POLITECHNIKA WROCŁAWSKA

Projekt i implementacja systemu kooperacji autonomicznych robotów mobilnych w zadaniach jednoczesnej lokalizacji i mapowania

Grzegorz Kiernozek
Jan Krzywda
Michał Rogowski

2014-11-24

Spis treści

1. Wstęp.

Projekt ten jest realizowany przez studentów kierunku Inżynieria Systemów na wydziale Informatyki i Zarządzania Politechniki Wrocławskiej. Dotyczy problemów sterowania autonomicznymi robotami mobilnymi w celu stworzenia obrazu odwzorowania terenu, który przebędą. Temat ten poruszany jest szeroko w literaturze pod hasłem SLAM.

1.1 Opis dziedziny

1.1.1 Początki problemu

Akronim SLAM pochodzi od angielskiego tłumaczenia problemu symultanicznej lokalizacji i mapowania. Terminem tym, jak podaje Hugh Durrant-Whyte[i], będący pionierem w opisywanej dziedzinie, określana jest jako *„próba zbudowania mapy nieznanego terenu z użycie mobilnego robota postawionego początkowo w nieznanej lokalizacji i nieznanym otoczeniu”*. Historia rozpoczęła się jednak już w roku 1986, kiedy to R.C. Smith, M. Self i P. Cheesman w swej pracy[ii] zaprezentowali podejście: *„pozwalające na reprezentację szczątkowych informacji, zwanych mapą stochastyczną i łączenie ich dzięki przeglądaniu stworzonych już map i pozyskiwaniu nowych.”.* We wstępie podkreślali oni inspirację pracami Taylora(1976)[iii] i Brooksa (1982)[iv] w których przedstawione zostały sposoby liczenia niepewności, odkrytego przez robota terenu, metodami analitycznymi. W dalszej części artykułu Panowie Smith, Self i Cheesman zaproponowali wyróżnienie trzech przestrzennych współrzędnych(x, y, θ) jako przestrzeni parametrów mobilnego robota, co stało się częścią wspólną znakomitej większości prac w czasie późniejszym.

1.1.2 Podejście probabilistyczne

W ogólnym zamyśle opisane zostało podejście, z czasem nazwane probabilistycznym, polegające na korekcie wcześniej uzyskanych danych aktualnymi odczytami z czujników. W podsumowaniu zaznaczyli oni także „This theory can be used, for example, to compute in advance whether a proposed sequence of actions (each with known uncertainty) is likely to fail due to too much accumulated uncertainty”.

1.1.3 Zastosowania

Z czasem rozwój techonologii oraz powszechność narzędzi takich jak GPS postawił znak zapytania przy ocenie potrzeby pracy nad postawionym w 1986r. problemie. Jak się jednak okazało na potrzebę dalszego rozwijania problemu, uwagę zwracają w swoich pracach: A. I. Mourikis, S. I. Roumelitios[v]. Przypominają oni, że GPS nie jest dostępny wszędzie, a mobilne roboty wykorzystywane są do specjalistycznych zadań, jak np. eksploracja przestrzeni kosmicznej, morskich głębin, eksploracja wraku Titanica opisana przez R. Eustice’a, H. Singh’a, J. Leonarda’a, M. Waltera’a i R. Ballard’a[vi] czy wnętrza ludzkiego organizmu (Cheen, Lopez, Soria, Sciascio, Pereira, Carelli)[vii].

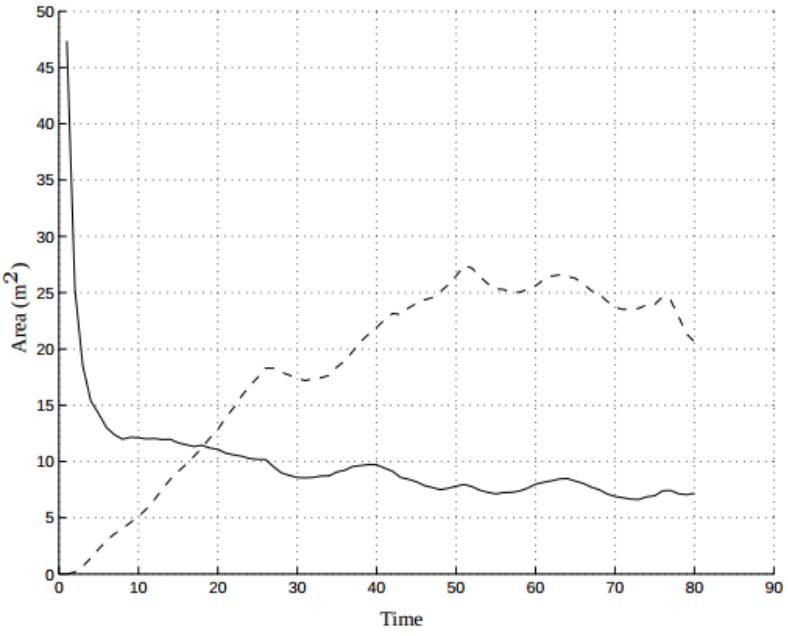
1.1.4 Nowoczesny model probabilistyczny.

Model probabilistyczny z czasem został rozwinięty, co opisane zostało w książce Larsa A. A. Anderssona[viii], który, uzupełnia podejście probabilistyczne o kooperację dwóch robotów.

Podstawowym narzędziem budowania mapy jest tutaj sieć bayesowska. Dodatkowo model teoretyczny, uzupełnia on opisem realizacji procesu zbierania informacji, o którym pisze: *„The purpose of sensors is to gather information about how the system behaves and how this corresponds to what is predicted using motion and observation models”*, a także opis możliwych realizacji kinetycznego ruchu robota.

1.1.5 Kooperacja robotów

Podejście Anderssona z czasem zyskało nazwę C-SLAM, którą wytłumaczyć można kooperacją dwóch, lub więcej robotów realizującą założenia problem SLAM. Takie podejście prezentowane jest przez wielu autorów, wspomnieć należy tutaj o pracy M. Di Marco[ix], w której prezentowany jest algorytm fuzji map jako minimalizacja problemu programowania liniowego. Utwierdzone zostaje przypuszczenie, że błędy podczas mapowania zostają wyraźnie zmniejszone dzięki zastosowaniu wielu robotów. Doskonale widać to na wykresie:



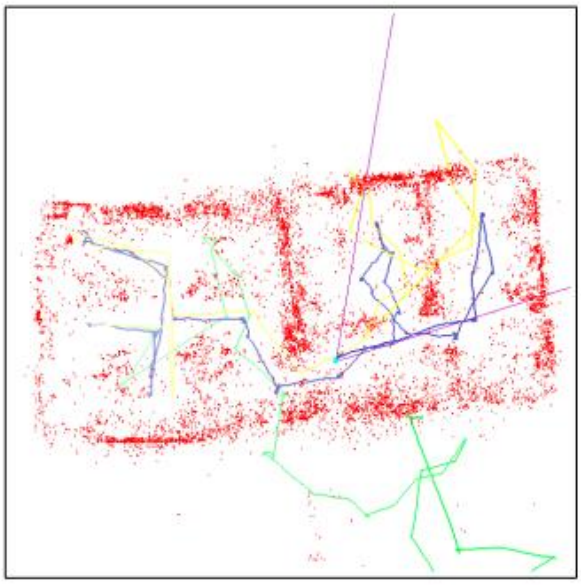
Rysunek 1 Źródło [IX] Porównanie niepewności pojedynczego robota w układzie samodzielnego robota(linia przerywana) i układzie wielu robotów (ciągła) uśredniony po 300 eksperymentach.

Widzimy, że po niepewności asymptotycznie dążącej do nieskończoności dla zerowego czasu, ulega ona zmniejszeniu i ustabilizowaniu, co wynika z licznych spotkań robotów i korekcji ich pozycji.

1.1.6 Zaawansowane metody matematyczne

Z czasem rozwiązania problemu SLAM zaczęto poszukiwać w bardziej skomplikowanych aparatach matematycznych. Na specjalną uwagę zasługuję zastosowanie metody Monte Carlo, stworzonej przez polskiego matematyka Stanisława Ulama w końcówce lat 40. XX wieku. Dla zadań lokalizacji użyteczna jest jej wersja zwana filtrem cząsteczkowym, zapostulowana w 1996 roku przez dwóch naukowców z Harvardu, J. Liu i R. Chen'a[x]. Co ciekawe, abstrakcie swojej pracy opisują oni zastosowania w ekonomii i inżynierii. Pomysł ten został z sukcesem zaszczepiony do problemu SLAM

przez: N. Fairfield'a[xi] (Problem SLAM w zatopionych tunelach), a także A. Howard'a[xii]. Przykładowy obraz utworzony z użyciem filtru cząsteczkowego ma postać:



Rysunek 2 Źródło[XII] Obraz uzyskany po przejeździe robota na czerwono zaznaczone są wyznaczone częki cząsteczką prawdopodobne przeszkody.

Warto wspomnieć, że ten ostatni dużą uwagę poświęca obsłudze spotkania dwóch mapujących robotów, cytując:

"In this latter case, we assume that pairs of robots will eventually 'bump into' one another, thereby determining their relative pose. We use this relative pose to initialize the filter, and combine the subsequent (and prior) observations from both robots into a common map."

Podejście to stało się inspiracją dla spotkań robotów w naszym projekcie. Uzupełniając, warto dodać iż alternatywą dla bazującego na kamerze filtra cząsteczkowego jest sonar, wykorzystujący fale radiowe jak ma to miejsce w pracy L. Kleeman'a[xiii]

Naturalnie szerokie zastosowanie mają także powszechnie znane metody, zaczerpnięte z dziedziny sztucznej inteligencji jak np. sztuczne sieci neuronowe, wspomagane filtrem Kalman'a pozbywającego się białego szumu. Takie podejście zastosowane zostało w pracy z roku 2007 autorstwa M. Choi'a Sakthivel'a i W.K. Chung'a[xiv].

1.1.7 Model topologiczny

Do listy konspektów matematycznych wykorzystanych na przestrzeni ostatnich lat należy dodać Topologię, będącą nauką o przestrzeniach abstrahującą od używania pojęcia odległości. Podejście takie zaobserwować można znaleźć w pracy A. Angeli, Stephane Doncieux, Jean-Arcady Meyer'a i David'a Filliat'a[xv] czy też H. Choset'a [xvi]. Polega ono na znajdowaniu miejsc szczególnych i oznaczaniu ich jako węzły. Poprzez poszukiwanie pętli możliwe jest znalezienie przeszkód, pełniących rolę topologicznych 1-wymiarowych dziur. Generuję to mapę topologiczną znajdującą się po prawej stronie rysunku, pochodzącego z artykułu [XV]:

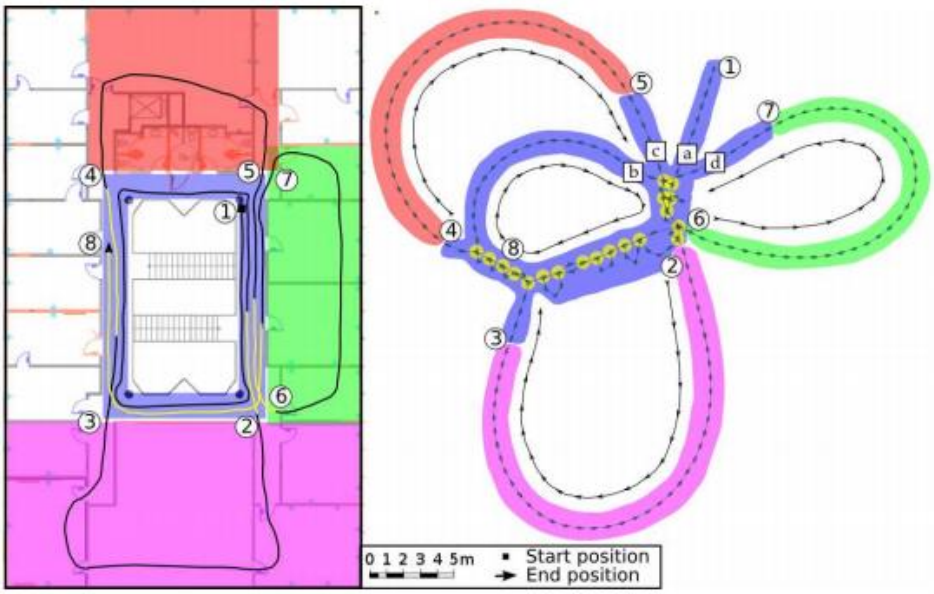


Fig. 5. Floor plan of the travelled environment superimposed with the trajectory of the camera (left part of the figure) and corresponding topological map (right part of the figure). The graph layout is performed using a simple "spring-mass" model [19]. See text for details.

Rysunek 3 Źródło[XV] Mapowany teren oraz odpowiadająca jemu uzyskana mapa topologiczna z zaznaczonymi węzłami i pętlami

1.1.8 Studium implementacji

Wreszcie ostatnim aspektem szeroko opisywanym w literaturze jest implementacja zaprezentowanych powyżej rozważań teoretycznych. W szczególności pomocne powinny się okazać pracę traktujące o realizacji poszczególnych elementów problemu SLAM. Jako przykład podać można pracę autorstwa S. Jeżewskiego, A. Wulkiewicza[xvii], którzy w swojej pracy z 2009r. przedstawiają tzw. koncepcyjną przestrzeń percepcyjną robota mobilnego, która powinna się wyróżniać:

- *Przechowywaniem dużego wycinka przestrzeni w pamięci RAM,*
- *Przechowywaniem danych ze zmienną dokładnością ale nie większą niż 10cm*
- *Przechowywanie danych terenu z różnymi atrybutami takimi jak: teren znany, teren nieznany, teren przejezdny, teren nieprzejezdny, itp.*

Jak pokażemy w dalszej części opracowania warunki te udaję się spełnić dla naszego projektu. W analizie narzędzi realizujących wymagane funkcję, przydatne okazują się być pracę M. Rostowskiej, M. Topolskiego i P. Skrzyprzyńczkiego[xviii], w której przestawiona jest implementacja systemu wielu robotów na robocie o nazwie „SanBot", a także L. Guyot'a, N. Heiniger'a, O. Michel'a i F. Rohrer'a[xix], w której zaproponowany został robot edukacyjny e-puck, posiadający wszystkie wymagane właściwości i będący przedmiotem naszego projektu. Zgłębiając zagadnienie jego opisu można trafić na wiele artykułów traktujących o jego szerokich zastosowaniach, autorstwa: A. Gutierrez[xx](programowanie roju), M. Jacobsson[xxi] (rozpoznawanie obrazów).

1.2 Cele pracy

W sytuacji gdy nie znamy danego terenu, może to być zamknięte pomieszczenie, a chcielibyśmy bez wchodzenia do niego poznać jego rozkład lub dowiedzieć się czy możliwe jest przejście przez niego przydałby się inny sposób, aby się tego dowiedzieć. Jednym z rozwiązań jest wykorzystanie urządzeń zwiadowczych takich jak roboty mobilne. Bez ryzyka dla człowieka można takiego robota wpuścić do danego pomieszczenia i może on przemierzyć powierzchnię, aby zrobić jej mapę. Aby proces ten był szybszy można wykorzystać większą ilość robotów. Jeśli umieścimy sterowanie w jednostce zewnętrznej, która przesyła rozkazy do poszczególnych robotów możemy nimi kierować w tai sposób, aby się nie dublowały powierzchnie, po których już któryś wcześniej przejechał. Po pewnym czasie ludzie będący na zewnątrz danego pomieszczenia mają pełen wgląd w jego rozkład tzn. gdzie znajdują się przeszkody, jakiej szerokości są przejścia, jaka jest faktyczna jego wielkość. Taka wiedza daje potem wiele możliwości np. służbom specjalnym planującym wejście na nieznany teren.

Będąc zainteresowanymi tym tematem postanowiliśmy opracować uproszczony system takiego mapowania nieznanego terenu z wykorzystaniem edukacyjnych robotów mobilnych - Epuck oraz jednostki sterującej, która będzie podejmowała decyzje, gdzie mają jechać.

1.3 Zakres pracy

Projekt wymagał organizacji pracy w wielu płaszczyznach i każdy jego wykonawca był odpowiedzialny za inny aspekt. Poniżej przedstawiona jest wykaz członków zespołu wraz z zadaniami, z które byli odpowiedzialni.

Grzegorz Kiernozek:

- Planowanie i koordynacja projektu.
- Kompleksowa obsługa środowiska symulacyjnego.
- Analiza modułów robota mobilnego oraz implementacja algorytmów ich wykorzystania.

Jan Krzywda:

- Badanie tematu na podstawie literatury naukowej.
- Opracowanie matematycznej reprezentacji systemu.
- Implementacja algorytmów eksploracji terenu, mapowania oraz współpracy robotów.

Michał Rogowski:

- Implementacja algorytmów obsługi mapy.
- Obsługa rzeczywistych robotów.
- Testowanie systemu.

2. Sformułowanie problemów

2.1 Sprecyzowanie głównego celu

Celem przedsięwzięcia jest opracowanie komputerowego systemu kooperacji autonomicznych robotów mobilnych. Zaprojektowane narzędzie ma za zadanie umożliwić grupie robotów mobilnych realizację następujących zadań:

- lokalizacji: zarówno samolokalizacji, jak i lokalizacji elementów środowiska - w tym pozostałych robotów,
- mapowania: tworzenia mapy badanego środowiska.

Lokalizacja:

Kluczowym elementem niezbędnym do działania któregokolwiek innego aspektu naszego projektu jest zdolność epuck-a do określania własnej lokalizacji. Musi on zatem wiedzieć jaką odległość przejechał i jakich zmian kierunku dokonał. Każdy zatem jego ruch musi być zapisywany oraz interpretowany przez nasz system. Zatem głównym zadaniem jest opracowanie sposobu realizacji tych wymagań.

Gdy robot jest w stanie określać własną lokalizację, następnym etapem jest określanie lokalizacji przeszkód terenowych, czyli istotnych punktów, które napotyka na swojej drodze i które będą umieszczane na generowanej przez system mapie. Epuck powinien zatem rozpoznać przeszkodę, zapamiętać ją oraz podjąć decyzję dotyczącą kierunku dalszej jazdy.

Mapowanie:

Dalszym elementem naszych badań jest przeniesienie wyników zadań lokalizacji na mapę, która jest tworzona w trybie rzeczywistym. Każdy ruch epucka jest na niej odnotowywany. Każdy z robotów nie wie nic o swoim położeniu, ale w trakcie jazdy tworzy mapę przejechanego terenu na podstawie własnego ruchu oraz rozpoznanych przeszkód. W celu skrócenia czasu mapowania całego terenu roboty w momencie spotkania się łączą narysowane przez siebie mapy. Dzięki temu nie muszą odwiedzać terenu, który już został odwiedzony przez innego robota.

Scenariusz:

1. Ustawiamy roboty na nieznanym dla nich terenie.
2. Roboty zaczynają jazdę.
3. Mapowanie przejechanego terenu.
4. Gdy wszystkie miejsca odwiedzone zakończ działanie.

Osiągnięcie głównego celu wynikającego z tematu naszej pracy nie byłoby możliwe bez opracowania rozwiązań dla problemów napotkanych po drodze.

2.2 Podstawowa reprezentacja świata i położenia e-pucka

To podstawowe zadanie wymaga stworzenia przejrzystej reprezentacji informacji o rzeczywistym środowisku trójwymiarowym w postaci możliwej do uzyskania z pomocą komputera. Sposób odzwierciedlenia realnego środowiska powinien pozwalać na zaznaczanie przeszkód, a także miejsc wolnych od nich. W calu realizacji zadania sterowania ważnym aspektem reprezentacji położenia e-pucka jest integracja kształtu mapy z informacją o położeniu robota wraz z zaznaczeniem aktualnego zwrotu robota i wykorzystania jej do planowania kolejnych kroków.

2.3 Model mapy i reprezentacji stanu

Kontynuując myśl z poprzedniego punktu model mapy powinien pozwalać na przeniesienie aktualnego stanu robota na użyteczne informacje możliwe do naniesienia w czasie rzeczywistym na mapę. Mapa powinna być edytowalna i pozwalać na prostą prezentację na ekranie komputera, a także na przechowywanie jej w jego pamięci. Dodatkowym atrybutem byłaby możliwość stworzenia własnej mapy, dzięki przejrzystemu interfejsowi użytkownika.

Z kolei stan e-pucka powinien być wyrażony możliwie najmniejsza liczba zmiennych wyrażających jednak pełną informację o aktualnej pozycji, orientacji i aktualnym stanie robota. Stan ten powinien być modyfikowalny i odpowiadać na oddziaływanie z otoczeniem.

2.4 Sterowanie

Problem ten dotyczy sposobu poruszania się e-pucka. Powinien on pozwalać na takie sterowanie, które zdecydowanie zmniejszy czas eksploracji terenu w porównaniu z np. błądzeniem losowym. Problem sterowania powinien być zintegrowany z modelem mapy i wykorzystywać informację zebrane przez robota w czasie symulacji.

Problem sterowania dotyczy także obsługi mapowania i korekcji pozycji, ponieważ e-puck podczas działania programu ma tendencję do tracenia aktualnej pozycji, moduł sterowania powinien umożliwić mu jej poprawę a także skorygować położenie w oparciu o przeszkodę lub innego e-pucka.

2.5 Mapowanie i lokalizacja

Uaktualniona mapa powinna być wyświetlana na ekranie komputera w czasie rzeczywistym, dzięki przesyłaniu informacji jednym z dostępnych w robocie interfejsów. Mapowanie powinno odbywać się w trybie ciągłym ze wzrostem informacji o otoczeniu powstającym wraz z upływającym czasem.

Problemem lokalizacji definiujemy jako odpowiedź na pytanie o początek, koniec lub jakikolwiek inny stan robota eksplorującego teren w odniesieniu do otaczających i zmapowanych elementów rzeczywistego świata. Lokalizacja powinna także pozwolić na wytworzenie jak największego obrazu mapy przy jednoczesnym

2.6 Współpraca robotów

Roboty powinny razem tworzyć wartość dodaną, a więc ich większa liczba powinna przekładać się na dokładniejszą lub trwającą mniej czasu realizację zadania mapowania i lokalizacji. Obsłużone także powinno zostać wydarzenie polegające na spotkaniu dwóch e-pucków pod dowolnym kątem, a także wydobycie z takiego zdarzenia maksymalnej liczby informacji przydatnej w procesie łączenia map.

Spotkanie robotów nie powinno prowadzić do komplikacji wynikających z pozytywnych odczytów z czujników podczerwieni czy też utraty ortogonalności lub przesunięcie o nieznany kąt w jeden z kierunków.

2.7 Obsługa map

Aby móc przeprowadzać symulacje w różnych układach powierzchni potrzebny jest prosty w obsłudze generator. Posłużyć nam do tego może aplikacja, w której za pomocą myszy rysuje się prostokąty na panelu pokrytym siatką. Następnie taki rysunek jest konwertowany na mapę odczytywaną przez program Webots. Takie rozwiązanie zapewnia możliwość szybkiego dostosowywania makiety symulacyjnej do testowania różnych zachowań epuck-a.

Mapa generowana przez roboty mobilne zapisywana jest do pliku .txt. Aby ją jednak zobaczyć w 3D i ewentualnie wykorzystać do innych symulacji potrzebny jest konwerter do pliku .wbt, który również znajduje się w programie z obsługą mapy.

Ostatnią funkcjonalnością jest porównywanie map wygenerowanej przez użytkownika i wygenerowanej przez roboty. Dzięki temu można sprawdzić dokładność odwzorowania terenu przez epuck-i. Najpierw rysujemy mapę myszą na panelu, następnie otwieramy ją w środowisku Webots, puszczamy symulację z zaprogramowanymi robotami i pobieramy wygenerowaną przez nią mapę. Program porównuje mapy obrócone względem siebie pod różnym kątem i o różnych rozmiarach znajdując najlepsze dopasowanie oraz wyświetla obliczony błąd wynikający z różnicy w mapach.

2.8 Obsługa map

Aby móc przeprowadzać symulacje w różnych układach powierzchni potrzebny był prosty w obsłudze generator. Posłużyła nam do tego aplikacja, w której za pomocą myszy rysuje się prostokąty na panelu pokrytym siatką. Następnie taki rysunek jest konwertowany na mapę odczytywaną przez program Webots. Takie rozwiązanie zapewnia możliwość szybkiego dostosowywania makiety symulacyjnej do testowania różnych zachowań epuck-a.

Mapa generowana przez roboty mobilne zapisywana jest do pliku .txt. Aby ją jednak zobaczyć w 3D i ewentualnie wykorzystać do innych symulacji potrzebny jest konwerter do pliku .wbt, który również znajduje się w programie z obsługą mapy.

Ostatnią funkcjonalnością jest porównywanie map wygenerowanej przez użytkownika i wygenerowanej przez roboty. Dzięki temu można sprawdzić dokładność odwzorowania terenu przez epuck-i. Najpierw rysujemy mapę myszą na panelu, następnie otwieramy ją w środowisku Webots, puszczamy symulację z zaprogramowanymi robotami i pobieramy wygenerowaną przez nią mapę. Program porównuje mapy obrócone względem siebie pod różnym kątem i o różnych rozmiarach znajdując najlepsze dopasowanie oraz wyświetla obliczony błąd wynikający z różnicy w mapach.

3. Rozwiązania

3.1 Algorytmy SLAM

3.1.1 Zadanie sterowania

Pierwszym krokiem do zrealizowania podstawowych zadań projektu, którymi były:

- Lokalizacja robota w nieznanym środowisku,
- Mapowanie środowiska z użyciem czujników na podczerwień i kamery,
- Sterowanie trajektorią, umożliwiające przebycie jak największej części mapy przy jednoczesnej minimalizacji czasu.
- Detekcja innego robota na podstawie danych z kamery.

Jest sformułowanie zadania sterowania. Zostało ono podzielone na kilka części, którymi są:

a) Zbieranie informacji pochodzących z czujników,
b) Wysyłanie informacji do urządzenia sterującego,
c) Uzupełnianie mapy w czasie rzeczywistym,
d) Podejmowanie decyzji o optymalnym kierunku ruchu w kolejnej iteracji na podstawie przesłanych odczytów i posiadanej mapy,
e) Uniknięcie kolizji robotów,
f) Zakończenie pracy algorytmu w momencie najlepszego odwzorowania mapy.

Za realizacje powyższych zadań sterowania będą odpowiedzialne funkcję i algorytmy opisane w poniższym opracowaniu. Zakłada się, że robot porusza się w kierunkach ortogonalnych do ścian pomieszczenia. Przeszkody oraz ściany mają kształt graniastosłupów o bokach odpowiednio równoległych i prostopadłych do siebie. Początkowa pozycja może być jednak dowolna z uwagi na jej korekcję w przypadku pierwszego spotkania przeszkody lub ściany. Robot używa diod LED w celu umożliwienia wykrycia w kamerze. Sterowanie kończy się zatrzymaniem robota i zwróceniem ostatecznej mapy.

3.1.2 Wskaźniki jakości sterowania

Ważnym aspektem pracy podczas formułowana algorytmu sterowania optymalnego był wybór wskaźników jakości, przydatnych przy testowaniu algorytmów. Głównym parametrem decydującym o sukcesie był stosunek pól naniesionych na mapę do ilości możliwych do zapełnia komórek na mapie e-pucka, co można zapisać jako:

$$\frac{n_{nieodwiedzone}}{n_{odwiedzone}} < 1$$

Typowe wartości tego współczynnika zależą od kształtu mapy, jednak w przypadku badań zakończonych sukcesem współczynnik ten osiągał wartość 0,1.

Innym ważnym kryterium jest czas potrzebny do zmapowania mapy. Celem było stworzenie mapy w możliwie najkrótszym czasie, zachowując jej topologię, a więc wzajemne położenie przeszkód, zaznaczenie możliwych przejazdów, rogów i innych miejsc charakterystyczny. Otrzymaną podczas symulacji mapę porównujemy z plikiem wejściowym szukając najlepszego dopasowania (mapy mogą być względem siebie przesunięte i/lub obrócone o 90°) na podstawie minimalizacji kwadratu różnicy odpowiadających sobie wartości pól macierzy reprezentujących poszczególne mapy.

3.1.3 Podstawowa reprezentacja świata i położenia e-pucka

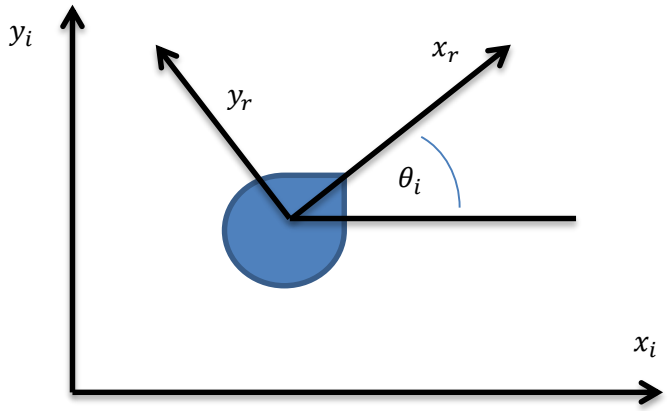
W tym paragrafie opiszemy zmienne opisujące położenie robota, a także ich związek z prędkościami możliwymi do odczytania z interfejsu robota.

3.1.3.1 Współrzędne ogólne

$$q_i = \begin{bmatrix} x_i \\ y_i \\ \theta_i \end{bmatrix}$$

3.1.3.1.1 Współrzędne związane z robotem

$$q_r = \begin{bmatrix} x_r \\ y_r \\ \theta_i \end{bmatrix}$$



Rysunek 4 Współrzędne związane z robotem w odniesieniu do nieruchomego układu współrzędnych

3.1.3.1.2 Macierz przejścia

Macierzą przejścia jest zwykła macierz obrotu wokół osi z o kąt θ.

$$R(\theta_i) = \begin{bmatrix} \cos\theta_i & sin\theta_i & 0 \\ -\sin\theta_i & cos\theta_i & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Współrzędne robota otrzymujemy poprzez obrót współrzędnych ogólnych, w następujący sposób

$$q_r = R(\theta_i) q_i$$

3.1.3.2 Prędkości

3.1.3.2.1 Prędkość liniowa

$$v = \omega_k r = r\dot{\varphi}$$

Gdzie, $r - promień\ koła \quad \dot{\varphi} = \omega_k - prędkość\ kątowa$ obliczona ze wzoru:

$$\dot{\varphi} = \frac{1}{2}(\dot{\varphi}_l + \dot{\varphi}_p)$$

Z fizyki wiemy, że prędkość liniową wyznaczamy zgodnie ze wzorem:

$$v = \omega_k r = \frac{r}{2}(\dot{\varphi}_l + \dot{\varphi}_p)$$

3.1.3.2.2 Prędkość obrotowa

$$\omega = \frac{u}{l}$$

Gdzie, u – prędkość liniowa na obwodzie e - Pucka, l – to promień e - Pucka.

$$\omega = \frac{r}{2l}(\dot{\varphi}_l - \dot{\varphi}_p)$$

3.1.3.2.3 Związek prędkości ze współrzędnymi, równanie stanu:

$$q_i = R^{-1}(\theta_i)\, q_r$$

Współrzędne e-pucka dobrane są w taki sposób, by ruch następował tylko wzdłuż osi x robota. Stąd można wprowadzić oznaczenia.

$$\dot{x}_r = v$$

$$\dot{y}_r = 0$$

$$\dot{\theta}_r = \omega$$

Co da się zapisać w postaci równania stanu:

$$\begin{cases} \dot{q}_i = R^{-1}(\theta_i)\, \dot{q}_r = R^{-1}(\theta_i) \begin{bmatrix} v \\ 0 \\ \omega \end{bmatrix} \\ q_i = q_{i0} + \int_0^T \dot{q}_i\, dt \end{cases}$$

3.1.4 Model mapy i reprezentacji stanu:

3.1.4.1 *Podstawowe założenia*

W naszym projekcie korzystać będziemy z przybliżenia mapy środowiska w postaci macierzy o wymiarach m x n. Stąd mamy:

$$m, n \rightarrow wymiar\ mapy$$

$$m * n - \ ilość\ komórek$$

Mapę przedstawić można jako macierz, w której każdy element ma postać:

$$x_{ij} \in \{0,1,2\}, gdzie\ i \in [1,n]\ , j \in [1,m]$$

Przy czym wartości komórek reprezentują:

0 – Brak przeszkody

1 – Przeszkoda

2 – Pole nieodwiedzone

3 – Narożnik mapy/krawędź mapy

4 – Miejsce spotkania e- pucków

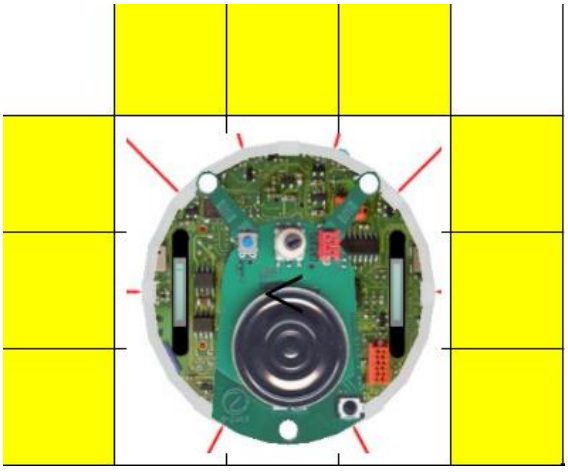
5 – Pola nieaktywne znajdujące się poza dostępnym obszarem

Tabela 1 Przykładowa mapa 10x10

5	3	0	0	0	0	0	0	2	2	2
5	3	0	0	0	0	0	0	0	2	2
5	3	0	0	2	0	0	0	0	0	0
5	3	0	0	0	0	0	0	0	0	0
5	3	0	0	1	1	1	1	0	0	0
5	3	0	0	1	1	1	0	0	0	0
5	3	0	0	0	1	0	0	0	0	0
5	3	0	0	0	0	0	0	0	0	0
5	3	0	0	0	0	0	0	0	0	0
5	3	0	0	0	0	0	0	0	0	0

3.1.4.2 *Reprezentacja E-pucka*

Warto nadmienić, że z uwagi na wielkość e-pucka w stosunku do zasięgu jego czujników, robot zajmował będzie 3x3 komórki na mapie, będąc w stanie rozpoznawać otoczenie w zasięgu 3+3+3 komórek. Doskonale obrazuję to poniższy schemat:



Rysunek 5 Reprezentacja e-pucka na mapie

3.1.4.3 *Przybliżenie współrzędnych ogólnych*

Dowolny wektor położeniowy q_i, będący wektorem wodzącym e-pucka reprezentowany jest przez odpowiadający mu wyraz macierzy o indeksach *m, n*:

$$q_i \sim x_{mn}$$

Transformacja wektora wodzącego do indeksów macierzowych przedstawia się następująco:

$$m = n - \left\lfloor \frac{y_i}{n} \right\rfloor + 1$$

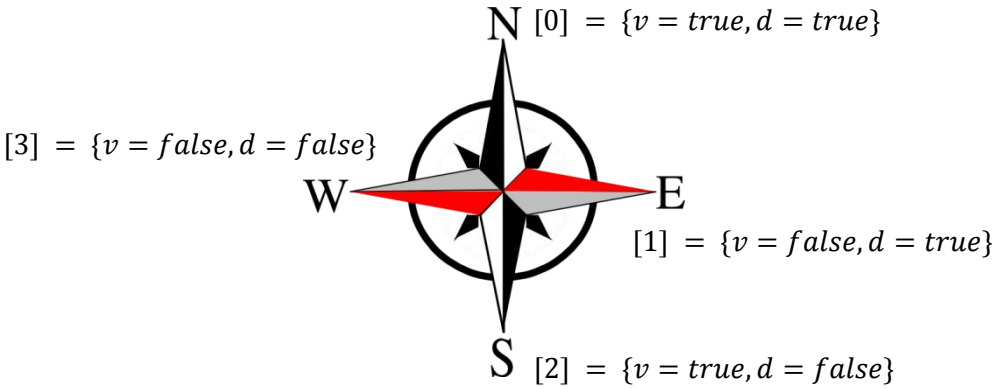
$$n = \left\lfloor \frac{x_i}{m} \right\rfloor + 1$$

3.1.4.4 *Przybliżenie kątów*

W naszym projekcie dopuszczalne jest jedynie poruszanie się prostopadle do przeszkód, stąd też

$$\theta_i \in \left\{0, \frac{\pi}{2}, \pi, \frac{3\pi}{2}\right\},$$

W związku z tym do reprezentacji kierunku robota wystarczą dwie zmienne boolowskie: ***vertical v, direction d.*** W związku z tym tworzą one wektor $\{v, d\}$ odpowiadający kierunkom geograficznym, które wraz z oznaczeniem liczbowym wykorzystywanym w projekcie do indeksowania macierzy kierunków, przedstawiają się następująco:



Rysunek 6 Reprezentacja kierunków za pomocą dwóch zmiennych binarnych v i d

3.1.5 Sterowanie

Opisane metody i algorytmy dotyczą rozwiązania zadania sterowania d) zdefiniowanego w pkt. 1, a więc opisują logikę stojącą za przemieszczaniem e – pucka po częściowo odkrytej mapie. Niektóre z nich nie zostały wykorzystane lub ulepszone z uwagi na niesatysfakcjonujące wyniki lub brak realizacji postawionego zadania.

3.1.5.1 *Model do optymalizacji trasy (niewykorzystany)*

3.1.5.1.1 Opis

Model ten polegał na minimalizacji czasu przejazdu pomiędzy dwoma wyznaczonymi punktami, na podstawie danych czasów obrotu i jazdy o jedno pole do przodu. Niestety znalezienie optymalnego rozwiązania jest niemożliwe analitycznie, a także kosztowne obliczeniowo jeśli chodzi o rozwiązywanie numeryczne. Stąd też pomimo zaimplementowania problemu i znalezieniu rozwiązania dla kilku punktów został porzucony. Sformułowanie algorytmu dostępne jest w załączniku

3.1.5.1.2 Wnioski:

Po odrzuceniu powyższego algorytmu postanowiliśmy zmienić strategię wybierając rozwiązania heurystyczne kosztem analitycznych, co doskonale widać w kolejnych opisanych algorytmach.

3.1.5.2 *Algorytm nagród i kar (zaniechany)*

3.1.5.2.1 Opis

Polega on na wyznaczeniu kolejnego kroku na podstawie odległości *dx* i dy pomiędzy aktualnym położeniem e-pucka, a z góry przyjętym celem. Algorytm brał pod uwagę aktualny zwrot e –pucka nagradzając jazdę do przodu i nakładając dodatkowe ograniczenia na zmianę kierunku jazdy dzięki przemnożeniu przez odpowiednio dostosowane stałe. Dokładny jego opis dostępny jest w załączniku

3.1.5.2.2 Wnioski:

Algorytm ten sprawdzał się w sytuacjach pozbawionych przeszkód, osiągając czasy przejazdu nieznacznie gorsze od metod optymalizacji zastosowanych do modelu 3.1. Problematyczna okazała się obecność przeszkód, która uwidoczniała wadę rozwiązania brak wyróżnionego kierunku omijania przeszkód dla celu znajdującego się na wprost.

3.1.5.2.3 Próba rozwiązania problemu:

Wprowadzony został współczynnik złości, który powodował deterministyczny ruch robota w losowym kierunku, w przypadku utknięcia. Z uwagi na komplikacje w procesie testowania algorytm został porzucony.

3.1.5.3 *Heurystyczna metoda potencjałów*

3.1.5.3.1 Opis

Polegająca na analogii do fizycznego oddziaływania kulombowskiego, dwóch naładowanych ciał. W odróżnieniu od obecnego w literaturze wprowadzania potencjału, z uwagi na wymaganie blisko-zasięgowego oddziaływania wprowadziliśmy siłę kulombowską:

$$\vec{F} = \frac{q_1 q_2}{r^2}\vec{e}$$

Gdzie q – ładunki oddziałujące ze sobą
r – odległość między nimi
e – wersor kierunkowy.

W naszym opisie robot posiada ładunek dodatni, przez co jest odpychany od dodatnio naładowanych przeszkód i przyciągany przez ujemnie naładowane nieodwiedzone pola. To podejście pozwala nam na wypełnienie zadania sterowania polegającego na wyborze kierunku ruchu, pozwalającego na eksplorację nieodkrytego środowiska. Decyzja polega na wyborze maksymalnego wzmocnienia spośród jednego z czterech dostępnych kierunków, scharakteryzowanych na rysunku 3.

3.1.5.3.2 Podejmowana decyzja

Metoda na postawie danych pobranych z mapy jest w stanie odtworzyć nam lokalne pole sił, reprezentowane czterema wartościami, będącymi rzutami na cztery kierunki możliwych kierunków ruchu e-pucka.

- Podejmowana jest decyzja o kierunku kolejnego kroku, przekładająca się na jazdę do przodu lub wykonanie obrotu:

$$kierunek = \{0, 1, 2, 3\}$$

Gdzie zgodnie z oznaczeniem przyjętym na rys. 2:
0 – północ,
1 – wschód,
2 – południe,
3 – zachód.

3.1.5.3.3 Dane i zmienne pomocnicze:

- Mapa:

$$x_{ij} = \{0,1,2,3,4,5\}$$

Gdzie oznaczenia są zgodne z wprowadzonymi w

- Ciąg odwiedzanych pól:

$$r = \{r_0, .., r_i, \dots, r_n\}$$
$$k = \{k_0, .., k_i, \dots, k_n\}$$

- Odległości liczona w kolumnach i wierszach do każdego z punktów;

$$dy = i - i_e$$

$$dx = j - j_e$$

Gdzie j_e, i_e jest odpowiednio kolumną i wierszem przeszkody od której siłę liczymy, i, j, kolumną i wierszem aktualnego położenia robota.

Odległość od przeszkody obliczana jest zgodnie z twierdzeniem pitagorasa:

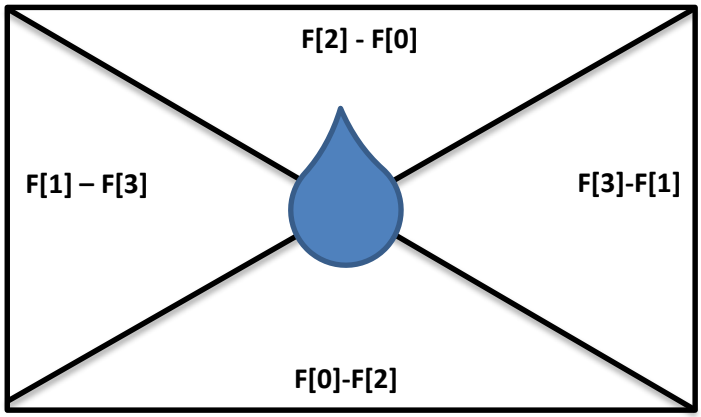
$$r^2 = dx^2 + dy^2$$

Ostatecznie więc pole sił w miejscu aktualnego położenia e-pucka o ładunku $q = 1$ liczone jest ze wzoru:

$$\vec{F} = \sum_{i,j}^{n} \frac{x_{ij}}{(i - i_e)^2 + (j - j_e)^2} [j - j_e, i - i_e]$$

3.1.5.3.4 Uproszczenie liczenia pola sił

Ponieważ pole jest wektorowe, to siły oddziaływania rzutować należy na cztery kierunki wyróżnione indeksami {0, 1, 2, 3}, odpowiadające kierunkom geograficznym. W tym celu sumowanie przeprowadzić należy dla każdego z czterech pól następnie dodając lub odejmując od odpowiadających elementów wektora ***force*** według schematu:



Rysunek 7 Schemat sumowania składowych sił

Gdzie F[0] oznacza wartości sił od przedmiotów znajdujących się na północ, a F[2] od tych znajdujących się na południe.

Da się do zapisać w postaci wzoru upraszczającego wzór wektorowy i rozkładającego go na cztery składowe, gdzie pierwsza z nich ma postać:

$$F_0 = \sum_{i,j}^{n} \frac{x_{ij}}{dx^2 + dy^2} \big(\theta(dy - dx)\theta(dy + dx) - \theta(dx - dy)\theta(-dy - dx)\big)$$

x_{ij} – punkt na mapie razem z wartością ładunku,

$dx\ i\ dy$ – odległość liczona odpowiednio kolumnami i wierszami,

$\theta(dy - dx)$ – funkcja Heavyside'a postaci:

$$\theta(x) = \begin{cases} 1\ dla\ x \geq 0 \\ 0\ dla\ x < 0 \end{cases}$$

Analogiczne wzory wyprowadzić można dla pozostałych kierunków.

3.1.5.3.5 :Wybór wartości ładunków

Dobrane zostały ładunki poszczególnych elementów mapy:

- Odkryta pusta komórka (0) = 0
- Przeszkoda(1) = 30
- Nieodkryta komórka (2) = -15
- Miejsce spotkania e- pucków (3) = 9999
- Róg/krawędź mapy (4) = 100
- Miejsca poza mapą (5) = 0

Dodatkowo nałożone zostało ograniczenie mówiące, że oddziaływanie jest brane pod uwagę jeżeli:

$$|dx| + |dy| < 100$$

3.1.5.3.6 Pseudokod realizujący ostatnie dwa punkty:

```
j0, i0 - indeksy komórki zajętej przez e - pucka.
wysumowanaSiła - wektor sił w kierunkach {północ, wschód, południe, zachód}
Dla każdej przeszkody w punkcie x o indeksach i,j{
        dx = i - i0
        dy = j - j0
        jeżeli (|dx|<2 i |dy|<2 lub |dx|+|dy|>100)
                kontynuuj
        w przeciwnym wypadku{
                siła = ładunki[mapa[i][j]]/(dx*dx+dy*dy)

                jeżeli (dy>dx){
                        jeżeli(dy>dx){
                                wysumowanaSiła[północ] += siła
                                wysumowanaSiła[południe] -= siła}
                        w przeciwnym wypadku{
                                wysumowanaSiła[zachód] += siła
                                wysumowanaSiła[wschód] -= siła}
                }
                w przeciwnym wypadku{
                        jeżeli(dy>dx){
                                wysumowanaSiła[wschód] += siła
                                wysumowanaSiła[zachód] -= siła}
                        w przeciwnym wypadku{
                        wysumowanaSiła[południe] += siła
                        wysumowanaSiła[północ] -= siła}
}
Zwróć największy indeks z wysumowanaSiła
```

$$kierunek\_jazdy = Max_{ind}(\boldsymbol{wysumowanaSiła} * \boldsymbol{direction})$$

Podobnie jak w poprzedniej metodzie również tutaj wynik przemnażany jest przez wektor ***direction*** dobrany w zależności od aktualnego ustawienia e-pucka, przy czym dla jazdy prosto 0.65, obrotu w prawo i w lewo 0.15 i kierunku przeciwnego do orientacji.

Przykładowa postać dla wartości wektora orientacji:[0] {v=true, d=true}:

$$direction = [0.65 \quad 0.15 \quad 0.0 \quad 0.15]$$

3.1.5.3.7 Wnioski

Algorytm ten dobrze dawał sobie radę z przestrzeniami o dużej liczbie wąskich korytarzy i przejść, miał jednak tendencję do utykania w rogach.

3.1.5.3.8 Próba rozwiązania problemów

Wprowadzone zostały dwie nowe kategorie danych na mapie są to odpowiednio

- 3 – miejsce spotkania e – pucków obdarzone ogromnym ładunkiem dodatnim 9999, o którym więcej w punkcie 6.1.

- 4 – rogi, obdarzone ładunkiem 1000, wynikającym z potrzeby odstraszania e-pucka od chęci eksploracji terenu przyległego do narożnika mapy. Sposób znajdowania rogu zostanie opisany w punkcie 6.2.

3.1.5.4 *Zmodyfikowana metoda potencjałów ze sprzężeniem sensorycznym*

3.1.5.4.1 Opis

Jest to uzupełnienie metody 5.3 informacjami z czujników, które mają nadrzędną kolejność w określaniu kierunku kolejnego kroku.

3.1.5.4.2 Działanie

Ponieważ, jak zostanie opisane w punkcie 4.3 czujniki podzielić można na trzy strefy, znajdujące się odpowiednio przed oraz po obu jego stronach, zasygnalizowanie choćby jednej przeszkody w którymś z tych kierunków wiąże się z przyjęciem wartości –inf dla sygnalizowanego kierunku, co schematycznie można zapisać:

$$if\ (czujnik\ w\ kierunku\ i)\{$$
$$force[i] = -inf;$$
$$\}$$

Co po uwzględnieniu wszystkich kombinacji daje pseudokod, którego fragment prezentujemy poniżej:

```
pionowy[iteracja] - czy aktualnie robot obrócony jest pionowo?
kierunek[iteracja] - czy aktualnie robot ma zwrot w dodatnim kierunku osi?

jezeli (pionowy[iteracja]){
        jeżeli (lewy sensor){
                jeżeli (kierunek[iteracja])
                        siła[3] = -inf;
                w przeciwnym wypadku
                        siła[1] = -inf;
        jeżeli (środkowy sensor){
                jeżeli (kierunek[iteracja])
                        siła[0] = -inf;
                w przeciwnym wypadku
                        siła[2] = -inf;
        jeżeli (prawy sensor){
                jeżeli (kierunek[iteracja])
                        siła[1] = -inf;
                w przeciwnym wypadku
                        siła[3] = -inf;
w przeciwnym wypadku{
        jeżeli (lewy sensor){
                jeżeli (kierunek[iteracja])
                        siła[0] = -inf;
                w przeciwnym wypadku
                        siła[2] = -inf;
        jeżeli (środkowy sensor){
                jeżeli (kierunek[iteracja])
                        siła[1] = -inf;
                w przeciwnym wypadku
                        siła[3] = -inf;
        jeżeli (prawy sensor){
                jeżeli (kierunek[iteracja])
                        siła[3] = -inf;
                w przeciwnym wypadku
                        siła[1] = -inf;
```

3.1.5.4.3 Wnioski

Algorytm ten sprawdza się w środowisku testowym, co skutkuję odwiedzeniem średnio 80% mapy przez samodzielnego e-pucka.

3.1.6 Mapowanie i lokalizacja

Budowanie mapy jest jednym z ważniejszych elementów naszego systemu, stąd też bardzo wiele uwagi poświęciliśmy na badania zastosowanych rozwiązań.

3.1.6.1 Metoda uzupełniania mapy

3.1.6.1.1 Opis

Mapa uzupełniana jest na podstawie odczytów z sensorów znajdujących się z przodu i z boku e-pucka, zgodnie z rys.3. Każdy z siedmiu czujników (pomijamy wskazania czujników, położonych po bokach, z tyłu e-pucka) ustawioną ma wartość graniczną, będącą odpowiednią

kombinacją odczytów z poszczególnych czujników podczerwieni. Dzięki temu z wykorzystaniem zmiennych mówiących o stanie e-pucka ***vertical, direction, x, y*** jesteśmy w stanie nanieść odczyty w odpowiednie miejsce macierzy, będącej mapą.

Rysunek 8 Czujniki zczytujące mapę

3.1.6.1.2 Możliwe wartości wejściowe:

- 0 – Odwiedzona pusta komórka,
- 1 – Przeszkoda,
- 2 - Nieodwiedzona komórka,
- 3 – Pole odstraszające, wynikające ze spotkania e-pucków,
- 4 – narożnik,
- 5 – element znajdujący się poza mapą.

3.1.6.1.3 Fragment kodu:

Fragment odpowiedzialny za rysowanie mapy:

```
if (vertical) {

        if (direction) {

                if (eMap[epuckY0][epuckX0 - 2] != 1)
                        eMap[epuckY0][epuckX0 - 2] = leftCenter; // West
                if (eMap[epuckY0 + 1][epuckX0 - 2] != 1)
                        eMap[epuckY0 + 1][epuckX0 - 2] = leftFront;

                if (eMap[epuckY0][epuckX0 + 2] != 1)
                        eMap[epuckY0][epuckX0 + 2] = rightCenter; // East
                if (eMap[epuckY0 + 1][epuckX0 + 2] != 1)
                        eMap[epuckY0 + 1][epuckX0 + 2] = rightFront;

                if (eMap[epuckY0 + 2][epuckX0 - 1] != 1)
                        eMap[epuckY0 + 2][epuckX0 - 1] = frontLeft; // North side
                if (eMap[epuckY0 + 2][epuckX0] != 1)
                        eMap[epuckY0 + 2][epuckX0] = frontCenter;
                if (eMap[epuckY0 + 2][epuckX0 + 1] != 1)
                        eMap[epuckY0 + 2][epuckX0 + 1] = frontRight;
```

3.1.6.2 *Wykrywanie obecności innego E-pucka*

3.1.6.2.1 Opis

Wykrywanie drugiego e-pucka realizowane jest z pomocą kamery, będącej na wyposażeniu robota. Rozpoznawanie oparte jest na prostym algorytmie, w którym zliczana

jest ilość pikseli o natężeniu koloru czerwonego powyżej wartości granicznej. Również arbitralnie podawana jest liczba wymaganych czerwonych pikseli potrzebnych do pozytywnej wartości na wyjściu.

3.1.6.2.2 Fragment kodu:

```
for (int height = 0; height < camera.getHeight(); height++) {
        for (int i = 0; i < camera.getWidth(); i++) {
                redFilter += " " + Camera.imageGetRed(camera.getImage(), camera.getWidth(), i, height);
                greenFilter += " " + Camera.imageGetGreen(camera.getImage(), camera.getWidth(), i, height);
                blueFilter += " " + Camera.imageGetBlue(camera.getImage(), camera.getWidth(), i, height);

                if (Camera.imageGetRed(camera.getImage(), camera.getWidth(), i, height) > minRedStrength) {
                        if (Camera.imageGetGreen(camera.getImage(), camera.getWidth(), i, height) < maxRestStrength) {
                                if (Camera.imageGetBlue(camera.getImage(), camera.getWidth(), i, height) < maxRestStrength)
                                        redSignal = true;
                                        redPixelsCount++;
                                }
                        }
                }
        }
```

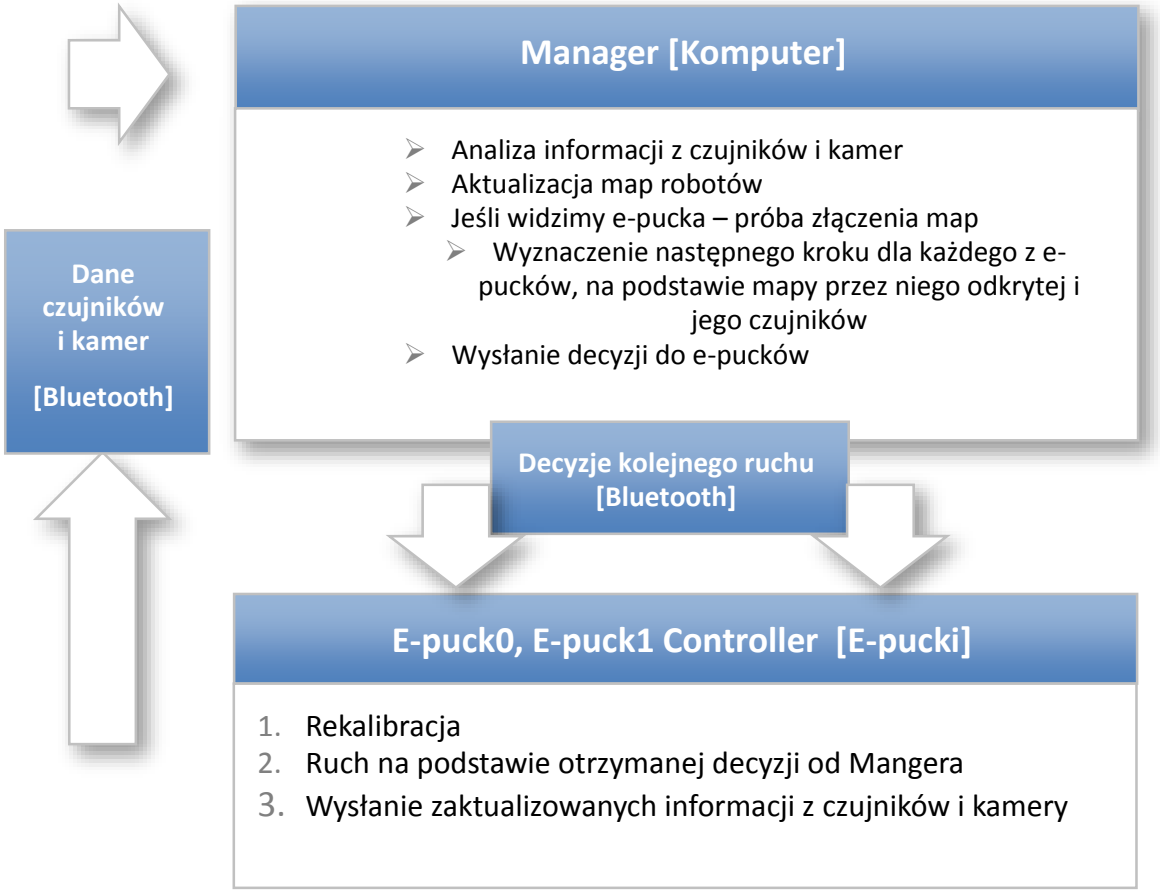
3.1.7 Współpraca robotów

3.1.7.1 *Współpraca z managerem od strony e- pucka* (kontroler):

3.1.7.1.1 Opis:

Współpracę z managerem realizuję algorytm zapisany w kontrolerze e-pucka. Każdy z robotów posiada swój kontroler wyposażony w reciever i emiter nadające na kanałach właściwych poszczególnym robotom. Kontroler posiada trzy stany (Waiting, Resolving, Calibrating), odpowiadające odpowiednio za czekanie na komendę, wykonywanie polecenia i poprawianie prostopadłości względem ściany. Podstawą komunikacji jest syncCode, który służy jako narzędzie do weryfikacji kroku iteracji, tak by komendy managera wykonywane były w odpowiednim momencie. W tej również klasie z racji wykonywania poleceń managera nadawana jest prędkość kół, a także ustalany krok czasowy na 1600ms/32ms w zależności od stanu.

Schemat współpracy kontrolera e-pucka z managerem:

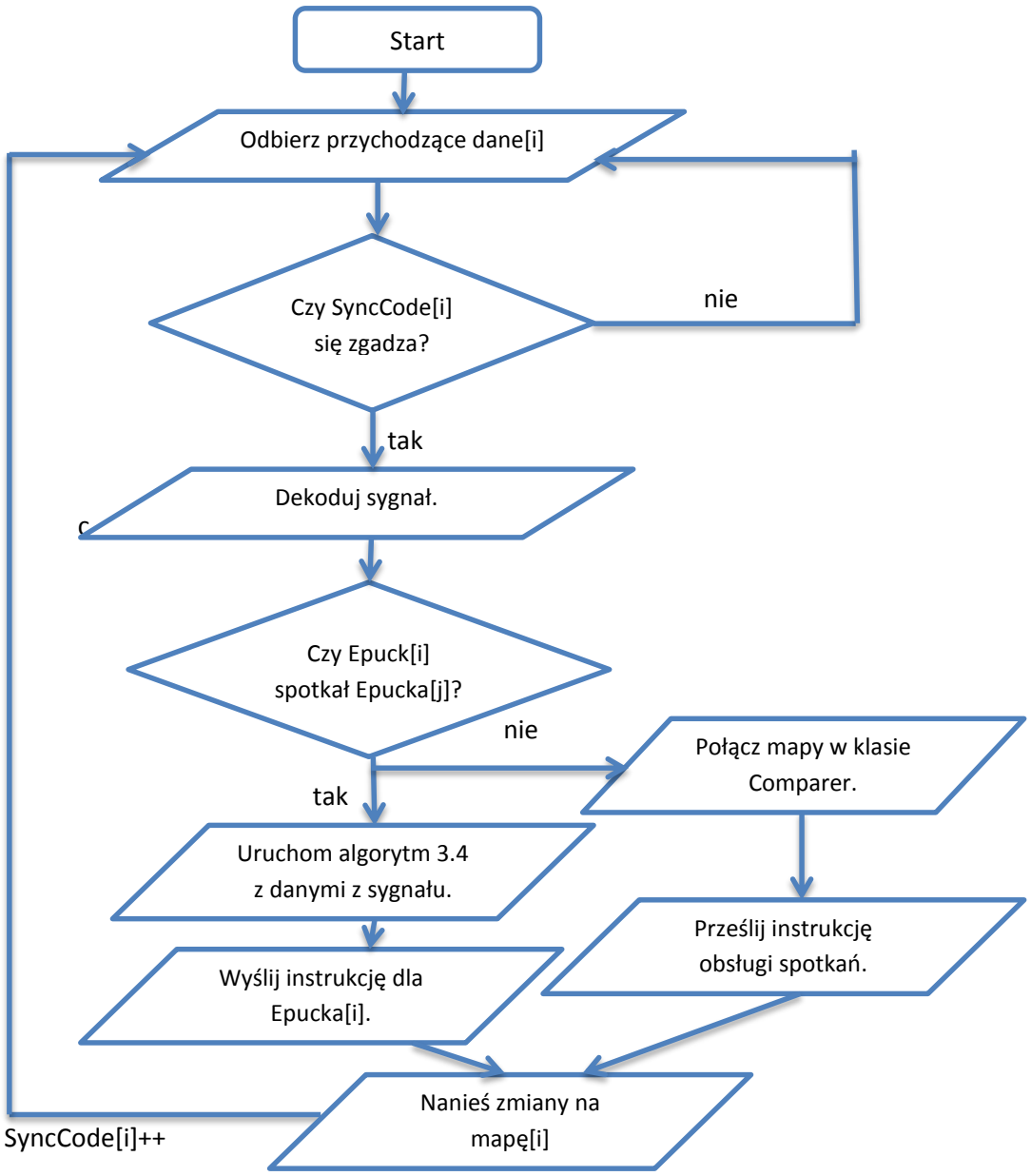

Manager [Komputer]
Analiza informacji z czujników i kamer
Aktualizacja map robotów
Jeśli widzimy e-pucka – próba złączenia map
Wyznaczenie następnego kroku dla każdego z e-pucków, na podstawie mapy przez niego odkrytej i jego czujników
Wysłanie decyzji do e-pucków
Dane czujników i kamer
[Bluetooth]
Decyzje kolejnego ruchu
[Bluetooth]
E-puck0, E-puck1 Controller [E-pucki]
1. Rekalibracja
2. Ruch na podstawie otrzymanej decyzji od Mangera
3. Wysłanie zaktualizowanych informacji z czujników i kamery

3.1.7.2 *Algorytm managera*

3.1.7.2.1 Opis:

Managerem nazywana jest klasa nadrzędna, odpowiedzialna za logikę robotów. To w niej obliczany jest kolejny krok zgodnie z algorytmem 5.4. Komunikacja odbywa się poprzez kody synchroniczne (zwane synccode), po których identyfikowane są kroki robota z decyzją managera.

3.1.7.2.2 Schemat blokowy głównego zadania managera (sterowania robotem):



Rysunek 10 Schemat blokowy klasy manager

3.1.7.2.3 Działanie:

Ważnym elementem pracy managera jest przechowywanie elementów takich jak recievery, syncCody, odebrane wiadomości i emitery, a także kontrolery EpuckControler w macierzach, co pozwalają na realizację pętli po wszystkich Epuckach. Dzięki temu E-pucki mogą działać niezależnie od siebie, co jest szczególnie ważne, gdy jeden z nich ulegnie awarii lub przełączy się w stan poprawy ortogonalności (6.3). Manager, podobnie jak roboty ma stany: ***resolving*** a także ***waiting***, obrazujące aktualną aktywność managera.

Inną ważną funkcją managera jest stworzenie obiektu klasy comparer w przypadku spotkania Epucków, prowadzącej do połączenia map.

3.1.7.2.4 Fragment kodu:

```
private void run() throws InterruptedException {
        int[] sensors = new int[7];
        int[] nextCommand = new int[nEpucks];

        System.out.println("hello I'm manager");
        for (int i = 0; i < nEpucks; i++) {
                syncCode[i] = 0;
                managerState[i] = State.WAITING;
                meetPoint.add(new ArrayList<int[]>());
        }

        while (step(TIME_STEP) != 1) {
                for (int i = 0; i < nEpucks; i++) {
```

3.1.7.2.5 Wnioski:

Algorytm ten jest mózgiem naszego projektu, ma dostęp do map, ich połączenia a także wszystkich klas podrzędnych co czyni go klasą odpowiedzialną za działanie systemu.

3.1.7.2.6 Fragment kodu:

```
switch (epuckState) {
case WAITING:
        if (receiver.getQueueLength() > 0) {
                TIME_STEP = 1600;
                messageGot = getInfoFromReceiver();
                System.out.println("GOT INFO: "+messageGot);

                if (syncCode == getMessageSyncCode(messageGot)) {
                        epuckState = State.RESOLVING;
                }
        } else {
                break;

        }
case RESOLVING:
        moveEpuck(getMessageCommand(messageGot));
        wasResolving = true;
        break;

case CALIBRATING:
        callIter++;
        if (sensorsService.orthogonalCheck() > 1.05)
                moveEpuck(11);
        if (sensorsService.orthogonalCheck() < 0.95)
                moveEpuck(-11);
        System.out.println(Math.abs(sensorsService.orthogonalCheck() - 1));
        if (callIter > 20){
                moveEpuck(2);
                epuckState = State.WAITING;
                wasResolving = false;
        }
        if (Math.abs(sensorsService.orthogonalCheck() - 1) < 0.05){
                moveEpuck(2);
```

3.1.7.2.7 Wnioski:

Dzięki nie wysyłaniu informacji w stanie ***calibrating,*** robot jest w stanie wykonywać proste operacje takie jest kalibrowanie i oczekiwanie na sygnał bez ingerencji managera.

3.1.7.3 *Warunek stopu:*

Eksploracja terenu przez e-pucka zakończyć się może z trzech powodów:

1. Wyjazd poza określone wcześniej granicę mapy (pogorszanie się mapy)
2. Pozostawienie określonego procenta nieodkrytych pól pomiędzy granicami mapy
3. Wariancja położenia obu współrzędnych mniejsza od wartości granicznej.

Dzięki temu te uzupełniające się warunki sprawiają, że robot zostanie zatrzymany w przypadku braku koncepcji na dalszą eksplorację (3), dokona niewłaściwej zmiany wcześniej otrzymanej mapy (2) lub odkryję interesujący nas procent pól.

Wyjazd poza granicę mapy realizowany jest po odcięciu części aktywnej, a więc po wprowadzeniu wartości 5 na mapie na zewnątrz prostokąta złożonego z wartości 4. Zakładamy, że funkcja zliczająca ilość pól oznaczonych 5 jest niemalejąca w przeciwnym wypadku e-puck się zatrzymuje. Wyrazić to można matematycznie:

$$n(x_{ij} = 5)_{i+1} - n(x_{ij} = 5)_i \geq 0$$

Gdzie $n(x_{ij} = 5)_i$ to funkcja zliczająca wystąpienia „5” w i-tej iteracji

Zgodnie z drugim warunkiem możliwe jest wprowadzenie zadowalającej nas postaci parametru c, dla warunku stopu:

$$Jeżeli\ \frac{n(x_{ij} = 2)_i}{n(x_{ij} \neq 2)_i} < c\ to\ stop$$

U nas $c = 0.15$

Trzecim rozwiązaniem, z którego najczęściej korzysta robot jest badanie wariancji ostatnich 15 kroków. Zgodnie ze wzorem wariancję wyznaczamy korzystając z estymatora jako:

$$\overline{\Delta x_\iota}^{\,2} = \frac{\sum_{j=1}^{15}(x_j - \bar{x})^2}{n}$$

Podobnie dla drugiej współrzędnej. Warunek stopu jest sprawą arbitralną w naszym wypadku wynoszącą:

$$Jeżeli\ \overline{\Delta x_\iota}^{\,2} + \overline{\Delta y_\iota}^{\,2} < 0.3\ to\ stop$$

*3.1.7.4 **Algorytm obrotu map**(zaniechany):*

3.1.7.4.1 Opis:

Algorytm ten wywoływany w klasie comparer polega na pobraniu listy przeszkód oraz krawędzi, by następnie dokonać ich translacji względem punktu spotkania robotów. Dzięki temu uzyskujemy listę punktów charakterystycznych na obu mapach z potencjalnym punktem obrotu przesuniętym do początku układu współrzędnych.

3.1.7.4.2 Opis matematyczny

Pierwszym krokiem jest transformacja współrzędnych każdego z Epucków o wektor $[dx, dy]$, gdzie dx i dy to współrzędna x i y miejsca spotkania. Realizowane jest to poprzez funkcję:

```
private ArrayList<double[]> translate(ArrayList<double[]> points, double dx, double dy){

        for(int i=0;i<points.size();i++){
                double[] coord = {(points.get(i)[0] + dx), points.get(i)[1] + dy};
                points.set(i, coord);
        }
        return points;

}
```

Za realizację obrotów odpowiedzialne są macierze obrotu, wykorzystane już poprzednio w przejściu ze współrzędnych ogólnych do tych związanych z robotem. Macierz obrotu jest więc postaci:

$$R(\theta) = \begin{bmatrix} \cos\theta & sin\theta & 0 \\ -\sin\theta & cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Dzięki temu obrócone położenia punktów, np. ścian wyrazić można jako:

$$\boldsymbol{w}' = R(\theta)\,\boldsymbol{w}$$

Przy czym w przypadku naszego projektu dopuszczalne wartości kąta $\theta_i \in \left\{0, \frac{\pi}{2}, \pi, \frac{3\pi}{2}\right\}$. Dzięki temu uzasadnione wydaję się użycie przeglądu zupełnego jako sposobu na minimalizowanie funkcji celu.

3.1.7.4.3 Funkcja celu:

$$f(\boldsymbol{w_1}, \boldsymbol{e_1}, \boldsymbol{w_2}', \boldsymbol{e_2}') = \sum_{i=1}^{n}\sum_{j=1}^{n} d(w_{1_i}, w_{2_j}') + \sum_{i=1}^{m}\sum_{j=1}^{m} d(e_{1i}, e_{2_j}') \rightarrow min$$

Gdzie $\boldsymbol{w_1}$ jest wektorem przeszkód mapy 1, a $\boldsymbol{w_2}'$ wektorem przesuniętych i obróconych przeszkód mapy 2, analogiczne oznaczenie dotyczy wektorów $\boldsymbol{e_1}$ oraz $\boldsymbol{e_2}'$.

Odległość d zdefiniowana jest jako kwadrat długości euklidesowej:

$$d(w_i, w_j) = (w_{x_i} - w_{x_j})^2 + (w_{y_i} - w_{y_j})^2$$

3.1.7.4.4 Rozwiązanie:

Rozwiązanie poszukiwane jest dzięki przeglądowi zupełnemu dopuszczalnych wartości kąta θ_i.Ponieważ jednak współrzędna przestrzenna robota w momencie spotkania różni się od prawidłowego punktu obrotu, tak uzyskana mapa nie jest jeszcze mapą pełną.

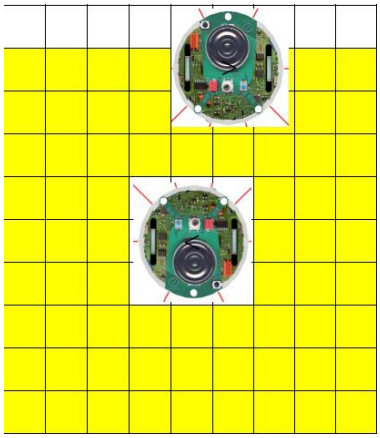
3.1.7.4.5 Powód skończenia prac

Z uwagi na możliwość wykorzystania tylko 4 kątów ortogonalnych metoda umożliwiająca obrót o dowolny kąt jest nadprogramowa. Większym jednak problemem była niska skuteczność być może wnikała z błędnie postawionej funkcji celu.

*3.1.7.5 **Algorytm nakładania się map** wraz z dopasowaniem:*

3.1.7.5.1 Opis:

Algorytmem obsługującym w pełni nakładanie się map jest połączenie obrotów o kąty $\theta_i \in \left\{0, \frac{\pi}{2}, \pi, \frac{3\pi}{2}\right\}$ z przeglądem zupełnym możliwości nałożenia się map na siebie, uzyskanych przez dobranie odpowiedniego obrotu zgodnie z algorytmem 5.3. Realizacją tego zadania podejmuje się algorytm nazwany shaker'em. Jego nazwa pochodzi od dopuszczalnych przesunięć mapy 2 względem mapy 1, gdzie dopuszczalne wektory translacji wykluczają, punkty w oczywisty sposób zajęte przez e-pucka.



Rysunek 11 Na żółto zaznaczone zostały potencjalne miejsca przesunięcia mapy1 względem mapy2

3.1.7.5.2 Opis matematyczny

Przesunięcia wykonywane są z użyciem funkcji użytej w poprzednim podpunkcie i następują po obrocie mapy 1 względem mapy 2 o dany kąt,, każde przesunięcie poddawane jest z kolei ocenie zgodnie z funkcją celu maksymalizującą zgrupowania ścian.

3.1.7.5.3 Funkcja celu:

$$f(w_1, e_1, w_2', e_2') = \sum_{ściany\ w_1, w_2'} W(w_1, w_2') + \sum_{brzegi\ e_1, e_2'} E(e_1, e_2')$$

Gdzie $\boldsymbol{w_1, w_2'}$ to wektory odpowiednio wektory przeszkód mapy1, a także obróconej i przesuniętej mapy 2. Natomiast e_1, e_2' są odpowiednio wektorami wodzącymi krawędzi mapy 1 i obróconej i przesuniętej mapy 2. W obu z nich każdy i-ty element jest postaci współrzędnych kartezjańskich:

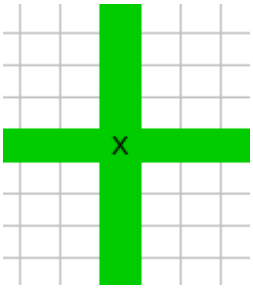
$$w_{1i} = \{w_{1x}, w_{1y}\}$$

Funkcja $W(w_1, w_2')$ to funkcja zliczająca ściany odległe od siebie o mniej niż 1,5 komórek. Warto zauważyć, że z uwagi na obroty operacje na liczbach całkowitych zostały zamienione na liczby dziesiętne.

Przykład pętli zliczającej akceptowane wzmocnienia:

```
for(int i=0;i < walls1.size();i++){
    for(int j=0;j < walls2.size();j++){
        x1 = walls1.get(i);
        x2 = walls2.get(j);
        if(Math.abs(x1[0]-x2[0])<1.5)
            d1++;
        if(Math.abs(x1[1]-x2[1])<1.5)
            d1++;
    }
}
```

Rozwiązanie to dla każdego punktu z mapy 2 tworzy korytarz, który można zobrazować na rysunku:



Ten sposób rozumowania pozwala nakładać na siebie brzegi oraz przeszkody w sposób niezwykle skuteczny.

3.1.7.5.4 Rozwiązanie:

Ostatecznie mapy nakładane są na siebie jako wynik dodawania dwóch macierzy, będących mapami pochodzącymi od dwóch e-pucków. Dzięki temu otrzymać możemy dopuszczalne wartości w macierzy wynikowej:

$$x_{ij} \in \{0,1,2,4,5,8\}$$

Gdzie indeksy oznaczają:

0 – Żaden z robotów nie zauważył przeszkody.
1 – Jeden robot zauważył przeszkodę.
2 – Oba roboty zauważyły przeszkodę.
4 – Jeden robot zauważył tu brzeg mapy.
5 – Jeden robot zauważył przeszkodę, a drugi brzeg mapy.
8 – Oba roboty zaznaczyły w tym miejscu brzeg mapy.

Dzięki takim wynikom łatwiej jest interpretować mapę wynikową w której wartości takie jak 2 czy 8 pokazują zgodność odczytów, a tym samym niemalże pewność istnienia tam wskazanego obiektu.

3.1.8 Obsługa zdarzeń (problemów):

3.1.8.1 *Odcinanie krawędzi mapy:*

3.1.8.1.1 Opis problemu:

Robot przyciągany jest przez nieodwiedzone pola, znajdujące się poza granicami mapy, w związku z tym w pierwszej kolejności zwykle ma za zadanie objechać mapę wyznaczając jej krawędzie.

3.1.8.1.2 Rozwiązanie:

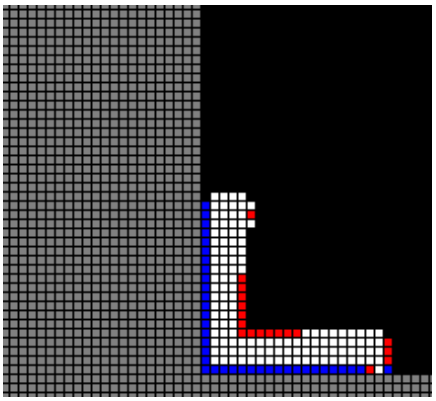
Dzieje się to z pomocą bardzo prostego skryptu zliczającego ilość przeszkód w porównaniu do pól pustych w poszczególnych wierszach i kolumnach. Kod takiego rozwiązania prezentuje się następująco:

```
for (int i=0; i<eMap.length;i++){
    if (obstaclesX[i][0]>14 && obstaclesX[i][1]<4){
        edgeX.add(i);
```

Warunki wewnątrz funkcji warunkowej oznaczają, że graniczne wartości wyglądają następująco:

$$n_{przeszkód} > 14 \ \wedge n_{wolnych\ komórek} < 4$$

Jeżeli ten warunek jest spełniony kolumna lub wiersz zaznaczana jest jako krawędź mapy, przy czym detekcja wnętrza odbywa się dzięki położeniu e-pucka względem nowopowstałej ściany.



Rysunek 12 Detekcja dwóch brzegów (kolor niebieski), z równoczesnym wyłączeniem z obliczania pola sił punktów o nowej wartości 5 znajdujących się poza dostępną przestrzenią (szary).

3.1.8.2 *Spotkanie robotów:*

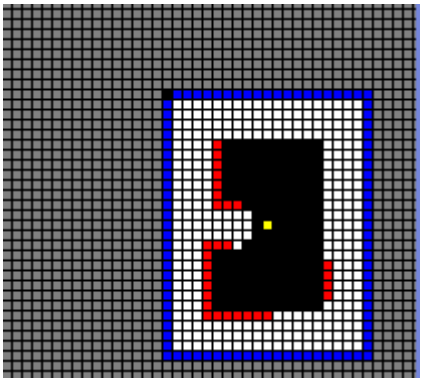
3.1.8.2.1 Opis problemu:

Problem pojawiał się w momencie spotkania dwóch e-pucków, skutkowało to:

- Traktowaniem e-pucka jako przeszkody i zaznaczaniem jej na mapie,
- Zderzeniem, powodującym utratę ortogonalności i przesunięcie względem poprzednio zmapowanego terenu,
- Wspólną eksploracją połączoną z częstymi spotkaniami prowadzącymi do powyższych problemów.
- Zauważenie spotkania tylko przez jednego robota, prowadzące to braku informacji o współrzędnych drugiego (brak miejsca obrotu)

3.1.8.2.2 Rozwiązanie:

- Rozwiązaniem problemu z czujnikami i pozbycie się sztucznej przeszkody było wyczyszczenie odczytów z przednich sensorów, umotywowane wolną przestrzenią konieczną do zauważenia drugiego robota.
- Brak zderzeń udało się rozwiązać dzięki specjalnemu punktowi, o bardzo silnym potencjale umiejscowionego 5 kratek przed e-puckiem spotykającym kompana. Dzięki temu, algorytm podpowiada ucieczkę z tego miejsca, nie ryzykując zderzeniem. Punkt ten oznaczany jest na mapie cyfrą 3, rysowany jest na żółto i posiada ładunek 9999, co opisane zostało w paragrafie 5.3.8.:



Rysunek 13 Mapa tuż po spotkaniu dwóch e-pucków w okolicach jej środka. Na żółto zaznaczony został punkt o ogromnym potencjale, powodujący odpychanie się od siebie obu e-pucków.

- Punkty oznaczone cyfrą 3 znikają po 600 iteracjach, co nie jest dużym wynikiem zważywszy, że manager w stanie ***waiting*** taktowany jest co 32 ms.
- Przy spotkaniu e-pucków manager pobiera oba zestawy współrzędnych, których używa do końca pracy programu.

3.1.8.3 *Wyznaczanie rogów:*

3.1.8.3.1 Opis problemu:

Problem wynikał z nadmiernego przyciągania komórek znajdujących się na domyślnej mapie e-pucka poza terenem możliwym do eksploracji, co pokazuję rysunek:

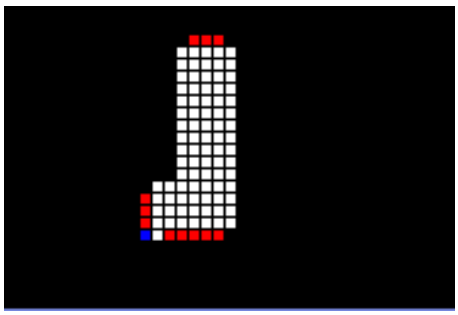
Skutkowało to:

- Powtarzaniem tych samych, nie wnoszących nic do eksploracji ruchów w okolicy rogu.
- Przyciągania rogów.
- Brakiem odwiedzonego środka mapy.

3.1.8.3.2 Rozwiązanie:

- Wprowadzenie potencjału odpychającego w roku, oznaczanego cyfrą 4, kolorem różowym i posiadającego ładunek 1000.
- Detekcja tego punktu, wymagała by dwa sąsiednie sensory pokazały przeszkodę, co skutkować powinno znalezieniem się w rogu.
- Za obsługę odpowiedzialny jest fragment kodu w kontrolerze e-pucka:

```
if (reward[0] + reward[1] == -2*inf) //prawy gorny rog
  epuckMap.eMap[y[i]+2][x[i]+2] = 4;
if (reward[1] + reward[2] == -2*inf) //prawy dolny rog
   epuckMap.eMap[y[i]-2][x[i]+2] = 4;
if (reward[2] + reward[3] == -2*inf) //lewy dolny rog
    epuckMap.eMap[y[i]-2][x[i]-2] = 4;
    if (reward[3] + reward[0] == -2*inf) //prawy gorny rog
            epuckMap.eMap[y[i]][x[i]-2] = 4;
```



Rysunek 14 Kolorem niebieskim oznaczony został lewy dolny rów, wywołany odczytami z czujników

3.1.8.4 *Poprawa ortogonalności:*

3.1.8.4.1 Opis problemu:

Ponieważ wymagamy prostopadłości do przeszkód, a obroty e-pucka, powodują propagowanie się błędu, rozumianego jako odchyłkę od kierunku poziomego lub pionowego, po kilku obrotach robot miał tendencję do tracenia ortogonalnej pozycji względem przeszkód, co skutkowało:

- Przesunięciami mapy wynikającej z pokonywania większych odległości pod katem i dodatkowego przesunięcia w osi innej od kierunku poruszania się.
- Uderzaniem w narożniki przeszkód.
- Obrotem mapy o 90 stopni w przypadku dużego błędu.

3.1.8.4.2 Rozwiązanie:

- Do kontrolera e- pucka dodany został stan ***calibrating*** , w którym robot nie wysyła danych do managera.
- Zamiast tego, dokonuje on obrót do momentu wyrównania się wskazań przednich czujników odległości.
- Stan ten aktywowany jest gdy sensory wykryją obecność przeszkody i stosunek przednich sensorów będzie różny od zadanej wartości.
- Odpowiedzialność w kodzie przerzucona jest na kilka klas, jednak główny rdzeń znajduję się w kontrolerze e-pucka:

```
else if (epuckState!= State.CALIBRATING && Character.getNumericValue(sensors.charAt(8))==1)
{
        emitterSendInfo(++syncCode, sensors,false);
        moveEpuck(2);
        if (sensorsService.ableToCalibrate() && Math.abs(sensorsService.orthogonalCheck() - 1) > 0.05){
                epuckState = State.CALIBRATING;
                callIter = 0;}
}
else
```

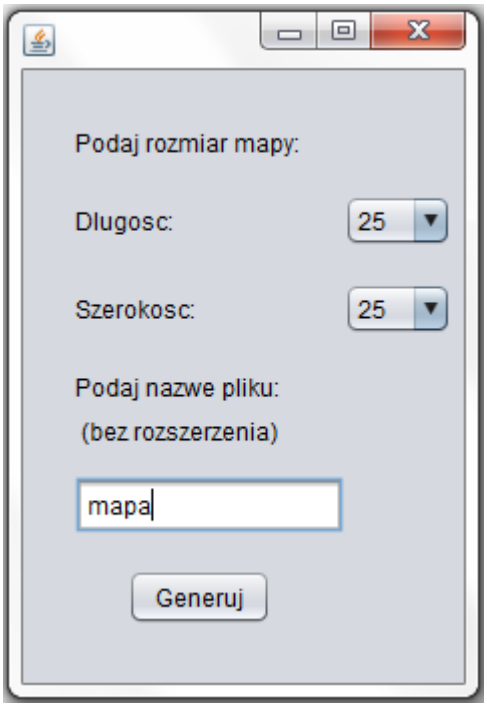
3.2 Obsługa map.

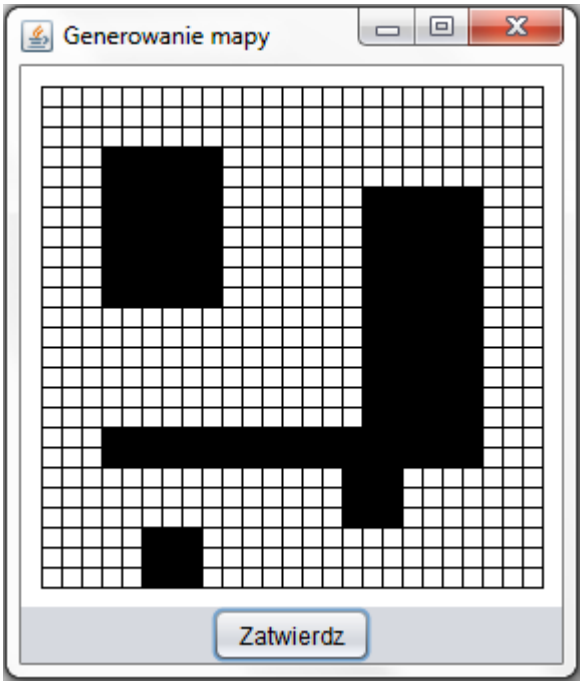
3.2.1 Generator mapy

Do realizacji tego komponentu użyliśmy aplikacji okienkowej napisanej w Javie, która posiada wszystkie potrzebne funkcje. Składa się z prostego menu przenoszącego nas do poszczególnych funkcjonalności.

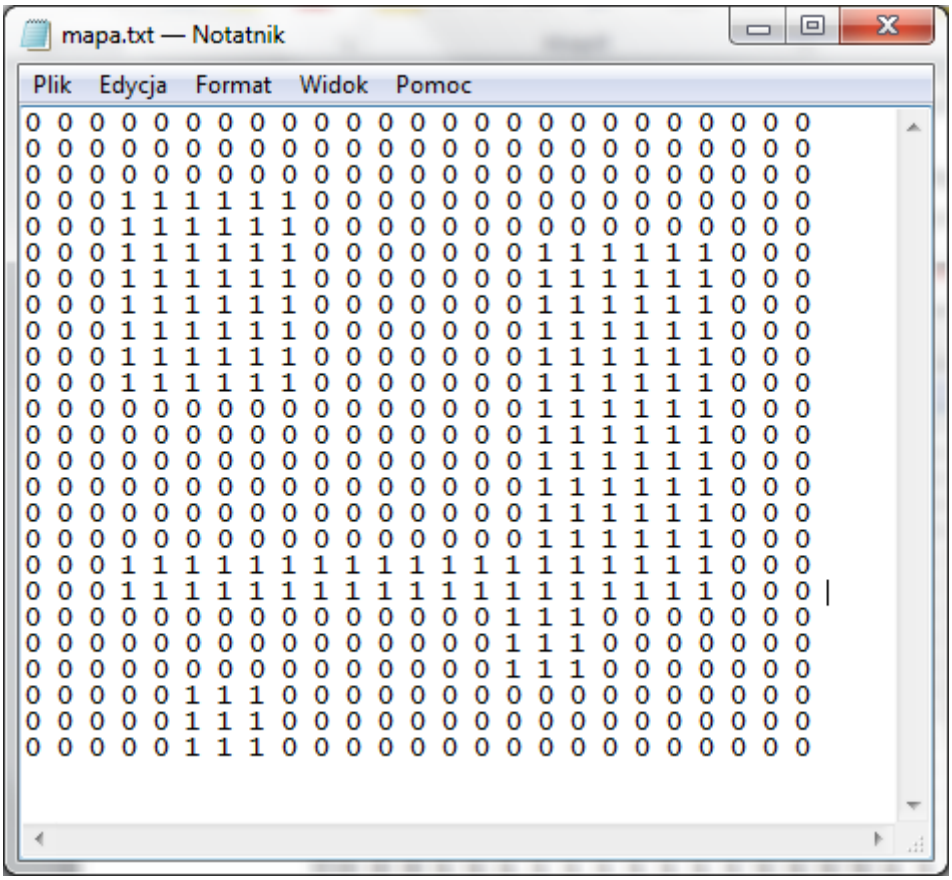
Panel rysowania mapy.

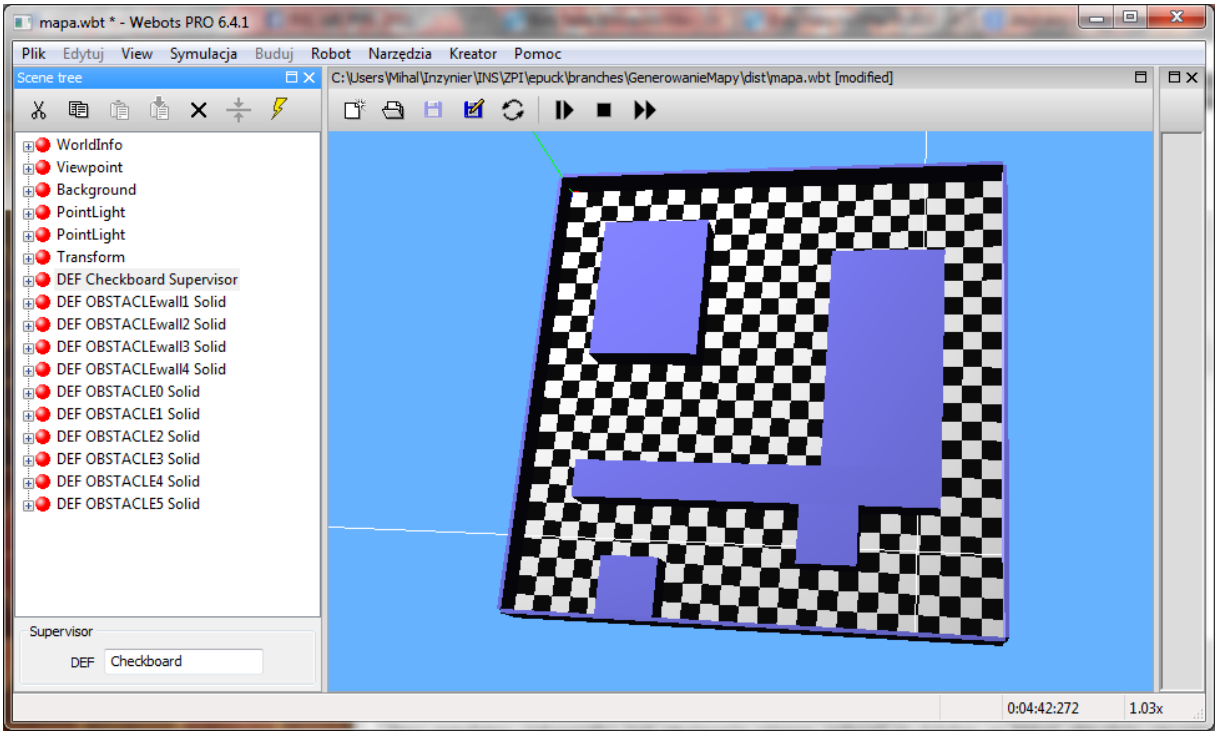
Przed właściwym rysowaniem mamy ekran, na którym możemy ustawić wysokość i szerokość wyrażoną w kratkach, co ułatwia dodawanie przeszkód. Dla porównania sam Epuck zajmuje 3x3 kratki.





Rysowanie polega na wciśnięciu lewego przycisku myszy, gdy kursor znajduje się na jednej kratce, przeciągnięcia go w inne miejsce i zwolnienia przycisku. Następnie po zatwierdzeniu mapa zostaje zapisana do dwóch plików: mapa.txt i mapa.wbt.

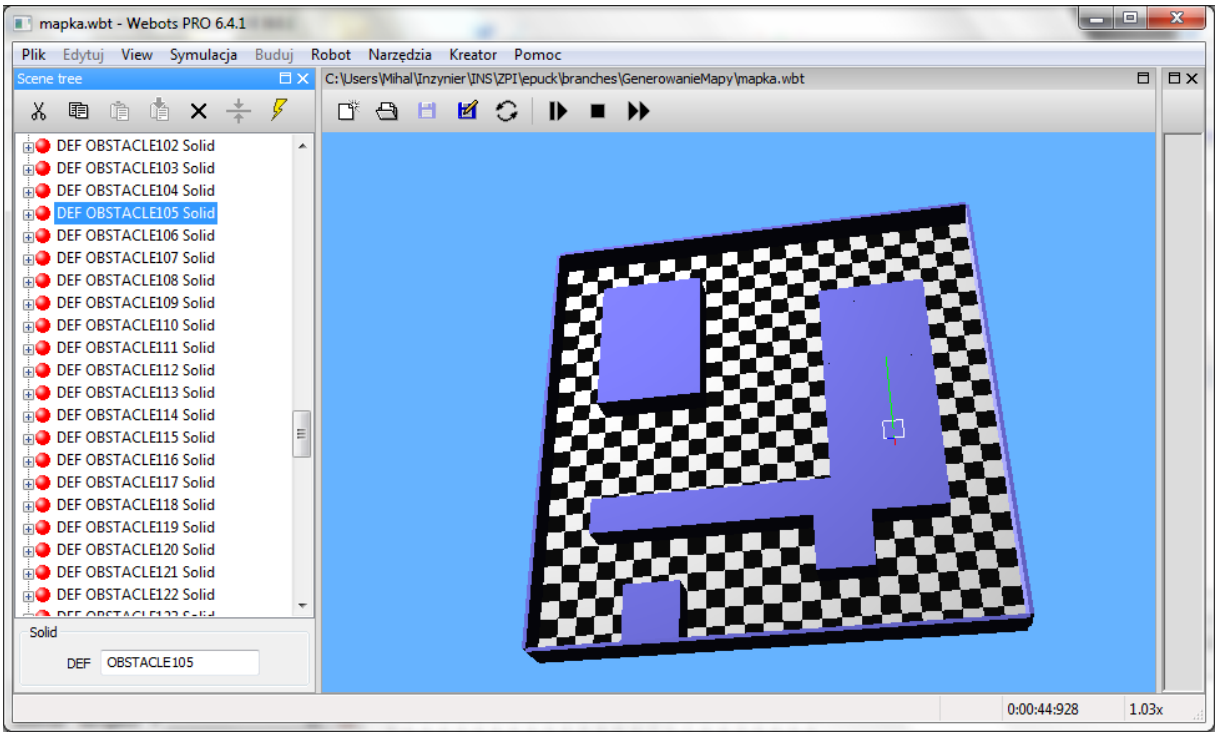




Podłoga i ściany dodają się automatycznie, dopasowując się do podanych przez użytkownika wymiarów.

3.2.2 Konwerter z .txt do .wbt

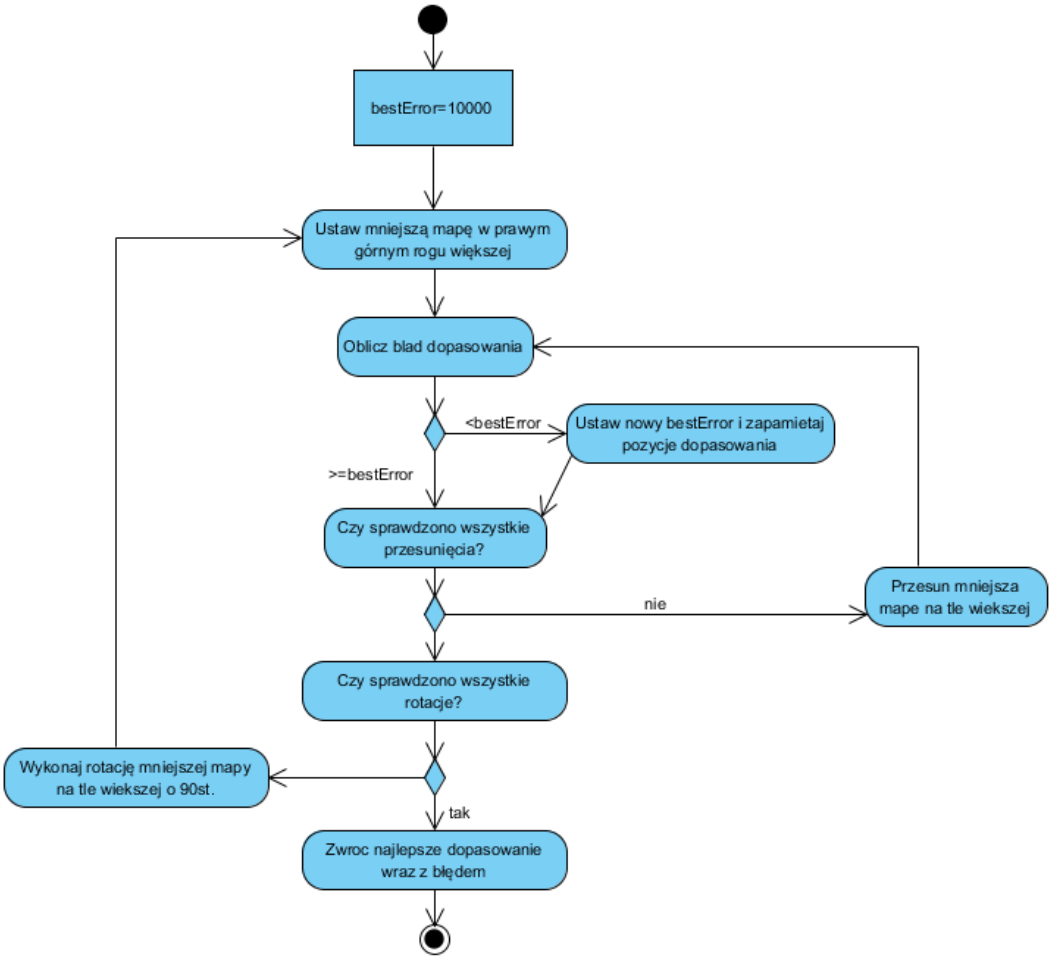
Mapa wygenerowana przez roboty w formacie .txt, aby zostać zwizualizowana musi zostać przekształcona w obiekty środowiska symulacyjnego. Tak jak na zdjęciu powyżej, wolna przestrzeń jest oznaczona cyfrą ‘0’, a przeszkoda cyfrą ‘1’. Konwersja polega na utworzeniu czystej mapy o wymiarach (liczonych w kratkach) takich samych jak w pliku tekstowym. Następnym etapem jest skanowanie macierzy w poszukiwaniu ‘1’-ek. Gdy program na takową trafi dodaje do mapy blok o podstawie kwadratu odpowiadający rozmiarem jednej kratce.



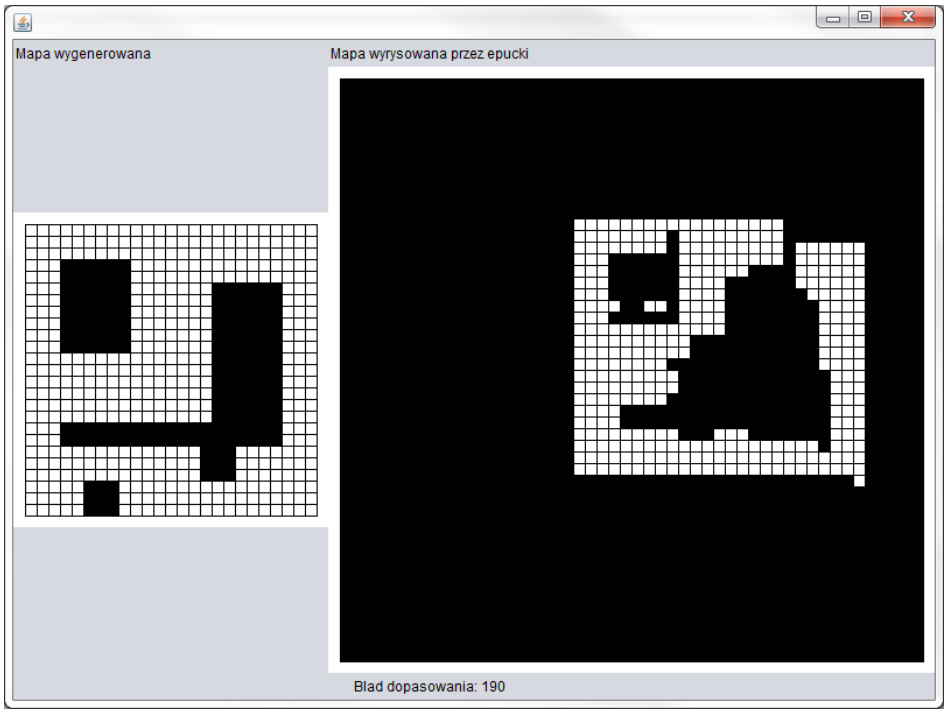
3.2.3 Porównywanie map

Funkcja ta jest wykorzystywana do sprawdzenia dokładności odwzorowania powierzchni przez epucki. Porównywane są zatem dwa pliki .txt, jeden wygenerowany przez użytkownika, a drugi przez roboty. Ten drugi ma większe wymiary, ponieważ nie znając dokładnego położenia początkowego epuck-a nakładany jest po bokach większy margines. Nie wiadomo jednak czy obie mapy są ułożone tak samo, gdyż może występować obrót o 90°, 180° i 270°. Aby porównanie w takich warunkach było możliwe potrzebne jest obracanie i przesuwanie mniejszej mapy na tle większej.

Algorytm wygląda następująco:



Przykładowy wynikowy ekran jest widoczny na załączonym rysunku.



4. Projekt systemu

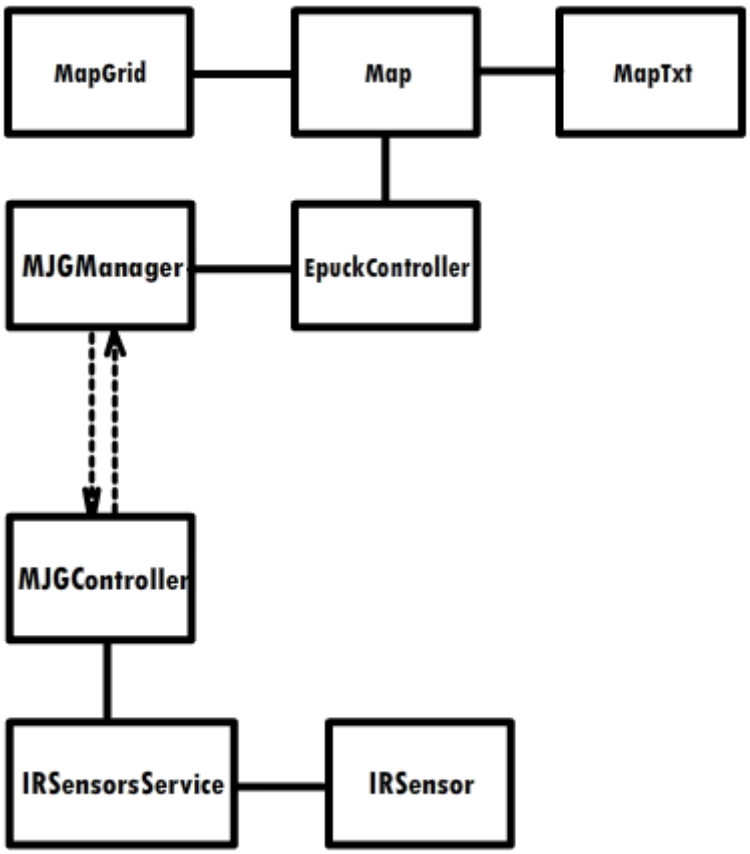
4.1 Schemat systemu

Schemat systemu przedstawia ogólną strukturę klas w projekcie.

MJGManager wraz z klasą EpuckController jest logiką działania epuck-ów, wspierającą się obiektami map.

MJGController jest sterownikiem epuck-a, który wykonuje rozkazy i zwraca wartości czujników i kamer.

Dokładniejszy opis poszczególnych klas znajduje się w punktach pod schematem.



4.2 Opis klas

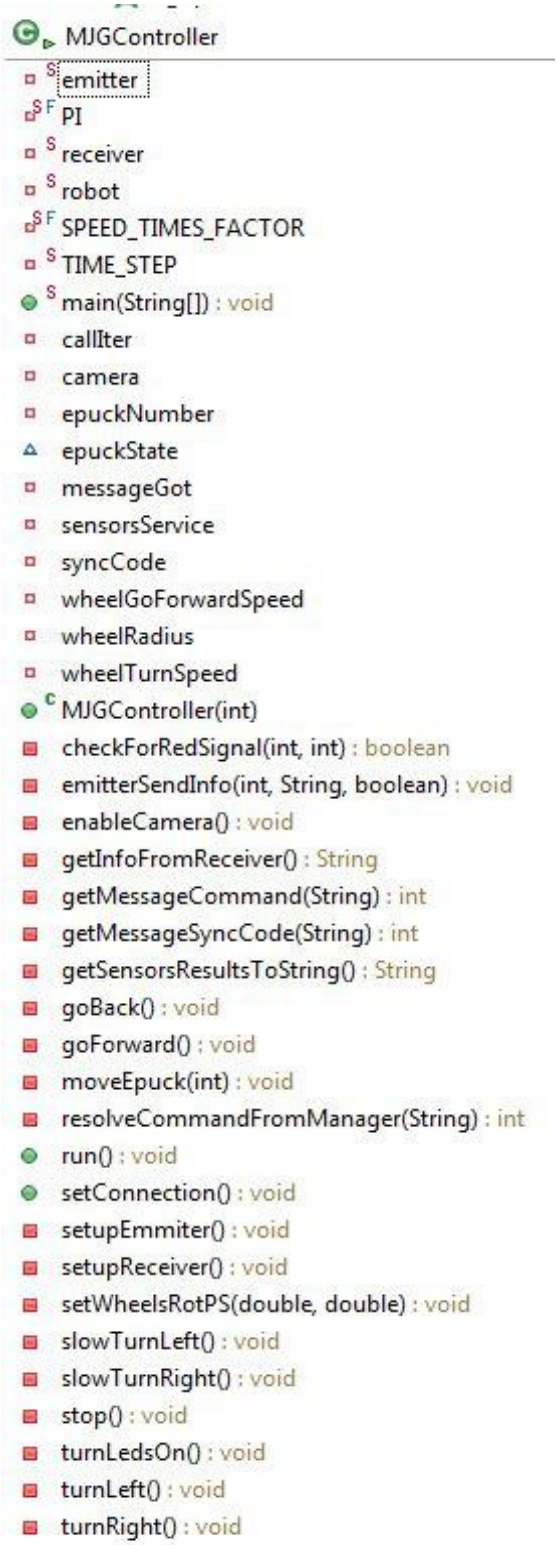
Opis klas pozwoli użytkownikowi na zrozumienie i zapoznanie się z logiką kodu umieszczonego w projekcie. Nie będzie tu samego kodu, lecz opis słowny pozwalający na zrozumienie działania danej metody/klasy bez konieczności wgłębiania się w semantykę. Będzie on bardzo pomocny przy próbie modyfikacji projektu dla swoich potrzeb.

Opisane zostaną główne metody i zmienne w danej klasie, poza nieznacznymi bądź mówiącymi samymi za siebie (np. toString()), bądź zawierające alternatywne rozwiązania, które mogą być pomocne przy własnych eksploracjach, jednak wymienione są wszystkie na wylistowaniu ze środowiska eclipse.

Metody i zmienne opisywane są w miarę możliwości w logicznej kolejności.

4.2.1 MJGController

Klasa MJGController jest kontrolerem odpowiedzialnym za odczyty danych z elementów epucka, komunikację z MJGManagerem oraz wykonywanie ruchów robota.



Zmienne:

Receiver receiver, Emitter emiter – nadajnik I odbiornik robota

double SPEED_TIMES_FACTOR – ustawienie przyspieszenia wykonywania zadań przez epucka – dzielnik TIME_STEP i czynnik prędkości nadawanej kołom.

int epuckNumber – numer epucka, ustawiany z hierarchii obiektów w środowisku z pola „controller args”

IRSensorsService sensorsService – obiekt klasy czujników obsługujący ich logikę.

double wheelTurnSpeed – prędkość pozwalająca obrócić się epuckowi o kąt 90 stopni w czasie jednej iteracji (TIME_STEP)

double wheelGoForwardSpeed – prędkość pozwalająca pojechać epuckowi do przodu na odległość 1/3 swojej średnicy w czasie jednej iteracji.

enum State {WAITING, RESOLVING, CALIBRATING} – stany epucka w zależności od których wykonuje rózne czynności

Metody:

void run() - uruchamia kamerę (enableCamera()), włącza diody (turnLedsOn()), wykonuje logikę ruchu epucka – sprawdza czy jest w stanie kalibracji, zatrzymuje epucka w celu oczekiwania na dalsze decyzje, wykonuje ruch zlecony przez managera, wysyła zaktualizowane dane z czujników i kamery.

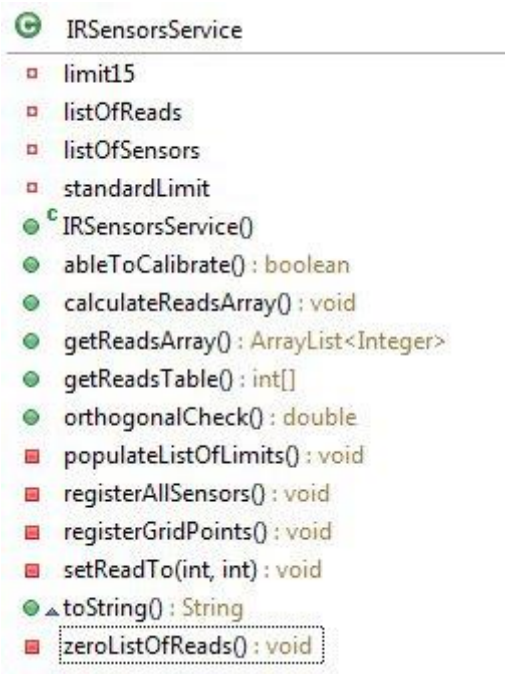
boolean checkForRedSignal(int minRedStrength, int maxRestStrength) – sprawdzamy, czy przed nami jest epuck po liczbie czerwonych pikseli na ekranie, w symulacji jest to liczba 3, a w rzeczywistości ze względu na większe szumy zalecane jest zwiększenie tej liczby do 10.

void setConnection() – ustawienie emiterów i receiverów.

int resolveCommandFromManager(String commandFromManager) – w zależności od informacji otrzymanej z MJGManager’a wykonujemy określony ruch (**void moveEpuck(int command)**) – -1 odpowiada za skręt w lewo, 0 za jazdę do przodu, 1 za skręt w prawo, 2 za zatrzymanie, -11 za wolny obrót w lewo (do kalibracji), 11 za wolny obrót w prawo (do kalibracji).

4.2.2 IRSensorsService

Klasa odpowiedzialna za kompleksową obsługę czujników epuck-a, od ich aktywacji, po odczyty oraz wnioskowanie. Z niej MJGController, a tym samym epuck posiada wiedzę o przeszkodach go otaczających.



Zmienne:

int standardLimit – zmienna opisująca limit zauważenia przeszkody dla czujników 0,2,3,4,6.

int limit15 – limit zauważenia przeszkody dla czujników 1,5 (powinien być inny ze względu na większą odległość pokonywaną przez sygnał czujnika.

ArrayList<IRSensor> listOfSensors – lista zawierająca obiekty sensorów

ArrayList<Integer> listOfReads – lista zawierająca odczyty poszczególnych czujników – wartości 0, gdzie nie ma wykrycia przeszkody i 1 gdy przeszkoda jest wykryta.

Metody:

public IRSensorsService() – konstruktor klasy, inicjuje rejestrację sensorów(registerAllSensors()), punktów wykrycia przeszkód dookoła epuck-a (registerGridPoints()), oraz uzupełnia listę ograniczeń dla konkretnych sensorów(populateListOfLimits()).

void calculateReadsArray() – najważniejsza metoda w tej klasie. Jest odpowiedzialna za wyliczenie wykrycia przeszkód na poszczególnych polach wokół epuck-a na podstawie danych z czujników.

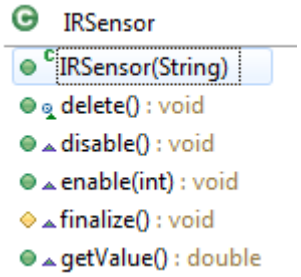
ArrayList<Integer> getReadsArray(),int[] getReadsTable() – metody zwracające odczyty z czujników w zależności od potrzebnego typu.

boolean ableToCalibrate() – metoda sprawdzająca, czy epuck posiada naprzeciw siebie przeszkodę, przy pomocy której może skalibrować swoją pozycję

double orthogonalCheck() – sprawdzamy czy epuck wymaga kalibracji (czy stosunek odczytu czujnika 3 do 4 nie przekracza granicy błędu).

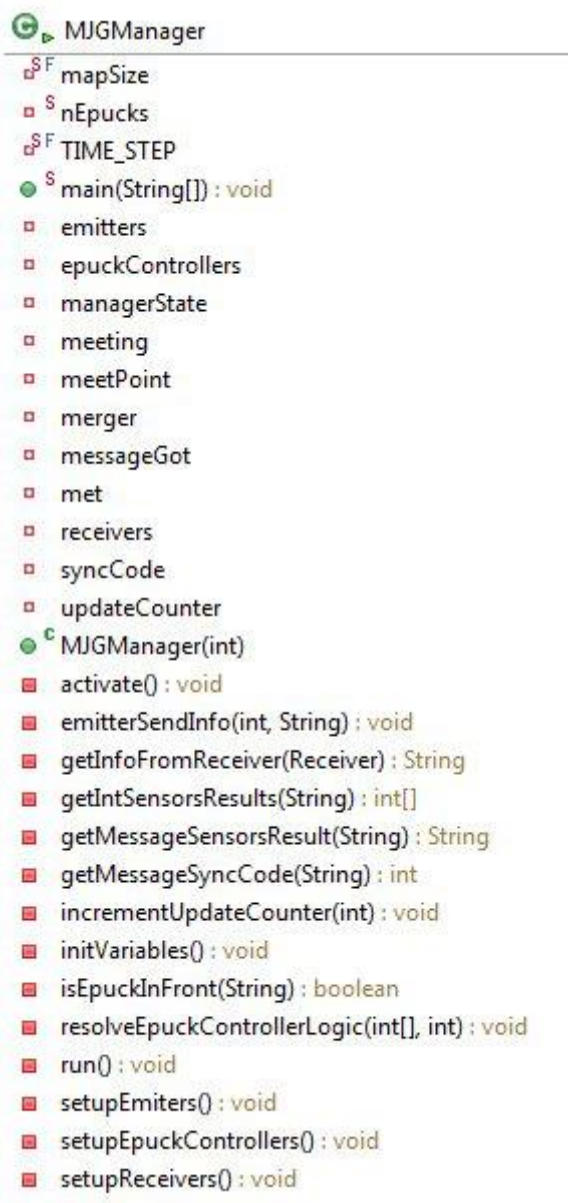
4.2.3 IRSensor

Klasa wykorzystana do obsługi czujników zbliżeniowych, dziedzicząca po klasie biblioteki - DistanceSensor.



4.2.4 MJGManager

Klasa odpowiedzialna za główną logikę systemu. Zawiera zarówno element mapowania jak i lokalizacji epuck-ów. Odbiera odczyty z kamer i czujników poszczególnych epuck-ów a następnie sprawdza spełnienie warunków rozpoznania drugiego epuck-a w kamerze, rejestruje odczyty z czujników na odpowiednich mapach, podejmuje próby łączenia map oraz wysyła następne dyspozycje ruchu epuck-ów.



Zmienne:

int mapSize – zmienna rozmiaru mapy

int nEpucks – tą zmienną musimy ustawić na pożądaną liczbę epucków

int TIME_STEP - zmienna określająca częstotliwość kolejnych iteracji wykonania algorytmu w milisekundach

Emitter[] emitters, Receiver[] receivers – tablica emiterów, każdy epuck posiada odpowiadający sobie emiter i receiver, z którym komunikuje się na kanale swojego numeru.

EpuckController[] epuckControllers - tablica przechowuje „mózgi operacyjne” poszczególnych epucków.

enum State {WAITING, RESOLVING} – stany managera, odpowiadające bezczynności czyli okresowi oczekiwania na kolejne informacje od epucków oraz stanowi rozwiązywania, czyli procesowaniu danych i wykonywania logiki systemu.

State[] managerState – tablica stanów managera dla poszczególnych epucków (możliwa praca asynchroniczna)

String[] messageGot – tablica przechowująca wiadomości od konkretnych epucków.

int[] syncCode – zmienna pozwalająca określić aktualność iteracji i synchronizację wiadomości pomiędzy epuckiem a managerem.

int[] updateCounter – zmienna pamiętająca liczbę iteracji wykonaną na każdym z danych konkretnego epucka.

int[][] meeting - Macierz punktów, w których roboty się zobaczyły, jest jednocześnie punktem wokół którego obracamy mapy w celu ich połączenia.

Comparer merger - obiekt klasy comparer, w którego konstruktorze przekazywane są mapy, a także punkt spotkania, potrzebny do wytworzenia mapy wynikowej.

boolean met - zmienna mówiąca o tym czy roboty się już spotkały, zmienna ta odpowiedzialna jest też za aktywowanie możliwości łącznia map.

Metody:

MJGManager(int number) – konstruktor klasy, jako parametr przyjmuje liczbę epuck-ów które mają uczestniczyć w symulacji. Przypisuje zmiennej nEpucks wartość, którą otrzymał w parametrze.

void activate() – metoda odpowiedzialna za aktywację elementów managera – ustawienie emiterów(**setupEmiters()**), ustawienie receiver-ów (**setupReceivers()**), zainicjowanie logik epuck-ów (**setupEpuckControllers()**)

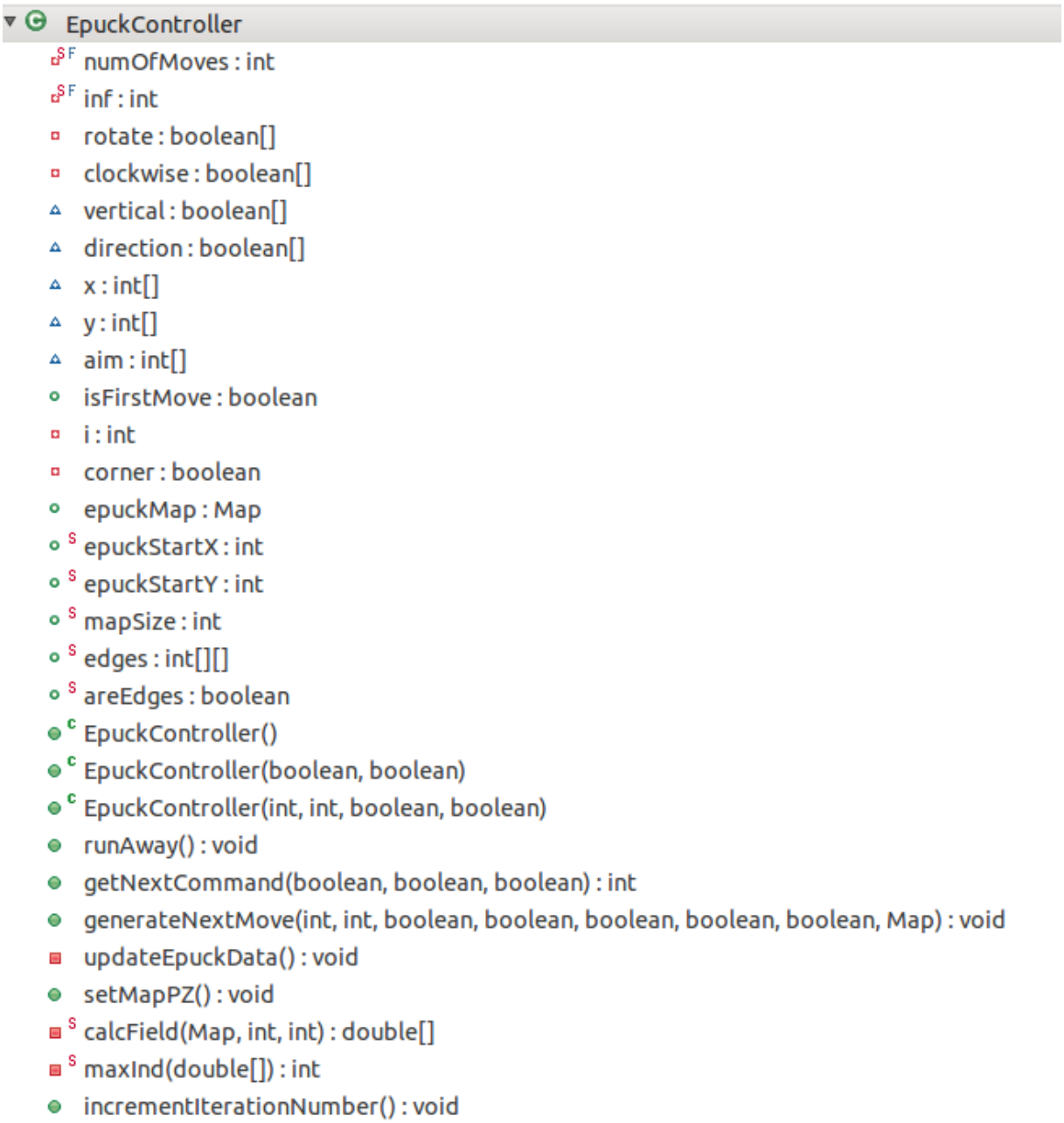
void emitterSendInfo(int epuck, String command) – wysłanie informacji o ruchu emiterem, do konkretnego epuck-a

String getInfoFromReceiver(Receiver receiver) – odebranie informacji z receiver-a

resolveEpuckControllerLogic(int[] sensorsResults, int epuck) - funkcja wywołująca realizację logiki epuck kontrolera polegająca na naniesieniu odczytów z sensorów na mapę, a także zaktualizowania pozycji i stanu e-pucka.

4.2.5 EpuckController

Klasa odpowiedzialna za logikę e-pucka, obliczająca nastepny krok, metodą potencjałów posiadająca mapę e-pucka, a także będąca częścią MJGManagera, który za jej pośrednictwem steruję e-puckiem. Klasa ta przechowuję też wszelkie informacje o e-pucku, łącznie z historią i pokonaną trajektorię.



Zmienne:

int numOfMoves – limit ruchów, po których należy skończyć eksplorację terenu,

int inf – zdefiniowana bardzo duża liczba rozumiana jako odpowiednik nieskończoności powstrzymująca robota przed jazdą w ścianę,

boolean[] rotate – tablica przechowująca czy w i-tej iteracji robot obrócił się (true) czy może pojechał prosto (false).

boolean[] clockwise – tablica przechowująca kierunek obrotu w i-tej iteracji. Obrót w prawdo odpowiada wartości true, z kolei w lewo wartości false. W przypadku jazdy prosto funkcja ta przyjmuje wartość true.

boolean[] vertical – tablica przechowująca „polaryzację" e-pucka odpowiadająca na pytanie czy w i-tej iteracji robot był ustawiony pionowo (true) czy poziomo (false)

boolean[] direction– tablica przechowująca kierunek e-pucka odpowiadająca na pytanie czy w i-tej iteracji robot był ustawiony w kierunku dodatnim osi, czyli w prawo lub do góry (true) czy ujemnym (false)

int[] x – tablica przechowująca współrzędną x e – pucka w i-tej iteracji.

int[] y – tablica przechowująca współrzędną y e – pucka w i-tej iteracji

int[] aim – tablica przechowująca cel(wybrany kierunek) w i-tej iteracji

boolean isFirsMove – zmienna kontrola wybierająca jazdę do przodu w pierwszej iteracji

int i – zmienna iteracyjna we wszystkich tablicach,

boolean corner - zmienna odpowiadająca za detekcję bycia w narożniku i trwający kilka iteracji „cooldown"

Map epuckMap – objekt klasy Map przechowujący mapę robota.

int epuckStartX – początkowa pozycja x e-pucka,

int epuckStartY – początkowa pozycja y e-pucka,

int mapSize – rozmiar mapy e-pucka,

int[][] edges – tablica 2x2 przechowująca aktualne informację o najbardziej zewnętrznych odkrytych punktach, z czasem zmieniająca się w dwa przeciwne narożniki mapy,

boolean areEdges – zmienna kontrolna przyjmująca wartość true, gdy zostały znalezione wszystkie krawędzie

Metody:

EpuckController() – domyślny kontroler tworzący mapę e-pucka zorientowanego na północ o współrzędnych x=10, y=10;

EpuckController(boolean epuckVertical**, boolean** epuckDirection**)** – konstruktor tworzący kontroler robota o zadanej orientacji początkowej wskazywanej dzięki dwóm zmiennym boolean odpowiednio epuckVertical i epuckDirection.

EpuckController(int xE**, int** yE**, boolean** epuckVertical**, boolean** epuckDirection**)** – konstruktor tworzący kontroler robota nie tylko o zadanej orientacji, ale także o zadanych współrzędnych porządkowych odpowiednio x i y.

runAway() – funkcja wykonująca procedurę wycofywania się w przypadku spotkania drugiego e-pucka.

getNextCommand(boolean left_sensor**, boolean** front_sensor**, boolean** right_sensor**)** – funkcja nadrzędna do, której przekazywane są wyniki z sensorów, w niej wykonywana jest funkcja generateNextMove. Funkcja ta zwraca komendę -1 (obrót w lewo), 1 (obrót w prawo), 0 (jazda prosto) lub 2 (stanie w miejscu)

generateNextMove(int epuckX**, int** epuckY**, boolean** epuckPolarisation**, boolean** epuckDirection**, boolean** left_sensor**, boolean** front_sensor**, boolean** right_sensor**, Map** epuckMap**)** – funkcja odpowiedzialna za wygenerowanie kolejnego kroku zgodnie z metodą potencjałów, na wyjściu podawany jest kierunek ze zbioru {0,1,2,3}, który następnie interpretowany w metodzie getNextCommand.

updateEpuckData() – Metoda uaktualniająca stan e-pucka po wykonaniu polecenia.

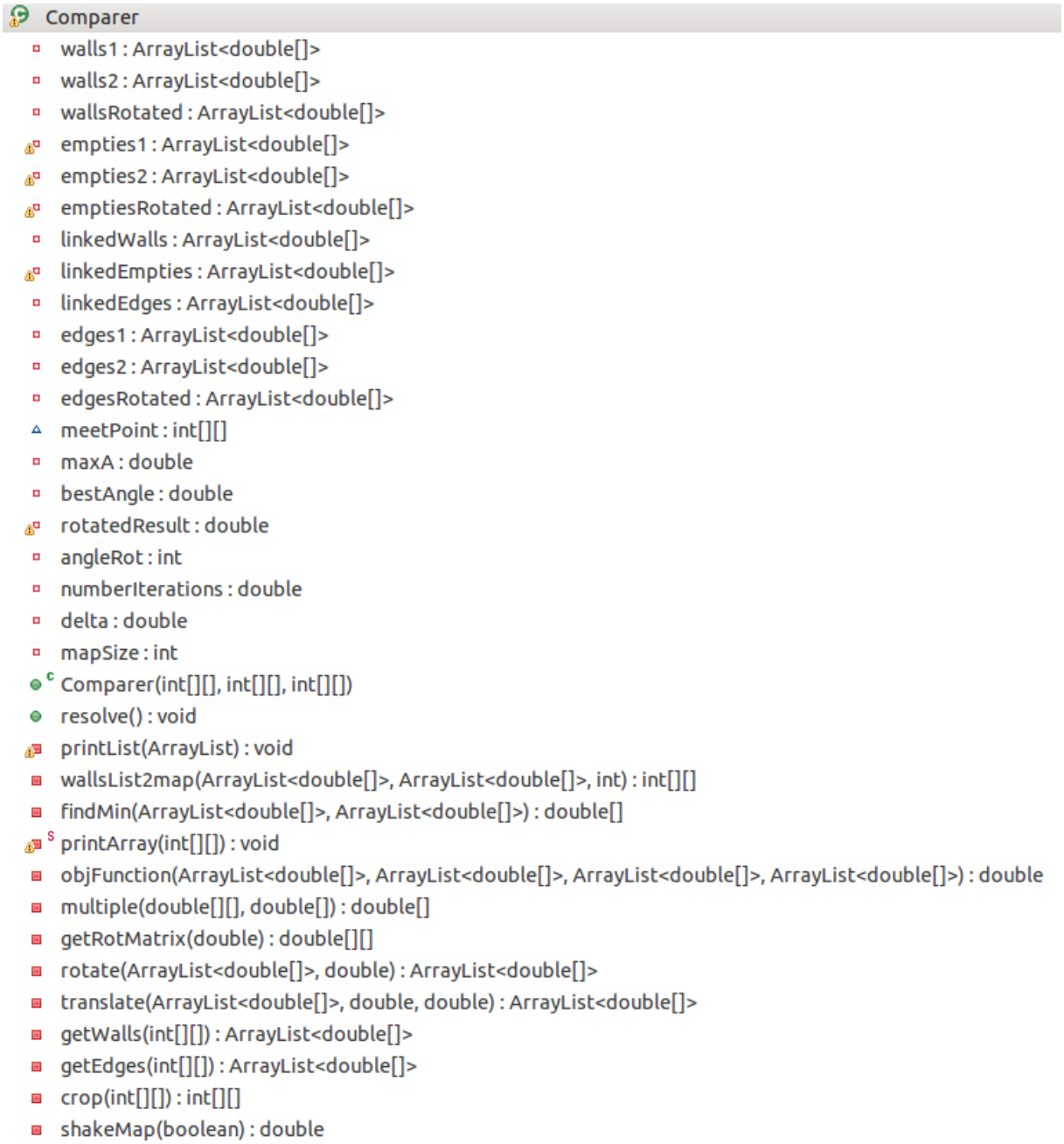
double[] calcField() – metoda wywołana w funkcji generateNextMove do obliczenia pola sił w danym miejscu. Na tej podstawie obliczany jest kierunek ruchu e-pucka.

Int maxInd(double[] dane**)** – funkcja zwracająca indeks maksymalnego wyrazu z wejściowej macierzy.

inceremenIterationNumber() – metoda zwiększająca zmienną i o jeden.

4.2.6 Comparer

Klasa ta odpowiada za porównywanie i łączenie otrzymanych map na podstawie lokalizacji przeszkód, a także krawędzi mapy. Wykorzystuje ona podstawowe przekształcenia afiniczne takie jak obroty i translację w calu nałożenia dwóch macierzy na siebie.



Zmienne:

ArrayList<double[]> walls1, walls2, wallsRotated – listy przechowujące współrzędne przeszkód w postaci wektora dwóch współrzędnych {x,y}

ArrayList<double[]> empties1, empties2, emptiesRotated – listy przechowujące współrzędne pustych pól w postaci wektora dwóch współrzędnych {x,y}

ArrayList<double[]> edges1, edges2, edgesRotated – listy przechowujące współrzędne krawędzi w postaci wektora dwóch współrzędnych {x,y}

ArrayList<double[]> linkedWalls, linkedEdges, linkedEmpties – listy przechowujące współrzędne odpowiednio połączone ściany, krawędzie i puste pola w postaci wektora dwóch współrzędnych {x,y}.

Int[][] meetPoint – punkt spotkania się dwóch robotów wokół którego wykonujemy obrót.

Metody:

Comparer (int[][] meet, **int[][]** eMap**, int[][]** eMap2**)** – kontruktor klasy z argumentami będącymi zestawem punktów spotkania oraz mapami obydwu robotów.

resolve() – metoda wywołująca główną logikę klasy, a więc liczenie najlepszego dopasowania dwóch map, kończąca się wyświetleniem mapy i zapisaniem wynikowej nałożonej mapy w postaci .txt

printList(ArrayList), printArray(int[][]) – metody do wyświetlania odpowiednio listy i macierzy w postaci tekstowej

wallsList2map (ArrayList<double[]> wallsAll**, ArrayList<double[]>** edgesAll**, int** size**) –** funkcja konwertujaca listy ścian I krawędzi na mapę, a więc macierz o rozmiarze size.

findMin(**ArrayList<double[]>** wallsAll**, ArrayList<double[]>**edgesAll**) –** funkcja mająca za zadanie znaleźć najmniejszy element połączonych list z argumentu w celu przesunięcia mapy i pozbycia się ujemnych wartości

objFunction(**ArrayList<double[]>** walls1**, ArrayList<double[]>** walls2**, ArrayList<double[]>** edges1**, ArrayList<double[]>** edges2**) –** funkcja licząca dopasowanie na postawie algorytmu opisanego w punkcie 3.

Multiple(double[][] A**, double[]** B**) getRotMatrix(double** angle**), rotate(double[] A, double** angle**), translate(double[]** A**, double** dx**, double** dy**)** – funkcje realizujące podstawowe operacje algebraiczne w kolejności mnożenie macierzy przez wektor, macierz rotacji, funkcja rotująca dane współrzędne i translacja współrzędnych.

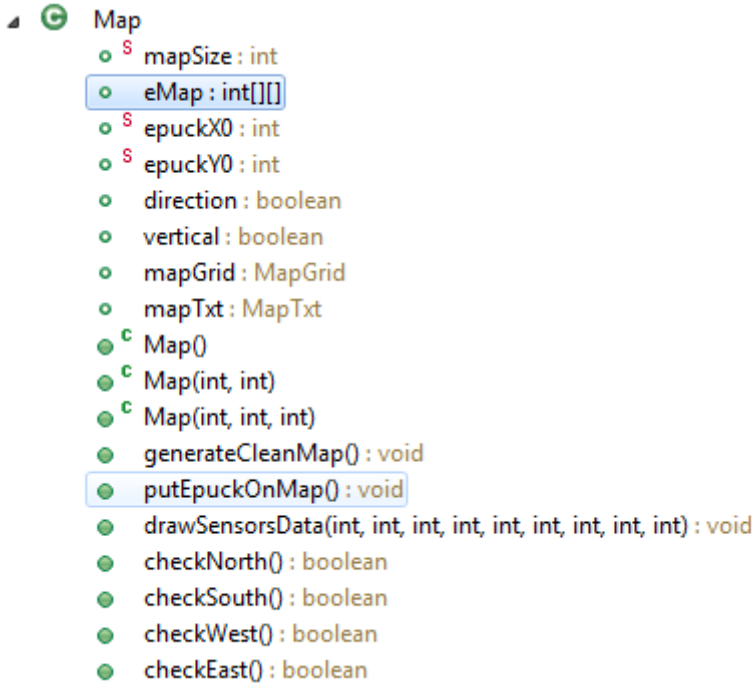
ArrayList<double[]> getWalls(int[][] eMap**) –** funkcja pozwalająca pobrać współrzędne przeszkód z mapy i zapisać je do listy.

ArrayList<double[]> getEdges(int[][] eMap**) –** funkcja pozwalająca pobrać współrzędne krawędzi z mapy i zapisać je do listy.

double shakeMap(boolean) – funkcja odpowiadająca za przegląd zupełny sąsiednich punktów w celu jak najlepszego nałożenia na siebie dwóch map. Jest to realizacja algorytmu z pkt. 3.

4.2.7 Map

Klasa ta odpowiada za moduł mapy w systemie. Zarządza ona mapą wyświetlaną podczas symulacji jak i zapisem wygenerowanej mapy do pliku tekstowego.



Zmienne:

mapSize – rozmiar kwadratowej mapy, na której epuck będzie wykonywał operacje mapowania

eMap - kwadratowa macierz, na której poruszający się robot będzie odwzorowywał przejechaną powierzchnię. Specjalnie jest dużo większa od rozmiarów makiety, aby można było ułożyć robota w dowolnym miejscu i była pewność, że nie wyjedzie poza nią.

epuckX0 – współrzędna x początkowego ułożenia robota na macierzy eMap (wartość domyślna 20).

epuckY0 – współrzędna y początkowego ułożenia robota na macierzy eMap (wartość domyślna 20).

vertical – pierwsza wartość określająca kierunek jazdy

direction – druga wartość określająca kierunek jazdy (dokładny opis w rozdziale 3.1.4.4)

mapGrid – obiekt wyświetlający tworzoną przez epuck-a mapę

mapTxt – obiekt zapisujący do pliku tworzoną przez epuck-a mapę

Metody:

Map() – konstruktor domyślny

Map(int,int) – konstruktor, który ustawia położenie początkowe robota

Map(int,int,int) - konstruktor, który ustawia położenie początkowe robota oraz rozmiar mapy, na której się znajduje

generateCleanMap() – funkcja ustawiająca wszystkie elementy eMap na wartość '2', czyli nieodwiedzone przez robota. Inicjalizuje również mapGrid i mapTxt.

putEpuckOnMap() – w miejscu wzkazanym przez epuckX0 i epuckY0 oraz przyległym (robot zajmuje 3x3 kratki) zmienia wartość elementów macierzy na symbolizujące odkrytą powierzchnię.

drawSensorsData(int,int,int, int,int,int, int,int,int) – funkcja aktualizuje obiekt mapGrid poprzez podanie jej eMap, który wyświetla na panelu aktualny stan mapy. eMap zostaje również zapisany przez obiekt mapTxt do pliku. Dodatkowo funkcja ta aktualizuje samą macierz eMap o wyniki interpretacji odczytów z czujników odległości, które są podawane jako jej argumenty.

checkNorth() – funcja sprawdza w macierzy eMap, czy na północ od aktualnej pozycji epuck- a znajduje się przeszkoda i zwraca odpowiedź.

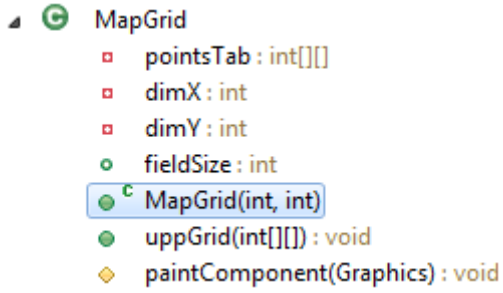
checkSouth() – funcja sprawdza w macierzy eMap, czy na południe od aktualnej pozycji epuck- a znajduje się przeszkoda i zwraca odpowiedź.

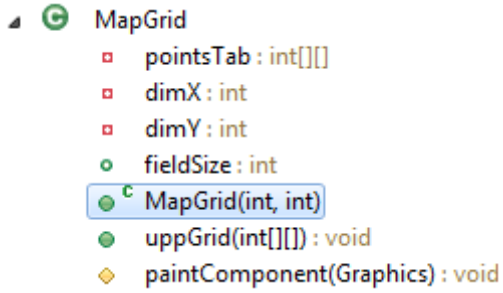
checkWest() – funcja sprawdza w macierzy eMap, czy na zachód od aktualnej pozycji epuck- a znajduje się przeszkoda i zwraca odpowiedź.

checkEast() – funcja sprawdza w macierzy eMap, czy na wschód od aktualnej pozycji epuck- a znajduje się przeszkoda i zwraca odpowiedź.

4.2.8 MapGrid

Klasa, której zadaniem jest wyświetlanie panelu z aktualną mapą, którą przejechał epuck.





Zmienne:

pointsTab – macierz punktów odwzorowująca przejechaną przez robota powierzchnię. Składa się z wartości :

0 – wolna przestrzeń, robot po niej przejechał i nic nie znalazł
1 – napotkana przeszkoda
2 – nieodkryty punkt
3 – punkt spotkania dwóch epuck-ów
4 – punkt ściany
5 – odcięta część mapy

dimX – liczba wierszy macierzy pointsTab

dimY – liczba kolumn macierzy pointsTab

fieldSize – liczba pikseli na wyświetlanym panelu odpowiadająca długości boku 1 punktu w macierzy

Metody:

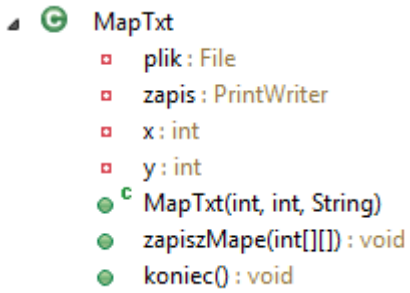
MapGrid(int, int) – konstruktor, inicjalizuje pointsTab i nadaje jej rozmiar

uppGrid(int[][]) – aktualizuje macierz z punktami, a następnie wywołuje funkcję repaint() należącą do klasy JPanel w celu naniesienia na niego nowych punktów odkrytych przez robota.

paintComponent(Graphic) – funkcja rysująca siatkę oraz punkty znajdujące się w macierzy pointsTab na panelu. Każda wartość ma swój indywidualny kolorystyczny odpowiednik.

4.2.9 MapTxt

Zadaniem tej klasy jest zapis wygenerowanej przez robota mapy do pliku .txt w celu późniejszych porównań z mapą narysowaną przez użytkownika.



Zmienne:

plik – plik, w którym zapisywana jest mapa

zapis – bufor odpowiadający za drukowanie znaków w pliku

x – liczba wierszy macierzy, która będzie zapisywana w tym pliku

y – liczba kolumn macierzy, która będzie zapisywana w tym pliku

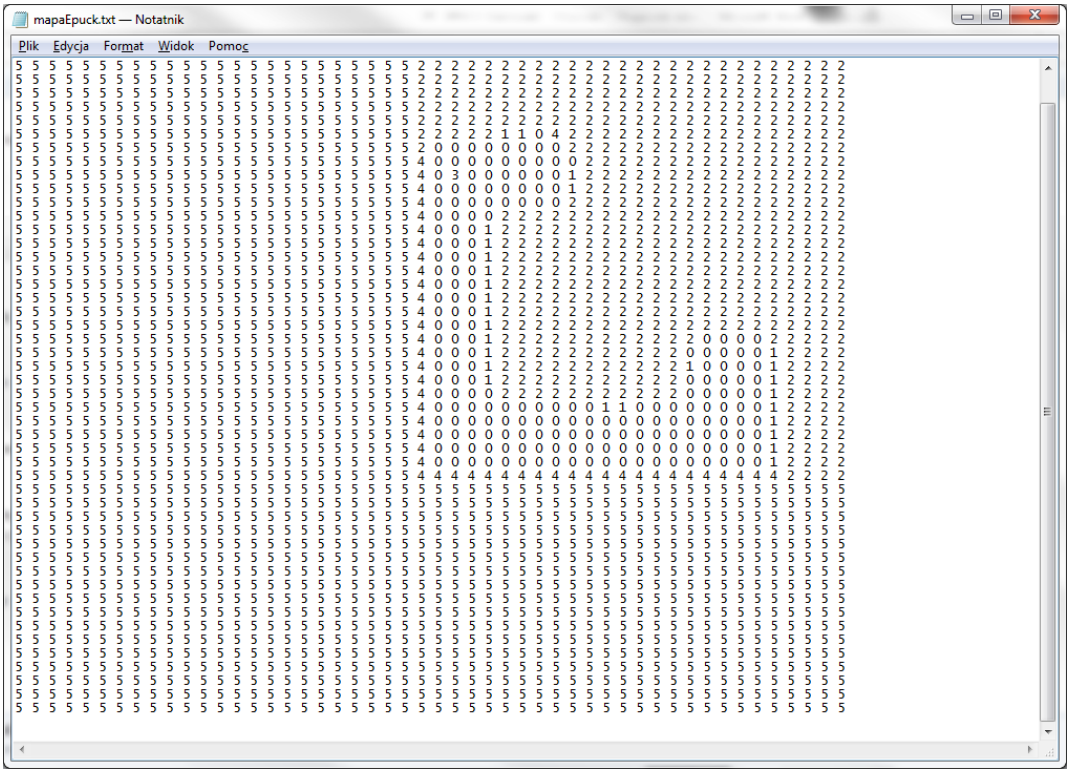
Metody:

MapTxt(int, int, String) – konstruktor ustalający rozmiary macierzy zapisywanej do pliku oraz jego nazwę

zapiszMape(int[][]) – funkcja zapisująca podają w argumencie macierz do pliku

koniec() – finkcja zamukająca bufor zapisu

Przykładowy plik stworzony za pomocą tej klasy:



5. Opis przebiegu pracy

Nasza praca jest dużym projektem zespołowym, w którym mieliśmy do wykonania wiele zadań oraz wiele problemów do rozwiązania. W trakcie rozwoju systemu natrafialiśmy na różne przeszkody oraz nowe zagadnienia, które należało opracować i zaimplementować. Harmonogram zawiera wszystkie tygodnie trwania naszego projektu oraz przypisane im zadania.

5.1 Problemy i trudności realizacji.

Podczas realizacji Zespołowego przedsięwzięcia inżynierskiego natrafiliśmy na wiele problemów, nie tylko analitycznych i algorytmicznych, ale również natury technicznej, szczególnie w momencie przejścia na rzeczywiste środowisko.

5.1.1 Środowisko symulacyjne

5.1.1.1 Wybór i konfiguracja środowiska

Na początku projektu założyliśmy, iż skorzystamy z darmowego środowiska symulacyjnego V-REP, lecz po wstępnym rozeznaniu okazało się, że nie spełnia ono naszych oczekiwań i jest niezbyt przyjazne w użytkowaniu oraz słabo udokumentowane. Dlatego po pewnym czasie wybraliśmy środowisko Webots, ze względu na posiadaną przez uczelnię licencję, oraz przykładowe projekty, z których mogliśmy dowiedzieć się o podstawach komunikacji.

Po wyborze środowiska sporą część czasu przez praktycznie cały projekt zajęła praca nad ekstrakcją części kodu z języka C i przeniesienie go na język Java, który pozwala na strukturę obiektową, bardziej zrozumiałą i lepszą do współpracy przy dużym projekcie.

5.1.1.2 Organizacja pracy

Jak w każdym projekcie zespołowym zaistniał problem organizacji pracy. Do rozwiązania problemu wybraliśmy spośród siebie „osobę decyzyjną", która kierowała projektem, wyznaczaniem zadań i wyborami ścieżek grupy, oczywiście po uprzedniej konsultacji z resztą członków.

Do dodatkowego usprawnienia tej struktury posłużyliśmy się dodatkowymi narzędziami:

- https://freedcamp.com/ - narzędzie do zarządzania projektem z listami „To-Do", panelem dyskusyjnym, kamieniami milowymi, opcją przypisywania zadań do osób i kalendarzem projektu.

- SVN – System kontroli wersji znacznie ułatwił wspólne tworzenie kodu, pozwolił na bezproblemową współpracę przy jednej wersji projektu i udostępniał opcję zarówno przejrzenia zmian jak i powrotu do poprzedniej wersji w razie błędu.

- Skype/Facebook – do uzgadniania zagadnień ad-hoc.

5.1.1.3 Złożoność projektu, inne zagadnienia

W projekcie zostało poruszone bardzo dużo zagadnień które same w sobie były bardzo dużym wyzwaniem. Obejmowały między innymi:

- Problem rozpoznania środowiska, podpięcia do niego odpowiednich bibliotek
- Praca na szczątkach dokumentacji, co wiązało się z pracą pomiędzy językami programowania
- Mapowanie (opisany szerzej w dokumentacji)
- Lokalizacja (opisana szerzej w dokumentacji)
- Komunikacja (brak bezpośredniego połączenia pomiędzy robotami a managerem, konieczność zastosowania komunikacji radiowej/bluetooth)

- Rozpoznawanie obrazów – rozpoznanie drugiego epucka za pomocą kamery
- Stworzenia generatora map
- Porównywania map

Ze względu na poczynione założenia większość rozwiązań musieliśmy wymyślić od początku wspierając się literaturą. To również miało znaczny wpływ na ocenę trudności projektu.

5.1.2 Środowisko rzeczywiste

5.1.2.1 Laboratorium

Pierwszym problemem przy przejściu z symulacji komputerowych na rzeczywiste roboty były problemy z odpowiednią wersją środowiska symulacyjnego. Na laboratoryjnych komputerach brak było wersji posiadającej dostęp do wszystkich funkcji. Mimo posiadania kluczy dostępu do wersji PRO nie zawsze one działały i zdarzyły się dni, gdzie byliśmy zmuszeni zaniechać pracę na prawdziwych epuckach.

Ostatecznie udało nam się znaleźć archiwalną wersję Webots, która zadziała na przeterminowanej licencji w wersji PRO, gdyż tylko taka umożliwia korzystanie z modułu Supervisora.

5.1.2.2 Komunikacja bluetooth

Następną przeszkodą do sprawnego testowania systemu na rzeczywistych robotach były problemy z komunikacją bluetooth. Na komputerach przenośnych przerywało połączenie, bądź nie udawało się w ogóle połączyć z robotami.

Po wielu próbach działania na politechnicznych modułach bluetooth zakupiliśmy własny, który jest na tyle dobry, że niwelował szumy, miał większy zasięg i nie tracił połączenia

5.1.2.3 Poślizg kół

Nie udało się z pełną dokładnością przenieść działania robotów z symulacji na rzeczywiste roboty. Jedną z różnic był poślizg kół na makiecie, w szczególności na złączeniach taśmą klejącą. Równa jazda zostawała wtedy zachwiana i delikatnie zbaczał z prostej drogi. Przy dłuższym działaniu systemu błąd ten się nakładał i na mapie pojawiały się nieprawdziwe dane.

Rozwiązanie na systematyczną eliminację błędu znaleźliśmy w autokalibracji, gdy robot dojedzie do ściany. Obraca się on wtedy w poszukiwaniu pozycji, w której odczyty z dwóch przednich czujników

są zbliżone do siebie. Po wyrównaniu swojej pozycji (ustawienie prostopadłe do przeszkody) kontynuuje algorytm jazdy.

5.1.2.4 Niedoskonałość czujników

Pomijając oczywiste zagadnienie szumów, które jednak nie były aż tak uciążliwe, poważnym problemem były jednak „ślepe pola" widzenia czujników. Epuck nawet stojąc parenaście milimetrów od przeszkody potrafił nie zarejestrować żadnych zmian w odczytach. Był to problem, który stał się ścianą zaporową dosyć mocno uniemożliwiającą powodzenie prób na rzeczywistych epuckach. Co ciekawe w symulacji „ślepe pola" praktycznie nie występowały, być może była to kwestia idealnych warunków odbicia materiałów.

6. Instrukcja dla użytkownika

6.1 Instalacja środowiska symulacyjnego

Do uruchomienia projektu wymagane jest środowisko symulacyjne Webots w wersji 6.3.2 (w tej wersji projekt był implementowany i rozwijany). Z uwagi na użycie w projekcie Supervisora, potrzebna jest licencja PRO. Na potrzeby symulacji w projekcie użyty został klucz Politechniki Wrocławskiej.

Oprogramowanie pobieramy ze strony wersji archiwalnych: http://www.cyberbotics.com/archive/windows/ , a następnie instalujemy przy użyciu standardowej konfiguracji.

Kolejnym krokiem jest instalacja Java SDK 7 (32bit). Użyta zostanie do kompilacji klas wewnątrz programu Webots. Pobieramy ją ze strony:

http://www.oracle.com/technetwork/java/javase/downloads/jdk7-downloads-1880260.html , a następnie instalujemy przy użyciu standardowych ustawień.

Aby Java była widziana przez kompilator w Webots, powinna być dodana jej ścieżka filderu „bin" do zmiennych środowiskowych w naszym systemie do zmiennej PATH. (Mój komputer > właściwości > zaawansowane ustawienia systemu > zmienne środowiskowe).

Do edycji plików .java używaliśmy środowiska Eclipse, ze względu na bardzo pomocne podpowiedzi kontekstowe pozwalające przeglądać metody dostępne w klasach. Zaimportować do Eclipse można cały folder jako nowy projekt, co powinno automatycznie podłączyć potrzebną bibliotekę Controller.jar znajdującą się w folderze /libs .

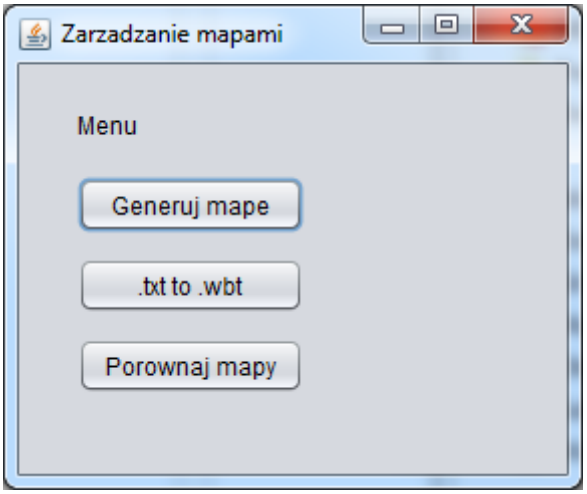
6.2 Opis struktury folderów projektu

Foldery projektu:

- /_materiały – folder z dosyć ciężką do znalezienia dokumentacją różnic pomiędzy wersjami Webots 5 a 6, które pozwalają na lepsze zrozumienie sposobu implementacji rozwiązań.
- /settings , /bin, /libs, /src /.classpath, /.project – foldery i pliki pozwalające na edycję projektu z poziomu IDE – Eclipse'a. W folderze /src są aktualne pliki .java, które następnie po edycji są kopiowane do poszczególnych kontrolerów środowiska Webots.
- /worlds – folder środowiska Webots zawierający mapy na których były przeprowadzane symulacje, ich tekstury oraz przykładowe projekty które pozwalały na „podejrzenie" sposobów pisania rozwiązań.
- /controllers – folder zawierający kontrolery pozwalające na obsługę obiektów w środowisku Webots. Zawierają pliki .java, które są następnie kompilowane i obsługują całą logikę zachowań.
- /controllers/MJGManager – w tym folderze znajduje się logika Supervisora/Managera
- /controllers/MJGController – w tym folderze znajduje się logika pojedynczych Epucków
- /controllers/MJGDummy – w tym folderze znajduje się logika testowego Epucka, którą można wykorzystać, aby przetestować rozpoznawanie Epucka z kamery.
- /data – folder w którym są pliki pomocnicze oraz gotowe do importu modele map i elementów symulacji.
- /plugins – folder, który docelowo ma zawierać różnorakie pluginy fizyki używane przez Webots, jednak w naszym projekcie jest pusty lecz jest wymagany przez środowisko.

6.3 Korzystanie z generatora map

Docelowe działanie naszego systemu wykonywać się będzie na rzeczywistych robotach i na rzeczywistej makiecie. Aby sprawdzić jak system radzi sobie z odwzorowaniem powierzchni można skorzystać z programu obsługującego mapy. W tym celu należy z możliwie największą dokładnością narysować wygląd rzeczywistej makiety w programie 'Zarządzanie mapami.jar' załączonym do projektu na płycie. Następnie zostaną wygenerowane pliki .txt oraz .wbt. Plik tekstowy posłuży do porównania map, a plik .wbt do odwzorowania realnego świata w środowisku symulacyjnym Webots.



- ekran główny aplikacji do zarządzania mapami.

6.4 Przygotowanie i uruchomienie symulacji

Aby uruchomić symulację należy wybrać przykładową mapę z folderu /worlds bądź wygenerować ją zgodnie z instrukcją w punkcie 6.3.

Po wybraniu mapy należy ustalić liczbę epucków, które mają uczestniczyć w symulacji (1-2). W przypadku nowej mapy, epucki można zaimportować z folderu /data najnowszą wersję (np. DifferentialWheels_0.85.wbo), podobnie należy zrobić z Supervisorem (EpuckSupervisor.wbo).

Po zaimportowaniu obiektów należy ustawić ich parametry – numery poszczególnych epucków należy przekazać w polu „controller params”(ukazuje się po rozwinięciu obiektu w hierarchii obiektów na scenie) numerując je kolejno od „1”, podobnie w przypadku Supervisora, jednak tu należy wpisać liczbę epucków które są na scenie.

Za każdym razem gdy przenosimy projekt z komputera na komputer, zalecane jest usunięcie pliku Makefile oraz .class z każdego z folderów kontrolerów. Oraz ponowne przekompilowanie kontrolerów w Webotsie.

Po ustawieniu pożądanej pozycji robotów można uruchomić symulację.

6.5 Działanie symulacji

Na początku symulacji w oknie środowiska otworzą się dla każdego epucka okno kamery i okno z mapą wygenerowaną przez konkretnego robota. Dzięki temu mamy podgląd na postęp odkrywania mapy również jeśli przyspieszymy symulację.

W momencie spotkania epucków zostaje podjęta przez Managera próba złączenia map wykorzystująca punkt spotkania, co spowoduje otwarcie nowych okien z wizualizacją prób

nałożenia się map. Kiedy wynik nałożenia się map będzie wizualnie zadowalający można przerwać symulację, bądź poczekać aż zostanie spełniony warunek stopu.

6.6 Przygotowanie i uruchomienie rozwiązań w realnym środowisku

Do uruchomienia projektu na realnych robotach potrzeba:

- Epucka
- Nadajnika bluetooth
- Komputera z zainstalowanym środowiskiem Webots (na nim są wykonywane obliczenia i pełni funkcję managera)
- Makiety środowiska

W środowisku webots powinniśmy wczytać projekt tak jak w przypadku prób symulacji. Różnicą jest korzystanie z czujników realnych robotów poruszających się po makiecie.

Roboty podłączamy za pośrednictwem interfejsu Bluetooth i sprawdzamy na którym porcie COM są podłączone.

Przechodzimy do środowiska webots, w którym klikamy podwójnym kliknięciem na poszczególne roboty i wybieramy w miejsce symulacji komunikację z realnym robotami po konkretnych portach COM.

6.7 Działanie w realnym środowisku

Po wykonaniu kroków z poprzedniego punktu uruchamiamy symulację w środowisku webots, które pozwoli na automatyzację procesu komunikacji pomiędzy Supervisorem a epuckami. Dalej teoretycznie proces powinien przebiegać tak jak w symulacji, lecz niestety istnieje seria problemów niekoniecznie związanych z programowaniem, których część opisana jest w punkcie 5 – opisie przebiegu pracy.

Nie należy zapomnieć również o ustawieniu parametrów charakterystycznych dla każdego epucka – granicznych wartości rozpoznawania przeszkody, oraz prędkości kół pozwalających przez czas 1,6 sek(TIME_STEP) na obrót o 90 stopni. Opis miejsc zmian jest opisany w punkcie 4 – opisie systemu.

6.8 Porównywanie mapy zadanej z wygenerowaną przez epucka

Po uruchomieniu systemu na rzeczywistej makiecie i rzeczywistych robotach mobilnych zostanie wygenerowany plik tekstowy zawierający macierz odwzorowującą zmapowany przez epucki

teren. Po wybraniu funkcji ‘Porownaj mapy’ w programie ‘MapManager.jar’ podajemy nazwy dwóch plików:

1. *Plik z wygenerowaną mapą:* plik .txt, który został wygenerowany w punkcie 6.3
2. *Plik z mapą zrobioną przez epucki:* plik .txt, który został wygenerowany przez system po wykonaniu zadania mapowania przez roboty mobilne. Znajduje się on w folderze /controllers/MJGController

Po wybraniu opcji ‘Porownaj’ program porównuje mapy, wyświetla je obok siebie w takiej samej orientacji oraz wyświetla wartość błędu dopasowania.

7. Opracowane, odrzucone algorytmy

7. Niewykorzystane algorytmy:
   7.1. Model optymalizacji trasy
      7.1.1.Zmienne decyzyjne:

- Jazda obrót w i-tym kroku:

$$d_i = \{0,1\}$$

- Kierunek ruchu w i-tym kroku: (Prawo/lewo lub przód/tył)

$$m_i = \{1, -1\}$$

      7.1.2.Dane i zmienne pomocnicze:

- Mapa:

$$x_{rk} = \{0,1\}$$

- Orientacja w i-tym kroku (Wertykalna/horyzontalna):

$$p_i = \{1,0\}$$

- Zwrot (Góra/Dół lub Prawo/Lewo):

$$z_i = \{1, -1\}$$

- Ciąg odwiedzanych pól:

$$r = \{r_0, .., r_i, \dots, r_n\}$$
$$k = \{k_0, .., k_i, \dots, k_n\}$$

- Czasy obrotu o 90 stopni t_o i przejazdu o jedno pole t_p:

      7.1.3.Funkcja celu:

$$F(d) = \sum_{i=1}^{n} (d_i * t_o + (1 - d_i) * t_p) \rightarrow MIN$$

7.1.4.Ograniczenia:

$$r_{i+1} = r_i + p_i * z_i * (1 - d_i) * m_i$$

$$k_{i+1} = k_i + (1 - p_i) * z_i * (1 - d_i) * m_i$$

$$p_{i+1} = p_i \oplus d_i = (p_i + d_i)\ mod\ 2$$

$$z_{i+1} = p_i(z_i * m_i) + (1 - p_i)(z_i * (1 - m_i))$$

$$x_{r_i k_i} = 0, \qquad dla\ r_i \in r, k_i \in k$$

7.1.5.Dane początkowe:|

- Początkowe położenie e-Pucka:

$$r_0, k_0, p_0, z_0$$

- Cel:

$$ciągi: r_n, k_n$$

7.1.6.Wnioski:

Zgodnie z zamieszczonymi w części właściwiej wnioskami algorytm ten został odrzucony z powodu złożoności jego rozwiązywania.

7.2. Algorytm nagród i kar (zaniechany):

7.2.1.Podejmowana decyzja:

- Kierunek kolejnego kroku

$$kierunek = \{0, 1, 2, 3\}$$

Gdzie zgodnie z oznaczeniem przyjętym na rys. 2:

0 – północ,
1 – wschód,
2 – południe,
3 – zachód.

7.2.2.Dane i zmienne pomocnicze:

- Mapa:

$$x_{rk} = \{0,1\}$$

- Orientacja w i-tym kroku (Wertykalna/horyzontalna):

$$p_i = \{1,0\}$$

- Zwrot (Góra/Dół lub Prawo/Lewo):

$$z_i = \{1, -1\}$$

- Cel:

$$x_c = x_{r_c k_c}$$

- Ciąg odwiedzanych pól:

$$r = \{r_0, .., r_i, \dots, r_n\}$$
$$k = \{k_0, .., k_i, \dots, k_n\}$$

- Odległości w poszczególnych osiach;

$$dy = k_c - k$$

$$dx = r_c - r$$

7.2.3.Funkcja celu:

$$kierunek = Max_{ind}(\boldsymbol{reward} * \boldsymbol{direction})$$

Gdzie zmienna ***reward*** jest wektorem o parametrach:

$$reward = \begin{bmatrix} \sin(\frac{\pi}{2} * \frac{dy}{m}) \\ \sin(\frac{\pi}{2} * \frac{dx}{n}) \\ -\sin(\frac{\pi}{2} * \frac{dy}{m}) \\ -\sin(\frac{\pi}{2} * \frac{dx}{n}) \end{bmatrix}$$

Więc dowolny element macierzy mieścił się w przedziale [-1,1]. Z kolei zmienna ***direction*** *była* wektorem transponowanym wag dla poszczególnych kierunków, przy czym dla jazdy prosto 0,65, obrotu w prawo i w lewo 0,15 i kierunku przeciwnego do orientacji e – pucka 0,0.

Przykładowa postać dla wartości wektora orientacji:[0] {v=true, d=true}:

$$direction = [0.65 \quad 0.15 \quad 0.0 \quad 0.15]$$

7.2.4.Powód odrzucenia:

Zgodnie z wnioskiem postawionym w części właściwiej wrażliwość na przeszkody odrzuciła to rozwiązanie jako obowiązujące.

8. Bibliografia

[i]Durrant-Whyte, H; *A Solution to the Simultaneous Localization and Map Building (SLAM) Problem* IEEE TRANSACTIONS ON ROBOTICS AND AUTOMATION, VOL. 17, NO. 3, JUNE 2001

[ii] Randall Smith, Matthew Self, Peter Cheesemans *Estimating Uncertain Spatial Relationships in Robotics*

[iii] Taylor, R. H. 1976. *A Synthesis of Manipulator Control Programs from Task-Level Specifications*. AIM-282. Stanford, Calif.: Stanford University Artificial Intelligence Laboratory

[iv] Brooks, R. A. 1982. *Symbolic Error Analysis and Robot Planning*. Int. J. Robotics Res. 1(4):29-68

[v] Nikolas Trawny, Anastasios I. Mourikis, Stergios I. Roumeliotis, Andrew E. Johnson and James F. Montgomery *Vision-aided inertial navigation for pin-point landing using observations of mapped landmarks* Special Issue: Special Issue on Space Robotics, Part III
Volume 24, Issue 5, pages 357–378, May 2007

[vi] Ryan Eustice, Hanumant Singh, John Leonard, Matthew Walter, Robert Ballard *Visually Navigating the RMS Titanic with SLAM Information Filters*,

[vii] Fernando A Auat Cheein, Natalia Lopez, Carlos M Soria, Fernando A di Sciascio, Fernando Lobo Pereira3, Ricardo Carelli *SLAM algorithm applied to robotics assistance for navigation in unknown environments* JOURNAL OF NEUROENGINEERING AND REHABILITATION

[viii] Lars A. A. Andersson, *Multi-robot Information Fusion*, Linkoping Studies in Science and Technology. Dissertations. No. 1209

[ix] M. Di Marco A. Garulli A. Giannitrapani A. Vicino, *DYNAMIC ROBOT LOCALIZATION AND MAPPING USING UNCERTAINTY SETS*

[x] Jun S. Liu & Rong Chen, *Sequential Monte Carlo Methods for Dynamic Systems*, Journal of the American Statistical Association Volume 93, Issue 443, 1998

[xi] Nathaniel Fairfield, George Kantor, David Wettergreen *Real-Time SLAM with Octree Evidence Grids for Exploration in Underwater Tunnels*

[xii] Andrew Howard, *Multi-robot Simultaneous Localization and Mapping using Particle Filters*, The International Journal of Robotics Research December 2006 vol. 25 no. 12 1243-1256

[xiii] Taylor. G. ; Kleeman. L, Diosi*, A Interactive SLAM using Laser and Advanced Sonar*, Robotics and Automation, 2005. ICRA 2005

[xiv] Choi, M., Sakthivel, R., and Chung, W.K. (2007). *Neural network-aided extended kalman filter for slam problem*. In ICRA, 1686–1690.

[xv] Adrien Angeli, St´ephane Doncieux, Jean-Arcady Meyer, David Filliat. *Incremental vision-based topological SLAM*. IEEE/RSJ International Conference on Intelligent Robots and Systems, 2008. IROS 2008, Sep 2008, Nice, France. pp.1031 - 1036, <10.1109/IROS.2008.4650675>. <hal-00647374>

[xvi] Howie Choset, Keiji Nagatani *Topological simultaneous localization and mapping (SLAM): toward exact localization without explicit localization*, Robotics and Automation, IEEE Transactions on

[xvii] Sławomir Jeżewski, Adam Wulkiewicz, *Koncepcja przestrzeni percepcyjnej robota mobilnego* automatyka 2009 TOM 13 zeszyt 3

[xviii] M. Rostowskiej, M. Topolskiego i P. Skrzyprzyńczkiego, *A modular mobile robot for multi-robot applications*

[xix] Guyot, L., Heiniger, N., Michel, O. and Rohrer, F. Teaching robotics with an open curriculum based on the e-puck robot, simulations and competitions. Proceedings of the 2nd International Conference on Robotics in Education, RiE 2011, September 15-16 Vienna, Austria. Roland Stelzer and Karim Jafarmadar (editors). ISBN: 978-3-200-02273-7.

[xx] Álvaro Gutiérrez, Alexandre Campo, Marco Dorigo, Jesus Donate, Félix Monasterio-Huelin, Luis Magdalena *Open e-puck range & bearing miniaturized board for local communication in swarm robotics* Robotics and Automation, 2009. ICRA'09. IEEE International Conference

[xxi] M. Jacobsson, S. Ljungblad, J. Bodin, J. Knurek, and L.E. Holmquist, *GlowBots: robots that evolve relationships. In International Conference on Computer Graphics and Interactive Techniques*. ACM New York, NY, USA, 2007.