

POLITECHNIKA WROCŁAWSKA
WYDZIAŁ ELEKTRONIKI

KIERUNEK: TELEINFORMATYKA

SPECJALNOŚĆ: PROJEKTOWANIE SIECI TELEINFORMATYCZNYCH

PRACA DYPLOMOWA
INŻYNIERSKA

Tworzenie mapy pomieszczenia za pomocą
autonomicznej, mobilnej platformy skanującej

Creating a room map using autonomous,
mobile scanning platform

AUTOR:

Tomasz Jakubowski

PROWADZĄCY PRACĘ:
dr inż. Paweł Trajdos, K32Wo4Do3

Spis treści

1. Wprowadzenie	5
1.1. Wstęp	5
1.2. Cel i zakres pracy	5
1.2.1. Cel	5
1.2.2. Zakres	6
1.3. Koncepcja projektu	6
1.4. Przegląd technologii	7
2. Projekt	10
2.1. Oprogramowanie PC	10
2.1.1. Struktura oprogramowania	10
2.1.2. Jazda autonomiczna	17
2.2. Firmware	18
2.3. Budowa platformy	20
2.3.1. Budowa mechaniczna	22
2.3.2. Budowa elektroniczna	25
3. Ewaluacja	29
3.1. Odometria	29
3.1.1. Kalibracja magnetometru	33
3.1.2. Filtr Kalmana	38
3.1.3. Poprawa jakości odometrii	39
3.2. Skan otoczenia i budowa mapy	43
4. Podsumowanie	53
Bibliografia	54
A. Opis załączonej płyty CD/DVD	58

Słownik skrótów i pojęć

- IoT** (ang. *Internet of Things*) *Internet Rzeczy* - koncepcja przedmiotów codziennego użytku samodzielnie komunikujących się za pośrednictwem sieci Internet bądź innego środka w celu gromadzenia i przetwarzania danych.
- ToF** (ang. *Time of Flight*) *Czas lotu* - tutaj w odniesieniu do rodzaju sensora laserowego mierzącego dystans za pomocą pomiaru czasu jaki upływa od wysłania impulsu świetlnego do odebrania jego odbicia od przeszkody.
- SLAM** (ang. *Simultaneous Localization And Mapping*) *Symultaniczna lokalizacja i mapowanie* - problem obliczeniowy dotyczący jednoczesnego tworzenia obrysu obiektów w otoczeniu na podstawie pomiarów i znanej lokalizacji jednostki mierzącej w przestrzeni oraz określenia położenia jednostki mierzącej w otoczeniu na podstawie dokonywanych przez nią pomiarów. Typowo algorytmy rozwiązujące ten problem korzystają z laserowych czujników odległości.
- VSLAM** (ang. *Visual Simultaneous Localization And Mapping*) - podejście do problemu SLAM z wykorzystaniem kamer i algorytmów przetwarzania obrazu.
- LIDAR** (ang. *Light Detection and Ranging*) - rodzaj sensora mierzący odległość przy pomocy wiązki laserowej.
- USB-UART** - rodzaj urządzenia elektronicznego pośredniczącego w komunikacji. Pozwala porozumiewać się urządzeniom, z których jedno korzysta z komunikacji USB a drugie z komunikacji szeregowej.
- FDM** (ang. *Fused Deposition Modelling*) - technologia druku 3D polegająca na układaniu roztopionego filamentu warstwami w celu utworzenia trójwymiarowego obiektu.
- PLA** (ang. *PolyLactic Acid*) - rodzaj sztywnego materiału wykorzystywanego jako filament w druku 3D
- TPU** (ang. *Thermoplastic PolyUrethane*) - elastyczny materiał do druku 3D
- EEPROM** (ang. *Electrically Erasable Programmable Read-Only Memory*) - rodzaj nieulotnej pamięci wykorzystywany w mikrokontrolerach
- MEMS** (ang. *MicroElectroMechanical System*) - mikroskopijne układy elektomechaniczne, najczęściej wykonywane w krzemie za pomocą roztworów trawiących.
- PC** (ang. *Personal Computer*) *Komputer Osobisty*
- Smart** (z ang. *sprytny*) - w kontekście tego dokumentu jest to jedno z haseł często występujących w nazwach i reklamach produktów wyposażonych w dowolnego rodzaju mikrokomputer z oprogramowaniem pozwalającym na komunikację sprzętu z aplikacją na smartfona bądź interfejsem przeglądarkowym, rzekomo ułatwiającym korzystanie z niego. Ubarwia ono obraz urządzenia w oczach konsumenta mając przekonać go do jego innowacyjności.
- Intelligent** (z ang. *inteligentny*) - podobnie jak "smart" w kontekście niniejszej pracy jest często wykorzystywanym określeniem nowoczesnych produktów.
- Firmware** - oprogramowanie wbudowane działające na mikrokontrolerze.

Arduino - rodzaj płyt drukowanych[42] z wbudowanymi mikrokontrolerami o publicznie dostępnymi schematach elektronicznych, również otwartoźródłowe środowisko służące do pisania, komplikacji i wgrywania programów na płytki.

STM32duino - nieoficjalna implementacja środowiska komplikacji (bibliotek) Arduino dla płyt z mikrokontrolerami STM32.

Rozdział 1

Wprowadzenie

1.1. Wstęp

Wraz z rozwojem technologii i wciąż rosnącym popytem na nowinki elektroniczne łatwo jest zaobserwować jak z biegiem czasu pewne rozwiązania adaptowane są do użytku codziennego. Sprzęt niegdyś będący jedynie drogą ciekawostką staje się akcesorium bez którego coraz trudniej jest wyobrazić sobie normalne życie. Tak było z komputerami osobistymi, płytami elektronicznymi, dostępem do Internetu, następnie do listy dołączyły smartfony, hulajnogi elektryczne czy douszne słuchawki bluetooth. Coraz popularniejsze stają się także rozwiązania IoT (ang. *Internet of Things*) z zakresu AGD - inteligentne pralki, lodówki, odkurzacze. I to właśnie ostatnie z rozwiązań szczególnie przykuło uwagę autora.

Podczas gdy wiele sprzętu reklamowanego z wykorzystaniem haseł takich jak "smart" czy "intelligent" z inteligencją nie ma zbyt wiele wspólnego, to ta granica zdaje się coraz mocniej zacierać - szczególnie w przypadku autonomicznych jednostek stworzonych w celu konserwacji powierzchni płaskich. I tutaj nasuwa się pytanie - Dlaczego? Otóż wiele z wymienionych wyżej urządzeń realizuje pojedyncze, wyznaczone, stosunkowo proste zadania a nowoczesnym dodatkiem do tego ma być łączność z siecią, zdalna obsługa z poziomu aplikacji czy duży, kolorowy wyświetlacz pokazujący aktualną pogodę. Oczywiście inteligentne odkurzacze również oferują podobne udogodnienia, jednak nie to wyróżnia je na tle innych akcesoriów. Tym czynnikiem jest złożoność, na pierwszy rzut oka prostego, zadania które realizują. Na drodze do budowy samodzielnie odkurzającego robota stoi wiele przeszkód związanych z lądowaniem, mapowaniem, nawigacją, omijaniem obiektów stojących na wyznaczonej trasie, radzeniem sobie ze zmieniającym się otoczeniem.

W tej pracy przedstawione zostanie autorskie podejście do jednego z tych problemów, przy którym zostaną wykorzystane zarówno własnoręcznie sporządzone jak i gotowe, publicznie dostępne rozwiązania. Tym problemem jest tworzenie mapy pokoju.

1.2. Cel i zakres pracy

1.2.1. Cel

Celem jest stworzenie działającego systemu składającego się z autonomicznego robota (platformy mobilnej) połączonego z aplikacją na komputerze stacjonarnym lub laptopie. Robot ten ma za zadanie samodzielnie poruszać się po pomieszczeniu i skanować je w poszukiwaniu przeszkód (ścian, foteli, nóg krzeseł, stołów itp.), a pozyskane dane przesyłać do komputera. W apli-

kacji z danych zebranych z otoczenia robota ma powstać dwuwymiarowa mapa pomieszczenia opisująca je na poziomej płaszczyźnie ok. 10 cm nad poziomem podłogi.

1.2.2. Zakres

- zaprojektowanie i zbudowanie jeżdżącego robota wyposażonego w laserowy sensor odległości i moduł Bluetooth do komunikacji z komputerem
- implementacja algorytmu pozwalającego na autonomiczne poruszanie
- napisanie aplikacji rysującej mapę pomieszczenia działającej na komputerze lub laptopie

1.3. Koncepcja projektu

Po pierwsze robot powinien posiadać zasilanie akumulatorowe - w przeciwnym wypadku wymagałyby podania zasilania za pośrednictwem przewodu, co byłoby niepraktyczne. Aby zasilania stało się na jak najwięcej czasu dobrze było, aby nie wykorzystywał zgromadzonej energii na obliczenia, które mogą zostać wykonane po stronie aplikacji go kontrolującej. Z tego względu do kontroli peryferiów platformy została wykorzystany mikrokontroler.

Aby móc sporządzić mapę potrzebne będą lokalizacje przeszkód w postaci punktów reprezentowanych na dwuwymiarowej płaszczyźnie. Taki punkt można obliczyć znając odległość i kierunek do przeszkody względem robota - potrzebuje on więc sensorów dzięki którym będzie w stanie tą odległość zmierzyć. Istnieją gotowe rozwiązania takie jak RPLidar A1M8 [9], lub RPLIDAR A2M6 [10], korzystające z lasera obracającego się na podstawie, jednak są na tyle kosztowne, że ostatecznie zdecydowano się na tańszą alternatywę - Lidar TF Luna. Różnica w cenie tańszego z dwóch wymienionych wcześniej czujników a wybranego to na dzień dzisiejszy ponad 1500 złotych. Sensor odległości zamontowany na wieżyczce obracanej za pomocą serwomechanizmu w zakresie od 0° do 180° jest wystarczający do realizacji tego zadania.

Żeby pozycja wcześniej wspomnianych punktów była odwzorowywała rozkład pomieszczenia platforma powinna realizować zadanie nawigacji zliczeniowej (odometrii). Dzięki temu możliwe jest oszacowanie aktualnej pozycji i przemieszczenia robota w przestrzeni względem jego pozycji startowej. Dopiero korzystając z tych danych połączonych z kierunkiem i dystansem do przeszkody można umieścić ją na mapie.

Dla mniejszego poślizgu oraz łatwego pokonywania niewielkich, nieznaczących przeszkód takich jak przewody, listwy zasilające, dywaniki oraz dla ułatwienia obliczeń związanych z odometrią robot powinien być wyposażony w system napędowy o odpowiednio małym poślizgu.

Należy również ustalić w jakim języku programowania napisane będą programy - wszakże w zależności od środowiska programista powinien spodziewać się innych możliwości i ograniczeń. Ze względu na możliwość szybkiego prototypowania i czytelność kodu do napisania oprogramowania PC wybrany został język Python[33]. Do zaprogramowania robota autor wybrał język C++[34]. Do reprezentacji graficznej mapy stworzonej przez pojazd można posłużyć się środowiskiem *Processing* [53] bądź podobną implementacją zgodną z językiem Python.

1.4. Przegląd technologii

Nie jest tajemnicą że inspiracją do tego projektu były coraz popularniejsze robotyczne odkurzacze. Warto jednak nadać pewien rys historii rozwoju tej branży i technologii które były stosowane na przełomie lat.

Pierwszym komercyjnie dostępnym robotem odkurzającym był Electrolux Trilobite[39]. Robot ten był wyposażony w sensory ultradźwiękowe dzięki czemu zachowywał odstęp od ścian. Sensory takie nie były wystarczające - ostre obiekty bądź takie o niewielkiej powierzchni mogły zostać przeoczone dlatego też dodatkowo miał on zderzak wciskający się podczas kontaktu z przeszkodą. Do tego wykrywał on uskoki takie jak np. schody za pomocą czujnika wykorzystującego promieniowanie podczerwone a także był w stanie omijać strefy ręczne wyznaczone magnetycznymi paskami umieszczonymi na powierzchni czyszczonej. Z czasem również konkurencja zaczęła się wdrażać w rynek i oferować podobne rozwiązania.

Pierwsze odkurzacze nie korzystały z zaawansowanych algorytmów nawigacji, mając do dyspozycji jedynie ograniczony zasób informacji o otaczającym je środowisku ze względu na proste czujniki w nich montowane. Poruszały się one w sposób wręcz chaotyczny, "odbijając" się od przeszkód pod różnymi kątami, nie czyszcząc równomiernie całej powierzchni. Nie mniej jednak takie rozwiązanie było na dane czasy wystarczające i samo w sobie bardzo innowacyjne - zwalniało użytkownika z konieczności odkurzania ręcznego.

Z biegiem lat, rozwojem algorytmów nawigacji oraz (co najważniejsze w kontekście sprzedaży urządzeń na rynku konsumenckim) taniejącą elektroniką i coraz mniej kosztownymi sensorami zaczęto implementować bardziej złożone rozwiązania. Na dzień dzisiejszy inteligentne odkurzacze korzystają z dwóch technologii:

- SLAM - za pomocą laserowego sensora robot mierzy odległość od swojego punktu do przeszkód wokół. Zmierzone odległości jest w stanie przenieść na mapę. Mapa może być wiadoczną z poziomu aplikacji zarządzającej robotem i służyć np. do wyznaczania stref które ma omijać. Ponadto pozwoli mu odnaleźć się w danym domu i wyznaczyć sobie ścieżkę, obliczyć bieżący procent wykonanej pracy czy też wrócić do bazy ładującej
- VSLAM (Visual SLAM) - w tym wypadku robot jest wyposażony w kamerę skierowaną bezpośrednio na sufit lub pod kątem. Dzięki zaawansowanym algorytmom przetwarzania obrazu wybiera on pewne punkty odniesienia na podstawie których szacuje swoją pozycję względem otoczenia. Technologia wymaga wyższej mocy obliczeniowej i jest mniej dokładna, nie mniej jednak pozostawia otwarte pole na innowacje w zakresie sztucznej inteligencji i rozpoznawania typu obiektów.

Ze względu na niewielkie doświadczenie autora w dziedzinie komputerowego przetwarzania obrazów, w niniejszej pracy wykorzystany zostanie jeden z algorytmów działających w technologii SLAM - Gmapping, oparty o filtr częstek typu *rao-blackwellized* [52][48][47].

W Tab. 1.1 przedstawiono porównanie wybranych, dostępnych aktualnie na polskim rynku sensorów. Można wydzielić tutaj dwie zasadnicze grupy - jednostki mierzące punktowo oraz te które mierzą pewien obszar w jednym lub dwóch wymiarach. Te pozycje które nie mają określonego *kąta widzenia w poziomie* należą do drugiej kategorii. Takie czujniki posiadają mechanizmy multipleksacji strumienia światlnego, np. w postaci obrotowej wieżyczki na której zamontowany jest laser. Należy również zaznaczyć, że maksymalna częstotliwość pomiarów w tych sensorach oznacza pomiar całego mierzonego obszaru, w czujnikach mierzących punktowo dotyczy ona pojedynczego punktu.

Niestety, o ile jest to rozwiązanie wygodne, zwalniające inżyniera z potrzeby budowania własnej wieżyczki, o tyle takie sensory są dużo droższe - najtańszy z nich *Slamtec RPLidar AIM8* kosztuje 500 złotych, będąc nawet poniżej półki cenowej droższych rozwiązań mierzących punktowo. Najtańszy z dostępnych *Benewake Lidar TF Luna* posiada dwudziestocentymetrową

martwą strefę, nie odstając znacząco od konkurencji, gdzie najmniejszą strefą charakteryzuje się ośmiokrotnie droższy sensor *SparkFun Lidar Lite v3HP*. Martwą strefę można zmniejszyć, montując sensor w cofnięciu od osi obrotu, należy wówczas dodać odpowiedni dystans do zmierzanej wartości. Minusem takiego rozwiązania jest większy rozmiar wiązki przy takiej samej odległości od obiektu.

Nie mniej jednak najniższy pobór energii, wystarczająca częstotliwość próbkowania i akceptowalny zasięg 8 metrów (przeciętny pokój ma wymiary od kilku do kilkunastu metrów mierząc od najodleglejszych punktów) są tutaj największymi atutami czujnika firmy *Benewake*. Mając na uwadze wyżej wymienione parametry oraz ograniczony budżet autor zdecydował się na wykorzystanie właśnie tego urządzenia.

Tab. 1.1: Przegląd dostępnych na rynku sensorów LIDAR

Producent, model	Zasięg	Kąt widzenia w poziomie	Dokładność	Rozbieżność wiązki	Pobór energii	Maks. częst. pomiarów	Cena
-	m	°	m	°	W	Hz	zł
Benewake Lidar TF Luna [4]	0,2-8	n/d	0,06 (2% dla >3m)	2	0,35	100	89
Benewake Lidar TFMini-S [8]	0,1-12	n/d	0,06 (1% dla >6m)	2	0,7	100	180
Benewake Lidar TFMini Plus [7]	0,1-12	n/d	0,05 (1% dla >6m)	3,6	0,55	1000	200
Benewake Lidar TF02 Pro [6]	0,1-40	n/d	0,05 (1% dla >5m)	3	1	100	400
Benewake Lidar TF02 [5]	0,4-22	n/d	0,06 (2% dla >5m)	3	1	100	490
Slamtec RPLidar A1M8 [9]	0,15-12	360	0,0005 (1% dla >1,5m)	1	2	10	500
SparkFun Lidar Lite v3 [12]	0-40	n/d	0,025	0,46	0,65	500	760
SparkFun Lidar Lite v3HP [13]	0,05-40	n/d	0,025	0,46	0,43	1000	800
Slamtec RPLidar A2M8 [11]	0,15-8	360	0,0005 (1% dla >1,5m)	0,9	2,25	15	1600
Slamtec RPLidar A2M6 [10]	0,15-18	360	0,0005 (1% dla >1,5m)	0,9	2,25	15	2900
Benewake Lidar CE30-A [3]	0,1 - 4	130	0,06	0,41	6	20	3700

Rozdział 2

Projekt

2.1. Oprogramowanie PC

W niniejszym rozdziale opisana zostanie implementacja aplikacji komputerowej do sterowania platformą, jak również środowisko w którym pracuje. Część elementów stosowana była zamieniane podczas przebiegu projektu - powody zmian oraz przebieg doświadczeń dokładnie opisuje rozdział 3. Pełne repozytorium z kodem źródłowym aplikacji sterującej, jak również opisanego w rozdziale 2.2 kodu samego robota dostępne jest publicznie w serwisie github.com [26].

2.1.1. Struktura oprogramowania

Robot wykonuje polecenia zadane przez aplikację sterującą oraz odpowiada na zapytania o dane. Aplikacja w ostatecznej wersji została zaimplementowana jako moduł platformy ROS[41] na systemie operacyjnym opartym o GNU/Linux. Po jej uruchomieniu ukazują się dwa okna - jedno jest oknem głównym do kontroli pojazdu, widoczne na rysunku 2.1. Jego interfejs został stworzony w środowisku *Qt5* dla języka Python [25]. Drugie to okno programu RViz w którym widoczny jest podgląd pozycji robota oraz mapy którą sporządza ukazane na rysunku 2.5.

Okno główne programu

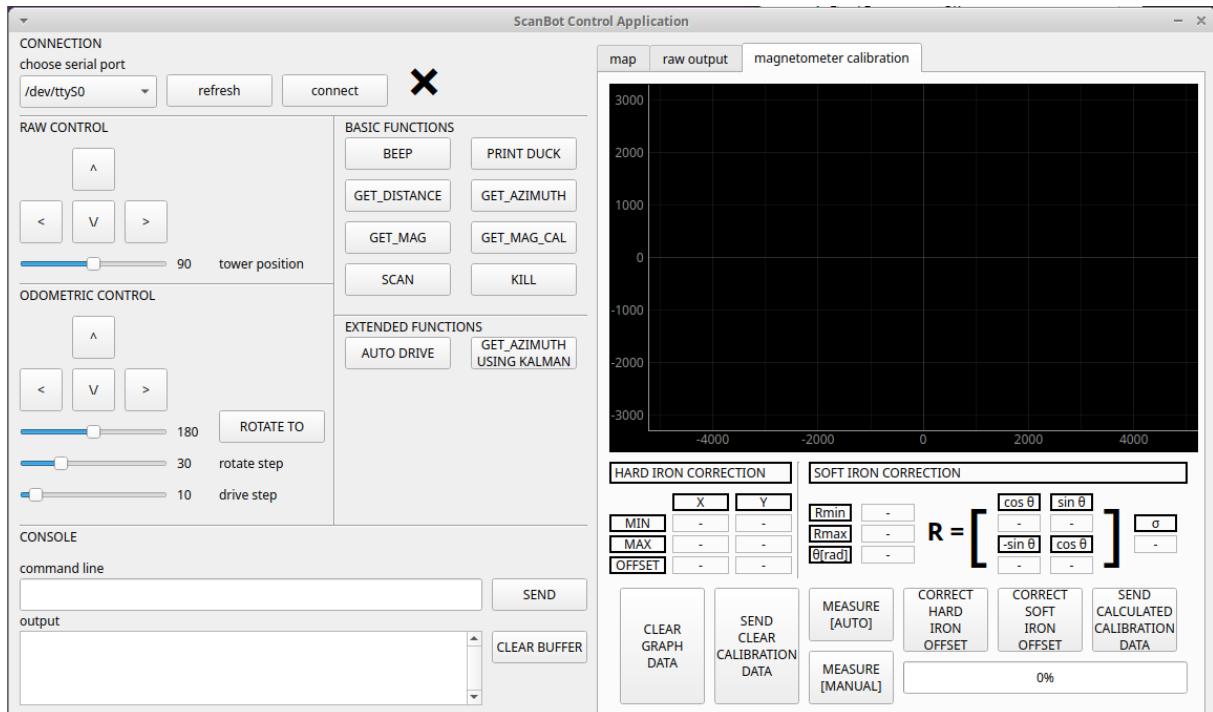
Okno główne zostało podzielone na segmenty oraz zakładki w celu odseparowania poszczególnych funkcjonalności. Opis poszczególnych sekcji:

Sekcja *CONNECTION* dotyczy połączenia portu szeregowego i zawiera elementy takie jak:

- Lista rozwijana do wyboru portu szeregowego do którego wpisany jest konwerter USB-UART
- Przycisk *refresh* odświeża listę
- Przycisk *connect* inicjuje połączenie
- Ikona statusu połączenia - krzyżyk oznacza jego brak, zegarek oznacza oczekiwanie a symbol fajki oznacza zainicjowane połączenie

Sekcja *RAW CONTROL* zawiera kontrolki sterujące robotem. Przytrzymanie każdego z przycisków odpowiada ruchowi robota w przód i w tył (kolejno strzałki w góre i w dół) lub obrotowi w lewo i w prawo (strzałki w lewo i w prawo). Po puszczeniu przycisku platforma zatrzymuje się bez zwłoki. Suwak służy do ustawiania pozycji wieżyczki pomiarowej, jest wyskalowany w stopniach.

Sekcja *ODOMETRIC CONTROL* służy do jazdy odometrycznej, tj. poruszając się za pomocą tych kontrolek uwzględniany jest przejechana trasa. Analogicznie jak w poprzedniej sekcji poruszanie kontrolowane jest przez strzałki, aczkolwiek tutaj nie należy ich przytrzy-



Rys. 2.1: Okno główne aplikacji sterującej

mywać a jedynie krótko kliknąć. Po kliknięciu zostanie pokonany pewien dystans po którym robot się zatrzyma. Dystans do przejechania (krok) ustawiany jest za pomocą suwaka *drive step* a wartość przez niego reprezentowana jest podana w centymetrach. Aby ustawić krok obrotu należy skorzystać z suwaka *rotate step*. Przycisk *ROTATE TO* służy do obracania robota na wyznaczony azymut, którego wartość wcześniej należy ustawić suwakiem znajdującym się obok niego. (Uwaga! Pojęcie azymutu w tym dokumencie oznacza kąt skierowania robota liczony od azymu geograficznego 90° , przeciwnie do ruchu wskazówek zegara, czyli tak jak matematycznie reprezentowany jest kąt).

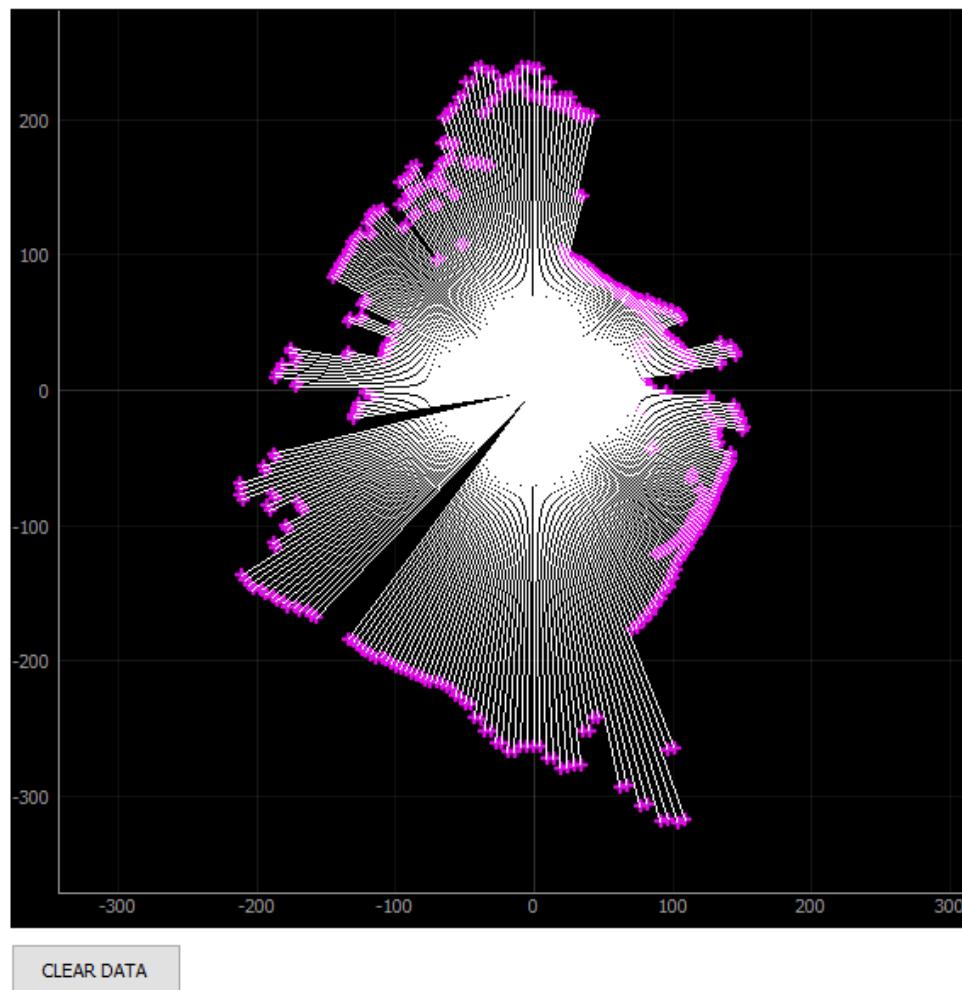
Sekcja *CONSOLE* pozwala na ręczne sterowanie robotem za pośrednictwem komend. Tutaj należy zaznaczyć że wszyskie funkcje sterujące oferowane przez program korzystają z tych komend odpowiednio je formułując i wysyłając do jednostki mobilnej. Pole tekstowe podpisane *command line* służy do wprowadzania komendy, która jest wysyłana po kliknięciu przycisku *SEND*. Owa komenda pojawi się wtedy w polu tekstowym poniżej oznaczonym *output*. Przycisk *CLEAR BUFFER* wyczyści okno podglądu wraz z buforem nadawczym i odbiorczym, co przydaje się przy (na szczęście bardzo rzadkich) problemach z zerwaną komunikacją lub utratą synchronizacji sekwencji przesyłanych komend i odpowiedzi.

Sekcja *BASIC FUNCTIONS* posiada kolekcję przycisków wykonujących podstawowe funkcje robota niezwiązane z poruszaniem.

- *BEEP* powoduje wydanie 3 krótkich dźwięków przez robota
- *PRINT DUCK* rysuje małą kaczkę na ekranie
- *GET_DISTANCE* mierzy i zwraca odległość od robota do przeszkody na którą aktualnie wycełowana jest wieżyczka
- *GET_AZIMUTH* mierzy i zwraca azymut w którym skierowany jest robot
- *GET_MAG* zwraca surowe dane z magnetometru (osie X i Y)
- *GET_MAG_CAL* zwraca dane kalibracji magnetometru

- *SCAN* skanuje otoczenie (obraca wieżyczkę i dokonuje serii 180 pomiarów odległości), po czym zwraca zmierzone wartości
- *KILL* odłącza sygnał sterujący od serwomechanizmów
- Sekcja *EXTENDED FUNCTIONS* pozwala na wykonywanie bardziej złożonych zadań
- *AUTO DRIVE* załącza algorytm autonomicznej jazdy robota
- *GET_AZIMUTH USING KALMAN* wykonuje serię pomiarów i za pomocą prostej implementacji filtru Kalmana[50] zwraca przefiltrowany wynik reprezentujący azymut w którym robot jest skierowany

Zakładka *map* (rysunek 2.2) zawiera wykres który przedstawia mapę zmierzonych punktów. Autor zdecydował o porzuceniu pomysłu z wykorzystaniem środowiska *Processing* ze względu na konieczność ewaluacji całego programu w tym środowisku. Przycisk *CLEAR DATA* służy do czyszczenia całego wykresu. W ostatecznej wersji, zamiast rysowania mapy w tej zakładce, wyświetlana jest ona w programie *RViz* opisany w dalszej części dokumentu. Wykres wyskalowany jest w centymetrach.



Rys. 2.2: Zakładka *map*

Zakładka *raw output* (rysunek 2.3) zawiera pełnowymiarowy podgląd na historię wysyłanych i odbieranych danych. Jest większą wersją okienka *output* sekcji *CONSOLE*.

Zakładka *magnetometer calibration* (rysunek 2.4) zawiera zestaw elementów wykorzystywanych do kalibracji magnetometru. Informacje o stosowanych metodach i przebiegu kalibracji znajdują się w rozdziale 3.1.

Rys. 2.3: Zakładka *raw output*

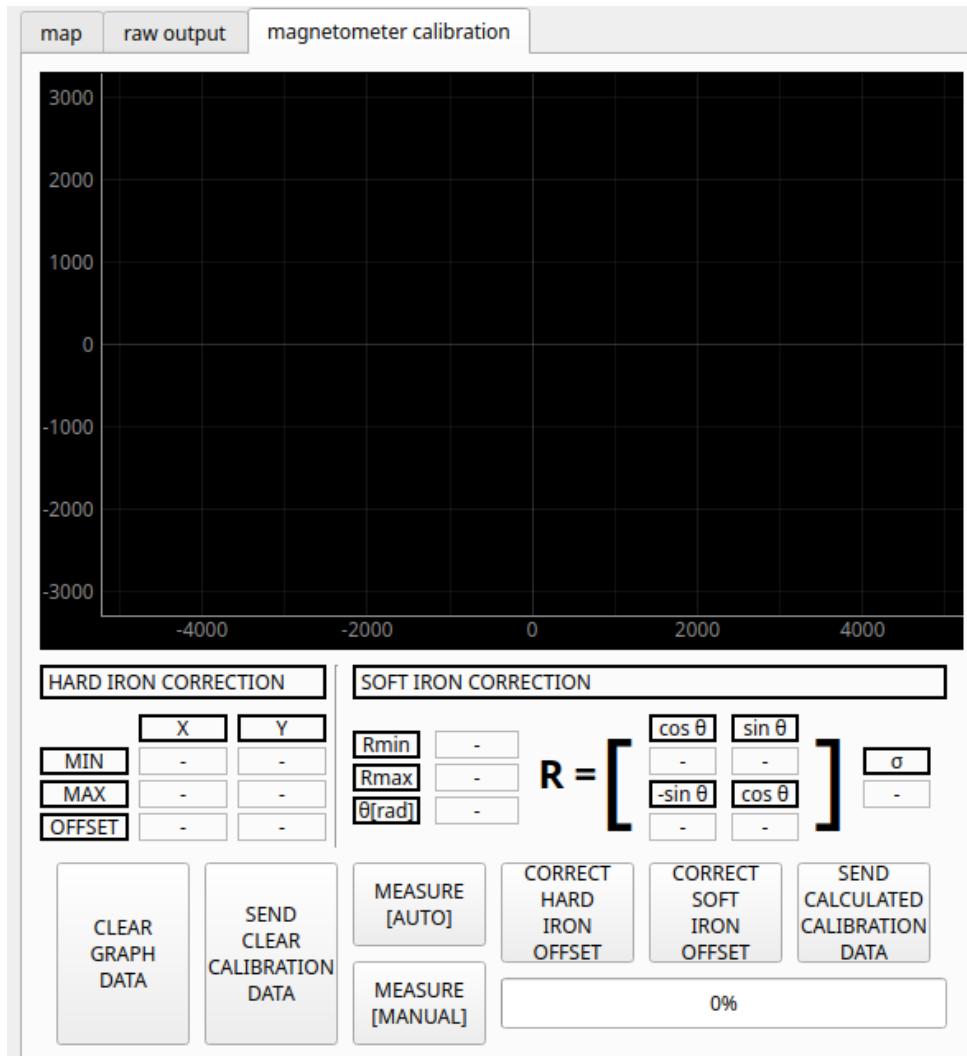
Program RViz

Z okna głównego programu RViz należy zwrócić uwagę jedynie na podgląd układu współrzędnych po prawej stronie okna. Wszystkie parametry są ustawiane automatycznie wraz ze startem aplikacji, nie ma potrzeby żadnej ręcznej konfiguracji. Więcej informacji o tym programie dostępne jest na stronie internetowej [38]. Podczas pracy programu aktualna pozycja robota jest reprezentowana jako obiekt z etykietą *base_link* - podczas jazdy jego pozycja jest aktualizowana względem etykiety *odom*. Na początku wszystkie układy współrzędnych są w pozycji zerowej. Wykres posiada siatkę wyskalowaną w metrach.

Środowisko sterujące

Aplikacja komputerowa po uruchomieniu w środowisku ROS widoczna jest jako jeden z węzłów (node). W celu sporządzania mapy komunikuje się z innymi węzłami za pośrednictwem tematów (topics). Na rysunku 2.6 ukazany jest schemat komunikacji poszczególnych węzłów. Wykorzystywane są trzy tematy:

- *tf* jest tematem na którym publikowane są wszelkie translacje, czyli zależności między układami odniesienia
 - *scan* - tutaj publikowane są wyniki pomiarów odległości po wykonaniu skanu przez robota

Rys. 2.4: Zakładka *magnetometer calibration*

- *map* - na tym temacie pojawiają się dane reprezentujące mapę, wygenerowane przez algorytm *gmapping*[48][35][37]

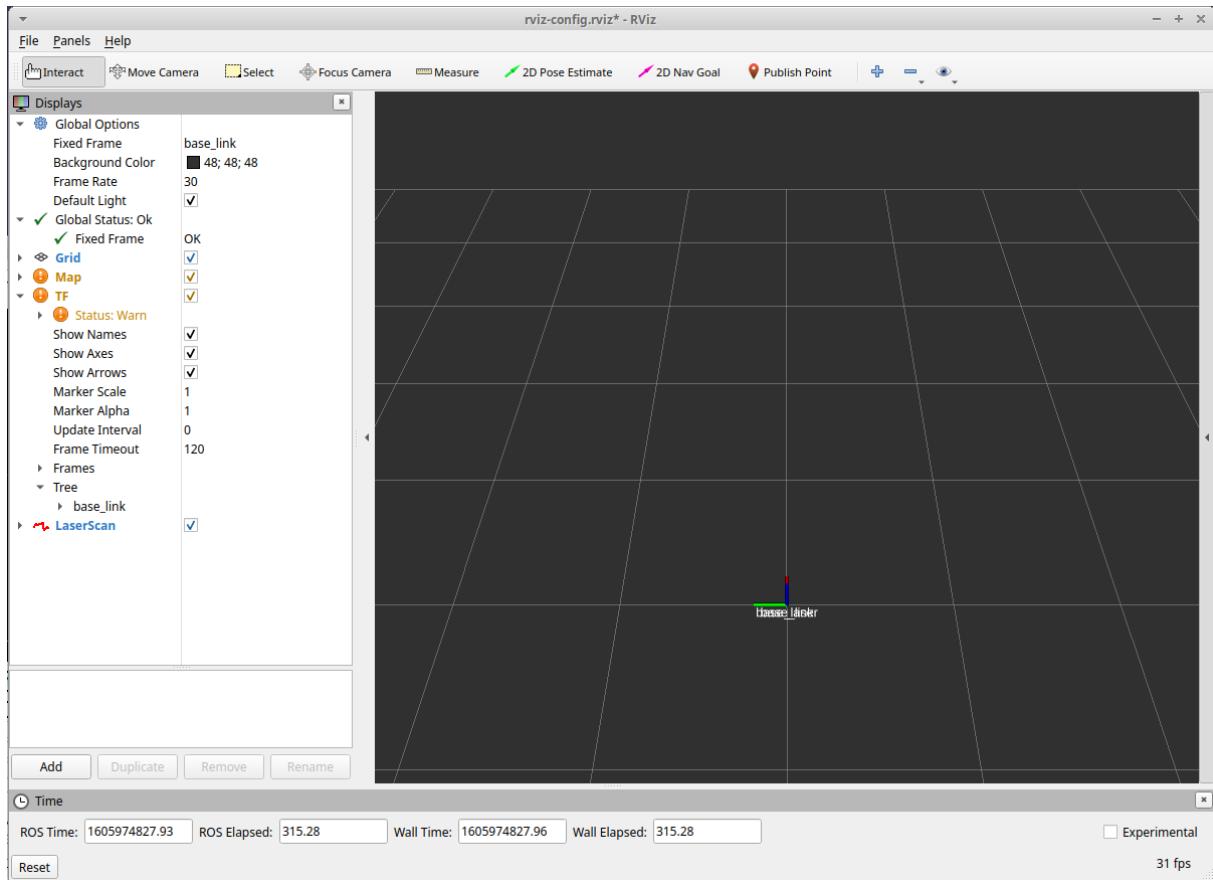
Ramki (frames), czyli w terminologii stosowanej w ROS układy odniesienia, a konkretnie ich względna pozycja są reprezentowane za pomocą wektorów zawierających dane o przesunięciu jak i obrocie. Więcej informacji na temat szczegółowego działania ramek można zasięgnąć na stronie internetowej ROS'a[41]. W niniejszym projekcie wykorzystywane są 4 ramki:

- *map* to główny układ odniesienia względem którego rysowana jest mapa
- *odom* jest punktem odniesienia względem którego działa algorytm nawigacji zliczeniowej
- *base_link* odnosi się do bazy pojazdu, tj. środka względem którego się obraca podczas zakręcania
- *base_laser* to pozycja sensora skanującego

Wszystkie translacje między ramkami publikowane są na temacie *tf*.

Węzeł *laser_static Pod broadcaster* cyklicznie publikuje informację o położeniu modułu skanującego względem bazy robota. Jako że znajduje się ona bezpośrednio nad środkiem obrotu platformy, a sporządzana mapa jest dwuwymiarowa, jest to wektor zerowy.

Węzeł *scanbot_communicator* publikuje swoją pozycję na temacie *tf* oraz podczas wykonywania skanu na kanale *scan*.



Rys. 2.5: Okno programu RViz

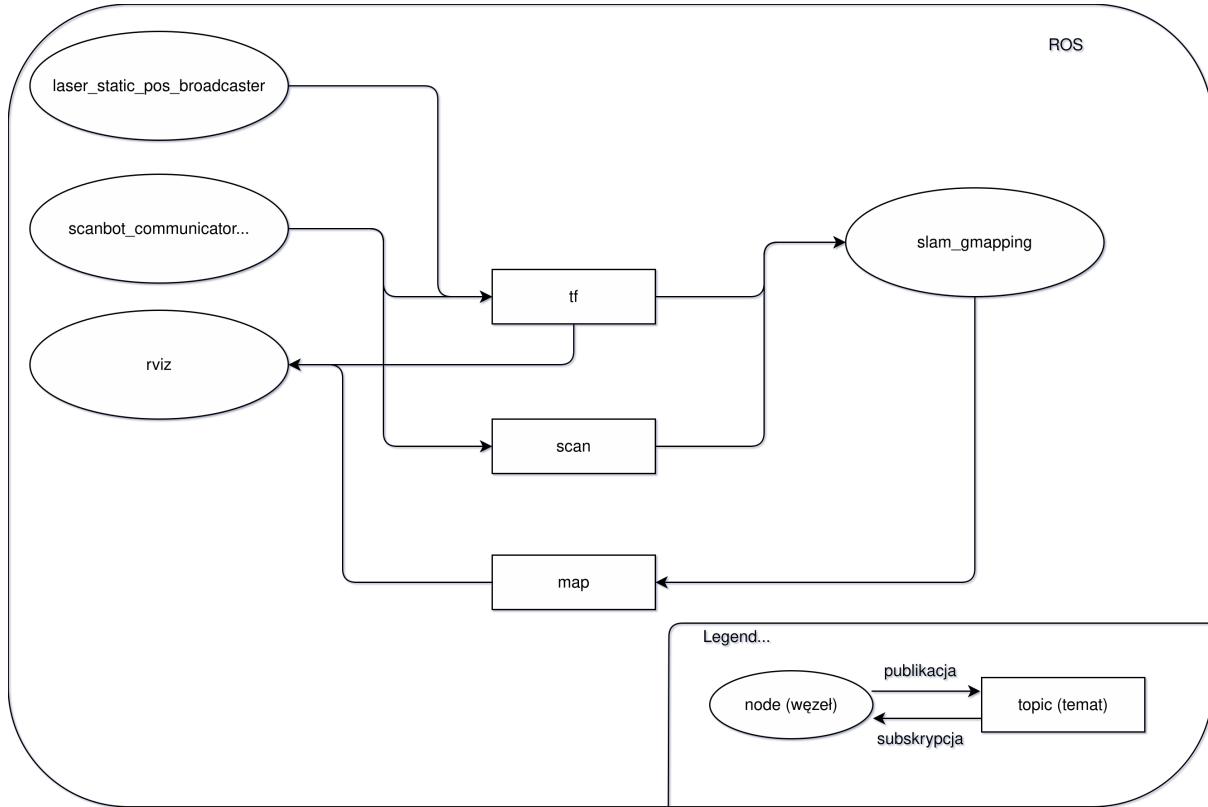
Węzeł *slam_gmapping* subskrybuje kanały *tf* i *scan*. Na ich podstawie odpowiednio filtrując punkty uzyskane z pomiarów i dopasowując je do poprzednich algorytm tworzy mapę przeszkód widzianych przez robota.

Węzeł *rviz* reprezentuje program RViz i subskrybuje dwa tematy - *tf* oraz *map*. Pozycje układów odniesienia reprezentowane są przez trójkolorowe trójwymiarowe obiekty, gdzie każdy z kolorów odpowiada jednej z osi (czerwony, zielony, niebieski odpowiadają kolejno osiom X, Y oraz Z). Ich położenie jest znane dzięki subskrypcji pierwszego z wymienionych tematów. Za pomocą danych przychodzących z drugiego z nich, program rysuje mapę pomieszczenia.

Struktura aplikacji sterującej

W tej sekcji przedstawiony zostanie zarys najbardziej znaczących klas aplikacji sterującej. Jako że sama jej budowa nie jest kluczowym aspektem niniejszej pracy, dlatego po szczegółowe informacje dotyczące jej funkcjonowania autor odsyła do kodu źródłowego znajdującego się w dodatku A.

Na diagramie 2.7 ukazano klasy głównego programu, z których najważniejsze zostaną omówione w tej sekcji. Głównym obiektem jest tutaj okno, instancja klasy *MainWindow*. Tutaj obsłużone jest kreowanie interfejsu graficznego, czyli ustawienie przycisków, pól tekstowych i innych elementów w odpowiednich miejscach, nadanie im identyfikatorów, ustawienie ciągów tekstowych, suwaków i innych elementów graficznych. Również akcje towarzyszące kliknięciom przycisków i przesuwaniu suwaków są tutaj przypisywane do odpowiednich funkcji. Funkcje te dla lepszej organizacji oraz czytelności zostały przeniesione do innych klas zorientowanych wokół konkretnych segmentów pracy aplikacji. Obiekty tych klas dostępne są



Rys. 2.6: Struktura środowiska ROS

poprzez zmienne wewnętrz główne klasy.

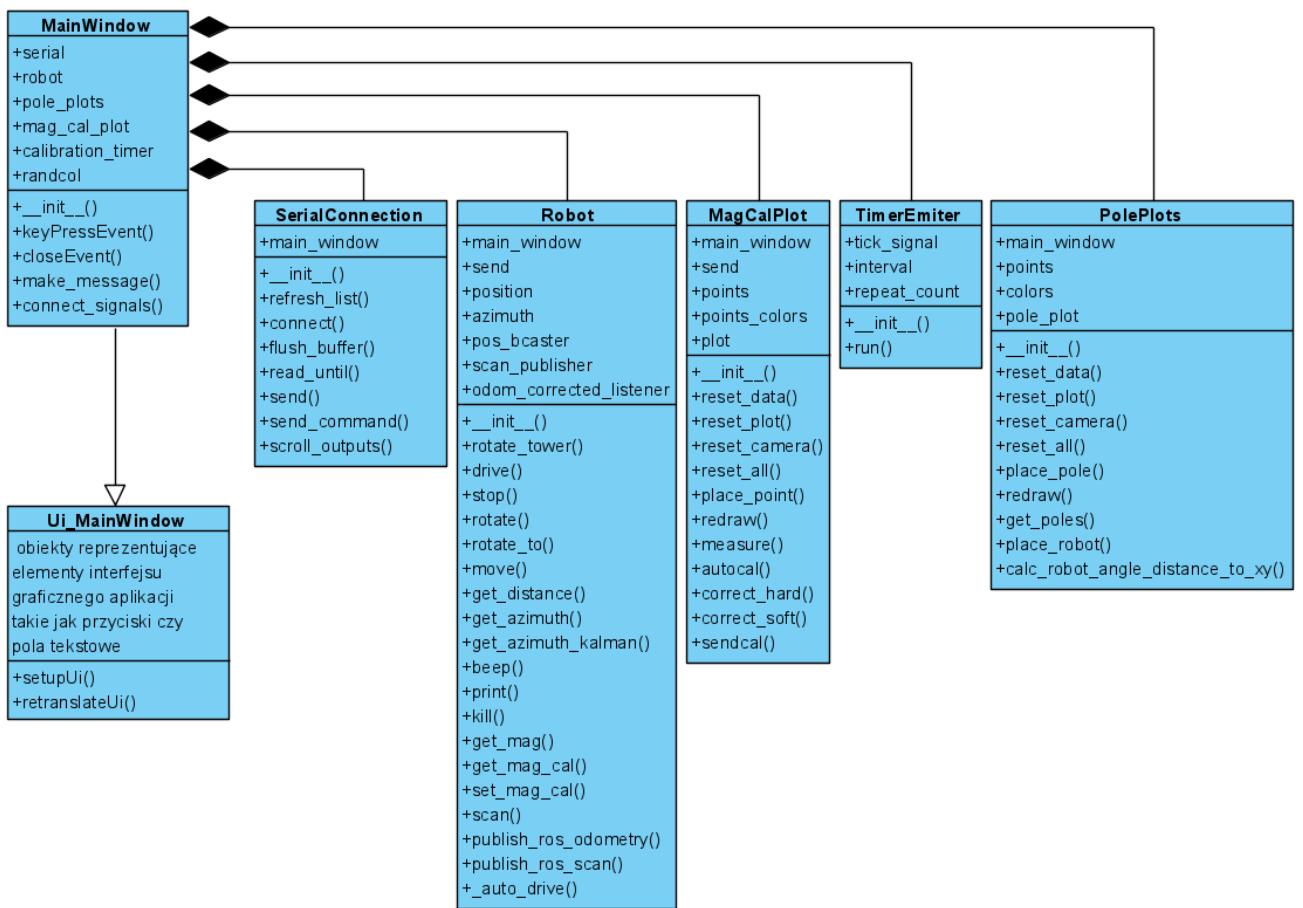
Klasa *MagCalPlot* obsługuje szereg funkcji związanych z kalibracją magnetometru ???. Zawiera przyciski uruchamiające funkcje związane z:

- Czyszczeniem wykresu
- Rysowaniem zmierzonych danych w postaci punktów na wykresie
- Automatycznym pomiarem surowych danych z magnetometru
- Ręcznym pomiarem surowych danych z magnetometru
- Korekcją błędu *hard iron* [51] [49]
- Korekcją błędu *soft iron* [49]
- Wysyłaniem danych kalibracyjnych do robota

Klasa *PolePlots* obsługuje rysowanie punktów oraz czyszczenie wykresu zakładki *map*. Przechowuje również dane o kolorach punktów jakie są na nim rysowane.

Klasa *Robot* odzwierciedla surowy interfejs robota, dodając warstwę abstrakcji na surowe komendy wydawane mobilnej platformie i rozszerzając je - przykładowo podczas każdego z wywołanych funkcją pomiarów azymutu, jest on od razu publikowany na temacie *tf*. Dodatkowo, w tej klasie obsłużony jest pomiar azymutu za pośrednictwem filtra Kalmana [50]. Również tutaj obsłużone jest publikowanie danych z odometrii i wykonywanych skanów na odpowiednich tematach, opisanych wcześniej w sekcji 2.1.1 jak i również algorytm samodzielnej jazdy robota.

Klasa *SerialConnection* służy do obsługi komunikacji interfejsu szeregowego konwertera USB-UART. Zawiera funkcje służące do nawiązywania połączenia, wysyłania komend i odbierania odpowiedzi od robota. Ponadto posiada uchwyt do klasy głównej, dzięki czemu wymieniane dane prezentuje zarówno w polu tekstowym w sekcji okna *CONSOLE* jak i w zakładce *raw output*.



Rys. 2.7: Diagram UML struktury klas aplikacji sterującej

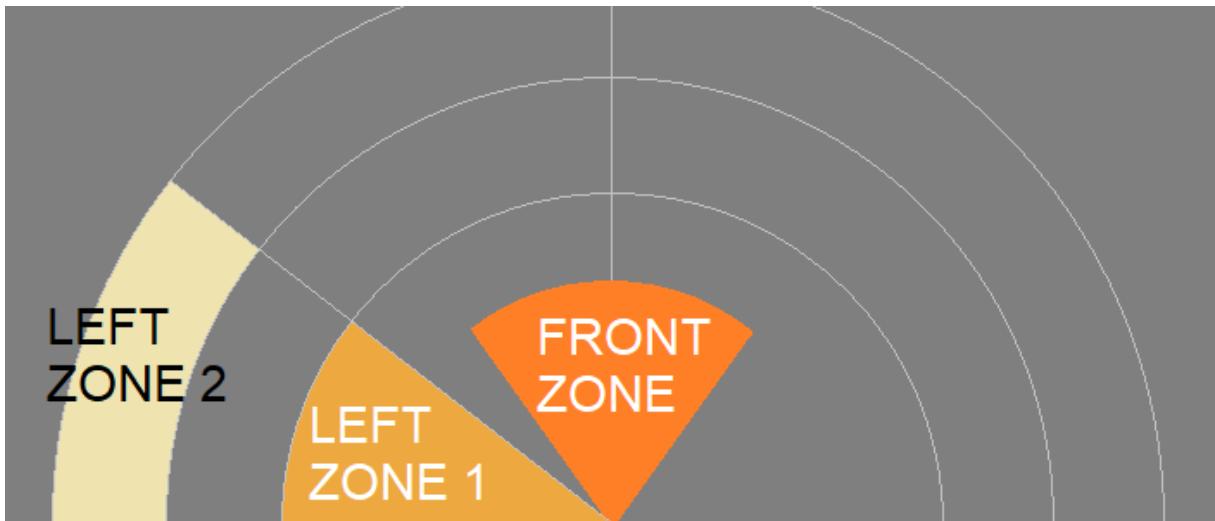
Klasa *TimerEmiter* jest wykorzystywana przy wykonywaniu ręcznych pomiarów podczas kalibracji magnetometru opisanej w rozdziale 3.1.1.

2.1.2. Jazda autonomiczna

W celu realizacji samodzielnej jazdy robot został wyposażony w autorski algorytm pozwalający na omijanie napotkanych przeszkód i poruszanie się wzduż ścian i podłużnych przeszkód. Możliwe było zastosowanie gotowego kodu programu, bądź implementację istniejących algorytmów w celu wykonania tego zadania, jednak z uwagi na edukacyjny charakter projektu autor postanowił opracować własne rozwiązanie. Operuje on na bardzo prostych zasadach - robot wykonuje kroki, pomiędzy którymi skanuje otoczenie i decyduje o podjętej akcji. Decyzja zapada na podstawie wykrycia przeszkód w wyznaczonych strefach. Ze względu na powiązanie z napisanym programem rysunek 2.8 przedstawia angielskie nazwy tychże stref. Ich kształt został wybrany ze względu na prostotę implementacji.

- *FRONT ZONE* (strefa F)
- *LEFT ZONE 1* (strefa L1)
- *LEFT ZONE 2* (strefa L2)

Należy zwrócić uwagę, że rysunek 2.8 ma charakter poglądowy i nie jest wykonany w skali. Platforma znajduje się tutaj w punkcie środkowym układu współrzędnych, rysunek przedstawia zakres kątów od 0° do 180° , liczonych od kierunku prawego przeciwne do ruchu wskazówek zegara. W tej samej kolejności przedstawiane są dane zeskanowane przez robota. Wymiary stref zostały dobrane metodą prób i błędów, dostrajane do momentu w którym robot



Rys. 2.8: Strefy algorytmu jazdy autonomicznej

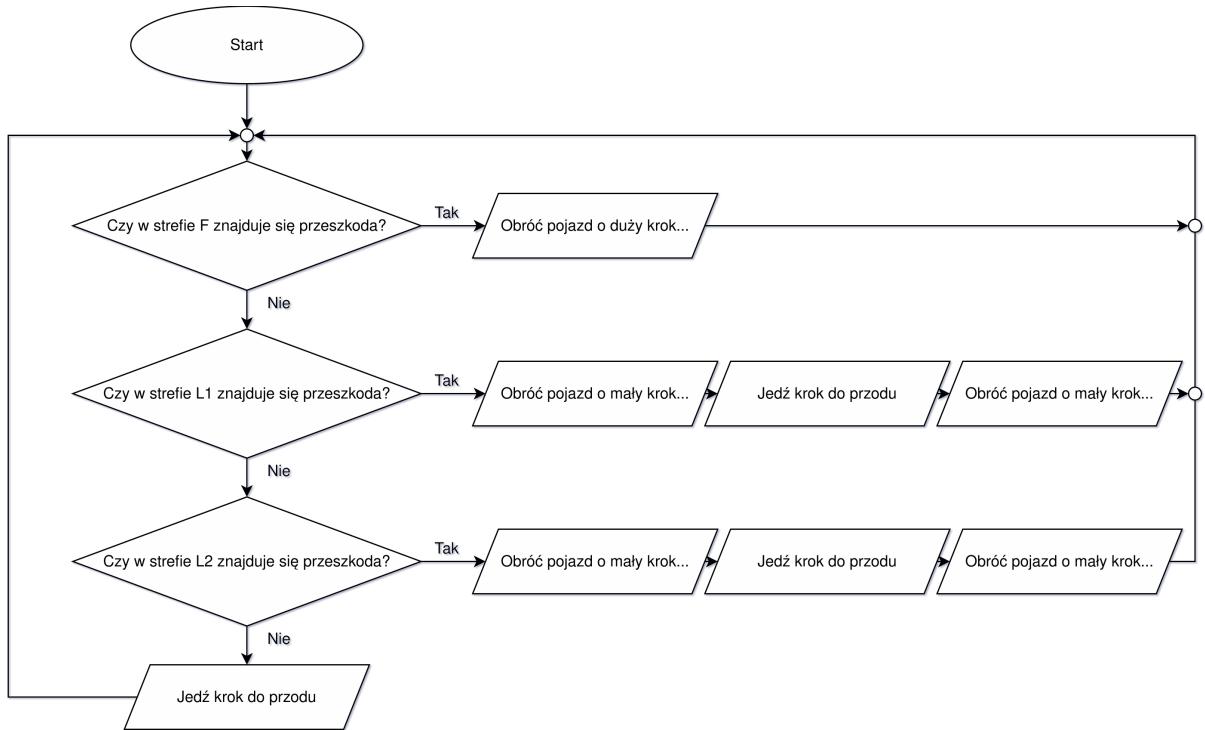
był w stanie samodzielnie okrążyć pokój bez wjeżdżania w przeszkody. Wielkości kroków to kolejne z parametrów którymi można regulować jakość pracy algorytmu. Rysunek 2.9 prezentuje jego funkcjonowanie w oparciu o wymienione poniżej wartości.

- zakres kątów obejmujących strefę F to $\langle 70^\circ, 110^\circ \rangle$
- zakres kątów obejmujących strefy $L1$ i $L2$ to $\langle 140^\circ, 180^\circ \rangle$
- promień strefy F wynosi 30 cm
- promień strefy $L1$ wynosi 50 cm
- strefa $L2$ jest wycinkiem zaczynającym się od promienia równego 70 cm, kończąca się na 90 cm
- mały krok jazdy do przodu wynosi 10 cm
- mały krok obrotu wynosi 10°
- duży krok obrotu wynosi 30°

2.2. Firmware

Dzięki zastosowaniu mikrokontrolera jako jednostki sterującej mobilną platformą skanującą po bór energii elektrycznej jest bardzo małe (bez peryferiów pobiera prąd o wartości <51mA[2]). Nakłada to niestety ograniczenia związane z mocą obliczeniową urządzenia i dostępną pamięcią w której przechowywany jest program.

Pierwszym kierunkiem w którym udał się autor były popularne dziś płytki Arduino, czego głównym powodem jest łatwość szybkiego prototypowania i szeroki wachlarz gotowych rozwiązań obsługujących różne sensory, serwomechanizmy, wyświetlacze i wiele innych. Płytnka nie powinna być zbyt duża, dlatego na uwadze były mniejsze egzemplarze takie jak np. Arduino Nano. Jednak ze względu na niskie taktowanie procesora (16MHz) i niewiele pamięci Flash (32kB) na program wybrana została alternatywa - tzw. płytka "blue pill"[36]. Jest to nieoficjalna płytka, podobna do Arduino. Można ją zaś programować w tym samym środowisku i wykorzystywać wiele (choć nie wszystkie) z bibliotek napisanych pod oficjalnie wspierane płytki. Jest tak dzięki zestawowi bibliotek STM32duino [27] zapewniających kompatybilność z wcześniejszym wspomnianym środowiskiem. Do komplikacji i wgrania programu zostało wykorzystane kompatybilne z Arduino - PlatformIO IDE [32].



Rys. 2.9: Algorytm jazdy autonomicznej

Program działający na platformie wykonuje proste polecenia odebrane od aplikacji sterującej i odpowiada prostym potwierdzeniem, wartością lub listą wartości. Składnia prezentuje się następująco:

KOMENDA:[PARAMETR_1][,PARAMETR_2][,PARAMETR_N]#

Każdy parametr oddzielony jest przecinkiem, w przypadku jego braku po komendzie od razu następuje znak # bez dwukropka. Poniżej wylistowano wszystkie dostępne komendy wraz z parametrami:

1. *DRIVE* - ustaw zadaną prędkość na gąsienicach. Przyjmuje dwa parametry liczbowe oznaczające prędkość lewej i prawej gąsienicy. Zakres wartości obu parametrów to $\langle -90, 90 \rangle$
2. *ROTATE_TOWER* - obróć wieżyczkę na żądaną pozycję. Przyjmuje jeden parametr liczbowy w zakresie $\langle 0, 180 \rangle$.
3. *KILL* - odłącz sygnał sterujący od serwomechanizmów.
4. *PRINT* - pokaż tekst na wyświetlaczu. Przyjmuje jeden parametr tekstowy jakim jest tekst do wyświetlenia.
5. *BEEP* - wydaj serię dźwięków o zdefiniowanej liczbie i czasie trwania. Przyjmuje dwa parametry liczbowe - pierwszy oznacza czas trwania impulsu w milisekundach, drugi liczbę impulsów.
6. *GET_MAG* - zwróć surowy pomiar z magnetometru. Zwraca dwa parametry, kolejno wartości pomiaru dla osi X i Y.
7. *GET_AZIMUTH* - zwróć azymut w którym skierowany jest robot. Zwracana wartość wyrażona jest w stopniach.
8. *GET_DISTANCE* - zmierz odległość od przeszkody. Zwraca wartość w centymetrach.
9. *GET_TIME* - zwróć wartość czasu od uruchomienia robota. Wartość wyrażona jest w milisekundach.

10. *MOVE* - przemieśc się o zadany dystans do przodu lub do tyłu. Przyjmuje jeden parametr jakim jest dystans w centymetmach.
11. *ROTATE_TO* - obróć się na zadany azymut. Przyjmuje jeden parametr wyrażony w stopniach.
12. *ROTATE* - obróć się o kąt. Przyjmuje jeden parametr wyrażony w stopniach.
13. *SCAN* - skanuj otoczenie. Zwraca serię pomiarów dla każdego z n kątów, gdzie $n \in \langle 0, 180 \rangle$. Pomiarzy oddzielone są przecinkami i wyrażone w centymetmach.
14. *SET_MAG_CAL* - ustaw wartości do kalibracji magnetometru. Przyjmuje cztery parametry - ujemną wartość przesunięcia X, ujemną wartość przesunięcia Y, parametr θ i parametr σ objaśnione dalej w rozdziale 3.1.1.
15. *GET_MAG_CAL* - pobierz wartości kalibracji magnetometru z robota. Zwracane parametry mają taką samą formę jak przyjmowane przez wyżej wymienione polecenie.
16. *RESET* - uruchom ponownie robota.

2.3. Budowa platformy

Budowa robota wymaga zainwestowania pewnej ilości środków finansowych w materiały, zużycie prądu oraz potrzebne narzędzia. Do wykonania platformy skanującej potrzebne są takie akcesoria jak:

- Drukarka 3D wraz z odpowiednim oprogramowaniem
- Akcesoria lutownicze - cyna, kalafonia, obcinak, ściągacz do izolacji
- Żywica epoksydowa, klej cyjanoakrylowy

Ze względów praktycznych spis części przedstawiony w tabeli 2.1 uwzględnia jedynie ceny materiałów.

Tab. 2.1: Spis części potrzebnych do budowy robota

Liczba Sztuk	Nazwa	Cena [zł]
1	Płytką “blue pill“ STM32F103 [23]	14,61
2	Moduł Bluetooth HC-05 [18]	35,90
1	Wyświetlacz OLED [43]	26,90
2	Przetwornica step-down [21]	7,80
1	Sensor Lidar TF Luna [29]	89,00
1	Filament DevilDesign PLA żółty [15]	79,99
1	Filament DevilDesign TPU czarny [16]	69,99
2	Ogniwo Litowo-Jonowe [19]	26,90
2	Para konektorów XT30 [20]	2,39
1	Ładowarka pakietów Litowo-Jonowych [45]	85,90
1	Zasilacz do ładowarki pakietów [44]	25,90
1	Serwomechanizm TowerPro MG-90S [30]	14,90
2	Serwomechanizm TowerPro MG-995 [31]	21,89
1	Przycisk wł/wył z podświetleniem [22]	33,34
1	Płytką uniwersalną dwustronną [24]	3,90
1	Rolka przewodu drucianego [28]	29,90
1	Konwerter USB-UART [17]	23,50
1	Brzęczyk z generatorem [1]	0,90
2	Enkoder EC-11 [14]	3,90
SUMA:		696,29

Płytką “blue pill“ została wybrana ze względów zarówno ekonomicznych jak i praktycznych. Jest stosunkowo tania i posiada wystarczająco szybki procesor i dostateczny zestaw interfejsów do obsługi pozostałych komponentów.

Moduł Bluetooooth HC-05 został wykorzystany ze względu na swoją niską cenę i popularność. Jest niezawodny i prosty w konfiguracji. Autor wykorzystywał go do wielu innych projektów i nie miał wątpliwości że zastosowanie go również w niniejszym projekcie będzie odpowiednie.

Wyświetlacz OLED również jest modułem niedrogim, korzysta z popularnego sterownika *SSD1306*. Jego zastosowaniem jest wyświetlanie wybranych parametrów podczas procesu *debugowania*, czyli usuwania błędów z kodu programu.

Przetwornice typu *step-down* pozwalają na regulację napięcia wyjściowego względem wejściowego, z akumulatora. Zastosowanie dwóch sztuk ma na celu separację zasilania serwomechanizmów od sensorów i płytki sterującej - po to, aby nie wprowadzać niepotrzebnych zakłóceń do elementów logicznych.

Marka filamentu stosowanego przy wydruku 3D nie ma większego znaczenia. Te przedstawione w spisie są subiektywną preferencją autora. Podobnie stosowane ogniwa Litowo-Jonowe, jednak ważne jest aby były w stanie zapewnić natężenie prądu powyżej 4 Amperów - ta wartość nie została konkretnie ustalona, jej dobór wynika z pewnych doświadczeń autora w podobnych projektach. Zgrubna zasada stosowana przy tym projekcie to zapewnienie dostępnego natężenia prądu znacznie większego niż suma poborów prądu elektrycznego poszczególnych komponentów (oddzielnie dla sekcji logicznej i serwomechanizmów). Przetwornica części logicznej została ustawiona na napięcie 5V, natomiast dla serwomechanizmów 6V dla zwiększenia momentu obrotowego.

Wybrana ładowarka wraz z kompatybilnym dla niej zasilaczem również nie jest jedyną możliwą do zastosowania - tutaj również czynnikiem decyzyjnym było doświadczenie autora, jak i również pewne wzgłydy ekonomiczne.

Serwomechanizm TowerPro MG-90S jest niedrogim i kompaktowym urządzeniem pozwalającym na obrót wieżyczki, zaś TowerPro MG-995 posiada odpowiedni moment obrotowy, jaki (drogą prób i błędów) okazał się wystarczający do poruszania platformy.

Przycisk załączający zasilanie posiada podświetlenie ze względów estetycznych. Natężenie prądu jakie jest w stanie przewodzić powinno spełniać takie same kryteria jak w przypadku przetwornic.

Płytki uniwersalna i przewód druciany posłużą połączeniu wszystkich elementów. W przypadku serwomechanizmów ze względu na większe natężenie prądu wykorzystano przewody do nich dołączone.

Konwerter USB-UART ułatwi procedurę połączenia platformy z aplikacją sterującą. Po połączeniu go z jednym z modułów HC-05 i komputerem zostanie nawiązane połączenie między dwoma modułami Bluetooth niezależnie od ustawień komputera. Pozwoli to na proste oraz szybkie wznowienie połączenia po restarcie aplikacji. Konwerter po podłączeniu widoczny jest w systemie operacyjnym jako port komunikacji szeregowej (`/dev/ttyUSBX` dla systemu GNU/Linux, gdzie X jest numerem każdorazowo przypisywanym do podłączonego urządzenia).

Brzęczyk z generatorem pozwala na wytwarzanie prostych komunikatów dźwiękowych. Jest kolejnym elementem przydatnym przy *debugowaniu* kodu programu, ponadto w prosty sposób może sygnalizować np. gotowość robota do pracy.

Enkoder obrotowy EC-11 jest niedrogim urządzeniem mechanicznym, służącym do zliczania obrotów. Pomoże w implementacji nawigacji zliczeniowej.

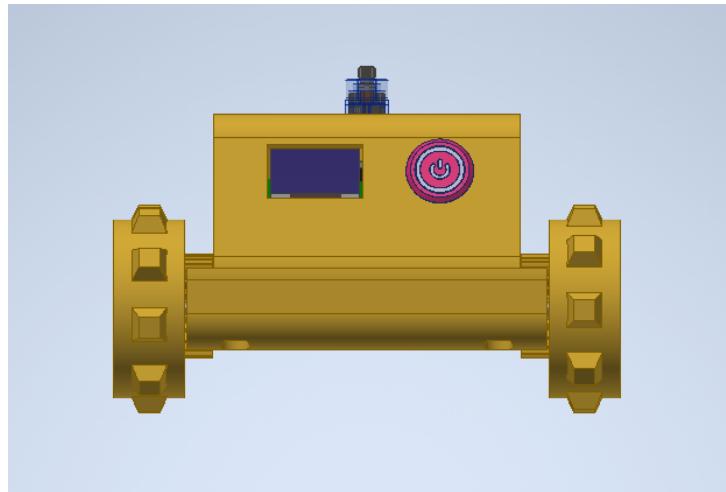
2.3.1. Budowa mechaniczna

Dużą przeszkodą w nawigacji zliczeniowej jest kumulujący się błąd. Wynika on z czynników niemożliwych do zwalczenia w całości, które są naturalne w każdym fizycznym obiekcie. Do tych czynników można zaliczyć:

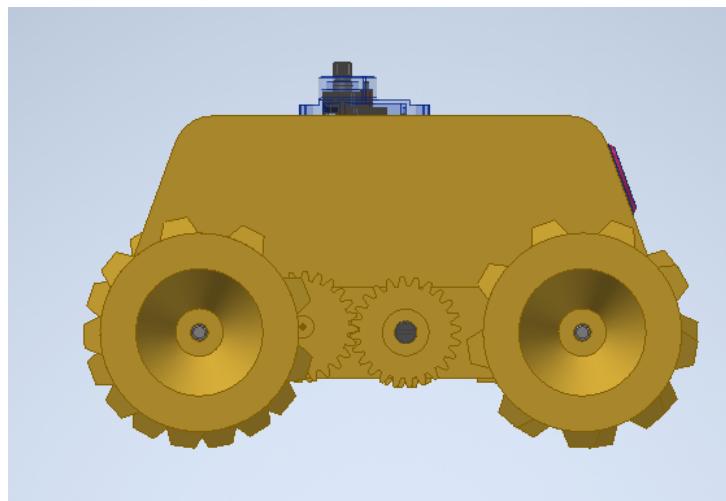
1. poślizg kół pojazdu
2. bezwładność - brak możliwości niezwłocznego zatrzymania pojazdu w punkcie
3. niedokładność urządzeń pomiarowych wynikającą z ich fizycznych właściwości, m. in. szumów i zakłóceń

Do pierwszych dwóch wymienionych czynników bezpośrednio odwołuje się jedna z decyzji projektowych. W pojeździe zostanie zastosowany gąsienicowy układ bieżny. Zwiększenie powierzchni kontaktu z podłożą zapewni bardziej stabilny obrót i jazdę platformy. Pomoże również w szybszym jej zatrzymaniu. Do trzeciego punktu sam projekt mechaniczny nie ma żadnego odniesienia, ta kwestia zostanie omówiona w podrozdziale ??.

Projekt został wykonany w programie Autodesk Inventor [40]. Pliki projektu są dostępne w dodatku A . Rzuty projektu przedstawione są na rysunkach 2.10, 2.11 i 2.12. Robot powstanie na drukarce 3D działającej w technologii FDM (ang. *Fused Deposition Modeling*) z materiału PLA (ang. PolyLactic Acid).



Rys. 2.10: Rzut główny platformy



Rys. 2.11: Rzut platformy z prawej strony

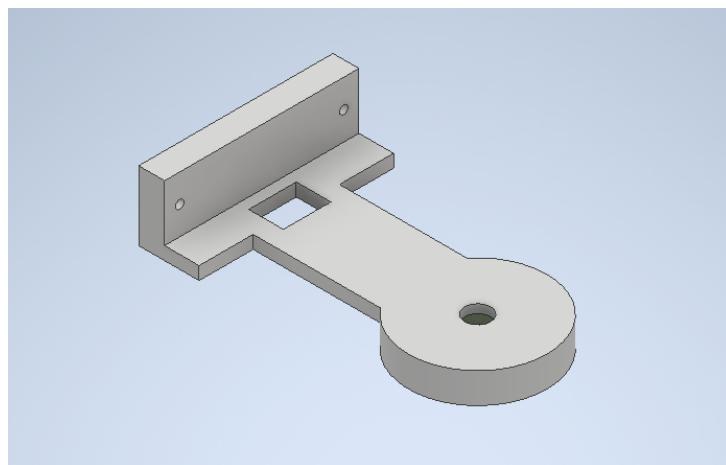


Rys. 2.12: Rzut platformy z góry

Na rysunkach 2.11 i 2.12 uwidocznione są przekładnie zębate. Te znajdujące się na środku są przekładniami zapewniającymi napęd - połączono je z serwomechanizmami napędowymi. Z jednej strony następuje przełożenie w stosunku 1:1 na tylne koła. Bliżej przedniej części robota (po której znajduje się wyświetlacz) następuje przełożenie na koła zębate przymocowane do enkoderów obrotowych, z takim samym stosunkiem jak w poprzednim przypadku. Ma to na celu synchronizację położenia kół napędowych i enkoderów. Dzięki takiemu rozwiązaniu pośrednio mierzony jest obrót kół i możliwa jest implementacja nawigacji zliczeniowej.

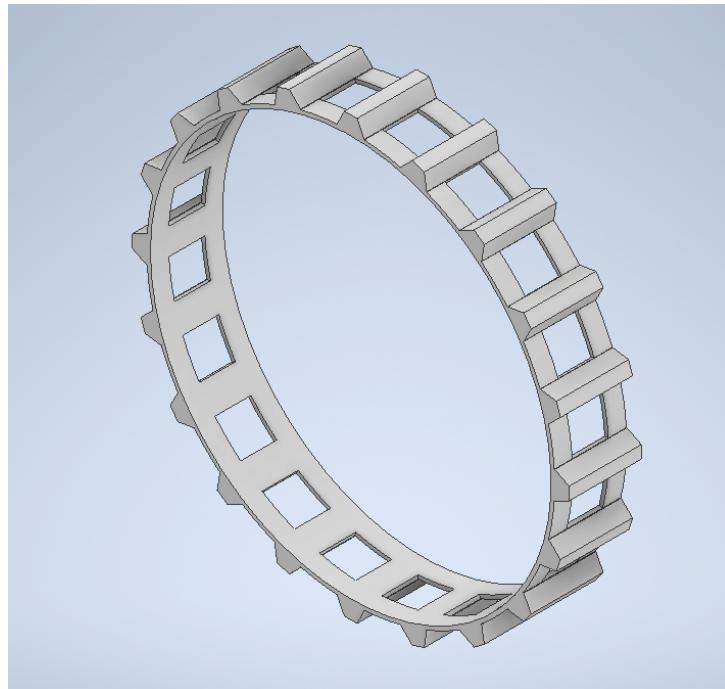
Przedni panel zawiera wyświetlacz służący prezentacji danych przydatnych przy rozwijaniu oprogramowania i procedurze usuwania błędów z kodu poprzez wyświetlanie pożądanych wartości na ekranie. Podświetlany przycisk znajduje się po prawej stronie wyświetlacza i służy do załączania zasilania.

Wieżyczka przedstawiona w rzucie izometrycznym na rysunku 2.13 została zaprojektowana osobno. Sensor jest zamontowany w przesunięciu względem osi w celu częściowej redukcji jego martwej strefy. Powstał w ten sam sposób i z tego samego materiału co reszta platformy.



Rys. 2.13: Rzut izometryczny wieżyczki

Gąsienica widoczna na rysunku 2.14 wykonana jest w innym materiale - elastycznym TPU (ang. *Thermoplastic PolyUrethane*). Potrzebne były dwa egzemplarze - po jednym na stronę. Aby zniwelować poślizg względem kół a także przenieść napęd z tylnych na przednie, jej projekt posiada otwory wpasowujące się w wypustki na kołach. Dla redukcji poślizgu względem miękkich podłoży sama posiada wypustki na powierzchni zewnętrznej.

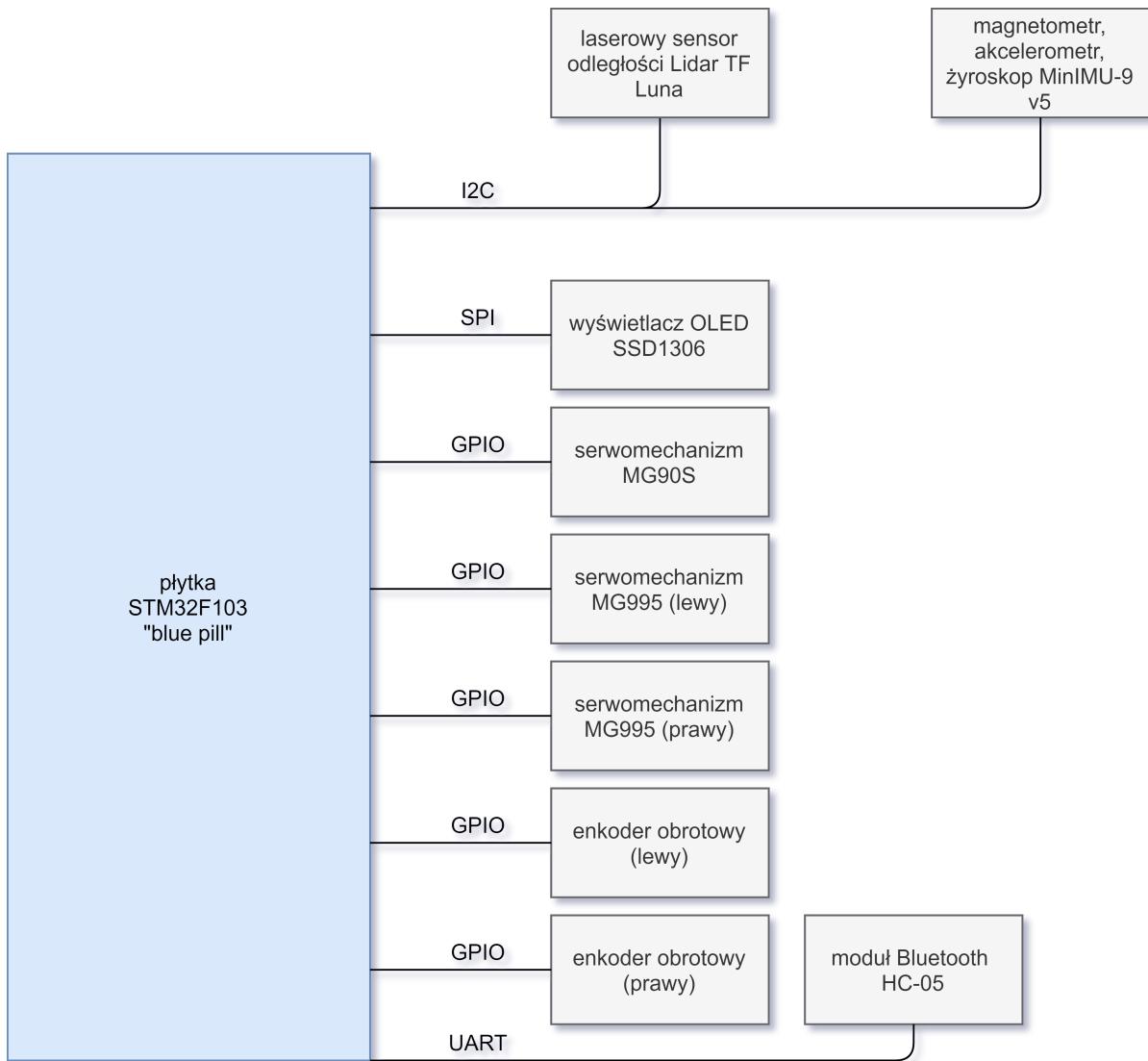


Rys. 2.14: Rzut izometryczny gąsienicy

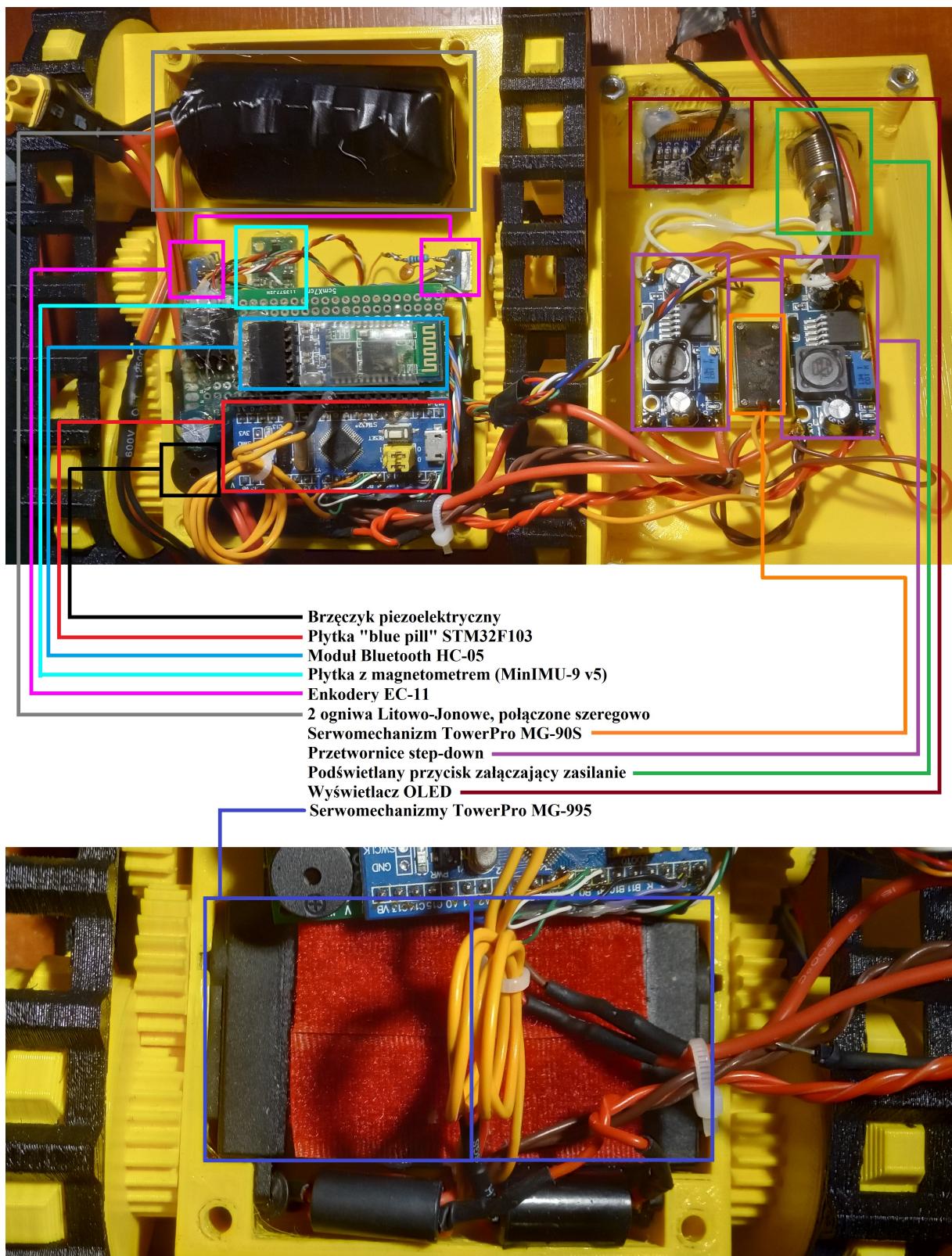
2.3.2. Budowa elektroniczna

Elementy elektroniczne robota połączono bez użycia płyt drukowanych. Płytkę z mikrokontrolerem została osadzona w płytce uniwersalnej i przylutowana za pomocą cyny. Wszelkie peryferia podłączono na stałe (przylutowano) bądź za pomocą złączy typu goldpin, o rozstawie 2,54mm. Dla czytelności schemat 2.15 przedstawia uproszczony rozkład połączeń sensorów z płytą sterującą "blue pill" oraz zastosowane interfejsy komunikacji.

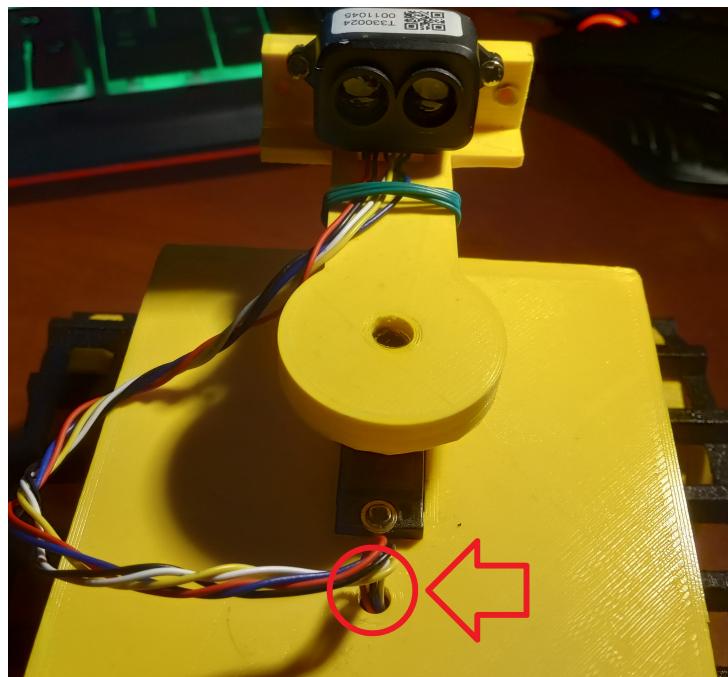
Rozmieszczenie elementów w przestrzeni wewnętrz obudowy robota obrazuje rysunek 2.16. Z racji, że sensor laserowy znajduje się na zewnątrz, przewody komunikacyjne zostały wyrowadzone przez otwór w obudowie widoczny na rysunku 2.17.



Rys. 2.15: Schemat peryferiów podłączonych do jednostki sterującej



Rys. 2.16: Lokalizacja elementów elektronycznych w robocie



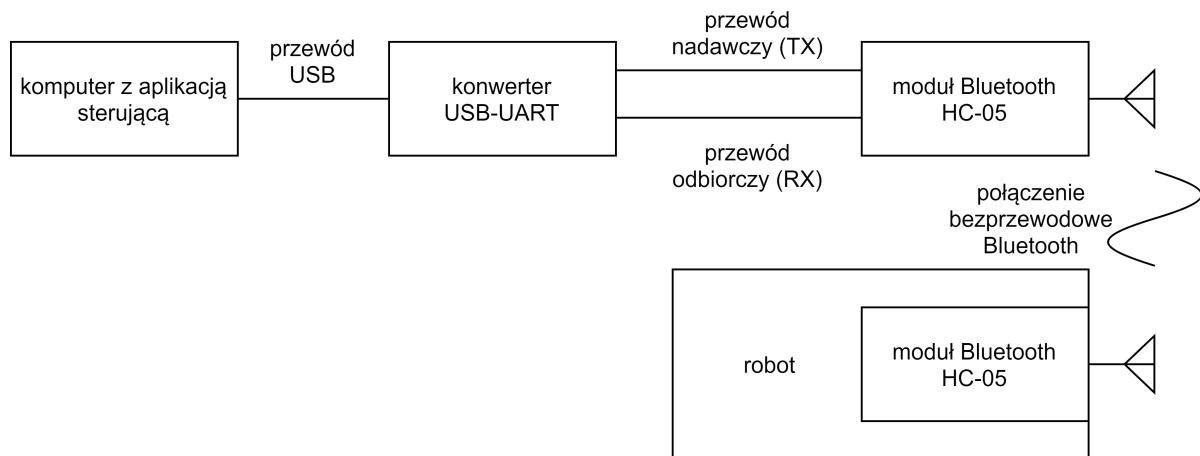
Rys. 2.17: Przewody sygnałowe i zasilające sensora laserowego

Rozdział 3

Ewaluacja

Poniższy rozdział opisuje sposób w jaki sposób w ramach powstałego projektu przebiegają procedury mające na celu stworzenie możliwie najbardziej oddającą rzeczywistość mapy pomieszczenia. Autor przedstawia problemy które napotkał dążąc do celu, ich genezę oraz konieczne zmiany i udoskonalenia. Również przybliżone zostały aspekty korzystania z poszczególnych dostępnych funkcjonalności.

Aby rozpocząć proces ewaluacji i udoskonalania projektu robot powinien mieć zapewnione stabilne połączenie z komputerem na którym pracuje środowisko sterujące. W tym celu po otwarciu okna głównego aplikacji należy skorzystać z opisanej wcześniej sekcji *CONNECTION* uprzednio łącząc elementy systemu w sposób przedstawiony na rysunku 3.1. Następnie należy wybrać odpowiedni port szeregowy i kliknąć przycisk służący do nawiązania połączenia. Po kilku sekundach, jeżeli procedura przebiegła pomyślnie, platforma wyda trzy krótkie sygnały dźwiękowe - oznacza to gotowość do przyjmowania poleceń.



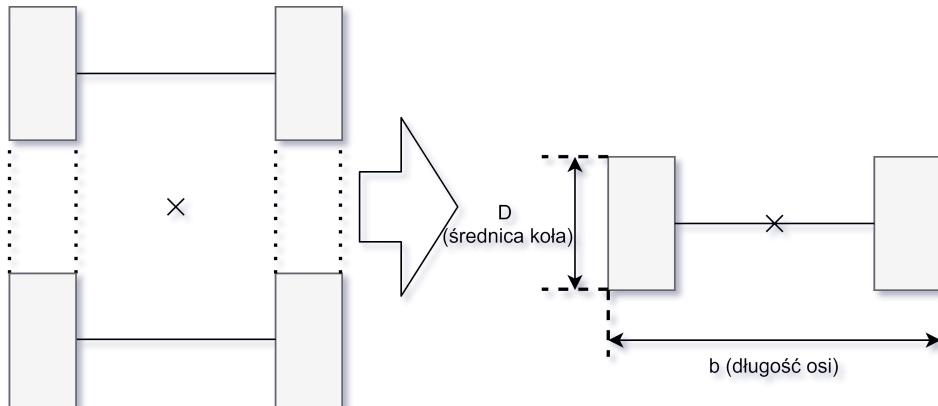
Rys. 3.1: Schemat połączenia z robotem

Opisana procedura jest konieczna przy każdym uruchomieniu systemu. Mając to na uwadze, można przystąpić do ewaluacji opisanej w kolejnych rozdziałach.

3.1. Odometria

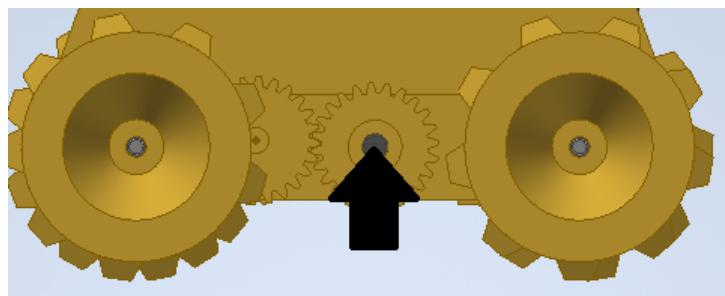
Robot korzysta z nawigacji zliczeniowej w celu połączenia pomiarów odległości zebranych z otoczenia i aproksymacji ich położenia na płaszczyźnie względem jednego ustalonego punktu. Zliczaniu podlegają dwie wartości - dystans przejechany wzdłuż oraz obrót pojazdu w miejscu. Szereg zmierzonych w ten sposób wartości umożliwia odtworzenie przejechanej ścieżki.

Rysunek 3.2 przedstawia rzeczywisty oraz uproszczony model podwozia robota. X na rysunku oznacza środek robota - pionową oś wokół której się obraca. Zamiast uwzględniać obie poziome osi i 4 koła, upraszcza się go do formy robota o jednej osi, z jedną parą kół. Gdy oba koła poruszają się w tę samą stronę, robot przesuwa się w przód lub w tył. Gdy pracują przeciwnie - obraca się wokół osi oznaczonej X.



Rys. 3.2: Model rzeczywisty i uproszczony

Pierwszym pomysłem było wykorzystanie enkoderów do pomiaru translacji pojazdu wzdłuż jego osi ruchu, pozostawiając zliczanie obrotu funkcjom korzystającym z magnetometru.



Rys. 3.3: Umiejscowienie enkodera z prawej strony robota

Enkodery zostały zamontowane w miejscu przedstawionym na rysunek 3.3, symetrycznie po obu stronach podwozia. Napęd podany od serwomechanizmu przez przekładnię żebatą przenosi napęd zarówno na tylne koło jak i enkoder, z przekładniami 1:1 w obu przypadkach. Dzięki takiemu rozwiązaniu pełny obrót enkodera odpowiada pełnemu obrotowi koła. Enkoder obrotowy EC-11 generuje 20 impulsów przy kącie obrotu 360° . Aby można było wyznaczyć pokonaną odległość, w pierwszej kolejności należy obliczyć stosunek ilości impulsów do przejechanego dystansu - w tym celu autor stworzył prosty wzór (3.1).

$$DPR = \frac{2\pi r}{p} \quad (3.1)$$

DPR - współczynnik (ang. Distance to Pulse Ratio) wyrażany w cm/impuls

r - promień koła pojazdu

p - liczba impulsów enkodera na obrót koła

Wiedząc ile impulsów generuje enkoder, oraz znając wymiary koła w łatwy sposób można obliczyć DPR, co uczyniono poniżej (3.2).

$$\begin{aligned} DPR &= \frac{2\pi 2,5cm}{20imp} \\ DPR &= 0.785 \frac{cm}{imp} \\ \frac{1}{DPR} &= 1.274 \end{aligned} \quad (3.2)$$

Znając współczynnik aby obliczyć przejechany dystans wystarczy zmierzyć liczbę impulsów jakie wystąpiły podczas przejazdu a następnie pomnożyć je przez DPR. Otrzymany wynik oznacza przesunięcie robota wyrażone w centymetrach. Jako że platforma posiada dwa enkodery, a nie jest możliwe zachowanie idealnie prostego toru jazdy, wyciągana jest średnia liczba impulsów. Ze względu na grubość i wypustki na gąsienicach oraz ich poślizg rzeczywisty dystans przejechany będzie inny od zadanego. Dla kompensacji tej różnicy wartość DPR została skorygowana ręcznie do takiej, przy której błąd przemieszczenia robota zawierał się w granicy $\pm 10\%$ na zadanym dystansie 100cm. Widoczna w kodzie programu, skorygowana wartość $\frac{1}{DPR}$ wynosi 1.325 co odpowiada $DPR = 0.755$. Listing 1 prezentuje funkcję realizującą to zadanie.

```
//MOVE
case 9:
{
    reset_encoders();
    float val = getArgument(command, 1).toInt() * 1.325;

    if (val > 0)
        driveMotors(90, 90);
    else
        driveMotors(-90, -90);

    val = abs(val);
    while ((left_encoder_counter + right_encoder_counter) / 2 < val)
    {
    }

    driveMotors(0, 0);
    delay(100);
    Serial2.println("OK");
}
```

Listing 1: Fragment kodu obsługującego polecenie *MOVE*

Podczas wykonywania polecenia *MOVE* na gąsienice zadawana jest pełna prędkość i w pętli sprawdzana jest średnia liczba impulsów wygenerowanych przez enkoder. Dla optymalizacji algorytmu, zamiast przeliczać przy każdym pomiarze liczbę impulsów razy wartość DPR, na początku zadana w parametrze wartość odległości jest mnożona przez $\frac{1}{DPR}$, i dalej w takiej formie ta wartość wykorzystywana przy operacji porównania. W momencie w którym średnia zliczona liczba impulsów przekroczy jej wartość, serwomechanizmy zostają zatrzymane.

Obrót (funkcja *ROTATE*) polega na zadaniu przeciwnych wartości prędkości na obie gąsienice. W celu obrotu poruszają się one przeciwbieżnie, z równymi prędkosciami. Pierwsza implementacja funkcji obrotu wyglądała jak przedstawiono poniżej:

Na początku mierzony jest azymut początkowy i obliczany jest azymut końcowy (tj. początkowy + zadana wartość obrotu). Dalej wykonywana jest ta sama funkcja co w przypadku

```

void rotateTo(int azimuth)
{
    if (calcAngleDistance(getAzimuth(), azimuth) > 0)
        driveMotors(-90, 90);
    else
        driveMotors(90, -90);

    while (true)
    {
        if (abs(calcAngleDistance(getAzimuth(), azimuth)) <= 10)
            break;
        delay(10);
    }
    driveMotors(0, 0);
    delay(100);
}

```

Listing 2: Funkcja będąca głównym elementem obsługi polecień *ROTATE* oraz *ROTATE_TO*

ROTATE_TO przyjmująca parametr azymutu końcowego. Podczas obrotu, w pętli, sprawdzana jest różnica kąta aktualnego od zadanego. Jeżeli wartość bezwzględna obrotu znajdzie się w zakresie $\pm 10^\circ$ robot zatrzyma się.

Takie rozwiązanie jest dobre, o ile magnetometr jest skalibrowany i funkcjonuje poprawnie. Niestety, podczas skanów okazało się że nie można polegać na pomiarach z tego sensora - więcej o tym znajduje się w sekcji 3.2. Z tego powodu koniecznym okazała się zmiana podejścia do pomiaru obrotu.

Listing 3 przedstawia nowe podejście obsługi polecenia *ROTATE*. Jest ono mniej dokładne od poprzedniego, bardziej podatne na dryf (błąd obrotu kumuluje się), natomiast taki pomiar jest odporny na zakłóczenia pola magnetycznego. Tym razem procedura jest analogiczna jak w przypadku ewaluacji polecenia *MOVE*. W pętli sprawdzana jest średnia liczba impulsów, jednak tym razem gąsienice poruszają się przeciwnie. Kąt obrotu na początku należy przemożyc przez pewien współczynnik tak aby odpowiadał on ilości impulsów. Platforma kończy ruch w momencie gdy średnia wartość przekroczy obliczony próg impulsów.

```

// ROTATE
case 11:
{
    reset_encoders();
    float val = getArgument(command, 1).toInt() * 0.164;
    if (val > 0)
        driveMotors(-90, 90);
    else
        driveMotors(90, -90);

    val = abs(val);
    while ((left_encoder_counter + right_encoder_counter) / 2 < val)
    {

        delay(100);
        driveMotors(0, 0);
        Serial2.println("OK");
    }

    break;
}

```

Listing 3: Nowa implementacja obsługi polecenia *ROTATE*

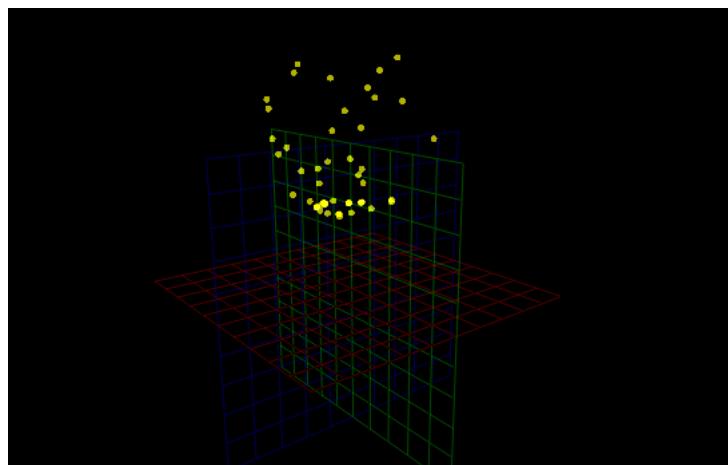
Stosunek kąta obrotu do liczby impulsów (0,164) został wyznaczony eksperymentalnie. Najpierw przyjęto wartość 1. Następnie zadawany był kąt obrotu 90° , po czym mierzony był rzeczywisty kąt obrotu. Jeżeli wynosił on więcej, współczynnik redukowano o 0,1; analogicznie gdy obrót był mniejszy niż zadano. Gdy tak zgrubna regulacja była niewystarczająca (albo za mały albo za duży obrót), krok został zmniejszony do 0,05. Po kolejnej zmianie kroku do 0,02 czynność powtarzano do momentu w którym dalsza korekcja nie była konieczna.

Taka implementacja została zachowana do końca projektu. Polecenie *ROTATE_TO* dalej korzysta ze starego sposobu z użyciem magnetometru, jednak nie jest ono wykorzystywane podczas pomiarów.

3.1.1. Kalibracja magnetometru

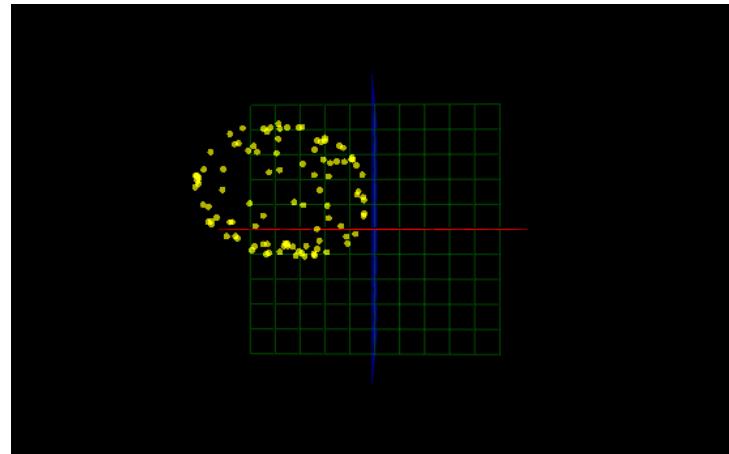
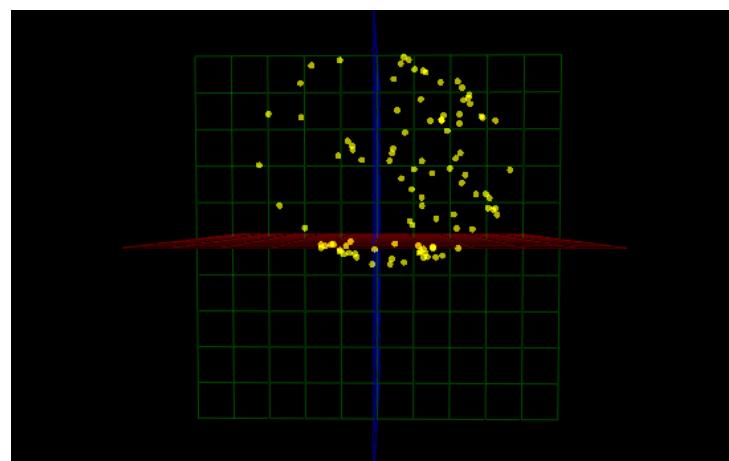
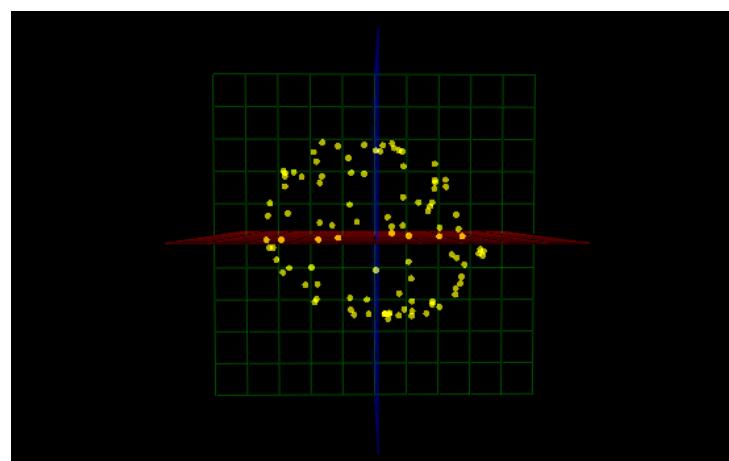
Wykorzystywany w projekcie moduł magnetometru jest urządzeniem czułym i bardzo podatnym na zakłócenia - szczególnie jeśli mierzony jest pole magnetyczne ziemi. Poza zakłóceniami z otoczenia, występują również te wynikające z samej budowy robota - wszelkie przewody przez które płynie prąd generują swoje własne pola, nawet same metalowe elementy interferują z pomiarem. Na te zakłócenia należy zwrócić uwagę i podjąć kroki mające na celu ich kompensację. Warto też wspomnieć, że w tej pracy jednostki uzyskane z pomiaru nie mają znaczenia - jedyne co jest potrzebne w celu uzyskania informacji o kierunku w którym robot jest zwrócony to kierunek i zwrot zmierzzonego wektora wartości natężenia pola.

Pierwszym problemem są zakłócenia typu *hard iron*. Ich obecność przejawia się w postaci stałego przesunięcia mierzonych we wszystkich trzech osiach wartości natężenia pola magnetycznego. Zmierzona wartości można przedstawić na trójwymiarowym wykresie (jak uczyniono w pierwszych wersjach aplikacji sterującej). Każda z osi wykresu odpowiada osi pomiaru natężenia pola magnetycznego. W idealnej sytuacji zbór punktów powinien być osadzony na sferze o środku w punkcie (0, 0, 0). Tak się jednak nie dzieje, co jest widoczne na rysunku 3.4.



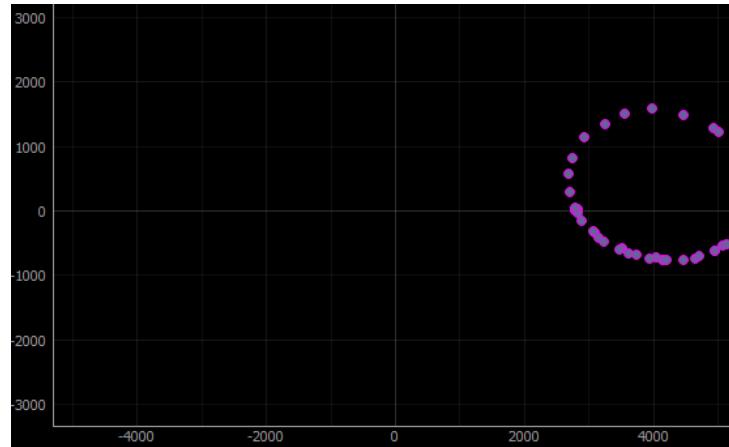
Rys. 3.4: Wykres obrazujący dane pozyskane z nieskalibrowanego magnetometru

Aby dokonać korekcji *hard iron* [51][49] dla każdej osi z osobna należy wyznaczyć minimalną i maksymalną zmierzona wartość. Sumę obu wartości dzieli się przez 2, a uzyskana liczba to przesunięcie (ang. *offset*). Aplikacja korekcji polega na odjęciu tej liczby od zmierzonyj wartości. Tą procedurę dla trzech osi przedstawiają kolejno rysunki 3.5, 3.6 i 3.7.

Rys. 3.5: Korekta przesunięcia *hard iron* dla osi XRys. 3.6: Korekta przesunięcia *hard iron* dla osi YRys. 3.7: Korekta przesunięcia *hard iron* dla osi Z

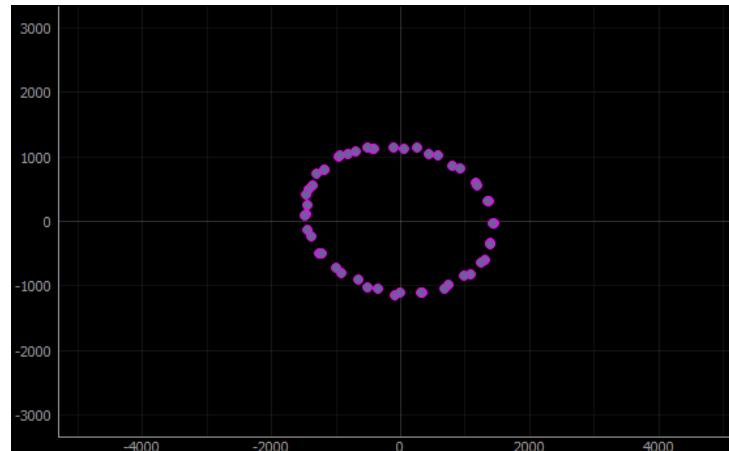
Niestety, kształt chmury punktów wciąż daleki jest od idealnej sfery. Efektem tego jest neliwiwość uzyskanego azymutu - zmiana rzeczywistego kąta skierowania platformy względem zmiany zmierzanej będzie się znaczco różniła w zależności od początkowej pozycji. Ten efekt

można by obejść za pomocą mapowania wartości za pomocą funkcji odpowiedniej krzywej, jednak wymagałoby to formułowania skomplikowanego równania, bądź wyznaczenia arbitralnego przebiegu funkcji. Istnieje jednak mniej wymagające obliczeniowo podejście z zastosowaniem rachunku macierzowego [49]. Dla uproszczenia obliczeń, zrezygnowano z pomiaru w trzech osiach, zamiast tego mierzone są jedynie natężenia pola w osiach X i Y (rysunek 3.8).



Rys. 3.8: Rozkalibrowany magnetometr na dwuwymiarowej płaszczyźnie

Tak jak w przypadku trzech wymiarów, w pierwszej kolejności dokonywana jest korekcja zniekształceń *hard iron* na wszystkich osiach (rysunek 3.9).



Rys. 3.9: Korekcja hard iron

Teraz można przystąpić do procedury kompensacji zniekształceń *soft iron*. Idealnie, po dokonaniu korekcji w dwóch wymiarach, punkty na wykresie można by umieścić na okręgu - tak

jednak nie jest. Uzyskany kształt bardziej przypomina elipsę, i ta właściwość zostanie wykorzystana. Przebieg procesu przedstawiono w formie pseudokodu (Algorytm 1).

Algorytm 1: Kompensacja zniekształceń *soft iron*

```

// Lista zmierzonych punktów w postaci par współrzędnych ( $P_x, P_y$ )
P
// Lista do której dodane zostaną odległości od (0, 0) do P
r

// Oblicz odległości punktów od środka układu współrzędnych
for  $i = 0, \dots, 180$  do
     $r_i = \sqrt{P_{ix}^2 + P_{iy}^2}$ 
end

// Wartości  $P_{min}$  i  $P_{max}$  odpowiadają kolejno końcom półosi małej i półosi wielkiej
// elipsy.
 $r_{min} = \min(r)$ 
 $r_{max} = \max(r)$ 
 $P_{min} = \min(P)$ 
 $P_{max} = \max(P)$ 

// Oblicz kąt obrotu elipsy
 $\theta = \arctan2(P_{max_y}, P_{max_x})$ 

// Wyznacz macierz R
 $R = \begin{bmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{bmatrix}$ 

// Oblicz parametr skalujący
 $\sigma = \frac{r_{min}}{r_{max}}$ 

// Lista do której zostaną dodane punkty P po przekształceniu
P2

// Obróć elipsę o kąt  $-\theta$  mnożąc punkty przez macierz
for  $i = 0, \dots, 180$  do
     $P2_i = \begin{bmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{bmatrix} \times \begin{bmatrix} P_{ix} \\ P_{iy} \end{bmatrix}$ 
end

// Lista do której zostaną dodane punkty P2 po przeskalowaniu
P3

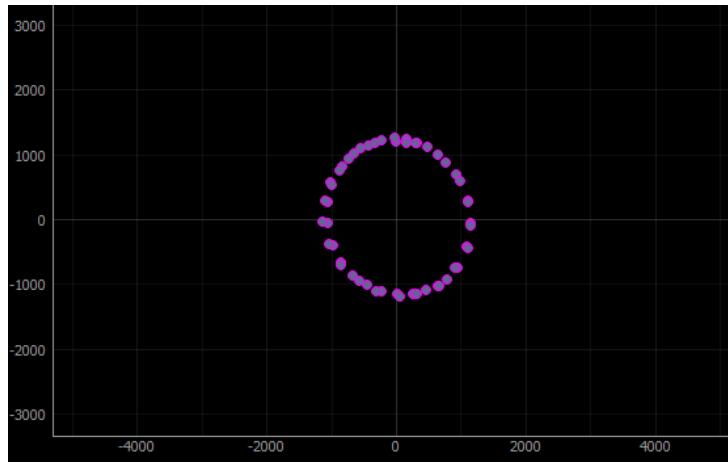
// Przeskaluj elipsę
for  $i = 0, \dots, 180$  do
     $P3_{ix} = P2_{ix} \times \sigma$ 
     $P3_{iy} = P2_{iy}$ 
end

// Lista P3 zawiera wartości wyjściowe po korekcji

```

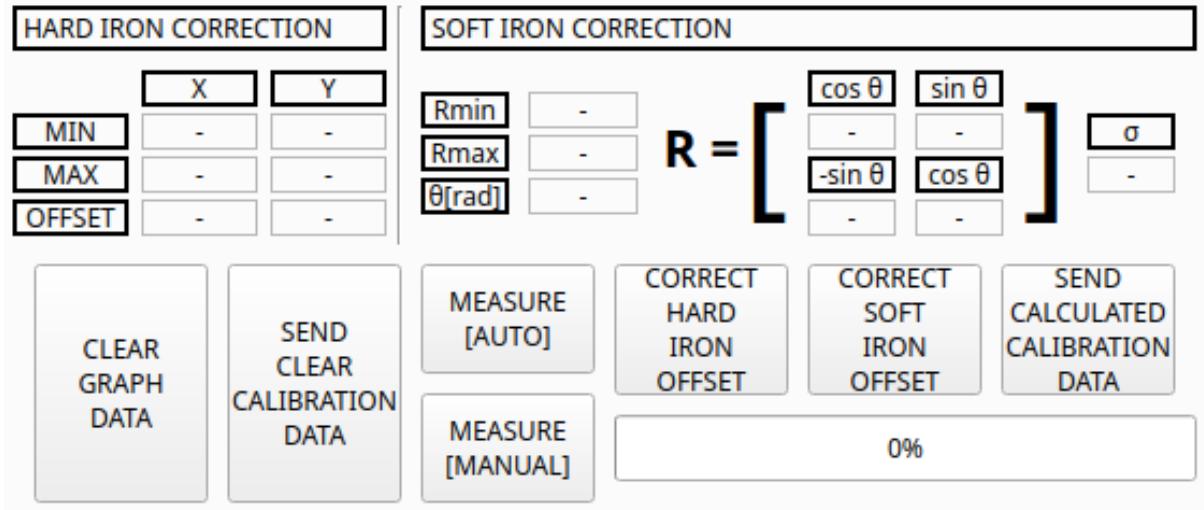
Ostatecznie chmura punktów tworzy okrąg co przedstawiono na rysunku 3.10. Efektem zastosowania kalibracji jest równomierny odczyt azymutu wraz z obrotem platformy. Każdy zmierzony i przefitrowany punkt może być teraz przeliczony na kąt wektora, który wskazuje azymut

- wystarczy skorzystać z dwuargumentowej funkcji $\arctan2(x, y)$, gdzie x i y są współrzędnymi punktu.



Rys. 3.10: Chmura punktów po dokonaniu korekcji obu typów zniekształceń

Finalna wersja aplikacji korzysta z dwóch osi magnetometru. W celu kalibracji należy skorzystać z zakładki *magnetometer calibration* głównego okna aplikacji sterującej. Rysunek 3.11 przedstawia najważniejsze elementy tej zakładki, wykorzystywane podczas półautomatycznego procesu kalibracji.



Rys. 3.11: Najważniejsze elementy sekcji kalibracji

Aby dokonać kalibracji, robot powinien być połączony z aplikacją sterującą. Dalej procedura przebiega jak następuje:

1. Należy przejść do zakładki *magnetometer calibration*
2. Jeżeli na wykresie znajdują się poprzednie pomiary, należy kliknąć przycisk *CLEAR GRAPH DATA* aby je usunąć
3. Należy wyzerować dane kalibracyjne znajdujące się w pamięci EEPROM robota. Służy do tego przycisk *SEND CLEAR CALIBRATION DATA*
4. Na tym etapie istnieją dwie możliwości przeprowadzenia pomiaru - za pomocą przycisku *MEASURE [AUTO]* platforma samodzielnie, krokowo, wykona obrót wokół własnej osi i zbierze serię pomiarów z magnetometru; za pomocą przycisku *MEASURE [MANUAL]*

dokona jedynie pomiarów, w tym czasie należy robota obracać ręcznie. Postęp pomiarów wizualizowany jest na pasku postępu w prawym dolnym rogu. Na wykresie pojawią się zebrane pomiary. W sekcji oznaczonej *HARD IRON CORRECTION* pojawiać się będą uzyskane dane dotyczące korekcji pierwszego z omawianych wcześniej zaburzeń - wartości minimalne i maksymalne dla każdej z osi oraz wartość przesunięcia (*OFFSET*). W sekcji *SOFT IRON CORRECTION* pojawiać się będą cyklicznie przeliczane wartości dotyczące korekcji zaburzeń *soft iron* - m.in. parametry $\theta \sigma$ i wartości macierzy R .

5. Za pomocą przycisku *CORRECT HARD IRON OFFSET* należy dokonać korekcji zaburzeń typu *hard iron* na podstawie obliczonych wartości przesunięcia. Efekt natychmiastowo ukaże się na wykresie.
6. Za pomocą przycisku *CORRECT SOFT IRON OFFSET* należy dokonać korekcji zaburzeń typu *soft iron*. Dane również brane są z obliczonych podczas procedury pomiaru. W tym momencie punkty na wykresie powinny być ułożone w okrąg. Jeżeli jest inaczej, oznacza to że w otoczeniu występują zaburzenia pola magnetycznego. Wtedy należy przemieścić robota w inne miejsce i powtórzyć wymienione czynności od nowa.
7. Na koniec, aby wysłać dane kalibracyjne do robota, należy kliknąć przycisk *SEND CALCULATED CALIBRATION DATA*.

Po dokonaniu procedury kalibracji pomiary dużo lepiej oddają rzeczywisty kierunek i zwrot platformy, jednak nawet podczas postoju platformy zwracana wartość azymutu nie pozostaje stała. Aby zniwelować to zjawisko konieczne jest zastosowanie filtru.

3.1.2. Filtr Kalmana

W układach sensorycznych robotów powszechnie wykorzytywany jest filtr Kalmana[50]. W tej sekcji wyjaśniona będzie zastosowana w projekcie, uproszczona implementacja takiego filtra.

```
def get_azimuth_kalman(self):
    q = 0.1
    estimate_err = 3
    measure_err = 3
    last_est = int(self.send("GET_AZIMUTH#"))

    for _ in range(20):
        measurement = int(self.send("GET_AZIMUTH#"))
        kalman_gain = estimate_err/(estimate_err + measure_err)
        current_est = last_est + kalman_gain*(measurement - last_est)
        estimate_err = (1 - kalman_gain)*estimate_err + abs(last_est-current_est)*q
        last_est = current_est
        ... (pominięto)

    current_est = int(current_est)
    self.azimuth = current_est
    self.publish_ros_odometry()
    return self.azimuth
```

Listing 4: Implementacja filtra Kalmana w języku Python

Ze względu na ograniczone zasoby mikrokontrolera, filtracja odbywa się po stronie aplikacji sterującej. Pomiar dokonywany jest podczas gdy platforma stoi nieruchomo, stąd wiadomo że estymowana wartość jest stała. Na początku potrzebne będzie kilka odgórnie ustalonych parametrów. Wartości *estimate_err* i *measure_err* odpowiadają kolejno wariancji estymatora i wariancji mierzonej wartości (normalnie wyznaczone na podstawie dokładności pomiaru przyrządu), jednak nie muszą one być ustalone na podstawie rzeczywistych parametrów urządzenia.

Ich wartości będą wpływać na działanie filtra - to czy dane będą bardziej "wygładzane", czy bardziej zależne od ostatniego pomiaru. W tym wypadku zostały wyznaczone eksperymentalnie. Parametr q nie był konieczny, służy on regulacji zwiększenia wariancji estymaty w przypadku gdy estymowana wartość jest zmienna w czasie. Został on również dobrany eksperymentalnie. Chociaż ma mały wpływ na pomiary statyczne - jest zaimplementowany na potrzeby przyszłego rozwoju projektu.

Kolejną rzeczą która jest potrzebna jest aktualna wartość ostatniej estymacji *last_estimate*. Przy odpowiednio długim pomiarze tę wartość można ustawić na dowolną, jednak aby wartość estymatora szybciej zbiegała do estymowanej wstępnie zostaje ustawiona na zmierzoną wartość.

Mając wszystkie parametry można przystąpić do uruchomienia filtra. Przebiega on w 20 iteracjach - wartość ta została dobrana arbitralnie w celu ewentualnej późniejszej korekcji. Nie mogła być zbyt duża, aby mikrokontroler był w stanie ukończyć operację pomiaru w rozsądny czasie. W każdej z iteracji zachodzą dwie fazy - predykcji i korekcji.

Faza predykcji polega na wyznaczeniu wartości oczekiwanej apriori $\hat{x}(t+1)^-$ i odchylenia standardowego apriori $\sigma^2(t+1)^-$ dla czasu $t+1$ na podstawie analogicznych wartości aposteriori dla czasu t , tj. $\hat{x}(t)^+$ i $\sigma^2(t)^+$. W tym wypadku z założenia wartość mierzona jest stała, dlatego przyjmuje się, że:

$$\begin{aligned}\hat{x}(t+1)^- &= \hat{x}(t)^+ \\ \sigma^2(t+1)^- &= \sigma^2(t)^+\end{aligned}\tag{3.3}$$

Kod programu korzysta ze zmiennych *est_error* i *current_est*. Wartości apriori i aposteriori nie muszą być przechowywane jednocześnie w pamięci - wystarczy nadpisać stare zmienne. Tutaj oznaczałoby to sformułowanie konstrukcji *est_error=est_error* i *current_est=current_est* co nie jest potrzebne i dlatego zostało pominięte.

Faza korekcji odbywa się w pętli. Najpierw dokonywany jest pomiar z sensora (*measurement*). Następnie obliczane jest wzmacnienie Kalmana (*kalman_gain*), czyli parametr wpływający na to jakie znaczenie nowy pomiar ma dla wartości estymatora oraz jego wariancji. Później, z jego wykorzystaniem obliczana jest nowa wartość estymatora (*current_est*). W kolejnej linijce aktualizowana jest wartość jego wariancji (*estimate_err*). Aktualnie wraz z każdym nowym przejściem pętli zbiega ona do niezerowej wartości, zależnej od parametru q . Gdyby ten parametr nie zaistniał, z każdą iteracją jej wartość zbiegałaby do zera. Na koniec wartość (*current_est*) kopowana jest do (*last_est*). Dzieje się to, ponieważ aktualizacja (*estimate_err*) potrzebuje przy obliczeniach wartości aktualnej oraz poprzedniej estymatora.

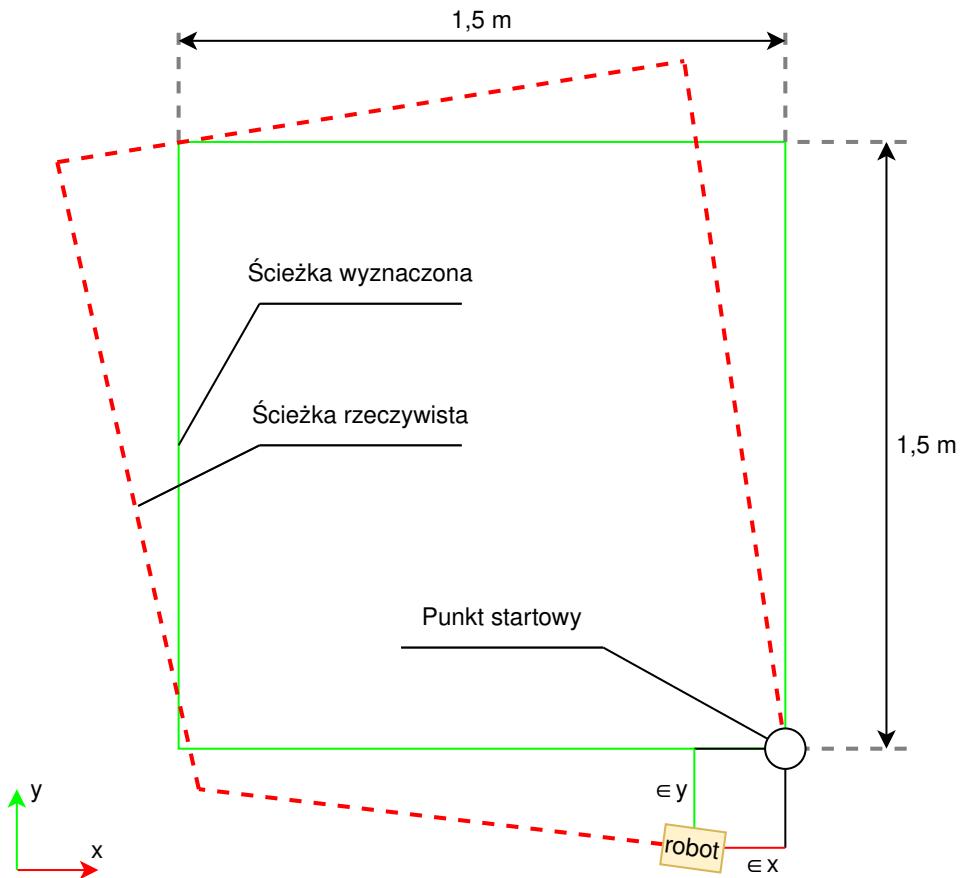
Po dwudziestokrotnym przejściu, wartość oczekiwana estymatora konwertowana jest na liczbę całkowitą, i zapisywana, dane o azymucie są publikowane na odpowiednim temacie i wartość jest zwracana przez funkcję.

3.1.3. Poprawa jakości odometrii

Ze względu na brak możliwości skonstruowania idealnego obiektu rzeczywisty ruch platformy nie pokrywa się z zamierzonym. Oś obrotu kół nigdy nie osiągnie wymiarów idealnie równych z projektem, podobnie same koła będą miały inne i różne od siebie średnice (oczywiście, z pewnym przybliżeniem mówi się że są one "równe"). Dodatkowo na nieidealny tor ruchu wpływa chropowatość powierzchni, drgania styków enkoderów i wiele innych czynników. Istnieje metoda która pozwala skorygować część z wymienionych czynników.

Wykorzystana zostanie metoda korekcji opisana w pracy *Correction of Systematic Odometry Errors in Mobile Robots*[46]. Korzysta ona z metody wyznaczania błędu przejazdu trasy zwanej

UMBmark (od ang. *University of Michigan Benchmark*). Polega ona na zbadaniu przesunięcia pozycji końcowej robota względem pozycji początkowej po pokonaniu wyznaczonej, kwadratowej ścieżki. Na rysunku 3.12 przedstawiono przykładowy przejazd w kierunku przeciwnym do ruchu wskazówek zegara.

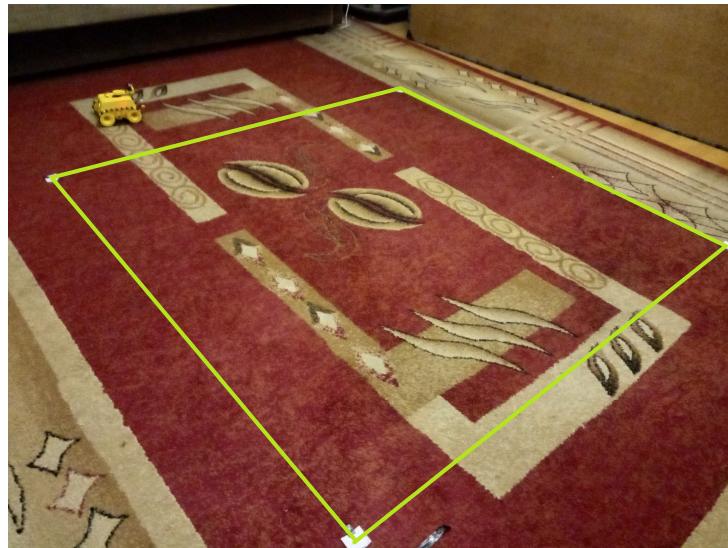


Rys. 3.12: Trasa przejazdu robota według *UMBmark*

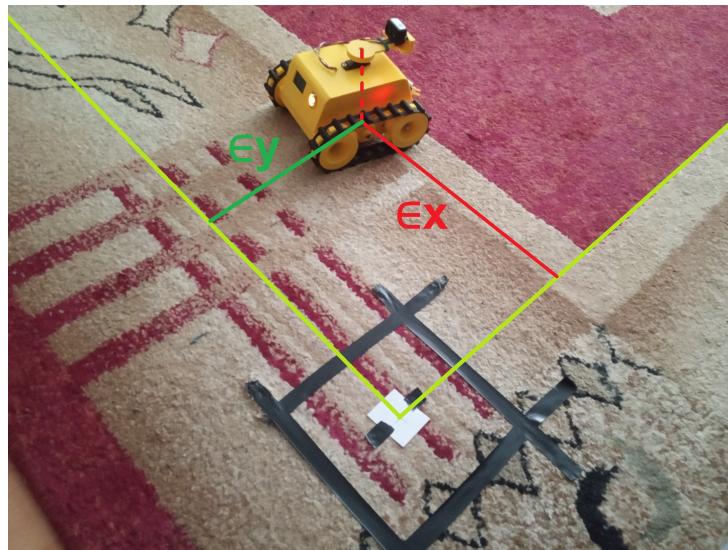
Metoda *UMBmark* polega na pięciokrotnym pomiarze błędów $\in x$ oraz $\in y$ po przejazdach zgodnie i przeciwnie do ruchu wskazówek zegara. Różne kierunki przejazdu pomogą zniwelować wpływ błędów niesystematycznych na tor jazdy platformy. Rozmiar kwadratowej ścieżki według dokumentu powinien wynosić 4×4 metry ($L = 4m$), jednak ze względu na ograniczoną przestrzeń autor zmniejszył obszar do rozmiaru $1,5 \times 1,5$ metra. Zdjęcia środowiska testowego przedstawiono na rys. 3.13 i 3.14. Do pomiarów użyto taśmy mierniczej zwijanej o dokładności ± 1 mm jednak ze względów praktycznych w pomiarach uwzględniono wartości zaokrąglone do pełnych centymetrów. Wyniki przedstawiono w tabeli 3.1.

Tab. 3.1: Wyniki pomiarów *UMBmark*

N. p.	Kierunek: \circlearrowleft		Kierunek: \circlearrowright	
	$\in x$	$\in y$	$\in x$	$\in y$
1	-28	5	62	28
2	-33	15	54	32
3	-2	4	-45	9
4	-5	-21	-55	-19
5	13	-12	-31	27



Rys. 3.13: Wyznaczona ścieżka przejazdu robota



Rys. 3.14: Punkt startowy i przesunięcie

Mając do dyspozycji wartości potrzebnych pomiarów można przystąpić do obliczeń. Najpierw dla każdego kierunku należy wyznaczyć środek ciężkości, czyli uśrednić wartości współrzędnych x i y , gdzie przykładowo dolny indeks x_{cgw} oznacza wartość współrzędnej x środka

ciężkości dla przejazdów zgodnych z ruchem wskazówek zegara (ang. *cg - center of gravity, cw - clockwise*). Analogicznie dla kierunku przeciwnego to będzie x_{cscw} . I tak:

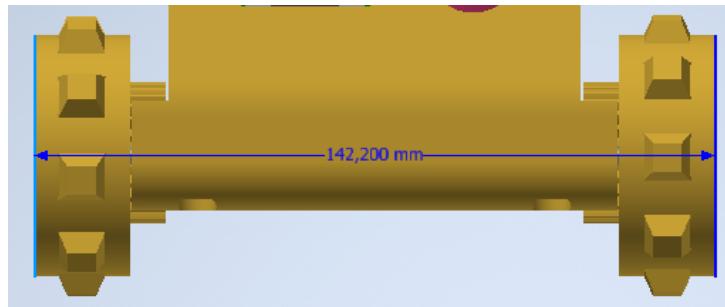
$$\begin{aligned} x_{cgccw} &= \frac{1}{n} \sum_{i=1}^n (\in x_{i,ccw}) = \frac{-28 - 33 - 2 - 5 + 13}{5} = -11 \\ y_{cgccw} &= \frac{1}{n} \sum_{i=1}^n (\in y_{i,ccw}) = \frac{5 + 15 + 4 - 21 - 12}{5} = -1,8 \\ x_{cgcw} &= \frac{1}{n} \sum_{i=1}^n (\in x_{i,cw}) = \frac{62 + 54 - 45 - 55 - 31}{5} = -3 \\ y_{cgcw} &= \frac{1}{n} \sum_{i=1}^n (\in y_{i,cw}) = \frac{28 + 32 + 9 - 19 + 27}{5} = 15,4 \end{aligned} \quad (3.4)$$

W kolejnym kroku zostaną obliczone współczynniki α oraz β :

$$\begin{aligned} \alpha &= \frac{x_{cgcw} + x_{cgccw}}{-4L} = \frac{-3 - 11}{-4 \times 1,5} = 2,33^\circ \\ \beta &= \frac{x_{cgcw} - x_{cgccw}}{-4L} = \frac{-3 + 11}{-4 \times 1,5} = -1,33^\circ \end{aligned} \quad (3.5)$$

Z pomocą współczynników α i β obliczane są błędy wylistowane poniżej. Tu należy zaznaczyć, że korekta dotyczy parametrów długości osi b i średnicy kół D schematu zastępczego robota przedstawionego wcześniej w niniejszym dokumencie na rysunku 3.2.

- Błąd E_d będący stosunkiem średnicy koła prawego do koła lewego
- Błąd E_b będący stosunkiem rzeczywistej długości osi do długości nominalnej



Rys. 3.15: Długość nominalna osi robota

Obliczanie współczynników R , E_d i E_b ukazano poniżej. Przydatny będzie tutaj parametr b oznaczający rozpiętość osi zastępczej pojazdu. Jest ona równa osiom rzeczywistym i wynosi 14,22 centymetrów, co uwidoczniono na rysunku 3.15.

$$\begin{aligned} R &= \frac{L : 2}{\sin(\beta : 2)} = \frac{1,5 : 2}{\sin(-1,33 : 2)} = -0,01 \\ E_d &= \frac{R + (b : 2)}{R - (b : 2)} = \frac{-0,01 + (14,22 : 2)}{-0,01 - (14,22 : 2)} = -0,997 \\ E_b &= \frac{90^\circ}{90^\circ - \alpha} = \frac{90^\circ}{90^\circ - 2,33^\circ} = 1,03 \end{aligned} \quad (3.6)$$

Parametr R posłużył jedynie obliczeniu pozostałych dwóch. Współczynnik E_d odnosi się do korekcji błędów typu b opisanych w pracy Borenstein'a (zakrzywiony tor jazdy) i jest równy stosunkowi średnic przeciwnie skierowanych kół (modelu zastępczego) pojazdu. Specjalnie, w odróżnieniu do innych wartości został zaokrąglony do trzeciego miejsca po przecinku - w innym wypadku wynosiłby 1. Z tego samego powodu można uznać, że korekcja uwzględniająca ten parametr nie jest konieczna - robot jeździ wystarczająco "prosto" ewentualne zakrzywienie toru jazdy wynika z czynników niesystematycznych (zmiany powierzchni, nierówności).

Jeżeli chodzi o E_b odnosi się on do korekcji błędów typu a - zbyt dużych lub za małych zakrętów wykonywanych w miejscu przez robota. Taki błąd wynika z różnicy między nominalnym a rzeczywistym rozstawem kół pojazdu. Stosunek tych wartości opisuje wartość tegoż współczynnika wynoszącą 1,03. Oznacza to, że robot podczas obrotu zachowuje się tak, jakby jego os była dłuższa o 3% niż przewidziano w projekcie. Mając na uwadze czynnik jakim jest poślizg gąsienic oraz zgrubny charakter aproksymacji jego modelu, autor na podstawie pewnych intuicji i poprzednich doświadczeń był w stanie stwierdzić, że taki błąd jest pomijalny. Dodatkowo, ze względu na eksperymentalny charakter implementacji zakrętu w niniejszym projekcie zastosowanie takiej korekcji byłoby bardziej skomplikowane, ponieważ długość osi nie jest nigdzie zdefiniowana.

3.2. Skan otoczenia i budowa mapy

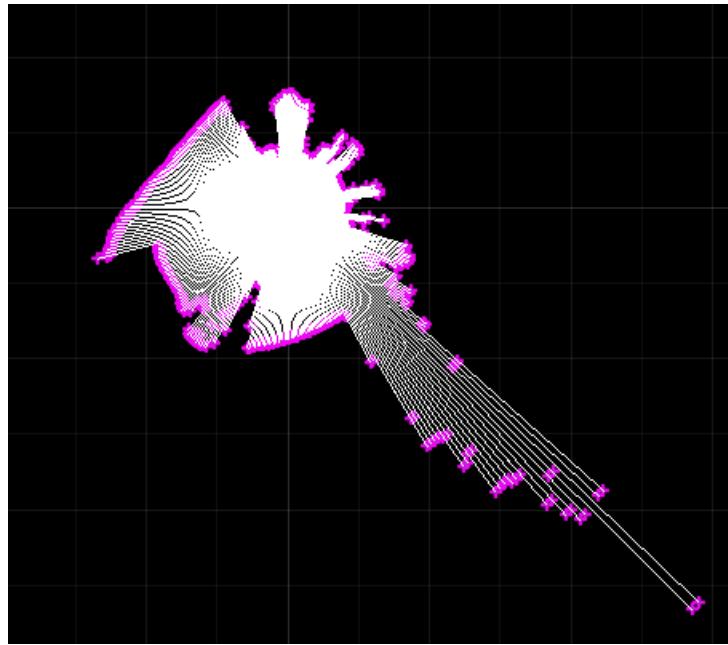
Samo zebranie pomiarów i przedstawienie ich w formie czytelnej dla człowieka nie jest trudne w realizacji. Pozyskane dane można przedstawić na wykresie, punkty połączyć prostymi liniami. Problem zaczyna się z agregacją wielu pomiarów, i na tym będzie koncentrował się ten rozdział.

Mając do dyspozycji dane z sensora *LIDAR* oraz enkoderów i magnetometru, można rozpoczęć proces skanowania. Pierwsze podejście oparte było o skan manualny, bez zliczania ścieżki - jedynie obrót był uwzględniony a dane ze skanera obracane o zczytany z sensora azymut. Robot został ustawiony w początkowej pozycji, uruchomiono procedurę skanowania za pomocą funkcji *SCAN*, następnie z racji, że skan zachodzi w zakresie kątów $\langle 0^\circ, 180^\circ \rangle$, obrócony o 180° (funkcja *ROTATE*) po czym ponownie wykonano skanowanie. Efekt przedstawia rysunek 3.16. Jak widać, ściany pokoju (górną i dolną część rysunku) nie są równoległe. W tym momencie okazało się, że konieczna będzie kalibracja magnetometru, omówiona w rozdziale 3.1. Po dokonaniu procedury kalibracji czynności powtórzono, rezultat ukazano na rysunku 3.17. Na dalej przedstawianych rysunkach skany mogą wyglądać nieco inaczej, głównie przez zamknięte lub otwarte drzwi pokoju - widoczne jest to na ostatnich dwóch wspomnianych rysunkach jako najdłuższe z promieni - w tym wypadku drzwi były otwarte a pomiar uwzględniał fragment przedpokoju autora. Skany przedstawiane są w zakładce *map* okna głównego aplikacji sterującej.

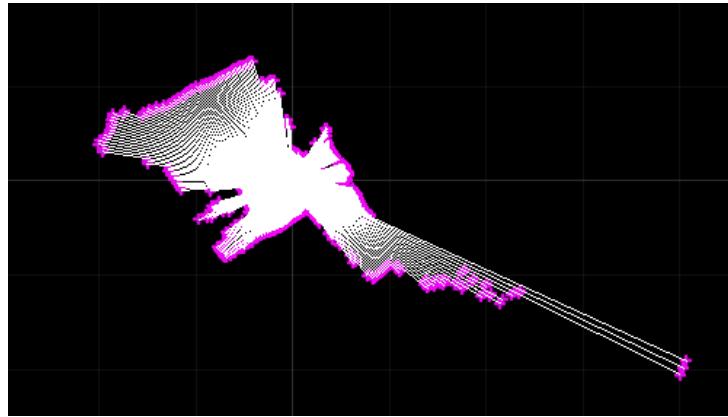
W opisanych krokach dwa przeprowadzane pomiary były od siebie niezależne - mierzyły odległość od innych obiektów. Kolejnym krokiem było wykonanie kilku pomiarów częściowo nakładających się na siebie, co przedstawiają rysunki 3.18, 3.19, 3.20 i 3.21. Szczególnie należy zwrócić uwagę na ostatni z nich - wyraźnie widać, że ostatni skan absolutnie nie pokrywa się z poprzednimi. Najwyraźniej sama informacja o aktualnej rotacji robota nie jest wystarczająca aby dokładnie umieścić punkty na płaszczyźnie.

W celu korekcji rzutowania punktów na płaszczyznę, pierwotnie stosowany był opracowany przez autora prosty, naiwny algorytm korygujący rotację. Jego działanie można opisać w kilku krokach. (Algorytm 2).

Listing 5 przedstawia fragment kodu przeprowadzający obliczanie współczynnika *score*. Tużże należy zaznaczyć, że ze względu na liczne zmiany w projekcie ten fragment nie zachował się w finalnej wersji kodu, można go natomiast podejrzeć na reposytorium w pliku *src-pc/main.py* na gałęzi *master* pod commit id wynoszącym *aba2c1510f1a661e6365f9cd4ca7dc782f013593*.



Rys. 3.16: Pierwszy skan pokoju

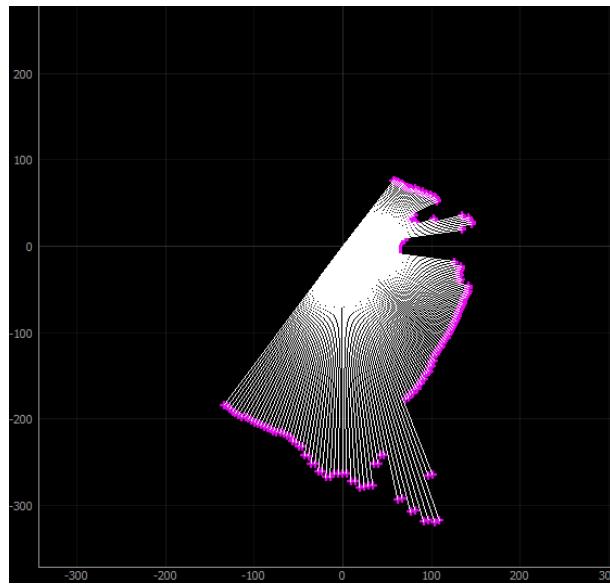


Rys. 3.17: Skan pokoju po zastosowaniu kalibracji magnetometru

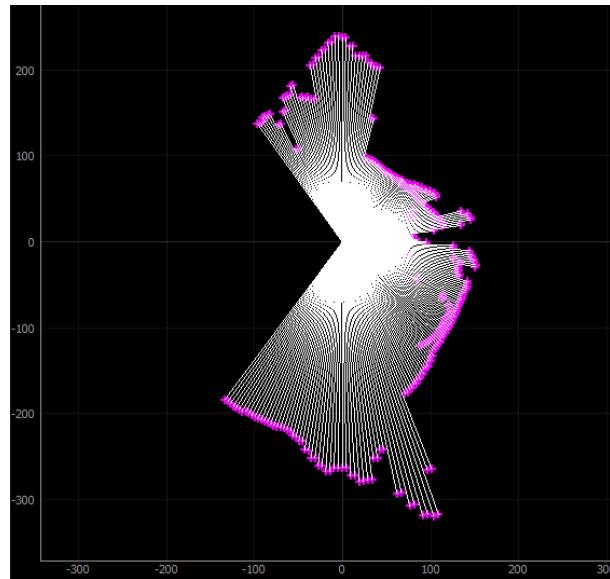
```
def calc_xy_overlap_score(self, points1, points2):
    score = 0
    for point in points1:
        distance = self.find_xy_closest_point_distance(
            [point[0], point[1]], points2)
        # important! this way we're avoiding very high 1/distance value
        distance = int(distance)
        if distance == 0:
            score += 100
        elif distance >= 100:
            pass
        else:
            score += int((1/distance)*100)
    return score
```

Listing 5: Obliczanie współczynnika *score* i korekcja kąta w celu dopasowania pomiaru do mapy

Współczynnik *score* obliczany jest na podstawie podobieństwa między pozycją punktów z najnowszego skanu względem tego, co już znajduje się na mapie. Aby móc skorzystać z powyż-

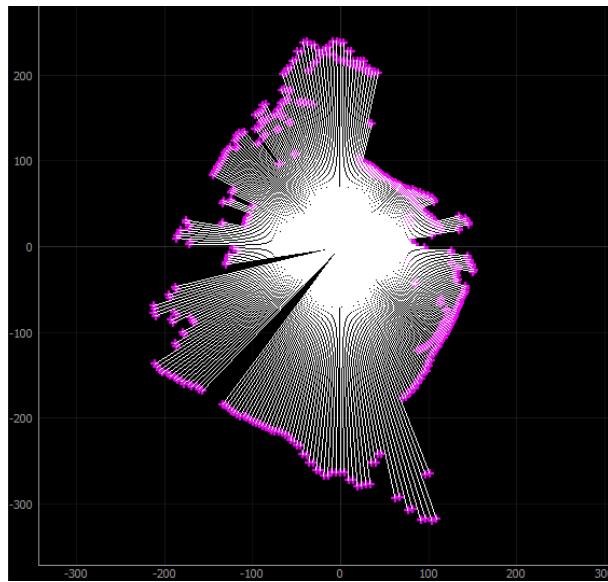


Rys. 3.18: Skan pierwszy

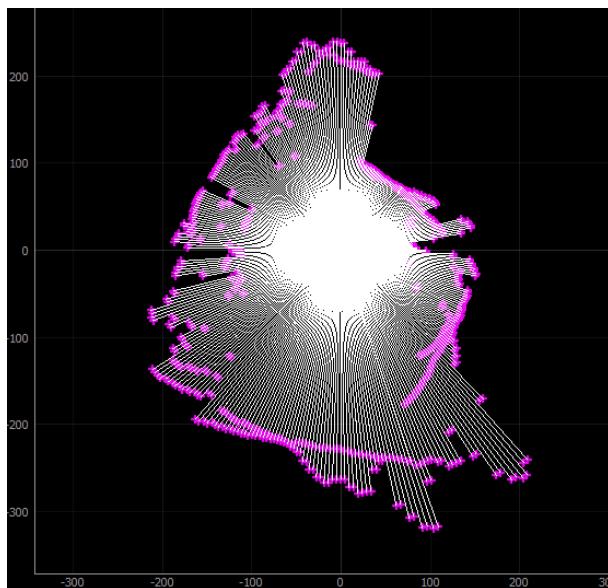


Rys. 3.19: Skan drugi

szej funkcji poszczególne pomiary dla skanu są przeliczane z postaci biegunowej na kartezjańską. Następnie jest ona uruchamiana, a do parametrów *points1* i *points2* kolejno przekazywane są listy punktów nowego skanu i punktów znajdujących się na mapie. W dalszej kolejności dla każdego z punktów skanu wykonywana jest operacja znalezienia najbliższego punktu na mapie i obliczenia odległości między nimi. Jeżeli punkty idealnie się pokrywają, do *score* dodawana jest wartość 100. Jeżeli ta odległość wynosi więcej lub równo 100 centymetrów, nie dodawana jest żadna wartość. W zakresie (1, 100) do współczynnika jest dodawana wartość równa $\frac{1}{odleglosc} * 100$, zaokrąglona do najbliższej liczby całkowitej.



Rys. 3.20: Skan trzeci



Rys. 3.21: Skan czwarty

```

if score > 2500:
    print(
        f"adjusted plot and robot rotation by {best_rotation} degrees.")
    rotated_data = self.rotate_points(filtered_data, best_rotation)
    self.robot.update_azimuth(self.robot.azimuth + best_rotation)
else:
    rotated_data = filtered_data

```

Listing 6: Decyzja o zastosowaniu korekty

Doświadczalnie został dobrany próg wartości *score* równy 2500. Jeśli współczynnik dla najlepszej korekty kąta wynosi więcej niż próg, korekta zostanie zaakceptowana - azymut robota zostanie poprawiony, skan również będzie skorygowany i rzutowany na mapę. Obrazuje to listing 6.

Algorytm 2: Korekcja rotacji zmierzonych punktów

```

// Pierwszy skan nie jest korygowany, ponieważ nie ma innych punktów na mapie
kąty, odległości = wykonaj_skan()
azymut = zmierz_azymut()
for i = 0, ..., 180 do
|   kąty[i] = kąty[i] + azymut
end

nanieś_na_wykres(kąty, odległości)

while program_jest_uruchomiony do
    kąty, odległości = wykonaj_skan()
    azymut = zmierz_azymut()
    for i = 0, ..., 180 do
        |   kąty[i] += azymut
    end

    // Słownik przechowujący wartości score
    // dla poszczególnych par klucz:wartość ( kąt_korekcji:score)
    score = {}

    for i = -10, ..., 10 do
        |   kąty_temp[i] = kąty[i] + i
        |   score[i] = oblicz_score(kąty_temp, odległości, punkty_na_mapie)
    end

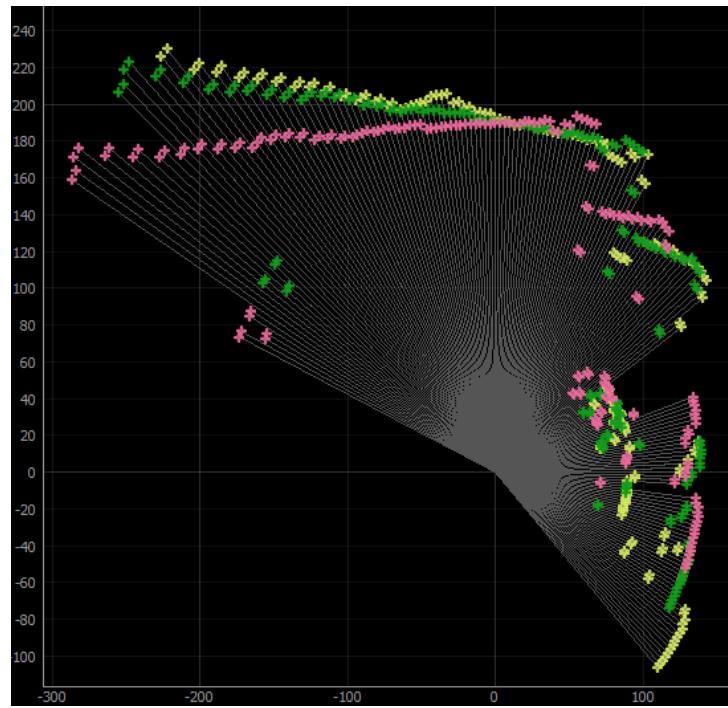
    if max(score.wartości()) > 2500 then
        |   kąt_korekcji = max(score.wartości()).klucz()
        |   for i = 0, ..., 180 do
        |       |   kąty[i] += kąt_korekcji
        |   end
        |   nanieś_na_wykres(kąty, odległości)
        |   azymut_robota += kąt_korekcji
    else
        |   // Nie stosuj korekcji
        |   nanieś_na_wykres(kąty, odległości)
    end
end

```

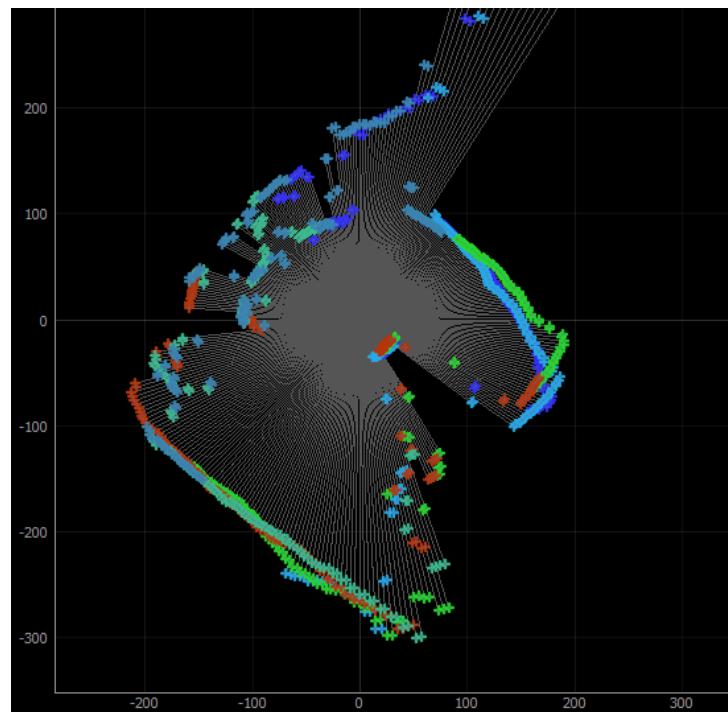
Na rysunku 3.22 przedstawiono działanie algorytmu korekcji kąta. Drugi ze skanów (punkty oznaczone kolorem żółtym) jest korygowany względem danych znajdujących się na mapie pochodzących ze skanu pierwszego (kolor różowy). Widoczne jest spore przesunięcie. Pomiar różowy dopasowywany był w zakresie $\pm 10^\circ$, osiągając maksymalną wartość $score$ przy -10° . Skorygowany pomiar został naniesiony na mapę z kolorem zielonym.

Ewaluacja tego algorytmu dla całego pokoju została ukazana na rysunkach 3.23 i 3.24.

Pomimo iż algorytm działa i spełnia założone zadanie - problem *SLAM* jest dużo bardziej złożony. Pierwotnie w planach autor planował wdrożenie podobnej korekcji dla translacji pojazdu oraz uśrednianie pozycji punktów na mapie. Jednak ze względu na dużą złożoność problemu i ograniczony czas autor zadecydował o skorzystaniu z istniejących rozwiązań. Pomocnym był tutaj zestaw narzędzi *ROS*. Zawiera on wiele gotowych komponentów wykorzystywanych w robotyce, m. in. moduły adresujące problem *SLAM*. Jednym z takich modułów jest *slam_gmapping*



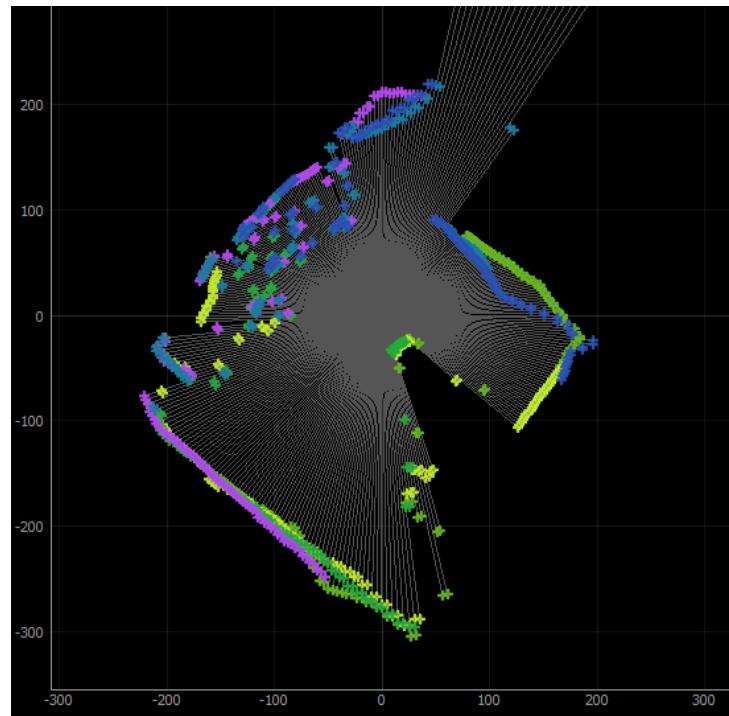
Rys. 3.22: Zobrazowana zasada działania korekcji kąta przy użyciu współczynnika *score*



Rys. 3.23: Pięć skanów otoczenia nałożonych z pomocą korekcji korzystającej ze współczynnika *score*

zawierający implementację algorytmu *gmapping*[48][35][37]. Od tego momentu mapa sporządzana jest w środowisku *ROS* i przedstawiana w programie *RViz*. Finalna wersja aplikacji sterującej współpracuje ze środowiskiem publikując odpowiednie dane na wyznaczonych w tym celu tematach. Dokładniej obrazuje to schemat opisany w podrozdziale 2.1.

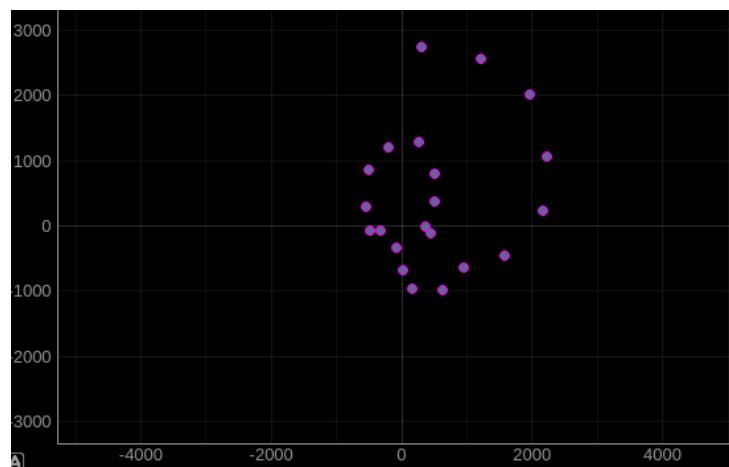
Kiedy całe środowisko zostało zestawione, przystąpiono do ręcznych testów jakości pracy platformy. Węzeł *gmapping* uruchomiony został z parametrami domyślnymi. Już przy pierwszej ewaluacji (rysunek 3.25) zauważono pewne nieprawidłowości. W tym momencie okazało



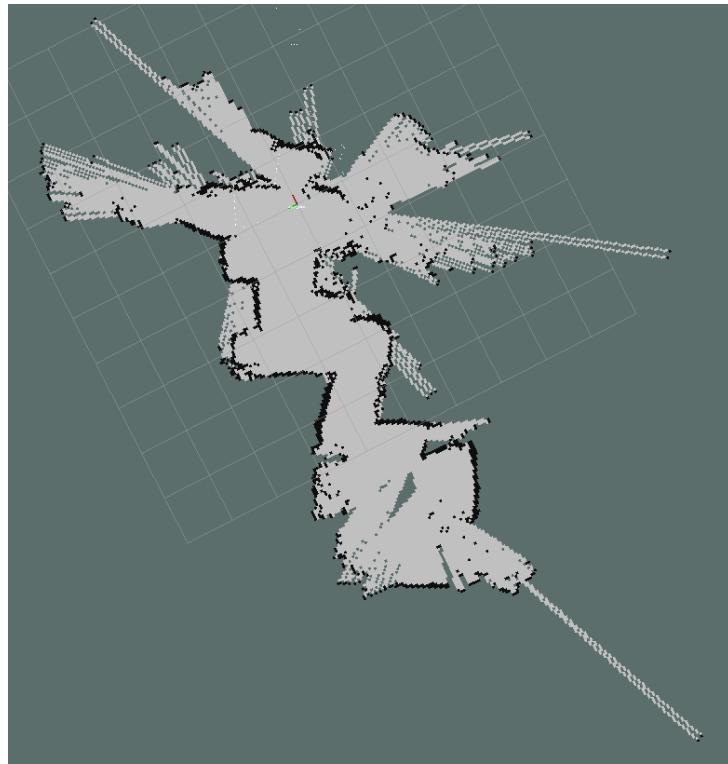
Rys. 3.24: Kolejne pięć skanów z korekcją kąta

się, że korzystanie z magnetometru przy odometrii nie było dobrym pomysłem. Pod podłogą prawdopodobnie w tym miejscu biegnie przewód elektryczny generując m. in. zmienne pole magnetyczne, zakłócając pracę magnetometru do takiego stopnia w którym odczytane dane są bezużyteczne i niemożliwe jest odtworzenie rzeczywistego pomiaru. Widoczne jest to w postaci bezładnie rozrzuconych punktów na górnej części rysunku, promieniujących od punktu w którym znajduje się robot.

Aby zweryfikować przypuszczenia autor wykonał serię pomiarów za pomocą przycisku *MESURE [AUTO]* z zakładki *magnetometer calibration*. To co zostało uwidocznione na rysunku 3.26 z pewnością nie przypomina okręgu co potwierdza że w tym obszarze występują zakłócenia. Z tego powodu zmieniono sposób zliczania obrotu, co zostało opisane w rozdziale 3.1.

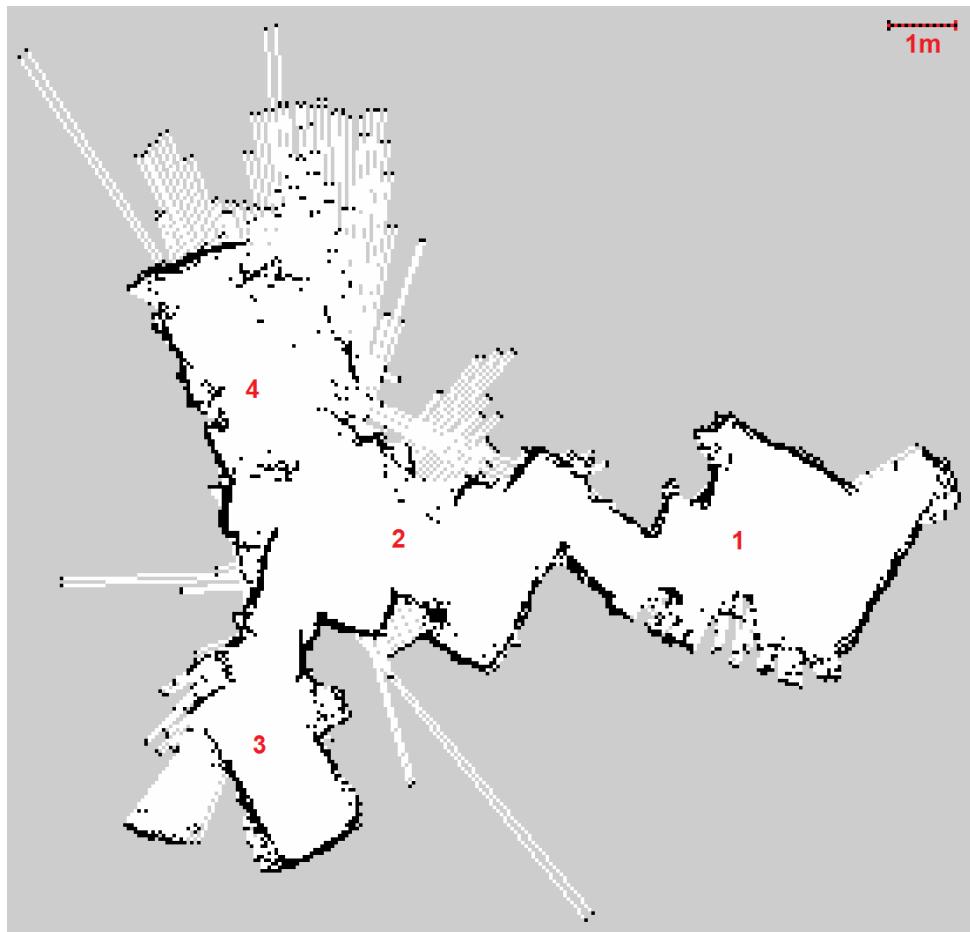


Rys. 3.26: Seria pomiarów ze skalibrowanego magnetometru w obszarze silnych zakłóceń



Rys. 3.25: Mapa domu, uszkodzona na skutek interferencji magnetycznych

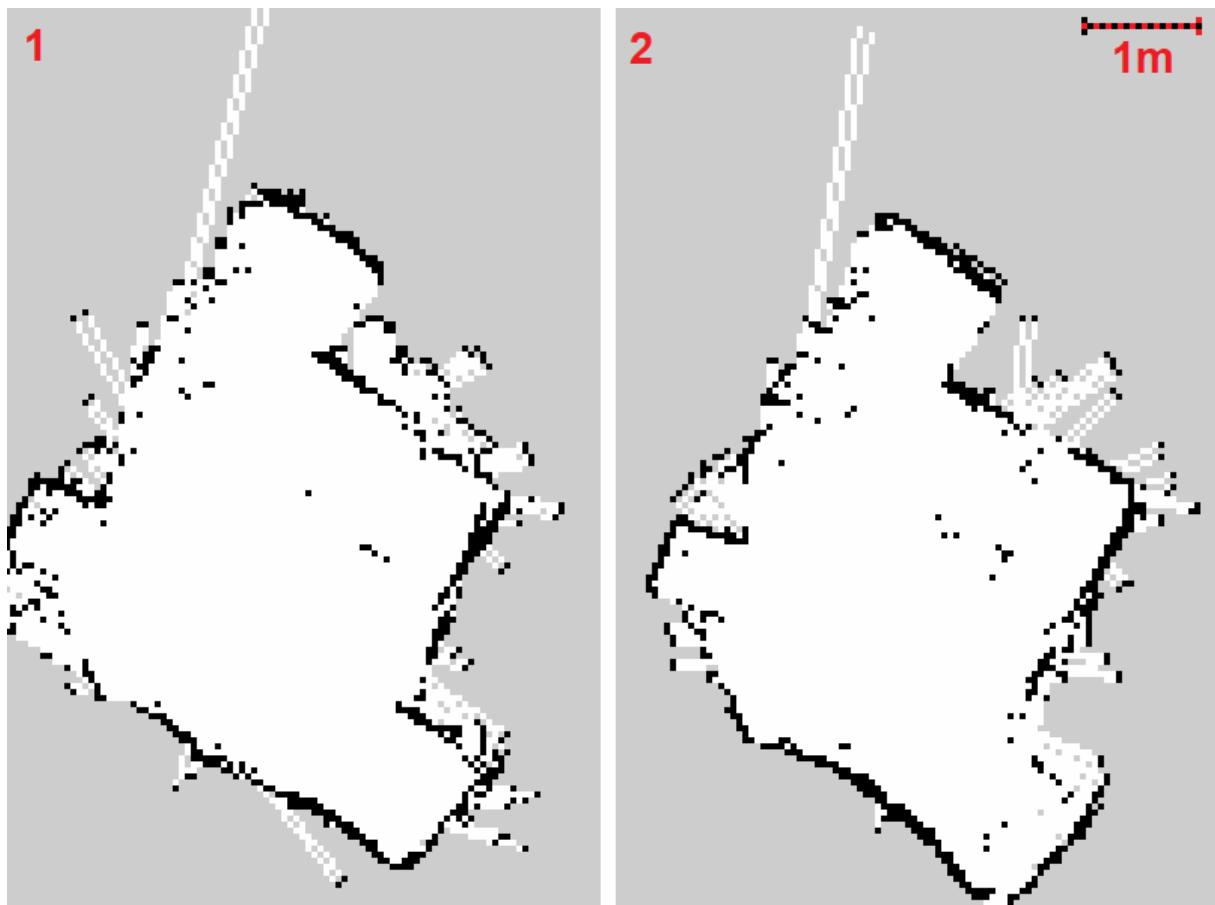
Niestety, wykorzystując mniej dokładną metodę zliczania algorytm *gmapping* również traci na jakości działania. Pomiar jest odporny na zakłócenia magnetyczne ale występuje problem przy wjeździe robota do nowych pomieszczeń. Nie mając wielu punktów odniesienia, z dużym błędem rotacji nowe pokoje są wyraźnie obrócone względem stanu rzeczywistego co przedstawia rysunek 3.27 na którym zaznaczono również kolejność skanowania pomieszczeń (1 - pokój autora, 2 - przedpokój, 3 - kuchnia, 4 - drugi pokój). Wpływ to bardzo negatywnie na późniejsze pomiary - mimo wszelkich starań nie udało się ukończyć skanu wszystkich pokojów.



Rys. 3.27: Mapa domu, obrót zliczany za pomocą enkoderów

Ze względu na długi czas ewaluacji skanu całego domu oraz szybko kumulujący się błąd, w dalszej części obszar został zawężony jedynie do pokoju autora. Zaimplementowany również został algorytm samodzielnej jazdy robota, opisany w podrozdziale 2.1.2.

Wyniki dwóch przejść robota po pomieszczeniu autora z wykorzystaniem autorskiego algorytmu jazdy autonomicznej przedstawiono na rysunkach 3.28.



Rys. 3.28: Mapa pomieszczenia powstała podczas jazdy autonomicznej robota

Rozdział 4

Podsumowanie

Podstawowy cel, jakim było stworzenie mapy pokoju został zrealizowany. Obrazy oddają rzeczywisty rozkład przeszkód w pokoju w stopniu pozwalającym na wizualną ocenę i zrozumienie struktury otoczenia. Ponadto robot jest w stanie samodzielnie poruszać się nie kolidując ze stacjonarnymi przeszkodami, a zastosowany algorytm jest niezwykle prosty w implementacji i pozwala na aplikację nawet w prostych systemach o znacznie mniejszej mocy obliczeniowej. Jakość mapy znajduje się na satysfakcjonującym poziomie, szczególnie biorąc pod uwagę półki cenowe zastosowanych peryferiów. Bez wątpienia skorzystanie z sensorów wyższej jakości wpłynęłoby pozytywnie na pracę całego systemu. Nie mniej jednak nawet z aktualną budową możliwe jest dopracowanie algorytmu mapowania poprzez dalszą, odpowiednią korekcję jego parametrów - tyczy się to również procedur związanych z jazdą autonomiczną platformy.

Komunikacja bezprzewodowa działała bezproblemowo, sprawdzony moduł HC-05 zapewnił bezbłędną transmisję danych. Jedyną słabą stroną było opóźnienie i wyraźne spowolnienie komunikacji w przypadku większych odległości między modułem podłączonym do komputera a tym wbudowanym w platformę.

Budowa mechaniczna robota jest obszarem szczególnego zadowolenia autora. Po wielu zmagańach podczas procesu opracowywania ostatecznego projektu udało się stworzyć podwozie zdolne do pokonywania wielu przeszkód w innym wypadku (przykładowo platformy bez gąsienic) niemożliwych bądź trudnych do pokonania. Taka konstrukcja bardzo pozytywnie sprawdziła się pod kątem stabilności toru jazdy, co wykazał przeprowadzony eksperyment opisany w rozdziale 3.1.

Platforma *ROS* pozwoliła na szybki rozwój projektu. Wiele dostępnych publicznie modułów umożliwia dalszy rozwój i poszerzenie funkcjonalności całego systemu. Jej interfejs jest bardzo elastyczny i możliwe jest łączenie wielu projektów napisanych w różnych językach programowania co upraszcza dodawanie własnych rozwiązań bądź modyfikacje już istniejących, lecz bezpośrednio niekompatybilnych.

Aplikacja sterująca upraszcza komunikację z platformą i w czytelny sposób prezentuje jej możliwości i wymieniane dane. Wraz z oprogramowaniem samego robota oraz środowiskiem *ROS* stanowi przyjazny, modularny ekosystem pozwalający na dalsze modyfikacje i przeprowadzanie kolejnych eksperymentów w obrębie problematyki *SLAM* i wielu innych zagadnień z dziedziny robotyki.

Bibliografia

- [1] Brzęczyk z generatorem. <https://botland.com.pl/pl/buzzery-generatory-dzwieku/786-buzzer-z-generatorem-5v-12mm-tht.html>. [dostęp 02.12.2020].
- [2] Dokumentacja techniczna mikrokontrolera stm32f103c8. <https://www.st.com/resource/en/datasheet/stm32f103c8.pdf>. [dostęp 24.11.2020].
- [3] Dokumentacja techniczna sensora benewake lidar ce30-a. <http://en.benewake.com/res/wuliu/docs/15772060131205712DE-LiDAR%20CE30-A%20Datasheet%20-%20V014-EN.pdf>. [dostęp 01.12.2020].
- [4] Dokumentacja techniczna sensora benewake lidar tf luna. <https://www.robotshop.com/media/files/content/b/ben/pdf/tf-luna-8m-lidar-distance-sensor-datasheet.pdf>. [dostęp 01.12.2020].
- [5] Dokumentacja techniczna sensora benewake lidar tf02. https://cdn.sparkfun.com/assets/1/6/1/d/8/tf02_datasheet-en.pdf. [dostęp 01.12.2020].
- [6] Dokumentacja techniczna sensora benewake lidar tf02 pro. https://www.robotshop.com/media/files/content/b/ben/pdf/benewake-tf02-pro-lidar-led-rangefinder-ip65-40m-manual_.pdf. [dostęp 01.12.2020].
- [7] Dokumentacja techniczna sensora benewake lidar tfmini plus. https://cdn.sparkfun.com/assets/1/4/2/1/9/TFmini_Plus_A02_Product_Manual_EN.pdf. [dostęp 01.12.2020].
- [8] Dokumentacja techniczna sensora benewake lidar tfmini-s. https://www.mybotshop.de/Datasheet/Benewake_TFMini-S_Datasheet.pdf. [dostęp 01.12.2020].
- [9] Dokumentacja techniczna sensora slamtec rplidar a1m8. <https://www.digikey.bg/htmldatasheets/production/3265529/0/0/1/A1M8.pdf>. [dostęp 01.12.2020].
- [10] Dokumentacja techniczna sensora slamtec rplidar a2m6. https://www.generationrobots.com/media/LD206_SLAMTEC_rplidar_datasheet_A2M6_v0.1_en.pdf. [dostęp 01.12.2020].
- [11] Dokumentacja techniczna sensora slamtec rplidar a2m8. https://cdn.sparkfun.com/assets/e/a/f/9/8/LD208_SLAMTEC_rplidar_datasheet_A2M8_v1.0_en.pdf. [dostęp 01.12.2020].
- [12] Dokumentacja techniczna sensora sparkfun lidar lite v3. https://download.kamami.pl/p562733-LIDAR_Lite_v3_Operation_Manual_and_Technical_Specifications.pdf. [dostęp 01.12.2020].
- [13] Dokumentacja techniczna sensora sparkfun lidar lite v3hp. https://cdn.sparkfun.com/assets/9/a/6/a/d/LIDAR_Lite_v3HP_Operation_Manual_and_Technical_Specifications.pdf. [dostęp 01.12.2020].

- [14] Enkoder ec-11. <https://botland.com.pl/pl/enkodery/6903-enkoder-z-przyciskiem-20-impulsow-12mm-ec11-pionowy.html>. [dostęp 02.12.2020].
- [15] Filament devildesign pla żółty. <https://3lian.pl/produkt/pla-175-zolty-dd/>. [dostęp 02.12.2020].
- [16] Filament devildesign tpu czarny. <https://3lian.pl/produkt/tpu-175-czarny-devil-design-330g-mala-szpula/>. [dostęp 02.12.2020].
- [17] Konwerter usb-uart. <https://botland.com.pl/pl/konwertery-usb-uart-rs232-rs485/5341-konwerter-usb-uart-ftdi-ft232rl-gniazdo-microusb-waveshare-11324.html>. [dostęp 02.12.2020].
- [18] Moduł bluetooth hc-05. <https://botland.com.pl/pl/moduly-bluetooth/2891-modul-bluetooth-hc-05-v2.html>. [dostęp 02.12.2020].
- [19] Ogniwko litowo-jonowe. <https://botland.com.pl/pl/akumulatory-lion/10036-ogniwo-18650-li-ion-sony-us18650vtc5a-2600mah.html>. [dostęp 02.12.2020].
- [20] Para konektorów xt30. <http://www.modele.sklep.pl/pl/Katalog/KONEKTORY-PRZEWODY-AKCESORIA/Konektory/M-WTYK-XT30-PARA.html>. [dostęp 02.12.2020].
- [21] Przetwornica step-down. <https://botland.com.pl/pl/przetwornice-step-down/2967-przetwornica-step-down-lm2596-32v-35v-3a-5903351241397.html>. [dostęp 02.12.2020].
- [22] Przycisk wł/wył z podświetleniem. <https://www.banggood.com/19mm-Metal-Waterproof-12-24V-5Pin-ON-OFF-Push-Button-Switch-LED-Power-Switch-p-1364500.html>. [dostęp 02.12.2020].
- [23] Płytki “blue pill” stm32f103. <https://kamami.pl/zestawy-uruchomieniowe-stm32/579165-stm32f103c8t6-bluepill-zestaw-evaluacyjny-z-mikrokontrolerem-stm32f103c8t6.html>. [dostęp 02.12.2020].
- [24] Płytki uniwersalna dwustronna. <https://botland.com.pl/pl/plytki-uniwersalne/2744-plytka-uniwersalna-dwustronna-40x60mm.html>. [dostęp 02.12.2020].
- [25] Qt for python - implementacja środowiska qt5 dla języka python. <https://doc.qt.io/qtforpython/index.html>. [dostęp 30.11.2020].
- [26] Repozytorium kodu niniejszego projektu. <https://github.com/Tomaszu97/ScanBot>. [dostęp 30.11.2020].
- [27] Repozytorium kodu stm32duino. <https://github.com/stm32duino>. [dostęp 24.11.2020].
- [28] Rolka przewodu drucianego. <https://botland.com.pl/pl/przewody-jednozylowe/12584-przewod-druciany-pcv-025mm-8-kolorow-rolka-250m.html>. [dostęp 02.12.2020].
- [29] Sensor lidar tf luna. <https://botland.com.pl/pl/skanery-laserowe/16638-laserowy-czujnik-odleglosci-lidar-tf-luna-8m-uarti2c.html>. [dostęp 02.12.2020].
- [30] Serwomechanizm towerpro mg-90s. <http://www.modele.sklep.pl/pl/Katalog/Serwa/Serwa-analogowe/Tower-Pro-MG-90S.html>. [dostęp 02.12.2020].

- [31] Serwomechanizm towerpro mg-995. <http://www.modele.sklep.pl/pl/Katalog/Serwa/Serwa-cyfrowe/Tower-Pro-MG-995.html>. [dostęp 02.12.2020].
- [32] Strona główna projektu platformio. <https://platformio.org/platformio-ide>. [dostęp 24.11.2020].
- [33] Strona główna projektu python. <https://www.python.org/>. [dostęp 24.11.2020].
- [34] Strona główna standardu języka c++. <https://isocpp.org/>. [dostęp 24.11.2020].
- [35] Strona informacyjna kodu i algorytmu gmapping. <https://openslam-org.github.io/gmapping.html>. [dostęp 24.11.2020].
- [36] Strona informacyjna płytka "blue pill". <https://stm32-base.org/boards/STM32F103C8T6-Blue-Pill.html>. [dostęp 24.11.2020].
- [37] Strona modułu gmapping do ros. <http://wiki.ros.org/gmapping>. [dostęp 24.11.2020].
- [38] Strona modułu rviz. <http://wiki.ros.org/rviz>. [dostęp 24.11.2020].
- [39] Strona opisująca historię robotycznych odkurzaczy. <http://www.vacuumcleanerhistory.com/vacuum-cleaner-development/history-of-robotic-vacuum-cleaner/>. [dostęp 24.11.2020].
- [40] Strona programu autodesk inventor. <https://www.autodesk.com/products/inventor/overview>. [dostęp 24.11.2020].
- [41] Strona projektu ros. <https://www.ros.org/>. [dostęp 24.11.2020].
- [42] Strona płytka arduino. <https://www.arduino.cc/en/Main/Products>. [dostęp 24.11.2020].
- [43] Wyświetlacz oled. https://kamami.pl/wyswietlaczeoledgraficzne/574495-modoled096i2cgvss-wyswietlaczoled-096-i2c-ze-sterownikiem-ssd1306.html?search_query=ssd1306&results=27. [dostęp 02.12.2020].
- [44] Zasilacz do ładowarki pakietów. <http://www.modele.sklep.pl/pl/Katalog/LADOWARKI-ZASILACZE-I-BALANCERY/ZASILACZE/Zasilacz-sieciowy-12V-5A-REDOX.html>. [dostęp 02.12.2020].
- [45] Ładowarka pakietów litowo-jonowych. <http://www.modele.sklep.pl/pl/Katalog/LADOWARKI-ZASILACZE-I-BALANCERY/LADOWARKI/REDOX-ALPHA-V2.html>. [dostęp 02.12.2020].
- [46] J. Borenstein, L. Feng. Correction of systematic odometry errors in mobile robots. 1995.
- [47] G. Grisetti, C. Stachniss, W. Burgard. Improved techniques for grid mapping with rao-blackwellized particle filters. *IEEE Transactions on Robotics*, 23(1):34–46, 2007.
- [48] G. Grisetti, C. Stachniss, W. Burgard. Improving grid-based slam with rao-blackwellized particle filters by adaptive proposals and selective resampling. *Proceedings of the 2005 IEEE International Conference on Robotics and Automation*, strony 2432–2437, 2005.
- [49] A. Kuncar, M. Sysel, T. Urbanek. Calibration of low-cost triaxial magnetometer. *MATEC Web of Conferences*, 2016.
- [50] J. Kędzierski. Filtr kalmana - zastosowania w prostych układach sensorycznych. 03 2016.
- [51] C. Mihai, A. Nilgesz, F. Birouas, R. Tarca. Hard iron distortion compensation for 3 axis magnetometer. *Recent Innovations in Mechatronics*, 2016.

- [52] K. Murphy. Bayesian map learning in dynamic environments. 05 2000.
- [53] C. Reas, B. Fry. *Getting Started With Processing*. O'Reilly Media Inc., 2010.

Dodatek A

Opis załączonej płyty CD/DVD

Na załączonej płycie znajdują się 3 foldery:

1. 3d-model - w tym folderze znajduje się trójwymiarowy projekt robota wykonany w programie Autodesk Inventor.
2. src-firmware - w tym folderze znajduje się kod źródłowy oprogramowania platformy napisany w środowisku *PlatformIO* [32].
3. src-pc - folder zawierający moduł *scanbot_communicator* w którym znajduje się aplikacja sterująca.

Ad.1

Plik *Assembly1.ipm* zawiera projekt podwozia wraz z górną pokrywą i kołami.

Plik *arm.ipt* to projekt wieżyczki na której zamieszczony jest sensor laserowy.

Plik *tracks_v2.ipt* zawiera projekt gąsienicy.

Ad.2

Cały folder należy otworzyć w środowisku *PlatformIO*.

Główny plik z programem (*main.cpp*) znajduje się w podfolderze *src*.

Ad.2

Właściwy moduł *ROS* znajduje się w podfolderze *scanbot_communicator*. Instrukcje dotyczące korzystania, w języku angielskim, znajdują się w pliku *README.md*. Dodatkowo plik *bag1.bag* zawiera nagranie komunikacji zachodzącej za pośrednictwem tematów *ROS* podczas przejazdu robota, wykorzystywane przy budowaniu ostatniej z przedstawionych w niniejszym dokumencie map. Instrukcja w jaki sposób odtworzyć to nagranie znajduje się we wspomnianym pliku *README.md*.