WYDZIAŁ
BUDOWY MASZYN
I LOTNICTWA
POLITECHNIKI RZESZOWSKIEJ

Damian Bielecki

Implementacja sterowania hierarchicznego mobilnym robotem kołowym z wykorzystaniem języka Python

Praca dyplomowa inżynierska

Opiekun pracy:
dr inż. Paweł Penar

Rzeszów, 2023

Spis treści

1. Wprowadzenie

Od lat można zauważyć zwiększająca się popularność różnych robotów, które bazując na wprowadzonej do systemu mapie autonomicznie poruszają się po terenie tak aby osiągnąć określony cel. Celem projektu inżynierskiego będzie zaimplementowanie algorytmu wyszukującym najkrótszą ścieżkę. Do przetestowania algorytmu zostanie napisany prosty program symulujący robota poruszającego się po wyznaczonej ścieżce. Kolejnym etap projektu to zbudowanie robota mającego zweryfikować zaproponowaną przez algorytm ścieżkę.

Głównym powodem podjęcia się implementacji takiego algorytmu jest chęć zapoznania się z systemami planującymi ruch robota i próba zaimplementowania takiego rozwiązania na fizycznym robocie.

Ze względu na cel projektu inżynierskiego oraz powód jego realizacji, przyjęto, że praca będzie złożona z przeglądu literatury, implementacji i symulacji algorytmu A* w języku Python oraz weryfikacji przyjętych rozwiązań obiekcie rzeczywistym.

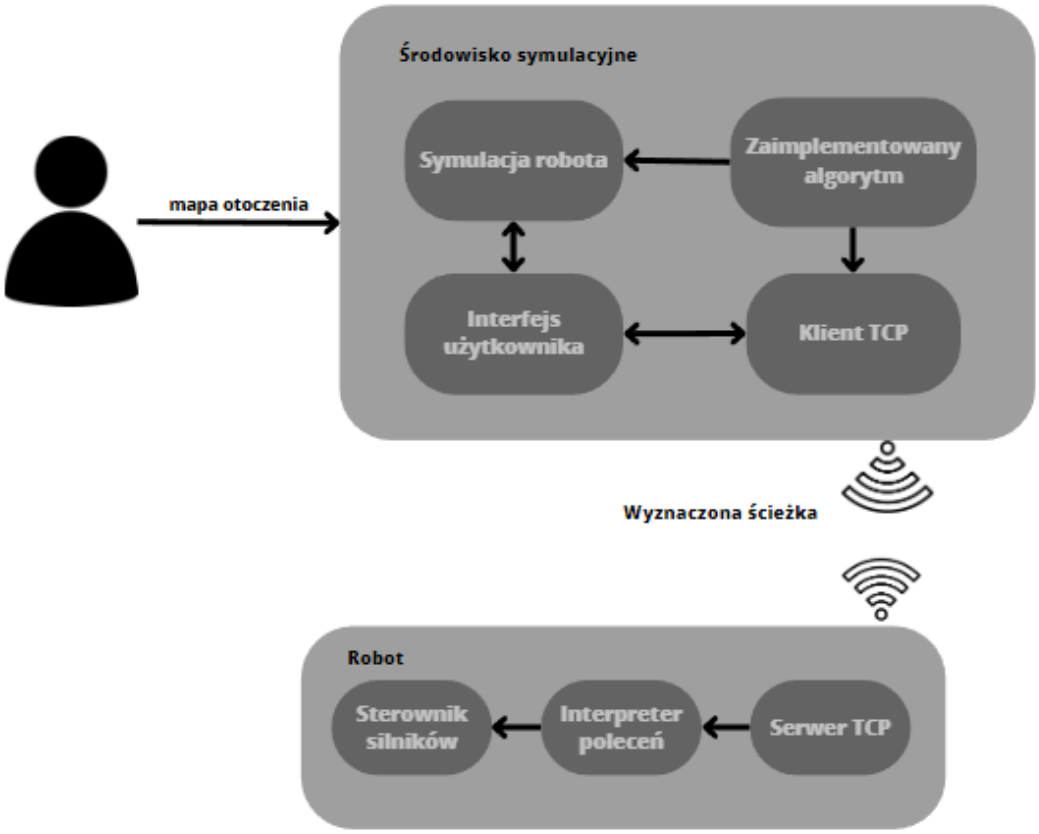
Mając na uwadze zakres pracy, jej treść podzielono na szereg rozdziałów. Rozdział pierwszy stanowi przegląd literatury oraz istniejących rozwiązań, które będą podstawą założeń przyjętych we opracowywanym rozwiązaniu. Rozdział drugi skupia się na genezie, charakterystyce i opisie działania wybranego algorytmu wyszukiwania najkrótszej ścieżki. Dodatkowo zostaną przedstawione inne algorytmy zwiększającą autonomie robota mobilnego. Rozdział trzeci przedstawia własną implementacje algorytmu A* oraz środowisko symulacyjne w języku Python. Kolejny rozdział to omówienie budowy robota mobilnego oraz jego programu sterującego. W tym kontekście szczególną uwagę poświęcono komunikacji oraz sterowaniu silnikami Rozdział piąty stanowi podsumowanie pracy i przedstawia przeprowadzone testy wykonane w środowisku symulacyjnym oraz rzeczywistym.

2. Przegląd literatury

2.1. Przegląd istniejących rozwiązań

2.2. Opis przyjętego rozwiązania

Algorytm wyznaczania najkrótszej ścieżki wraz z środowiskiem testowym został napisany w języku Python. Implementacja algorytmu nie wymaga żadnych dodatkowych bibliotek. Do zbudowania środowiska symulacyjnego zostały użyte biblioteki graficzne do rysowania kształtów geometrycznych oraz podstawowych elementów graficznego interfejsu użytkownika. Weryfikacja działania algorytmu została przeprowadzona na zbudowanym trójkołowym robocie. Oprogramowanie robota zostało napisane w języku C++ i dodatkowych bibliotekach udostępnionych przez producentów użytych sterowników.



Rysunek 2.1: Ogólny schemat projektu

Użytkownik po uruchomieniu środowiska symulacyjnego będzie miał możliwość edycji mapy opartej na siatce, określenia punktu początkowego i końcowego. Kolejnym krokiem jest wyznaczenie ścieżki oraz jej graficznej reprezentacji. Tak wyznaczoną ścieżkę można wysłać do robota operującego w środowisku rzeczywistym.

3. Algorytm A*

3.1. Geneza powstania

Algorytm A* powstał w ramach projektu Shakey, zapoczątkowanego w 1966 roku przez Charles Rosen'a. Celem projektu było zbudowanie robota, który potrafiłby planować własne działania. Zbudowany robot wyróżniał się na tle innych tym że integrował kilka różnych modeli sztucznej inteligencji pracującej jako jeden system.

Robot został zbudowany z:

- kamery telewizyjnej i dalmierza optycznego - system wizyjny do obserwacji środowiska
- łącza radiowego - służącego do komunikacji z bazą, odbierania i wysyłania komend
- detektora uderzeń - pozwalający na zatrzymanie robota w przypadku kolizji

Komunikacja odbywała się poprzez wysyłane radiowo, tekstowe polecenia mające określoną strukturę np.: GOTO D4 - co oznaczało automatyczne przemieszczenie się robota do wskazanej pozycji

Rysunek 3.2: Robot Shakey i Charles Rosen, inicjator projektu [2]

3.2. Opis działania algorytmu

A* to heurestyczny algorytm wyznaczający najkrótszą możliwą ścieżkę w grafie. Jest to algorytm zupełny i optymalny, a więc zawsze zostanie wyznaczone optymalne rozwiązanie. Ze

względu na przeszukiwanie oparte na grafie algorytm działa najlepiej na strukturze drzewiastej. Zadaniem algorytmu jest minimalizacja funkcji:

$$f(x) = h(x) + g(x) \tag{3.1}$$

gdzie: $f(x)$ - minimalizowana funkcja, $g(x)$ - to rzeczywisty koszt dojścia do punktu x.

Funkcja $h(x)$ to funkcja heurestyczna, oszacowuje ona koszt dotarcia od punktu x do wierzchołka docelowego

Zalety:

- jest kompletny i optymalny
- może przeszukiwać skomplikowane mapy
- jest najwydajniejszym takim algorytmem

Wady:

- jego wydajność w znacznej mierze zależy od funkcji heurestycznej
- Każda akcja ma stały koszt wykonania
- nie nadaje się do często zmieniającego się otoczenia robota, wymaga ponownego przeliczenia

Przykładowe funkcje heurestyczne:

- Funkcja euklidesowa

$$h(x) = 10 * \sqrt[2]{(x_1 - x_2)^2 + (y_1 - y_2)^2} \tag{3.2}$$

- Geometria Manhattanu (innaczej metryka miejska)

$$h(x) = |x_2 - x_1| + |y_2 - y_1| \tag{3.3}$$

Gdzie: x_1 i y_1 to współrzędne wyznaczanego punktu, x_2 i y_2 to koordynaty celu

3.3. Analiza innych algorytmów

- **RRT(Rapidly-exploring random tree)**[6] - to algorytm generujący i łączący losowe punkty w przestrzeni. Z każdym wygenerowanym wierzchołkiem sprawdzane jest czy ten omija przeszkody. Jego działanie kończy się gdy węzeł jest wygenerowany we wskazanym regionie lub zostanie osiągnięty limit. Jego zalety to:

- Balans pomiędzy zachłannością a eksploracją
- prosty i szybki w implementacji
- Zbiega się z rozkładem próbkowania

Jego wady:

- jest czuły na metrykę
- duża złożoność obliczeniowa

g

- **SLAM(Simultaneous localization and mapping)** [7] - jest metodą używaną w autonomicznych pojazdach. Pozwana na zbudowanie oraz lokalizacje pojazdu względem otoczenia na mapie. Algorytm ten nie wyznacza bezpośrednio ścieżki omijającej przeszkody jednak w praktyce jest ważnym elementem takiego systemu. Każdy ruch robota w rzeczywistym środowisku jak i sama mapa obarczona jest pewnym błędem powodującym że robot jest w innym miejscu niż zakładamy. Możemy wyróżnić działanie algorytmu w oparciu o lidary albo o metody wizyjne. Wadą metody jest wysoki koszt obliczeniowy i narastający z czasem błąd pozycji co w konsekwencji może spowodować utratę pozycji.

Rysunek 3.3: Przykład mapowania metodą SLAM[7]

4. Implementacja

4.1. Implementacja algorytmu

Mechanizm wyszukiwania najkrótszej ścieżki został zamknięty w jednym module o nazwie aStar. Na moduł składa się klasa AStar ze wszystkimi potrzebnymi metodami oraz dodatkowa klasa Node reprezentująca pojedynczy punkt przeszukiwanego grafu.

Wyznaczanie najkrótszej ścieżki rozpoczyna się od wyznaczenia kosztu punktu startowego i utworzenia zbioru z nieprzeszukanymi wierzchołkami, do którego dopisujemy punkt początkowy.

```
    openList = []
    startNode.h = self.heurestic(startNode.getCords(), endNode.getCords
   ())
    startNode.g = 0
    heappush(openList, startNode)
```

Listing 1: Przygotowanie danych

Wewnętrzna funkcja heurestic przyjmująca pozycje dwóch punktów odpowiada za wyliczenie optymistycznego kosztu przejścia od punktu x do wierzchołka docelowego. Takie podejście pozwala na szybką podmianę funkcji bez znaczących zmian w programie. Wykorzystywana funkcja heurestyczna to równanie nr. (3.2)

W kolejnym kroku uruchamiana jest pętla, która wykonywana jest dopóki w zbiorze otwartym znajdują się nie odwiedzone elementy. Z listy pobierany jest element o najmniejszej liczbie punktów co oznacza, że dany wierzchołek drzewa ma największe szanse być najlepszym rozwiązaniem. Jeżeli pobrany element nie jest celem to dalej pobierani i przetwarzani są wszyscy jego sąsiedzi. W tym rozwiązaniu, dla zaoszczędzenia pamięci, symulator przechowuję tylko przeszkody. W związku z tym do pobrania sąsiadów używana jest specjalna funkcja sprawdzająca czy na mapie o wskazanych koordynatach istnieje punkt. Jeżeli takowy obiekt nie istnieje to oznacza, że algorytm może użyć tych współrzędnych do trasowania ścieżki. Od tej części programu zależy, czy algorytm ma wyznaczyć ścieżkę uwzględniając skoki po skosie.

```
neighborNode = self._findNodeOnList(openList, newCords)
if neighborNode is None:
    neighborNode = Node(newCords)
    neighborNode.g = COST
    neighborNode.h = self.heurestic(newCords, endNode.getCords())
    neighborNode.parent = currentNode
    openList.append(neighborNode)

distance_from_curr_to_neighbor = COST
scoreFromStartToCurrentNeighbor =  currentNode.g +
   distance_from_curr_to_neighbor
```

```
11
12 if neighborNode.getScore() <= currentNode.getScore():
13     neighborNode.g = scoreFromStartToCurrentNeighbor
14     neighborNode.h = scoreFromStartToCurrentNeighbor + self.heurestic(
   newCords, endNode.getCords())
15     neighborNode.parent = currentNode
```

Listing 2: Wyznaczenie kosztu ścieżki

Powyższy kod odpowiada za wyznaczenie kosztu przejścia do sąsiada. Jeżeli punkt nie istnieje jeszcze w otwartym zbiorze to jest tworzony i do niego dodawany. Zgodnie z założeniem całkowity koszt to suma rzeczywistego kosztu dystansu i wyniku funkcji heurestycznej. Rzeczywisty koszt (funkcja $g(x)$) jest ustalony na sztywno i zależy od współrzędnych, do których skaczemy. Jeżeli następny punkt jest na wprost to koszt wynosi 10. Jako iż przechodzimy po siatce z węzłami o stałej i równej odległości to koszt skoku do sąsiada po skosie został wyznaczony z twierdzenia Pitagorasa. Należy zwrócić uwagę, że jeżeli całkowity koszt jest mniejszy od aktualnego to węzeł ma ustawianego rodzica, ma to duże znaczenie w przypadku wyjścia z pętli i wyznaczenia najkrótszej trasy spośród wszystkich kosztów.

```
1 if currentNode == endNode:
2     path = []
3     while currentNode is not None:
4         path.append(currentNode)
5         currentNode = currentNode.getParent()
6     return path
7 else:
8     return []
```

Listing 3: Przygotowanie danych

Po zakończeniu pętli, sprawdzane jest czy została znaleziona ścieżka, jeżeli takowa istnieje to w kolejnej pętli wyznaczane są przy pomocy pola wskazującego na rodzica kolejne punkty trasy. Jeżeli węzeł nie jest punktem końcowym to trasa nie została znaleziona i zwracana jest pusta lista.

4.2. Implementacja środowiska testowego

Środowisko symulacyjne zostało napisane w języku Python i bibliotekach pygame[3] i pygame_gui[4], zapewniających obsługę graficznego interfejsu użytkownika.

```
1 if __name__ == '__main__':
2     app = App()
3     app.main()
```

Listing 4: Uruchomienie aplikacji

Obsługa mapy

Dla logicznego podzielenia programu została napisana główna klasa programu o nazwie App. Takie podejście pozwoliło na odseparowanie poszczególnych funkcji aplikacji. W pierwszej kolejności konstruktor obiektu aplikacji, generuje okno, tworzy graficzne elementy użytkownika oraz ustawia odpowiednie flagi informujące o stanie aplikacji. Dalej wywoływana jest funkcja main wczytująca odpowiednią mapę z pliku, a następnie uruchamiająca główną pętlę programu działającą w 60 klatkach na sekundę.

Rysunek 4.4: Schemat głównej pętli programu

Aby uprościć sobie symulowanie różnych scenariuszy i rozmiarów map otoczenia, ta ładowana jest ze wskazanego w programie pliku. Wskazany plik jest w formacie json i podzielony jest na dwie sekcje. Pierwsza dostarcza informacji o samej mapie, natomiast druga o poruszającym się po niej robocie. Sekcja mapy zawiera lokalizacje do pliku tekstowego z zapisanymi elementami.

```
1 (self.map, self.robot) = MapLoader.fromJson("maps/m02.json")
```

Listing 5: Uruchomienie aplikacji

Ładowaniem wszystkich tych informacji zajmuje się specjalna metoda w klasie MapLoader, wywołana przed uruchomieniem pętli głównej programu. Wczytywana mapa jest w postaci

siatki i każdy jej element ma stały rozmiar. Symulator rozróżnia kilka typów węzłów:

- brak elementu - możliwy przejazd
- ściana - przeszkoda do ominięcia przez algorytm
- punkt początkowy - od niego rozpoczyna się wyznaczanie ścieżki, jest zdefiniowany przez użytkownika
- punkt końcowy - analogicznie jak w poprzednim przypadku
- ścieżka - element wyznaczonej trasy

Wszystkie wymienione typy zapisane są w klasie Tile, reprezentującej pojedynczy rysowany i zapisywany w pliku kwadrat. Użytkownik może w każdej chwili działania aplikacji edytować mapę a przed zamknięciem programu zaktualizowana mapa jest zapisywana do pliku. Wyjątkiem jest obiekt będący elementem ścieżki, ten nie jest zapisywany do pliku, ponieważ generowany jest dynamicznie i służy tylko do reprezentacji wygenerowanej ścieżki.

Wyznaczanie ścieżki i symulacja

Po załadowaniu program oczekuje na akcje użytkownika i ewentualne uruchomienie wyznaczania ścieżki lub całej symulacji.

```
1 elif ev.ui_element == self._startAStar:
2     star = AStar(self.map.map, self.map.getSize())
3     star.diagonalJump(self._diagonalJump)
4     star.diagolnalCorrection(self._diagonalCorrextion)
5     self.path       = star.findPath2(Node(self.map.getStartCords()),
      Node((self.map.getEndCords())))
6     self._pathExist = True
7     self.robot.stopSimulation()
8     self.robot.hideRobot()
```

Listing 6: Uruchomienie aplikacji

Po kliknięciu w przycisk odpowiadający za wyznaczenie i pokazanie ścieżki, program tworzy obiekt wyznaczający trasę i ustawia dodatkowe opcje algorytmu (np. skok po przekątnych). Dalej pobierana jest wyznaczona trasa będąca tablicą kolejnych punktów, po których należy przejść aby dojść do celu. Na końcu ustawiana jest odpowiednia flaga programu, zatrzymywana jest symulacja i ukrywany jest robot (mamy nowo wyznaczoną trasę). Przycisk uruchamiający symulacje sprawdza czy istnieje ścieżka i w razie potrzeby ją generuje. Następnie trasa podawana jest do obiektu robota odpowiadającego za symulacje i dalej jest ona uruchamiana.

Przejście robota po ścieżce

Symulator robota pobiera pierwszy punkt, do którego ma dotrzeć. Zmiana pozycji odbywa się poprzez regulator typu P z ustawioną nastawą na wskazany punkt. Wartość wzmocnienia proporcjonalnego wczytana jest z pliku i oznaczona jest jako szybkość robota. Dalszy etap to sprawdzenie czy symulowany robot dotarł do celu.

```
if abs(deltaPos0) < 0.2 and abs(deltaPos1) < 0.2:
            self._currentTarget -= 1
```

Listing 7: Uruchomienie aplikacji

Realizowane jest to poprzez sprawdzenie czy wartość bezwzględna z różnicy aktualnej oraz docelowej pozycji jest mniejsza niż 0,1. Po osiągnięciu punktu, ustawiany jest nowy indeks tablicy wskazujący na kolejny cel. Jeżeli na początku aktualizacji indeks jest ujemny to robot przejechał po całej wyznaczonej trasie.

Komunikacja z robotem

Po wyznaczeniu ścieżki symulator na polecenie użytkownika może połączyć się ze zbudowanym robotem aby ten osiągnął założony cel. Zgodnie ze schematem [**?**] robot po połączeniu z siecią WiFi i utworzeniu serwera TCP oczekuje na wysyłane komendy. Aplikacja podczas uruchamiania, tworzy gniazdo, które potem zostanie wykorzystane do połączenia.

```
self._robotSocket       = socket.socket(socket.AF_INET, socket.
   SOCK_STREAM)
```

Listing 8: Utworzone gniazdo

Dla zwiększenia elastyczności utworzone są dwa pola tekstowe pozwalające na wprowadzenie adresu ip oraz portu z uruchomionym serwerem. Po wprowadzeniu wymaganych danych operator może spróbować nawiązać połączenie z robotem poprzez specjalny przycisk.

```
if self._robotSocketFlag is True:
    self._robotSocket.close()

print(f"Laczenie z robotem {(self._textRobotIp.get_text(), int(self.
   _textRobotPort.get_text()))}")
self._robotSocket.settimeout(1)
try:
    self._robotSocket.connect((self._textRobotIp.get_text(), int(self.
   _textRobotPort.get_text())))
    self._robotSocketFlag = True
    self._sendCommandToRobot("setMode auto")
    print("Podlaczono!!!")
except Exception as err:
    self._robotSocketFlag = False
    print(f"Blad polaczenia z robotem")
    print(err)
```

Listing 9: Nawiązanie połączenia

W pierwszej kolejności sprawdzana jest flaga informująca o stanie połączenia, jeżeli jest aktywne to jest ono zamykane. Na utworzonym gnieździe wywoływana jest funkcja connect z wprowadzonymi przez użytkownika parametrami. Cała operacja zamknięta jest w bloku try..except a więc jeżeli program nie połączy się z robotem to wyświetlana jest odpowiednia informacja. Podczas łączenia należy zwracać uwagę na podsieci, w których jest robot i sterujący laptop. Dodatkowo robot posiada dynamiczne ip co zostało szerzej opisane w rozdziale o oprogramowaniu robota. Jeżeli żaden wyjątek nie został rzucony to wysyłana jest pierwsza komenda ustawiająca tryb robota na automatyczny.

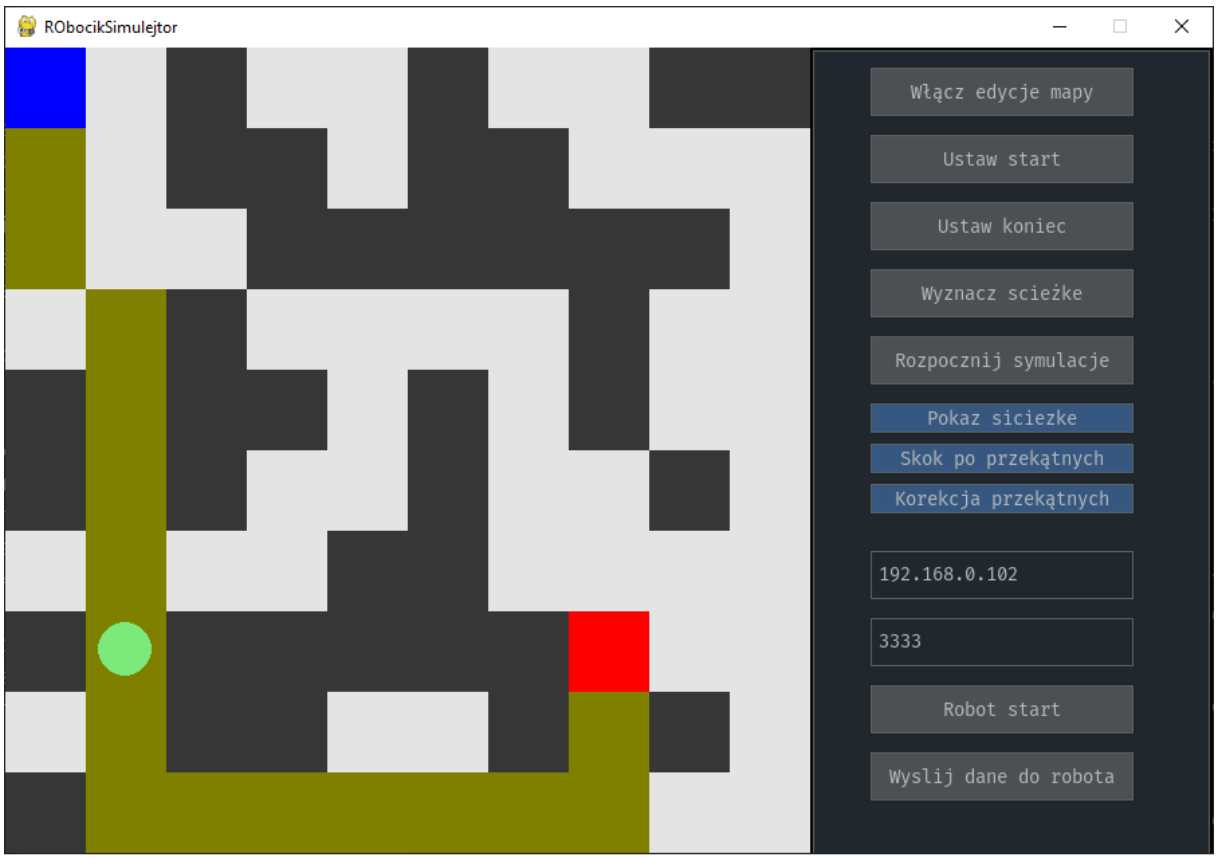
```
def _sendCommandToRobot(self, cmd):
    if self._robotSocketFlag is False:
        print("Robot nie podlaczony!!")
        return

    try:
        self._robotSocket.send(str.encode(cmd))
    except OSError as err:
        print("Robot nie podlaczony!!!")
        self._robotSocket.close()
        self._robotSocketFlag = False
        return
```

Listing 10: Wysyłanie komendy do robota

Program może wysłać dowolną komendę robota poprzez metodę sendCommandToRobot. Podobnie jak przy połączeniu, metoda wysyłająca dane może rzucić wyjątek o braku połączenia. W takim przypadku wyświetlana jest odpowiednia informacja i resetowana jest flaga połączenia.

Finalna wersja programu

Na poniższym zrzucie widać uruchomiony program z trwającą symulacją. Po prawej stronie został umieszczony panel z interfejsem użytkownika. Po lewej stronie widoczna jest wczytana mapa. Kolorem niebieskim został oznaczony punkt startowy a czerwonym końcowy. Żółte kratki oznaczają wyznaczoną przez algorytm ścieżkę, po której przejdzie robot. Szare pola to przeszkody, które robot ma ominąć. Robot oznaczony jest przez jasno-zielone koło, które przesuwa się po mapie.



Rysunek 4.5: Widok finalnej wersji programu

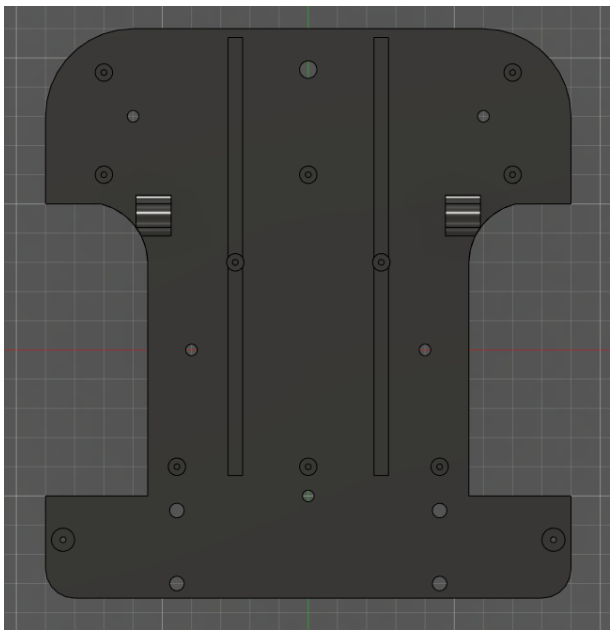
5. Projekt robota

5.1. Założenia projektowe

Wykonany robot powinien być jak najmniejszy i najprostszy w wykonaniu oraz sterowaniu tak aby móc przetestować przy jego pomocy działanie algorytmu A*. Robot będzie zbudowany z platformy, do której zostaną przyczepione napędy, elektronika sterująca oraz bateria. Do platformy zostaną przymocowane dwa gotowe moduły napędowe składające się z silnika, przekładni oraz dużego koła. Aby pojazd stał stabilnie, doczepione zostanie trzecie koło obracające się swobodnie w każdym kierunku. Całość będzie sterowana przy pomocy mikroprocesora ESP32 oraz dwukanałowego sterownika silników DC opartym na układzie L298n. Za zasilanie będzie odpowiadał litowo-jonowy akumulator 4S.

5.2. Projektowanie robota w środowisku CAD

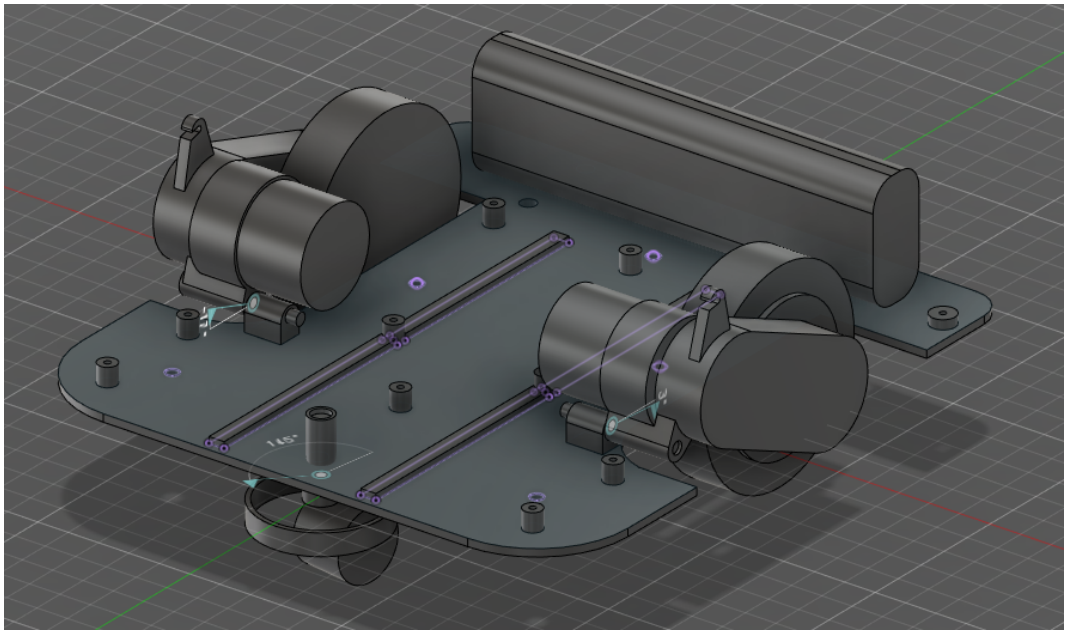
Podstawa robota utrzymująca wszystkie komponenty zostanie wydrukowana na drukarce 3D, a model platformy i pozostałych posiadanych elementów został wykonany w programie Fusion 360. Wykonane modele modułów napędowych, przedniego kółka oraz baterii pozwoliły na optymalne rozmieszczenie wszystkich elementów.



Rysunek 5.6: Widok z góry na podwozie robota

Model został przygotowany w programie Ultimaker Cura i wydrukowane z filamentu PLA. Temperatur głowicy wynosiła $220°C$ a stołu $60°C$. Aby przyśpieszyć wydruk wysokość warstwy została ustawiona na 0,2mm. Żeby zwiększyć wytrzymałość i lepiej zgrzać warstwy,

temperatura głowicy została lekko zawyżona względem wymagań producenta filamentu, a model posiada dwa dodatkowe wsporniki wzdłuż platformy.

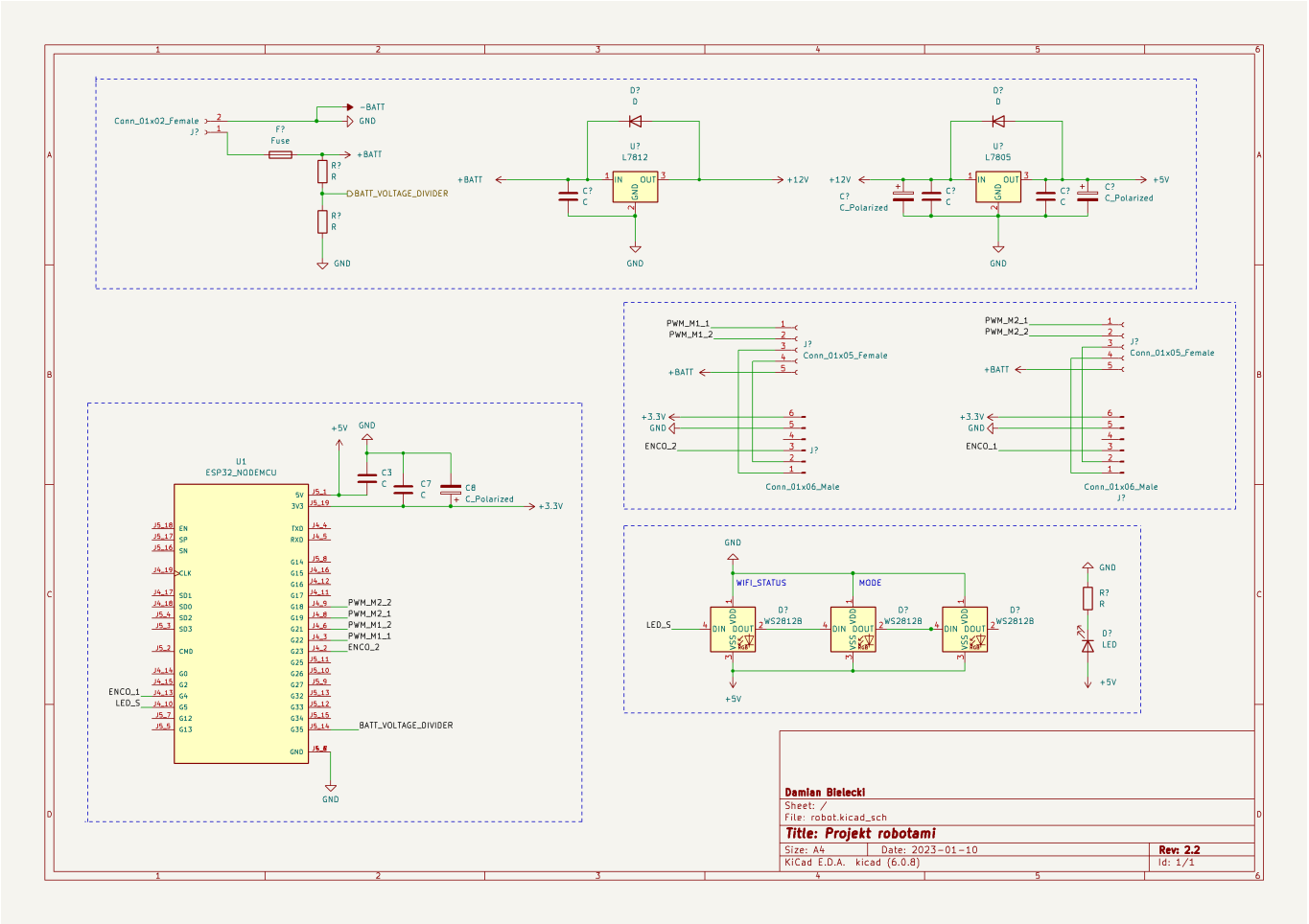


Rysunek 5.7: Model robota z modułami

Na powyższym zdjęciu widać zaprojektowaną ramę wraz z odpowiednio ustawionymi gotowymi modułami. Moduły napędowe zaczepione są przy pomocy fabrycznego trzpienia wciskanego w ramę. Takie rozwiązanie oznacza że napęd może obracać się w wokół własnej osi. Aby zapewnić stały i równomierny docisk koła do podłoża, napęd został połączony sprężyną z ramą.

5.3. Elektronika sterująca robotem

Do bezprzewodowego sterowania robotem zostanie wykorzystany moduł ESP32, dla którego zostanie przygotowana odpowiednia płytka z wyprowadzeniami do enkoderów silnika oraz ich sterownika. Schemat i projekt płytki drukowanej został wykonany w programie KiCad.



Rysunek 5.8: Schemat połączeń pomiędzy modułami

Przedstawiony powyżej schemat został podzielony na kilka sekcji. Najbardziej rozbudowana jest sekcja zasilania, która poprzez dwa stabilizatory liniowe obniża napięcie z 16,8V do 5V. Mikroprocesor zasilany jest napięciem 3,3V przygotowanym przez układ wlutowany na płytce deweloperskiej. Każda linia zasilania filtrowana jest przez kondensator elektrolityczny i ceramiczny, wynika to z dużego poboru prądu podczas komunikacji z WiFi.

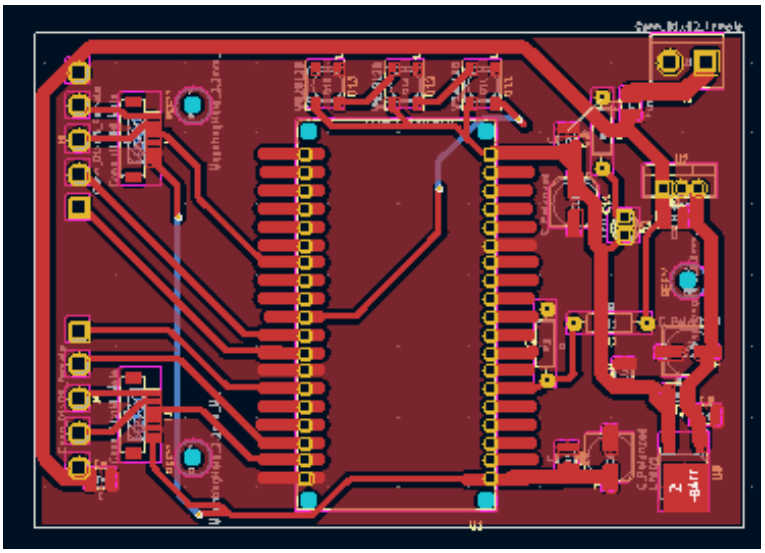
Kolejny segment zawiera po dwa złącza na silnik. Jedno złącze służy do połączenia z silnikiem, zasilania i odbierania danych z enkodera. Drugie złącze służy do połączenia sterownika z silnikiem i dwoma kanałami PWM, pozwalającymi na płynne sterowanie kierunkiem i prędkością obrotową silnika.

Trzeci segment zawiera świecące diody programowalne służące do sygnalizacji stanu robota. Zaletą wykorzystanych tych diod jest szeregowe połączenie i bezpośrednie zasilanie z lini 5V. Ostatnia podłączona dioda jest na stałe i sygnalizuje podłączoną baterię.

Ostatnia sekcja to połączenia pomiędzy modułem deweloperskim ESP32 a resztą układu. Mikroprocesor pracuje w logice 3,3V a sterownik silników 5V. Mimo tego, nie są potrzebne żadne

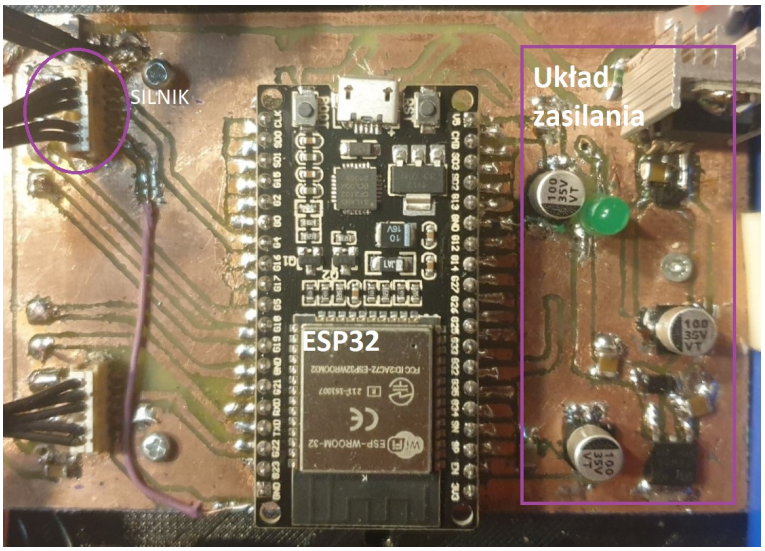
układy zamieniające poziomy napięć. Dodatkowo obok wyprowadzeń modułu widać dodatkowe kondensatory filtrujące zasilanie.

Użyte moduły napędowe posiadają enkoder inkrementalny zbudowany z czujnika halla i tarczy magnetycznej zamocowanej na wale silnika. Silniki sterowane są modułem z układem L298n, a prędkość ustalana jest poprzez odpowiednio generowany sygnał PWM.



Rysunek 5.9: Projekt PCB

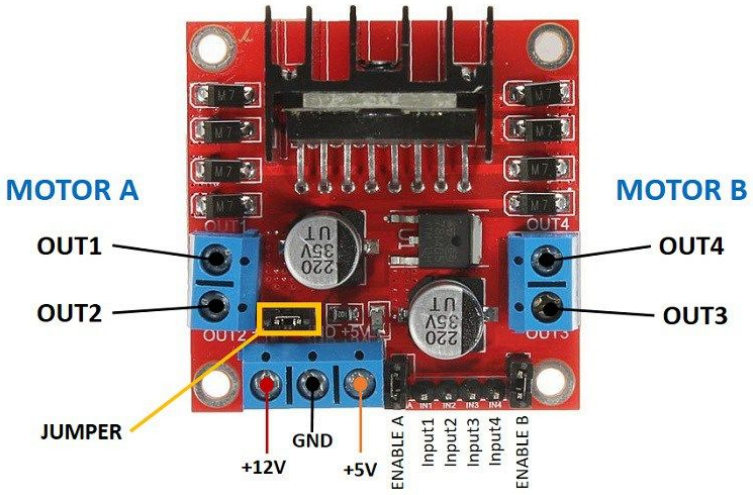
Na widocznym powyżej zdjęciu widać ostateczną wersje płytki drukowanej. Większość połączeń na płytce zrealizowana jest poprzez górną warstwę miedzi, jednak widać kilka ścieżek w dolnej warstwie.



Rysunek 5.10: Wykonana płytka

Płytka została wykonana na jednostronnym laminacie poprzez naniesienie utwardzalnej

światłem UV maski. Pozostała nie utwardzona część maski została zmyta w roztworze węglanu sodu. Tak przygotowana płytka została utwardzona w 40% roztworze chlorku żelaza(|||). Połączenia na dolnej warstwie zostały wykonane przy pomocy dodatkowych kabli.

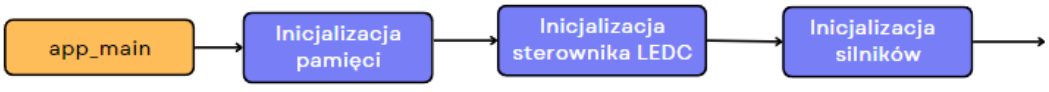


Rysunek 5.11: Wykorzystany moduł do sterowania silnikami

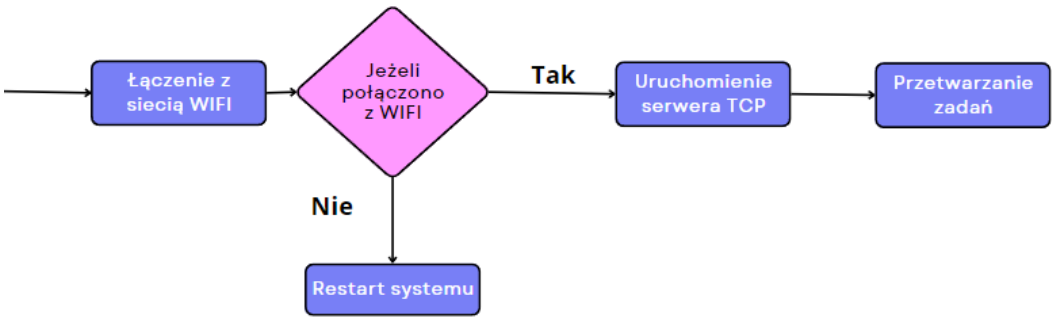
Wykonana płytka połączona jest z zewnętrznym dwukanałowym sterownikiem silników opartym na układzie L298n. Sterownik maksymalnie po może obsłużyć silniki do 46V i 2A na kanał. Jego duża zaleta to praca w logice 5V oraz zabezpieczenie przed niedozwolonymi stanami i nadmierną temperaturą.

5.4. Oprogramowanie robota

Program sterujący robotem został napisany w C++. Szkielet aplikacji bazuje na projekcie utworzonym przez framework IDF w wersji 4.4 udostępnionym przez producenta użytego procesora. Po za tym użyta została biblioteka implementująca system czasu rzeczywistego FreeRTOS i ASIO do obsługi połączenia TCP.



Rysunek 5.12: Schemat programu cz.1



Rysunek 5.13: Schemat programu cz.2

Działanie programu rozpoczyna się od wywołania funkcji app_main, inicjalizacji funkcji systemowych i sterowników silników. W dalszej kolejności nawiązywana jest łączność z siecią WiFi. W przypadku braku połączenia system resetuje się. Po poprawnym połączeniu uruchamiany jest kontekst biblioteki ASIO, a następnie uruchomienie serwera TCP. Klienci po połączeniu do utworzonego serwera wysyłają komendy tekstowe wraz z odpowiednimi argumentami, które robot odpowiednio przetwarza.

5.4.1. Sterowanie silnikami

Sterowanie silnikami odbywa się poprzez klasę MotorController, która dziedziczy po klasie PIDController implementującej regulator PID. Obiekt automatycznie tworzy timer uruchamiający co 100ms metodę aktualizującą wyjścia sterujące silnikiem. Aktualna prędkość wyznaczana jest na podstawie przerwania wyzwalanego przez enkoder silnika. Przerwanie inkrementuje licznik, czyszczony przez wcześniej opisany timer. Nastawy regulatora PID zostały dobrane eksperymentalnie, człon proporcjonalny wynosi 8 a całkujący i różniczkujący 0,1. Regulator PID możemy opisać przy pomocy wzoru:

$$u(x) = e(x) * P + \int e(x) * I + \partial e(x) * D \tag{5.4}$$

Gdzie: $e(x)$ – jest błędem; P,I,D – to stałe odpowiednio członu proporcjonalnego, całkującego i różniczkującego

$$e(x) = y_n ast(x) - y_a ktu(x) \tag{5.5}$$

Gdzie: $y_n ast(x)$ – to nastawa regulatora, $y_a ktu(x)$ – jest rzeczywistą zmierzoną wartością

Powyżej przedstawiony regulator został zaimplementowany w funkcji calcOutput klasy PIDController.

```
1 float PIDController::calcOutput(float current, float set)
2 {
```

```
  float error = set - current;
  float der = (error - this->_lastValue)/(_timeStep* 0.001);

  float output =  (this->_p * error) +
          (this->_i * error * this->_timeStep * 0.001) +
          (this->_d * der);

  this->_lastValue = error;
  return output;
}
```

Listing 11: Zaimplementowany w C++ regulator PID

Do wyznaczenia części różniczkującej potrzebna wartość błędu z poprzedniego wywołania pętli a ta zapisywana jest do zmiennej prywatnej klasy lastValue. Całkowanie zrealizowane jest poprzez pomnożenie przez krok dyskretyzacji. Stałe regulatora ustawiane są poprzez wywołanie konstruktora klasy.

Tak zaimplementowany regulator używany jest do wyznaczenia sygnału PWM, bezpośredniego sterującego układem L298n a ten silnikami.

```
if(abs(setSpeed) - 1 > 0)
{
  pidToPwm = (int) this->calcOutput(abs(this->
   getCalculatedEngineRadialSpeed()), abs(setSpeed));
  if(setSpeed < 0)
  {
    pwmCh = 1;
  }
  else
    pwmCh = 0;

  pidToPwm += ledc_get_duty(LEDC_LOW_SPEED_MODE, this->_chanels[pwmCh])
   ;
  pidToPwm = std::min(pidToPwm, 8192);
  pidToPwm = std::max(pidToPwm, 0);
}

// ustawienie wyjsc
ledc_set_duty(LEDC_LOW_SPEED_MODE, this->_chanels[pwmCh], (int)pidToPwm
   );
ledc_update_duty(LEDC_LOW_SPEED_MODE, this->_chanels[pwmCh]);

// ustawienie drugiego kanalu
ledc_set_duty(LEDC_LOW_SPEED_MODE, this->_chanels[(pwmCh==0)?1:0], 0);
ledc_update_duty(LEDC_LOW_SPEED_MODE, this->_chanels[(pwmCh==0)?1:0]);
```

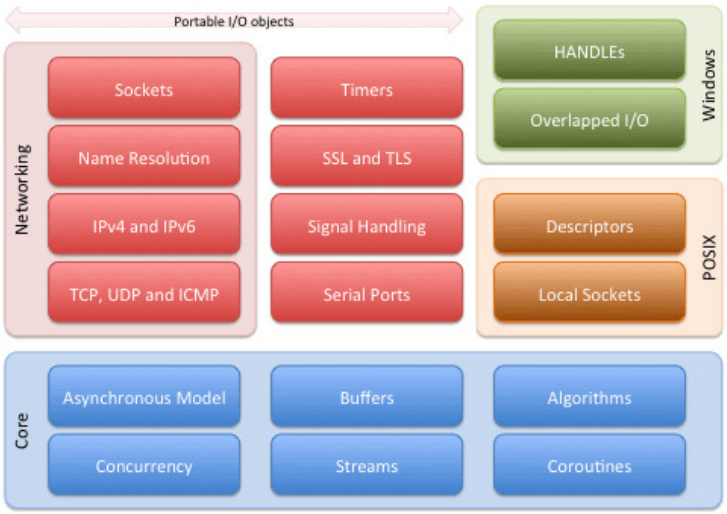
Listing 12: Wyznaczenie modulacji PWM

W pierwszej kolejności sprawdzana jest zadana prędkość, jeżeli ta jest zbyt niska to na piny wystawiane są bezpośrednio stany niskie. Jeżeli ustawiona prędkość jest poprawna to wyznaczamy wyjście regulatora. Sterowanie kierunkiem obrotów odbywa się poprzez znak zadanej prędkości a więc regulator otrzymuje wartości bezwzględne. Wyjście regulatora PID sumowane

jest z aktualną nastawą. Wyjścia pwm skonfigurowane są w częstotliwości 5kHz i rozdzielczości 13bitów. Przed wysłanie nastaw do kontrolera pwm, te są obcinane do obsługiwanych zakresów (tj. 0 - 8192).

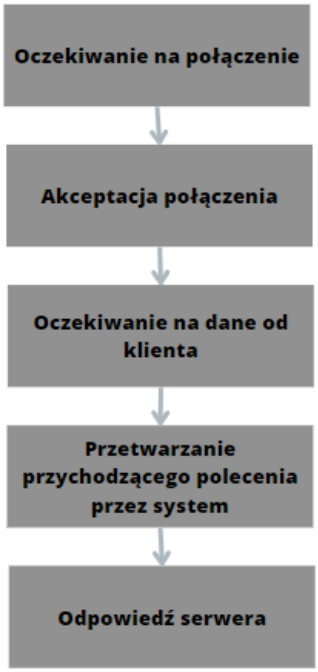
5.4.2. Serwer TCP

Po podłączeniu zasilania i uruchomieniu systemu robot oczekuje na polecenia wysłane do robota poprzez protokół internetowy TCP. Serwer został napisany w oparciu o bibliotekę ASIO, pozwalającą na asynchroniczną obsługę wejścia i wyjścia (w tym sieci).



Rysunek 5.14: Schemat biblioteki ASIO [5]

Biblioteka została napisana w C++ i pracuje w oparciu o standard POSIX wspierany również przez wykorzystywane ESP32. Do akceptacji przychodzących połączeń została napisana klasa Server. Konstruktor przyjmuję referencje do kontekstu utworzonego w funkcji głównej programu. Uruchomienie kontekstu jest ważną częścią biblioteki ASIO, przetwarzane są w niej wszystkie asynchroniczne operacje. Aby zapewnić jak najlepszy czas przetwarzania, kontekst uruchomiony jest w oddzielnym wątku przyłączonym do drugiego fizycznego rdzenia.



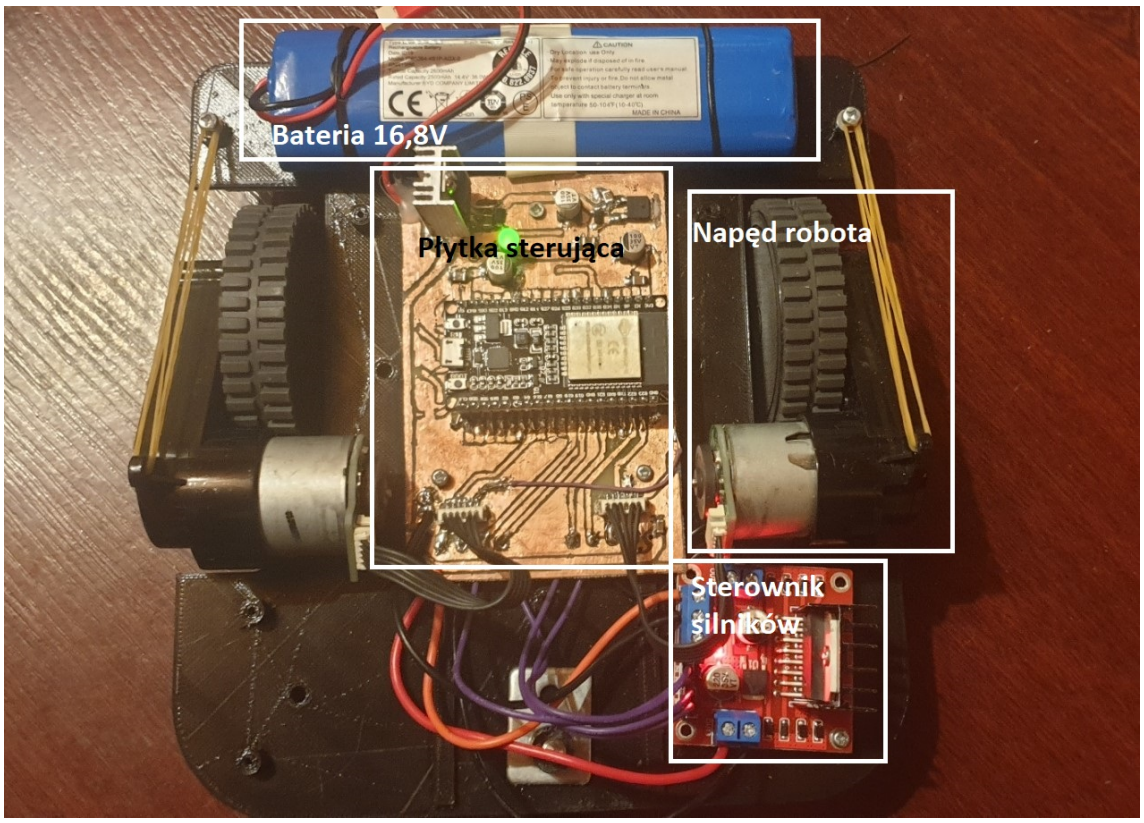
Rysunek 5.15: Schemat działania serwera TCP

Po zaakceptowaniu przychodzącego połączenia, obiekt połączenia dodawany jest do specjalnej tablicy przechowującej wszystkich klientów. Dane mogą być równocześnie przetwarzane w kilku miejscach a więc używane są dzielone wskaźniki, które automatycznie usuwają dane jeżeli nikt z nich nie korzysta. Na każdym aktywnym połączeniu prowadzony jest nasłuch, dzięki czemu możemy mieć równocześnie podłączony program do generowania ścieżki i drugi pozwalający na podgląd parametrów robota. Klient wysyła polecenia w formie tekstowej, te przekazywane są do klasy systemowej odpowiednio interpretujące ich przeznaczenie. Jeżeli komenda tego wymaga (np. zwrócenie prędkości obrotowej silników) to dane są odsyłane do klienta w postaci tekstu i oddzielonych przerwą liczb.

Należy pamiętać że podczas testowania i łączenia z różnymi sieciami, serwer dhcp będzie przydzielał różne adresy ip i należy to uwzględnić w programie łączącym się z robotem.

Wykonany robot

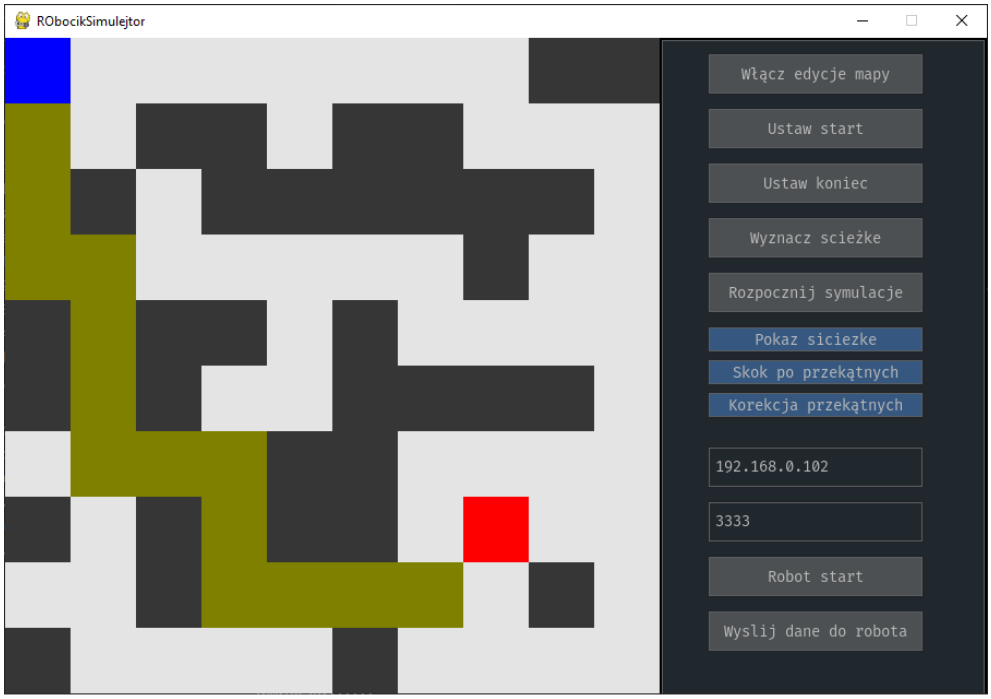
Poniżej widać zbudowanego i uruchomionego robota. Na zdjęciu możemy zauważyć że napęd został przyczepiony przy pomocy gumek, ponieważ oryginalne sprężyny tworzyły zbyt dużą siłę i robot stał krzywo.



Rysunek 5.16: Zbudowany robot

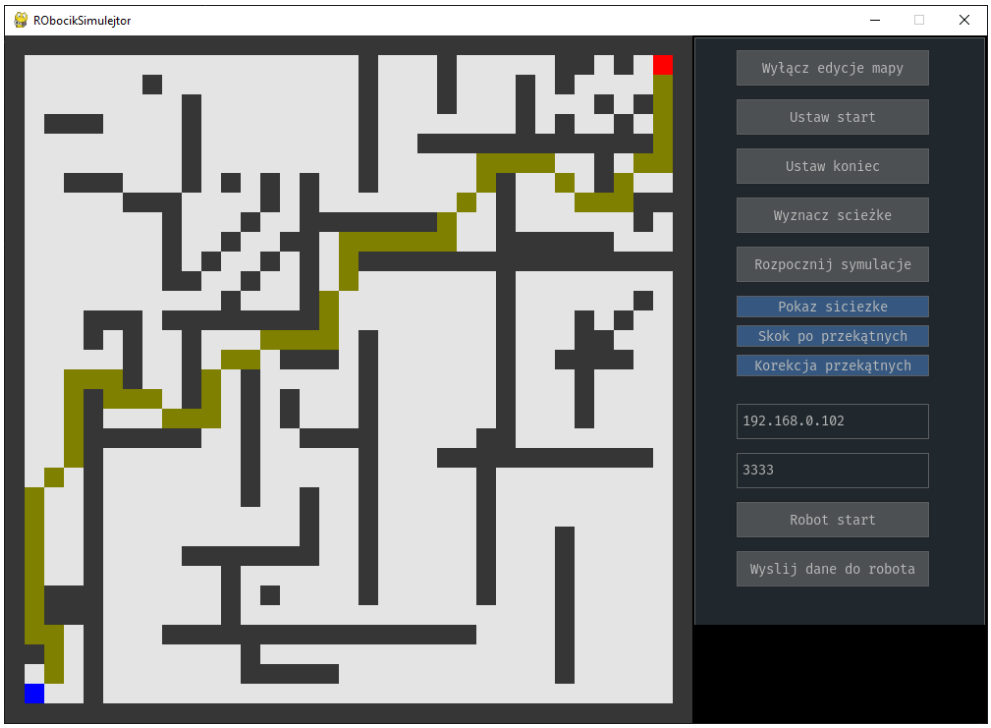
6. Przeprowadzone testy

6.1. Test poprawności trasowania

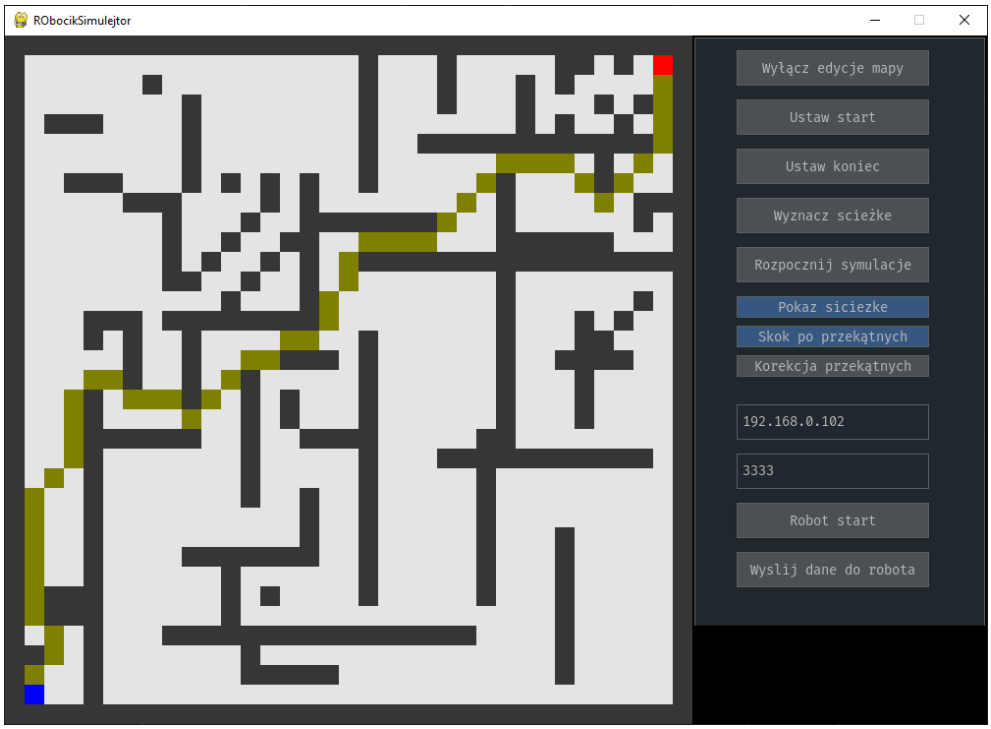


Rysunek 6.17: Podstawowy test trasowania

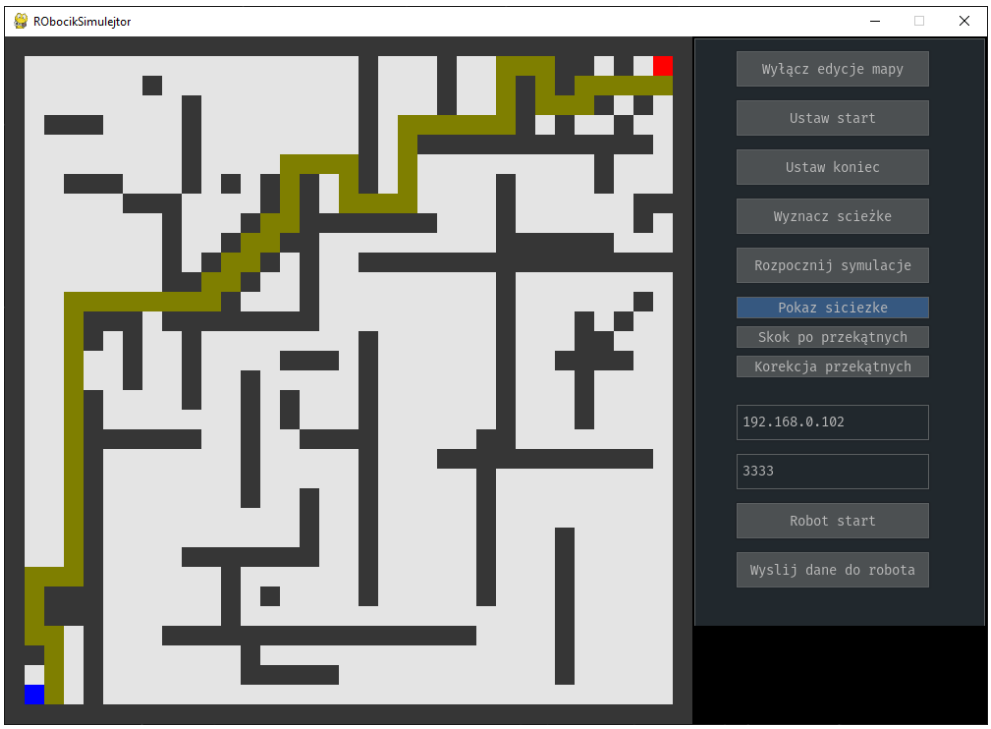
6.2. Wpływ ustawień na wyznaczoną ścieżkę



Rysunek 6.18: Wyznaczona ścieżka, włączona korekcja i przejście po przekątnej



Rysunek 6.19: Wyznaczona ścieżka, korekcja przekątnych jest wyłączona



Rysunek 6.20: Najkrótsza trasa, wyłączone przejście po przekątnych

6.3. Sprawdzenie wpływu funkcji heurestycznej na ścieżkę

6.4. Test komunikacji z fizycznym robotem

Literatura

[1] https://docs.espressif.com/projects/esp-idf/en/latest/esp32/get-started/index.html. Dostęp 04.01.2023.

[2] https://medium.com/dish/75-years-of-innovation-shakey-the-robot-385af2311ec8 Dostęp 04.01.2023.

[3] https://www.pygame.org/docs/ Dostęp 06.01.2023

[4] https://pygame-gui.readthedocs.io/en/latest/ Dostęp 06.01.2023

[5] https://think-async.com/Asio/ Dostęp 06.01.2023

[6] https://www.cs.cmu.edu/ motionplanning/lecture/lec20.pdf Dostęp 10.01.2023

[7] https://www.mathworks.com/discovery/slam.html Dostęp 10.01.2023