Weryfikacja własności ruchowych układów nieholonomicznych na przykładzie robota mobilnego Pioneer 3-DX z elementami ROSa*

Joanna Ratajczak, Katarzyna Zadarnowska†
Laboratorium Robotów Autonomicznych
Wydział Elektroniki
Politechnika Wrocławska

Spis treści

1 Cel ćwiczenia 2

2 Wymagania wstępne 2

3 Zasady bezpieczeństwa 2

4 Wprowadzenie do systemu operacyjnego ROS 3
- 4.1 Pakiety . . . 3
- 4.2 Węzły . . . 4
- 4.3 Tematy i wiadomości . . . 4
- 4.4 Usługi . . . 5
- 4.5 W razie problemów ☺ . . . 6

5 Dostępne komendy 6

6 Przykładowy program 8

7 Symulator 9

8 Zadania do wykonania 9

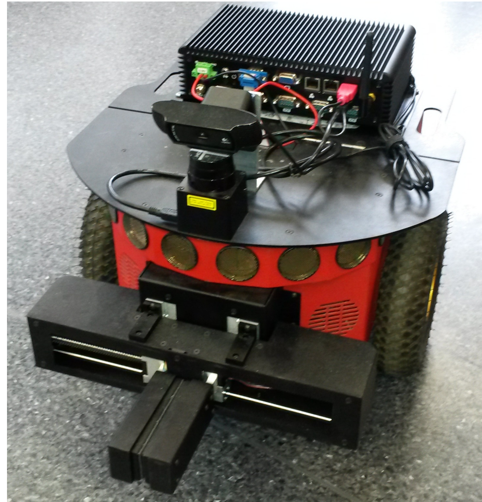
9 Sprawozdanie 11

A Składnia języka Phyton 12
- A.1 Instrukcja warunkowa `if` . . . 12
- A.2 Pętla `while` . . . 13
- A.3 Pętla `for` . . . 13

---

*Ćwiczenie laboratoryjne przeznaczone do realizacji w ramach kursu Robotyka (2) – data ostatniej modyfikacji dokumentu 12 marca 2018

†Katedra Cybernetyki i Robotyki



Rysunek 1: Pioneer 3-DX

1 Cel ćwiczenia

Celem ćwiczenia jest weryfikacja własności ruchowych układów nieholonomicznych, zbadanie związku pomiędzy generatorami układu a kierunkami ruchu. Ponadto w trakcie ćwiczenia będą implementowane proste programy na robocie współpracujące z systemem odometrii i układem sonarów.

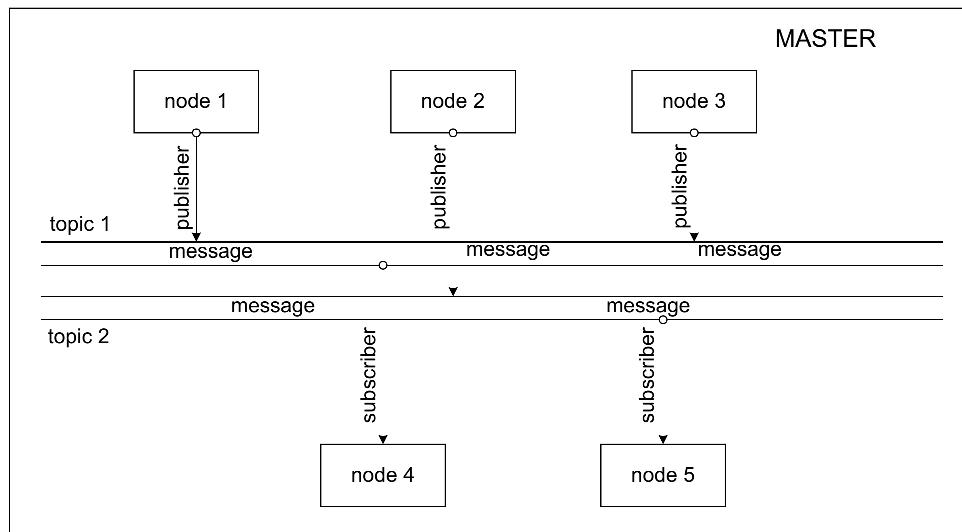
2 Wymagania wstępne

Przed przystąpieniem do realizacji ćwiczenia należy:

1. powtórzyć wiadomości z zakresu ograniczeń nakładanych na układy robotyczne, modelowania układów nieholonomicznych oraz metod sprawdzania sterowalności takich układów,
2. znać podstawy programowania w języku Python,
3. powtórzyć wiadomości z zakresu podstaw pracy z Linuksem,
4. zapoznać się z Instrukcją bezpieczeństwa i higieny pracy w Laboratorium Robotów Autonomicznych [1],
5. zapoznać się z Instrukcją bezpieczeństwa i higieny pracy przy obsłudze robotów Pioneer P3-DX [2],

3 Zasady bezpieczeństwa

Robot Pioneer 3DX jest robotem mobilnym dwukołowym (rys. 1). Przed przystąpieniem do pracy z robotem należy uruchomić program pozwalający na awaryjne zatrzymanie robota (`rosrun estop estop _robot:=PIONIERx`, gdzie `x` jest numerem przydzielonego do zajęć robota). Ponieważ przybliżona waga robota wynosi $m = 9[kg]$ zaleca się pracę na niskich prędkościach, tak aby możliwa była reakcja na nieprzewidziane sytuacje. Należy pilnować, aby robot nie wyjeżdżał poza obszar stanowiska i nie dopuszczać, aby wjeżdżał w przeszkody (krzesła, stoliki, ściany). W przypadkach awaryjnych należy wcisnąć



Rysunek 2: Schemat projektu na platformie ROS

„wyłącznik bezpieczeństwa" na komputerze, co spowoduje włączenie hamulców. Po przeniesieniu robota w bezpieczne miejsce oraz wprowadzeniu poprawek do programu należy ponownie kliknąć przycisk „wyłącznik bezpieczeństwa". Podczas wykonywania programu robota można również zatrzymać wywołując komendę `SetSpeed(0,0,1)`.

4 Wprowadzenie do systemu operacyjnego ROS

ROS (Robot Operating System) to platforma programistyczna do tworzenia oprogramowania oraz sterowania robotów. Stanowi ona zbiór bibliotek pozwalających na sterowanie i symulację pracy robota. W szczególności, ROS zawiera podstawowe procesy systemowe obsługujące urządzenia sprzętowe robota, sterowanie niskopoziomowe, implementacje wykonywania typowych funkcji, komunikację międzywątkową oraz zarządzanie pakietami. Ideę środowiska ROS prezentuje rysunek 2.

Oprogramowanie robotów tworzy się w postaci zbioru małych, działających współbieżnie i w dużej mierze niezależnych od siebie programów, nazywanych węzłami (ang. *nodes*). W celu ułatwienia procesu wymiany danych pomiędzy tymi węzłami w systemie ROS jest wykorzystywany węzeł pełniący funkcję węzła nadrzędnego (*ROS Master*), który dba o poprawność komunikacji zachodzącej między pozostałymi węzłami systemu. Węzeł ten uruchamia się wydając polecenie `roscore`. Węzły komunikują się między sobą poprzez wysyłanie wiadomości (ang. *messages*). Wiadomości zorganizowane są w tematach (ang. *topic*). Węzły, które chcą dzielić informacje (ang. *publishers*) publikują wiadomości w obrębie odpowiedniego tematu, natomiast węzły odbierające wiadomości, tzw. subskrybenci (ang. *subscribers*) subskrybują wiadomości. Węzeł nadrzędny powinien działać przez cały czas trwania pracy w systemie ROS, należy więc uruchmić go w oddzielnym terminalu (polecenie `roscore`) i pozostawić uruchomionym na czas pracy. Zatrzymanie węzła nadrzędnego następuje po wysyłaniu sygnał SIGINT (`Ctrl-C`).

4.1 Pakiety

Oprogramowanie systemu ROS stanowi zbiór pakietów. Pakiety z kolei są zbiorem plików realizujących określony cel. Pakiety posiadają swoją dokumentację (plik package.xml).

Poniżej umieszczono listę przydatnych poleceń systemu ROS dotyczących pakietów:

- lista wszystkich dostępnych pakietów zdefiniowanych w systemie ROS
  `rospack list`
- znajdowanie katalogu zawierającego dany pakiet
  `rospack find` *`package-name`*
- przeglądanie zawartości katalogu danego pakietu
  `rosls` *`package-name`*
- przejście z bieżącego kaalogu do katalogu danego pakietu
  `roscd` *`package-name`*

4.2 Węzły

Węzły* to wykonywalne instancje programów systemu ROS. Wyróżniamy następujący zestaw poleceń dotyczących węzłów:

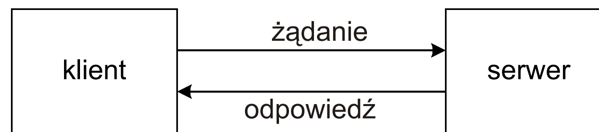
- tworzenie węzła (uruchamianie programu systemu ROS)
  `rosrun` *`package-name executable-name`*
- tworzenie węzła wraz z nadaniem mu nazwy
  `rosrun` *`package-name executable-name`* `__name:=`*`node-name`*
- wyświetlenie listy uruchomionych węzłów
  `rosnode list`
- uzyskiwanie informacji na temat węzła
  `rosnode info` *`node-name`*
- zatrzymanie pracy (zabicie) węzła
  `rosnode kill` *`node-name`*
- usunięcie z listy zatrzymanego (zabitego) węzła
  `rosnode cleanup`

4.3 Tematy i wiadomości

System ROS udostępnia następujący zbiór poleceń dotyczących tematów† i wiadomości przesyłanych w obrębie tematów:

- lista aktywnych tematów
  `rostopic list`
- wyświetlenie wiadomości przesyłanych aktualnie w obrębie danego tematu
  `rostopic echo` *`topic-name`*
- mierzenie częstotliwości publikowania wiadomości (wiadomość/sec)
  `rostopic hz` *`topic-name`*
- mierzenie przepustowości wiadomości (bajt/sec)
  `rostopic bw` *`topic-name`*

*http://wiki.ros.org/ROS/Tutorials/UnderstandingNodes
†http://wiki.ros.org/ROS/Tutorials/UnderstandingTopics



Rysunek 3: Schemat komunikacji w obrębie klient–serwer

- uzyskiwanie informacji na temat danego tematu
  `rostopic info` *`topic-name`*
- uzyskiwanie szczegółowych informacji na temat przesyłanych wiadomości
  `rosmsg show` *`message-type-name`*
- publikowanie wiadomości
  `rostopic pub -r` *`rate-in-hz topic-name message-type message-content`*

4.4 Usługi

Usługi[‡] (ang. *services*) oferują komunikację dwukierunkową: węzeł wysyła informację do innego węzła i oczekuje na odpowiedź. Rysunek 3 obrazuje przepływ informacji w ramach usług. Zatem węzeł–klient wysyła żądanie/prośbę (ang. *request*) do węzła–serwera, ten realizuje żądanie i odsyła odpowiedź (ang. *response*) do węzła–klienta. Usługi, podobnie jak wiadomości, posiadają konkretny typ danych, jednak podzielony jest on na dwie części: typ danych żądania i typ danych odpowiedzi.

Korzystanie z usług z linii poleceń wygląda następująco:

- lista aktywnych usług
  `rosservice list`
- lista usług oferowanych przez dany węzeł
  `rosnode info` *`node-name`*
- odnajdywanie węzła oferującego daną usługę
  `rosservice node` *`service-name`*
- odczytywanie typu danych konkretnej usługi
  `rosservice type` *`service-name`*
- szczegółowe informacje na temat typu danych usługi (osobno dla żądania i odpowiedzi)
  `rossrv show` *`service-data-type-name`*
- wywołanie usługi
  `rosservice call` *`service-name request-content`*
  przy czym należy pamiętać, że pole *request-content* powinno zawierać listę wartości uzupełniających poszczególne pola żądania

Programowe wywoływanie usług wygląda następująco:

Program – klient

Na początku należy dołączyć odpowiedni plik nagłowkowy definiujący typ danych żądania i odpowiedzi usługi

[‡]http://wiki.ros.org/ROS/Tutorials/WritingServiceClient(c++)

```
#include<package_name/type_name.h>
```

Następnie, po zainicjowaniu węzła (ros::init) i utworzeniu obiektu węzła (ros::NodeHandle), należy utworzyć obiekt klienta, który zajmie się wywołaniem usługi

```
ros::ServiceClient client = node_handle.serviceClient<service_type>
(service_name);
```

przy czym *node_handle* to nazwa obiektu definiującego węzeł, *service_type* to nazwa obiektu usługi zadeklarowanej wcześniej w pliku nagłówkowym, *service_name* to nazwa usługi wywoływanej. Obiekt klienta zwraca szczegółowe informacje na temat wywoływanej usługi.
W kolejnym kroku tworzony jest obiekt żądania, który wyśle dane do serwera i obiekt odpowiedzi klas

```
package_name::service_type::Request
package_name::service_type::Response
```

W ostatnim kroku należy uzupełnić pola obiektu żądania i wywołać usługę

```
bool success = service_client.call(request,response);
```

Powyższa metoda lokalizuje węzeł serwera, przesyła dane żądania, czeka na odpowiedź i zwraca dane odpowiedzi w odpowiednim obiekcie (`Response`). Metoda `call` zwraca wartość typu boolean mówiącą o tym, czy wywołanie usługi zakończyło się powodzeniem. Jeśli tak, mamy dostęp do danych odpowiedzi.

Program – serwer

Sposób tworzenia programu–serwera jest opisany w rozdz. 8.4 [3]. Przykładowe programy klienta i serwera wraz z ich szczegółowym opisem można odnaleźć w rozdz. 2 [4].

4.5 W razie problemów ☺

Jeśli ROS zachowuje się inaczej niż oczekujemy można skorzystać z polecenia

```
roswtf
```

które sprawdza poprawność wywołań (w tym zmienne środowiskowe, zainstalowane pliki, uruchomione węzły i inne). W celu zapoznania się z koncepcją programowania w systemie ROS, warto wykorzystać symulator turtlesim [5].

5 Dostępne komendy

W celu realizacji komunikacji z robotem wykorzystany zostanie pakiet RosAria [7], którego podstawowym zadaniem jest zapewnienie komunikacji z robotami mobilnymi (m.in. Pioneerami) i ich wewnętrznymi urządzeniami (np. sonarami) w systemie ROS. RosAria wykorzystuje bibliotekę ARIA napisaną przez *Omron Adept Mobile Robots*. To ARIA odpowiedzialna jest za dostarczenie interfejsu z robotem mobilnym oraz jego czujnikami i innymi urządzeniami wbudowanymi, a zatem pozwala odbierać dane z robota oraz nim sterować. RosAria natomiast zapewnia kompatybilność owego interfejsu z platformą ROS. Informacje z robota, jak również sterowania (prędkością czy też przyspieszeniem) implementowane są w postaci węzłów (na temat tworzenia, kompilacji oraz uruchamiania

węzłów publikujących oraz subskrybujących można poczytać w podrozdziałach 3.2 – 3.4 pozycji [3] lub w [5]).Węzły publikują oraz subskrybują wiadomości w obrębie odpowiednich tematów. I tak, w ramach tematu `ROSARIA/cmd_vel` zadawane/subskrybowane są prędkości robota. Typ takiej wiadomości `geometry_msgs/Twist` składa się z dwóch pól opisujących prędkość liniową $[m/s]$ oraz prędkość kątową $[rad/s]$ robota

```
geometry_msgs/Vector3 linear
float64 x
float64 y
float64 z
geometry_msgs/Vector3 angular
float64 x
float64 y
float64 z
```

Każda z prędkości jest więc wektorem (x, y, z) (elementy typu *double*) opisującym jej poszczególne współrzędne.

Pozycja robota odczytywana jest w ramach tematu publikującego `ROSARIA/pose`. Typ wiadomości to `nav_msgs/Odometry`. Pozycja jest obliczana w ARII na podstawie odometrii kół (informacje z enkoderów i/lub żyroskopu). Tak obliczona pozycja kątowa każdego z kół pozwala na obliczenie prędkości kół $\omega_P[m/s]$ i $\omega_L[m/s]$, a te z kolei pozwalają na obliczenie prędkości postępowej $v[m/s]$ i kątowej $\omega[rad/s]$ robota.

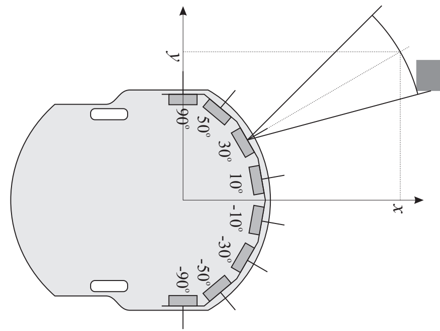
Informacje z sonarów są publikowane w temacie `ROSARIA/sonar`, a typ wiadomości opisuje się za pomocą talicy dwuwymiarowej `sensor_msgs/PointCloud`. Do sterowania chwytakiem wykorzystuje się usługi `RosAria/gripper_open`, `RosAria/gripper_close`, `RosAria/gripper_up` i `RosAria/gripper_down`. W [7] można znaleźć informacje na temat pozostałych tematów oferowanych w ramach pakietu RosAria.

W tym ćwiczeniu programowanie robota Pioneer odbywa się w języku Python. Do komunikacji z robotem laboratoryjnym służy moduł `RosAriaDriver_package` [§] dostępny w laboratorium. Moduł został napisany na bazie pakietu RosAria tak, by realizował zadania postawione w ćwiczeniu. Niemniej, należy pamiętać, że ma on strukturę analogiczną do tej w RosArii. Aby nawiązać komunikację konieczne jest podanie nazwy robota w programie. Poniżej przedstawiono dostępne komendy umożliwiające pracę z robotem mobilnym

`SetSpeed(linear,angular,time)` – ustawienie prędkości liniowej [m/s] (linear) i kątowej [rad/s] (angular) robota, `time` określa czas trwania ruchu. Dla `time`=0 funkcja zadaje podane prędkości ruchu na czas nieokreślony (do następnej jej zmiany)[¶], jednakże ze względu na zastosowane w robotach systemy bezpieczeństwa (watchdog), tak wywołana komenda nie spowoduje, że robot będzie poruszał się z zadaną prędkością „w nieskończoność”,
`SetSpeedLR(left,right,time)` – ustawienie prędkości kół [m/s], `time` określa czas trwania ruchu (zdefiniowany jak w przypadku `SetSpeed`),
`GoTo(n)` – powoduje jazdę robota przez `n` metrów,
`GripperOpen()` – otwiera chwytak,
`GripperClose()` – zamyka chwytak,
`GripperUp()` – podnosi chwytak,
`GripperDown()` – opuszcza chwytak,
`ResetPose()` – zeruje pozycję robota,

[§] Damian Barański. Dokumentacja klasy RosAriaDriver. http://panamint.ict.pwr.wroc.pl/~dbaransk/rosaria_drive/dox/html/index.html, 2015.

[¶] Należy zauważyć, że zadanie niezerowego czasu ruchu wstrzymuje wykonywanie programu na podany okres (jak po użyciu funkcji `sleep(#)` z modułu *time*).



Rysunek 4: Rozkład sonarów na robocie

`GetPose()` – zwraca pozycję robota,
`ReadSonar()` – odczyt pomiaru odległości z sonarów.

Odczyt czujników zwracany jest jako tablica 8–elementowa indeksowana od 0 zawierająca kolejno `[kąt(−90°), kąt(−50°), kąt(−30°), kąt(−10°), kąt(10°), kąt(30°), kąt(50°), kąt(90°)]`, w której każdy element zawiera współrzędne (x, y) przeszkód w układzie lokalnym robota (patrz rys. 4).

A zatem polecenia `SetSpeed()` oraz `SetSpeedLR()` napisano na bazie `ROSARIA/cmd_vel`, `GetPose()` i `Reset(Pose)` bazują na `ROSARIA/pose`, natomiast `ReadSonar()` napisano w oparciu o `ROSARIA/sonar`.

6 Przykładowy program

```
#!/usr/bin/env python
#-*- coding: latin2 -*-

from drive import RosAriaDriver

def fun(p):
  p.SetSpeed(0.05,0,2)        #ustawienie prędkości i czasu ruchu
  p.SetSpeed(0,0,0)           #zatrzymanie robota
  print "Koniec"              #wypisanie w terminalu

p=RosAriaDriver('/PIONIERx')  #wprowadzenie właściwej nazwy robota
try:
   fun(p)
   p.SetSpeed(0,0,0)
finally:
   p.SetSpeed(0,0,0)          #w sytuacji awaryjnej
```

Aby uruchomić program[‖] należy w konsoli wpisać polecenie

`python NazwaProgramu.py`

[‖]Przykładowy program można skopiować z `/opt/ROS_Lab1_5/python/`

7 Symulator

Przed wykonaniem programu na robocie należy przetestować go na dostępnym w laboratorium symulatorze MobileSim. W tym celu należy

1. uruchomić symulator MobileSim z mapą laboratorium**
2. wyeksportować przestrzeń nazw poleceniem
   `export ROS_NAMESPACE='unikatowa_nazwa'`
3. uruchomić RosArie komendą
   `rosrun rosaria RosAria`
4. w programie wprowadzić właściwą nazwę
   `p=RosAriaDriver('/unikatowa_nazwa')`
5. wykonać program poleceniem
   `python nazwa_programu.py`

8 Zadania do wykonania

1. Przeprowadź robota w taki sposób, aby zakreślił on kwadrat o zadanej długości boku b:

   (a) sterując prędkością liniową i kątową (`SetSpeed`),

   (b) sterując prędkością w kołach (`SetSpeedLR`).

   Parametry geometryczne robota są następujące: rozstaw kół $0.326[m]$, średnica koła $0.195[m]$.

2. Zakreśl robotem ósemkę w górnej pętli sterując prędkościami liniową i kątową, a w dolnej prędkościami w kole lewym i prawym.

3. Wiedząc, że na ruch robota nałożone jest ograniczenie w postaci braku poślizgu bocznego, a zmienne zadaniowe są następujące $q = (x, y, \theta)^T \in R^3$, bezdryfowy układ sterowania może przybrać formę

$$\dot{q} = \begin{bmatrix} \cos(\theta) & \cos(\theta) \\ \sin(\theta) & \sin(\theta) \\ -\frac{1}{d} & \frac{1}{d} \end{bmatrix} \begin{pmatrix} \eta_1 \\ \eta_2 \end{pmatrix}, \tag{1}$$

   gdzie d jest długością odcinka łączącego środek osi kół z punktem styku koła do podłoża. Jaki sens mają elementy η_1 i η_2? Zweryfikuj swoje przypuszczenia na robocie.

4. Jak powinna wyglądać macierz $G(q)$ układu sterowania

$$\dot{q} = G(q) \begin{pmatrix} u_1 \\ u_2 \end{pmatrix} \tag{2}$$

   jeśli przyjmiemy, że u_1 jest prędkością liniową i u_2 jest prędkością kątową robota?

** `/usr/local/MobileSim/L15.map`

5. Jakie relacje występują pomiędzy prędkościami w kołach a prędkościami liniową i kątową w tym robocie? Wiedząc, że $d = 0.163[m]$ napisz program weryfikujący eksperymentalnie otrzymane wyliczenia.

6. Dla robota poruszającego się bez poślizgu wzdłużnego i bocznego, opisanego we współrzędnych uogólnionych $q = (x, y, \theta, \phi_1, \phi_2)^T$, energia kinetyczna robota jest dana

$$E_k(q, \dot{q}) = \frac{1}{2}m_c(\dot{x}^2 + \dot{y}^2) + \frac{1}{2}I_c\dot{\theta}^2 + \frac{1}{2}I_k(\dot{\phi_1}^2 + \dot{\phi_2}^2) = \frac{1}{2}\dot{q}^T Q\dot{q},$$

gdzie m_c jest masą całkowitą robota, I_z jest całkowitym momentem bezwładności a I_k jest momentem bezwładności koła. Zakładając, że macierz sterowań przyjmuje postać

$$B = \begin{bmatrix} 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}^T, \tag{3}$$

oblicz model dynamiki robota.

7. Włączając jedną wybraną składową sterowań a wyłączając pozostałe, np. $u_1 = 1$ i $u_2 = 0$ powodujemy ruch robota wzdłuż trajektorii pola wektorowego – np. generatora g_1. W wyniku czego możemy powiedzieć, że stosując odpowiednie stałe sterowania uzyskujemy możliwość poruszania się po przestrzeni rozpiętej przez generatory układu $P = \text{span}_R\{g_1, g_2\}$. Zbadaj eksperymentalnie jakie kierunki ruchu otrzymujemy bezpośrednio z generatorów. Jakie jeszcze inne kierunki możemy wygenerować przy pomocy sterowań dla układu (1), a jakie dla układu (2)?

8. Formuła Campbella–Bakera–Hausdorffa–Dynkina mówi o tym, że po odpowiednim złożeniu czterech segmentów sterowań uzyskujemy dla małych czasów t

$$\begin{aligned} \exp(t^2[X, Y] + o(t^3)) &= \exp(tX)\exp(tY)\exp(-tX)\exp(-tY) = \\ &\exp(tX)\exp(tY)\exp(tX(-1))\exp(tY(-1)). \end{aligned}$$

Co oznacza, że jeśli włączymy sterowanie działające na generator X przez czas t, następnie przez taki sam czas włączymy sterowanie generatorem Y, później sterowanie generatorem $-X$ i na końcu $-Y$, to otrzymamy sterowanie generujące ruch wzdłuż pola wektorowego $[X, Y]$ w czasie t^2.

Wiedząc, że kierunek otrzymany poprzez nawias Liego możemy zrealizować za pomocą sterowań posługując się formułą Campbella–Bakera–Hausdorffa–Dynkina (formułą CBHD) przygotuj eksperyment mający na celu zweryfikowanie poprawności, obliczonego przez siebie w poprzednim zadaniu, nowego (trzeciego) kierunku ruchu robota. Sprawdź, że kierunek $\exp(t^2[X, Y] + o(t^3))$ uzyskasz również poprzez cykliczną zmianę elementów składowych, i tak

$$\begin{aligned} \exp(t^2[X, Y] + o(t^3)) &= \exp(tX)\exp(tY)\exp(-tX)\exp(-tY) = \\ &\exp(tY)\exp(-tX)\exp(-tY)\exp(tX) = \\ &\exp(-tX)\exp(-tY)\exp(tX)\exp(tY) = \\ &\exp(-tY)\exp(tX)\exp(tY)\exp(-tX) \end{aligned}$$

Co otrzymamy jeśli wygenerujemy ruch $\exp(tX)\exp(tY)\exp(-tY)\exp(-tX)$?

9. Zaobserwuj wpływ czasu sterowania na otrzymany ruch w kierunku nawiasu Liego. Wykonaj na robocie manewr parkowania równoległego.

10. Wyzeruj położenie robota korzystające z odometrii. Następnie wiedząc, że średnica koła robota jest równa $2r = 0.195[m]$ przejedź robotem dystans zadany przez prowadzącego w linii prostej i ponownie odczytaj położenie robota.

11. Postaw robota w oznaczonym miejscu na podłodze i korzystając z odometrii odczytaj położenie robota. Następnie przeprowadź robota po ścieżce kołowej (zgodnie z ruchem wskazówek zegara) tak, aby wrócił do punktu początkowego i ponownie odczytaj wartości określające położenie. Powtórz ćwiczenie tym razem jeżdżąc w kierunku przeciwnym do ruchu wskazówek zegara. Jakie są współrzędne robota po wykonaniu ruchu? Czy kierunek jazdy miał wpływ na odczyty?

12. Powtórz poprzednie ćwiczenie tym razem poruszając robotem po krzywej innej niż okrąg (możliwie jak najbardziej łamanej)[††]. Czy wyniki są jednakowe w przypadku jazdy po gładkiej i niegładkiej krzywej?

13. Odczytaj wskazania wybranego sonaru dla różnych odległości od przeszkody i na ich podstawie sporządź charakterystykę czujnika.

14. [‡‡] Napisz program, który na podstawie odczytów z sonarów będzie zatrzymywał robota przed przeszkodą.

15. [‡‡] Zaproponuj algorytm do bezkolizyjnej jazdy robotem w środowisku z przeszkodami.

9 Sprawozdanie

Sprawozdanie z przebiegu ćwiczenia powinno zawierać:

- Imię i nazwisko autora, numer i termin grupy, skład grupy, temat ćwiczenia, datę wykonania ćwiczenia.
- Cel ćwiczenia.
- Opis przebiegu i efektu wykonania realizowanych zadań.
- Wnioski końcowe.

Literatura

[1] M. Janiak. *Instrukcja bezppieczeństwa i higieny pracy w Laboratorium Robotów Autonomicznych L1.5*. Katedra Cybernetyki i Robotyki, Politechnika Wrocławska, Instrukcja Laboratorium Robotów Autonomicznych L1.5, 2015.

[2] Mariusz Janiak, Aleksandra Grzelak. *Instrukcja bezpieczeństwa i higieny pracy przy obsłudze robotów Pionneer P3-DX*. Katedra Cybernetyki i Robotyki, Politechnika Wrocławska, Instrukcja Laboratorium Robotów Autonomicznych L1.5, 2015.

[††] W celu wykonania tego ćwiczenia warto posłużyć się `rqt`, które umożliwia sterowanie robotem bezpośrednio z klawiatury. Należy uruchomić `rqt`, następnie z menu wybrać `Plugins->Robot tools->Robot steering`. Proszę upewnić się, że prędkości będą publikowane w odpowiednim temacie `/PIONIERx/RosAria/cmd_vel`

[‡‡] Zadanie nie jest przeznaczone do realizacji przez osoby uczęszczające na kurs Roboty manipulacyjne i mobilne.

[3] J. M. O'Kane. *A Gentle Introduction to ROS*. Independently published, 2013. Available at http://www.cse.sc.edu/~jokane/agitr/.

[4] A. Martinez i E. Fernandez. *Learning ROS for Robotics Programming. A practical, instructive, and comprehensive guide to introduce yourself to ROS, the top - notch, leading robotics framework*, 2013. Available at http://www.cse.sc.edu/~jokane/agitr/.

[5] K. Zadarnowska. *Wprowadzenie do systemu operacyjnego ROS: symulator turtlesim.* Katedra Cybernetyki i Robotyki, Politechnika Wrocławska, Instrukcja Laboratorium Robotów Autonomicznych L1.5, 2015.

[6] http://wiki.ros.org

[7] http://wiki.ros.org/ROSARIA

A Składnia języka Phyton

Podstawowym złożonym typem danych w języku Python jest lista. Lista jest zapisywana jako wartości rozdzielone przecinkami i ujęta w nawiasy kwadratowe. Kolejne elementy listy nie muszą być tego samego typu, np. `a=['cos', 100, True]`.

Dostęp do elementu listy uzyskuje się stosując operator indeksowania, np. `a[1]`, a pierwszy element posiada indeks 0. Użycie indeksów ujemnych powoduje zwrócenie odpowiedniej wartości elementu listy licząc od końca. Indywidualne elementy listy można dowolnie zmieniać, np. `a[1]=a[1]+2`. Dla list o polach dwuelementowych np. `a=[[x_1,y_1], [x_2,y_2],[x_3,y_3]]` dostęp do elementu o wartości `x_2` uzyskujemy poprzez `a[1][0]`.

A.1 Instrukcja warunkowa `if`

Przykładowa instrukcja warunkowa wygląda następująco

```
if x > 0:
        return 1
elif x==0:
        return 0
else:
        return -1
```

W Pythonie każda niezerowa wartość całkowita, każda niepusta linia lub ciąg znaków ma wartość logicznej prawdy. Standardowe operatory porównania są takie same jak w języku `C`: `<` mniejszy, `>` większy, `==` równy, `!=` różny, `<=` mniejszy lub równy oraz `>=` większy lub równy. Grupowanie wyrażeń odbywa się poprzez wcięcia. Każda linia tej samej grupy musi być wcięta o tą samą liczbę znaków. Zagnieżdżenie grup uzyskuje się przez zagnieżdżenie wcięć.

```
if x > 0:
        return 1
        print "większa od 0"
elif x==0:
        return 0
        print "równa 0"
else:
```

```
        return -1
        print "mniejsza od 0"
```

A.2 Pętla `while`

W Pythonie mamy też kilka rodzajów pętli. Pierwsza z nich to pętla `while`.

```
licznik = 0
wartosc = 15

while licznik <= wartosc:
        print licznik
        licznik += 1
```

Pętla ta będzie wypisywała wartość licznika tak długo jak długo warunek $licznik \leqslant wartosc$ będzie spełniony.

A.3 Pętla `for`

Kolejną pętlą jest pętla `for`.

```
for i in range(10):
        print i
```

W tym wypadku wypisuje ona kolejne liczby od 0 do 9. Komenda `range(10)` generuje listę 10 wartości $[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]$. Komenda `range(5,10)` definiuje ciąg zaczynający się od 5, a `range(0,10,2)` generuje ciąg ze stałym krokiem równym 2. `For` może też iterować inne typy danych takie jak napisy, listy, itp.

```
str = "Wyraz"
for i in str:
        print i

str = ["abc", "def"]
for i in str:
        print i
```