Painters

<niegdysiejszy tytuł roboczy>

Autorzy:  
Bartłomiej Strózik  
Adam Szreter

# Założenia i cele projektu

Celem projektu było zaprogramowanie pary robotów mobilnych tak, aby zdołały przejechać narysowaną w programie graficznym i wgraną do ich pamięci trasę, skutecznie unikając kolizji. Podczas realizacji zadania roboty miały opierać się na znajomości własnej pozycji (na podstawie pomiaru obrotu kół za pomocą enkoderów) i komunikacji radiowej (w technologii Bluetooth Low Energy).

# Zasoby

Nasze roboty oparliśmy o następujące zasoby sprzętowe:

* płytka ESP32 DevKitC V4 (z wbudowanym modułem bluetooth) (dokumentacja: [ESP32](https://www.espressif.com/sites/default/files/documentation/esp32_datasheet_en.pdf), [DevKitC-V4](https://www.espressif.com/sites/default/files/documentation/esp32-devkitc-v4_reference_design.zip))
* podwozie Magician Chassis z silnikami Dagu DG01D-A130 GearMotor (dokumentacja: [silniki](https://botland.com.pl/pl/index.php?controller=attachment&id_attachment=363))
* sterownik silników Pololu TB6612FNG ([dokumentacja](https://botland.com.pl/pl/index.php?controller=attachment&id_attachment=83))
* moduł enkoderów z robota SparkFun RedBot ([dokumentacja](https://botland.com.pl/pl/index.php?controller=attachment&id_attachment=520))

Do programowania robotów wykorzystaliśmy język c++ i środowisko Arduino, w tym Arduino IDE 1.8.9.

# Repozytorium

## Adres

Projekt został upubliczniony przez umieszczenie go w publicznym repozytorium na GitHubie, dostępnym pod adresem: <https://github.com/tyrrr-aj/Painters.git>

## Struktura repozytorium

W głównym katalogu projektu znajdują się następujące pliki i katalogi:

* plik *Painters.ino* – jest to standardowy plik źródłowy platformy Arduino, zawierający główny kod wykonania programu (funkcje *setup* i *loop*).
* plik *Pins.h* – opisuje on odwzorowanie logicznych funkcji pinów w kodzie na ich fizyczne numery (zależne od zestawienia okablowania konkretnego egzemplarza robota)
* katalog *src* - zostały w nim umieszczone wszystkie biblioteki specyficzne dla projektu
* katalog *data* – przechowuje on pliki z trasami, jakie może przejechać robot

# Fizyczna konstrukcja robotów

Każdy z robotów użytych w projekcie jest zbudowany na podwoziu Magician Chassis. Niezbędne połączenia elektryczne zrealizowaliśmy na płytkach stykowych (*breadboard*), choć docelowo najlepiej byłoby zastąpić je układami lutowanymi, np. na uniwersalnych płytkach lutowniczych. Ograniczają się one de facto do połączenia odpowiednich pinów płytki ESP32 DevKitC z wejściami sterownika silników i rozprowadzenia zasilania, oprócz tego niezbędne jest podłączenie silników do wyjścia sterownika i enkoderów do pinów GPIO płytki. Przy podłączaniu silników, warto pamiętać, że w podwoziu Magician Chassis jeden z silników jest ustawiony „do góry nogami” – należy go wpiąć w sterownik na odwrót, niż wynikałoby to z dokumentacji (inaczej będzie się kręcił w przeciwną stronę, niż zakładamy).

Układ można zasilać na trzy sposoby – przez USB, stałym napięciem 5V lub stałym napięciem 3V. W praktyce oznacza to, że robot powinien być wyposażony albo w powerbank, albo koszyk na baterie – my wybraliśmy drugą z tych opcji, umieszczając w nim cztery akumulatory AA o napięciu 1.2V. Przy zasilaniu układu inaczej niż przez USB, warto pamiętać o dwóch rzeczach:

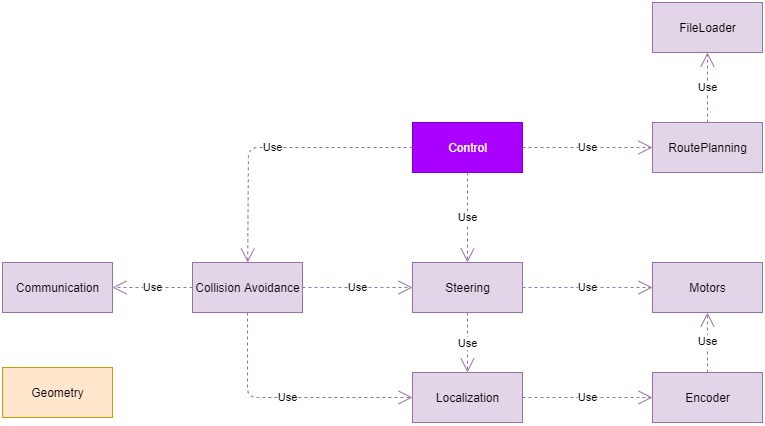
1. ESP32 DevKitC ma osobne piny do zasilania każdym z tych napięć
2. W każdym momencie, płytka może być zasilana z tylko jednego ze źródeł – w szczególności, przy programowaniu robota przez USB należy pamiętać o rozłączeniu zasilania bateryjnego (w naszym przypadku zapomnienie o tym nie spowodowało uszkodzenia płytki, ale nie znaczy to, że nie może)

Wybór metody zasilania jest dowolny, warto jednak pamiętać, że musi ono obsłużyć wykorzystane silniki – napięcie 3.3V może być do tego celu niewystarczające.

Przydatna dokumentacja:

* Omówienie, których pinów ESP32 można używać w jaki sposób, szczególnie użyteczna jest podsumowująca tabelka: <https://randomnerdtutorials.com/esp32-pinout-reference-gpios/>  
  (numery pinów na płytce DevKitC odpowiadają tym z tabelki/dokumentacji ESP32)
* Schemat wyprowadzeń sterownika silników TB6612FNG: <https://botland.com.pl/pl/sterowniki-silnikow-moduly/32-pololu-tb6612fng-dwukanalowy-sterownik-silnikow-135v1a.html>

# Struktura projektu



Projekt został zbudowany modułowo. Głównym komponentem jest klasa **Control**, pełniąca zasadniczy nadzór nad robotem. Jej zadaniem jest prowadzenie robota po zadanej trasie (otrzymanej z modułu **RoutePlanning**) – do czego wykorzystuje moduł **Steering**, i utrzymywanie poprawnego działania modułu **CollisionAvoidance**. Moduł ten, jak sama nazwa wskazuje, odpowiada za monitorowanie, czy robotowi nie grozi kolizja, i reagowanie, jeśli dojdzie do takiej sytuacji. Wykorzystuje w tym celu znajomość lokalizacji robota (**Localization**) i komunikację z partnerem (**Communication**).

Do kontroli nad ruchem robota zarówno klasa **Control**, jak i **CollisionAvoidance** wykorzystują moduł **Steering**. Udostępnia on funkcjonalności takie, jak jazda do określonego punktu albo zatrzymanie robota i późniejsze wznowienie ruchu. Wykorzystuje on informacje o bieżącej lokalizacji i kursie (**Localization**) i wydaje polecenia silnikom (**Motors**).

Klasa **Localization** monitoruje aktualną względną pozycję i obrót robota – punkt odniesienia stanowi ustawienie początkowe. Posługuje się ona odczytami z enkoderów (**Encoder**). Moduł **Encoder** odpowiada za monitorowanie sygnałów z fizycznych enkoderów i udostępnianie liczby zliczonych ticków. Do realizacji tego zadania potrzebuje informacji o aktualnym kierunku ruchu, którą otrzymuje bezpośrednio od modułu **Motors** – odpowiedzialnego za kontrolę nad silnikami.

Zadaniem klasy **RoutePlanning** jest odczytanie trasy opisanej w pliku z programu graficznego i zbudowanie na jej podstawie uporządkowanej listy punktów, które robot musi odwiedzić. Do załadowania pliku z pamięci wykorzystuje klasę **FileLoader**.

Moduł **Geometry** jest użytkowym modułem, udostępniającym różnorodne funkcje związane z planimetrią, niecharakterystyczne dla projektu, ale niezbędne w wielu miejscach programu. Wiele z modułów projektu korzysta z różnych składowych modułu **Geometry** – powiązania te zostały pominięte na schemacie, aby nie zaciemniać obrazu.

# Omówienie poszczególnych modułów

## Control

### Opis modułu

Klasa *Control* jest główną klasą, spaja wszystkie części projektu ze sobą, można powiedzieć, że łączy 3 składowe: składową rozplanowania trasy dla podwozia - *RoutePlanning*, składową *Steering* ze sterowaniem oraz nadzorem nad poruszaniem się robota oraz ostatnią składową: *CollisionAvoidance* unikania kolizji poprzez komunikację.

### Działanie modułu

Do modułu dostarczana jest trasa jako kolekcja punktów typu *Point* i wskaźnik do obiektu biblioteki z instrukcjami obsługi kolizji *Collision Avoidance*. Klasa *Control* na ich podstawie obsługuje sterowanie poprzez przekazywanie mu kolejnych destynacji. W programie jest wywoływana tylko jedna metoda tego modułu.

## Steering

### Działanie modułu

Odpowiedzialnością klasy *Steering* są matematyczne obliczenia służące do obliczenia kierunku i drogi jaką ma przebyć robot oraz wyznaczenia kierunku obrotu jaki ma on wykonać. Wykorzystuje się do tego własności wektorów i punktów matematycznych oraz geometrię analityczną, ściślej odejmowanie, dodawanie, tworzenie wektorów i iloczyn wektorowy. Posiadaną przez podwozie pozycję pobiera się używając klasy *Localization*.

Drugą odpowiedzialnością jest zmapowanie wyników powyższych obliczeń na odpowiednie instrukcje dla silników, gdyż moduł ten współpracuje z klasą *Motors*. Polega ono na uaktywnieniu odpowiedniej akcji silników i kontrolowania ruchu do momentu osiągnięcia celu. Podczas jazdy za każdym razem aktualizuje się pozycję w *Localization*, która służy potem do sprawdzenia osiągnięcia celu.

### Proces tworzenia

Według początkowej koncepcji klasa *Steering* miała przekazywać liczone kąty i odległości do agregowanej klasy *Chassis*, w której mielibyśmy zarówno lokalizację jak i silniki, a poruszanie się polegałoby na przeskalowaniu odległości lub kąta na czas obrotu. Oczywiście szybko się zreflektowaliśmy, ze jest to niewykonalne, a przede wszystkim ­– za dużo odpowiedzialności w jednej klasie.

Szybko zmieniliśmy klasę *Chassis* na klasę *Localization*, którą ograniczyliśmy do samego przechowywania pozycji. Klasa *Steering* stała się w pełni odpowiedzialna za sterowaniem silnikami ­– co jest dla niej bardzo naturalne, bo to ona wie dokładnie gdzie jest, dokąd zmierza i jak szybko się porusza podwozie. Cecha ta umożliwia zatrzymywanie podwozia albo obracanie nim zależnie od sytuacji.

Obliczenia matematyczne jako pierwszy etap „działania” tej klasy były również przeniesione w inne miejsce (moduł „*Angle*”) i to rozwiązanie było być może lepsze od obecnego. Pozwoliłoby na przeniesienie wszelkich wzorów matematycznych i uzyskiwanie wektorów przemieszczenia, czy kątów z jednego miejsca. Zrezygnowaliśmy z tej klasy, gdyż w naszym przypadku wzory były bardzo nieskomplikowane, a najwięcej pracy w tej klasie było poświęcone liczeniu kąta, które później okazało się nie być konieczne.

Ostatecznie klasa *Steering* przechowuje dwa wektory: wektor przemieszczenia (pomiędzy punktem źródłowym, a docelowym) wektor położenia celu (czyli zaczepiony w punkcie (0,0) ukł. wsp. i kończący się na punkcie docelowym). Dzięki pierwszemu ustala się zwrot podwozia, do którego się dąży, a dzięki drugiemu przeprowadza się jazdę do momentu osiągnięcia celu.

### Działanie modułu

Klasa *Motors* tworzy warstwę abstrakcji nad silnikami. Udostępnia typowy zestaw metod, reprezentujących możliwe polecenia dla silnika – obrót do przodu/do tyłu z zadaną mocą i zatrzymanie (z aktywnym hamowaniem – *stop()* i bierne wyłączenie silnika – *coast()*). Detale obsługi silników są ukryte w prywatnych metodach, które można podmienić w razie użycia innego sprzętu.

Opcjonalną odpowiedzialnością obiektu *Motors* jest przekazywanie obiektowi *Encoder* informacji o tym, w którą stronę aktualnie kręci się dane koło – aby to robił, wystarczy wywołać na nim metodę *addEncoder()*.

### Proces tworzenia

Kluczem do łatwego napisania tej biblioteki okazał się wybór odpowiedniego fizycznego sterownika silników – użyty przez nas TB6612FNG pozwala sterować każdym z silników za pomocą trzech pinów. Dwa z nich (oznaczone tu jako IN1, IN2) należy ustawić w stan wysoki/niski, ustawiając w ten sposób kierunek obrotu silnika, a na trzeci podać sygnał PWM[[1]](#footnote-1), sterujący mocą silnika.

Do generowania fali PWM wykorzystaliśmy wbudowane mechanizmy ESP32 – są one bardzo przystępnie opisane np. tutaj: <https://randomnerdtutorials.com/esp32-pwm-arduino-ide/>.

Naszą bibliotekę oparliśmy o oficjalną bibliotekę dla robota RedBot firmy SparkFun Electornics, opartego w znacznej mierze o podobne komponenty (w szczególności – ten sam sterownik silników). Można ją znaleźć tutaj: <https://github.com/sparkfun/RedBot/archive/master.zip> (po pobraniu i rozpakowaniu zipa, należy otworzyć pliki *Libraries/Arduino/src/RedBotMotors.h* i *Libraries/Arduino/src/RedBotMotors.cpp.*

### Ogólna uwaga do obsługi pinów

Wygodną obsługę pinów zapewniła nam biblioteka *Arduino.h*. Wszystkie powiązania fizycznych pinów z nazwami używanymi w kodzie definiuje plik *Pins.h* w katalogu głównym projektu

## Encoder

## Localization

## FileLoader

## RoutePlanning

### Opis modułu

Moduł *RoutePlanning* odpowiada za przetworzenie zbioru punktów wczytanych z pliku na uporządkowaną kolekcję (reprezentowanej przez std::vector z C++) tych samych punktów. Ma to na celu ułożenie trasy, po której będzie poruszał się robot.

### Działanie modułu

Klasa *RoutePlanning* agreguje obiekt FileLoadera, który umożliwia mu operacje na plikach. Jedną z odpowiedzialności jest tutaj sparsowanie pliku tekstowego wygenerowanego przez program graficzny do zaznaczania punktów w układzie współrzędnych do listy obiektów klasy *RpPoint*.

Do generowania wykorzystaliśmy darmową, wieloplatformą aplikację Wykresy z pakietu dynamicznego oprogramowania GeoGebra. Można je pobrać ze strony <https://www.geogebra.org/download?lang=pl>.

W planowaniu trasy korzysta się z metody *getPath()*, która tworzy obiekt klasy *Solver* i przekazuje mu kolekcję punktów.

**Działania Solvera**

*Solver* wybiera punkt, od którego ma zacząć, a następnie szuka punktów najbliższych mu w metryce euklidesowej.

Algorytm Solvera jest zoptymalizowany o obszary zagęszczenia punktów. Nie pozwala on na pozostawienie obszarów gęstych punktów, do których „kiedyś się wróci”. Z tego powodu wybór punktów zaczynany jest od wykrojenia obszaru o promieniu w odległości od obecnego punktu.

W wykrojonym obszarze buduje się trasę. Każdy następny punkt trasy to najbliższy w metryce euklidesowej punkt w stosunku do punktu obecnego. Z obszaru wybierane są wszystkie punkty, czyli zanim zakończy się etap związany z obszarem musi on być w całości pokryty.

Gdy wykrojony obszar zostanie pokryty, tworzy się kolejny obszar. Startujemy z ostatniego punktu z właśnie pokrytego obszaru.

## CollisionAvoidance

## Communication

### Działanie modułu

1. PWM – fala prostokątna, w której informację niesie stopień wypełnienia fali (ang. *duty cycle*) [↑](#footnote-ref-1)