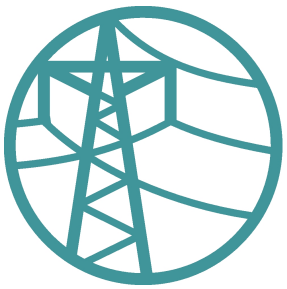


WYDZIAŁ
ELEKTROTECHNIKI
I INFORMATYKI
POLITECHNIKI RZESZOWSKIEJ

Michał Bazan

Porównanie wybranych algorytmów nawigacyjnych

PRACA MAGISTERSKA

Opiekun pracy:
dr inż. Dariusz Rzońca

Rzeszów, 2024

Spis treści

1. Wstęp **7**

2. Inżynierska część projektu **8**

2.1. Budowa robota 8

2.1.1. Wymagania sprzętowe 8

2.1.2. Schemat elektryczny 10

2.1.3. Testy 11

2.1.4. Konstrukcja mechaniczna 11

2.2. System operacyjny robota 13

2.2.1. Wymagania dotyczące oprogramowania 13

2.2.2. Implementacja 14

2.2.3. Obsługiwane polecenia 19

2.2.4. Testy oprogramowania 20

3. Wprowadzenie do omawianych algorytmów **22**

3.1. Algorytm PID 22

3.2. Algorytm genetyczny 24

3.3. Algorytmy nawigacyjne 26

3.3.1. Algorytmy statyczne 26

3.3.2. Algorytmy dynamiczne 27

4. Badania **29**

4.1. Optymalizacja nastaw regulatora PID prędkości obrotowej 29

4.1.1. Stanowisko pomiarowe 29

4.1.2. Identyfikacja obiektu 29

4.1.3. Wyznaczenie nastaw metodą klasyczną 32

4.1.4. Algorytm genetyczny 32

4.1.5. Porównanie otrzymanych wyników 38

4.2. Porównanie statycznych algorytmów nawigacyjnych w terenie bez przeszkód 41

4.2.1. Przebieg eksperymentu 41

4.2.2. Porównanie otrzymanych wyników 43

4.3. Porównanie statycznych algorytmów nawigacyjnych w terenie z przeszkodami 46

4.3.1. Przebieg eksperymentu 46
4.3.2. Porównanie otrzymanych wyników 48
4.4. Porównanie dynamicznych algorytmów nawigacyjnych w terenie z przeszkodami 50
4.4.1. Przebieg eksperymentu 50
4.4.2. Porównanie otrzymanych wyników 51
5. Podsumowanie i wnioski końcowe 53
5.1. Optymalizacja nastaw regulatora PID 53
5.2. Porównanie statycznych algorytmów nawigacyjnych w terenie bez przeszkód 53
5.3. Porównanie statycznych algorytmów nawigacyjnych w terenie z przeszkodami 54
5.4. Porównanie dynamicznych algorytmów nawigacyjnych w terenie z przeszkodami 54
5.5. Wkład własny 54
Literatura 55

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,

d) GA (eng. Genetic Algorithm) - algorytm genetyczny.

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 [6] optymalizacji regulatorów PID oraz wybranych algorytmów pod kątem kryteriów takich jak optymalizacja długości trasy i 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 sieciowy,

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

Przy realizacji części inżynierskiej zastosowano procedury ASPICE [8], co zapewniło wysoką jakość procesu budowy robota oraz implementacji oprogramowania. Takie podejście pozwoliło na przeprowadzenie rzetelnych badań i wyciągnięcie wiarygodnych wniosków.

W następnych rozdziałach pracy przedstawiono szczegółowy opis inżynierskiej części projektu oraz wyniki przeprowadzonych badań. W rozdziale poświęconym inżynierskiej części projektu omówiono budowę robota oraz jego system operacyjny. Komponenty te 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 porównanie 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 dwóch kluczowych elementów projektu, które skupiały się na budowie fizycznej robota oraz implementacji oprogramowania w języku C++ z uwzględnieniem unit testów. 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, implementacji elektroniki sterującej oraz budowy konstrukcji mechanicznej. 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.

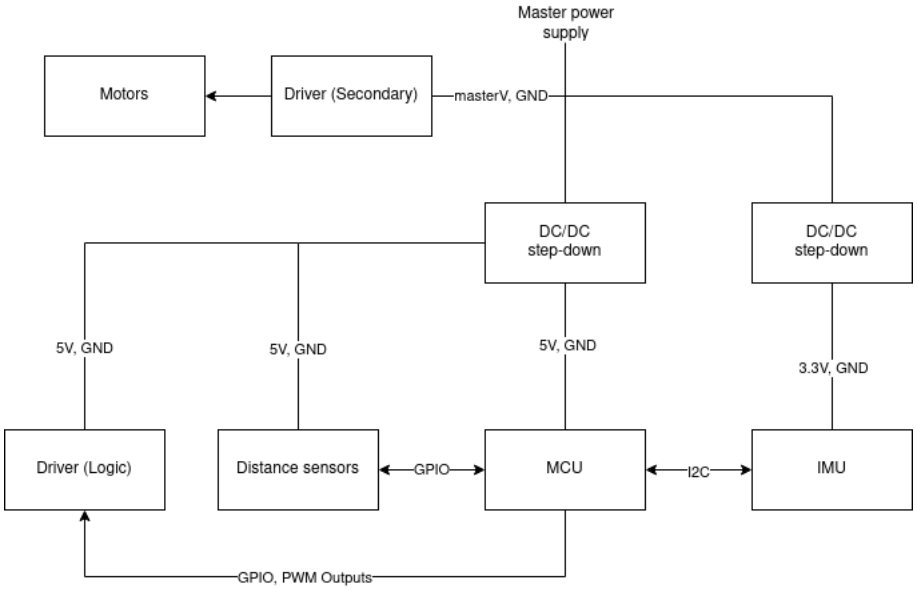
Tabela 2.1. Wymagania wysokiego poziomu HWE1

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 wymagań wysokiego poziomu ukazanych w tabeli 2.1, sformułowano 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.

Tabela 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 co najmniej 200 cm.
HWE_2_070	Czujniki odległości powinny udostępniać interfejs komunikacyjny kompatybilny z interfejsami mikrokontrolera.



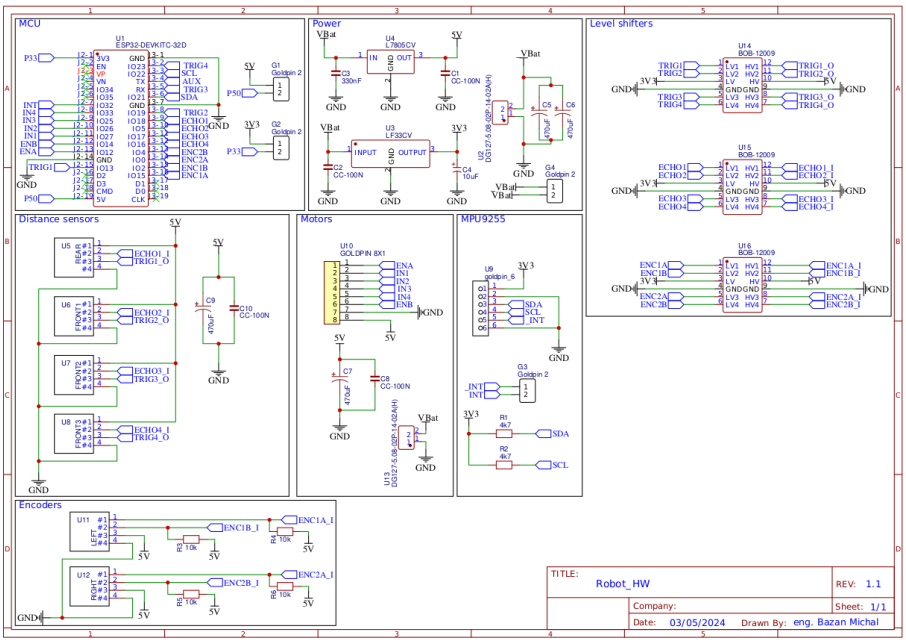
Rys. 2.1. Schemat blokowy sprzętu

Rysunek 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.

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.



Rys. 2.2. Schemat elektryczny

Rysunek 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,

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.

2.1.4. Konstrukcja mechaniczna

Konstrukcja mechaniczna robota (rysunek 2.4) została zaprojektowana jako trzykołowy pojazd, gdzie zastosowano dwa koła sterowalne oraz jedno koło niesterowalne, pełniące funkcję podpory konstrukcji. Komponenty elektroniczne zostały solidnie zamocowane na drewnianej płycie montażowej. Położenie każdego elementu zostało starannie zaplanowane w celu maksymalnego wykorzystania dostępnej przestrzeni. Kluczowym aspektem było minimalizowanie odległości między poszczególnymi blokami, co pozwoliło uniknąć problemów z połączeniami i zapewniło efektywną integrację bloków funkcjonalnych robota. Projekt konstrukcji uwzględniał również ergonomiczne rozmieszczenie elementów w celu ułatwienia dostępu do poszczególnych części w przypadku konserwacji i napraw. Dzięki szczegółowemu planowaniu, robot nie tylko spełniał wymagania funkcjonalne, ale również charakteryzował się kompaktowym i przejrzystym układem komponentów, co zwiększało jego niezawodność i łatwość obsługi. Rysunek 2.3 ukazuje rozmieszczenie bloków funkcjonalnych robota:

a) czujnik odległości tył,

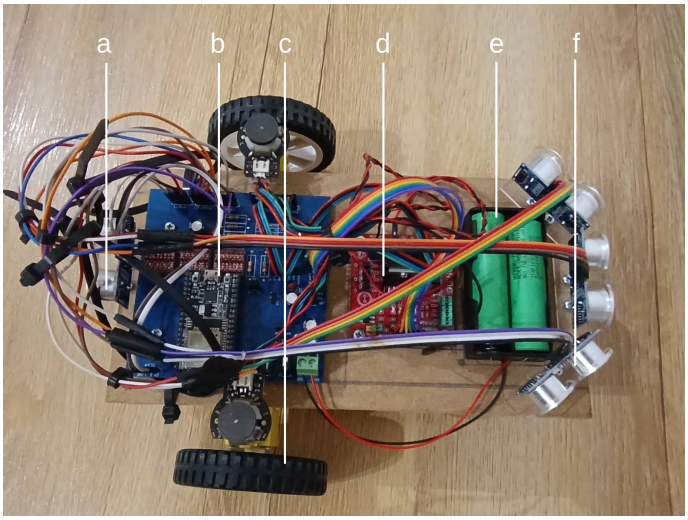
b) płytka z mikrokontrolerem odpowiedzialna za komunikację i sterowanie blokami,

c) silniki prądu stałego z enkoderami,

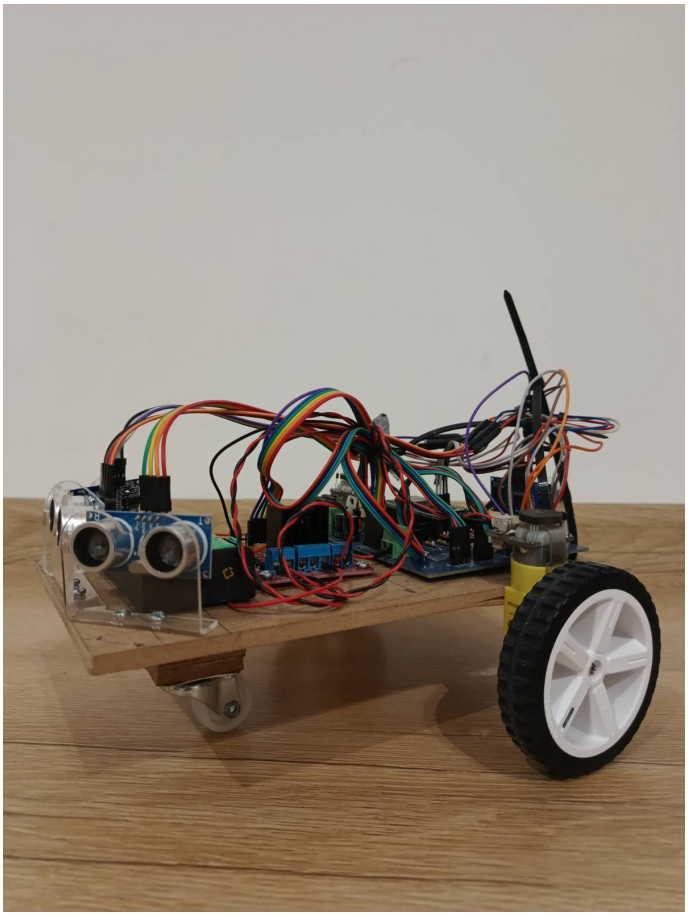
d) dwukanałowy sterownik silników prądu stałego,

e) akumulatory zasilające elektronikę i silniki,

f) czujniki odległości przód.



Rys. 2.3. Rozmieszczenie bloków



Rys. 2.4. Wykonany robot mobilny

2.2. System operacyjny robota

Niniejsza sekcja skupia się na implementacji oprogramowania w języku C++[3], 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 przez interfejs sieciowy.

2.2.1. Wymagania dotyczące oprogramowania

Na podstawie wcześniej omówionych wymagań sprzętowych oraz celów tego projektu, sformułowano następujące wymagania wysokiego poziomu (tabela 2.3). Wymagania te definiują kluczowe funkcjonalności, jakie powinno posiadać oprogramowanie, aby zapewnić niezawodność działania robota.

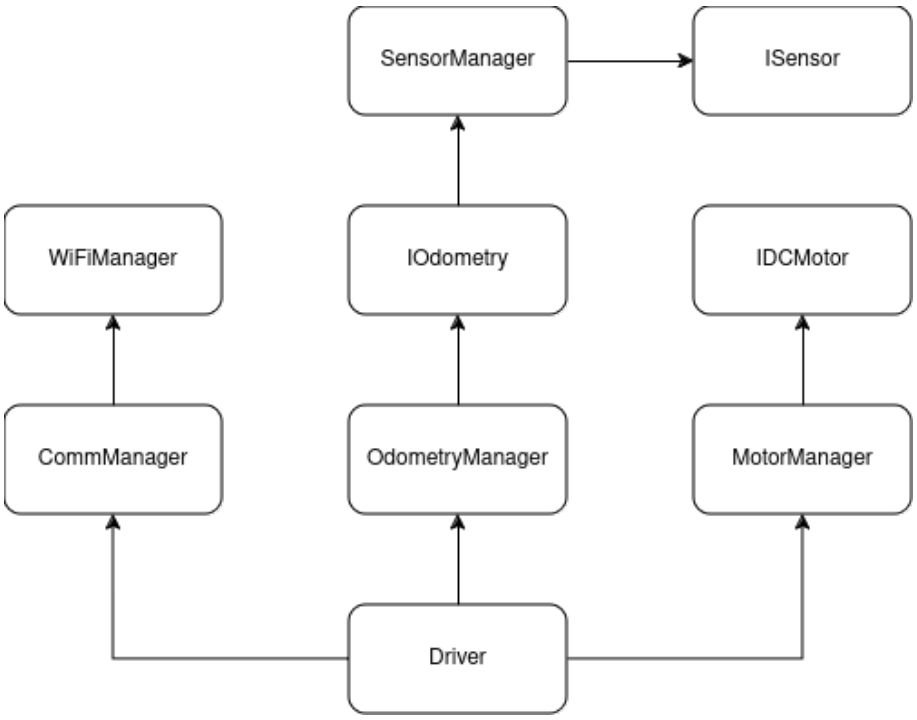
Tabela 2.3. Wymagania wysokiego poziomu SWE2

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 (tabela 2.4). Te szczegółowe wymagania określają specyficzne zadania i interfejsy, które muszą być zaimplementowane, aby spełnić założenia projektowe.

Tabela 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.



Rys. 2.5. Architektura oprogramowania

Na rysunku 2.5 przedstawiono architekturę systemu operacyjnego robota, która została zaprojektowana w oparciu o wcześniej zdefiniowane wymagania niskiego poziomu (tabela 2.4). 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 zarządzają różnymi aspektami operacyjnymi, od odczytu danych z czujników po sterowanie silnikami. Ważnym ele-

mentem implementacji jest również prosty szablon regulatora PID oraz stos oparty na *std::array*[9].

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 enumerator na liczby całkowite, co umożliwia łatwiejszy dostęp do konkretnego urządzenia. Konstruktor inicjuje obiekt tablicą uchwytów reprezentujących urządzenia, a *init* inicjuje sensory. Metoda *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* (listing 2.3) zarządza różnymi algorytmami odometrii robota i aktualizuje jego pozycję na podstawie danych z sensorów. Enumerator *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* (listing 2.4) 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* (listing 2.5) 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 efektywny podział czasu procesora.

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 przedstawione zostały tematy publikowane przez robota wraz z opisem i typowymi danymi przesyłanymi przez te kanały takie jak odpowiedzi na komendy i wiadomości służące do debugowania. 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.

Tabela 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.

Tabela 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 ’cmd not supported’ w przypadku przesłania błędnej komendy.

Tabela 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

Niniejsza sekcja skupia się na opisie przeprowadzonych testów weryfikujących zgodność oprogramowania z wymaganiami określonymi w specyfikacji SWE3.

Tabela 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

Tabela 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 'robot initialized' na 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* [5] z interwałem 50 ms.

Testy oznaczone jako *SQT_02* 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. Wprowadzenie do omawianych algorytmów

3.1. Algorytm PID

Algorytm PID [10] 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 zbyt 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 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 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

Algorytm genetyczny jest jednym z najpopularniejszych algorytmów ewolucyjnych, które są inspirowane procesami ewolucyjnymi w naturze. Główną ideą algorytmu genetycznego jest symulacja mechanizmów dziedziczenia, mutacji, selekcji naturalnej i krzyżowania genetycznego w celu rozwiązania problemów optymalizacyjnych. Podstawowe kroki działania algorytmu genetycznego obejmują:

a) inicjalizacja populacji - tworzenie początkowej populacji rozwiązań, zazwyczaj generowanych losowo,

b) selekcja - wybór jednostek najlepiej przystosowanych (najlepszych rozwiązań) do przetrwania i reprodukcji. Istnieje wiele różnych metod selekcji, takich jak ruletkowa, turniejowa, rankingowa itp.,

c) krzyżowanie - tworzenie nowych rozwiązań poprzez wymianę fragmentów między dwoma rodzicami. Jest to symulacja procesu krzyżowania genetycznego w naturze,

d) mutacja - losowa zmiana pewnych cech wybranych jednostek w celu wprowadzenia różnorodności genetycznej w populacji,

e) ocena przystosowania - określenie jakości każdego rozwiązania w populacji na podstawie funkcji oceny (funkcji przystosowania), która zazwyczaj opisuje, jak dobrze dany osobnik spełnia postawione wymagania,

f) zakończenie algorytmu - algorytm genetyczny zwykle kończy działanie po osiągnięciu określonego warunku stopu. Przykładowymi warunkami stopu mogą być przekroczenie założonej ilości generacji czy osiągnięcie rozwiązania optymalnego lokalnie lub globalnie.

Algorytmy genetyczne znajdują zastosowanie w różnych dziedzinach, takich jak optymalizacja kombinatoryczna, uczenie maszynowe, projektowanie sieci neuronowych, inżynieria ewolucyjna i wiele innych. Ich siła tkwi w zdolności do eksploracji przestrzeni rozwiązań w poszukiwaniu optymalnych lub zbliżonych do optymalnych wyników, nawet w przypadku problemów o dużym stopniu złożoności i wielowymiarowości.

Najważniejszymi parametrami algorytmu genetycznego są:

a) rozmiar populacji - określa liczbę osobników w każdym pokoleniu. Wartość tego parametru wpływa na zróżnicowanie genetyczne populacji oraz szybkość działania algorytmu. Zbyt mała populacja może prowadzić do zbyt wąskiego przeszukiwania przestrzeni rozwiązań, podczas gdy zbyt duża populacja może prowadzić do długiego czasu obliczeń,

b) liczba generacji - określa, ile pokoleń zostanie maksymalnie wygenerowanych przez algorytm genetyczny przed zakończeniem działania,

c) funkcja oceny - przypisuje wartość przystosowania każdemu osobnikowi w populacji, opisując, jak dobrze dany osobnik rozwiązuje problem optymalizacyjny. Celem algorytmu jest maksymalizacja lub minimalizacja tej funkcji zależnie od charakteru rozwiązywanego problemu,

d) metoda selekcji - selekcja wybiera osobniki z populacji, które zostaną wykorzystane do reprodukcji w kolejnym pokoleniu. Popularne metody selekcji to ruletkowa, turniejowa, rankingowa i proporcjonalna,

e) operator krzyżowania - krzyżowanie jest operacją genetyczną, która łączy cechy dwóch rodziców, aby stworzyć potomstwo. Istnieje wiele różnych metod krzyżowania, takich jak jednopunktowe krzyżowanie, dwupunktowe krzyżowanie, jednorodne krzyżowanie, krzyżowanie arithmetic blend itp.,

f) operator mutacji - mutacja wprowadza losowe zmiany w genotypie osobników w populacji aby zwiększyć różnorodność genetyczną i zapobiec zatrzymaniu się w lokalnych optimum,

g) prawdopodobieństwo mutacji i krzyżowania - określają one prawdopodobieństwo wystąpienia operacji krzyżowania i mutacji dla każdego osobnika w populacji. Zbyt wysokie prawdopodobieństwo może prowadzić do zbyt szybkiego zbiegania się algorytmu do lokalnego optimum, podczas gdy zbyt niskie może prowadzić do braku różnorodności w populacji.

3.3. Algorytmy nawigacyjne

Algorytmy nawigacyjne są kluczowym elementem systemów robotycznych, nawigacji GPS, gier komputerowych oraz wszelkich innych aplikacji wymagających efektywnego wyznaczania tras. Działają one na zasadzie przeszukiwania grafów, gdzie węzły reprezentują punkty a krawędzie reprezentują ścieżki między tymi punktami. W zależności od charakterystyki środowiska i wymagań aplikacji można wyróżnić dwa główne rodzaje algorytmów nawigacyjnych: statyczne i dynamiczne.

3.3.1. Algorytmy statyczne

Algorytmy statyczne operują w środowiskach, które nie ulegają zmianom w trakcie działania algorytmu. Oznacza to, że raz zbudowany graf nie zmienia swojej struktury a wszystkie wagi krawędzi pozostają stałe. Algorytmy te są bardzo efektywne w stabilnych środowiskach, gdzie trasy można wyznaczyć raz i wykorzystać je wielokrotnie bez konieczności przeliczania. Przykładami takich algorytmów są Dijkstra i A*.

Algorytm Dijkstry [11] jest jednym z najstarszych i najbardziej znanych algorytmów nawigacyjnych. Został zaproponowany przez E. W. Dijkstrę w 1956 roku. Jest to algorytm typu 'najpierw najlepszy' (ang. best-first), który znajduje najkrótszą ścieżkę z jednego węzła startowego do wszystkich innych węzłów w grafie z nieujemnymi wagami krawędzi.

Działanie algorytmu składa się z następujących kroków:

a) inicjalizacja - ustawienie odległości do węzła startowego na 0, a do wszystkich innych węzłów na nieskończoność. Wszystkie węzły są oznaczone jako nieodwiedzone,

b) wybór nieodwiedzonego węzła o najmniejszej znanej odległości (na początku jest to węzeł startowy),

c) aktualizacja odległości - dla wybranego węzła, zaktualizowanie odległości do jego sąsiadów, jeśli nowo obliczona odległość jest mniejsza niż aktualnie znana,

d) oznaczenie wybranego węzła jako odwiedzonego,

e) powtórzenie kroków b-d, aż wszystkie węzły zostaną odwiedzone.

Algorytm A* [12] jest rozszerzeniem algorytmu Dijkstry, który wykorzystuje heurystykę w celu poprawienia efektywności wyszukiwania. Został opracowany przez Petera Hart'a, Nilsa Nilssona i Bertrama Raphaela w 1968 roku. A* jest algorytmem typu 'najpierw najlepszy', który łączy zalety wyszukiwania zachłannego i dynamicznego programowania.

Działanie algorytmu składa się z następujących kroków:

a) inicjalizacja - ustawienie odległości do węzła startowego na 0 i obliczenie wartości heurystycznej dla wszystkich węzłów (przykładowo, odległość Manhattan lub euklidesowa do celu),

b) wybór węzła o najmniejszej wartości funkcji f(x) = g(x) + h(x), gdzie g(x) jest koszt dotarcia do węzła x, a h(x) jest oszacowany koszt dotarcia z węzła x do celu,

c) aktualizacja odległości do sąsiadów wybranego węzła i ich wartości funkcji f,

d) oznaczenie wybranego węzła jako odwiedzonego,

e) powtórzenie kroków b-d, aż węzeł celowy zostanie odwiedzony.

3.3.2. Algorytmy dynamiczne

Algorytmy dynamiczne są zaprojektowane do działania w zmieniających się środowiskach. Mogą one dostosowywać swoje obliczenia w odpowiedzi na zmiany w grafie takie jak zmieniające się wagi krawędzi lub dodawanie/odejmowanie węzłów. Algorytmy te

są bardziej złożone, ale niezbędne w sytuacjach, gdzie środowisko jest nieprzewidywalne lub zmienia się w czasie rzeczywistym. Przykładami algorytmów dynamicznych są D* i D* Lite.

Algorytm D* (Dynamic A*) [13] został zaproponowany przez Anthony'ego Stenz w 1994 roku. Jest to algorytm dynamicznego przeszukiwania grafów, który jest szczególnie użyteczny w środowiskach o zmieniających się warunkach. D* pozwala na aktualizację już obliczonych tras w odpowiedzi na zmiany w grafie.

Działanie algorytmu składa się z następujących kroków:

a) inicjalizacja - wyznaczenie najkrótszej ścieżki do celu przy użyciu standardowych metod (np. Dijkstry),

b) replanowanie - w przypadku zmiany w grafie (np. pojawienie się przeszkody), algorytm replanuje trasę od miejsca zmiany do celu, wykorzystując istniejące informacje i aktualizując tylko zmienione fragmenty grafu,

c) aktualizacja kosztów - algorytm modyfikuje koszty krawędzi i dostosowuje trasę w odpowiedzi na zmiany,

d) kontynuacja - powtarzanie procesu replanowania w przypadku dalszych zmian, aż do osiągnięcia celu.

D* Lite [14] jest uproszczoną i zoptymalizowaną wersją algorytmu D* opracowaną przez Svena Koeniga i Maxa Likhacheva w 2002 roku. D* Lite zachowuje główne zalety D*, ale jest łatwiejszy do implementacji i bardziej efektywny obliczeniowo.

Działanie algorytmu składa się z następujących kroków:

a) inicjalizacja - podobnie jak w D*, inicjalizowanie grafu i wyznaczanie początkowej trasy do celu,

b) replanowanie - w przypadku zmiany w grafie, algorytm replanuje trasę, ale korzysta z bardziej zoptymalizowanego sposobu aktualizacji węzłów i krawędzi,

c) aktualizacja kosztów - podobnie jak w D*, aktualizacja kosztów krawędzi i dostosowanie trasy, ale z wykorzystaniem efektywniejszych struktur danych i operacji,

d) kontynuacja - powtarzanie procesu replanowania w przypadku dalszych zmian, aż do osiągnięcia celu.

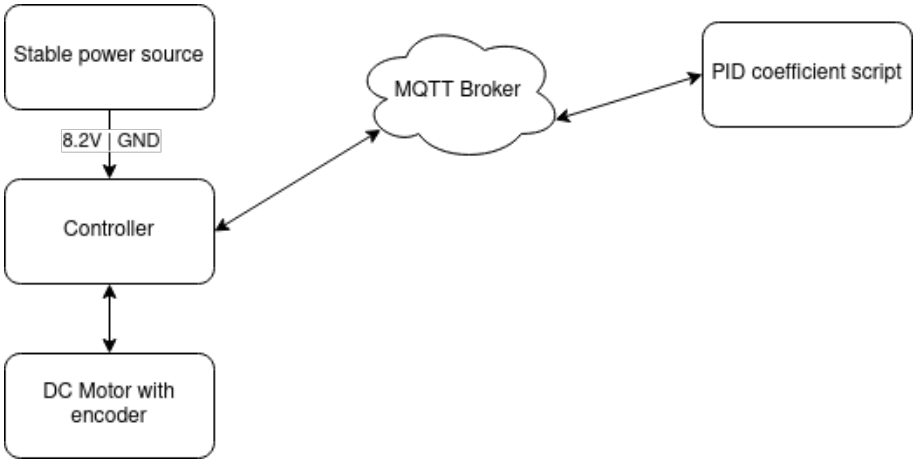
4. Badania

4.1. Optymalizacja nastaw regulatora PID prędkości obrotowej

Sekcja skupia się na opisie eksperymentu, który miał na celu porównanie jakości regulacji prędkości obrotowej silnika poprzez wyznaczenie nastaw metodą Zieglera-Nicholsa [15] i algorytmem genetycznym.

4.1.1. Stanowisko pomiarowe

Do przeprowadzenia eksperymentu wykorzystano stanowisko składające się z silnika prądu stałego, enkodera do pomiaru prędkości obrotowej oraz kontrolera. Stanowisko umożliwiało kontrolowanie prędkości obrotowej silnika oraz pomiar jej wartości w czasie rzeczywistym.

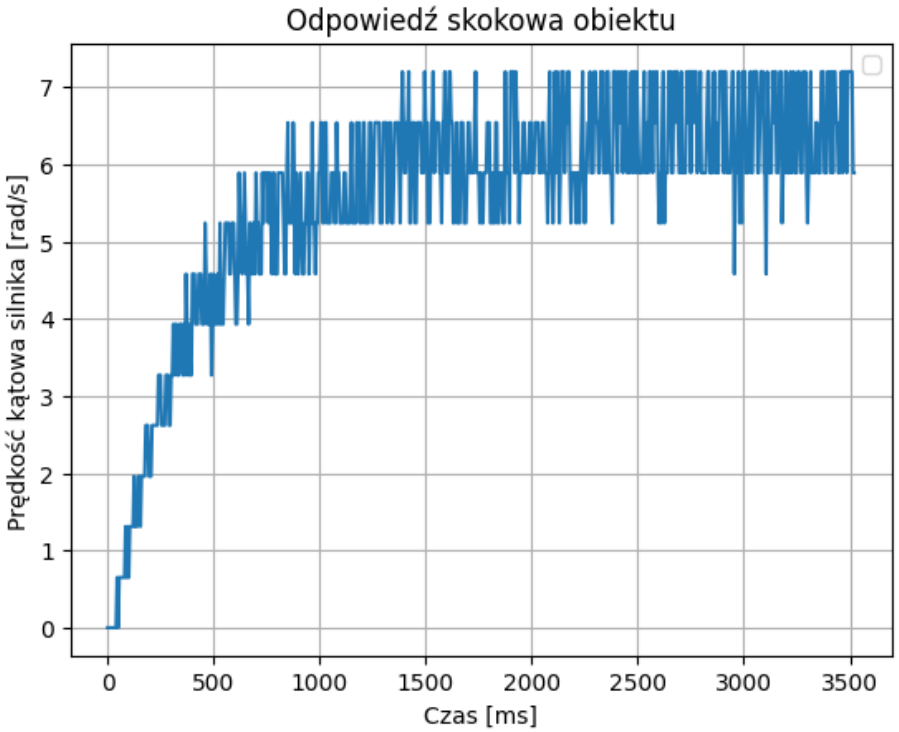


Rys. 4.1. Schemat blokowy stanowiska pomiarowego

4.1.2. Identyfikacja obiektu

Wykorzystując interfejs sieciowy kontrolera, przeprowadzono eksperyment, podczas którego monitorowano odpowiedź obiektu na skokowy sygnał sterujący. W trakcie tego eksperymentu rejestrowano zmiany prędkości obrotowej silnika w odpowiedzi na zmiany sygnału sterującego (rysunek 4.2). Dane te zostały następnie poddane analizie w celu wyznaczenia charakterystyki dynamicznej obiektu.

Aby dopasować odpowiedź obiektu do modelu transmitancji pierwszego rzędu, zastosowano metodę aproksymacji [7]. Polegała ona na dopasowaniu odpowiedniego modelu matematycznego do zebranych danych eksperymentalnych. W tym przypadku, model transmitancji pierwszego rzędu pozwolił na odpowiednie odwzorowanie zacho-



Rys. 4.2. Odpowiedź obiektu

wania obiektu w kontekście zmian sygnału sterującego. Dzięki temu można było przewidywać reakcję obiektu na różne zmiany sterowania.

Listing 4.1. Kod aproksymujący odpowiedź

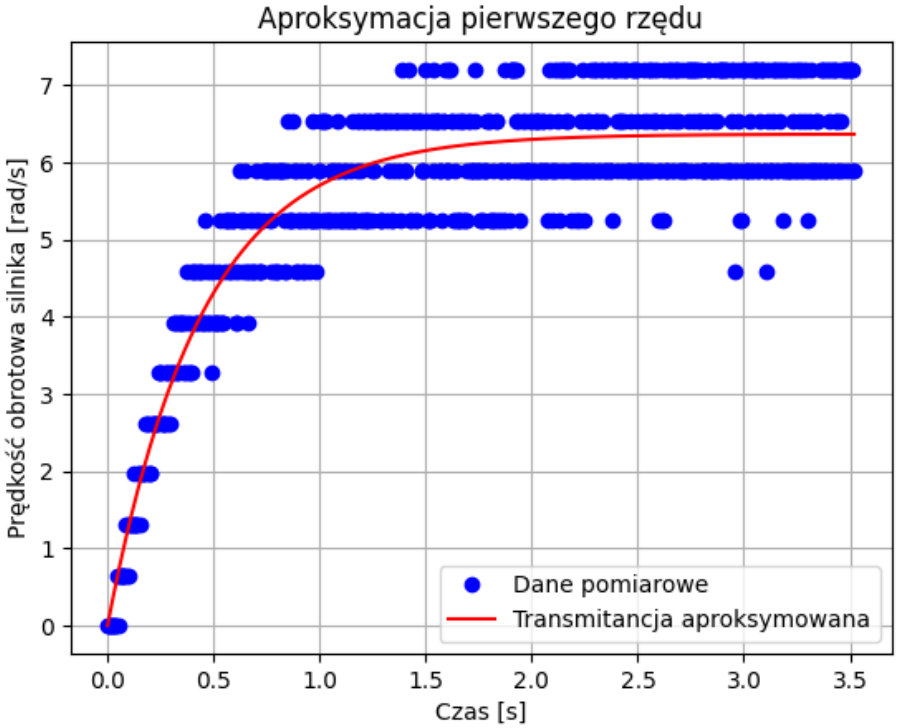
```
import numpy as np
from scipy.optimize import curve_fit

time_step=5/1000

t = np.arange(0, len(actuals) * time_step, time_step)
odpowiedz_skokowa=actuals

def first_order_transfer_function(t, K, T):
    return K * (1 - np.exp(-t / T))

popt, pcov = curve_fit(first_order_transfer_function, t, odpowiedz_skokowa)
K, T=popt

print("Wzmocnienie (K):", K)
print("Stala czasowa (T):", T)
```

Kod przedstawiony na listingu 4.1 służy do analizy danych eksperymentalnych reprezentujących odpowiedź skokową obiektu oraz aproksymowania tych danych do modelu transmitancji pierwszego rzędu. Najpierw tworzone są dane czasowe t, a następnie dane odpowiedzi skokowej *odpowiedz_skokowa*. Definiowana jest funkcja *first_order_transfer_function*, która odpowiada transmitancji pierwszego rzędu. Następnie korzystając z funkcji *curve_fit* z biblioteki *scipy.optimize*, dopasowywany jest

model transmitancji pierwszego rzędu do danych odpowiedzi skokowej. Funkcja, jako wynik działania, zwraca wartość wzmocnienia K oraz stałą czasową T.

Rysunek 4.3 ukazuje aproksymację modelu transmitancji pierwszego rzędu do danych pomiarowych. Wyznaczoną transmitancję obiektu opisuje wzór 4.1.

$$G(s) = \frac{6.37}{1 + 0.44s} \tag{4.1}$$



Rys. 4.3. Aproksymacja transmitancji

4.1.3. Wyznaczenie nastaw metodą klasyczną

Kod umieszczony na listingu 4.2 wykorzystuje bibliotekę *numpy* do obliczenia nastaw regulatora PID na podstawie zadanych wartości wzmocnienia i czasu wyprzedzenia. Na początku definiuje wartości stałych K i T, a następnie wykorzystuje je do obliczenia współczynnika krytycznego (Ku) oraz okresu oscylacji (Tu). Następnie na podstawie Ku i Tu wyznacza nastawy PID.

Listing 4.2. Kod z obliczeniami i wyświetleniem wyników

```
import numpy as np

K = 6.365318519867311
T = 0.4425290896955204

Ku = 4 / (np.pi * K)

Tu = np.pi * T

Kp_PID = 0.20 * Ku
Ti_PID = 0.50 * Tu
Td_PID = 0.12 * Tu
```

4.1.4. Algorytm genetyczny

Optymalizacja nastaw regulatora PID jest kluczowym zagadnieniem w automatyce, które ma na celu poprawę wydajności systemów sterowania, w tym regulacji prędkości obrotowej silników. W kontekście fizycznego robota, możliwe jest utworzenie pętli uczącej, która będzie wprowadzać nowe współczynniki PID i testować je na rzeczywistym sprzęcie. Ogólny algorytm PID oraz komunikacja są obsługiwane przez oprogramowanie wgrane do mikrokontrolera robota.

Z punktu widzenia automatyki, silnik prądu stałego jest stosunkowo prostym obiektem sterowania, co sprawia, że wiele zestawów parametrów może być odpowiednich do automatycznej regulacji prędkości. W związku z tym zdecydowano się na użycie algorytmu genetycznego do wyboru optymalnego zestawu współczynników PID.

Skrypt w języku Python zawierający implementację algorytmu genetycznego będzie uruchomiony na lokalnym serwerze. Każdy osobnik w populacji będzie testowany na fizycznym robocie w celu sprawdzenia jego dopasowania (*fitness*). Algorytm będzie uruchamiany trzykrotnie z różnymi zestawami parametrów:

a) **Pierwszy eksperyment**

- Rozmiar populacji: 100
- Maksymalna liczba epok: 25
- Prawdopodobieństwo krzyżowania: 70%

- Liczba rodziców dla nowego osobnika: 10
- Prawdopodobieństwo mutacji: 15%

b) **Drugi eksperyment**

- Rozmiar populacji: 200
- Maksymalna liczba epok: 25
- Prawdopodobieństwo krzyżowania: 70%
- Liczba rodziców dla nowego osobnika: 15
- Prawdopodobieństwo mutacji: 15%

c) **Trzeci eksperyment**

- Rozmiar populacji: 300
- Maksymalna liczba epok: 25
- Prawdopodobieństwo krzyżowania: 70%
- Liczba rodziców dla nowego osobnika: 25
- Prawdopodobieństwo mutacji: 15%

Ocena jakości regulacji będzie dokonywana na podstawie maksymalnego błędu regulacji, który docelowo musi być mniejszy lub równy 2%. Algorytm zakończy swoje działanie po osiągnięciu maksymalnej liczby epok lub gdy zostanie spełniony warunek maksymalnego błędu regulacji.

Funkcja dopasowania, którą algorytm będzie maksymalizować, została określona jako odwrotność średniego błędu względnego (wzór 4.2) .

$$Fitness = \frac{1}{error} \tag{4.2}$$

gdzie: $error$ - wartość średniego błędu względnego

Listing 4.3. Konfiguracja MQTT

```
import paho.mqtt.client as mqtt
import json
import time

MQTT_BROKER = "mqtt-dashboard.com"
MQTT_PORT = 1883
MQTT_TOPIC_PID = "robot/set/pid"
MQTT_TOPIC_MOTORS = "robot/set/motors"
MQTT_TOPIC_LOG = "robot/pid/log"
MQTT_TOPIC_RESPONSE = "robot/request"

mqtt_client = mqtt.Client("PID Ctl")

logs = []
experiment_ongoing = False

def on_log_message(client, userdata, message):
    global experiment_ongoing
    log_data = json.loads(message.payload.decode())
    if len(logs) != 0:
        logs.append(log_data)
    elif len(logs) == 0 and log_data["id"] == 0:
        logs.append(log_data)

def send_mqtt_message(topic, message):
    mqtt_client.publish(topic, json.dumps(message))
```

Kod przedstawiony na listingu 4.3 nawiązuje połączenie z brokerem MQTT, który umożliwia komunikację pomiędzy serwerem a robotem. Kluczowe elementy konfiguracji obejmują określenie adresu brokera oraz portu. Następnie zdefiniowano tematy MQTT do wysyłania i odbierania wiadomości dotyczących ustawień PID, sterowania silnikami oraz logowania. Dodatkowo zdefiniowano funkcje pomocnicze, takie jak *send_mqtt_message* do wysyłania wiadomości oraz *on_log_message* do obsługi wiadomości logowania.

Ta sekcja przedstawia funkcję *run_experiment* (listing 4.4), która inicjalizuje i przeprowadza eksperymenty z nowymi nastawami PID na robocie. Kluczowe kroki obejmują ponowne połączenie z brokerem MQTT i subskrybowanie tematów logowania, wysłanie nowych wartości współczynników PID do robota, uruchomienie silników robota i monitorowanie ich przez określony czas oraz zbieranie danych logowania do późniejszej analizy.

Listing 4.4. Funkcja uruchamiająca eksperyment

```
def run_experiment(Kp, Ti, Td):
    global logs
    global mqtt_client
    global experiment_ongoing
    mqtt_client = mqtt.Client("PID ctl")
    mqtt_client.connect(MQTT_BROKER, MQTT_PORT)
    mqtt_client.subscribe(MQTT_TOPIC_LOG)
    mqtt_client.on_message = on_log_message

    mqtt_client.loop()
    logs = []
```

```
12
13      pid_settings = {"Kp": Kp, "Ti": Ti, "Td": Td}
14      send_mqtt_message(MQTT_TOPIC_PID, pid_settings)
15
16      motors_speed = {"left": 0, "right": 50}
17      send_mqtt_message(MQTT_TOPIC_MOTORS, motors_speed)
18      experiment_ongoing = True
19      start_time = time.time()
20      while time.time() - start_time < 2:
21          mqtt_client.loop()
22
23      motors_speed = {"left": 0, "right": 0}
24      send_mqtt_message(MQTT_TOPIC_MOTORS, motors_speed)
25
26      start_time = time.time()
27
28      while time.time() - start_time < 1:
29          mqtt_client.loop()
30
31      send_mqtt_message(MQTT_TOPIC_RESPONSE, "")
32      experiment_ongoing = False
```

Funkcja *calculate_errors* (listing 4.5) służy do obliczania błędów na podstawie danych logowania zebranych podczas eksperymentu. Kluczowe kroki obejmują obliczanie błędów absolutnych i względnych dla każdej próbki danych oraz obliczanie średniego błędu względnego, który jest używany jako miara jakości regulatora PID.

Listing 4.5. Funkcja obliczająca błędy

```
1  def calculate_errors(logs):
2      absolute_errors = []
3      relative_errors = []
4
5      for i in range(len(logs)):
6          measurement = logs[i]
7          absolute_error = abs(measurement['target'] - measurement['actual'])
8
9          if measurement['target'] != 0:
10             relative_error = (absolute_error / measurement['target']) * 100
11         else:
12             if measurement['actual'] == measurement['target']:
13                 relative_error = 0
14             else:
15                 relative_error = float(100)
16
17         absolute_errors.append(absolute_error)
18         relative_errors.append(relative_error)
19
20     mean_relative_error = (sum(relative_errors) / len(relative_errors)) if(len(
       relative_errors)) != 0 else 10000
21
22     if(mean_relative_error) == 0.0:
23         return 10000
24
25     return mean_relative_error
```

Funkcja *fitness_func* (listing 4.6) ocenia jakość każdego zestawu współczynników PID, wykonując eksperyment z tymi współczynnikami i obliczając wynikowy błąd względny. Wynik jest używany przez algorytm genetyczny do wyboru najlepszych rozwiązań. Kluczowe elementy obejmują przeprowadzenie eksperymentu z podanymi

współczynnikami PID, wyliczenie średniego błędu względnego oraz zwrócenie wartości funkcji dopasowania, która jest odwrotnością błędu względnego.

Listing 4.6. Funkcja dopasowania

```
def fitness_func(ga_instance, solution, solution_idx):
    global logs
    global log_file

    Kp, Ti, Td = solution
    run_experiment(Kp, Ti, Td)

    mean_relative_error = calculate_errors(logs)
    logs = []
    append_data_to_csv(log_file, [Kp, Ti, Td, mean_relative_error, ga_instance.
    generations_completed])
    print(mean_relative_error)

    fitness = 1 / (mean_relative_error + 0.001)
    return fitness
```

Funkcja *run_genetic_algorithm* (listing 4.7) inicjalizuje i uruchamia algorytm genetyczny z podanymi parametrami. Kluczowe kroki obejmują definicję problemu optymalizacyjnego poprzez określenie liczby genów (współczynników PID) oraz przestrzeni poszukiwań, inicjalizację algorytmu genetycznego z parametrami takimi jak liczba pokoleń, liczba rodziców, prawdopodobieństwo krzyżowania i mutacji oraz wykonanie algorytmu i zapisanie najlepszego rozwiązania.

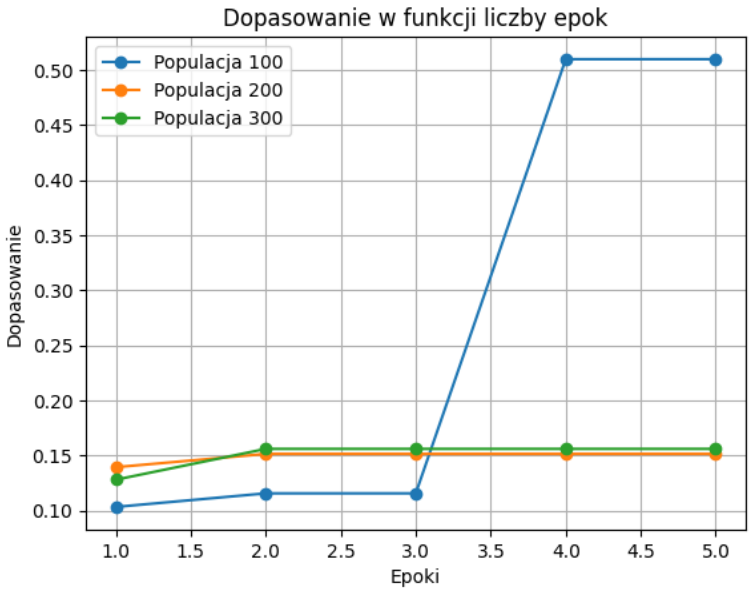
Listing 4.7. Uruchamianie algorytmu genetycznego

```
def run_genetic_algorithm(pop_size, max_epochs, crossover_prob, num_parents_mating,
    mutation_prob, experiment_num):
    num_genes = 3

    on_generation = lambda x: save_best_solution(x, experiment_num)

    ga_instance = pygad.GA(num_generations=max_epochs,
                           num_parents_mating=num_parents_mating,
                           sol_per_pop=pop_size,
                           num_genes=num_genes,
                           gene_space={'low': 0, 'high': 30},
                           crossover_probability=crossover_prob,
                           mutation_percent_genes='default',
                           mutation_probability=mutation_prob,
                           K_tournament=num_parents_mating,
                           fitness_func=fitness_func,
                           on_generation=on_generation,
                           mutation_type='scramble',
                           crossover_type="uniform",
                           parent_selection_type='tournament',
                           keep_parents=0,
                           gene_type=float,
                           random_mutation_min_val=10,
                           random_mutation_max_val=20
                           )

    ga_instance.run()

    best_solution = ga_instance.best_solution()
    return best_solution
```

W tej sekcji (listing 4.8) kod inicjalizuje i przeprowadza serię eksperymentów z różnymi zestawami parametrów algorytmu genetycznego. Kluczowe kroki obejmują zdefiniowanie zestawów parametrów dla trzech różnych eksperymentów oraz iterację przez każdy zestaw parametrów, uruchamianie algorytmu genetycznego dla każdego z nich i zapisanie najlepszych rozwiązań.

Listing 4.8. Pętla przeprowadzająca eksperymenty

```
experiments_params = [
    {"pop_size": 100, "max_epochs": 25, "crossover_prob": 0.70, "num_parents_mating"
    : 10, "mutation_prob": 0.15},
    {"pop_size": 200, "max_epochs": 25, "crossover_prob": 0.70, "num_parents_mating"
    : 15, "mutation_prob": 0.15},
    {"pop_size": 300, "max_epochs": 25, crossover_prob: 0.70, num_parents_mating:
    20, mutation_prob: 0.15}
]

for i, params in enumerate(experiments_params):
    print(f"Starting experiment {i+1} with params: {params}")
    best_solution = run_genetic_algorithm(
        pop_size=params["pop_size"],
        max_epochs=params["max_epochs"],
        crossover_prob=params["crossover_prob"],
        num_parents_mating=params["num_parents_mating"],
        mutation_prob=params["mutation_prob"],
        experiment_num=i+1
    )
    print(f"Best solution for experiment {i+1}: {best_solution}")
```



Rys. 4.4. Dopasowanie w funkcji czasu

Uruchomienie tak przygotowanego skryptu pozwoliło na uzyskanie trzech zestawów nastaw regulatora PID. Rysunek 4.4 ukazuje dopasowanie najlepszego osobnika danej populacji w funkcji czasu (epok). Biorąc pod uwagę kształty wykresów, można dojść do następującego wniosku: ze względu na prostotę obiektu, jakim jest silnik DC,

wiele różnych od siebie zestawów nastaw regulatora jest w stanie usatysfakcjonować stawiane przed nim wymagania, przez co algorytm genetyczny popada w stagnację po osiągnięciu lokalnego minimum.

4.1.5. Porównanie otrzymanych wyników

Podczas przeprowadzania eksperymentu otrzymano następujące nastawy regulatora PID:

a) **Ziegler-Nichols**

- Kp: 0.04
- Ti: 0.695
- Td: 0.166

b) **GA eksperyment 1**

- Kp: 6.48
- Ti: 2.93
- Td: 2.34

c) **GA eksperyment 2**

- Kp: 29.50
- Ti: 14.54
- Td: 1.37

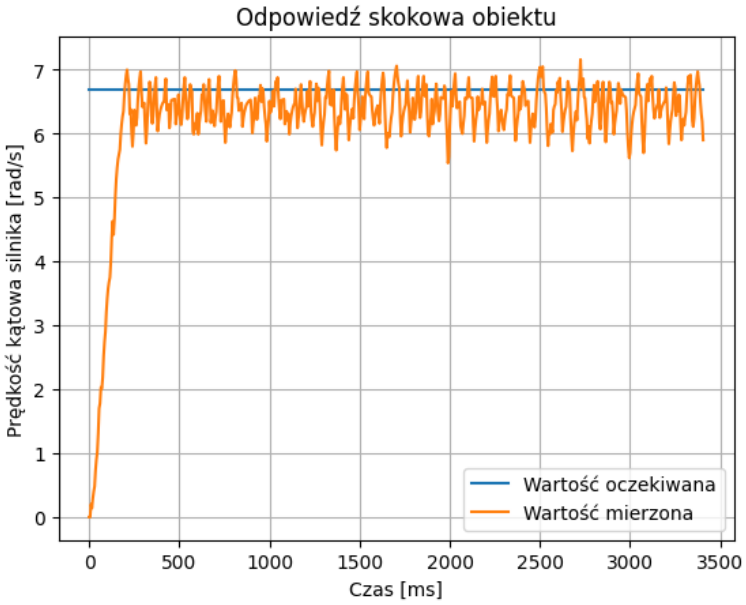
d) **GA eksperyment 3**

- Kp: 11.98
- Ti: 1.59
- Td: 1.34

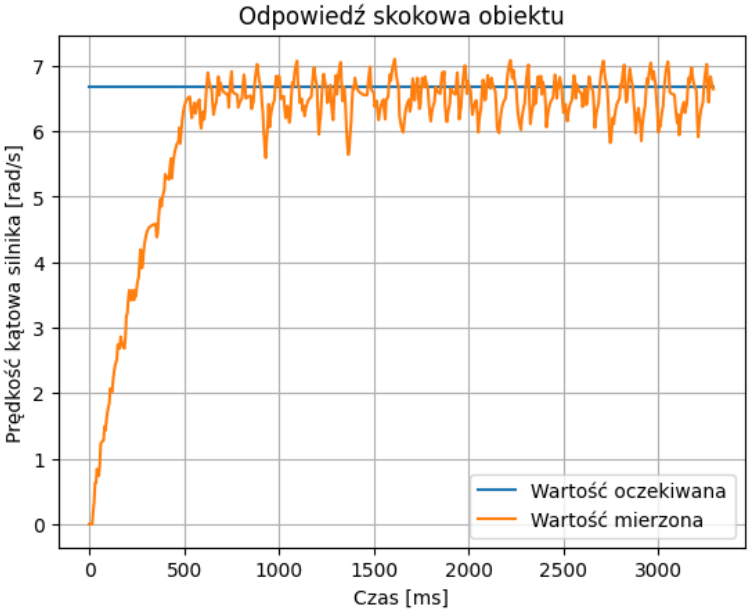
By porównać jakość wyznaczonych nastaw, należało przestroić regulator i zbadać odpowiedź obiektu na zmianę sygnału sterującego.

Tabela 4.1. Zestawienie wyników

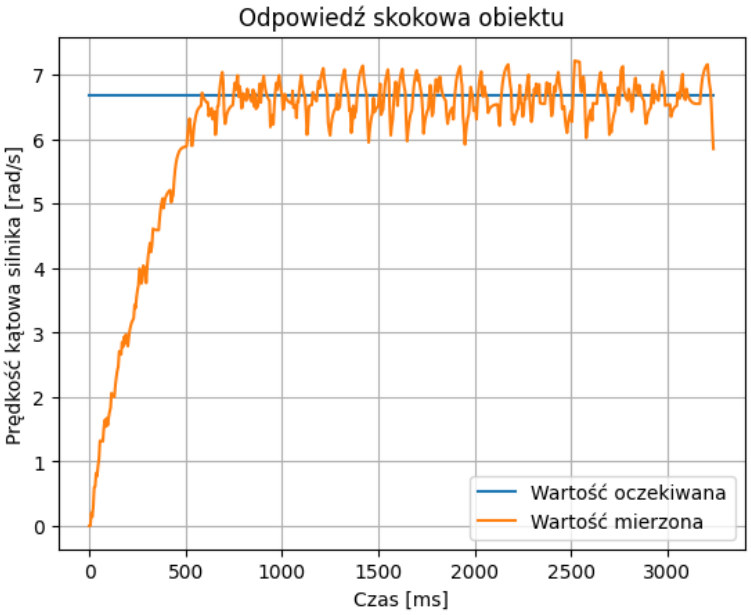
Metoda	Czas ustalania [ms]	Uwagi
Ziegler-Nichols	190	Niedoregulowanie
GA eksperyment 1	630	Niedoregulowanie
GA eksperyment 2	600	Satysfakcjonująca regulacja
GA eksperyment 3	600	Satysfakcjonująca regulacja



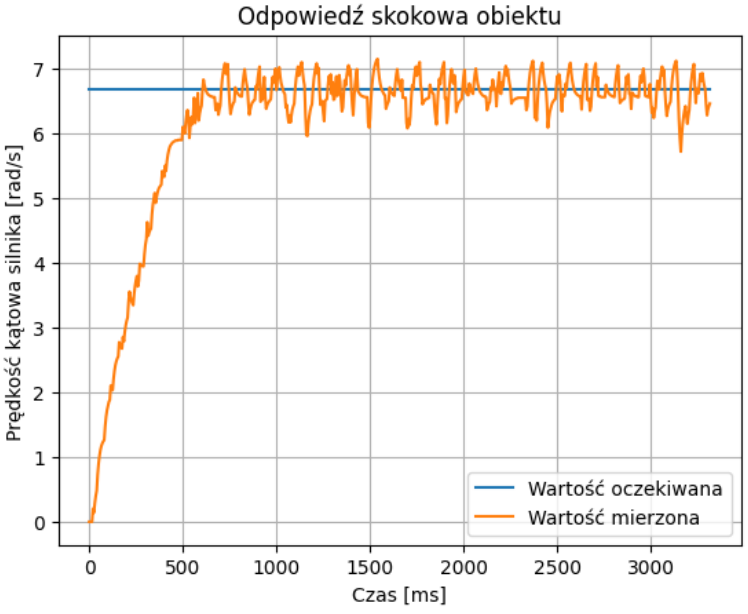
Rys. 4.5. Nastawy wyznaczone metodą Zieglera-Nicholsa



Rys. 4.6. Nastawy wyznaczone przez algorytm genetyczny (1)



Rys. 4.7. Nastawy wyznaczone przez algorytm genetyczny (2)



Rys. 4.8. Nastawy wyznaczone przez algorytm genetyczny (3)

Po sprawdzeniu odpowiedzi obiektu na wyznaczone nastawy, podjęto decyzję o wykorzystaniu wyniku trzeciego eksperymentu. Wybór jest uzasadniony kilkoma kluczowymi czynnikami. Przede wszystkim, w porównaniu z innymi metodami, nastawy te zapewniły satysfakcjonującą regulację systemu (rysunek 4.8). Czas ustalania wynosił 600 ms, co jest wynikiem porównywalnym z najlepszymi nastawami uzyskanymi w innych eksperymentach. Dodatkowo, podczas eksperymentu nastawy GA 3 wykazały najlepszy wynik *fitness*, co oznacza, że algorytm genetyczny skutecznie dostroił parametry regulatora w celu optymalnej regulacji systemu. W związku z tym, można uznać, że nastawy te są najkorzystniejsze pod względem jakości regulacji i efektywności działania systemu. Podczas testów z nastawami wyznaczonymi metodą Zieglera-Nicholsa (rysunek 4.5) zaobserwowano, że jedno z kół nie chciało wystartować, co świadczy o niedostatecznej regulacji. Oznacza to, że te nastawy nie były w stanie zapewnić właściwej kontroli nad całym systemem, co może prowadzić do niestabilności i niesatysfakcjonującej wydajności. Czas ustalania wynoszący 190 ms jest krótki, ale ze względu na problem z jednym z kół, regulacja nie była wystarczająco dobra.

4.2. Porównanie statycznych algorytmów nawigacyjnych w terenie bez przeszkód

Niniejsza sekcja skupia się na opisie eksperymentu, który miał na celu porównanie wydajności algorytmu A* z różnymi heurystykami odległości (Manhattan, Euklidesowa, Max) oraz algorytmu Dijkstry na różnych rozmiarach grafów. Celem było zrozumienie, który algorytm lepiej nadaje się do rozwiązywania problemów przeszukiwania grafów w kontekście optymalizacji czasu działania i długości trasy, jaką ma pokonać robot.

4.2.1. Przebieg eksperymentu

Mapa, po której przemieszczał się robot, to prostokąt o wymiarach 200x50 cm. Aby dokładnie ocenić efektywność algorytmów wyznaczania trasy, eksperyment uwzględnił kilka różnych rozmiarów map:

a) oprócz podstawowej mapy o wymiarach 200x50 cm, przygotowano również mapy o większych rozmiarach: 200x200 cm, 500x500 cm, 1000x1000 cm oraz 2000x2000 cm,

b) każda z tych map była reprezentowana jako graf, gdzie węzły odpowiadały punktom na mapie, a krawędzie określały możliwe połączenia między nimi.

Listing 4.9. Porównanie algorytmów

```
from graph_ops import load_graph_from_file
graph_dims = [(200, 50), (200, 200), (500, 500), (1000, 1000), (2000, 2000)]
graphs = {}

for dim in graph_dims:
    fname = f'graph_{dim[0]}_{dim[1]}.pkl'
    graphs[fname] = load_graph_from_file(fname)

start = (0, 0)
goal = (198, 40)

heuristics={
    "manhattan" : manhattan_distance,
    "euclidean" : euclidean_distance,
    "max" : max_distance
}

for k, v in heuristics.items():
    print(f"Funkcja obliczajaca odleglosc w algorytmie A*: {k}")
    for gname, graph in graphs.items():
        print(gname)
        path1 = a_star(graph, start, goal, v)
        path2 = dijkstra(graph, start, goal)
        print('\n')
```

Kod przedstawiony na listingu 4.9 ukazuje skrypt odpowiedzialny za zbadanie algorytmów przeszukujących grafy. Kod rozpoczyna się od wczytania wcześniej wygenerowanych grafów i zdefiniowania heruystyk wykorzystanych w eksperymencie:

a) Manhattan (wzór 4.3),

b) Euklidesowa (wzór 4.4),

c) Czebyszewa (wzór 4.5).

$$d_{\text{Manhattan}}(p,q) = |p_x - q_x| + |p_y - q_y| \tag{4.3}$$

$$d_{\text{Euklidesowa}}(p,q) = \sqrt{(p_x - q_x)^2 + (p_y - q_y)^2} \tag{4.4}$$

$$d_{\text{Czebyszewa}}(p,q) = \max(|p_x - q_x|, |p_y - q_y|) \tag{4.5}$$

gdzie: p, q – to punkty w dwuwymiarowej przestrzeni, gdzie każdy punkt jest reprezentowany przez parę współrzędnych (p_x, p_y) i (q_x, q_y), d – oznacza odległość pomiędzy punktem p i q

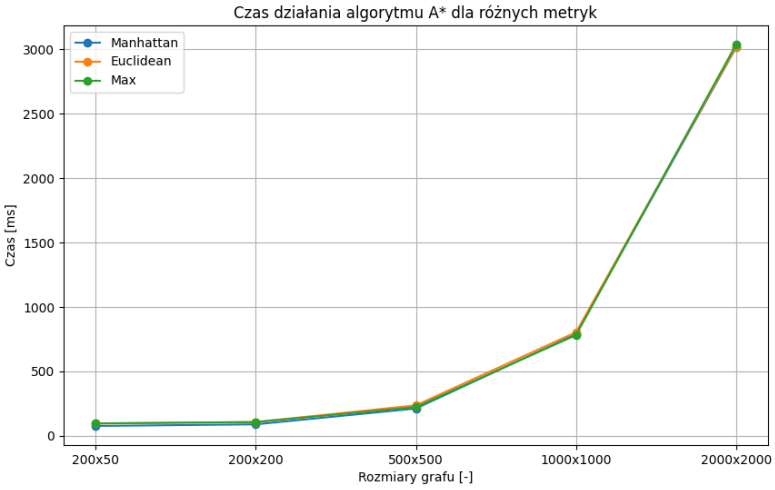
Kolejnym krokiem jest przejście przez wszystkie heurystyki oraz wczytane grafy w celu przeprowadzenia eksperymentu porównawczego. Skrypt, dla każdego grafu, wyznacza trasę z punktu startowego *start* do punktu docelowego *goal* z wykorzystaniem określonej heurystyki.

4.2.2. Porównanie otrzymanych wyników

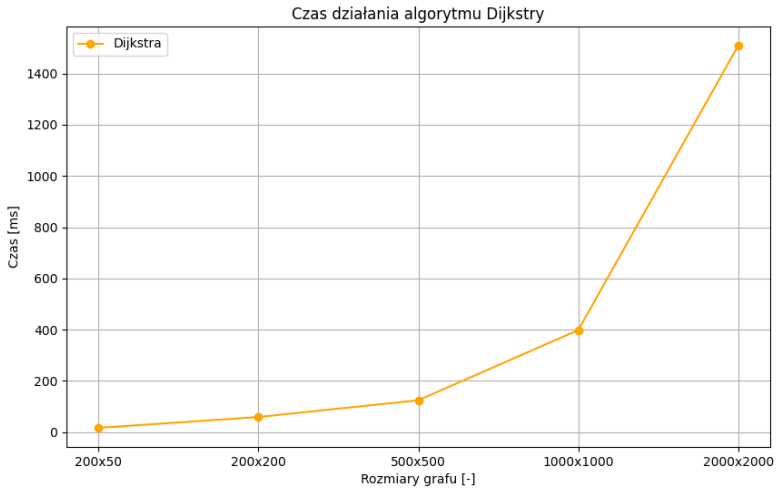
Na podstawie przedstawionych wyników eksperymentu porównującego algorytmy A* z różnymi heurystykami oraz algorytm Dijkstry (tabela 4.4), można wysunąć następujące wnioski:

a) algorytm Dijkstry wypadł lepiej pod względem czasu wykonania w porównaniu do algorytmu A* (rysunek 4.11). Osiągnął on znacznie krótsze czasy wykonania dla wszystkich rozmiarów badanych grafów. Zwykle A* powinien działać szybciej, ponieważ używa heurystyki do prowadzenia wyszukiwania, co teoretycznie powinno ograniczyć liczbę odwiedzanych węzłów.

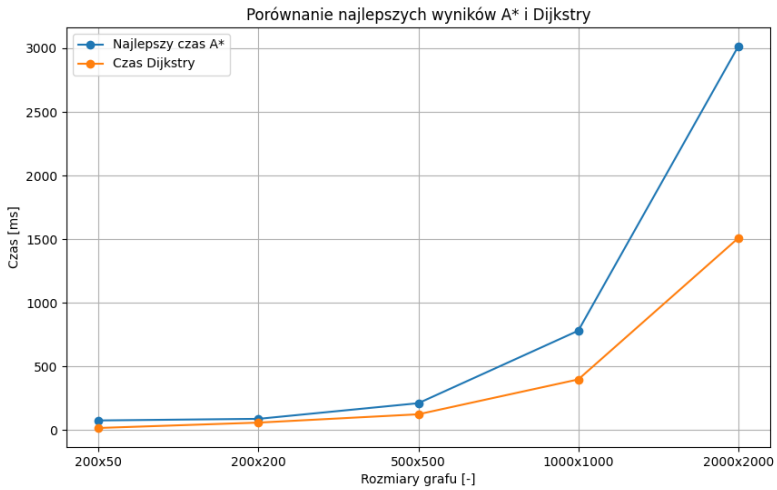
b) heurystyka Manhattan osiągała najkrótsze czasy wykonania dla większości badanych przypadków jednocześnie minimalnie odstając od pozostałych. W bardziej złożonym środowisku może to prowadzić do oszczędności czasu, ponieważ odległość manhattańska jest obliczeniowo najmniej złożoną heurystyką spośród badanych.



Rys. 4.9. Czas działania algorytmu A*



Rys. 4.10. Czas działania algorytmu Dijkstry



Rys. 4.11. Porównanie czasu działania algorytmów

Tabela 4.2. Wyniki algorytmu A* dla różnych heurystyk

Heurystyka	Graf	Czas [ms]
Manhattan	graph_200_50.pkl	76.2
	graph_200_200.pkl	88.9
	graph_500_500.pkl	211.8
	graph_1000_1000.pkl	789.2
	graph_2000_2000.pkl	3014.2
Euclidean	graph_200_50.pkl	95.2
	graph_200_200.pkl	107.0
	graph_500_500.pkl	236.4
	graph_1000_1000.pkl	804.1
	graph_2000_2000.pkl	3019.3
Max	graph_200_50.pkl	96.4
	graph_200_200.pkl	106.7
	graph_500_500.pkl	222.9
	graph_1000_1000.pkl	781.5
	graph_2000_2000.pkl	3038.1

Tabela 4.3. Wyniki algorytmu Dijkstry

Graf	Czas [ms]
graph_200_50.pkl	17.1
graph_200_200.pkl	59.3
graph_500_500.pkl	124.7
graph_1000_1000.pkl	398.5
graph_2000_2000.pkl	1508.4

Tabela 4.4. Tabela porównawcza najlepszych wyników A* i Dijkstry

Graf	Najlepszy czas A* [ms]	Czas Dijkstry [ms]
graph_200_50.pkl	76.2	17.1
graph_200_200.pkl	88.9	59.3
graph_500_500.pkl	211.8	124.7
graph_1000_1000.pkl	781.5	398.5
graph_2000_2000.pkl	3014.2	1508.4

4.3. Porównanie statycznych algorytmów nawigacyjnych w terenie z przeszkodami

W niniejszej sekcji przedstawiono opis eksperymentu, który rozszerza badanie efektywności algorytmów A* z różnymi heurystykami oraz algorytmu Dijkstry na teren z przeszkodami. Celem eksperymentu było porównanie, który z tych algorytmów lepiej radzi sobie ze znajdowaniem najkrótszej trasy w grafach reprezentujących mapę z przeszkodami. Dodatkowo, oprócz optymalizacji czasu działania, analizowano zdolność algorytmów do efektywnego omijania przeszkód i wyznaczania tras optymalnych pod względem długości. Eksperyment ten ma na celu dostarczenie danych na temat wyboru odpowiedniego algorytmu nawigacyjnego w różnych warunkach terenowych.

4.3.1. Przebieg eksperymentu

W celu porównania wydajności różnych heurystyk oraz algorytmu Dijkstry w kontekście znajdowania najkrótszej ścieżki w grafach reprezentujących mapy z przeszkodami, przeprowadzono serię eksperymentów. Każdy eksperyment składał się z następujących kroków:

a) wygenerowanie siatki grafu o wymiarach 200x50,

b) losowe dodanie 130 przeszkód o różnych rozmiarach,

c) określenie punktu startowego oraz celu na mapie,

d) przeprowadzenie procesu wyszukiwania najkrótszej ścieżki za pomocą algorytmu A* i Dijkstry,

e) pomiar długości znalezionych ścieżek i porównanie efektywności obu algorytmów.

Każdy z eksperymentów został powtórzony pięciokrotnie w celu uzyskania średnich wyników oraz zapewnienia wiarygodności otrzymanych danych. Wyniki eksperymentów zostały szczegółowo udokumentowane, prezentując zarówno długości ścieżek uzyskane przez każdy z algorytmów, jak i wizualizacje zaznaczające przebieg tych ścieżek na mapie. Eksperymenty te miały na celu nie tylko zidentyfikowanie najbardziej efektywnego algorytmu w kontekście czasu obliczeń, ale również ocenę ich zdolności do omijania przeszkód oraz znajdowania optymalnych tras na mapach złożonych z różnorodnych przeszkód. Dokładny przebieg eksperymentu został przedstawiony na listingu 4.10.

Listing 4.10. Porównanie algorytmów

```
for i in range(5):
    graph = create_grid_graph(200, 50)
    add_random_obstacles(graph, 130, 10, 10)

    draw_map(graph, None, f'map_{i}.png')


    start = (0, 30)
    goal = (199, 20)

    heuristics={
        "manhattan" : manhattan_distance,
        "euclidean" : euclidean_distance,
        "max" : max_distance
    }

    print(f"iteracja: {i}")
    path2 = dijkstra(graph, start, goal)
    print(f"Dlugosc trasy Dijkstra: {calculate_path_length(path2)}")
    draw_map(graph, path2, f'map_dijkstra_{i}.png')


    for k, v in heuristics.items():
        print(f"Funkcja obliczajaca odleglosc w algorytmie A*: {k}")
        path1 = a_star(graph, start, goal, v)
        print(f"Dlugosc trasy A*: {calculate_path_length(path1)}, { k }")
        draw_map(graph, path1, f'map_astar_{k}_{i}.png')
```

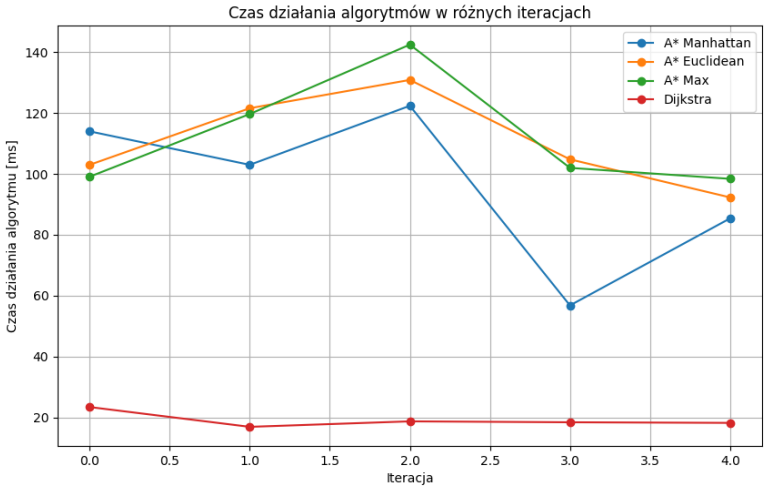
4.3.2. Porównanie otrzymanych wyników

Podsumowanie wyników przedstawionych w tabeli 4.5 wskazuje, że algorytm Dijkstry osiągnął najlepsze wyniki pod względem czasu działania dla wszystkich iteracji, utrzymując przy tym identyczną długość wyznaczonej trasy (247 dla iteracji 0, 1, 2, 4 oraz 233 dla iteracji 3). Algorytmy A* z różnymi heurystykami uzyskały takie same wyniki pod względem długości trasy, jednak wymagały więcej czasu obliczeniowego w porównaniu do Dijkstry (rysunek 4.12).

Iteracja	**Algorytm**	**Czas działania algorytmu [ms]**	**Długość trasy**
0	A* Manhattan	114.0	247
	A* Euclidean	103.0	247
	A* Max	99.1	247
	Dijkstra	23.4	247
1	A* Manhattan	103.0	243
	A* Euclidean	121.6	243
	A* Max	119.7	243
	Dijkstra	16.9	243
2	A* Manhattan	122.4	247
	A* Euclidean	130.9	247
	A* Max	142.5	247
	Dijkstra	18.7	247
3	A* Manhattan	56.8	233
	A* Euclidean	104.8	233
	A* Max	102.0	233
	Dijkstra	18.4	233
4	A* Manhattan	85.4	243
	A* Euclidean	92.3	243
	A* Max	98.4	243
	Dijkstra	18.2	243

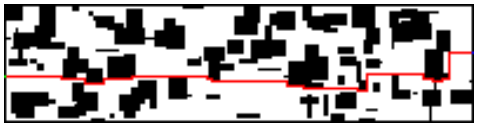
Tabela 4.5. Porównanie czasu działania i długości trasy dla algorytmu A* z różnymi heurystykami oraz algorytmu Dijkstry

Na niżej ukazanych obrazkach przedstawiono wyniki działania algorytmów przeszukiwania grafu dla dwóch wybranych iteracji. Każdy zestaw obrazków składa się z dwóch głównych sekcji: mapy ogólnej, porównania ścieżek dla algorytmów A* z różnymi heurystykami oraz algorytmu Dijkstry.

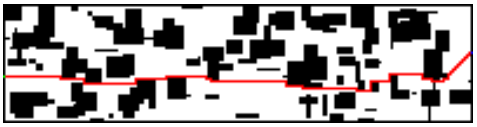


Rys. 4.12. Porównanie czasu działania algorytmów

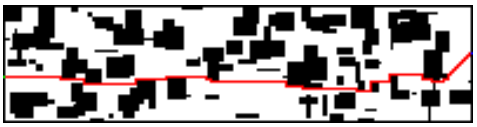
Rys. 4.13. Mapa dla iteracji 3



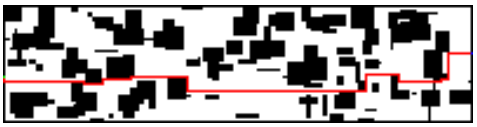
A* z heurystyką Manhattan



A* z heurystyką Euklidesową

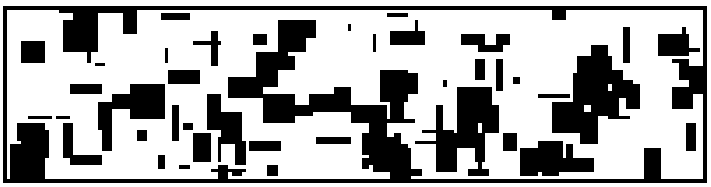


Max

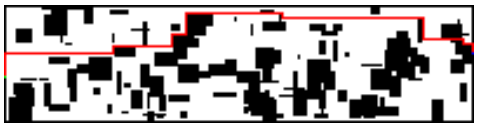


Dijkstra

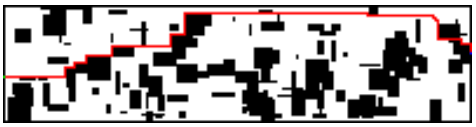
Rys. 4.14. Porównanie wyznaczonych ścieżek dla iteracji 3



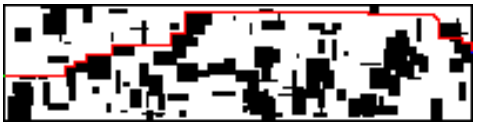
Rys. 4.15. Mapa dla iteracji 4



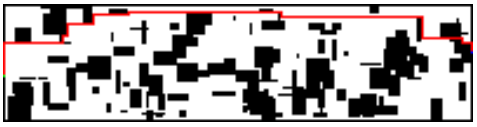
A* z heurystyką Manhattan



A* z heurystyką Euklidesową



Max



Dijkstra

Rys. 4.16. Porównanie wyznaczonych ścieżek dla iteracji 4

4.4. Porównanie dynamicznych algorytmów nawigacyjnych w terenie z przeszkodami

Niniejsza sekcja koncentruje się na opisie przeprowadzonego eksperymentu mającego na celu porównanie efektywności dwóch dynamicznych algorytmów nawigacyjnych: D* oraz D* Lite. Głównym celem badania jest ocena, który z tych algorytmów lepiej radzi sobie z nawigacją we wcześniej nieznanym środowisku, które może zawierać przeszkody.

4.4.1. Przebieg eksperymentu

Eksperyment skoncentrował się na porównaniu dwóch dynamicznych algorytmów nawigacyjnych: D* i D* Lite z algorytmem A* z metryką Manhattan słżącym jako algorytm bazowy. Badanie miało na celu ocenę, który z algorytmów dynamicznych lepiej radzi sobie z adaptacją do zmieniającego się środowiska oraz efektywnością znajdowania optymalnych ścieżek w terenie z przeszkodami. W każdej iteracji eksperymentu wykorzystano mapę o rozmiarach 100x100 cm z wyznaczonymi punktami startowymi i celowymi. Następnie, w zależności od iteracji, zmieniano rozmieszczenie i liczbę przeszkód w środowisku, po którym poruszał się robot.

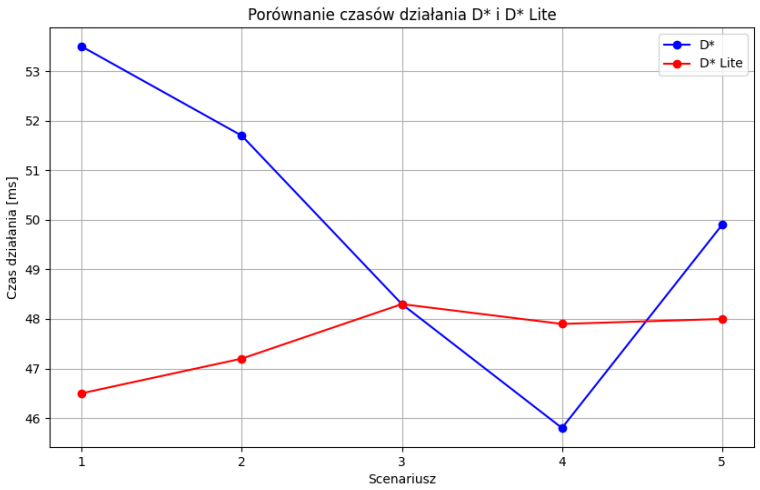
Algorytmy testowane były na tej samej mapie, tych samych punktach startowych i docelowych. Po wykryciu przeszkody, każdy z algorytmów rozpoczynał proces wyznaczania ścieżki od nowa, biorąc pod uwagę aktualne warunki terenowe.

4.4.2. Porównanie otrzymanych wyników

Na podstawie danych przedstawionych w tabeli 4.6 można wysunąć następujące wnioski:

a) w przypadku wszystkich scenariuszy, algorytm D* Lite wykazał się minimalnie lepszym czasem działania w porównaniu do D* (rysunek 4.17). Taki wynik może być spowodowany tym, że w badanych scenariuszach występowała stosunkowo mała ilość napotkanych przeszkód oraz mapa była małych rozmiarów, co ograniczało liczbę operacji wymaganych do znalezienia optymalnej ścieżki,

b) robot pokonał praktycznie tę samą odległość w każdym z pięciu przeprowadzonych scenariuszy testowych. Z uwagi na minimalnie lepszy czas działania, jaki uzyskał algorytm D* Lite, warto zdecydować się na jego wybór, szczególnie w kontekście optymalizacji czasu obliczeń w bardziej wymagających środowiskach.



Rys. 4.17. Porównanie czasu działania algorytmów

Iteracja	Ilość przeszkód	Czas działania dstar [ms], Pokonana odległość	Czas działania dstar_lite [ms], Pokonana odległość
1	4	53.5, 112	46.5, 112
2	4	51.7, 105	47.2, 105
3	0	48.3, 101	48.3, 101
4	3	45.8, 106	47.9, 107
5	2	49.9, 115	48.0, 115

Tabela 4.6. Porównanie czasów działania i pokonanych odległości dla D* i D* Lite

5. Podsumowanie i wnioski końcowe

W niniejszym rozdziale przedstawiono podsumowanie oraz wnioski końcowe wynikające z przeprowadzonych eksperymentów dotyczących optymalizacji nastaw regulatora PID oraz porównania różnych algorytmów statycznych i dynamicznych w kontekście nawigacji.

5.1. Optymalizacja nastaw regulatora PID

Podczas badań nad optymalizacją nastaw regulatora PID przeprowadzono szereg eksperymentów, których celem było znalezienie optymalnych parametrów regulacji. W szczególności, eksperyment z algorytmem genetycznym (GA 3) wykazał się najbardziej satysfakcjonującymi wynikami pod względem jakości regulacji oraz czasu ustalania. Nastawy te charakteryzowały się czasem ustalania na poziomie 600 ms, co jest akceptowalnym wynikiem w porównaniu do innych eksperymentów. W przeciwieństwie do tego, próby z nastawami metodą Zieglera-Nicholsa nie zapewniły odpowiedniej stabilności, co objawiało się problemami z działaniem jednego z elementów systemu.

Podsumowując, najlepszym wyborem okazały się nastawy uzyskane z algorytmu genetycznego (GA 3), ze względu na skuteczność w regulacji systemu.

5.2. Porównanie statycznych algorytmów nawigacyjnych w terenie bez przeszkód

Analizując wyniki eksperymentu porównującego algorytmy A* z różnymi heurystykami oraz algorytm Dijkstry, można wysunąć kilka istotnych wniosków. Algorytm Dijkstry osiągnął znacznie lepsze czasy wykonania w porównaniu do A*, co może być zaskakującym wynikiem, biorąc pod uwagę, że A* powinien działać szybciej. Heurystyka Manhattan wykazała się jako najbardziej efektywna pod względem szybkości obliczeń, co czyni ją najlepszym wyborem dla tego konkretnego eksperymentu.

Podsumowując, algorytm Dijkstry był najbardziej efektywny pod względem czasu wykonania i powinien być preferowany w przypadku prostych zastosowań nawigacyjnych bez przeszkód.

5.3. Porównanie statycznych algorytmów nawigacyjnych w terenie z przeszkodami

Eksperymenty porównujące algorytmy A* z Dijkstrą w obecności przeszkód potwierdziły, że algorytm Dijkstry nadal był najbardziej efektywny pod względem czasu działania, przy jednoczesnym utrzymaniu identycznej długości wyznaczonej trasy. Wyniki te sugerują, że pomimo zastosowania różnych heurystyk w algorytmie A*, jego wydajność nadal była gorsza niż prosty algorytm Dijkstry w kontekście nawigacji w złożonym środowisku.

Podsumowując, algorytm Dijkstry może być zaimplementowany do zastosowań nawigacyjnych w prostych środowiskach z przeszkodami, zapewniając zarówno efektywność czasową, jak i optymalizację długości trasy.

5.4. Porównanie dynamicznych algorytmów nawigacyjnych w terenie z przeszkodami

Badania porównawcze algorytmów D* i D* Lite w różnych scenariuszach wykazały, że D* Lite osiąga nieznacznie lepsze czasy wykonania. Prawdopodobnie jest to rezultat mniejszej liczby przeszkód oraz mniejszej skali badanych map. Pomimo minimalnej różnicy w czasie działania, algorytm D* Lite wydaje się być bardziej efektywnym rozwiązaniem do zastosowań w bardziej wymagających środowiskach nawigacyjnych.

Podsumowując, ze względu lepszą efektywność czasową i potencjał do skutecznego działania w złożonych scenariuszach nawigacyjnych, algorytm D* Lite może być preferowany jako rozwiązanie problemu nawigacyjnego.

5.5. Wkład własny

Za wkład własny autor uważa:

a) projekt, implementację oraz testy elektroniki i oprogramowania robota zgodnie z procedurami ASPICE,

b) fizyczną realizację konstrukcji robota,

c) implementację skryptów niezbędnych do przeprowadzenia opisanych w tej pracy badań,

d) przeprowadzenie badań, obróbka danych i analiza wyników.

Literatura

[1] Repozytorium z projektem: https://github.com/DevxMike/master_degree

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

[3] Bjarne Stroustrup: The C++ Programming Language. Addison-Wesley Professional 2013

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

[5] Warren Gay: FreeRTOS for ESP32-Arduino. Elektor International Media b.v. 2020

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

[7] Andrzej Dębowski: Automatyka. Podstawy teorii. Wydawnictwo Naukowe PWN 2008

[8] Standard ASPICE: https://mfiles.pl/pl/index.php/Automotive_SPICE

[9] Dokumentacja *std::array*: https://en.cppreference.com/w/cpp/container/array

[10] Regulator PID: https://pl.wikipedia.org/wiki/Regulator_PID

[11] Algorytm Dijkstry: http://algorytmy.ency.pl/artykul/algorytm_dijkstry

[12] Algorytm A*: https://pl.wikipedia.org/wiki/Algorytm_A*

[13] Algorytm D*: https://en.wikipedia.org/wiki/D*

[14] Algorytm D*Lite: https://en.wikipedia.org/wiki/D*#D*_Lite

[15] Metoda Ziegler-Nichols: https://en.wikipedia.org/wiki/Ziegler%E2%80%93Nichols_method

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

STRESZCZENIE PRACY DYPLOMOWEJ MAGISTERSKIEJ

PORÓWNANIE WYBRANYCH 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 SELECTED NAVIGATION ALGORITHMS

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

Thesis 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.