



# **PHOTON ROBOT ARENA PLAYTHROUGH**

## **– Programing with Python**

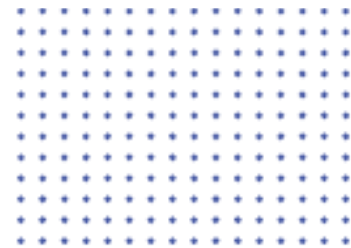
Lucas R / Marcin P / 2022 University of Technology





# SPIS TREŚCI

WSTĘP .....	3
[1] Omijanie przeszkody na drodze .....	4
[2] Pomiar areny.....	9
[3] Arena z przeszkodami .....	15
[4] Line Follower .....	22
[5] Generowanie obręczy w pozycji stacjonarnej – wstęp do mapowania .....	26
[6] Mapowanie areny .....	30
[7] Mapowanie całej areny wraz z przeszkodami .....	34
[8] Wyznaczanie trasy - propagacja fali .....	37
[9] Turniej sumo .....	41



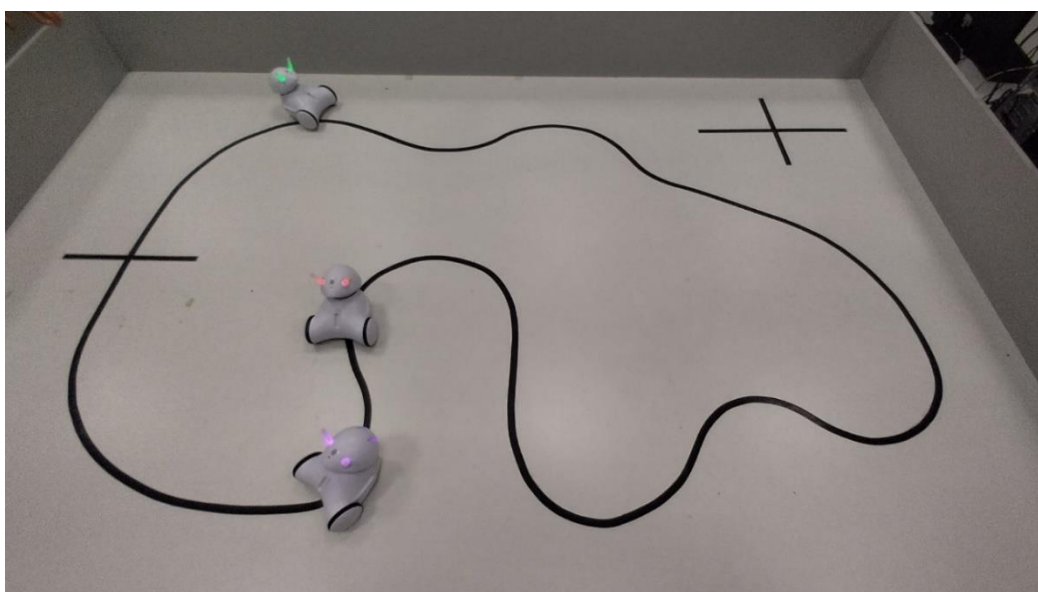
## O czym jest ten dokument?

W poniższym dokumencie omawiamy 9 zadań, które wykonaliśmy przy pomocy robota edukacyjnego Photon oraz areny podczas całego semestru zajęć laboratoryjnych na Politechnice, jest to między innymi podsumowanie całego semestru naszej pracy, nauki oraz przygotowywanych sprawozdań, a wszystko to jest złączone w całość w formie przystępnej instrukcji z wizualizacjami oraz kodem do własnego odtworzenia. Zadania były rozwiązywane w oficjalnym IDE producenta przy pomocy języka programowania Python z odpowiednią biblioteką Photon. Udostępniamy ten dokument z wszystkimi kodami Python za darmo dla wszystkich zainteresowanych w celach edukacyjnych i do własnego użytku.

## Czym jest robot Photon?

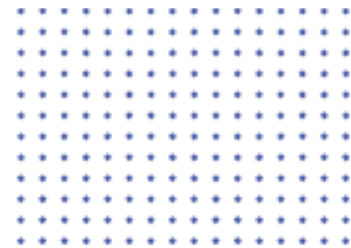
W skrócie, Photon to kompaktowy robot, który został zaprojektowany z myślą o wspomaganiu procesu nauczania programowania i robotyki w szkołach. Pochodzi z Polski, a stworzony został przez firmę Photon Education która oprócz samego robota tworzy również oprogramowanie IDE do Photona oferujące możliwość programowania w języku Python, Scratch oraz graficznego interfejsu z blokami. Choć Photon jest głównie skierowany do dzieci i młodzieży do nauki podstaw programowania, jego możliwości sięgają znacznie dalej. Zaawansowani użytkownicy mają szansę eksplorować bardziej skomplikowane zadania, takie jak mapowanie pomieszczeń czy integracja z sztuczną inteligencją i uczeniem maszynowym.

Strona producenta: <https://photon.education/pl/>



Rys.1. Arena, na której były wykonywane zadania

# [1] OMIJANIE PRZESZKODY NA DRODZE



## Zadanie do wykonania

Celem ćwiczenia jest napisać program, za pomocą którego robot Photon omijać będzie przeszkody. Robot ma okrążyć arenę jeżdżąc blisko ścian, zgodnie z ruchem wskazówek zegara, omijając jednocześnie napotkaną przeszkodę w postaci pudełka.

## Kod robota Photon

```
from photonrobot import *

def naprowadz():
    #Funkcja naprowadzająca robota do ściany
    for i in range(1):
        dis = []
        photon.rotate_left(30,50)
        DL1 = photon.get_distance_from_obstacle()
        photon.rotate_left(30,50)
        DL2 = photon.get_distance_from_obstacle()
        photon.rotate_right(60,50)
        DC = photon.get_distance_from_obstacle()
        photon.rotate_right(30,50)
        DR1 = photon.get_distance_from_obstacle()
        photon.rotate_right(30,50)
        DR2 = photon.get_distance_from_obstacle()
        dis.append(DL2)
        dis.append(DL1)
        dis.append(DC)
        dis.append(DR1)
        dis.append(DR2)
        photon.rotate_left(120,50)
        index = 0
        for j in dis:
            if j == min(dis) and index>0:
                photon.rotate_right(30*index,50)
                index+=1
        photon.go_forward(math.floor(min(dis)*0.5), 50)
    D = photon.get_distance_from_obstacle()
    if D >= 5:
        photon.rotate_right(90,50)
    elif D < 5:
        photon.go_backward(5,50)
        photon.rotate_right(90,50)

photon.change_color("purple")
czy_lewo = False
czy_prosto = False
w = False
cykl = 0
while True:
    distance = photon.get_distance_from_obstacle()
    if distance > 20:
        czy_prosto = False
    elif distance < 20:
        czy_prosto = True
    photon.rotate_left(90, 50)
    distance = photon.get_distance_from_obstacle()
    if distance < 20:
        czy_lewo = True
        photon.rotate_right(90, 50)
    else:
        czy_lewo = False
        photon.rotate_right(90, 50)
    if not czy_prosto and czy_lewo:
        photon.go_forward(20, 50)
    if not czy_prosto and not czy_lewo:
        photon.rotate_left(90, 50)
        photon.go_forward(20, 50)
    if czy_prosto and czy_lewo:
        photon.rotate_right(90, 50)
    if czy_prosto and not czy_lewo:
        photon.rotate_right(360, 50)
    cykl=cykl+1
    if cykl==15:
        naprowadz()
        cykl=0
```

## Realizacja zadania

Photon w każdej iteracji pętli dokonuje weryfikacji istnienia jakiejkolwiek przeszkody przed sobą i po swojej lewej stronie. Dane te są zapisywane w zmiennych "czy\_prosto" oraz "czy\_lewo". Zapis "czy\_prosto=True" oznacza, że robot wykrył przeszkodę przed nim (analogicznie jest w przypadku "czy\_lewo"). Możliwe są cztery przypadki:

### a. Przeszkoda znajduje się przed robotem i po jego lewej:



W takiej sytuacji robot powinien obrócić się w prawo o  $90^\circ$

### b. Przeszkoda znajduje się tylko przed robotem:



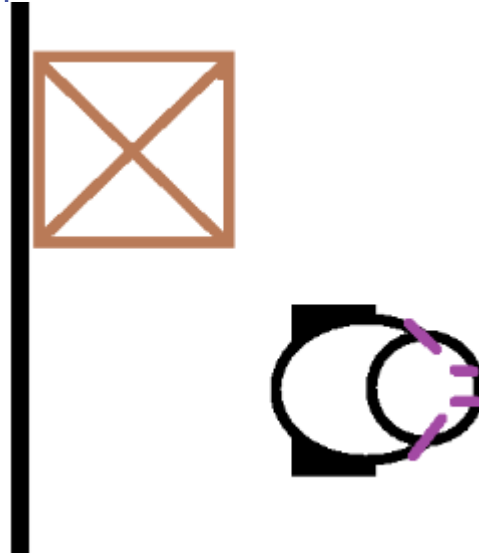
Teoretycznie do tej sytuacji nie powinno dojść, ale w przypadku otrzymania takiego wyniku robot obróci się o  $360^\circ$  by zasygnalizować zaistniałą sytuację.

### c. Przeszkoda znajduje się tylko po lewej stronie robota:



Robot powinien wykonać ruch do przodu na odległość 20cm

#### d. Robot nie wykrył przeszkód

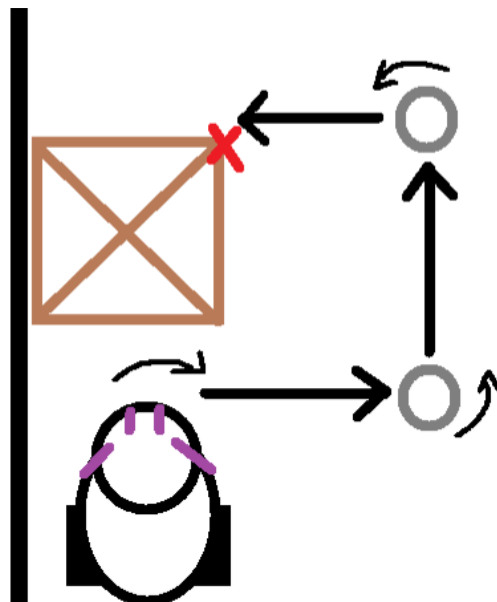


Robot powinien obrócić się w lewo i wykonać ruch do przodu na odległość 20cm

Dodatkowo co każde piętnaście iteracji pętli robot dokona korekty pozycji. Jest to konieczne z tego powodu, że ruch robota nie jest idealny i w momencie, gdy otrzymuje on komendę obrotu o  $90^\circ$  w rzeczywistości obraca się o wartość obarczoną jakimś błędem. Im więcej obrotów robot wykona tym bardziej będzie zbaczać z zaprogramowanej trasy a przy założeniu, że Photon sprawdza istnienie przeszkód w każdym cyklu programu dosyć szybko zboczy on ze swojej pierwotnej ścieżki.

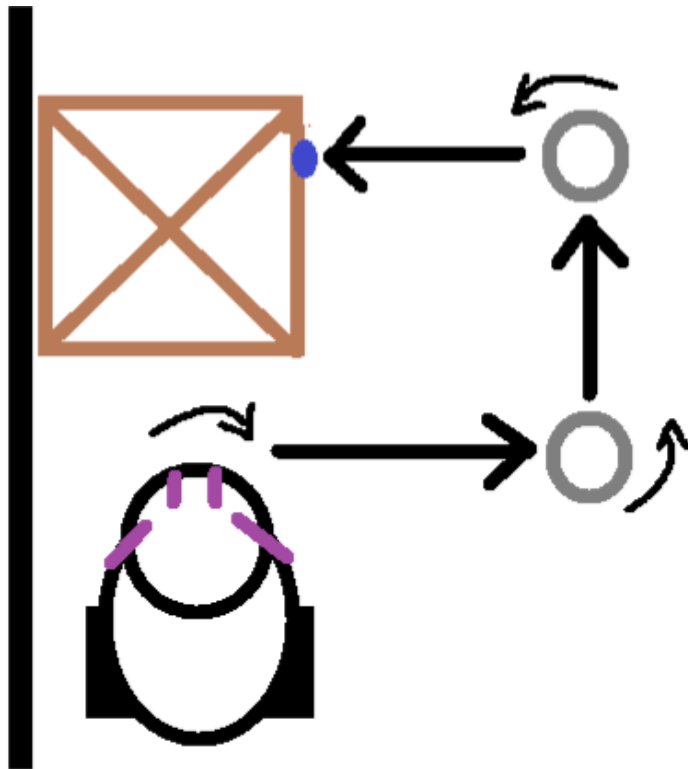
### Problemy

Największym problemem jaki napotkaliśmy była sytuacja, gdy czujnik robota nie wykrywał przeszkody przed nim i przy jeździe do przodu mimo wszystko zaczepiał on o przeszkodę co powodowało dalsze komplikacje przy wykonywaniu programu.

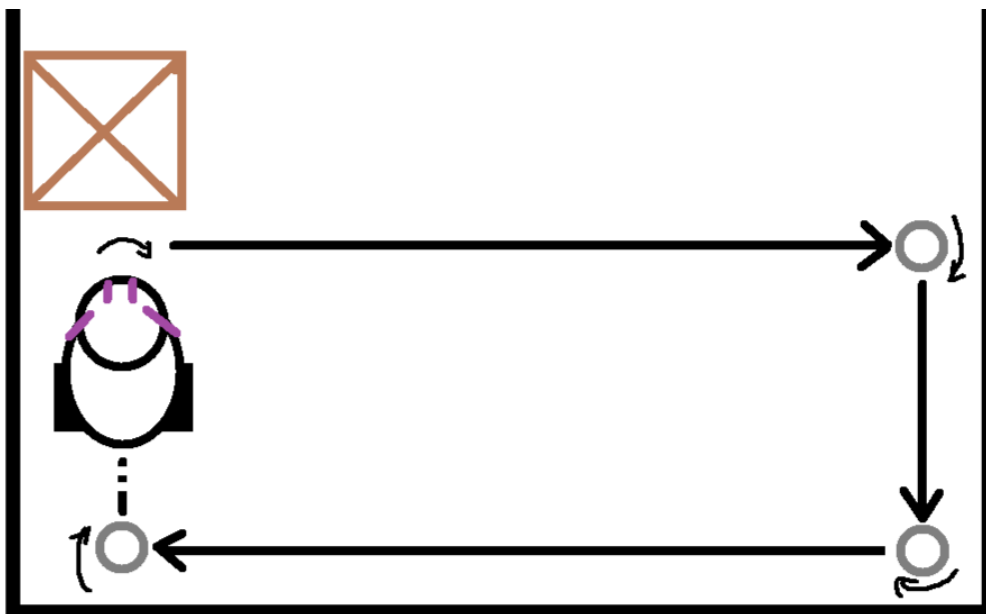


Aby pozbyć się tego problemu moglibyśmy przy sprawdzaniu obecności przeszkody na lewo od robota dodać, aby wykonywał dodatkowy obrót o np.  $5^\circ$  w ten sposób upewniając się, że nie zahaczy on o żadną przeszkodę kołem.

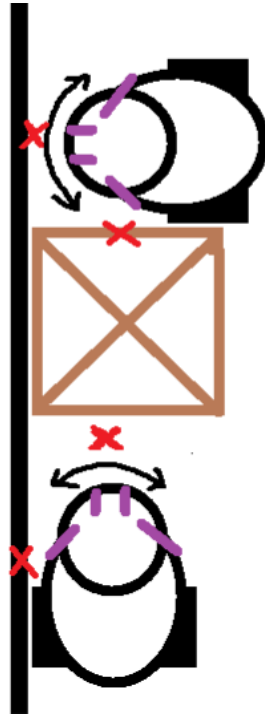
Kolejnym problemem jaki się pojawiał powstawał, gdy przy próbie ominięcia przeszkody robot za bardzo oddalał się od przeszkody przez co kończył on w sytuacji, gdy nie wykrywał kartonu po jego lewej i ustawiał się do niego frontalnie. Problem ten można byłoby rozwiązać rozszerzając kod o instrukcje do wykonania w zaistniałej sytuacji, niestety z powodu ograniczonego czasu nie zdążyliśmy zaimplementować tego rozwiązania.



Występowała również sytuacja, w której robot w jakiś sposób ignorował warunki i pudełko traktował jak zwykły róg. Problem pojawiał się sporadycznie, gdy w kodzie wkradły się małe błędy, które szybko dało się naprawić



Ostatnia sytuacja, w której robot nie mógł obrócić się lewo lub prawo bez punktu kolizyjnego. Problem pojawiał się, gdy pozycja początkowa Photona była zbyt blisko ściany lub w sytuacji, gdy Photon ominął karton, ale miał bardzo mało miejsca po swojej lewej stronie.



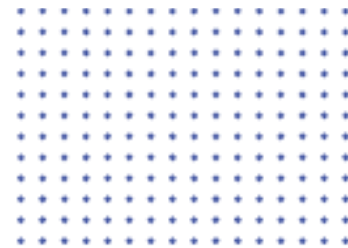
## Wnioski

Podczas pisania kodu do robota podczas określania odległości przejazdu okazało się, że Photon nie przyjmuje wartości zmiennoprzecinkowych, przez co należy wykorzystać funkcję matematyczne do zaokrąglania liczb zmiennoprzecinkowych do liczb całkowitych. Dostyc problematyczne okazało się to, że istniała możliwość niewykrycia przeszkody przez robota, ale przy ruchu do przodu Photon zaczepiał kołem o przeszkodę.

Nie udało się nam zmierzyć czasu, o ile Photonowi udawało się jeździć dookoła areny wzdłuż ścian to próby ominięcia kartonu kończyły się w 80% przypadków niepowodzeniem. Jednym z ciekawych sposobów na ominięcie przeszkody byłoby "ominięcie trójkątne". Rozwiązuje ona bowiem problem gubienia kątów i redukuje konieczność stosowania korekty pozycji, bo zamiast 5 obrotów 90 stopniowych robot robi 3 obroty kolejno po 45, 90, 45 stopni. Możliwe, że gdybyśmy sami zaprojektowali w kodzie taki typ obracania rozwiązalibyśmy kilka problemów z którymi się mierzyliśmy.



# [2] POMIAR ARENY



## Zadanie do wykonania

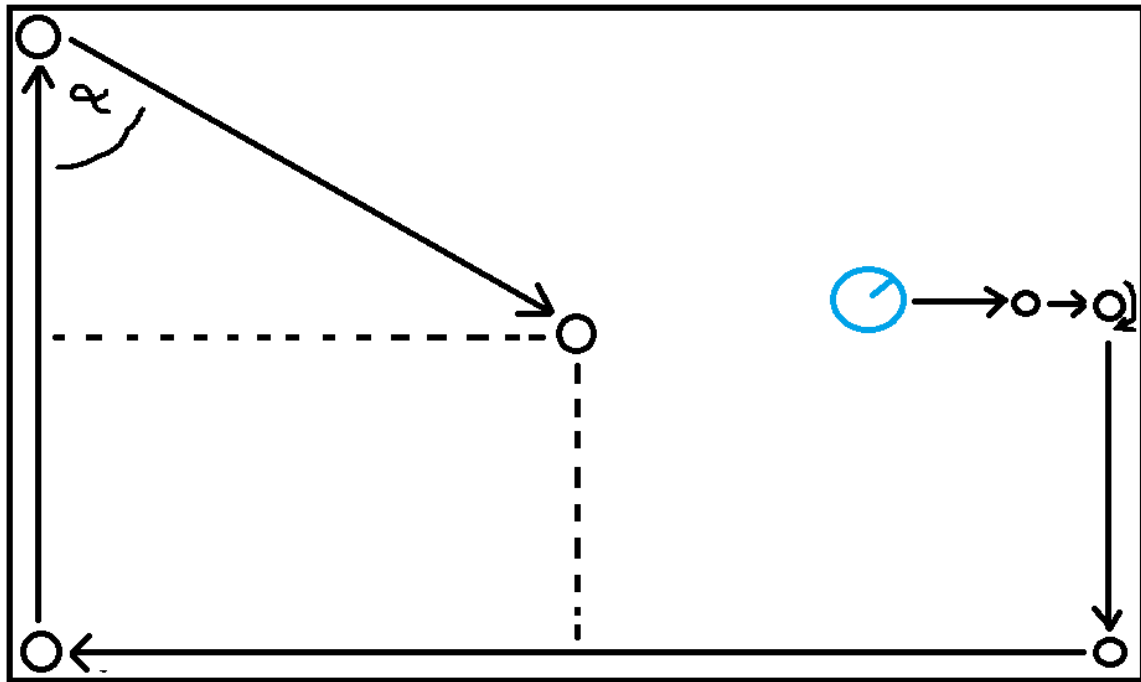
Celem ćwiczenia jest napisać program, za pomocą którego robot Photon dokona pomiaru wymiarów areny zaczynając z dowolnego miejsca po czym powróci na jej środek. Robot ma zaczynając z dowolnego miejsca na arenie powinien się ustawić prostopadle do ściany, zmierzyć wymiary areny a na koniec powrócić na jej środek.

## Kod robota Photon

<pre>from photonrobot import * import math photon.change_color("purple") for i in range(2):     dis = []     photon.rotate_left(30,50)     DL1 = photon.get_distance_from_obstacle()     photon.rotate_left(30,50)     DL2 = photon.get_distance_from_obstacle()     photon.rotate_right(60,50)     DC = photon.get_distance_from_obstacle()     photon.rotate_right(30,50)     DR1 = photon.get_distance_from_obstacle()     photon.rotate_right(30,50)     DR2 = photon.get_distance_from_obstacle()     dis.append(DL2)     dis.append(DL1)     dis.append(DC)     dis.append(DR1)     dis.append(DR2)     photon.rotate_left(120,50)     index = 0     for j in dis:         if j == min(dis) and index&gt;0:             photon.rotate_right(30*index,50)             index+=1     photon.make_sound("dog")     photon.go_forward(math.floor(min(dis)*0.5), 50) D = photon.get_distance_from_obstacle() if D &gt;= 5:     photon.rotate_right(90,50) elif D &lt; 5:     photon.go_backward(5,50)     photon.rotate_right(90,50) photon.make_sound("cat") D = photon.get_distance_from_obstacle() while D&gt;10:     photon.go_forward(math.floor(D*0.7), 50)     D = photon.get_distance_from_obstacle()</pre>	<pre>photon.rotate_right(90,50) #tworzą do mierzenia photon.change_color("blue") photon.go_backward(60,25) M = [] tmp = 0 for i in range(2):     tmp = 0     D = photon.get_distance_from_obstacle()     while D &gt; 10:         D = photon.get_distance_from_obstacle()         if D &lt; 20:             tmp+=D             break         else:             photon.go_forward(20, 50)             tmp += 20     D = photon.get_distance_from_obstacle()     tmp+= 15 #Dodanie długości robota     M.append(tmp)     photon.rotate_right(90,50)     D = photon.get_distance_from_obstacle()     photon.go_backward(40,25)     photon.go_forward(5, 50)     photon.make_sound("cat")     zal = M[0]/M[1]     alfa = math.atan(zal)     alfa = math.degrees(alfa)     photon.rotate_right(90-math.floor(alfa),50)     print(math.floor(alfa))     print(zal)     print(alfa)     final = math.sqrt((M[0]/2)**2 + (M[1]/2)**2)     photon.go_forward(math.floor(final)-10,100)     photon.rotate_right(360,100)     print(M)     print(M[0]*M[1])</pre>
---	--

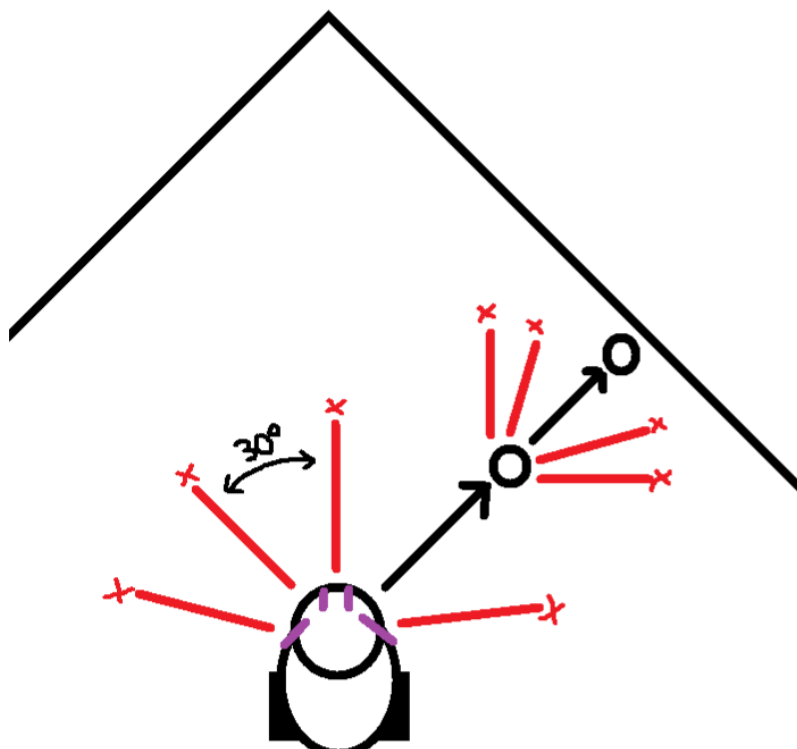
## Realizacja zadania

Realizacja zadania i kod, który go realizuje dzieli się na 3 etapy, pozycjonowanie, mierzenie, powrót do centrum areny. Przykład trasy, którą przejedzie robot widać na poniższym rysunku (jest to tylko 1 z wariantów, możliwa jest również opcja, w której robot pozycjonuje się do dłuższego boku areny).

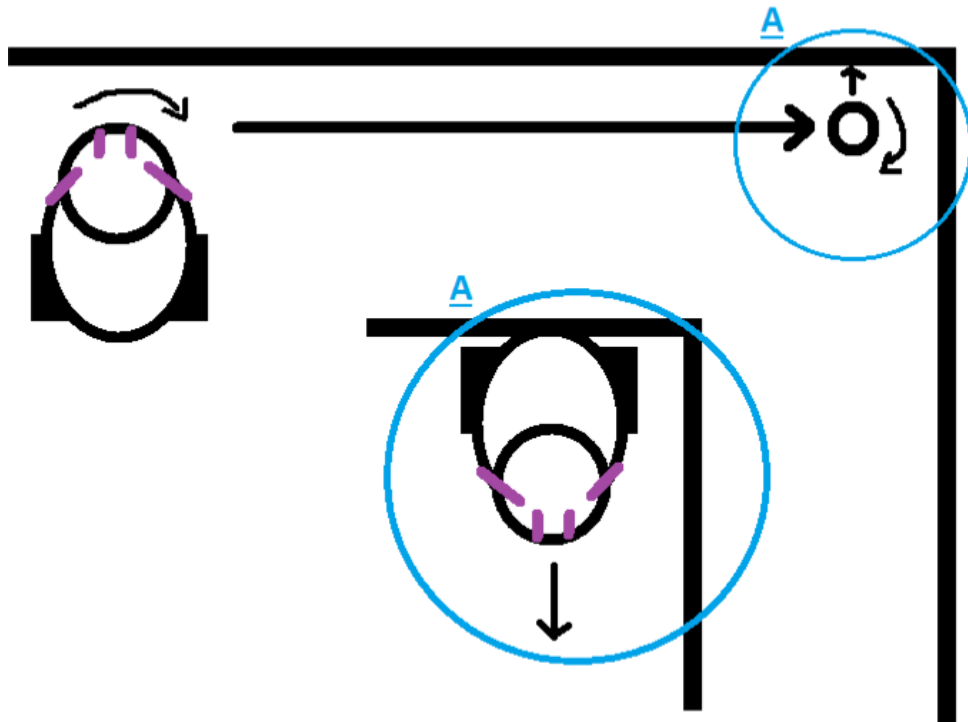


Schemat drogi którą realizuje kod programu robota

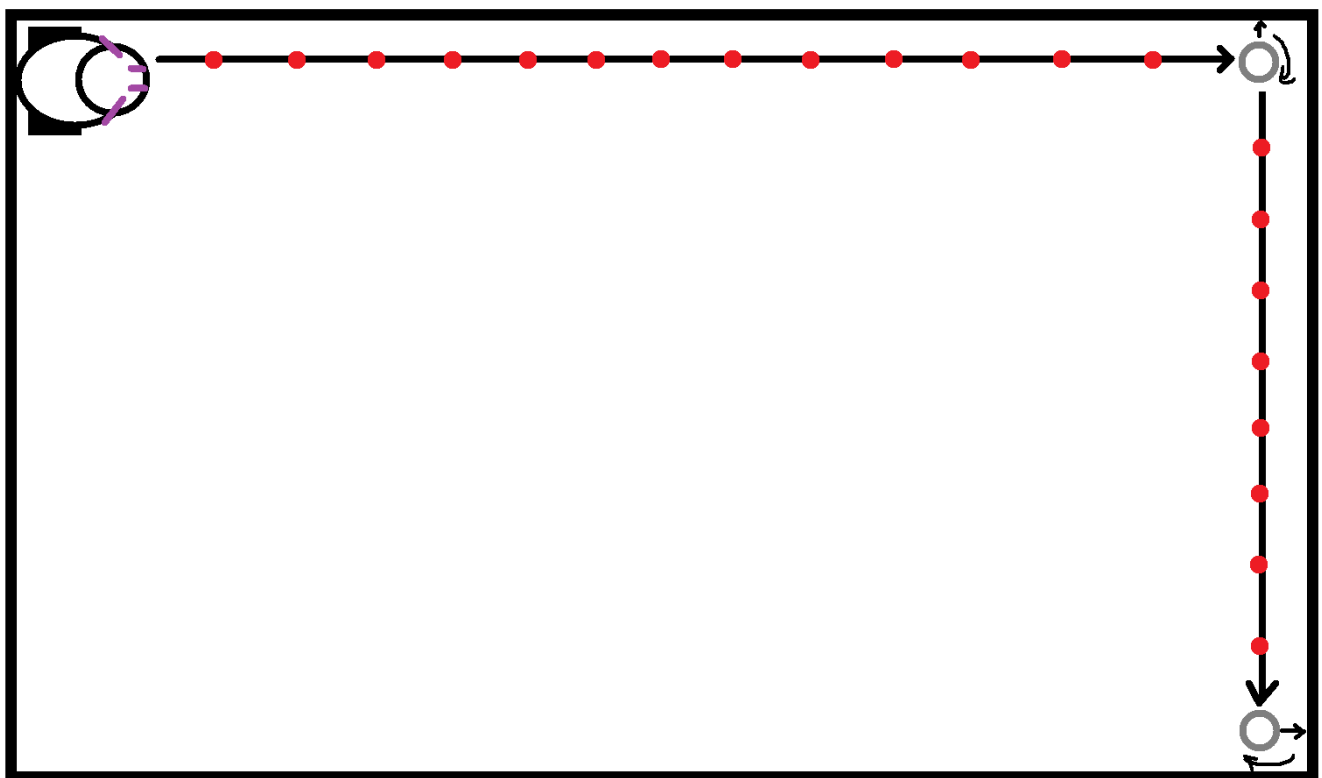
**Etap 1:** Pozycjonowanie, robot na początku czytuje 5 różnych wartości dystansów od ściany (obraca się o wartości 30 stopni), wybierana jest najmniejsza wartość i robot podjeżdża o połowę wartości dystansu w tym kierunku i dla lepszej precyzji powtarza jeszcze raz mierzenie (gdy odległość od ściany będzie zbyt blisko robot odsunie się o 5 cm).



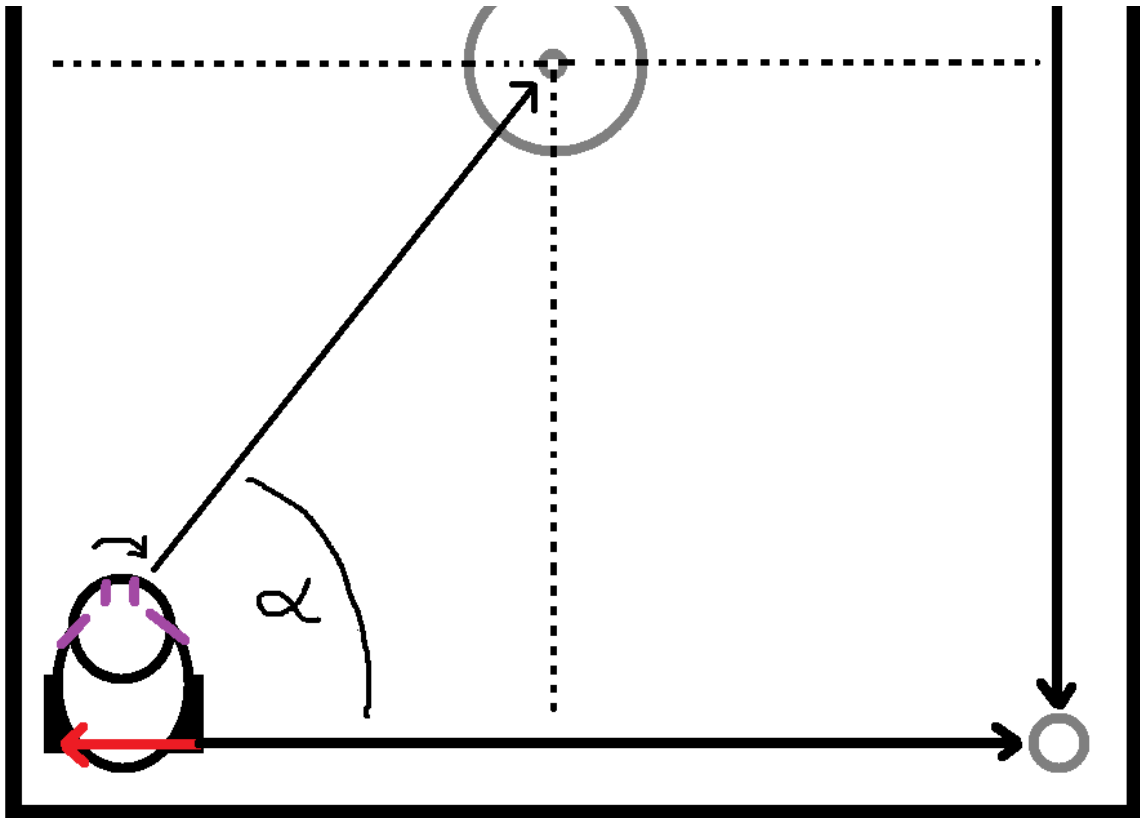
Etap 1.5: robot ustawiony prostopadłe obraca się w prawo i jedzie do rogu (zgodnie z ruchem wskazówek zegara), obraca się do mierzenia i cofa, aby jego kółka stykały się ze ścianą.



Etap 2: Mierzenie, robot podjeżdża co 20 cm i zapisuje wartości do zmiennej, gdy odległość od ściany będzie mniejsza od 20cm, robot obróci się w prawo, wypozycjonuje się jak w etapie 1 i do zmiennej doda 15cm (długość robota) i wartość odległości od ściany (mniejszej od 20cm), powtórzy algorytm dla drugiej ściany.



**Etap 3:** Powrót do centrum, robot wylicza przy pomocy 2 zmierzonych odległości kąt alfa (funkcja arctanges), obraca się o ten kąt następnie jedzie na odległość równą połowie przyprostokątnej (wyliczonej z twierdzenia pitagorasa), obraca się o 360 stopni a by zasygnalizować koniec programu.



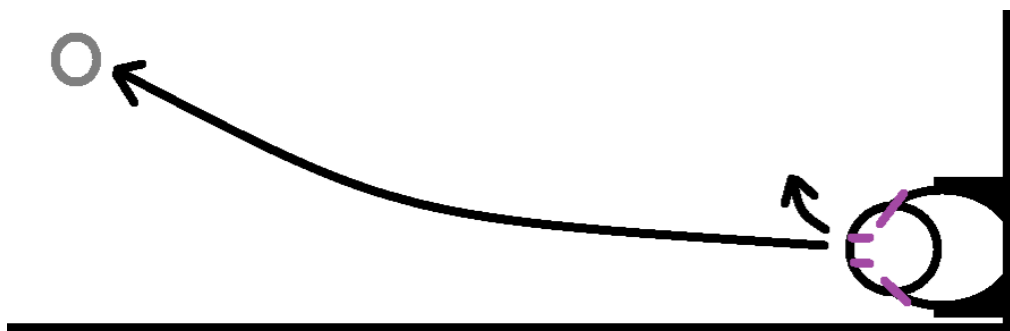
## Wynik

[1 : 39 : 75]	W. rzeczywiste [cm]	W. zmierzone [cm]	Błąd względny [%]
Ściana 1	278 cm	274 cm	1,44%
Ściana 2	200 cm	191 cm	4,50%
Pole pow. areny	55600 cm <sup>2</sup>	52334 cm <sup>2</sup>	5,87%

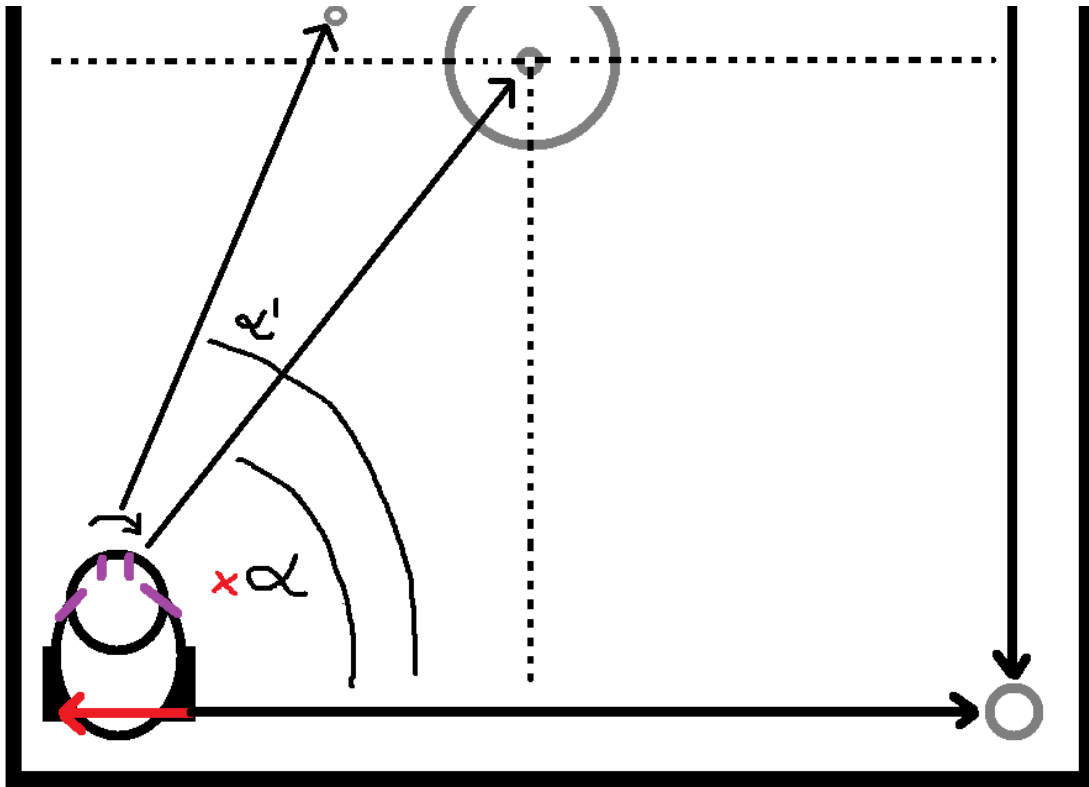
Cały kod wraz z wykonaniem go przez robota Photon zajął nam 1 min: 39 s: 75ms

## Problemy

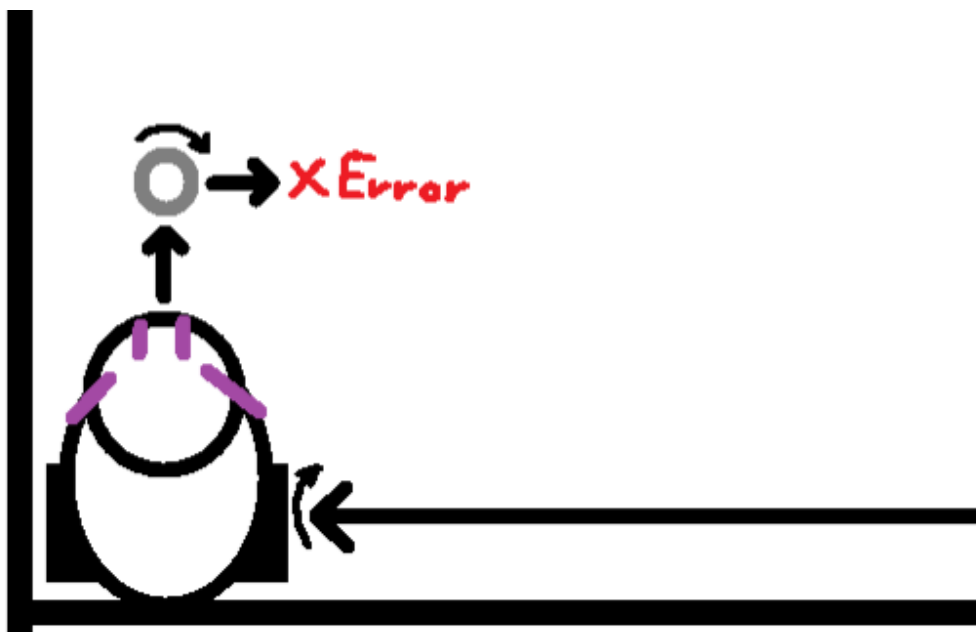
**Problem 1 i 2:** Robot, na którym pracowaliśmy miał poluzowane jedno koło przez co “znosiło” go w prawo podczas jazdy. Drugą rzeczą było to, że gdy ustawiony kołami do ściany miał ruszyć szybko to przez tarcie mógł skrócić znacznie w prawo. Konieczna była wymiana robota na inną sztukę.



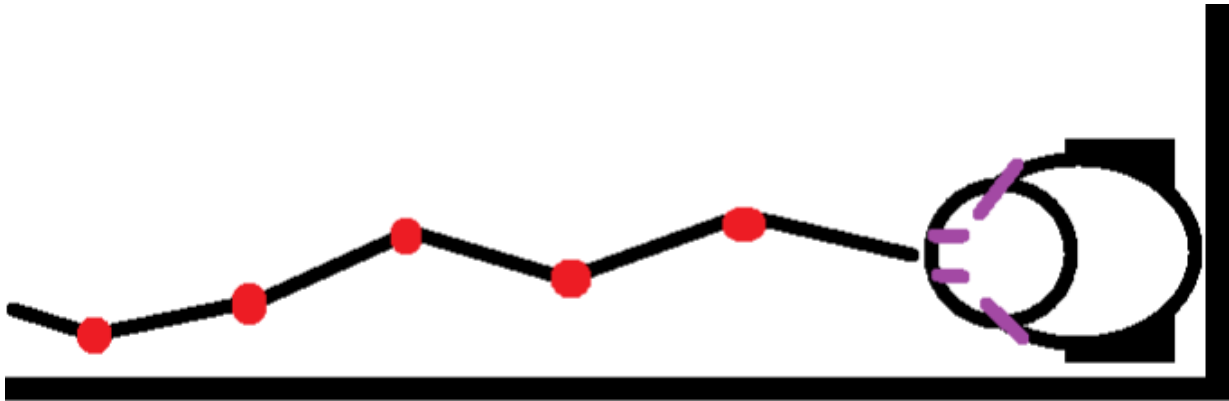
**Problem 3:** przy etapie 3 robot skręcał o zły kąt lub nie skręcał wcale. Było to spowodowane tym, że w nie wiedzieliśmy, że funkcje trygonometryczne z matematycznej biblioteki Pythona przyjmują i zwracają radiany a nie stopnie. Po konwersji radianów na stopnie problem więcej się nie pojawiał.



**Problem 4:** Zauważyliśmy, że robot niedokładnie odmierzał odległość od przeszkód a niekiedy wykrywał obiekt, gdy nic się przed nim nie znajdowało, co powodowało np. Przedwczesne przejście do etapu 3 i z powodu braku pomiaru drugiej długości, błąd w postaci dzielenia przez 0. Okazało się bowiem, że powodem tego problemu był przetłuszczony czujnik odległości, który to wystarczyło wyczyścić.



**Problem 5:** Jednym z problemów jaki napotkaliśmy była nierówna jazda robota tzn. w naszym kodzie robot poruszał się do przodu o 20 cm przez co Photon dosyć często się zatrzymywał. Zauważyliśmy, że przy rozpoczęciu ruchu robot nie jedzie idealnie prosto co miało bezpośredni wpływ na pomiar (robot lekko skręcał i nie jechał równoległe do ściany). Problem finalnie nie został naprawiony, ale nie wyglądał aż tak dramatycznie jak na obrazku.



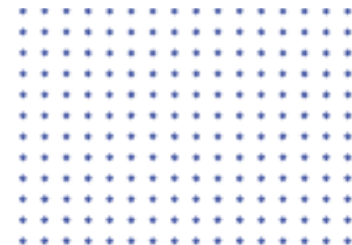
## Wnioski

Czas trwania całego procesu pomiaru areny trwał jedną minutę i trzydzieści dziewięć sekund, prędkość robota wynosiła 50% jego możliwości. Moglibyśmy skrócić ten czas zwiększając prędkość jazdy robota, jednakże nie chcieliśmy by większe wartości przyspieszenia wpłynęły na dokładność pomiaru wymiarów areny. Dobrym sposobem na powrót robota na środek areny po zakończeniu pomiaru było wykorzystanie funkcji arctangens (do obliczenia wartości kąta o jaki ma się on obrócić by być skierowanym w odpowiednią stronę) oraz twierdzenia pitagorasa, aby obliczyć na jaką odległość ma on pojechać do przodu. Jeżeli chodzi o dokładność pomiaru to odbiegał on od rzeczywistych wymiarów areny (rzeczywiste wymiary: 278cm x 200cm; zmierzone wymiary: 274cm x 191cm).

Do rozwiązania problemu 5 moglibyśmy zwiększyć długość odcinków o jaką Photon poruszał się do przodu podczas pomiaru. W wyniku tego robot rzadziej by się zatrzymywał i lepiej trzymałby się zaplanowanej trasy, początkowo nawet mieliśmy zaprojektowany kod, który zamiast robić przystanki co 20 cm to jechać o większą odległość a nawet do samego końca i w ten sposób skrócić czas działania programu. Zrezygnowaliśmy i wyrzuciliśmy to rozwiązanie z kodu, ponieważ uznaliśmy, że pomiary mogą być mniej dokładne oraz przy możliwych błędach z zjeżdżaniem na prawo itp. Jak się później okazało nasze obawy były błędne i powinniśmy zostać przy pierwotnym pomysśle.

Przydatnym przy tak złożonym kodzie okazało się stosowanie zmiany kolorów robota i dźwięki jako debugging aby lepiej orientować się w którym etapie jest robot, żeby łatwiej nam było naprawiać i usprawniać kod. Na przyszłość warto ustalić jest też jedna główna zmienna prędkości, która będzie uwzględniona na początku kodu, aby móc łatwiej wykonywać testy na różnych prędkościach.

# [3] ARENA Z PRZESZKODAMI



## Zadanie do wykonania

Celem ćwiczenia jest napisać program, za pomocą którego robot Photon będzie omijać napotkane przeszkody. Zadanie należy zrealizować używając tylko funkcji ruchowych “infinity”. Robot zaczynając z dowolnego miejsca na arenie powinien się poruszać do przodu aż do napotkania przeszkody, którą to powinien ominąć.

## Kod robota Photon

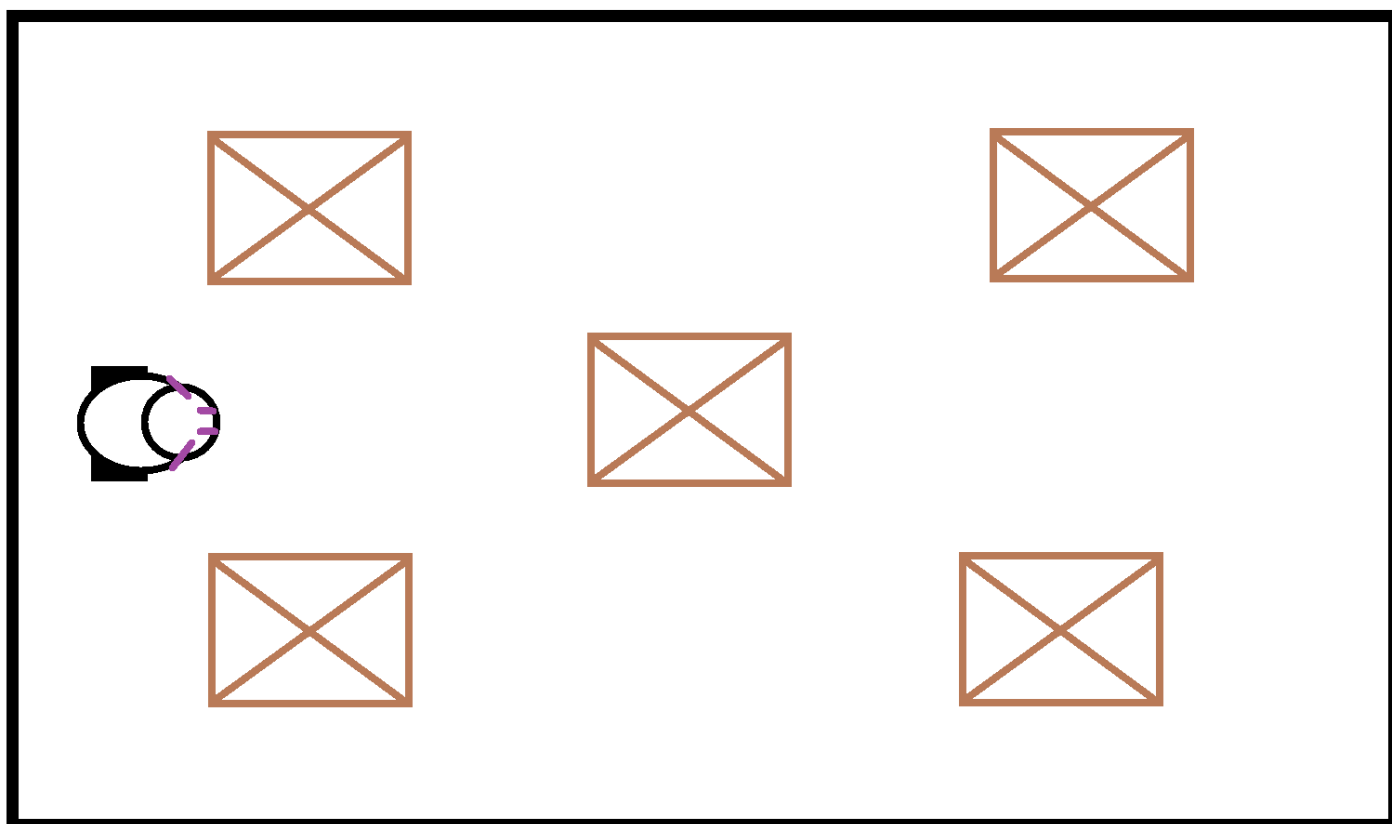
<div><b>METODA LOSOWA</b></div> <pre>from photonrobot import *  import time import random import math  photon.change_color("black") speed = 77 rpm = 50  while True:     D = photon.get_distance_from_obstacle()     photon.go_forward_infinity(speed)     if D &lt; 35: #(speed/100)*40 + (speed* 0.1):         photon.stop()         r1 = random.randint(90, 180)         r2 = random.randint(0, 9)         obr = (56 * (rpm/100)) / 60         if rpm == 100:             czas = obr * (r1 / 360)         else:             czas = obr * (r1 / 360) * (10 - (0.1 * rpm))         #czas = parseDou         if r2 &gt; 4:             photon.rotate_right_infinity(rpm)             time.sleep(czas)             print('Skrecono ',r1,' stopni w prawo', obr, math.floor(czas))             photon.stop()         else:             photon.rotate_left_infinity(rpm)             time.sleep(czas)             print('Skrecono ',r1,' stopni w lewo', obr, math.floor(czas))             photon.stop()</pre>
--

## Realizacja zadania

Realizacja zadania wykonaliśmy za pomocą 2 różnych programów, pierwszy po napotkaniu przeszkody obraca się w losowym kierunku do momentu, gdy przed nim nie będzie żadnego obiektu. Drugi program działa w następujący sposób - po napotkaniu przeszkody Photon wykonuje obrót o  $360^\circ$ , podczas którego co  $30^\circ$  dokonuje pomiaru odległości (by wykonywać pomiary z większą częstotliwością należałoby zmniejszyć prędkość obrotu) i dodaje je do tablicy. Następnie wybiera największą wartość (czyli 255 - czujnik nic nie wykrył) i obraca się w tym kierunku.

Wzory do obliczeń wartości, które zostały uwzględnione w kodzie:

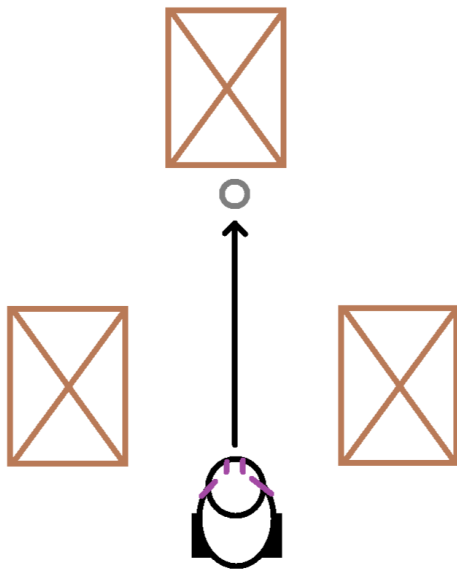
PROGRAM 1:	PROGRAM 2:
$Obr = \frac{56 \cdot \left( \frac{V_{obr}}{100} \right)}{60} \quad \left[ \frac{obr}{s} \right]$ <p>Obr to wartość obrotów na sekundę, z uwzględnieniem prędkości ustalonej w procentach [%].</p>	
$Czas = Obr \cdot \left( \frac{kąt}{360} \right) [s]$ <p>Czas jest wartością, ile zajmie wykonanie danego kąta. W tym przypadku kąt jest liczbą losową z przedziału <math>\langle 90, 180 \rangle</math></p>	$Czas = Obr \cdot \frac{(index \cdot kąt)}{360} [s]$ <p>Czas jest wartością, ile zajmie wykonanie danego kąta. W tym przypadku, kąt jest uwzględniany przez ilość pomiarów podzielony przez 360, a index jest liczbą indexem wartości, która ma największą wartość w tablicy.</p>



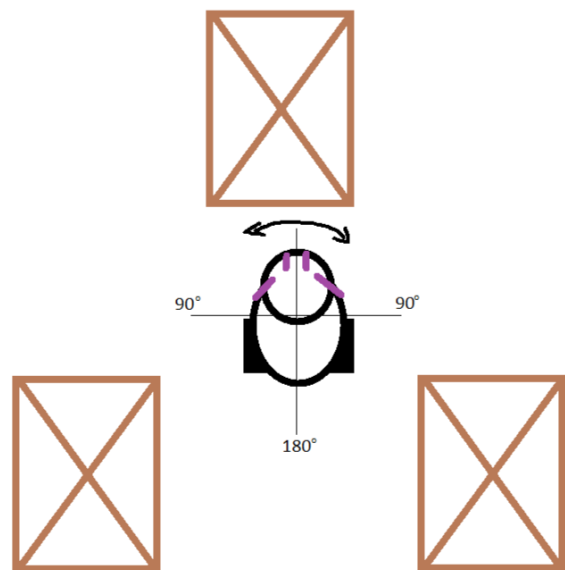
Rys.3.1. Arena, po której poruszać miał się robot wraz z omijaniem przeszkód w postaci pudełek.



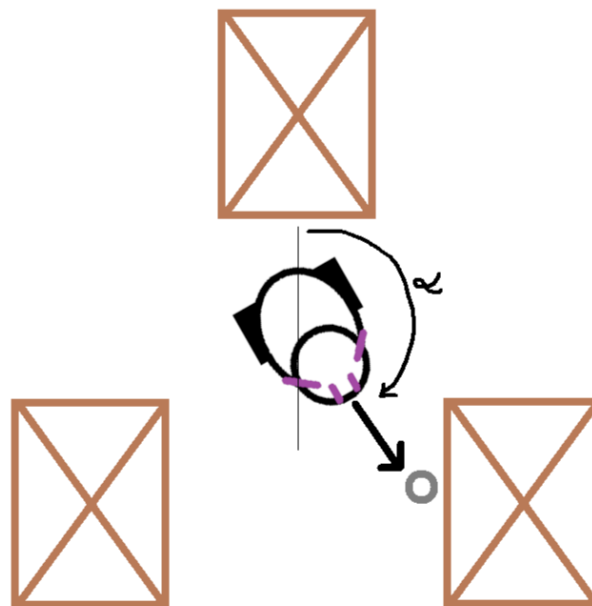
## PROGRAM 1: Droga wybierana losowo



**Etap 1:** robot jedzie do przodu aż do napotkania przeszkody.

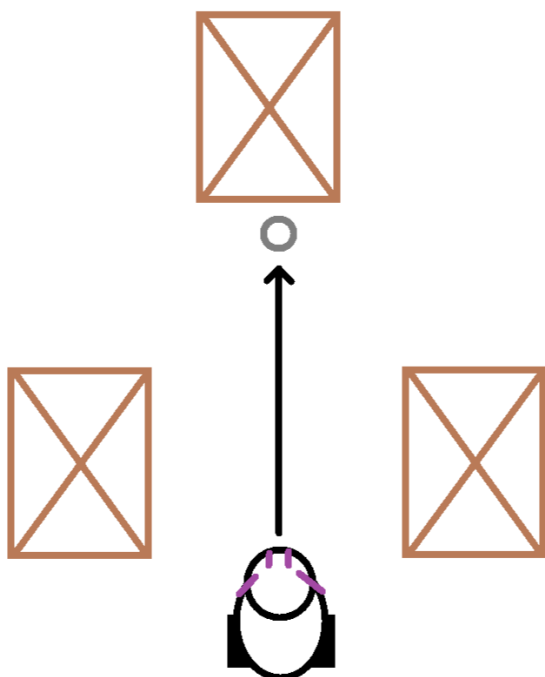


**Etap 2:** robot zbliża się do przeszkody po czym pseudolosowo generowany jest kąt obrotu.

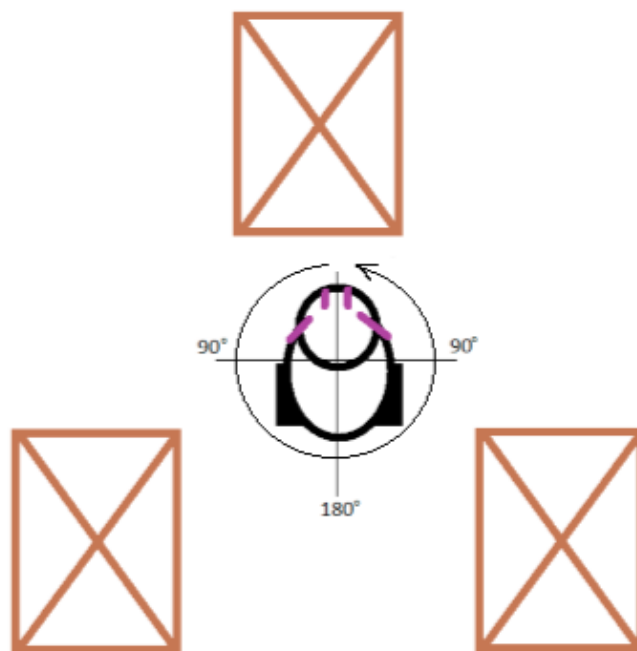


**Etap 3:** robot obraca się o wygenerowany kąt i następuje odczyt z czujnika odległości. Jeżeli dalej znajduje się przed nim przeszkoda to ponownie generowany jest kąt oraz sprawdzany jest stan czujnika. Proces ten jest powtarzany, dopóki nie czujnik nie wykryje przeszkody.

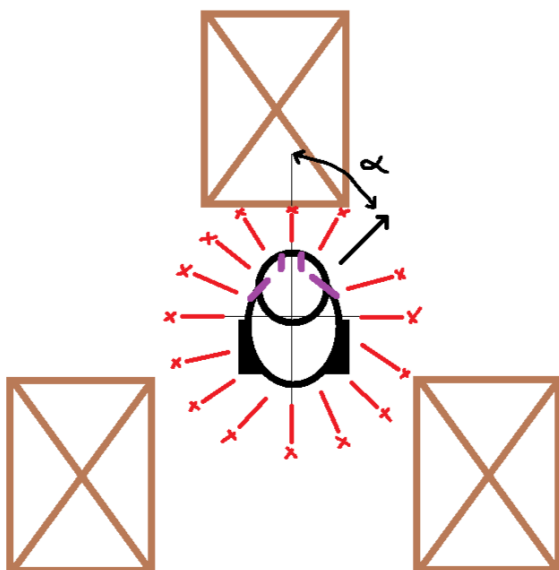
## PROGRAM 2: Droga wybierana na podstawie tablicy pomiarów



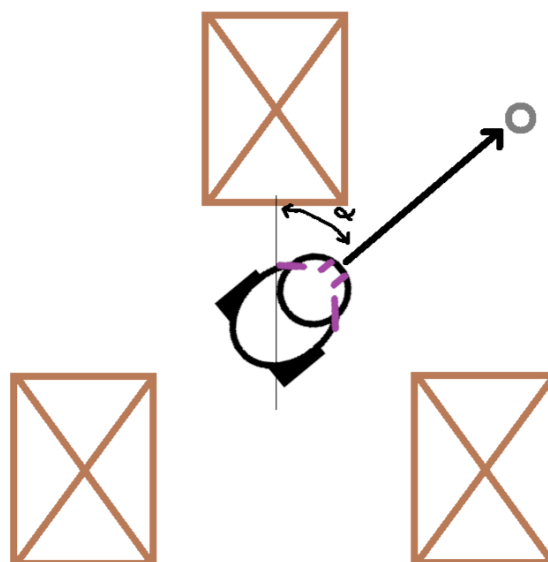
Etap 1: robot jedzie do przodu aż do napotkania przeszkody.



Etap 2: robot zbliża się do przeszkody po czym wykonuje obrót o 360° wykonując pomiar co 30°.



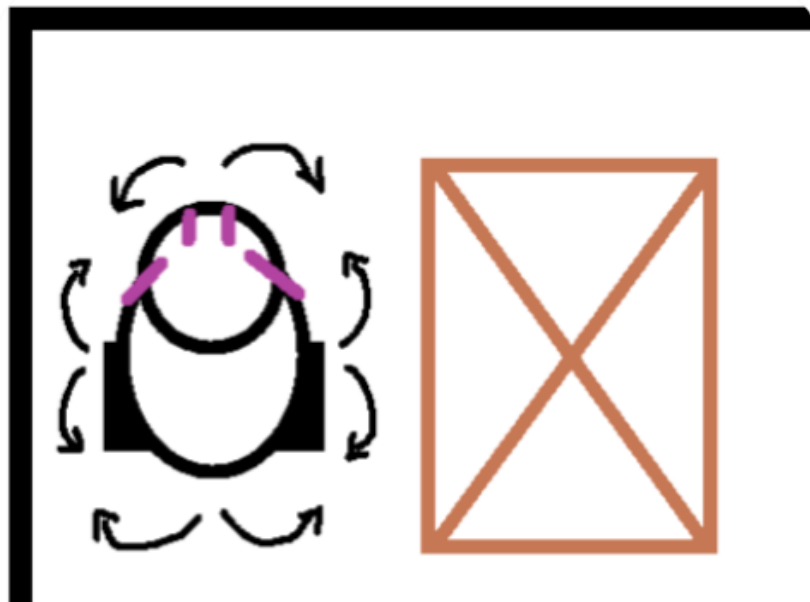
Etap 2.5: Schematyczny rysunek dokonanych pomiarów.



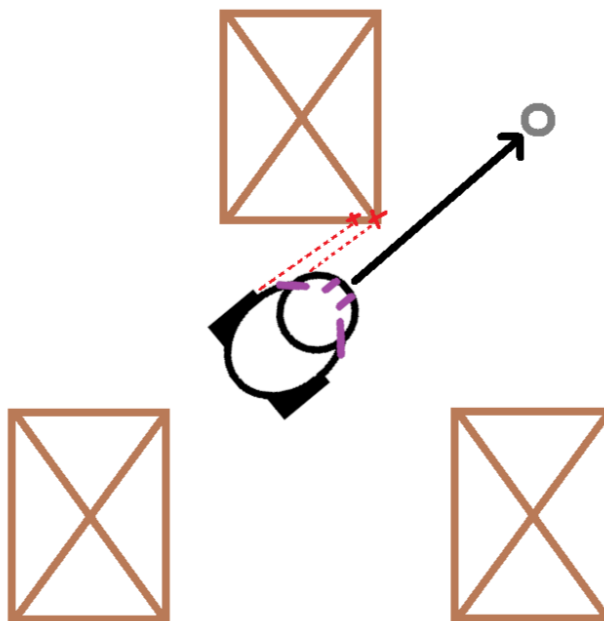
Etap 3: robot wybiera kierunek pozbawiony przeszkód (wartość maksymalna czujnika odległości) i obraca się o odpowiedni kąt.

## Problemy

### Kod z obrotem o losowy kąt

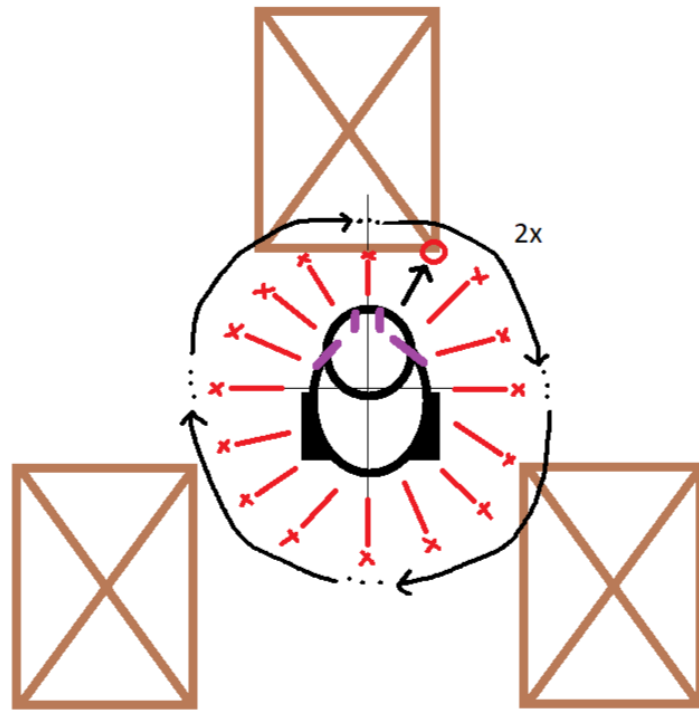


Jednym z problemów jaki napotkaliśmy przy metodzie obrotu o losowy kąt był taki, gdy możliwości ruchu robota były ograniczone przez otoczenie co skutkowało tym, że czas oczekiwania na to, że robot wyjedzie z takiej sytuacji był zbyt losowy np. robot mógł się dobrze ustawić po jednej lub stu iteracjach.

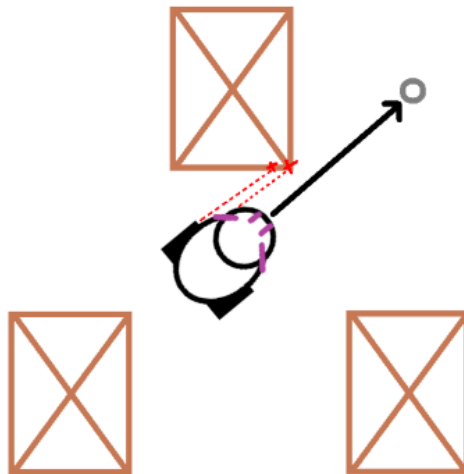


Kolejnym problemem było to, że po wykonaniu obrotu i nie wykryciu przeszkody, robot podczas jazdy do przodu wciąż mógł zahaczyć o karton (spowodowane jest to szerokim rozstawem kół z tyłu Photona oraz tym, że czujnik odległości sprawdza obecność przeszkody w wąskim zakresie przed robotem). By tego uniknąć moglibyśmy zrobić tak, że robot po obrocie losowy kąt dodatkowo obraca się o  $10^\circ$  w prawo i lewo by sprawdzić czy nie natrafi na przeszkodę.

## Kod z wykorzystaniem tablicy pomiarów



Ponieważ do wykonania zadania należało użyć funkcji “infinity” trzeba było obliczyć, ile zajmuje obrót o  $360^\circ$  oraz co, ile należy dokonywać pomiaru odległości (tak by pomiar był dokonywany co  $30^\circ$ ). Było to o tyle problematyczne, że prędkość obrotowa Photona nie jest stała (wpływa na to np. ustawienie przedniego koła) co skutkuje powstawaniem niedokładności. Prowadzi to do tego, że robot ostatecznie obróci się o zły kąt i uderzy w przeszkodę zamiast ją ominąć.



Przy wykorzystaniu programu z tablicą pomiarów po raz kolejny pojawia się problem “zahaczania” o przeszkody. Aby go rozwiązać wpadliśmy na pomysł, aby zwiększyć częstotliwość pomiaru odległości tak by dokonywany był on co  $10^\circ$ , żeby następnie wyliczać średnią z trzech kolejnych pomiarów (w ten sposób poszerzamy zakres “widzenia” robota i znajdujemy pozycję z najmniejszym prawdopodobieństwem zahaczenia o przeszkodę).

Ostatecznie robot obróci się w kierunku, w którym owa średnia jest najwyższa i ruszy bez przeszkód. Niestety przez ograniczoną ilość czasu nie zaimplementowaliśmy tego rozwiązania.

## Wnioski

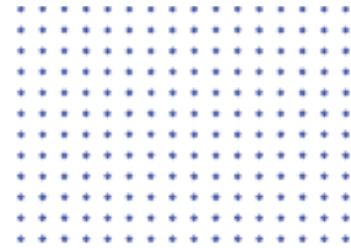
Z uwagi, że w zadaniu konieczne było korzystanie tylko z funkcji ruchowych "infinity" konieczne było skorzystanie z danych z pierwszego sprawozdania. Mianowicie zmierzaliśmy, że 100% prędkość obrotowa robota to ok. 55,83 RPM  $\approx$  56 RPM. Przy pomocy tej wartości można było wyznaczyć, ile obrotów robot wykona w sekundę, a co za tym idzie jaki czas będzie potrzebny do pokonania danego kąta. Ta ostatnia była bardzo istotna, bo to dzięki niej byliśmy w stanie zatrzymywać działanie funkcji ruchowych "infinity" po odpowiednim czasie, aby móc poruszyć się o dany kąt.

Niestety nigdy nie udało nam się idealnie wykonać ruchu obrotowego. Naprawialiśmy to dodając do kodu jakiś zapas czasu (np. +10%) który czasami działał bardzo dobrze, natomiast komplikował późniejsze etapy pisania kodu, ponieważ wszędzie musieliśmy uwzględniać to, że dodajemy pewne wartości do naszych wzorów, że finalnie w programie drugim skończyliśmy z tak zwany "kodem spaghetti" którego nie udało nam się naprawić, aby działał, jak należy.

Tak jak wspomnieliśmy w poprzednim sprawozdaniu, zaimplementowaliśmy opcję ustalania prędkości na samym starcie programu, aby łatwiej nam było testować jego działanie, dzięki temu dostosowaliśmy też wzory, aby kod dobrze obliczał czas jednego obrotu dla każdej możliwej wpisanej przez nas prędkości obrotu (1-100%). Prędkość ruchu i obrotu na poziomie 50% jest optymalna do testów i na niej najlepiej działał nam robot, przy 100% prędkości robot potrafił częściej gubić kąty i uderzać w kartony, mimo to również radził sobie dobrze.

Różnicą między programem pierwszym a drugim jest to, że program drugi jest bardziej optymalny. Zamiast losowości, która działa najczęściej dość średnio, robot w przypadku wykrycia przeszkody wybiera obrót w kierunku, gdzie jest największa odległość i działa przy tym ciągle, w teorii, bo nie udało nam się naprawić części błędów programu drugiego.

# [4] LINE FOLLOWER



## Zadanie do wykonania

Celem ćwiczenia jest napisać program, za pomocą którego robot Photon będzie podążać za linią i korygować trasę bez używania już istniejącej funkcji line-follower. Robot zaczynając z punktu startowego ma wykonać okrążenie w jak najkrótszym czasie.

## Kody robota Photon

### KOD 1 – Zapamiętywanie ostatniej korekcji:

```
from photonrobot import *
import time
photon.change_color("red")

def info(line):
    print("L: ",line.front_right, " ", "C: ",line.front_center, " ", "R: ",line.front_left)

def switch_kier(x):
    if x.front_left: #prawy
        return 1
    elif x.front_right: #lewy
        return 0

ost = 0 # 0 to lewo | 1 to prawo
v_obr = 5
kierunek = 0
while True:
    # pom = False
    photon.go_forward_infinity(10)
    line = photon.get_line_sensors()
    info(line)
    kierunek = switch_kier(line)

    while (not line.front_center): #and ((not line.front_right) and (not line.front_left)):
        if kierunek==0: #lewy
            photon.rotate_left_infinity(v_obr)
            while True:
                line = photon.get_line_sensors()
                if(line.front_center):
                    photon.stop()
                    ost = 0
                    break
        elif kierunek==1: #prawy
            photon.rotate_left_infinity(v_obr)
            while True:
                line = photon.get_line_sensors()
                if(line.front_center):
                    photon.stop()
                    ost = 1
                    break
        while (not line.front_right) and (not line.front_left) and (not line.front_center):
            line = photon.get_line_sensors()
            if ost == 0: #lewy
                photon.rotate_left_infinity(v_obr)
            if ost == 1: #elif
                photon.rotate_right_infinity(v_obr)
    photon.stop()
    line = photon.get_line_sensors()
```

## KOD 2 – Rozważanie wszystkich warunków (nieoptymalne):

```
from photonrobot import *
import time
photon.change_color("red")

def info(line):
    print("L: ",line.front_right, " ", "C: ",line.front_center, " ", "R: ",line.front_left)

def switch_kier(x):
    if x.front_left: #prawy
        return 1
    elif x.front_right: #lewy
        return 0

ost = 0 # 0 to lewo | 1 to prawo
v_obr = 5
kierunek = 0
while True:
    # pom = False
    photon.go_forward_infinity(10)
    line = photon.get_line_sensors()
    info(line)
    kierunek = switch_kier(line)

    while (not line.front_center): #and ((not line.front_right) and (not line.front_left)):
        if kierunek==0: #lewy
            photon.rotate_left_infinity(v_obr)
            while True:
                line = photon.get_line_sensors()
                if(line.front_center):
                    photon.stop()
                    ost = 0
                    break
            elif kierunek==1: #prawy
                photon.rotate_left_infinity(v_obr)
                while True:
                    line = photon.get_line_sensors()
                    if(line.front_center):
                        photon.stop()
                        ost = 1
                        break
        while (not line.front_right) and (not line.front_left) and (not line.front_center):
            line = photon.get_line_sensors()
            if ost == 0: #lewy
                photon.rotate_left_infinity(v_obr)
            if ost == 1: #elif
                photon.rotate_right_infinity(v_obr)
        photon.stop()
        line = photon.get_line_sensors()
```

## KOD 3 – wersja działająca:

```
from photonrobot import *
photon.change_color("red")
def info(line):
    print("L: ",line.front_right, " ", "C: ",line.front_center, " ", "R: ",line.front_left)

v_obr=6
v = 21
while True:
    photon.go_forward_infinity(v)
    line = photon.get_line_sensors()
    if line.front_left==True:
        photon.rotate_right_infinity(v_obr)
        while line.front_center==False:
            line = photon.get_line_sensors()
    elif line.front_right==True:
        photon.rotate_left_infinity(v_obr)
        while line.front_center==False:
            line = photon.get_line_sensors()
```

## Realizacja zadania

### a. Kod 1 - zapamiętywanie ostatniej korekcji

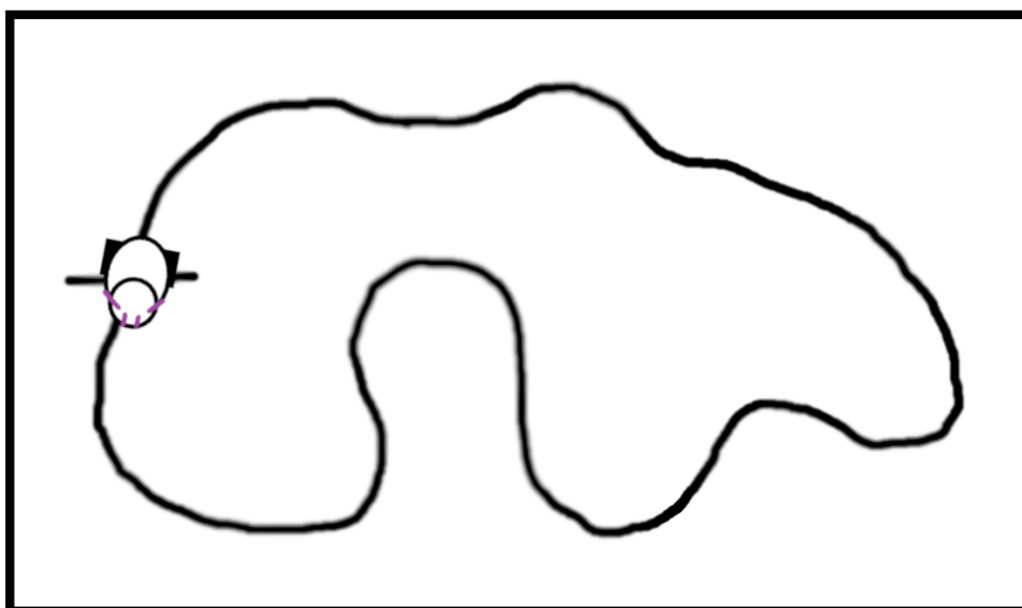
Wydawać by się mogło, że ten kod miał być najoptymalniejszy ze wszystkich, ponieważ zawierał opcję zapamiętywania ostatniego skrętu by zapobiegać obrotom o 180 stopni względem linii, co dręczyło nas przy testach. Jednakże kod nie chciał nam się wykonać, mimo że był chudszy i prostszy od swojego poprzednika. Robot miał pobierać z czujników wartości wykrycia czarnego koloru, dopóki nie znalazł na lewej lub prawej czarnego to miał jechać prosto, w innym przypadku w zależności od tego po której stronie znajdowała się linia to miał obracać się do momentu aż czujnik środkowy wykryje czarny kolor. Robot następnie zapisywał kierunek obrotu i wychodził z pętli. W przypadku braku wykrycia na wszystkich czujnikach miał się obracać w kierunku, który był zapisany w zmiennej.

### b. Kod 2 - rozważanie wszystkich warunków

Ten kod polega na rozważeniu wszystkich przypadków, w których zmieniają się stany czujników. Niestety okazało się, że przez obciążenie związane z nieustanną komunikacją z komputerem, kod ten był nieoptymalny i za wolno reagował na zmianę stanów czujników, co skutkowało tym, że robot gubił linię. Ponadto był to pierwszy kod pisany na zajęciach.

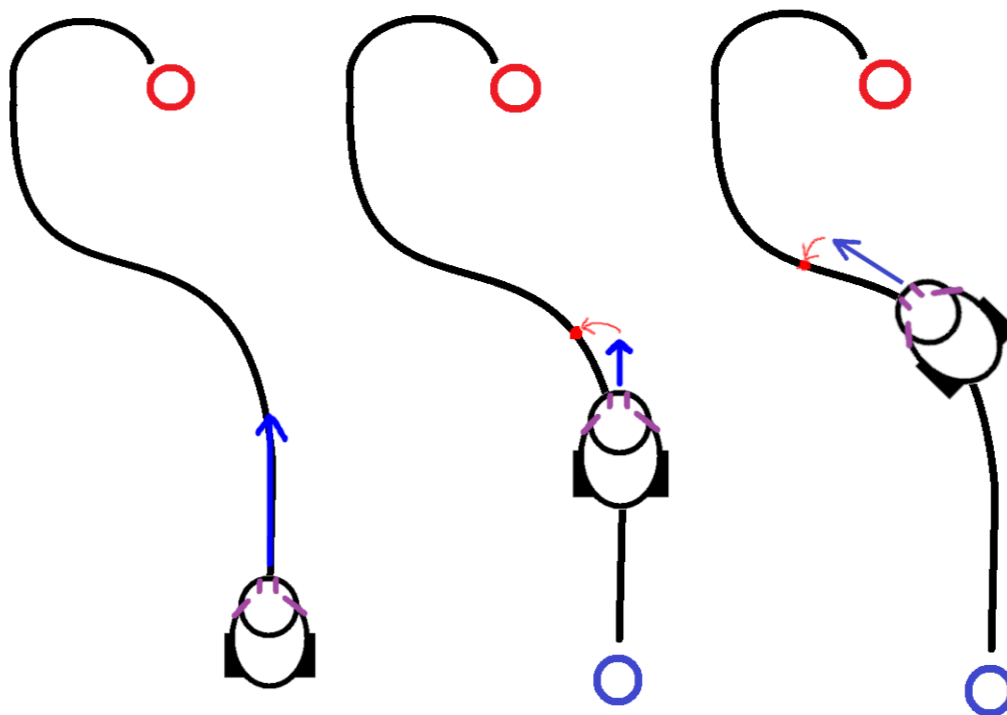
### c. Kod 3 – wersja działająca

W tym kodzie zmieniliśmy nasze podejście, tzn. Zamiast rozważać każdą możliwą kombinację czujników czy też zapamiętywać ostatni kierunek korekcji, postanowiliśmy ustawić jako domyślną akcję robota jazdę do przodu i dopiero gdy wykryje zmianę stanu jednego ze skrajnych czujników następuje korekcja (robot zaczyna się obracać w odpowiednim kierunku aż do momentu, gdy środkowy czujnik wykryje linię). Dzięki prostemu kodowi nie było już problemu z komunikacją, jednakże Photon wykonywał pełne okrążenie w czasie 4 min 28 sek co nie było wybitnym wynikiem. Można byłoby nieco poprawić ten czas poprzez odpowiednie dobranie parametrów prędkości jazdy i obrotu.



Rys.4.1. Arena z trasą robota





Rys.4.2. Wizualizacja działania kodów

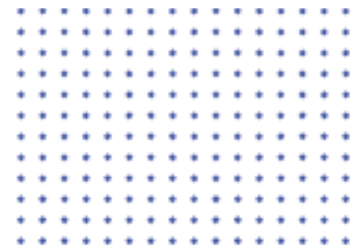
Robot jedzie prosto, gdy czujnik środkowy wykrywa czarny, jeśli nie będzie sygnału a na bocznych czujnikach wykryje linię to ma dokonać korekty, żeby czujnik środkowy znalazł się nad linią.

## Wnioski

Ostatecznie najprostszy kod (ten przy którym pomógł nam merytorycznie prowadzący) okazał się być najbardziej optymalny, najprostszy oraz jako jedyny pokonał całą trasę. Zbytne obciążanie komunikacji między Photonem a stacją komputerową poprzez zbyt częstą wymianę danych może prowadzić do powstawania błędów w wykonywaniu programu. Robot podczas wykonywania wszystkich powyższych kodów zatrzymywał się w czasie korekcji co nie miało miejsca w przypadku użycia gotowej, wbudowanej funkcji do podążania za linią. Do najprostszego line-followera potrzeba jedynie dwóch czujników, lewego i prawego. Niepotrzebnie skomplikowaliśmy sobie sprawę wykorzystując czujnik środkowy jako warunek do budowania instrukcji warunkowych i pętli.

Po konsultacji z grupą przeciwną (tą z najlepszym czasem) porównaliśmy swoje kody w celu wyciągnięcia wniosków, refleksji, jakie błędy popełniliśmy i co dało się poprawić, zauważyliśmy, że nasze kody nie różniły się za bardzo, składniowo wyglądają podobnie. Kod kolegów porównywany był z naszym kodem nr.1 i 3 opisanej wyżej w sprawozdaniu. Grupa przeciwna zdecydowała się na wyrzucenie sprawdzania środkowego czujnika co poskutkowało najlepszym czasem, dodatkowo ograniczyli pętle “while” i prawie wszystko robili na instrukcjach warunkowych, ponadto też zastosowali zapisywanie ostatniego kierunku, nasz kod dodatkowo zawsze stosuje zmienną główną do prędkości, aby wygodnie zmieniać je w kodzie. Widocznie kluczem do sukcesu były te najprostsze rozwiązania i nieprzekombinowanie sprawy, ale chociaż udało się nam dojechać Photonem do mety, czyli pokonać cały tor, więc aż tak dużej porażki w wykonywaniu zadania nie było, osiągnęliśmy dostateczny czas.

# [5] GENEROWANIE OBRĘCZY W POZYCJI STACJONARNEJ – WSTĘP DO MAPOWANIA



## Zadanie do wykonania

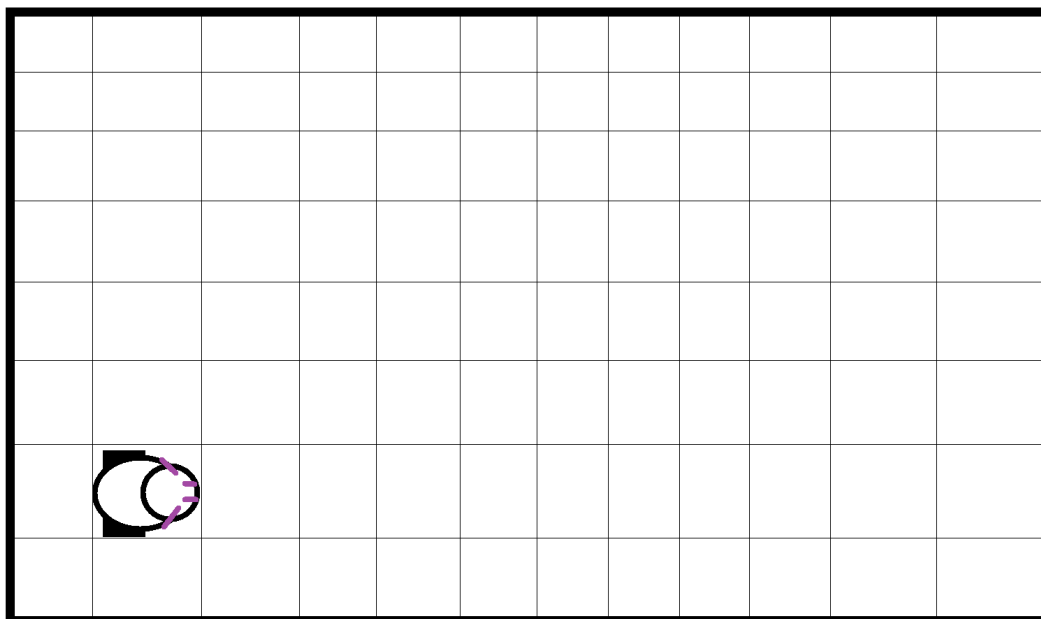
Celem ćwiczenia jest napisać program, za pomocą którego robot Photon będzie tworzył mapę areny z oznaczeniem pozycji robota oraz jego najbliższego otoczenia. Sam robot podczas tego procesu jest stacjonarny. Robot będąc w określonym miejscu ma zmapować najbliższe otoczenie i wyznaczyć swój obszar roboczy. Wynik ma zostać wyświetlony w formie tablicy.

## Kod robota Photon

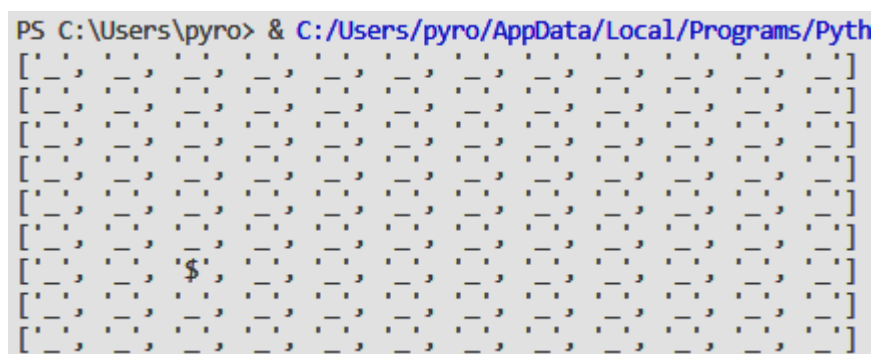
```
from photonrobot import *
import math
photon.change_color("pink", ColorMode.eyes)
photon.change_color("blue", ColorMode.ears)
d = 25
x1=2
y1=2
alfa = 0
k_obr = 15
length = 12
tab = [["_"]*12 for _ in range(9)] #generowanie pustej tablicy
for i in range(8,-1,-1):
    print(tab[i])
##### generowanie tablic /\
while alfa<360:
    pom = photon.get_distance_from_obstacle()
    if pom > 125:
        pom=125
    pom = pom + length #dodanie do w. pomiaru różnicę odległości między czujnikiem a osią obrotu
    cp1 = math.cos(math.radians(alfa))*(pom/d) #delta X
    sp1 = math.sin(math.radians(alfa))*(pom/d) #delta Y
    tab[round(sp1+y1)][round(cp1+x1)] = "o" #Zaznaczenie wykrytego obiektu lub granicy w. czujnika
    # Z KODU ZOSTAŁ WYKLUCZONY NIESTABILNY KOD WYPEŁNIANIA OBRĘCZY!!!
    alfa += k_obr
    photon.rotate_left(k_obr-3, 50)

tab[y1][x1]="$" #zaznaczenie pozycji robota
for i in range(8,-1,-1):
    print(tab[i])
```

## Realizacja zadania



Rys.5.1. Mapa areny (kwadraty mają wielkość 25x25 cm, rysunek koncepcyjny)

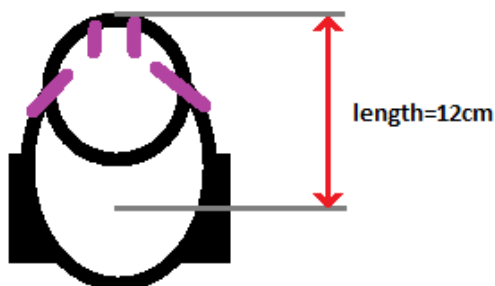


Rys.5.2. Tablica 12 x 9 wygenerowana przez kod wraz ze zaznaczoną pozycją robota

```
tab = [["_"]*12 for _ in range(9)]  
for i in range(8,-1,-1):  
    print(tab[i])
```

Rys.5.3. Fragment kodu odpowiedzialny za generowanie i wyświetlanie pustej tablicy

Zadanie zakłada, że znamy początkową pozycję robota. Po utworzeniu wzorcowej tablicy robot mierzy odległość od przeszkody i zapisuje wartość w zmiennej "pom" a następnie dodawana jest odległość od czujnika robota do jego osi obrotu.

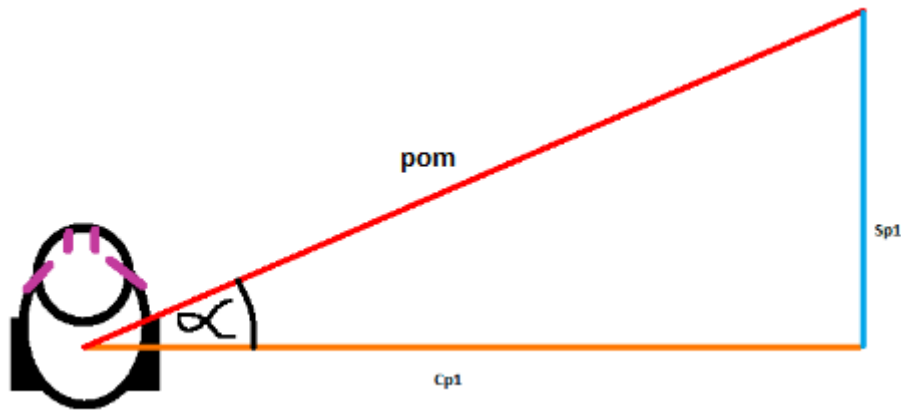


Rys.5.4. Rysunek przedstawiający odległość między czujnikiem odległości a osią obrotu robota

Kolejnym krokiem rozłożenie danego pomiaru na składowe “x” oraz “y” aby móc określić współrzędne przeszkody lub maksymalnego zasięgu widzenia. Dzielenie przez “d” czyli przez długość jednej komórki areny) ma na celu przeliczenie odległości na dystans w “komórkach”.

```
pom = pom + length
cp1 = math.cos(math.radians(alfa))*(pom/d) #delta x
sp1 = math.sin(math.radians(alfa))*(pom/d) #delta y
```

Rys.5.5. Fragment kodu odpowiedzialny za określanie współrzędnych

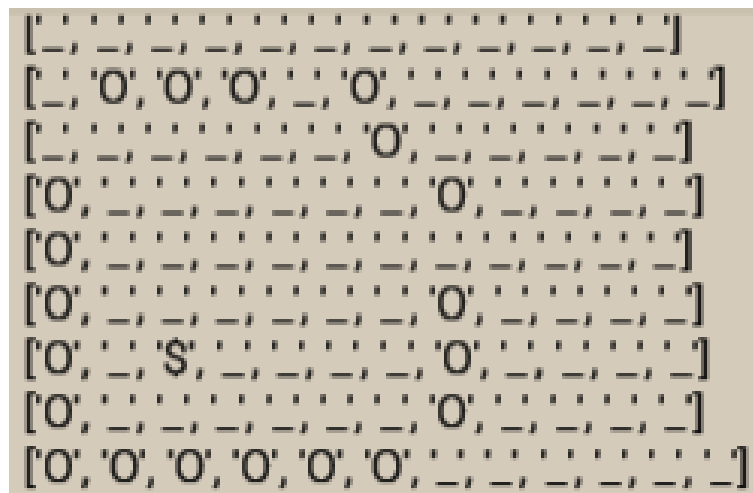


Rys.5.6. Rysunek przedstawiający “rozbitcie” pomiaru na składową “x” (Cp1) oraz “y” (Sp1)

Ostatnim krokiem jest wpisanie do odpowiedniej współrzędnej w tablicy znaku oznaczającego krawędź obszaru roboczego (uwzględniane są współrzędne początkowe robota).

```
tab[round(sp1+y1)][round(cp1+x1)] = "o"
```

Po takim pomiarze robot obraca się o 15 stopni i cały proces określania współrzędnej krawędzi jest ponownie wykonywany. Po wykonaniu 24 takich pomiarów wyznaczanie obszaru roboczego jest zakończone i wyświetlany jest wynik.

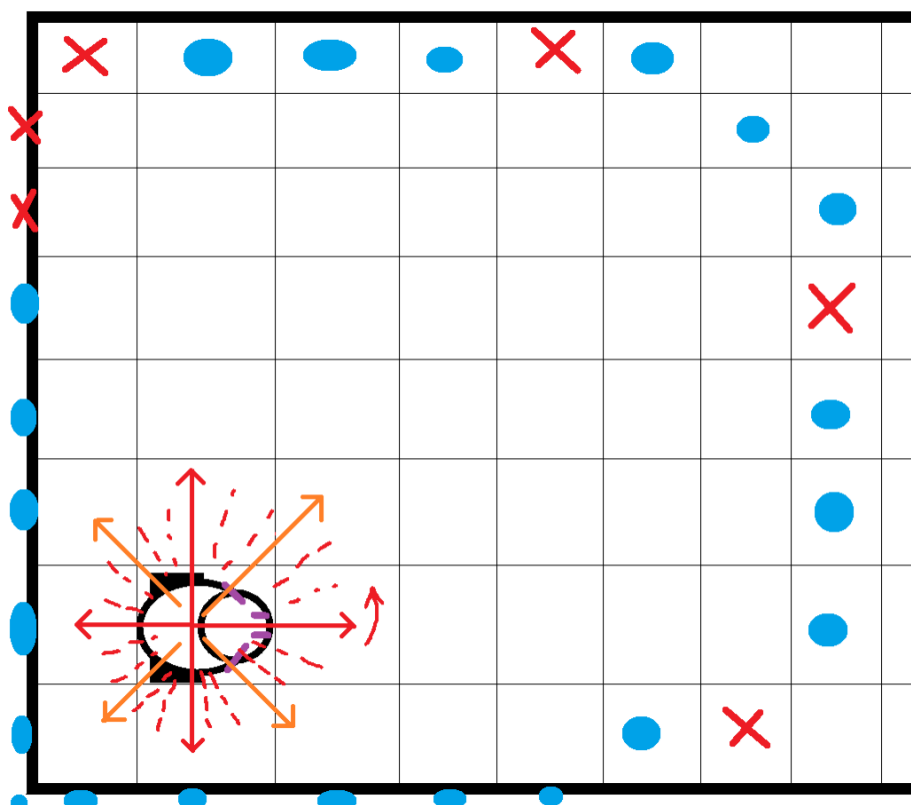


Rys.5.7. Wynik końcowy procesu mapowania

## Wnioski

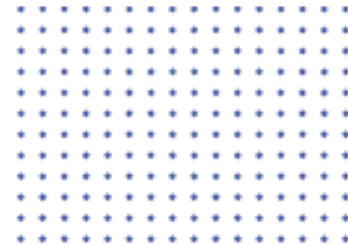
Początkowo mieliśmy problemy z zapisywaniem wartości do tablicy jak i odczytywaniem ich, tzn. Sposób generowania tablicy który wykorzystaliśmy na początku z 2 pętlami for i funkcją "append()" okazał się problematyczny z powodu referencji i powiązań (wynikających z samego działania języka Python). Powodowało to, że zapisanie wartości do jednej komórki, powielalo ją (w tym samym wierszu) w innych kolumnach. Wyświetlając taką tablicę widzieliśmy poziome paski z wartościami np. "O". Problem udało się rozwiązać stosując inną deklarację tablicy, którą udało nam się znaleźć online na portalach związanych z językiem Python.

Jednym z problemów mapowania w ten sposób jest pojawianie przerw w krawędzi obszaru roboczego. Jest to spowodowane między innymi tym, że konieczne jest zastosowanie zaokrągleń podczas przeliczenia współrzędnych. Początkowo stosowaliśmy funkcji "int()" która domyślnie zaokrąślała wszystkie wartości w dół (w rzeczywistości "ucina" ona wartości występujące po przecinku), jednak zamieniliśmy to na zaokrąglanie do najbliższej wartości (funkcja "round"). Nie naprawiliśmy niestety problemu nadpisywania tych samych komórek podczas "mapowania" obręczy przy kątach robota które nie były wielokrotnością 90 stopni.



Rys.5.8. Rysunek przedstawiający przerwy w krawędzi obszaru roboczego (oznaczone jako "x")

# [6] MAPOWANIE ARENY



## Zadanie do wykonania

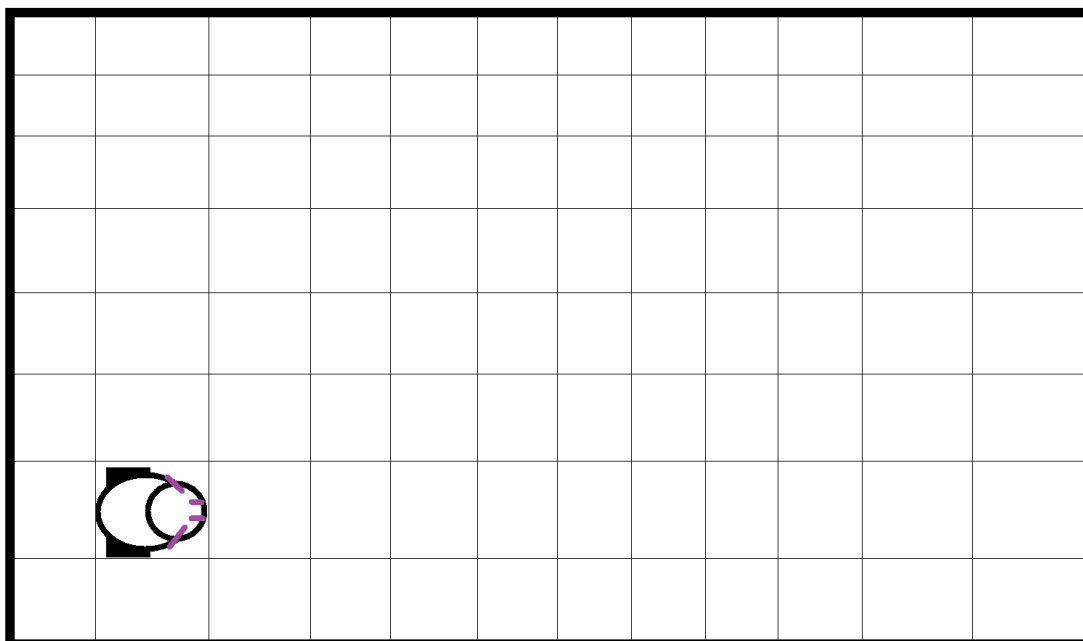
Celem ćwiczenia jest napisać program, za pomocą którego robot Photon będzie tworzył mapę areny z oznaczeniem pozycji robota oraz jego najbliższego otoczenia. Sam robot podczas tego procesu jest stacjonarny. Robot będąc w określonym miejscu ma zmapować najbliższe otoczenie i wyznaczyć swój obszar roboczy. Wynik ma zostać wyświetlony w formie tablicy.

## Kod symulacyjny

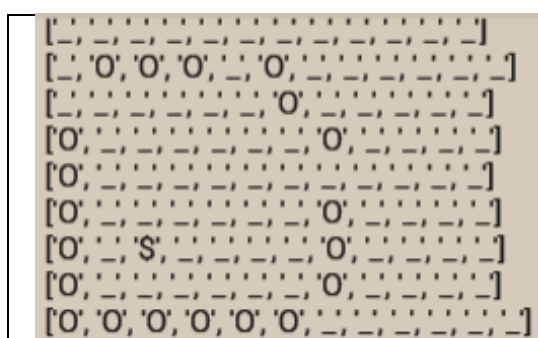
```
import math
i=0
d = 25
x_poz=2
y_poz=2
alfa = 0
k_obr = 15
length = 12
tab = [["_"]*12 for _ in range(9)] #generowanie pustej tablicy 12 x 9
for i in range(8,-1,-1):
    print(tab[i])
##### generowanie tablic /\ | dane pomiarowe \/
tab_x=[5,5,5,4,3,1,0,-1,-2,-2,-2,-2,-2,-2,-2,-1,-1,0,1,1,2,3,5]
tab_y=[0,1,3,4,5,5,5,3,2,1,1,0,-1,-1,-2,-2,-2,-2,-2,-2,-2,-1]
tab_p=[125,125,125,125,125,125,125,125,80,55,43,25,25,43,43,55,43,43,25,43,43,55,80,125]
#####3 WYPEŁNIANIE OBRĘCZY
while alfa<360: #PEŁNY OBRÓT 360 STOPNI
    #pom = photon.get_distance_from_obstacle()
    #pom = pom + length
    pom = tab_p[i] + length
    if pom > 120:
        pom=120
    de_x = math.cos(math.radians(alfa))*(pom) #DELTA X
    de_y = math.sin(math.radians(alfa))*(pom) #DELTA Y
    x_in = max(math.floor(de_x/25) + x_poz,0) #INDEX X TABLICY Z DODANĄ POZYCJĄ
    y_in = max(math.floor(de_y/25) + y_poz,0) #INDEX Y TABLICY Z DODANĄ POZYCJĄ

    if pom>=120:
        tab[y_in][x_in] = "o" #zapisywanie do tablicy wystąpienie krańca zasięgu czujnika
    else:
        tab[y_in][x_in] = "x" #zapisywanie do tablicy wystąpienie przeszkody/ściany
    while pom>=26:
        pom-=25
        de_x = math.cos(math.radians(alfa))*pom
        de_y = math.sin(math.radians(alfa))*pom
        x_in = max(math.floor(de_x/25) + x_poz,0)
        y_in = max(math.floor(de_y/25) + y_poz,0)
        if tab[y_in][x_in] == "_":
            tab[y_in][x_in] = "u"
        alfa += k_obr
    i+=1
#####3
tab[y_poz][x_poz] = "s" #ZAZNACZENIE POZYCJI ROBOTA
print("-----")
for i in range(8,-1,-1):
    print(tab[i])
print("-----")
```

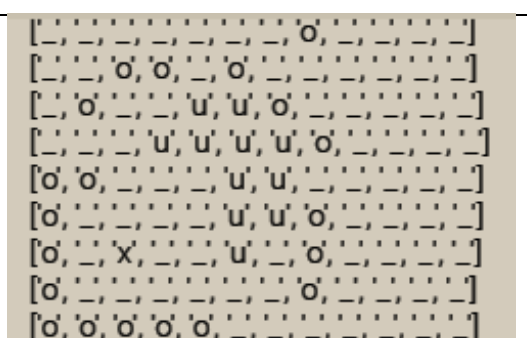
## Realizacja zadania



Rys.6.1. Mapa areny (kwadraty mają wielkość 25x25 cm, rysunek koncepcyjny)

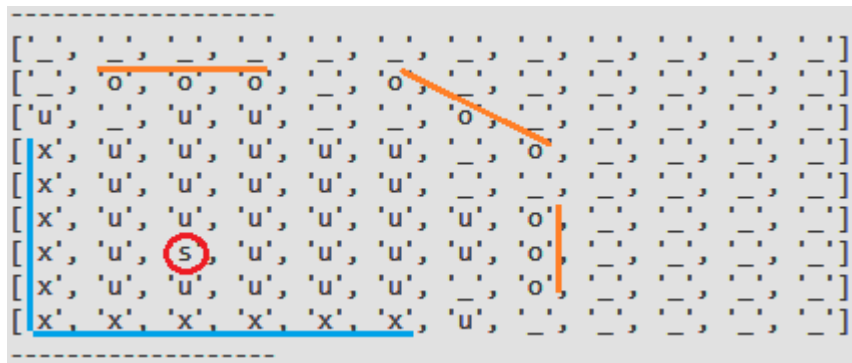


Rys.6.2. Wygenerowana obwód przy pustej arenie i pozycji początkowej.

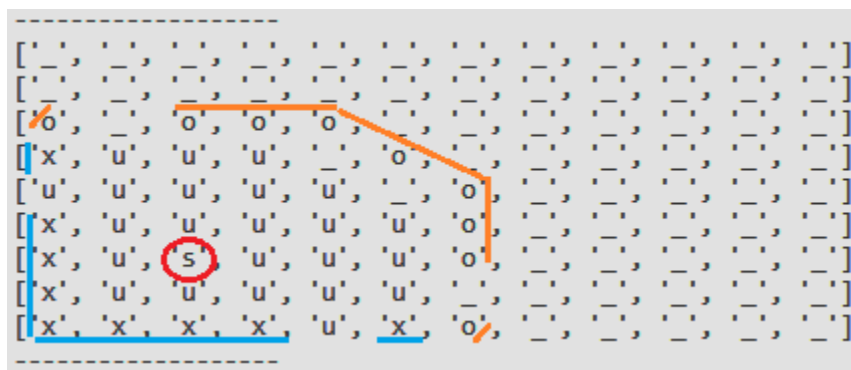


Rys.6.3. Pierwsze podejście do wypełnienia obręczy.

Rysunki nr 6.2 i 6.3 nie pokazują zaznaczonej różnicy między wykrytym obiektem lub ścianą a granicą odczytu czujnika, dodatkowo problemem algorytmu z rysunku nr 6.3 było to, że ograniczał się do wypełniania tylko 1 ćwiartki, w celu wypełnienia całości wymagało to jednak modyfikacji. Z uwagi, że nie udało nam się mimo wielu prób skończyć zadania na zajęciach laboratoryjnych, dlatego przy pomocy danych, zasymulowaliśmy działanie robota (pomiar, odczyty itp.) w warunkach domowych, to też kod z punktu 1 dotyczy tylko tej symulacji a nie kodu robota.

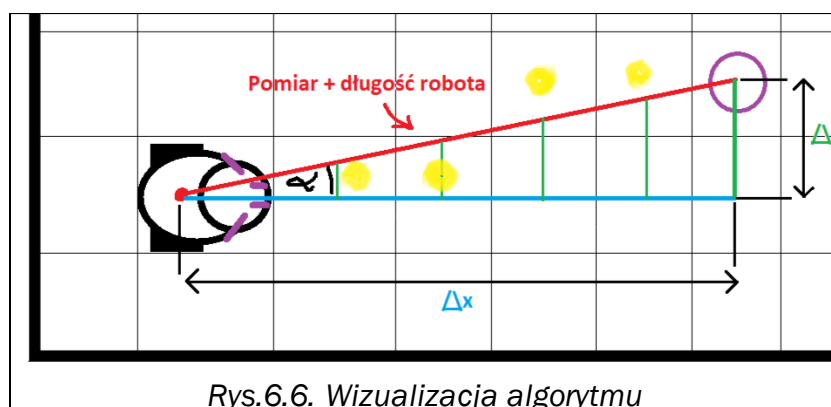


Rys.6.4. Symulacyjne wypełnienie z danymi X, Y i wartościami pomiarów

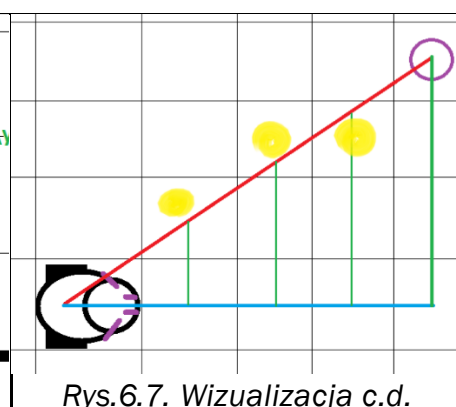


Rys.6.5. Symulacyjne wypełnienie z wartościami pomiarów i liczonych na jej podstawie wartości współrzędnych X i Y.

Początkowo wartości  $\Delta X$  oraz  $\Delta Y$  wczytywaliśmy z tablicy danych pomiarowych, to też nie było w kodzie zbyt wielu obliczeń, wynik takiej symulacji (rys.6.4) zostawiał w tablicy kilka istotnych dziur. Naprawiliśmy to poprzez uzależnienie wartości  $\Delta X$  oraz  $\Delta Y$  od wartości symulowanego pomiaru (ta jak działać to powinno przy pracy z robotem), ta symulacja (rys.6.5) przebiegała lepiej, widocznych było mniej dziur których niestety nie udało się usunąć z uwagi na zbyt zaokrąglone dane pomiarowe które posiadaliśmy.



Rys.6.6. Wizualizacja algorytmu



Rys.6.7. Wizualizacja c.d.

Rysunki 6.6 i 6.7 przedstawiają w sposób graficzny wizualizację jednego kroku z 24 (dla jakiegoś pojedynczego kąta  $\alpha$ ) algorytmu wypełniania. Fioletowe kółko symbolizuje przeszkodę lub ścianę, którą wykrył czujnik, natomiast żółte kropki symbolizują komórki tablicy które zostaną wypełnione przez algorytm. Algorytm po wygenerowaniu granicy wchodzi w pętlę i odejmuje od pomiaru wartość 25 i dla nowej wartości liczy nowe współrzędne X i Y tablicy do wypełnienia (żółte kropki na rysunkach). Pętla kończy się, gdy wartość pomiaru jest mniejsza niż 26, aby nie nadpisać pozycji robota.



## Problemy

<pre> pom = tab_p[i] + length if pom &gt; 120:     pom=120 de_x = math.cos(math.radians(alfa))*(pom) de_y = math.sin(math.radians(alfa))*(pom) x_in = max(math.floor(de_x/25) + x_poz,0) y_in = max(math.floor(de_y/25) + y_poz,0)  pom = pom + length cp1 = math.cos(math.radians(alfa))*(pom/d) #delta x sp1 = math.sin(math.radians(alfa))*(pom/d) #delta Y </pre> <p>Rys.6.8. Porównanie fragmentów kodu</p>	<p>Podczas zajęć, gdy robot był ustawiony w innej pozycji niż [2,2] algorytm wchodził w wartości ujemne i często skutkowało błędem i zatrzymaniem programu. Przy pomocy funkcji max() dbamy o to, aby wartości indexów były zawsze dodatnie. Funkcja max zwraca wartość większą z 2 podanych argumentów, argumentem pierwszym jest index tablicy z dodaną pozycją robota, a drugim wartość zero. Gdy wartość indexu będzie ujemna, funkcja max() wybierze 0 jako największą wartość.</p>
<pre> if pom&gt;=120:     tab[y_in][x_in] = "o" else:     tab[y_in][x_in] = "x"  tab[round(sp1+y1)][round(cp1+x1)] = "o" </pre> <p>Rys.6.9. Porównanie fragmentów kodu</p>	<p>Jako że rozdzieliliśmy wartości DeltaX/Y i IndexyX/Y na 4 różne zmienne, a indexy mają już dodaną wartość pozycji to kod przypisywania wartości do tablicy jest bardziej czytelny i mniej problematyczne są zmiany.</p>

## Wnioski

Mieliśmy duże problemy z trygonometrią i problematyką zadania, dopiero zabranie zadania do domu i przemyślenie go dłużej pozwoliło nam napisać algorytm, który działa. Mimo że dane pomiarowe które zapisaliśmy na zajęciach przypadły, udało nam się aproksymować wartości pomiarów przy pomocy zrzutów ekranu które mieliśmy. Zadanie nie było aż tak trudne, jak się wydawało. Nie udało się wykonać pokazać testu wypełniania z przeszkodą w formie kartonu na środku, niestety nie mieliśmy tego zrzutu ekranu.

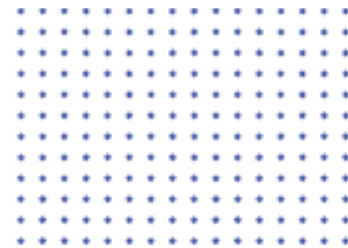
```

##### generowanie tablic /\ | dane pomiarowe \/
tab_x=[5,5,5,4,3,1,0,-1,-2,-2,-2,-2,-2,-2,-2,-2,-1,-1,0,1,1,2,3,5]
tab_y=[0,1,3,4,5,5,5,5,3,2,1,1,0,-1,-1,-2,-2,-2,-2,-2,-2,-2,-1]
tab_p=[125,125,125,125,125,125,125,125,80,55,43,25,25,43,43,55,43,43,25,43,43,55,80,125]

```

Rys.6.10. Zasymulowane dane pomiarowe, tablice tab\_x i tab\_y są nieużywane w kodzie.

# [7] MAPOWANIE CAŁEJ ARENY WRAZ Z PRZESZKODAMI



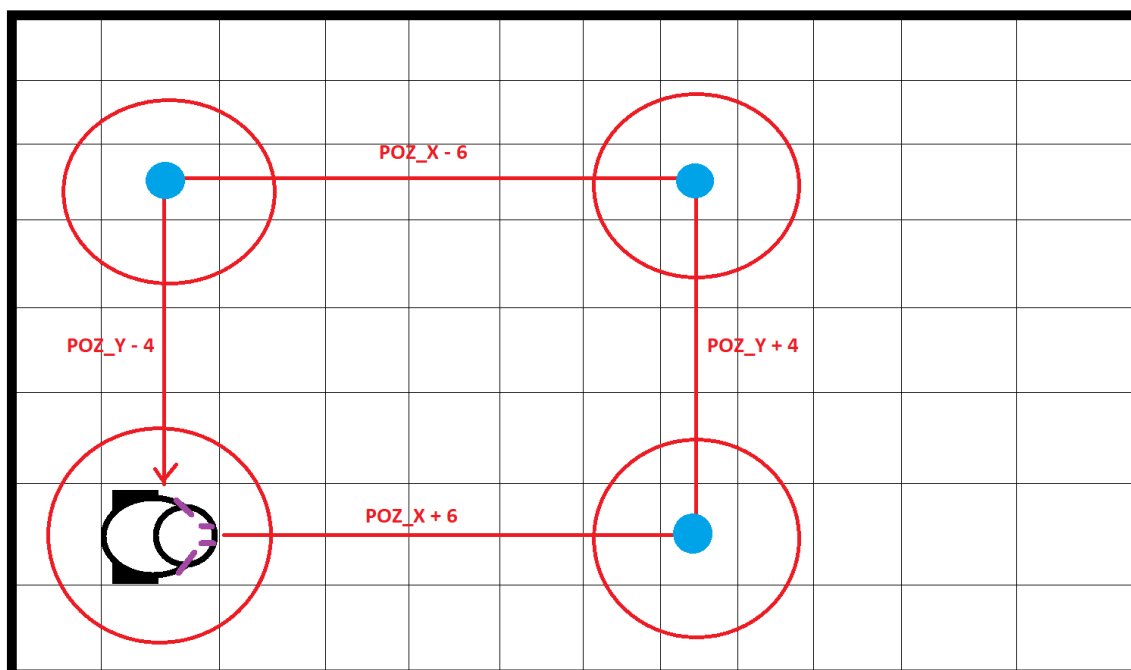
## Zadanie do wykonania

Celem ćwiczenia jest modyfikacja programu wykonanego dotychczas (mapowanie terenu w najbliższym otoczeniu robota) i sprawienie, aby Photon zmapował całą arenę poruszając się po niej i powrócił do pozycji startowej. Robot zaczynając z określonego miejsca powinien zmapować całą arenę wraz z występującymi na niej przeszkodami i powrócić do pozycji startowej.

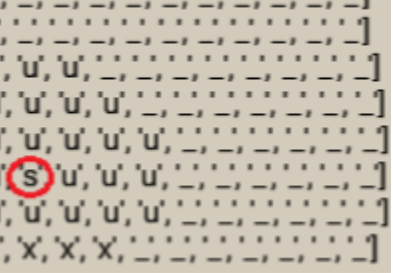
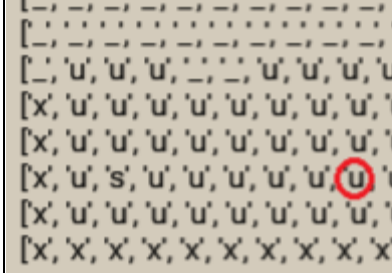
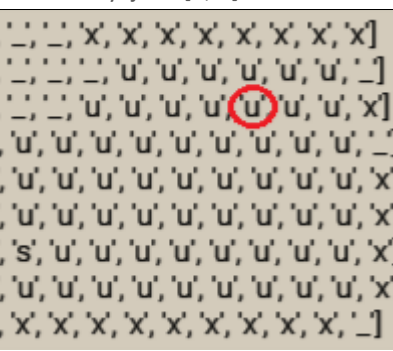

## Kod robota Photon

<pre> from photonrobot import * import math photon.change_color(Colors.blue, "ears") photon.change_color(Colors.red, "eyes")d = 25 x_poz=2 y_poz=2 alfa = 0 k_obr = 15 length = 12 tab = [["_"]*12 for _ in range(9)] def mapa(x_poz, y_poz, tab, alfa): def poka(tab): def Jazda(poz_x,poz_y,tryb): tab=mapa(x_poz,y_poz,tab,0) tab[y_poz][x_poz] = "s" poka(tab) ##### dod_x=6 dod_y=0 Jazda(dod_x,dod_y,"index") x_poz += dod_x y_poz += dod_y print(x_poz,"   ",y_poz) tab = mapa(x_poz,y_poz,tab,0) poka(tab) ##### dod_x=0 dod_y=4 Jazda(dod_x,dod_y,"index") x_poz += dod_x y_poz += dod_y print(x_poz,"   ",y_poz) tab = mapa(x_poz,y_poz,tab,0) poka(tab) ##### dod_x=-6 dod_y=0 Jazda(dod_x,dod_y,"index") x_poz += dod_x y_poz += dod_y print(x_poz,"   ",y_poz) tab = mapa(x_poz,y_poz,tab,0) poka(tab) ##### dod_x=0 dod_y=-4 Jazda(dod_x,dod_y,"index") </pre>	<pre> def poka(tab): print("-----") for i in range(8,-1,-1): print(tab[i]) print("-----")  def Jazda(poz_x,poz_y,tryb): pom = photon.get_distance_from_obstacle() if tryb=="index": poz_x=poz_x*25 poz_y=poz_y*25 if poz_x&gt;0: photon.go_forward(poz_x, 25) elif poz_x&lt;0: photon.go_backward(-poz_x, 25) if poz_y&gt;0: photon.rotate_left(87, 25) photon.go_forward(poz_y, 25) photon.rotate_right(87, 25) elif poz_y&lt;0: photon.rotate_right(87, 25) photon.go_forward(-poz_y, 25) photon.rotate_left(87, 25)  def mapa(x_poz, y_poz, tab, alfa): while alfa&lt;360: #PEŁNY OBRÓT 360 STOPNI pom = photon.get_distance_from_obstacle() pom = pom + 12#length if pom &gt; 120: pom=120 de_x = math.cos(math.radians(alfa))*(pom) de_y = math.sin(math.radians(alfa))*(pom) x_in = max(math.floor(de_x/25) + x_poz,0) y_in = max(math.floor(de_y/25) + y_poz,0) if not (pom&gt;=120): tab[y_in][x_in] = "x" while pom&gt;=26: pom-=25 de_x = math.cos(math.radians(alfa))*pom de_y = math.sin(math.radians(alfa))*pom x_in = max(math.floor(de_x/25) + x_poz,0) y_in = max(math.floor(de_y/25) + y_poz,0) if tab[y_in][x_in] == "_": tab[y_in][x_in] = "u" alfa += 15 photon.rotate_left(12, 25) return tab </pre>
--	---

## Realizacja zadania



Rys.7.1. Mapa areny plan jazdy robota w teście (kwadraty mają wielkość 25x25 cm)

	
<p>Pozycja -&gt; [2, 2]</p>	<p>Pozycja -&gt; [8, 2]</p>
	
<p>Pozycja -&gt; [8, 6]</p>	<p>Pozycja -&gt; [2, 6]</p>

*Tab.7.1. Zarejestrowane etapy mapowania całej areny w jednym cyklu działania programu*

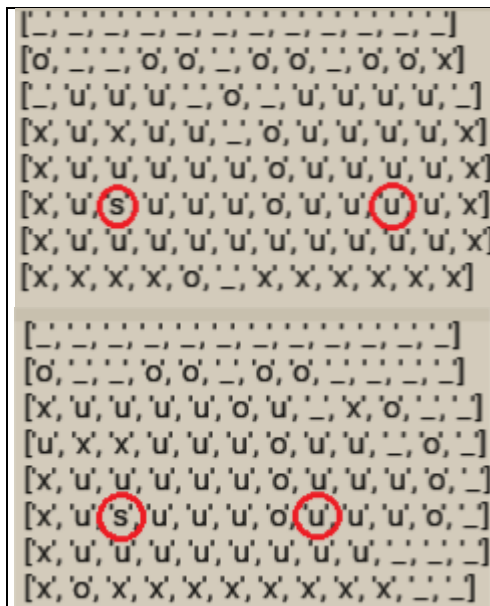
Celem zadania było zmapowanie całej areny i możliwych przeszkód, które mogły się na niej znajdować, poprzez zmianę pozycji i kilkakrotne mierzenie. Zadanie wykonaliśmy z 4 pomiarami, robot poruszał się po małym prostokącie widocznym na rys. 7.1.

Działanie kodu nie zmieniło się znacząco względem wcześniejszego sprawozdania, natomiast żeby wykonywać kilka mapowań i zmian pozycji robota wymagane było zastosowanie funkcji, aby kod nie zawierał paru-set linijek i był czytelny dla wszystkich.

Zastosowaliśmy 3 funkcje:

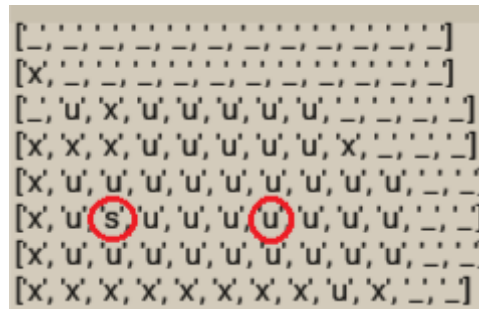
- **Poka(tablica)** - Funkcja wyświetla w okienku terminalu stan tablicy, jako argument przyjmuje tablicę, nie zwraca żadnej wartości
- **Jazda(x,y,tryb)** - Funkcja służy do zmiany pozycji robota na arenie, argument "x" oznacza o ile ma przejechać robot w osi X, argument "y" oznacza o ile ma przejechać robot w osi Y, argument tryb jest to rodzaj przełącznika między tym czy wartości X oraz Y mają być w postaci indexów czy odległości w centymetrach. Najlepiej jest zostawić w trybie "index", funkcja nie zwraca żadnej wartości
- **Mapa(x,y,tab,alfa)** - Funkcja służy do mapowania przestrzeni, w której znajduje się robot. Argumentami "x" i "y" jest aktualna pozycja robota, argument tab to tablica, na której mają być zapisywane zmiany w otoczeniu, argument alfa jest kąt nachylenia robota w jakiej się znajduje obecnie (jeżeli jest ustawiony w kierunku tak jak na rys.1. to wartość najlepiej zostawić jako 0). Funkcja zwraca wartość tablicy.

## Problemy



Rys.7.2. Problemy mapowania

Podczas zajęć, gdy robot po zmapowaniu przestrzeni (oraz pierścienia) w jednej współrzędnej przechodził w drugą, to zdarzało mu się nadpisywać wartość tablicy o kolejny pierścień na przestrzeni już zmapowanej. Problem próbowaliśmy naprawić dodając nowy warunek w generacji pierścienia co nie wyszło nam do końca. Problem rozwiązaliśmy finalnie poprzez usunięcie generacji pierścienia i skrócenia zasięgu mapowania.

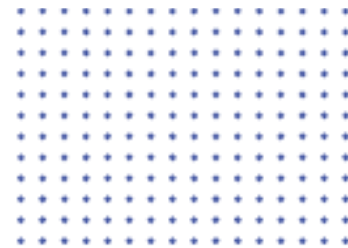


Rys.7.3. Mapowanie po naprawie kodu

## Wnioski

Kod nie był trudny do wykonania, właściwie posiadaliśmy wszelkie umiejętności i wstępny kod więc wyrobiliśmy się z zadaniem w ciągu jednych zajęć. Początkowo wykonaliśmy bardzo głupi błąd z obracaniem się robota w funkcji Jazda(), gdzie uznaliśmy, że kąt 90 stopni w rzeczywistości to będzie 12\*6 czyli 72 stopnie w kodzie robota, co skutkowało, że robot tracił kąt i wylądowywał w błędnej pozycji. Uznaliśmy, że tak ma być, ponieważ gdy robot mapował przestrzeń wokół niego to obracał się o 15 stopni na pomiar co w kodzie widniało jako 12 stopni. Ten błąd został szybko naprawiony przez nas. Nie dodaliśmy jednej funkcji, na której nam zależało, mianowicie podróżowania robota po przekątnych względem ścian, aby w ten sposób skracać trasę.

# [8] WYZNACZANIE TRASY - PROPAGACJA FALI



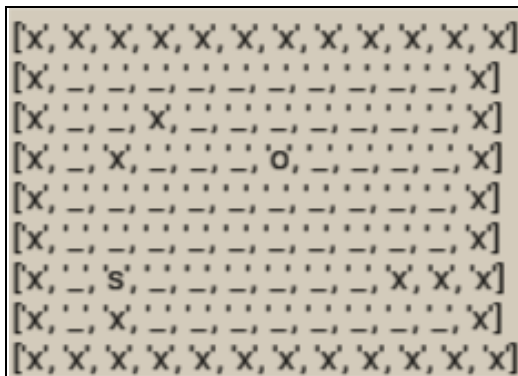
## Zadanie do spełnienia

Celem ćwiczenia jest modyfikacja programu wykonanego dotychczas (mapowanie terenu w najbliższym otoczeniu robota) i sprawienie, aby Photon zmapował całą arenę, skończył w danym punkcie i na podstawie danych które posiada wyznaczył sobie trasę bez kolizji do punktu startowego., tzn. wykorzystać algorytm propagacji fali, aby wyznaczyć trasę do danego punktu.

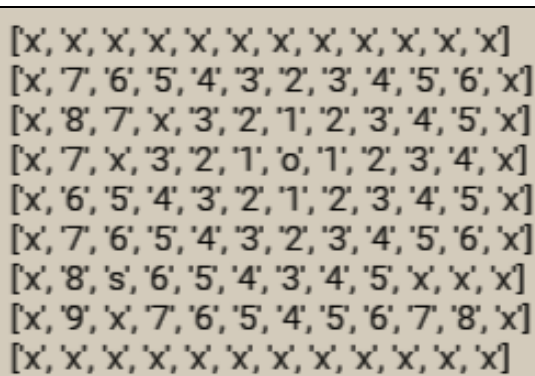
## Kod robota Photon

<pre>def wypelnij(tab,ile):     for i in range(9):         tab[i][0] = znak         tab[i][11] = znak     for j in range(12):         tab[0][j] = znak         tab[8][j] = znak     for k in range(ile):         x = random.randint(1, 10)         y = random.randint(1, 7)         tab[y][x] = znak</pre>	<pre>def Nast_Komorka(x,y):     return{         "x+": [x+1,y],         "x-": [x-1,y],         "y+": [x,y+1],         "y-": [x,y-1]     }.values() def poka(tab):     print("-----")     for i in range(8,-1,-1):         print(tab[i])     print("-----")</pre>
<pre>#from photonrobot import * import random znak = "x" y_poz = 2 #pozycja X startowa x_poz = 2 #pozycja Y startowa tab = [["_"]*12 for _ in range(9)]  xx=6 #pozycja X aktualna yy=5 #pozycja Y aktualna wypelnij(tab,15) tab[yy][xx]="o" tab[y_poz][x_poz]="s" poka(tab) tab = propagacja(tab,xx,yy) poka(tab) tab = trasa(tab,x_poz,y_poz) poka(tab) i, j = 0, 0 for wiersze in tab:     j=0     for w in wiersze:         if w not in ["x","s","_", "o","#"]:             tab[i][j]="_"             j+=1         i+=1     poka(tab)</pre>	<pre>def propagacja(tab,x,y):     i=1     wiktor = []     tmp = []     for next_xy in Nast_Komorka(x, y):         next_x, next_y = next_xy         if tab[next_y][next_x] == "_":             tab[next_y][next_x] = str(i)             tmp.append([next_x, next_y])    print(wiktor)     while i&lt;20:         wiktor = tmp         tmp = []         i+=1         for xy in wiktor:             x, y = xy             for next_xy in Nast_Komorka(x, y):                 next_x, next_y = next_xy                 if tab[next_y][next_x] == "_":                     tab[next_y][next_x] = str(i)                     tmp.append([next_x, next_y])    wiktor = []     return tab#, maks</pre>
<pre>def trasa(tab,x,y):     #x,y &lt;--- START     znaczek = "#"     wiktor = []     min = 25     while tab[y][x] != "o":         for next_xy in Nast_Komorka(x, y):             next_x, next_y = next_xy             if tab[next_y][next_x] == "o":                 x, y = next_x, next_y                 return tab             elif tab[next_y][next_x] not in ["x","s",znaczek,"_"]:                 if int(tab[next_y][next_x]) &lt; min:                     tmp_x, tmp_y = next_x, next_y                     min = int(tab[next_y][next_x])                     #print(min)                 x = tmp_x                 y = tmp_y             tab[y][x]=znaczek     return tab</pre>	

## Realizacja zadania



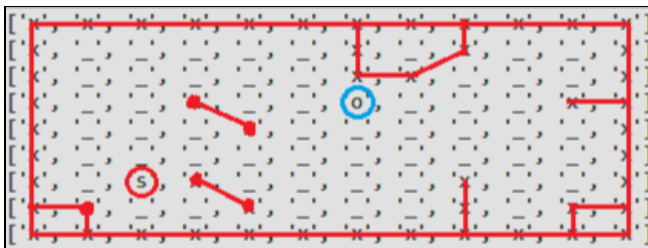
Rys.8.1. Generowanie areny na Photon'ie



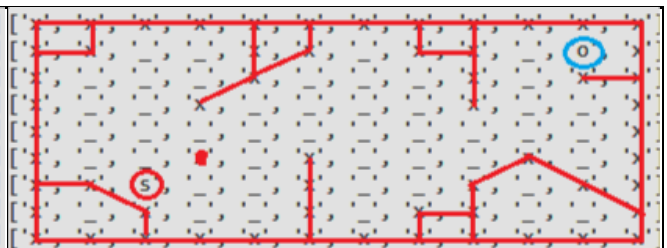
Rys.8.2. Propagacja fali na robocie Photon

Wynik naszych prób na zajęciach

Kod symulacyjny który napisaliśmy składa się 3 głównych etapów: tworzenia pustej mapy wraz z sztucznymi przeszkodami, propagacji fali i na końcu wyznaczenia trasy.

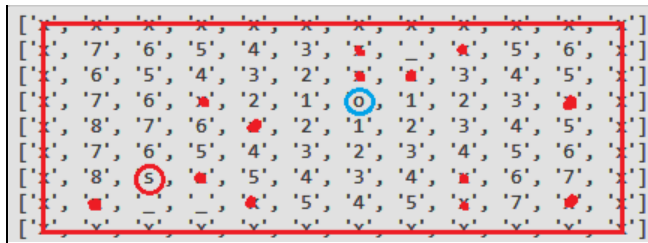


Rys.8.3. Wygenerowana arena - 1

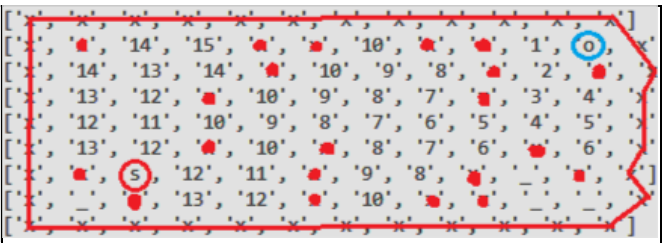


Rys.8.4. Wygenerowana arena - 2

**Etap 1:** Wygenerowana pusta arena z losowo umieszczonymi przeszkodami.

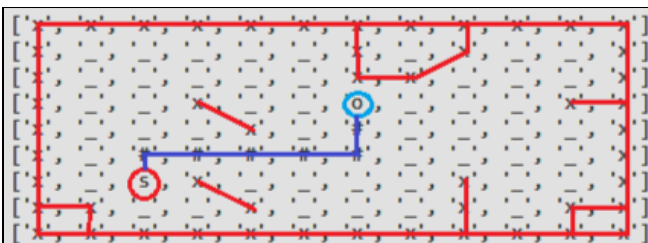


Rys.8.5. Propagacja fali - 1

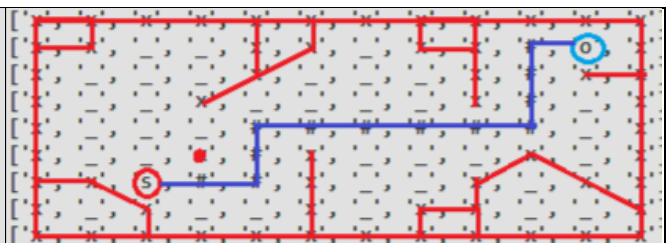


Rys.8.6. Propagacja fali - 2

**Etap 2:** Algorytm propagacji fali od pozycji robota (niebieski) do pozycji docelowej (czerwony).



Rys.8.7. Wyznaczanie trasy - 1



Rys.8.8. Wyznaczanie trasy - 2

**Etap 3:** Wyznaczanie bezkolizyjnej trasy

Do wykonania zadania w sposób symulacyjny wykorzystaliśmy następujące funkcje:

- **Poka(tablica)** - Funkcja wyświetla w okienku terminalu stan tablicy, jako argument przyjmuje tablicę, nie zwraca żadnej wartości.
- **Wypełnij(tab,ile)** - Funkcja wypełnia pustą tablicę obramowaniem w postaci znaku "x" oraz losowo wypełnia arenę przeszkodami (znak "x") tyle razy, ile poda się w argumencie "ile". Z uwagi, że liczby są pseudolosowe to argument "ile" nie zawsze wypełni arenę tyloma "x'ami" ile podał użytkownik, ponieważ czasami nadpisuje pozycje już wypełnione. Funkcja zwraca wypełnioną tablicę.
- **Nast\_Komorka(x,y)** - Prosta funkcja zwracająca tablicę zawierającą współrzędne punktów sąsiednich od podanych jako argument współrzędnych [x,y]. Punkty sąsiednie oznaczają pozycję w górę/dół oraz lewo/prawo od punktu [x,y].
- **Propagacja(tab,x,y)** - Funkcja, która przypisuje wagę do pustych pól, czyli algorytm propagacji fali. Wykorzystuje wcześniejszą funkcję do mapowania wag do poszczególnych pustych pól. Zaczynając od pozycji [X, Y] wyznacza falę, pierwsze sąsiednie wartości mają wagę 1, późniejsze 2 itd.. Algorytm ignoruje oraz omija ściany i przeszkody, zwraca tablicę z zaznaczonymi wagami.
- **Trasa(tab,x,y)** - Funkcja, która na podstawie pozycji końcowej określa trasę. Algorytm nie znajduje natomiast najkrótszej drogi, tylko drogę ogólnie. Funkcja działa w następujący sposób, z pozycji [X, Y] przy pomocy funkcji Nast\_Komorka() wybiera współrzędną której waga jest najmniejsza (jeżeli jest ich kilka to wybierze się ta pierwsza którą zapisał algorytm) i nadpisuje to miejsce znacznikiem trasy, czyli "#". Przechodzi do tej współrzędnej i powtarza szukanie do momentu aż nie trafi na pozycję "o" czyli pozycji aktualnej robota. Funkcja zwraca tablicę.

<pre>i, j = 0, 0 for wiersze in tab:     j=0     for w in wiersze:         if w not in ["x", "s", "_", "o", "#"]:             tab[i][j]="_"             j+=1         i+=1 poka(tab)</pre>	Fragment kodu, który nie jest funkcją, ale mógłby być. Służy do czyszczenia tablicy z wag po propagacji fali, zostawiając przy tym przeszkody i znaczniki.
---	--

Rys.8.9. Fragment kodu czyszczącego tablicę



## Problemy

```
-----
Traceback (most recent call last):
  File "c:\Users\pyro\Desktop\PYTHON\Wyznaczanie trasy\propagacja.py", line 119, in <module>
    tab = trasa(tab,x_poz,y_poz)
          ^^^^^^^^^^^^^^^^^^^^^
  File "c:\Users\pyro\Desktop\PYTHON\Wyznaczanie trasy\propagacja.py", line 104, in trasa
    x = tmp_x
        ^^^^^
UnboundLocalError: cannot access local variable 'tmp_x' where it is not associated with a value
PS C:\Users\pyro> █
```

Rys.8.10. Błąd w wyznaczaniu trasy – niemożliwe wyznaczenie.

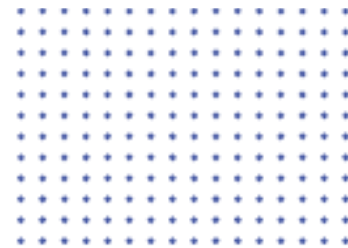
Czasami symulacyjne generowanie przeszkód tworzyło arenę, w której nie możliwe było wyznaczenie jakiegokolwiek trasy. Problemu nie naprawiliśmy z uwagi, że na zajęciach najprawdopodobniej nie doszłoby do takiej sytuacji, oraz że to sztuczne wypełnianie areny powoduje ten problem.

## Wnioski

O ile udało nam się sporządzić algorytm propagacji fali i wyznaczania na jej podstawie ścieżki nadal w kodzie brakuje implementacji tej opcji do robota Photon tzn. Tablica posiada zaznaczoną ścieżkę, ale nic to nie daje robotowi. Napisana wcześniej (wcześniejsze sprawozdanie) funkcja Jazda(x,y,tryb) przyjmuje wartości X, Y jako odległości do przemieszczenia, w naszym kodzie brakuje jeszcze ok.1-2 funkcje, które na podstawie trasy (tablicy) zwracałyby wektor współrzędnych i przetwarzały go na małe odcinki dróg, tak aby można było wykorzystać w funkcji Jazda(). Dodatkowo w celach optymalizacji kodu, mogliśmy zmienić fragment kodu z rys.9. jako małą funkcję.



# [9] TURNIEJ SUMO



## Zadanie do spełnienia

Celem ćwiczenia jest napisanie programu, aby Photon, utrzymał się na arenie i wypchnął robota przeciwników poza ring, czyli obszar walki. Do zadania można wykorzystać dostępne czujniki robota Photon. Każdy walczy z każdym po trzy razy i wygrywa ta drużyna, która zdobędzie najwięcej zwycięstw. Zadaniem jest wygrać turniej, spychając przeciwników samemu pozostając na arenie.

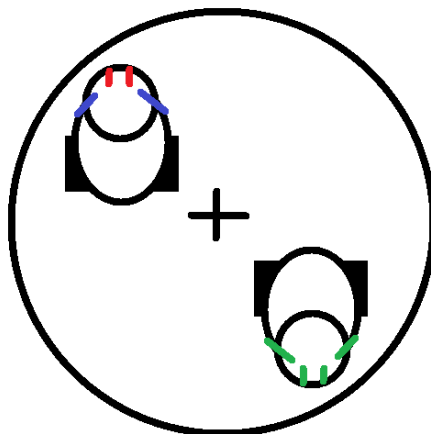
## Kod robota Photon

```
from photonrobot import *
from photonrobot import *
import time
photon.change_color("blue",ColorMode.ears)
photon.change_color("red",ColorMode.eyes)
while True:
    photon.rotate_right(90, 70)          # obrót o 90 stopnie w prawo
    photon.go_forward(27, 80)            # Jazda do przodu na 27cm

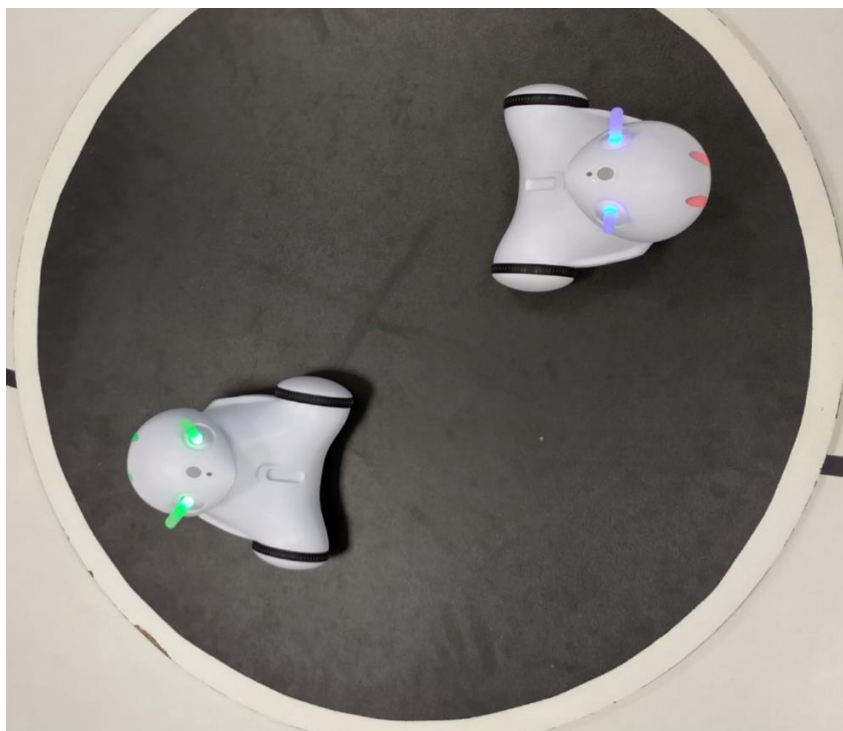
from photonrobot import *
import time
photon.change_color("blue",ColorMode.ears)
photon.change_color("red",ColorMode.eyes)
photon.rotate_right(135, 70)            # Początkowy obrót by ustawić się przodem do przeciwnika
while True:
    photon.rotate_right_infinity(15)     # Obrót w celu wykrycia przeciwnika
    d = photon.get_distance_from_obstacle() # Pomiar odległości
    if d<=40:
        photon.stop()
        d = photon.get_distance_from_obstacle()
        while d<=40:
            d = photon.get_distance_from_obstacle()
            photon.go_forward(25, 80)     # Jazda do przodu
            photon.go_backward(10, 80)    # Niewielki Powrót
```

## Realizacja turnieju

Walka odbywała się na okrągłej arenie a roboty były ustawione w przeciwległych ćwiartkach spozycjonowane były tyłem, równoległe do siebie (Rys. 9.1)



Rys.9.1. Rysunek poglądowy areny - dojo  
41

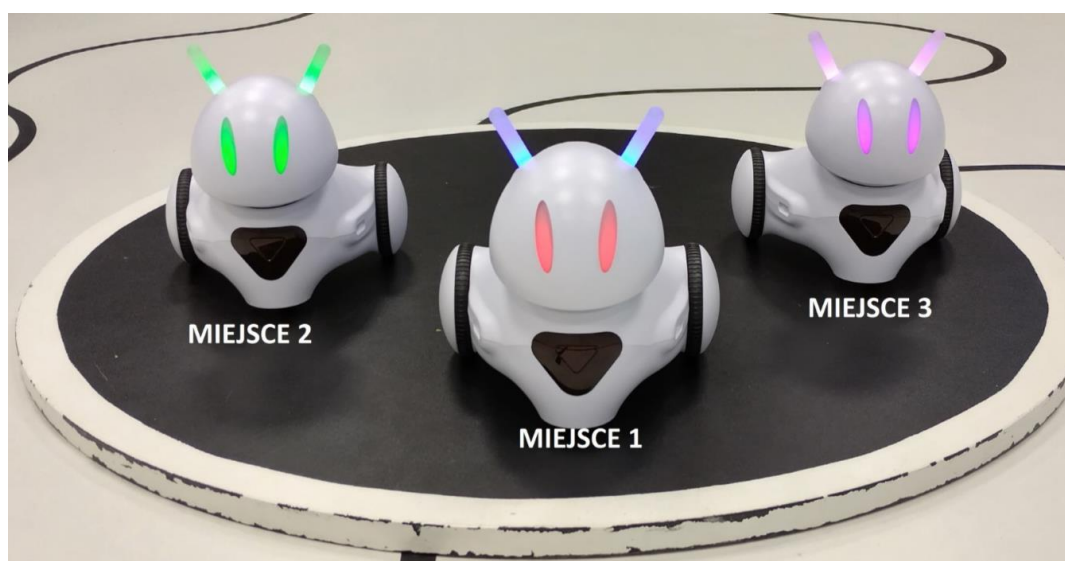


Rys.9.2. Widok rzeczywisty areny - dojo

W walkach losowo wybieraliśmy jeden z dwóch kodów/taktyk na walkę:

- **Taktyka "kwadratu"**, która polegała na poruszaniu się po arenie po wyznaczonym przez nas kwadracie jednocześnie spychając przeciwnika z drogi. Dużym atutem tej taktyki był początkowy moment walki, gdzie nasz robot od razu jechał w bok uciekając przeciwnikowi i powodując, że dużo czasu marnował na szukanie naszego robota. Wadą tej taktyki jest to, że robot porusza się po wyznaczonej przez nas trasie i w ogóle nie reaguje na to co się dzieje na arenie.
- **Taktyka "klasyczna"**, która polega na znalezieniu przeciwnika i poruszanie się do przodu w celu jego zepchnięcia a następnie niewielki powrót, aby samemu nie wypaść z areny.

## Podium



Rys.9.3. Podium robotów wraz z zaznaczonymi miejscami

## Tabela wyników

	379	380	396
379		2:1	3:0
380			3:0
396			

Numery odpowiadają identyfikatorom poszczególnych robotów Photon.

## Wnioski

Nasza wygrana okazała się co prawda zaskoczeniem, ponieważ nie mieliśmy zaawansowanego kodu, czy jakiś wydajnych algorytmów do walki. Dużym szczęściem i wpływ na wygraną miały błędy przeciwników w trakcie walk, wygraliśmy turniej, lecz po poprawie kodu przez grupę przeciwną zrobiliśmy krótki rewanż, który przegraliśmy. Przyznajemy, że nasz kod był mniej optymalny, a grupa z drugiego miejsca przewidziała więcej rzeczy i posiadała szybszy algorytm.

W napisanych przez nas algorytmach oprócz tak zwanej taktyki “kwadratu”, mieliśmy zaimplementować taki kod, żeby robot Photon poruszał się po kształcie dowolnego wielokąta gwiaździstego (np. Pentagramu, octagramu) lub wielokąta foremego (np. pentagonu, octagonu) której liczbę wierzchołków na starcie określałby użytkownik, a Photon na tej podstawie sam wyliczałby trasę, tak żeby skutecznie unikać przeciwnika lub atakować go z niespodziewanych kątów. Brakowało nam jedynie czasu na wykonanie takiego kodu wraz ze zrozumieniem matematycznym, geometrycznym i problematyką.