POLITECHNIKA WARSZAWSKA

Dokumentacja WR

Arkadiusz Pawlukiewicz Tomasz Zieliński

8 kwietnia 2017

1 KONSTRUKCJA ROBOTA

Przy konstrukcji wykorzystaliśmy:

- Programowalną kostkę EV3
- Dwa czujniki światła
- Jeden średni silnik
- Dwa duże silniki
- Sensor soniczny

Mieliśmy wykonać zadanie w Pythonie albo c++, wybraliśmy Pythona. Program podzieliśmy na dwa pliki, jeden rozwiązujący dwa pierwsze zadania a drugi zadanie 3.

2 ZADANIE 1. LINE FOLLOWER

Naszym zadaniem było skonstruowanie robota, potrafiącego poruszać się wzdłuż linii. Trasa zbudowana jest z odcinków prostych, łagodnych zakrętów, ostrych zakrętów i skrzyżowań. Robot jedzie prosto przez skrzyżowania.

2.1 SENSORY ŚWIATŁA

Sensor mierzy natężenie światła odbitego od podłoża. Na podstawie natężenia światła odbitego, określają jakiego koloru była powierzchnia. (czarny kolor pochłania światło, a biały kolor odbija) Wartość odczytywaliśmy przy pomocy funkcji value(). Trzeba uważać żeby nie ustawić ich za wysoko, bo światło otoczenia miało by zbyt duży wpływ na odczyty. W pokonywaniu skrzyżowań konieczne było zastosowanie dwóch sensorów światła, żeby odróżnić je od zakrętów. Zastosowaliśmy skalowanie wartości odczytanych z czujników z powodu ich niedoskonałości. Na podstawie odczytów z sensorów wykrywamy gdzie jest linia.

2.2 Silniki

Silniki elektryczne o liniowej charakterystyce. Do ich sterowania używaliśmy głównie dwóch wygodnych funkcji:

```
run_forever(speed_sp), run_timed(time_sp, speed_sp)
```

run_forever(speed_sp) ustawia prędkość zadaną dla silnika
run_timed(time_sp, speed_sp) ustawia prędkość zadaną dla silnika i zeruje ją po zadanym czasie

2.3 Sensor Soniczny

Umieściliśmy go z przodu robota. Obracamy nim za pomocą średniego silnika przekazując siłę układem zębatek. Wartość odczytywaliśmy przy pomocy funkcji value(). Interpretujemy ja jako odległość najbliższego przedmiotu w linii prostej od sensora.

2.4 Regulator PID

Regulator w oparciu o odczyty z czujników światła manipuluje prędkością na silnikach. Jego celem jest utrzymanie wartości wyjściowej układu na z góry zadanym poziomie. Składa się z trzech członów.

```
def updateError(self):
        self.lastError = self.error
        scaledLeft = (lls.value() - 10) * 1.12
        scaledRight = (rls.value() - 7) * 1.43
        self.error = scaledRight - scaledLeft
```

2.4.1 Proporcjonalny

Człon który na podstawie sygnału podanego na wejście regulatora wytwarza sygnał sterujący proporcjonalny do sygnału. Wartość wyjściowa jest proporcjonalna do zmiany sygnału.

```
self.error
```

2.4.2 Całkujący

Wartość sterowania jest proporcjonalna do sumy uchybu. Człon ten na podstawie poprzednich zmian stara się zmienić sterowanie, by zapobiec przyszłym zmianom. Resetujemy sumę co jakiś czas.

```
def integral(self):
        self.elepsed = self.elepsed + 1
        if self.elepsed > 5:
                self.sum = 0
                self.elepsed = 0
```

```
        self.sum = self.sum + self.error
        return self.sum
```

2.4.3 RÓŻNICZKUJĄCY

Reaguje na zmiany uchybu a nie na jego wartość. Przy pomocy tego członu staramy się osiągnąć stan równowagi w jak najmniejszym czasie, jednakże gdy jego wartość jest zbyt duża możemy doprowadzić układ do niestabilności, zmiany będą tak duże że błąd zamiast się zmniejszać będzie się stale zwiększać.

```
def derivative(self):
        return self.error - self.lastError
```

2.5 DOBÓR WSPÓŁCZYNNIKÓW

Dobór współczynników przy pomocy metody prób i błędów, zazwyczaj zmienialiśmy więcej niż jeden parametr na raz (co nie jest dobrą praktyką) jego ostateczną postacią było :

```
4*self.proportional() + 3*self.derivative() + 0.1*self.integral()
```

Nasz robot bardzo szybko dochodził do stabilności mimo mało znaczącej części całkującej
Moc na silnikach wyznaczaliśmy jako 250 +c oraz 250-c na drugim silniku. Ograniczyliśmy ją do max 1000 , oraz -1000
Początkowo, robot nie radził sobie z zakręcaniem na zakrętach o kącie prostym, zastosowaliśmy zmiany:

- Nieznacznym zmniejszeniu jego podstawowej prędkości(z 200 na 150)
- Zwiększenie rozstawu czujników światła
- Przybliżenie czujników światła bliżej „centrum" robota, początkowo były one zanadto wyciągnięte

Umieściliśmy czujniki nieco bliżej osi. Jednakże potem zwiększyliśmy całe sterowanie, mnożąc je przez 1.7, robot zaczął radzić sobie na każdym zakręcie z maksymalną prędkością 250 (być może dlatego że baterie 4 mieliśmy z pudełka „puste baterie")

3 ZADANIE 2. OMINIĘCIE SZEŚCIANU

Zadanie takie samo jak poprzednio, tylko dodatkowo na trasie robota pojawia się sześcian.
Zadanie rozwiązaliśmy następującym algorytmem:
Sprawdzamy, w każdym obiegu pętli głównej (podczas jazdy wzdłuż linii z zastosowaniem PID) czy w odległości mniejszej niż 8, od sensora znajduje się dowolny obiekt. Jeśli coś się znalazło wtedy:

1. Obracamy sensorem o 90 Stopni w lewo

2. Wykonujemy obrót w prawo (do tyłu żeby nie uderzać w sześcian)
3. Jedziemy prosto, aż robot znajdzie się poza sześcianem (sprawdzamy to przy pomocy sensora)
4. Wykonujemy obrót w lewo
5. Jedziemy chwile prosto, aż robot znajdzie się koło sześcianu, po czym jedziemy dalej prosto odczytując wartości sensora, aż sensor przestanie dostrzegać sześcian.
6. Jedziemy prosto aż robot znajdzie się cały poza sześcianem.
7. Wykonujemy obrót w lewo (o jakieś 90 stopni)
8. Jedziemy Prosto aż znajdziemy linie, po czym wykonujemy obrót w prawo i wracamy do głównej pętli.

Rozwiązanie:
Nasze Rozwiązanie powinno poradzić sobie z sześcianami o większych wymiarach, jednakże skręty w prawo i w lewo nie zawsze były idealne i zdarzało nam się uderzać w sześcian
Początkowo nasz robot czasami nie zdawał sobie sprawy z istnienia sześcianu i po prostu w niego uderzał, był to problem z wykryciem sześcianu, spowodowany użyciem funkcji aproximity, zamiast value.

4 Zadanie 3. Odometria

Naszym zadaniem jest pokonanie trasy wczytanej do pamięci robota, w formie swego rodzaju mapy. Składa się ona ze współrzędnych kolejnych punktów do których musimy dotrzeć. Przy tym zadaniu powinniśmy wykorzystać odometrię. Jednakże zadanie zostało uproszczone tylko do kątów prostych, robot nie musi radzić sobie z pokonywaniem łuków i obrotu o kąt inny niż 90 stopni.

4.1 Błędy stałe

- Błędy w pomiarach, np. rozstawu kół
- Błędy wynikające ze sprzętu (np. wadliwy lewy silnik)
- Wynikające np. z braku dokładnych pomiarów

Z błędami stałymi można sobie poradzić doświadczalnie wyznaczając błędy po czym mnożąc zadane wartości przez stałą.

4.2 Błędy losowe

- Nierówna powierzchnia
- Poślizg robota

Błędy losowe są dużo trudniejsze do zwalczenia, aczkolwiek, jeśli poruszamy się w stałym środowisku można je skompensować. Podczas laboratorium wymnożyliśmy pewną stałą przez 1.02, co pozwoliło skompensować poślizg robota.

4.3 Planowany sposób rozwiązania

Będziemy liczyć liczbę tick'ów poszczególnych silników, na tej podstawie będziemy obliczać w jakim położeniu się znajdujemy. Jeśli znamy swoje położenie i położenie punktu do którego mamy dotrzeć, możemy określić w którą stronę oraz jak długo powinniśmy jechać. Po dojechaniu do poszczególnych punktów robot sprawdza czy powinien zakręcić , po czym jedzie prosto do celu.

4.4 Implementacja Rozwiązania

4.4.1 Plik Controller.py

Definiuje klasę Controller do zarządzania silnikami i odczytu ich parametrów. Umożliwia jazdę na wprost i obrót w zadanym kierunku.

4.4.2 Plik Map.py

Definiuje klasę Map, która przechowuje trasę dla robota w postaci listy punktów. Pozwala na sekwencyjny dostęp do kolejnych punktów. Po odczytaniu ostatniego następnym zwróconym będzie pierwszy. Wartości współrzędnych zapisane są względem podłoża.

4.4.3 Plik Odometry.py

Definiuje klasę Odometry. Zajmuje przechowywaniem i aktualizacją położenia robota. Aktualizacja następuje w oparciu o odczyty obrotu kół z silników. Na podstawie posiadanego położenia potrafi określić czy jesteśmy blisko punktu i czy jesteśmy ustawieni w kierunku zbliżania się do niego.

4.4.4 Plik Timer.py

Definiuje klasę Timer. Odpowiednie wykorzystanie jej funkcji pozwala na obieg pętli głównej w czasie nie krótszy niż zadany. Jej działanie oparte jest o *time.sleep()* i *time.time()* W ten sposób nie obciążamy nadmiernie procesora.

4.4.5 Plik Main.py

Zawiera pętle główną programu. Powołuje do życia obiekty klasy Controller, Map, Odometry, Timer, TouchSensor i zarządza ich współpracą.

4.5 Wykorzystane wzory

N_i liczba odczytanych impulsów z koła $i \in l, p$
C_i rozdzielczość enkodera - liczba impulsów na jeden obrót koła $i \in l, p$

$$\Delta\varphi_i = \frac{2\pi N_i}{C_i}, i \in l, p$$

$$\Delta p = \frac{r}{2}\left(\Delta\varphi_l + \Delta\varphi_r\right)$$

$$\Delta\theta = \frac{r}{d}\left(\Delta\varphi_l - \Delta\varphi_r\right)$$

$$\theta_{k+1} = \theta_k + \Delta\theta$$

$$x_{k+1} = x_k + \Delta p cos\left(\theta + \frac{\Delta\theta}{2}\right)$$

$$y_{k+1} = y_k + \Delta p sin\left(\theta + \frac{\Delta\theta}{2}\right)$$