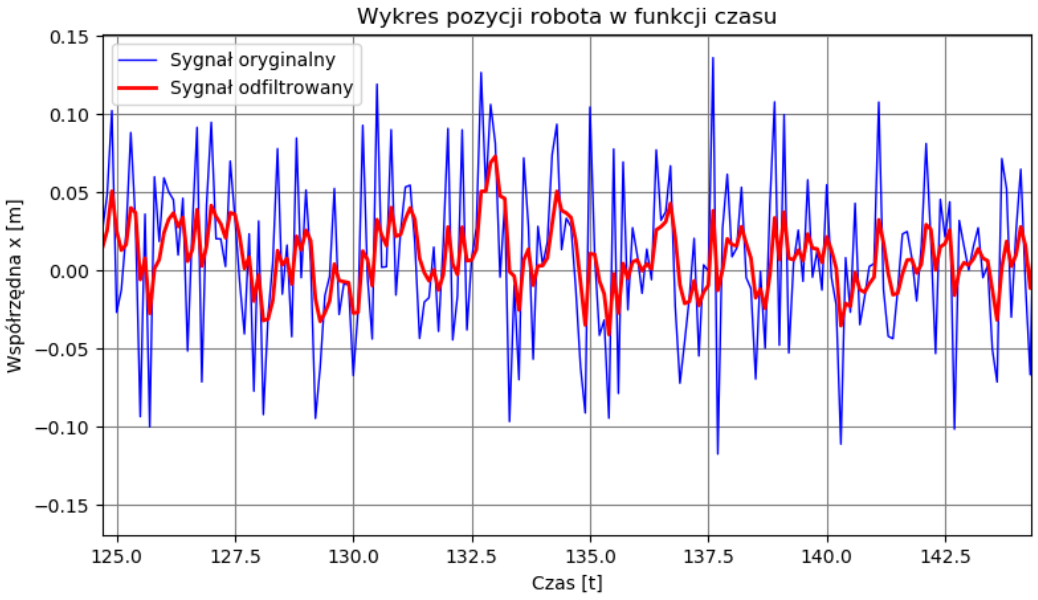
$$\boldsymbol{u}_n = \begin{bmatrix} 0 \end{bmatrix}$$

Macierz niepewności pomiarowej przyjęto eksperymentalnie jak poniżej. Przyjęto wstępnie niepewność pomiarową jako 7 cm dla wartości x oraz y.

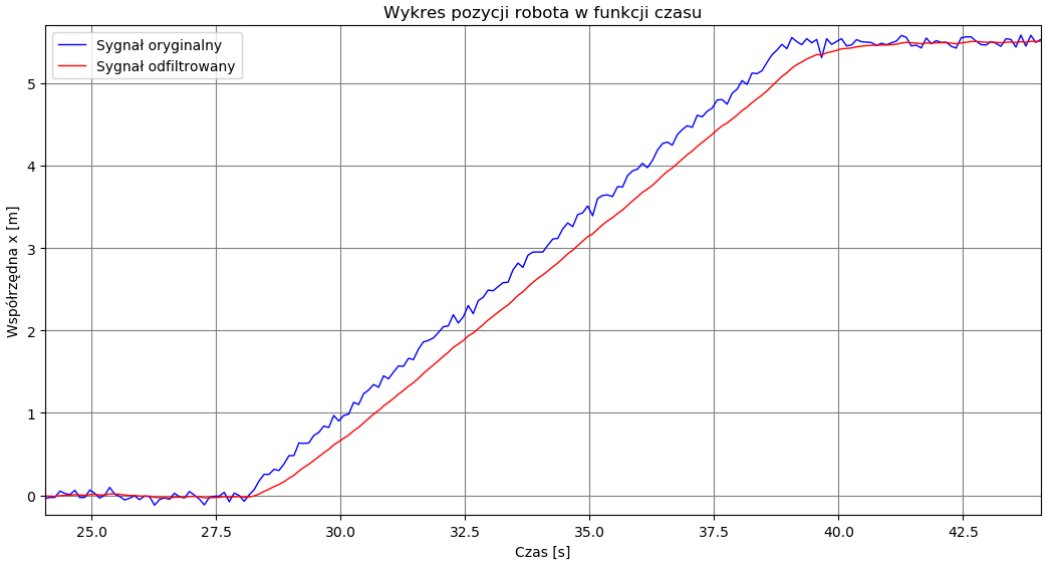
$$\boldsymbol{R}_n = \begin{bmatrix} 0.005 & 0 \\ 0 & 0.005 \end{bmatrix}$$

Macierz szumu procesowego również dobrano doświadczalnie. Szum procesowy można początkowo przyjąć jako 10 mm dla wartości x oraz y.

$$\boldsymbol{Q} = \begin{bmatrix} 0.0001 & 0 \\ 0 & 0.0001 \end{bmatrix}$$



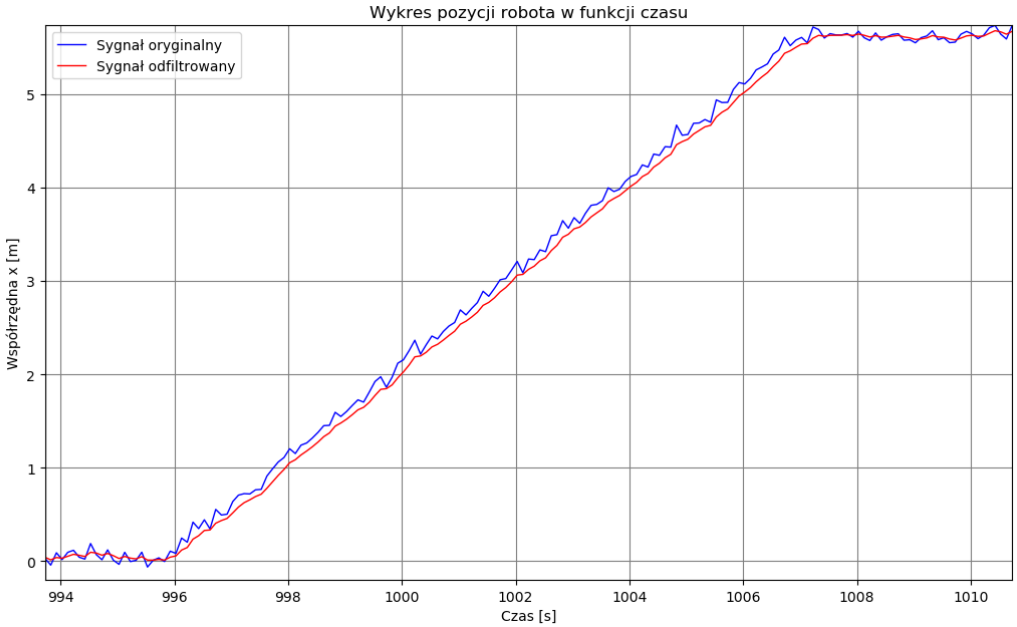
Rysunek 32: Wykres pozycji robota przy zastosowaniu filtru Kalmana



Rysunek 33: Wykres pozycji robota podczas jego ruchu przy zastosowaniu filtru Kalmana

Patrząc na wykres z rysunku 32 można zauważyć, że przebieg danych odfiltrowanych został wygładzony względem sygnału oryginalnego. Jednak na wykresie obecnym na rysunku 33 można zauważyć, że podczas przemieszczania się wyjście filtru nie nadąża za sygnałem oryginalnym. Jest to spowodowane zbyt niską wartością szumu procesowego, jaka została przyjęta. By zmniejszyć opóźnienie, należy zwiększyć wartość elementów macierzy szumu procesowego. Przyjęto teraz szum procesowy równy wartości około 3 mm dla wielkości x oraz y.

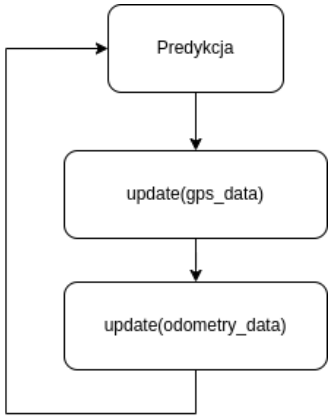
$$\boldsymbol{Q} = \begin{bmatrix} 0.0008 & 0 \\ 0 & 0.0008 \end{bmatrix}$$



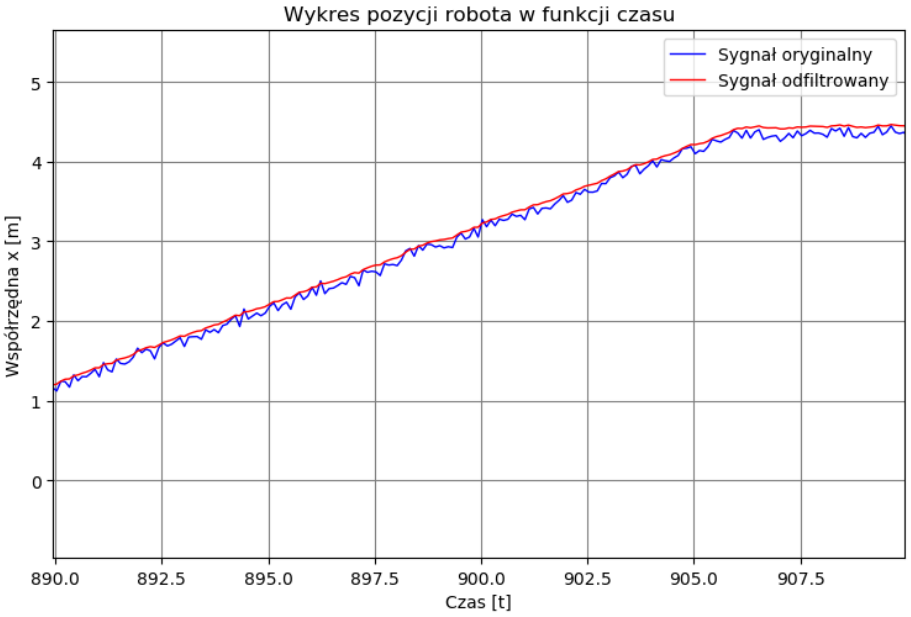
Rysunek 34: Wykres pozycji robota podczas jego ruchu przy zastosowaniu filtru Kalmana bez opóźnienia w nadążaniu za sygnałem oryginalnym

Patrząc na wyniki z wykresu 34 można zauważyć, że filtr szybciej nadąża za sygnałem oryginalnym niż w przypadku pokazanym na rysunku 33. Zwiększenie szybkości nadążania jest obarczone pogorszeniem jakości filtracji, co jest możliwe do zaakceptowania.

Warto się przyjrzećm, jak dołożenie pomiarów z odometrii wpłynie na dokładność uzyskanych pomiarów położenia robota. Macierze stałych są tożsame z macierzami użytymi w obliczeniach dotyczących GPS. Cały algorytm został wzbogacony o dodatkowy etap aktualizacji pomiarów związanym z odometrią. Czyli ostatecznie zastosowane tutaj zostały dwa etapy aktualizacji pomiarów, jak pokazano na rysunku 35.



Rysunek 35: Schemat algorytmu fuzji odometrii i wskazań GPS



Rysunek 36: Wyniki fuzji odometrii i wskazań GPS

Odwołując się do wykresów 34 oraz 36 należy zauważyć, że jakość filtracji jest podobna. Dlatego nie istnieje potrzeba, by do oprogramowania autopilota wyposażać w drugi etap aktualizacji danych.

18 Testy oprogramowania

18.1 Przeprowadzone testy

W celu przetestowania i zweryfikowania poprawności działania oprogramowania przygotowano dwa testy. Elementy oprogramowania są dobrane wg opisów w rozdziałach 14 oraz 13. Testy przeprowadzono w środowisku symulacyjnym. W pierwszym teście sprawdzono ogólną poprawność działania oprogramowania. W drugim teście natomiast sprawdzono dokładność dojazdu.

Podczas pierwszego testu sprawdzono poprawność działania oprogramowania. Trasę pierwszego testu można zobaczyć na rysunku 37. Robot ma za zadanie pokonać wyznaczoną trasę i wrócić na miejsce startowe. Robot podczas symulacji pokonał wyznaczoną trasę. Zakończył on misję w odległości 0,339 m od punktu startowego. Długość i szerokość robota wynoszą 0,5 m, zatem błąd jest mniejszy niż jego rozmiar. Pierwszy test potwierdził poprawność wykonywania misji.

Podczas drugiego testu sprawdzono dokładność dojazdu robota do poszczególnych punktów. Drugą trasę pokazano na rysunku 38. Robot również pokonał tę trasę i wrócił na miejsce startowe. W tabeli 1 pokazano wyniki tego testu. Obliczone współrzędne oznaczono jako (x, y), natomiast współrzędne, na których robot się zatrzymał, oznaczono jako (x', y'). W ostatniej kolumnie zaprezentowany został błąd pozycji. Jest to odległość między docelowym obliczonym punktem a punktem, do którego robot dojechał.

l.p	x [m]	x' [m]	y [m]	y' [m]	Błąd [m]
1	-13.358	-13.055	7.024	-7.007	0.303
2	-37.626	-37.279	-6.411	-8.789	2.403
3	-68.461	-68.607	8.661	7.785	0.888
4	-67.460	-67.935	30.007	29.746	0.542
5	-12.245	-12.661	28.370	25.422	2.977
6	-0.334	-0.056	14.253	14.997	0.794
7	0.000	0.519	0.000	0.360	0.632

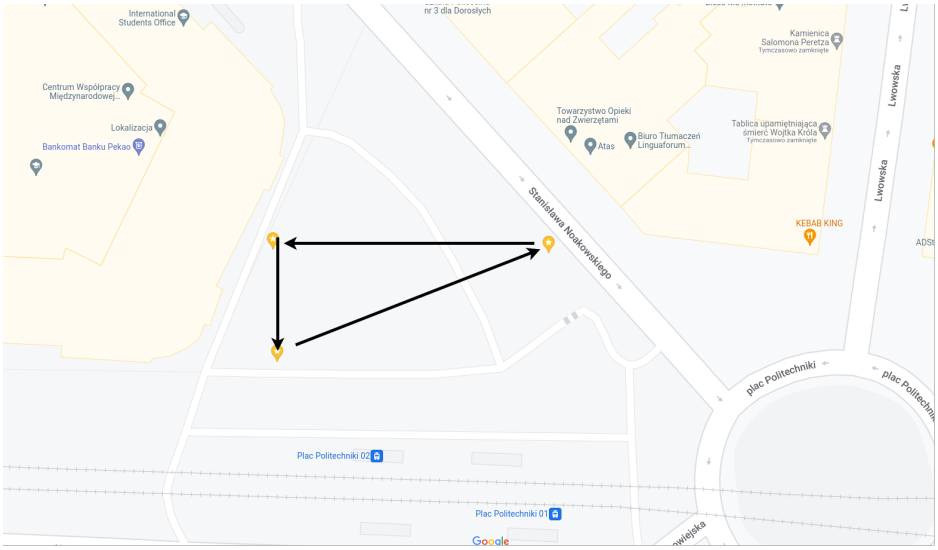
Tabela 1: Tabelka prezentująca wyniki z pierwszej misji

Patrząc na wiersze 2 oraz 5 w tabeli 1 można z zauważyć, że robot po zakończeniu dojazdu był oddalony odpowiednio około 2.5 m oraz 3 m. Takie zachowanie należy skorygować. Można to rozwiązać, modyfikując algorytm wykonywania misji w taki sposób, by po dojeździe do zadanego punktu sprawdzał on swoją odległość od obliczonego punktu. Jeśli będzie ona większa niż z góry zadana (ustalana przez użytkownika), to powinien on jeszcze raz wykonać podróż do tego samego punktu.

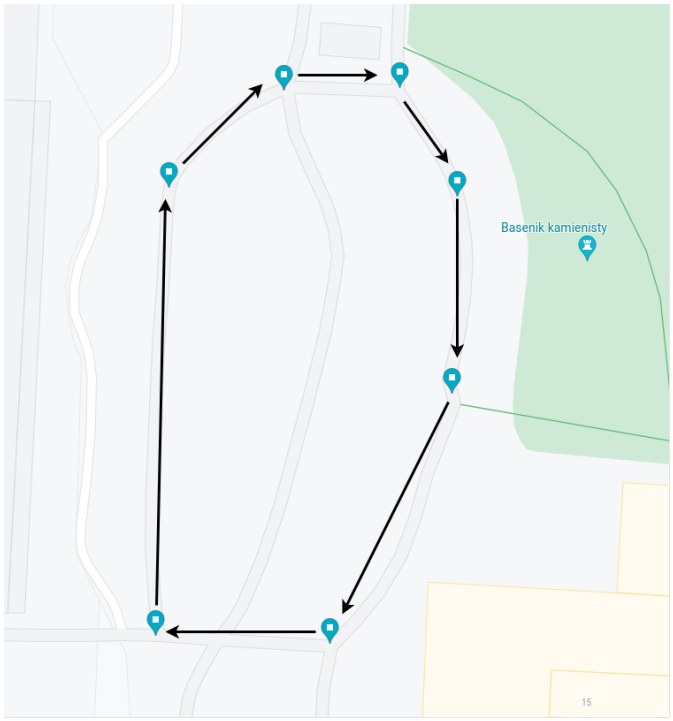
l.p	x [m]	x' [m]	y [m]	y' [m]	Błąd [m]
1	-13.358	-13.055	7.024	-7.080	0.368
2	-37.626	-37.336	-6.411	-7.425	0.300
3	-68.461	-68.332	8.661	8.937	0.305
4	-67.460	-67.718	30.007	29.721	0.385
5	-12.245	-12.159	28.370	28.059	0.323
6	-0.334	0.118	14.253	15.062	0.926
7	0.000	0.426	0.000	15.062	0.521

Tabela 2: Tabelka prezentująca wyniki z drugiej misji

W tabeli 2 zaprezentowano wyniki testu poprawionego algorytmu. W poprawce do algorytmu przyjęto założenie, że jeśli robot jest dalej niż 1 m od obliczonego punktu, to podróż do niego należy powtórzyć. Wyniki są zadowalające. W większości przypadków otrzymano błąd około 0,3 m, co jest zadowalające.



Rysunek 37: Mapa przedstawiajaca pierwszą trasę



Rysunek 38: Mapa przedstawiajaca drugą trasę

18.2 Wynik eksperymentów

Wyniki eksperymentów potwierdziły wypełnianie założeń dot. oprogramowania autopilota opisanych w rozdziale 1. Posługując się odczytami z czujników robot w sposób prawidłowy obliczył odpowiednie wektory przemieszczenia, kąty obrotu i podążał do odpowiednich punktów. Węzeł dokonujący pomiarów z kompasu w sposób prawidłowy określał orientację robota, wtedy gdy była ona wymagana. Węzeł dokonujący pomiarów pozycji również odczytał pozycję robota w sposób prawidłowy z zadaną dokładnością. Również komunikacja pomiędzy węzłami przebiegła w sposób zgodny z oczekiwaniami. Dobrane filtry również spełniły swoje zadanie w wystarczający sposób. Robot obecny w symulacji przebył całą wytyczoną trasę docierając do punktów z wymaganą dokładnością.

19 Podsumowanie

Implementacja oprogramowania takiej klasy przysporzyła kilka problemów. Również przyniosło to z sobą pewien zastrzyk wiedzy dla programistów podobnych rozwiązań.

Tworzenie symulacji również okazało się postawić kilka wyzwań. Najbardziej oczywiste było znalezienie oraz przetestowanie paczek symulujących wskazania czujników. Wytestowanych zostało kilka paczek i wybrane zostały te, które zostały wylistowane w rozdziale 15.2. ROS oraz Gazebo oferują wiele pakietów służących jako sterowniki robota. Przetestowano ich kilka, przy czym w większości z nich istniał problem ze współczynnikiem tarcia. Robot ślizgał się i nie mógł ruszyć z miejsca lub nie mógł utrzymać stabilności nawet przy znikomych przyspieszeniach.

Kolejnym problemem były obarczone szumem wskazania GPS, IMU oraz magnetometru. Repozytorium paczek ROS-a dostarcza kilka gotowych filtrów, które wystarczy tylko skonfigurować, jednak ich działanie pozostawiało wiele do życzenia, dlatego wszystkie wymagane filtry (oprócz filtru Sawickiego-Golaya) należało zaimplementować samodzielnie, co zapewniło dodatkowo przejrzystość tego rozwiązania.

Zaletami przedstawionego rozwiązania są prostota oraz zintegrowanie z ROS-em, który dostarcza wiele użytecznych paczek. Kolejną zaletą tego rozwiązania jest modularność, jeśli chcemy zmienić fragment kodu, to ta zmiana nie będzie mieć wpływu na inne elementy oprogramowania.

Wadą tego rozwiązania jest to, że wymaga ono systemu operacyjnego. Takie oprogramowanie można byłoby zaimplementować na mikrokontrolerze np. STM32. Obecnego oprogramowania nie można przenieść na mikrokontrolery bez implementacji go od nowa. Wymagałoby to mniej zaawansowanych technicznie komponentów elektronicznych. Kolejną wadą jest brak omijania przeszkód, gdy robot natrafi na przeszkodę, to wtedy zderzy się z nią. By ją usunąć, to robot przykładowo mógłby być wyposażony w czujnik laserowy wykrywający przeszkody wokół niego. Będąc w posiadaniu danych otrzymanych z takiego czujnika można byłoby zaimplementować algorytm omijania przeszkód. Algorytm dojazdu do punktu powinien być zmieniony na taki, który koryguje trajektorię w czasie jazdy robota.

Dzięki temu projektowi uzyskano wiedzę na temat projektowania oprogramowania rozproszonego. Dodatkowo zauważono, że nie zawsze warto, jest korzystać z gotowych implementacji, gdyż są poza naszą kontrolą, mogą być trudne w konfiguracji bądź przestarzałe.

Ten projekt ma duży potencjał na rozwój. Można ten pomysł przenieść i rozwinąć na każdym pojeździe naziemnym. Jednak taki rozwój wymagałby dużego zespołu złożonego z różnej klasy specjalistów, mechaników, elektroników i programistów.

Listings

Spis rysunków

Spis tabel

Literatura

[1] Nirmal Kj, Joice Mathew, A. G. Sreejith, Mayuresh Saprotdar (2016) Indian Institute of Astrophysics, Bangalore, India *Noise modeling and analysis of an IMU-based attitude sensor: improvement of performance by filtering and sensor fusion*

[2] Mattia Butta (2012) IEEE *Sources of Noise in a Magnetometer Based on Orthogonal Fluxgate Operated in Fundamental Mode*

[3] Kleinbauer, Rachel (2004) Universität Stuttgart, Helsinki *Kalman filtering implementation with Matlab*

[4] Sathiamoorthy Manoharan (2009) University of Auckland,New Zealand *On GPS Tracking of Mobile Devices*

[5] Norhafizan Ahmad, Raja Ariffin Raja Ghazilla, and Nazirah M. Khairi (2013) University of Malaya *Reviews on Various Inertial Measurement Unit (IMU) Sensor Applications*

[6] Aronowitz F. The laser gyro. In: Ross M., editor. Laser Applications. Volume 1. Academic Press; New York, NY, USA: 1971

[7] Sasiadek, J. Z. Sensor fusion. Annual Reviews in Control, 26(2), 203–228; Department of Mechanical Aerospace Engineering, Carleton University: 2002

[8] Limi Kalita (2014), Department of Computer Science and Engineering, Assam Down Town University, Guwahati, India. *Socket Programming*

[9] Anthony Williams (2019) *Język C++ i przetwarzanie współbieżne w akcji. Wydanie II*

[10] Gyroscope Technology and Applications: A Review in the Industrial Perspective [(czas dostępu 21/11/2022)]; Link: `https://www.ncbi.nlm.nih.gov/pmc/articles/PMC5677445/`

[11] Akcelerometr [(czas dostępu 30/11/2022)]; Link: `https://www.tme.eu/pl/news/library-articles/page/22568/Akcelerometr-jak-dziala-i-do-czego-sluzy/`

[12] Sensor Noise [(czas dostępu 11/11/2022)]; Link: `https://www.stemmer-imaging.com/en/knowledge-base/sensor-noise/`

[13] Sensor Noise Model [(czas dostępu 12/11/2022)]; Link: `https://classic.gazebosim.org/tutorials?tut=sensor_noise&cat=sensors`

[14] Współrzędne geograficzne [(czas dostępu 19/11/2022)]; Link: `https://pl.wikipedia.org/wiki/Wsp%C3%B3%C5%82rz%C4%99dne_geograficzne`

[15] Compass Heading Using Magnetometers [(czas dostępu 19/12/2022)]; Link: `https://cdn-shop.adafruit.com/datasheets/AN203_Compass_Heading_Using_Magnetometers.pdf`

[16] Differential Drive Kinematics [(czas dostępu 12/12/2022)]; Link: `https://www.cs.columbia.edu/~allen/F17/NOTES/icckinematics.pdf`

[17] ROS controllers [(czas dostępu 10/11/2022)]; Link: `http://wiki.ros.org/ros_controllers?distro=noetic`

[18] Ackerman steering controller [(czas dostępu 11/11/2022)]; Link: http://wiki.ros.org/ackermann_steering_controller?distro=noetic

[19] Diff drive controller [(czas dostępu 10/12/2022)]; Link: http://wiki.ros.org/diff_drive_controller?distro=noetic

[20] Using Gazebo plugins with ROS [(czas dostępu 01/11/2022)]; Link: https://classic.gazebosim.org/tutorials?tut=ros_gzplugins

[21] Twist Message [(czas dostępu 09/12/2022)]; Link: http://docs.ros.org/en/api/geometry_msgs/html/msg/Twist.html

[22] ROS Documentation main page [(czas dostępu 01/11/2022)]; Link: http://wiki.ros.org/Documentation

[23] Multidimensional Kalman Filter [(czas dostępu 13/11/2022)]; Link: https://www.kalmanfilter.net/kalmanmulti.html

[24] Actionlib [(czas dostępu 02/11/2022)]; Link: http://wiki.ros.org/actionlib

[25] Services [(czas dostępu 02/11/2022)]; Link: http://wiki.ros.org/Services

[26] Topics [(czas dostępu 02/11/2022)]; Link: http://wiki.ros.org/Topics

[27] Real-time operating system [(czas dostępu 02/11/2022)]; Link: https://en.wikipedia.org/wiki/Real-time_operating_system

[28] Pattern: Microservice Architecture [(czas dostępu 02/11/2022)]; Link: https://microservices.io/patterns/microservices.html

[29] Gazebo [(czas dostępu 03/11/2022)]; Link: https://gazebosim.org/home

[30] Urdf [(czas dostępu 03/11/2022)]; Link: http://wiki.ros.org/urdf

[31] Ackerman Steering [(czas dostępu 04/11/2022)]; Link: https://www.xarg.org/book/kinematics/ackerman-steering/

[32] XACRO [(czas dostępu 03/11/2022)]; Link: http://wiki.ros.org/xacro

[33] XML Specifications [(czas dostępu 03/11/2022)]; Link: http://wiki.ros.org/urdf/XML

[34] ROS messages [(czas dostępu 02/11/2022)]; Link: http://wiki.ros.org/msg

[35] Specjalne zastosowania filtrów cyfrowych [(czas dostępu 15/11/2022)]; Link: https://sound.eti.pg.gda.pl/~greg/dsp/07-SpecjalneFiltry.html

[36] Rodzina protokołów TCP/IP [(czas dostępu 19/11/2022)]; Link: https://home.agh.edu.pl/~pmarynow/pliki/siec/is/TCP.pdf

[37] What Is Aa Inertial Measurement Unit [(czas dostępu 16/11/2022)]; Link: https://www.vectornav.com/resources/inertial-navigation-articles/what-is-an-inertial-measurement-unit-imu

[38] Kalman Filter In One Dimension [(czas dostępu 17/11/2022)]; Link: https://www.kalmanfilter.net/kalman1d.html

[39] A geometric interpretation of the covariance matrix [(czas dostępu 18/11/2022)]; Link: https://www.visiondummy.com/2014/04/geometric-interpretation-covariance-matrix/

[40] Odometry [(czas dostępu 15/11/2022)]; Link: https://docs.donkeycar.com/parts/odometry/

[41] Inertial Labs. [(czas dostępu 20/11/2022)]; Link: https://inertiallabs.com/ahrs.html

[42] Elektronika Praktyczna [(czas dostępu 6/11/2022)]; Link: https://ep.com.pl/files/3937.pdf

[43] Principles of GPS [(czas dostępu 8/11/2022)]; Link: https://www.gisresources.com/wp-content/uploads/2013/11/Principles-of-GPS-4-13-04.pdf

[44] The Math Behind GPS [(czas dostępu 8/11/2022)]; Link: https://letstalkscience.ca/educational-resources/backgrounders/math-behind-gps

[45] Magnetometer basics for mobile phone applications [(czas dostępu 23/11/2022)]; Link:http://www.nicap.org/madar/smart_phones/Magnetometer_Basics_For_Mobile_Phone_Applications-2012.pdf

[46] Review of visual odometry: types, approaches, challenges, and applications [(czas dostępu 8/15/2022)]; Link: https://link.springer.com/article/10.1186/s40064-016-3573-7

[47] Robo-Rats Locomotion: Synchro Drive [(czas dostępu 04/01/2023)]; Link: https://groups.csail.mit.edu/drl/courses/cs54-2001s/synchro.html

[48] Diffrential wheeled robot [(czas dostępu 06/12/2022)]; Link: https://en.wikipedia.org/wiki/Differential_wheeled_robot#/media/File:Differential_Drive_Kinematics_of_a_Wheeled_Mobile_Robot.svg

[49] Math behind GPS [(czas dostępu 06/12/2022)]; Link: https://letstalkscience.ca/educational-resources/backgrounders/math-behind-gps

[50] Magnetometer Basics For Mobile Phone Applications [(czas dostępu 07/12/2022)]; Link: http://www.nicap.org/madar/smart_phones/Magnetometer_Basics_For_Mobile_Phone_Applications-2012.pdf

[51] Rodzaje i parametry enkoderów [(czas dostępu 10/12/2022)]; Link: https://www.simex.pl/pl/aktualnosci/c,artykuly/jak-dobrac-enkoder

[52] Implementacja symulatora IMU [(czas dostępu 15/01/2023)]; Link: https://github.com/ros-simulation/gazebo_ros_pkgs/blob/a5c968497ff2c3cc353daa03e12929b4dd3197b2/gazebo_plugins/src/gazebo_ros_imu_sensor.cpp#L120

[53] Metody komunikacji międzyprocesowej w systemach lokalnych i rozproszonych [(czas dostępu 17/01/2023)]; Link: http://kajtsweb.webd.pl/pwr/kajt/materialy/INT%206.%20Metody%20komunikacji%20miedzyprocesowej%20w%20systemach%20lokalnych%20i%20rozproszonych.pdf

[54] Linux manual page [(czas dostępu 15/01/2023)]; Link: https://man7.org/linux/man-pages/man2/shmop.2.html

[55] Using named pipes [(czas dostępu 15/01/2023)]; Link: https://www.ibm.com/docs/en/zos/2.2.0?topic=io-using-named-pipes

[56] Microservices vs. monolithic architecture [(czas dostępu 07/01/2023)]; Link: https://www.atlassian.com/microservices/microservices-architecture/microservices-vs-monolith

[57] Roll and Pitch Angles From Accelerometer Sensors [(czas dostępu 05/01/2023)]; Link: https://mwrona.com/posts/accel-roll-pitch/