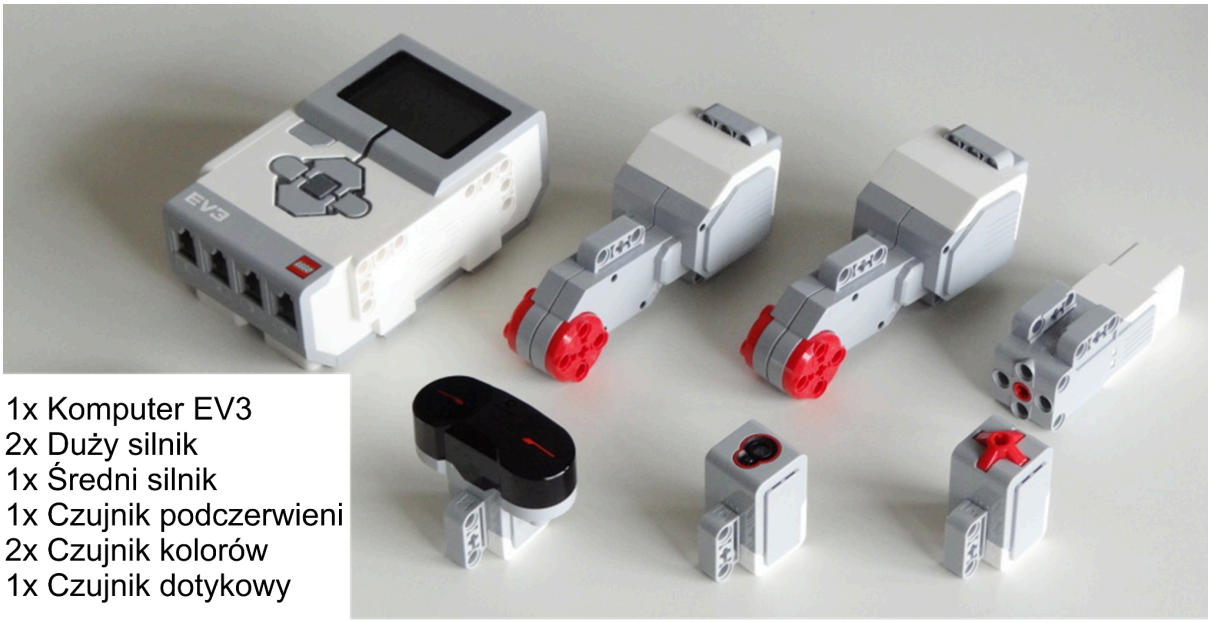
Wstęp do robotyki dla informatyków

Jakub Kowalczyk (318 676), Jan Filipecki (305 969)

Link do repozytorium: https://gitlab-stud.elka.pw.edu.pl/jkowalc5/wri-lab

Opis konstrukcji robota

Elementy

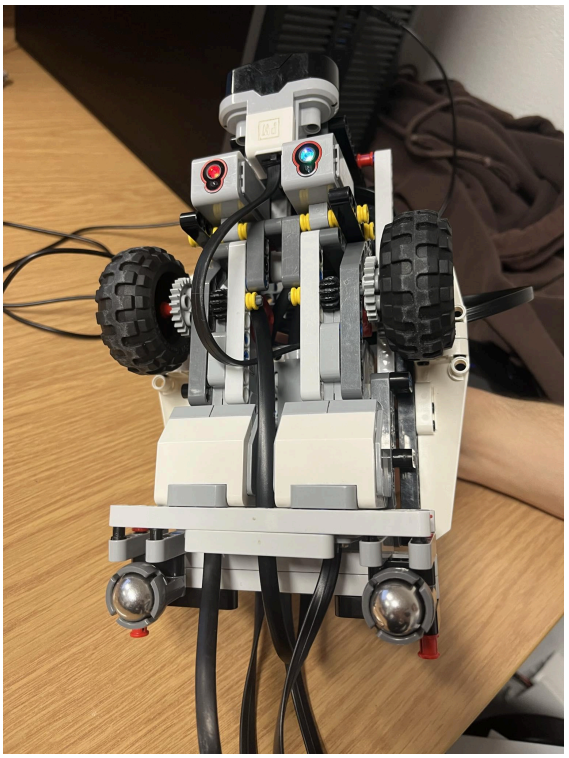


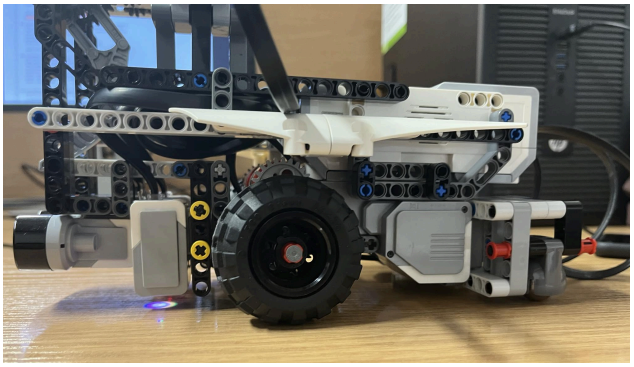
1x Komputer EV3
2x Duży silnik
1x Średni silnik
1x Czujnik podczerwieni
2x Czujnik kolorów
1x Czujnik dotykowy

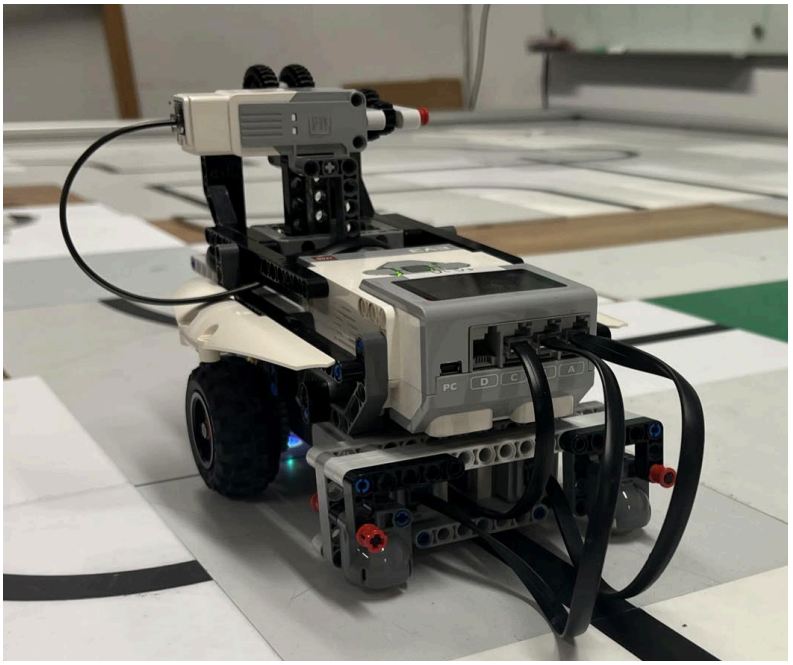
Sprawozdanie rozpoczynamy od krótkiego opisu dostępnych dla nas elementów zestawu LEGO Mindstorms EV3:

- **Komputer EV3** - To główna jednostka sterująca. Pozwala na podłączenie do czterech silników oraz 4 sensorów jednocześnie. Komunikacja z kostką odbywa się poprzez WiFi i interfejs ssh. Kostka wspiera oprogramowanie w języku Python oraz we własnym wysokopoziomowym języku Mindstorms, opartym na Pythonie.
- **Duży silnik** - Pozwala na budowanie robotów jezdnych i w porównaniu do średniego motora jest dużo silniejszy, co również sprawia że jest dość duży.
- **Średni silnik** - Przez to że jest dużo mniejszy pozwala na wykorzystanie go w charakterze serwomechanizmu.
- **Czujnik podczerwieni** - Jego głównym zastosowaniem jest pomiar odległości.
- **Czujnik kolorów** - Posiada dwa tryby - pomiaru kolorów (wartości składowych RGB), oraz pomiaru intensywności światła odbitego.
- **Czujnik dotykowy** - Działa jak zwyczajny przycisk binarny.

Baza jezdna

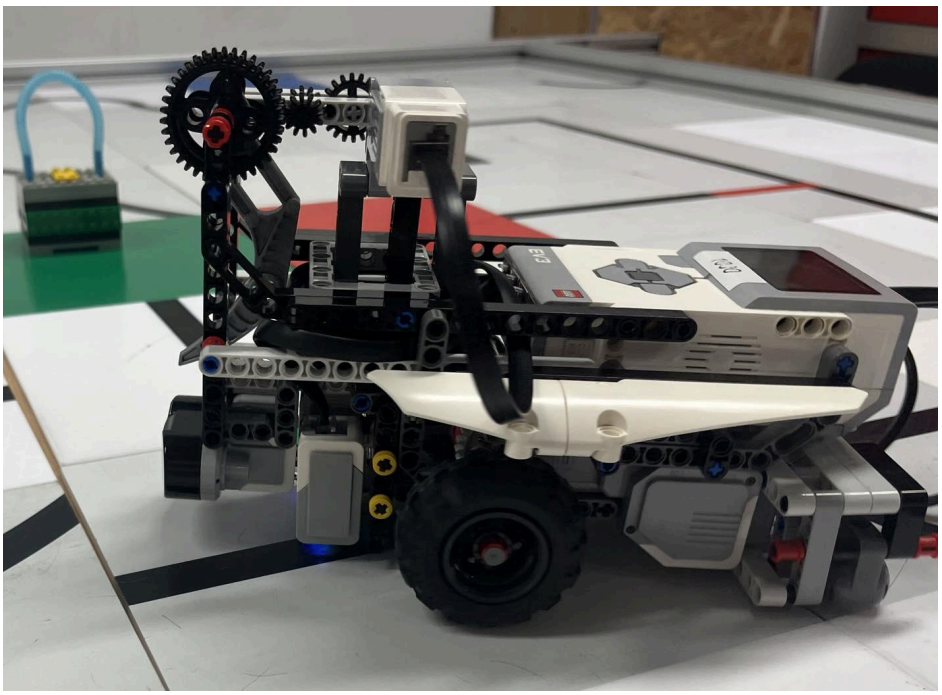


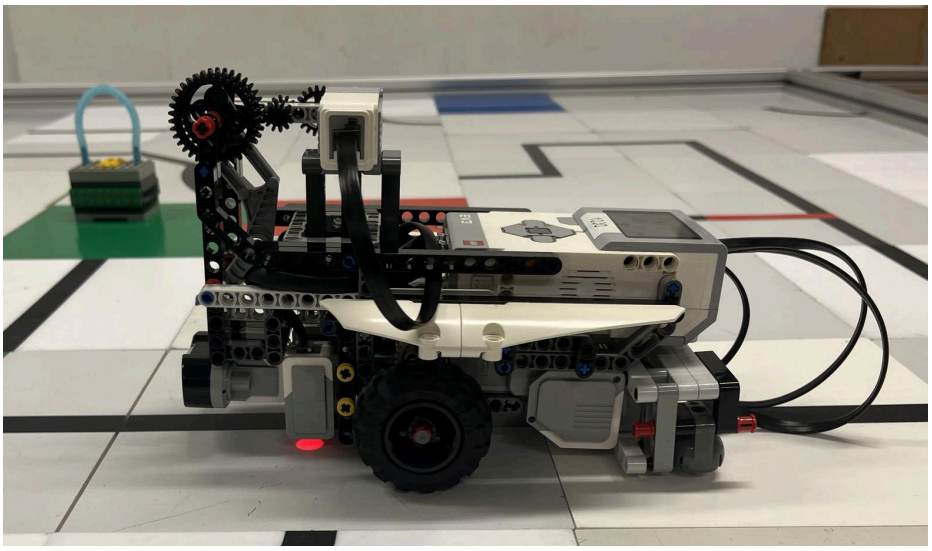




Ze względu na prostotę sterowania i budowy postanowiliśmy zaimplementować napęd różnicowy, inaczej zwany napędem gąsienicowym. Z początku silniki montowaliśmy po bokach robota, jednak po testach promienia skrętu, postanowiliśmy zamontować je pod klockiem sterującym, w celu zapewnienia odpowiedniej mobilności i kompaktowości robota. Wykorzystaliśmy również zębatki, by odpowiednio obniżyć osie kół, tak by robot nie tarł podwoziem o podłoże. Silniki zamontowaliśmy również tak, by osie były możliwie jak najbliżej czujników, o których mowa poniżej. By umożliwić robotowi swobodę skręcania oraz

stabilność, z tyłu konstrukcji zamontowaliśmy dwa omnikierunkowe łożyska kulkowe, które zapewniają robotowi dwa stopnie swobody. Całość została odpowiednio wzmocniona by podparcie było odpowiednie.





Czujniki

Nasz robot do sterowania wykorzystuje dwa czujniki kolorów, umieszczone na dole konstrukcji, skierowane w stronę ziemi. Zostały one zamontowane odpowiednio odległe od siebie, tak, by zapewnić dobrą czułość wykrywania linii, stąd zastosowanie małych żółtych “nakrętek”. Robot posiada również zamontowany z przodu czujnik odległości, by pozwolić mu na wykrywanie, czy znajduje się odpowiednio blisko przedmiotu, który chce podnieść.

Ramię

Aby zrealizować zadanie drugie, musieliśmy wyposażyć robota w mechanizm podnoszenia przedmiotu. Jako że obiekt, który został wyznaczony do przenoszenia posiada “ucho”, postanowiliśmy zbudować ramię ze swego rodzaju pazurem. Ramię jest napędzane silnikiem średnim z przekładniami użytymi, by zwiększyć moment obrotowy ramienia.

Zadanie 1. Podążanie wzdłuż linii (Linefollower)

Zadaniem robota jest przejechanie całej trasy po wyznaczonej linii. Opis planszy znajduje się na stronie [KNR Bionik](#).

Wersja naiwna

Początkowo zaimplementowaliśmy naiwną wersję podążania za linią bez regulacji PID. Program korzystał z dwóch czujników koloru ColorSensor (po jednym z lewej i prawej strony robota) do wykrywania linii. Z czujników odczytywane były wartości jasności podłoża, gdzie wartość 100 oznacza najjaśniejszy kolor, a 0 - najciemniejszy kolor. Przyjęliśmy wartość progową koloru białego jako odczyt jasności powyżej wartości 50, ze względu na nierównomierną “biel” czy zadrapania płytek na torze.

Algorytm wykonywania skrętów czy jazdy prosto został opisany w komentarzach kodu naive_linefollower.py Prędkość silników przy jeździe i skręcaniu oraz wartości progowe czujników zostały wyznaczone eksperymentalnie. Tak prosty 50-linijkowy program pozwolił na całkiem dobre podążanie za linią - robot mógł pokonać prostą trasę; miewał jedynie czasami problemy ze stabilnością oraz z pokonywaniem bardziej kłopotliwych fragmentów trasy takich jak ostre zakręty czy pętelki.

```
button = Button()
SPEED = SpeedPercent(20) # prędkość podstawowa
TURN_SPEED = SpeedPercent(-10) # prędkość do skrętu koła
LIGHT_THRESHOLD = 50  # minimalna wartość jasności dla białego
koloru
paused = False
def follow_line():
    global paused
    while True:
        if button.enter: # zatrzymanie robota
            paused = not paused
            if paused:
                tank_drive.off()
                print("Pausing...")
            else:
                print("Resuming...")
            sleep(0.5)

        if not paused:
```

```
            left_intensity =
left_color_sensor.reflected_light_intensity
            right_intensity =
right_color_sensor.reflected_light_intensity
            # Jeśli oba czujniki są na białym = czarna linia jest
pomiędzy, jedź prosto
            if left_intensity > LIGHT_THRESHOLD and
right_intensity > LIGHT_THRESHOLD:
                tank_drive.on(SPEED, SPEED)
            # Jeśli tylko lewy czujnik jest na białym = prawy
czujnik wykryje czarny kolor, skręć w prawo
            elif left_intensity > LIGHT_THRESHOLD:
                tank_drive.on(TURN_SPEED, SPEED)
            # Jeśli tylko prawy czujnik jest na białym = lewy
czujnik wykryje czarny kolor, skręć w lewo
            elif right_intensity > LIGHT_THRESHOLD:
                tank_drive.on(SPEED, TURN_SPEED)
            # Jeśli oba czujniki są poza białą planszą, zatrzymaj
się
            else:
                tank_drive.on(SPEED, SPEED)
        sleep(0.1)  # Opóźnienie, aby uniknąć zbyt częstego
odczytu
```

Wersja z regulacją PID

Metoda PID, czyli Proporcjonalny-Integracyjny-Różniczkujący, jest popularnym algorytmem regulacji stosowanym w automatyce. Wykorzystuje on trzy składowe: proporcjonalną (P), całkującą (I) i różniczkującą (D), aby skutecznie kontrolować wyjście systemu w odpowiedzi na różnice między wartością zadaną a rzeczywistą. Składowa proporcjonalna odpowiada za natychmiastową reakcję na bieżące odchylenie od wartości zadanej. Im większe odchylenie, tym większa korekta wykonana przez ten element. Składowa całkująca integruje odchylenia w czasie, co pozwala na eliminację błędów ustalonych. Działa ona na podstawie sumowania odchylenia w czasie i stosuje korektę proporcjonalną do całkowitego odchylenia w historii. Składowa różniczkująca przewiduje tendencje zmiany odchylenia w przyszłości. Działa ona na podstawie tempa zmian odchylenia, co pozwala na zapobieganie przewidywanym przeciążeniom systemu.

```
def follow_line(last_error=0, integral=0):
    left_color_sensor.mode = ColorSensor.MODE_COL_REFLECT
    right_color_sensor.mode = ColorSensor.MODE_COL_REFLECT
    sleep(SLEEP_AFTER_MODE_CHANGE)
    last_error, integral, pid_output = calculate_pid(
```

```
        last_error, integral, left_offset, right_offset)
    pid_scaled = pid_output * DEFAULT_SPEED / PID_SCALE
    left_speed = max(min(DEFAULT_SPEED - pid_scaled, MAX_SPEED),
-MAX_SPEED)
    right_speed = max(min(DEFAULT_SPEED + pid_scaled, MAX_SPEED),
-MAX_SPEED)

    tank_drive.on(left_speed, right_speed)  # Jazda prosto
    return last_error, integral
```

Powyższa implementacje różni się jednak trochę od standardowego podejścia, przez zastosowanie skalowania sterowania. **pid_scaled** jest służy jako dodatkowy parametr wpływający jedynie na stosunek wyjścia sterowania do domyślnej prędkości, by ułatwić strojenie. Jeszcze jedną różnicą jest element ograniczający wyjście wartości sterowania poza ustalony zakres, tak by nie zachodziły bardzo duże wychylenia.

Zastosowanie PIDu w naszym projekcie znacznie zwiększyło płynność sterowania. Aby jednak umożliwić algorytmowi działanie musieliśmy odpowiednio nastroić jego parametry. Dobieraliśmy współczynniki metodą inżynierską, iteracyjnie testując wpływ każdej zmiany, podążając za wskazaniami metody Zieglera-Nicholsa.

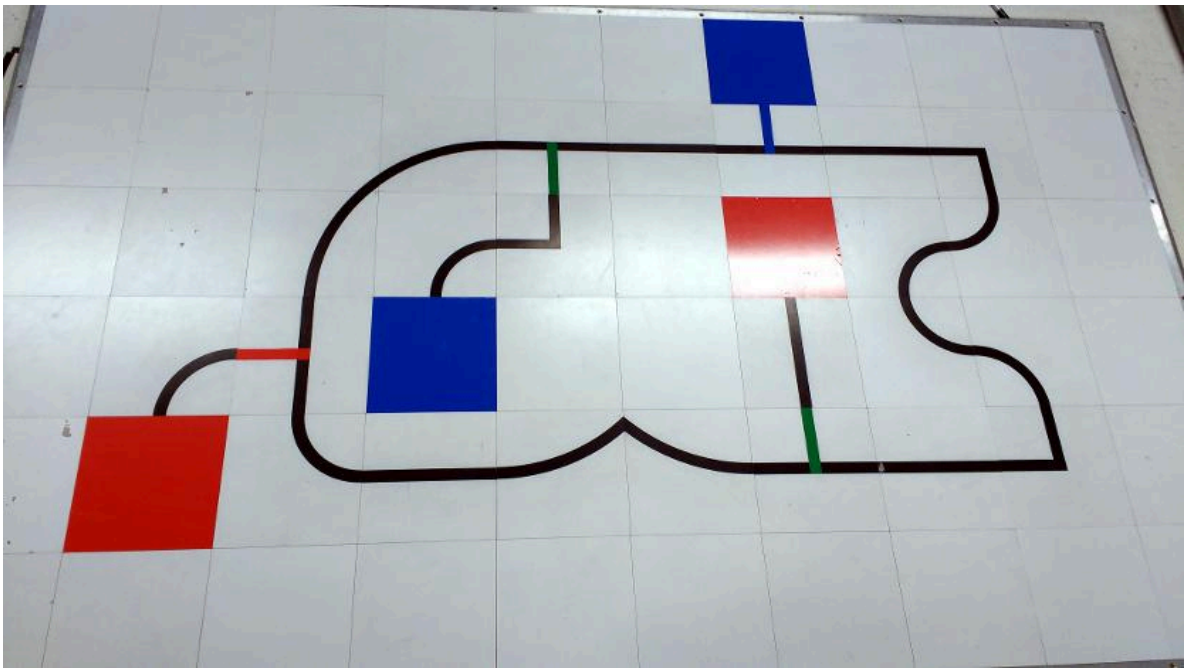
Podczas strojenia parametrów, zdaliśmy sobie również sprawę z tego, że czujniki po pierwsze nie są odpowiednio skalibrowane, czyli tym samym nie dokonują pomiarów w taki sam sposób, a po drugie ich wadliwości nie są deterministyczne, czyli z każdym uruchomieniem możemy otrzymać inne wychylenia. W związku z tym zaimplementowaliśmy poniższy algorytm kalibracji.

```
def calibrate():
    left_color_sensor.calibrate_white()
    right_color_sensor.calibrate_white()
    sleep(SLEEP_AFTER_MODE_CHANGE)
    left_intensity = left_color_sensor.reflected_light_intensity
    right_intensity = right_color_sensor.reflected_light_intensity
    print("left = " + str(left_intensity))
    print("right = " + str(right_intensity))
    offset = (left_intensity - right_intensity) / 2
    return -offset, offset
```

Algorytm ten zakłada, że robot został postawiony początkowo (na czas kalibracji) na białej płytce, następnie zapisuje znalezione wartości dla każdego sensora jako wartość domyślną dla koloru białego, tym samym dodając wartość offsetu tak, by oba sensory “widziały” tą samą wartość dla białego. Warto również wspomnieć, że na początku kodu wołamy również domyślne funkcje kalibracji dla sensorów EV3, by zapewnić jeszcze większą dokładność pomiarów. Zastosowanie tego rozwiązania finalnie bardzo poprawiło jakość wykrywania krawędzi linii.

Zadanie 2. Transporter

Zadaniem robota jest przetransportowanie obiektów z punktów bazowych do punktów docelowych. Punkt bazowy i punkt docelowy oznaczone są przez kolorowe elementy planszy. Rozwidlenie do odpowiedniego koloru jest zaznaczone na czarnej linii trasy.

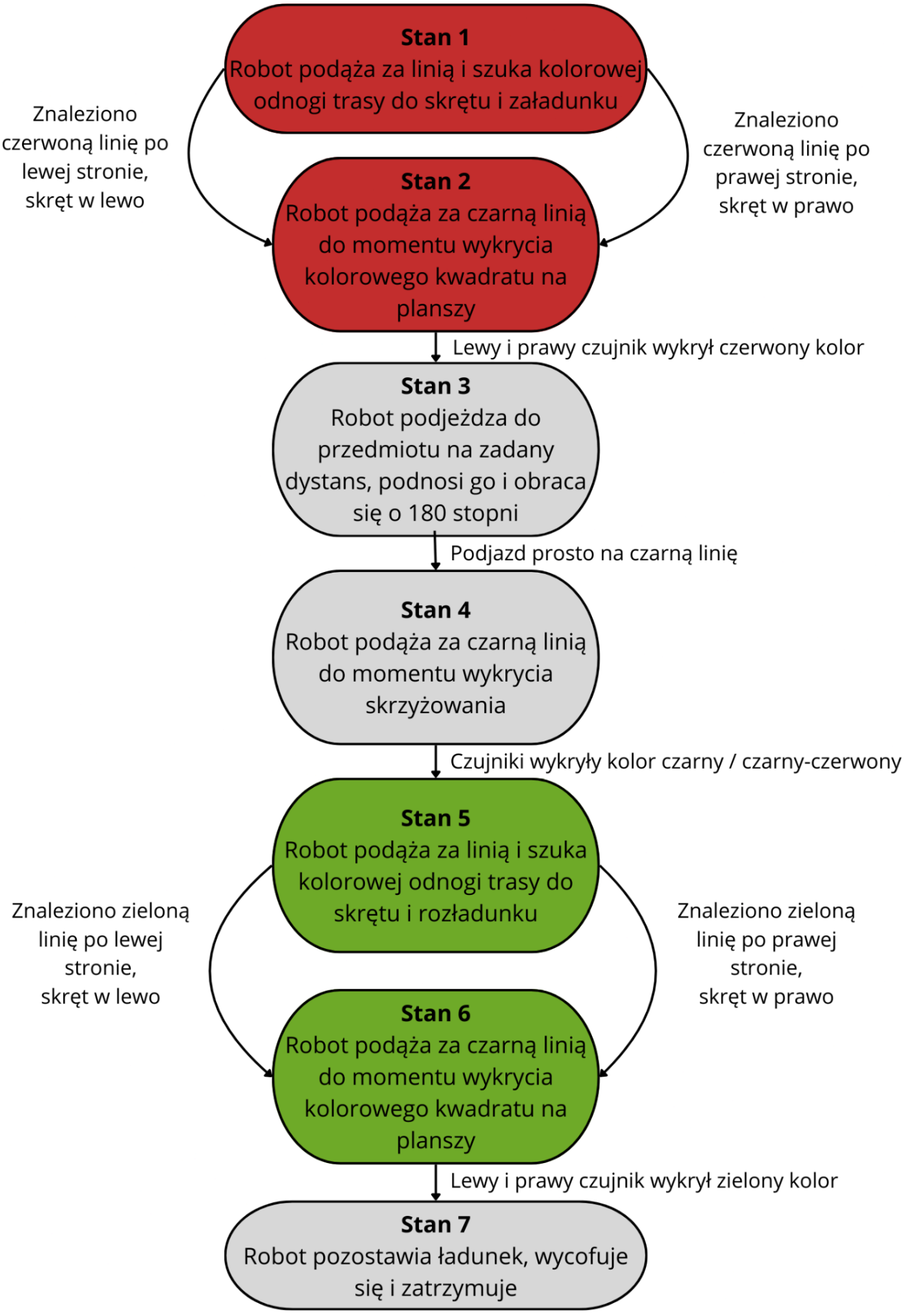


Założenia i struktura programu

Przy zadaniu Transporter wykorzystaliśmy jako podstawę funkcję *follow_line* bazującą na regulacji PID. Wykorzystaliśmy również funkcję *calibrate* opisaną wcześniej do podstawowej kalibracji odczytu jasności podłoża przy śledzeniu linii.

Na początku skupiliśmy się na dobraniu prawidłowych wartości i testowaniu rutynowych czynności robota koniecznych do transportu obiektu tj. podnoszenie i opuszczanie ramienia, skręty w lewo i w prawo o 90 stopni oraz obrót o 180 stopni po podniesieniu ładunku. Rozpoczęliśmy również testowanie rozpoznawania kolorów, co okazało się niebanalnym zadaniem.

Jednakże, przed męczeniem się z kalibracją czujników koloru, stworzyliśmy podstawę do wykonania całego zadania - automat stanów robota, który przedstawia poniższy schemat:



Schemat automatu stanów robota

Kalibracja czujników koloru

Na koniec zajęliśmy się dostosowaniem prawidłowego rozpoznawania kolorów przez czujniki. W podstawowej wersji programu wykorzystujemy biblioteczną zmianę trybów odpowiednio do sprawdzania jasności podłoża (włączane na początku funkcji *line_follower*) oraz do wykrywania kolorów (uruchamiane na początku funkcji *check_color*). Po każdej zmianie dodany jest mały czasowy offset, aby pozwolić na prawidłowe “wczytanie” trybu.

Funkcja *check_color* w pliku *transporter.py*:

```
def check_color():
    left_color_sensor.mode = ColorSensor.MODE_COL_COLOR
    right_color_sensor.mode = ColorSensor.MODE_COL_COLOR
    sleep(SLEEP_AFTER_MODE_CHANGE)
    return left_color_sensor.color, right_color_sensor.color
```

```
def follow_line(last_error=0, integral=0):
    left_color_sensor.mode = ColorSensor.MODE_COL_REFLECT
    right_color_sensor.mode = ColorSensor.MODE_COL_REFLECT
    sleep(SLEEP_AFTER_MODE_CHANGE)
[...]
```

Taki system doskonale radził sobie z rozpoznawaniem czerwonego koloru (jako koloru do załadunku), jednakże przy innych kolorach miał problemy. Żółty i zielony były trudno wykrywalne, a przy wykrywaniu niebieskiego koloru, często mylony był on z czarnym. Zdecydowaliśmy się zatem na wybór zielonego koloru jako koloru rozładunku, jednak przy niektórych próbach robot wymagał ponownego przestawienia, aby mógł wykryć kolor do skrętu.

Postanowiliśmy zatem przetestować wersję z odczytami wartości RGB i sprawdzaniem zakresu każdej składowej w celu przypisania odpowiednio wykrytego koloru. Kod tej wersji programu znajduje się w pliku transporter_rgb.py. Niestety, sprawdzanie zakresu trzech składowych rozregulowało nam PID ze względu na większą kosztowność obliczeniową, a nie przyniosło lepszych rezultatów. Nasze czujniki działały mocno niedeterministycznie przez co zdefiniowane zakresy wartości RGB musiałyby się “zazębiać”, a odczytywany kolor nie byłby jednoznaczny.

Funkcja *check_color* w pliku *transporter_rgb.py*:

```
    class Color(Enum):
        RED = [242, 41, 25]
        BLUE = [42, 119, 235]
        GREEN = [36, 150, 97]
        WHITE = [255, 255, 255]
        BLACK = [34, 36, 28]
        YELLOW = [255, 255, 60]
    COLOR_PICK_UP = Color.RED.value
    COLOR_DROP_OFF = Color.BLUE.value
```

```
    def check_color(target_color):
        left_color_sensor.mode = ColorSensor.MODE_COL_COLOR
        right_color_sensor.mode = ColorSensor.MODE_COL_COLOR
        sleep(SLEEP_AFTER_MODE_CHANGE)
        left_rgb = left_color_sensor.rgb
        right_rgb = right_color_sensor.rgb
        left_within_range = all(abs(left_rgb[i] -
target_color[i]) <= 20 for i in range(3))
        right_within_range = all(abs(right_rgb[i] -
target_color[i]) <= 20 for i in range(3))
        return left_within_range, right_within_range
```

Napotkane problemy i wnioski

Podczas testowania robota dużo czasu zajęło nam dobranie odpowiednich wartości PID. Jest to zadanie ogólnie nie proste, gdyż nawet małe zmiany w wartościach sterowania mogą drastycznie pogorszyć wydajność algorytmu. Poza tym wartości PID nie są od siebie niezależne, co jeszcze bardziej zwiększa trudność tego zadania.

Największym napotkanym problemem była jednak wadliwość czujników kolorów. Niezależnie od tego, czy wykorzystywaliśmy interfejs zapewniony przez bibliotekę EV3 do rozpoznawania kolorów, czy naszą własną implementację, czujniki działały względnie dobrze jedynie dla koloru czerwonego, co było niewystarczające.

W ogólności sądzimy, że jedynym mankamentem naszego projektu była trudność z pracą z czujnikami kolorów. Możliwe że warto byłoby jeszcze przetestować więcej instancji czujników, bądź dłużej poszukać odpowiedniego algorytmu kalibracji, na co niestety nie starczyło nam czasu.