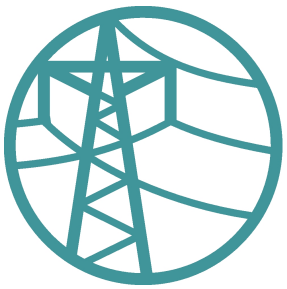


WYDZIAŁ
ELEKTROTECHNIKI
I INFORMATYKI
POLITECHNIKI RZESZOWSKIEJ

Michał Bazan

Porównanie algorytmów nawigacyjnych

PRACA MAGISTERSKA

Opiekun pracy:
dr inż. Dariusz Rzońca

Rzeszów, 2024

Spis treści

Wykaz symboli, oznaczeń i skrótów

a) PID (eng. Proportional-Integral-Derivative) - skrót od Proporcjonalno-Całkująco-Różniczkującego regulatora, który jest powszechnie stosowany w systemach sterowania,

b) MCU (eng. Microcontroller Unit) - Jednostka Mikrokontrolera,

c) SQT (eng. Software Qualification Test) to proces testowania oprogramowania w celu zweryfikowania, czy spełnia ono określone wymagania i standardy jakościowe. SQT ma na celu potwierdzenie, że oprogramowanie działa zgodnie z założeniami i spełnia oczekiwania użytkowników oraz wymagania funkcjonalne i niefunkcjonalne.

1. Wstęp

W dzisiejszych czasach, wraz z dynamicznym rozwojem technologii mobilnych, algorytmy nawigacyjne i uczenia maszynowego [2] odgrywają kluczową rolę w różnorodnych aplikacjach, począwszy od systemów nawigacji w samochodach po autonomiczne roboty poruszające się w różnych środowiskach. Biorąc pod uwagę aktualność tych zagadnień i rosnące zapotrzebowanie, zdecydowano o przeprowadzeniu badań dotyczących systemów nawigacyjnych.

Celem tej pracy jest zbadanie heurystycznych metod [4] optymalizazcji regulatorów PID oraz porównanie wybranych algorytmów pod kątem kryteriów takich jak dokładność wyznaczania trasy oraz wydajność obliczeniowa w różnych warunkach terenowych.

Zakres pracy obejmuje dwie części:

a) część inżynierska - wykonanie robota mobilnego, implementacja systemu wbudowanego oraz implementacja oprogramowania sterującego robotem poprzez dostępny interfejs,

b) część badawcza - badanie algorytmów uczenia maszynowego do optymalizacji nastaw regulatorów PID oraz porównanie wybranych algorytmów nawigacyjnych.

Zastosowanie standardu ASPICE [5] zapewnia wysoką jakość procesu budowy robota oraz implementacji oprogramowania co pozwola na przeprowadzenie rzetelnych badań i wyciągnięcie wiarygodnych wniosków.

W dalszej części pracy szczegółowo omówione zostaną poszczególne etapy realizacji każdego z tych komponentów, prezentując zarówno teoretyczne założenia, jak i praktyczne wyniki osiągnięte w ramach projektu.

W następnych rozdziałach pracy przedstawiono szczegółowy opis inżynierskiej części projektu oraz wyniki przeprowadzonych badań. W rozdziale poświęconym inżynierskiej części projektu omówiono budowę robota, jego system operacyjny oraz prostą stację operatorską, które stanowią podstawę do realizacji badań algorytmicznych. Następnie, w rozdziale dotyczącym badanych algorytmów, zaprezentowano teoretyczne i praktyczne aspekty algorytmu genetycznego, algorytmów nawigacyjnych statycznych oraz dynamicznych. Kolejny rozdział skupia się na opisie przeprowadzonych badań, w których dokonano optymalizacji nastaw regulatora PID oraz przeprowadzono porów-

nanie skuteczności algorytmów nawigacyjnych. Te szczegółowe analizy mają na celu ocenę wydajności poszczególnych rozwiązań oraz identyfikację optymalnych metod nawigacyjnych dla zbudowanego robota.

2. Inżynierska część projektu

W tym rozdziale przedstawiono kompleksowy opis prac związanych z implementacją oraz funkcjonowaniem robota. Niniejszy rozdział stanowi szczegółowe omówienie trzech kluczowych elementów projektu, które skupiały się na budowie fizycznej robota zgodnie z procedurami SPICE, implementacji oprogramowania w języku C++ z uwzględnieniem unit testów oraz oprogramowania do sterowania robotem. Dzięki zastosowaniu tych trzech elementów możliwe było zapewnienie nie tylko skutecznej implementacji samego robota, ale również jego oprogramowania oraz efektywnego zarządzania nim w czasie rzeczywistym. Dokładna dokumentacja wszystkich komponentów projektu oraz testów znajduje się na repozytorium Github [1].

2.1. Budowa robota

Pierwszym aspektem, który został przedstawiony, jest proces budowy robota. Opisane zostały tutaj szczegóły dotyczące wyboru komponentów oraz implementacji elektroniki sterującej. Proces ten obejmuje kilka kluczowych etapów, które mają na celu zapewnienie, że robot będzie w stanie spełniać wszystkie założone funkcje i wymagania. W szczególności skupiono się na doborze odpowiednich czujników, kontrolerów, silników oraz systemu zasilania, które umożliwią robotowi prawidłowe działanie w różnych warunkach.

2.1.1. Wymagania sprzętowe

W celu dobrania właściwych elementów do budowy robota, sformułowano niżej umieszczone wymagania wysokiego poziomu. Te wymagania określają kluczowe funkcje i cechy, jakie powinny posiadać komponenty sprzętowe robota, aby zapewnić jego pełną funkcjonalność i niezawodność. W tabeli 2.1 zestawiono najważniejsze z tych wymagań, które dotyczą obsługi czujników, komunikacji bezprzewodowej, napędu oraz zarządzania zasilaniem.

Tablica 2.1. Wymagania wysokiego poziomu HWE1

ID_HWE1	Opis
HWE_1_010	Sprzęt powinien wspierać obsługę czujników odległości.
HWE_1_020	Kontroler powinien udostępniać moduł WiFi.
HWE_1_030	Robot powinien być wyposażony w szczotkowe silniki DC z enkoderami.
HWE_1_040	Sprzęt powinien wspierać obsługę enkoderów.
HWE_1_060	Sprzęt powinien mieć zaimplementowany odpowiedni system dystrybucji zasilania.

Na podstawie wyżej ukazanych wymagań wysokiego poziomu, sformułowano następujące wymagania niskiego poziomu. Te szczegółowe wymagania określają dokładne specyfikacje techniczne i parametry, które muszą być spełnione przez komponenty, aby zapewnić zgodność z ogólnymi celami projektu. W tabeli 2.2 zestawiono kluczowe wymagania dotyczące interfejsów komunikacyjnych, zasilania oraz czujników.

Tablica 2.2. Wymagania niskiego poziomu HWE2

ID_HWE2	Opis
HWE_2_010	Wszystkie interfejsy komunikacyjne powinny wspierać logikę 3V3.
HWE_2_020	Mikrokontroler powinien być wyposażony w moduł WiFi.
HWE_2_030	System dystrybucji zasilania powinien zasilić logikę.
HWE_2_040	System dystrybucji zasilania powinien zasilić silniki.
HWE_2_060	Czujnik odległości powinien mieć zakres pomiarowy wynoszący conajmniej 200 cm.
HWE_2_070	Czujniki odległości powinny udostępniać interfejs komunikacyjny kompatybilny z interfejsami mikrokontrolera.

Niżej umieszczona figura 2.1 ukazuje schemat blokowy utworzony na podstawie wcześniej zdefiniowanych wymagań sprzętowych. Wskazuje, jakie interfejsy i poziomy napięć zasilania zostały wykorzystane pomiędzy poszczególnymi blokami. Sprzęt zdefiniowany w ten sposób oraz zbiór wymagań stawianych przed urządzeniem umożliwił dobór właściwych komponentów.

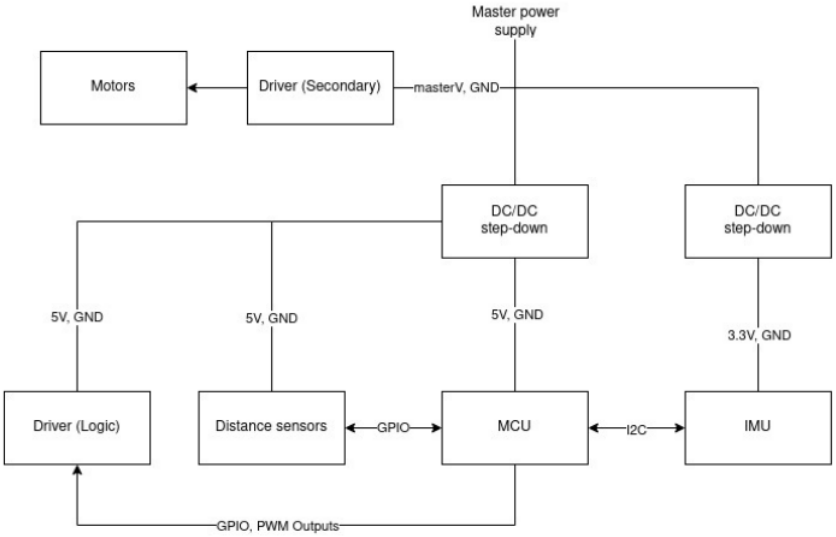


Figure 1: Hardware block diagram

Rys. 2.1. Schemat blokowy sprzętu

2.1.2. Schemat elektryczny

Niniejsza sekcja skupia się na wyjaśnieniu schematu elektrycznego robota. Na podstawie umieszczonych w poprzedniej sekcji wymagań dokonano wyboru komponentów, które zostały wykorzystane w projekcie, ale dokładny opis elementów został umieszczony w dokumentacji tej części projektu na repozytorium Github [1].

W celu ułatwienia procesu implementacji i zmitygowaniu potencjalnych błędów, schemat elektryczny został podzielony na bloki zgodnie z podziałem ukazanym na rysunku 2.1.

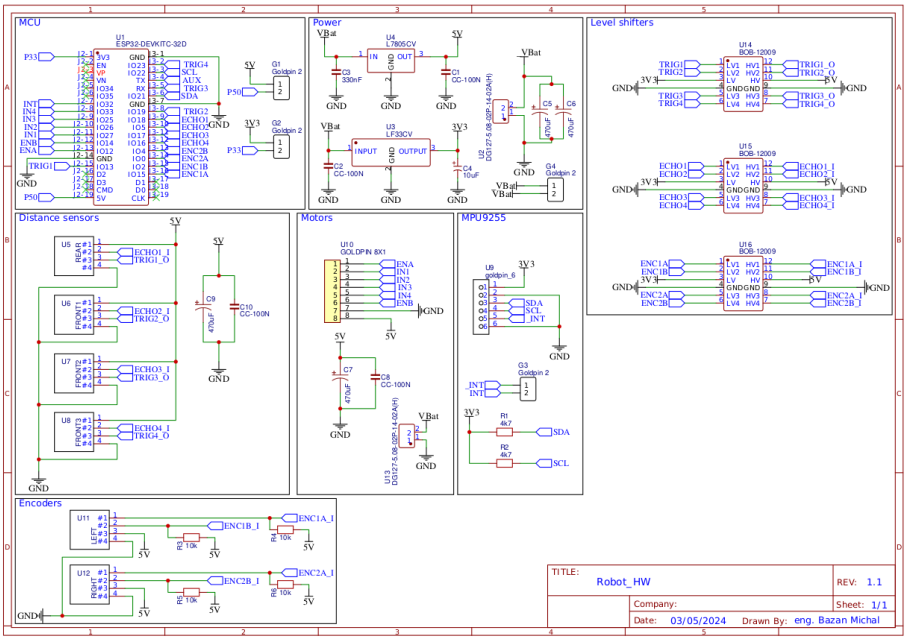
Wyżej umieszczona ilustracja 2.2 ukazuje połączenia pomiędzy blokami schematu:

a) MCU - blok definiuje wejścia i wyjścia sterujące oraz połączenia interfejsów komunikacyjnych,

b) Power - sekcja odpowiedzialna za dystrybucję zasilania,

c) Level shifters - konwertery poziomów logicznych, które zapewniają kompatybilność poziomów sygnałów elektrycznych,

d) Distance sensors - blok definiuje połączenia pomiędzy mikrokontrolerem i czujnikami odległości,



Rys. 2.2. Schemat elektryczny

e) Motors - sekcja ukazuje sygnały sterujące silnikami,

f) MPU9255 - blok został zaimplementowany, ale nie jest wykorzystywany,

g) Encoders - ta sekcja ukazuje sygnały wyjściowe z enkoderów.

2.1.3. Testy

Testowanie tej części projektu polegało głównie na weryfikacji założeń i projektu płytki, dlatego aspekt ten nie został poruszony w tej sekcji. Szczegółowa dokumentacja znajduje się w zdalnym repozytorium [1]. Testy obejmowały sprawdzenie zgodności z wymaganiami, poprawność połączeń oraz funkcjonalność poszczególnych bloków.

2.2. System operacyjny robota

Ta sekcja skupia się na implementacji oprogramowania w języku C++, obejmującej zarówno projektowanie, jak i implementację funkcjonalności. System operacyjny robota zarządza wszystkimi aspektami operacyjnymi robota, od odczytu danych z czujników, poprzez przetwarzanie i analizę tych danych, aż po sterowanie silnikami i komunikację zewnętrzną.

2.2.1. Wymagania dotyczące oprogramowania

Na podstawie wcześniej omówionych wymagań sprzętowych oraz celów tego projektu, sformułowano następujące wymagania wysokiego poziomu. Wymagania te definiują kluczowe funkcjonalności, jakie powinno posiadać oprogramowanie, aby zapewnić niezawodność działania robota.

Tablica 2.3. Wymagania wysokiego poziomu SWE2

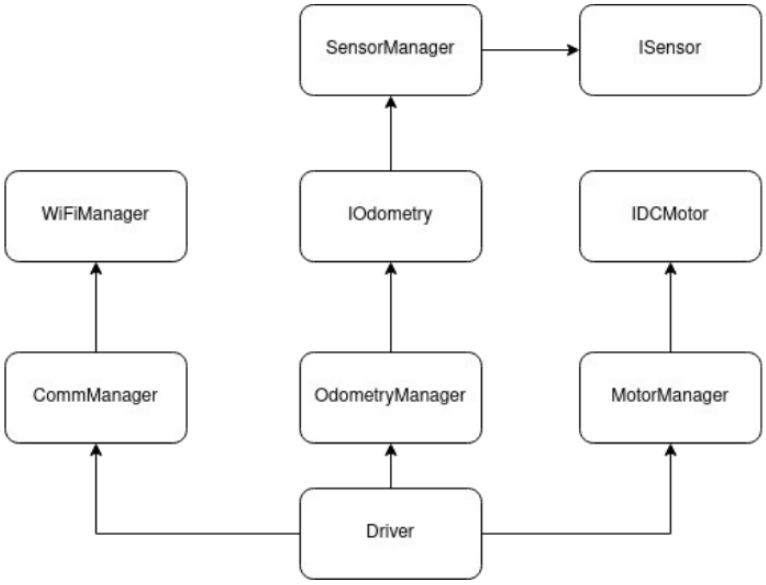
ID_SWE2	Opis
SWE_2_010	Oprogramowanie powinno odczytywać sensory periodycznie.
SWE_2_020	Interfejs WiFi powinien być wykorzystany do komunikacji z oprogramowaniem sterującym.
SWE_2_030	Oprogramowanie powinno udostępniać interfejs do sterowania silnikami.
SWE_2_040	Komunikacja powinna wykorzystywać prosty protokół komunikacyjny do wymiany danych pomiędzy robotem a oprogramowaniem sterującym.
SWE_2_050	Oprogramowanie powinno implementować algorytm wyznaczający odometrię.

W celu wyłonienia konkretnych bloków, z jakich powinno składać się oprogramowanie, zdefiniowano następujące wymagania niskiego poziomu. Te szczegółowe wymagania określają specyficzne zadania i interfejsy, które muszą być zaimplementowane, aby spełnić wymagania wysokiego poziomu.

Tablica 2.4. Wymagania niskiego poziomu SWE3

ID_SWE3	Opis
SWE_3_010	Oprogramowanie powinno odczytywać sensory co 50ms.
SWE_3_020	Oprogramowanie powinno udostępniać interfejs do wymiany danych.
SWE_3_030	Oprogramowanie powinno udostępniać dane z sensorów na żądanie.
SWE_3_040	Komunikacja powinna opierać się o wykorzystanie protokołu MQTT.
SWE_3_050	Oprogramowanie powinno udostępnić interfejs do sterowania silnikami.
SWE_3_060	Implementacja powinna wykorzystywać prostą odometrię.

Na podstawie wymagań umieszczonych w tabeli 2.4 zdefiniowano niżej ukazaną architekturę systemu operacyjnego robota.



Rys. 2.3. Architektura oprogramowania

Na rysunku 2.3 przedstawiono architekturę systemu operacyjnego robota, która została zaprojektowana w oparciu o zdefiniowane wymagania niskiego poziomu. Architektura ta składa się z kilku kluczowych modułów:

a) CommManager - odpowiada za połączenie sieciowe i komunikację za pośrednictwem protokołu MQTT,

b) SensorManager - posiada uchwyt do wszystkich dostępnych sensorów, które obsługuje periodycznie,

c) OdometryManager - posiada uchwyt do obiektu typu SensorManager i realizuje odometrię na podstawie odczytów z czujników,

d) MotorManager - steruje silnikami zależnie od nadchodzących komunikatów,

e) Driver - odpowiada za poprawną inicjalizację systemu, wywoływanie funkcji callbackowych z bloku odpowiedzialnego za komunikację oraz ustawienie prędkości obrotowej silników zależnie od uchybu regulacji.

2.2.2. Implementacja

Implementacja systemu operacyjnego robota składa się z 12 klas, z których każda pełni określoną rolę w funkcjonowaniu robota. Klasy te są odpowiedzialne za zarządzanie różnymi aspektami operacyjnymi, od odczytu danych z czujników po sterowanie silnikami. Jednym z ważnych elementów implementacji jest także zaimplementowany prosty szablon regulatora PID oraz stos oparty o std::array[6].

Listing 2.1. Klasa CommManager

```
template<class StringType, std::size_t topic_num>
class CommManager{
public:
    using messageType = std::tuple<StringType, StringType>;
    using onMsg_cb = std::function<void(char*, byte*, unsigned int)>;

    static messageType createMessage(const StringType& topic, const StringType&
    payload) noexcept;

    CommManager(
        WiFiManager<StringType>&& w,
        const StringType& uname,
        const StringType& broker,
        const StringType& password,
        WiFiClient& c,
        std::array<StringType, topic_num>&& topics,
        const onMsg_cb& cback,
        const StringType& ID = "Robot"
        ) noexcept;

    MQTTStatus poolCommManager() noexcept;

    bool sendMessage(messageType&& message) noexcept;

private:
    NetworkStatus poolNetwork() noexcept;
    SemaphoreHandle_t MQTTQueueMutex;

    enum : uint8_t{
        ConnectingNetwork = 0,
        ConnectingBroker,
        Timeout,
        CheckingStatus
    };
    uint8_t m_internalState;
    WiFiManager<StringType> m_wifiMgr;
    custom::stack<messageType, stack_depth> m_messageStack;
    StringType m_broker;
    StringType m_password;
    StringType m_uname;
    StringType m_ID;
    const std::array<StringType, topic_num> m_topics;
    PubSubClient m_mqttClient;
    onMsg_cb m_message_cb;
};
```

Klasa CommManager (Listing 2.1) zarządza komunikacją między robotem a oprogramowaniem sterującym poprzez protokół MQTT i połączenie WiFi. Oferuje funkcje odczytu danych z czujników, wysyłania komunikatów i reakcji na przychodzące komunikaty. Metoda createMessage tworzy nową wiadomość MQTT na podstawie

tematu i treści. W konstruktorze inicjalizowane są połączenie WiFi, parametry MQTT i lista subskrybowanych tematów. poolCommManager zarządza połączeniem z WiFi i brokerem MQTT, podejmując odpowiednie działania w zależności od stanu połączenia. sendMessage dodaje wiadomość do kolejki komunikatów. Klasa zapewnia niezawodne połączenie i efektywną komunikację, co jest kluczowe dla funkcjonowania robota w różnych scenariuszach działania.

Listing 2.2. Klasa SensorManager

```
enum class SensorMapping : uint8_t{
    LEFT_Encoder,
    RIGHT_Encoder,
    DST_Front_1,
    DST_Front_2,
    DST_Front_3,
    DST_Rear
};

constexpr uint8_t translate(SensorMapping s){
    return static_cast<uint8_t>(s);
}

class SensorManager{
public:
    void init();
    SensorManager(std::array<sensorPointer, numSensors>&& s);
    void poolSensors();
    ISensor& getSensor(SensorMapping s);

private:
    std::array<sensorPointer, numSensors> sensors;
};
```

Klasa SensorManager (Listing 2.2) zarządza sensorami robota, przechowując ich wskaźniki i udostępniając interfejs odczytu danych. Typ SensorMapping reprezentuje indeksy sensorów, ułatwiając dostęp do nich. Metoda translate przekształca enum na liczby całkowite, co umożliwia łatwiejszy dostęp do konkretnego urządzenia. Konstruktor inicjuje obiekt, a init inicjuje sensory. poolSensors odpytuje sensory, a getSensor zwraca referencję do konkretnego sensora.

Listing 2.3. Klasa OdometryManager

```
enum ActiveOdometry : std::size_t {
    SimpleOdo = 0,
    AdvancedOdo = 1
};

constexpr std::size_t availableAlgorithms = 2;

class OdometryManager{
public:
    OdometryManager(
        std::array<IOdometry*, availableAlgorithms>&& odo,
        SensorManager& s
    ) noexcept;

    void updatePosition() noexcept;
    void setActiveOdometry(ActiveOdometry o) noexcept;
    std::size_t getActiveOdometry() const noexcept;
    const position getPosition() const noexcept;
    void resetActiveOdometry() noexcept;
private:
    std::array<IOdometry*, availableAlgorithms> m_odometryAgents;
    SensorManager& m_sensorMgr;
    std::size_t m_active;
};
```

Klasa OdometryManager zarządza różnymi algorytmami odometrii robota i aktualizuje jego pozycję na podstawie danych z sensorów. Enum ActiveOdometry definiuje dostępne algorytmy odometrii, lecz w tej implementacji dostępna jest tylko prosta odometria. Konstruktor przyjmuje tablicę wskaźników do obiektów IOdometry reprezentujących różne algorytmy odometrii oraz referencję do obiektu SensorManager. Metoda updatePosition aktualizuje pozycję robota na podstawie danych z sensorów i aktywnego algorytmu odometrii. setActiveOdometry ustawia aktywny algorytm odometrii, getActiveOdometry zwraca jego indeks, a getPosition zwraca aktualną pozycję. Metoda resetActiveOdometry resetuje aktywny algorytm i ustawia zerowe współrzędne. Klasa zapewnia elastyczne zarządzanie algorytmem odometrii i precyzyjne poruszanie się robota.

Listing 2.4. Klasa MotorManager

```
class MotorManager{
public:
    enum class settingType{
        setAngularTarget,
        updatePwm
    };

    using motor_array = std::array<IDCMotor*, motor_num>;
    using speed_array = std::array<int32_t, motor_num>;
    using angular_array = std::array<float, motor_num>;

    void init() noexcept;
    void setSpeed(const speed_array& speeds, settingType t = settingType::updatePwm)
     noexcept;
    MotorManager(motor_array&& m, float i = inertia_coef) noexcept;
    void InertiaCoef(float i) noexcept;
    float InertiaCoef() const noexcept;
```

```
17      void poolMotors() noexcept;
18      const speed_array* CurrentSpeed() const noexcept;
19      const speed_array* DesiredSpeed() const noexcept;
20      const angular_array* TargetAngular() const noexcept;
21
22  private:
23      float m_inertiaCoef;
24      motor_array motors;
25      speed_array target_percent_speed;
26      speed_array current_percent_speed;
27      angular_array target_angular_speed;
28  };
```

Klasa MotorManager zarządza silnikami prądu stałego robota, kontrolując ich prędkość obrotową i kierunek. settingType definiuje różne typy ustawień silników, a motor_array, speed_array, i angular_array to odpowiednio tablice wskaźników do silników, prędkości silników i prędkości kątowych. Metoda init inicjuje silniki, setSpeed ustawia prędkość zależnie od typu ustawienia, a konstruktor inicjuje obiekt, przyjmując tablicę wskaźników do silników i opcjonalny współczynnik inercji. poolMotors aktualizuje stany silników, a CurrentSpeed, DesiredSpeed, i TargetAngular zwracają odpowiednio aktualną prędkość, docelową prędkość i prędkości kątowe silników. Klasa zapewnia elastyczne zarządzanie silnikami i kontrolę nad ich prędkościami, kluczową dla ruchu i sterowania robotem.

Listing 2.5. Klasa Kernel

```
1   class Kernel{
2   public:
3       static void init();
4       static void main();
5
6       /* motion management */
7       static Motor::DCMotor motorLeft;
8       static Motor::DCMotor motorRight;
9       static Motor::MotorManager motorManager;
10      static uint32_t mapper(uint32_t);
11      static constants::motors::types::motor_pid pidLeft;
12      static constants::motors::types::motor_pid pidRight;
13
14      /* communications management */
15      static constants::comm::types::job_stack_t jobStack;
16      static Comm::MQTT::CommManager<String, constants::comm::subscribedTopics>
        commMgr;
17      static void MQTTcallback(char*, byte*, unsigned int);
18
19      /* sensor management */
20      static Sensor::DistanceSensor rear;
21      static Sensor::DistanceSensor front_left;
22      static Sensor::DistanceSensor front;
23      static Sensor::DistanceSensor front_right;
24      static Sensor::Encoder encoderLeft;
25      static Sensor::Encoder encoderRight;
26
27      static Sensor::SensorManager sensorMgr;
28
29      static Sensor::simpleOdometry odoAgent;
30      static Sensor::OdometryManager odoMgr;
31  };
```

Klasa Kernel pełni kluczową rolę jako centralny moduł zarządzający systemem. Odpowiada za inicjalizację systemu oraz koordynację jego głównych funkcji. Metoda init przygotowuje system do działania, a main zarządza jego główną logiką. Kluczowe funkcje obejmują zarządzanie ruchem (silniki, regulator PID), komunikacją (MQTT), sensorami (odległości, enkodery) oraz odometrią. Kernel integruje te komponenty, zapewniając ich współpracę i efektywną działalność.

2.2.3. Obsługiwane polecenia

Sekcja przedstawia interfejs sieciowej komunikacji z robotem poprzez protokół MQTT. Tabela 2.5 zawiera tematy subskrybowane przez robota wraz z opisem ich przeznaczenia oraz przykładowymi danymi. Tabela ta obejmuje kanały do debugowania, przesyłania komend oraz ustawiania prędkości silników. Natomiast w tabeli 2.6 znajdują się tematy publikowane przez robota, wraz z opisem i typowymi danymi przesyłanymi przez te kanały, takimi jak odpowiedzi na komendy i wiadomości debugujące. W tabeli 2.7 znajdują się dostępne komendy, ich opisy oraz oczekiwane odpowiedzi. Komendy obejmują pobieranie odczytów z czujników, resetowanie pozycji, pobieranie danych odometrycznych oraz zatrzymywanie silników.

Tablica 2.5. Tematy subskrybowane przez robota

Robot subskrybuje	Opis	Dane
robot/echo/in	Temat wykorzystywany do debugu, system odpowiada ciągiem tekstowym na temacie robot/echo/out.	Dowolny ciąg tekstowy zgodny z ASCII.
robot/cmd/in	Temat do przesyłania zdefiniowanych komend (tab. 2.7).	Komenda jako ciąg tekstowy zgodny z ASCII.
robot/set/motors	Temat do przesyłania żądanych prędkości silników.	Dane w formacie JSON: { „left” : x, „right” : y }, gdzie x oraz y to prędkości wyrażone w procentach.

Tablica 2.6. Tematy publikowane przez robota

Robot subskrybuje	Opis	Dane
robot/echo/out	Temat, na który system przesyła wiadomości dotyczące debugu.	Dowolny ciąg tekstowy zgodny z ASCII.
robot/cmd/response	Odpowiedź na przesłaną komendę.	Dane w formacie JSON (odczyty z sensorów, odometria), brak odpowiedzi lub ciąg tekstowy ćmd not supported"w przypadku przesłania błędnej komendy.

Tablica 2.7. Dostępne komendy

Komenda	Opis	Odpowiedź
get_sensors	Pobierz odczyty z czujników.	Dane z czujników w formacie JSON.
reset_odo	Zeruj pozycję.	Brak odpowiedzi.
get_odo	Pobierz wyznaczone dane odometryczne.	Dane w formacie JSON.
get_all	Pobierz dane z sensorów i pozycję robota.	Dane w formacie JSON.
halt	Zatrzymaj silniki.	Brak odpowiedzi.

2.2.4. Testy oprogramowania

W tej sekcji opisano przeprowadzone testy weryfikujące zgodność oprogramowania z wymaganiami określonymi w specyfikacji SWE3.

Tablica 2.8. Test SQT_01

Treść wymagania	Weryfikacja	Wynik testu
Oprogramowanie powinno odczytywać sensory co 50ms.	Zadanie odczytywania sensorów jest zaplanowane w systemie freeRTOS co 50ms.	Wynik pozytywny

Tablica 2.9. Test SQT_02

Weryfikacja	Oczekiwane	Obserwacja
Podłączenie płytki do zasilania	Płytka powinna być poprawnie zasilona	Płytka została poprawnie zasilona
Wgranie najnowszej wersji oprogramowania na płytkę	Najnowsze oprogramowanie powinno zostać wgranę na płytkę	Najnowsze oprogramowanie zostało wgrane na płytkę
Oczekiwanie na wiadomość żobot initializedńa temacie robot/echo/out.	Wiadomość powinna zostać odebrana.	Wiadomość została odebrana.
Przesłanie wiadomości na temacie robot/set/motors, aby ustawić prędkość silników bliską 40% prędkości maksymalnej.	Silniki powinny kręcić się z prędkością bliską 40% prędkości maksymalnej.	Silniki kręcą się z prędkością bliską 40% prędkości maksymalnej.
Przesłanie komendy halt na temacie robot/cmd/in.	Silniki powinny się zatrzymać.	Silniki zatrzymały się.
Przesłanie komendy get_all w celu uzyskania wszystkich danych.	Uzyskano dane z sensorów i pozycję robota.	Uzyskano dane z sensorów i pozycję robota.
Przesłanie komendy reset_odo w celu resetowania odometrii.	Wiadomość powinna zostać wysłana.	Wiadomość została wysłana.
Przesłanie komendy get_odo w celu uzyskania pozycji robota.	Uzyskano pozycję, która wskazuje na zero.	Uzyskano pozycję, która wskazuje na zero.

Test SQT_01 miał na celu potwierdzenie, czy oprogramowanie regularnie odczytuje dane z czujników odległości co 50 ms, zgodnie z wymaganiami. Mimo że w projekcie nie wykorzystano IMU, to test został dostosowany do odczytu danych z czujników odległości, a zadanie zostało zaplanowane jako praca freeRTOS z interwałem 50 ms.

Testy oznaczone jako SQT_02SWE, skupiły się na weryfikacji podstawowych funkcji oprogramowania, w tym dostarczania zasilania i programowania sprzętu, wymiany danych, kontroli silników itp. Każde z tych wymagań zostało poddane testowi, a wynikiem było potwierdzenie, że oprogramowanie spełnia te kryteria.

W obu przypadkach testy zakończyły się sukcesem, co potwierdza zgodność oprogramowania z wymaganiami określonymi w specyfikacji SWE3. Oprócz przeprowadzenia testów kwalifikacyjnych, oprogramowanie zostało pokryte testami jednostkowymi.

3. Badane algorytmy

3.1. Algorytm PID

Algorytm PID [7] jest podstawowym narzędziem używanym w systemach sterowania. Celem tego regulatora jest minimalizacja różnicy między wartością zadaną a rzeczywistą wartością procesu poprzez odpowiednie dostosowanie sygnału sterującego. Regulator PID składa się z trzech komponentów (wzór 3.4):

a) składnik proporcjonalny (wzór 3.1) - generuje sygnał sterujący proporcjonalny do bieżącego błędu. Wyższa wartość współczynnika wzmocnienia prowadzi do szybszej reakcji systemu, ale może powodować oscylacje lub niestabilność,

b) składnik całkujący (wzór 3.2) - eliminuje stałe odchylenie przez sumowanie błędów w czasie. Powoduje to wzrost sygnału sterującego, dopóki błąd nie zostanie zredukowany do zera. Współczynnik nie powinien być zbyt duży, ponieważ może powodować przeregulowania i oscylacje,

c) składnik różniczkujący (wzór 3.3) - reaguje na szybkość zmiany odpowiednio korygując sygnał sterujący. Pomaga w tłumieniu oscylacji i poprawia stabilność systemu. Zbyt duża wartość wzmocnienia może jednak sprawić, że regulator będzie agresywnie zmieniał wartość sterującą.

$$P = K_p \cdot e(t) \tag{3.1}$$

$$I = K_i \cdot \int_0^t \cdot e(\tau) d\tau \tag{3.2}$$

$$D = K_d \cdot \frac{de(t)}{dt} \tag{3.3}$$

$$PID = K_p \cdot e(t) + K_i \cdot \int_0^t \cdot e(\tau) d\tau + K_d \cdot \frac{de(t)}{dt} \tag{3.4}$$

gdzie: K_p – wzmocnienie członu proporcjonalnego, K_i – wzmocnienie członu całkującego, K_d – wzmocnienie członu różniczkującego, $e(t)$ – błąd regulacji

W regulatorze analogowym PID wszystkie operacje matematyczne są wykonywane w sposób ciągły przy użyciu komponentów elektronicznych, takich jak oporniki,

kondensatory i wzmacniacze operacyjne. Ze względu na wykorzystane technologie, w prezentowanym rozwiązaniu zastosowana została dyskretna wersja algorytmu, która w porównaniu do regulatora ciągłego, ograniczona jest niższą dokładnością całkowania i różniczkowania.

Listing 3.1. Kod implementacji cyfrowego algorytmu PID

```
template <typename ValueType, typename outType, outType MaxOut, outType MinOut,
    int32_t maxErrorSum, int32_t minErrorSum, uint32_t samplingMs = 20>
class pid{
public:
    /*
        Kp - proportional Gain
        Ti - integrator constant
        Td - derivative constant
        Ts - sampling time in ms
    */
    constexpr pid(float Kp, float Ti, float Td) noexcept:
        m_Kp{ Kp }, m_Ki{ (Kp / Ti) * (samplingMs * 0.001f) }, m_Kd{ (Kp * Td)/(
    samplingMs * 0.001f + 0.1f*Td) },
        m_errorSum{ 0 }, m_previousError{ 0 } {}

    void reinit(float Kp, float Ti, float Td){
        m_Kp = Kp;
        m_Ki = (Kp / Ti) * (samplingMs * 0.001f);
        m_Kd = (Kp * Td)/((samplingMs * 0.001f) + 0.1f*Td);
        m_errorSum = m_previousError = 0;
    }

    outType getOutput(ValueType actual, ValueType desired){
        auto error = desired - actual;
        float P = m_Kp * error;

        m_errorSum += static_cast<int32_t>(error);

        if(m_errorSum > maxErrorSum){
            m_errorSum = maxErrorSum;
        }
        else if(m_errorSum < minErrorSum){
            m_errorSum = minErrorSum;
        }

        float I = m_Ki * m_errorSum;
        float D = m_Kd * (error - m_previousError);

        auto output = static_cast<outType>(P + I + D);

        if(output > MaxOut){
            return MaxOut;
        }
        else if(output < MinOut){
            return MinOut;
        }
        else{
            return output;
        }
    }

private:
    float m_Kp;
    float m_Ki;
    float m_Kd;
    ValueType m_errorSum;
    ValueType m_previousError;
};
```

Kod ukazany na wyżej umieszczonym listingu 3.1 definiuje szablon klasy pid, który reprezentuje cyfrowy algorytm PID. Kluczowymi elementami tego algorytmu są trzy główne składowe: proporcjonalny, całkujący i różniczkujący.

Algorytm ten jest parametryzowany przez różne wartości, takie jak wzmocnienia (Kp, Ki, Kd) oraz ograniczenia (MaxOut, MinOut). Wartości te pozwalają na dostosowanie algorytmu do specyfiki danego systemu.

W metodzie getOutput, algorytm oblicza sygnał sterujący na podstawie bieżącej wartości i wartości zadanego stanu. Składa się na to trzy kroki:

a) Wyznaczenie składowej proporcjonalnej jako iloczynu wzmocnienia proporcjonalnego i aktualnego uchybu regulacji,

b) Obliczenie składowej całkującej jako sumy błędów z poprzednich kroków, pomnożonej przez wzmocnienie całkujące,

c) Obliczenie składowej różniczkującej jako różnicy błędu aktualnego i poprzedniego stanu, pomnożonej przez wzmocnienie różniczkujące.

3.2. Algorytm genetyczny

3.3. Algorytmy nawigacyjne

4. Badania

4.1. Optymalizacja nastaw regulatora PID prędkości obrotowej

4.2. Porównanie algorytmów nawigacyjnych w terenie bez przeszkód

4.3. Porównanie algorytmów nawigacyjnych w terenie z przeszkodami

5. Podsumowanie i wnioski końcowe

Załączniki

Literatura

[1] https://github.com/DevxMike/master_degree

[2] Francois Chollet: Deep Learning. Praca z językiem Python i biblioteką Keras. Helion 2019

[3] Paweł Cichosz: Systemy uczące się. WNT 2007

[4] Riccardo Poli, William B. Langdon, Nicholas F. McPhee, John R. Koza: A Field Guide to Genetic Programming. Lulu Enterprises Uk Ltd 2008

[5] https://mfiles.pl/pl/index.php/Automotive_SPICE

[6] https://en.cppreference.com/w/cpp/container/array

[7] https://pl.wikipedia.org/wiki/Regulator_PID

POLITECHNIKA RZESZOWSKA im. I. Łukasiewicza Rzeszów, 2024
Wydział Elektrotechniki i Informatyki

STRESZCZENIE PRACY DYPLOMOWEJ MAGISTERSKIEJ

PORÓWNANIE ALGORYTMÓW NAWIGACYJNYCH

Autor: Michał Bazan, nr albumu: EF-163881
Opiekun: dr inż. Dariusz Rzońca
Słowa kluczowe: (Algorytmy nawigacyjne, Robot mobilny, Inżynieria, ASPICE, Machine Learning)

Praca koncentruje się na badaniu wybranych algorytmów nawigacyjnych oraz metod optymalizacji nastaw regulatorów PID z wykorzystaniem zbudowanego robota mobilnego, który został skonstruowany zgodnie z procedurami ASPICE. Celem jest ocena dokładności, szybkości wyznaczania trasy oraz ogólnej wydajności tych algorytmów. Analiza wyników pozwoli wyciągnąć wnioski dotyczące skuteczności i efektywności badanych algorytmów nawigacyjnych. Dzięki zastosowaniu standardów ASPICE zapewniona została wysoka jakość procesu budowy robota, co umożliwia rzetelne i wiarygodne badania nad jego funkcjonalnością i algorytmami nawigacyjnymi.

RZESZOW UNIVERSITY OF TECHNOLOGY Rzeszow, 2024
Faculty of Electrical and Computer Engineering

MSC THESIS ABSTRACT

COMPARISON OF NAVIGATION ALGORITHMS

Author: Michał Bazan, album number: EF-163881
Supervisor: Dariusz Rzońca, dr. engineer
Key words: (Navigation algorithms, Mobile robot, Engineering, ASPICE, Machine Learning)

The work focuses on the study of selected navigation algorithms and methods for optimising PID controller settings using a built mobile robot that has been constructed according to ASPICE procedures. The aim is to evaluate the accuracy, routing speed and overall performance of these algorithms. Analysis of the results will allow conclusions to be drawn regarding the effectiveness and efficiency of the navigation algorithms studied. Through the use of ASPICE standards, the high quality of the robot construction process is ensured, enabling reliable and credible research into its functionality and navigation algorithms.