Planowanie Ruchu Robotów - projekt

Algorytm CBHD

Zespół:
Jędrzej Stolarz 20277
Dawid Chechelski, 197002
Artur Błażejewski, 200541
Michał Burdka 195370

Semestr letni 2016/2017

Politechnika Wrocławska
Wydział Elektroniki
Automatyka i Robotyka
ARR

1 Wstęp teoretyczny

Obiektem zainteresowania są roboty mobilne o dwóch sterowaniach (w ogólniejszym przypadku mogłoby być ich więcej), których dynamikę opisuje równanie stanu właściwe dla bezdryfowych układów sterowania.

$$\dot{q} = g_1(q)u_1 + g_2(q)u_2 = G(q)u$$

Możliwe jest w pewnym przybliżeniu otrzymanie przemieszczania równoważnemu ruchowi w kierunkach, które z powodu ograniczeń nieholonomicznych są niedostępne przy pomocy pojedynczych sterowań. Kierunki takie (w postaci pól wektorowych) otrzymuje się z operacji zwanej *nawiasem Liego* określonej w następujący sposób:

$$[g_1, g_2] = \frac{\partial g_2}{\partial q} g_1 - \frac{\partial g_1}{\partial q} g_2$$

Ruch w ten sposób otrzymanym kierunku realizuje się za pomocą formuły CBHD. Mówi ona, że:

$$e(tX)e(tY) = e(tX + tY + \frac{t^2}{2}[X, Y] + \frac{t^3}{12}[[X, Y], X] - \frac{t^3}{12}[[X, Y], Y] - ...)$$

gdzie $e(tZ)$ oznacza działanie strumienia pola wektorowego Z przez czas t. Z powyższego wzoru wynika, że rezultatem złożenia tych dwóch strumieni jest przemieszczenie wynikające z działania każdego z tych strumieni z osobna oraz z reszty, której dla małych czasów t najbardziej znaczącym składnikiem jest jest pole wektorowe $[X, Y]$. Składnik $tX + tY$ można wyeliminować składając ruch z czterech segmentów:

$$e(tX)e(tY)e(-tX)e(-tY) = e(t^2[X, Y] + ...)$$

Zwrócić należy uwagę, że procedura opisana po lewej stronie powyższego wyrażenia jest na swój sposób cykliczna, tzn. równoważny rezultat (z dokładnością do błędu wynikającego z reszty wyższych stopni) otrzyma się rozpoczynając od dowolnego z czterech segmentów, lub nawet w dowolnej chwili każdego z segmentów. Ostatecznie więc przyjmuje jedną z czterech postaci:

$$\begin{gathered}
e(t_0X)e(tY)e(-tX)e(-tY)e((t - t_0)X) \\
e(t_0Y)e(-tX)e(-tY)e(tX)e((t - t_0)Y) \\
e(-t_0X)e(-tY)e(tX)e(tY)e((t_0 - t)X) \\
e(-t_0Y)e(tX)e(tY)e(-tX)e((t_0 - t)Y)
\end{gathered}$$

gdzie $t_0 \leqslant t$.

2 Implementacja

Powstała aplikacja pozwala na wizualicję rzeczywistego przemieszczenia robota, na którym zastosowano sekwencję sterowań zgodną z formułą CBHD w celu otrzymania przemieszczenia w kierunku $[g_1, g_2]$. W każdym przypadku zakładano zerowe warunki początkowe.

Dostępne klasy robotów

- monocykl

  Równanie stanu:

$$\begin{pmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \end{pmatrix} = \begin{pmatrix} \cos\theta \\ \sin\theta \\ 0 \end{pmatrix} u_1 + \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} u_2$$

  gdzie: x,y - współrzędne określające położenie robota, θ - orientacja robota względem osi X.

  Wygenerowane nowe pole wektorowe:

$$[g_1, g_2] = \begin{pmatrix} \sin\theta \\ -\cos\theta \\ 0 \end{pmatrix}$$

  Zgodnie z założeniem o zerowych warunkach początkowych widać, że:

$$[g_1, g_2] = \begin{pmatrix} 0 \\ -1 \\ 0 \end{pmatrix}$$

  co oznacza, że jest to ruch w kierunku prostopadłym do aktualnej orientacji robota.

- samochód kinematyczny

  Równanie stanu:

$$\begin{pmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \\ \dot{\phi} \end{pmatrix} = \begin{pmatrix} \cos\theta\cos\phi \\ \sin\theta\cos\phi \\ \frac{1}{l}\sin\phi \\ 0 \end{pmatrix} u_1 + \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \end{pmatrix} u_2$$

gdzie x, y - współrzędne określające położenie robota, θ - orientacja głównej osi samochodu, ϕ - orientacja kół względem głównej osi.

Wygenerowano pole wektorowe:

$$[g_1, g_2] = \begin{pmatrix} \cos\theta \sin\phi \\ \sin\theta \sin\phi \\ -\frac{1}{l}\cos\phi \\ 0 \end{pmatrix}$$

co przy zerowych warunkach początkowych daje:

$$[g_1, g_2] = \begin{pmatrix} 0 \\ 0 \\ -\frac{1}{l} \\ 0 \end{pmatrix}$$

i odpowiada zmianie orientacji samochodu "w miejscu".

Opis klas i najważniejszych metod

Roboty są reprezentowane przez obiekty klas, które dziedziczą po klasie `MobileRobot`: `Unicycle` oraz `KinematicCar`. W klasie bazowej zdefiniowane są pola `q` oraz `u`. Są to wektory określające aktualną konfigurację oraz wartości zmiennych sterujących. Klasa bazowa posiada również deklaracje dwóch metody czysto wirtualnych `dq()`: oraz `dqLie()` posiadających implementację w klasach pochodnych, obliczających prędkości dla aktualnych konfiguracji: w pierwszym przypadku dla rzeczywistego równania stanu $\dot{q} = G(q)u$, zaś w drugim dla pola wektorowego wygenerowanego przy pomocy nawiasu Liego.

Metodą wartą szczególnej uwagi jest metoda `executeMotion()`. Jej wywołanie powoduje zmianę konfiguracji robota wywołaną włączeniem określonego (pojedynczego) sterowania na określoną wartość i określony czas. To w niej zaimplementowany jest algorytm numerycznie rozwiązujący równanie stanu (więcej na ten temat w paragrafie *Dyskretyzacja i rozwiązanie równania stanu*. Posiada ona również możliwość zadziałania na robota strumieniem pola wektorowego $[g_1, g_2]$ przekazując jako ostatni argument tej metodzie wskaźnik do metody $dqLie()$.

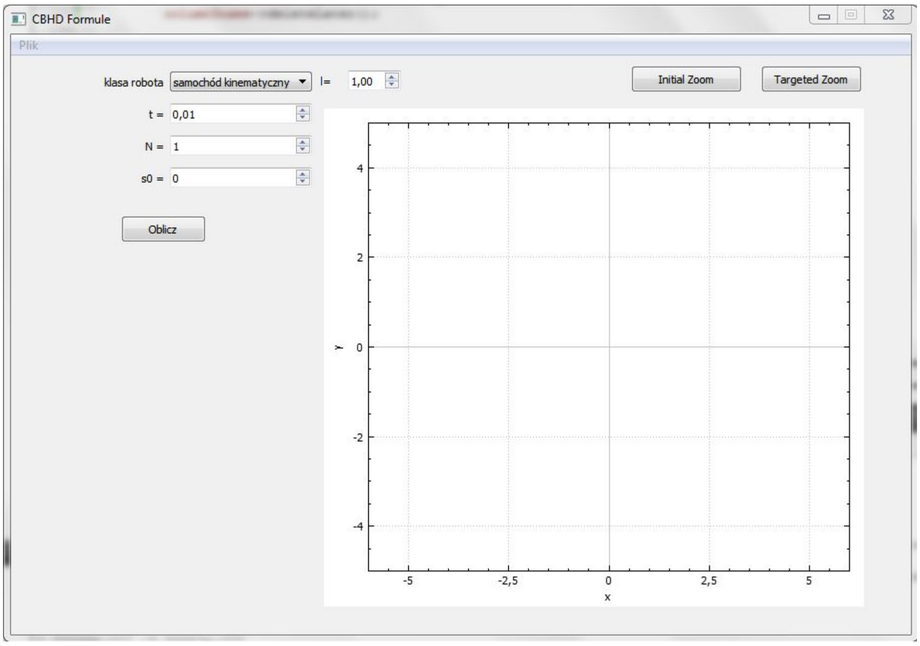
Klasa `Trajectory` służy do przechowywania trajektorii $q(t)$. Obiekty tej klasy są zwracane przez funkcję `executeMotion()`. Polem tej klasy jest tablica dwuwymiarowa `traj`, której pierwsza kolumna określa kolejne chwile czasu zaś kolejne wartości poszczególnych zmiennych stanu w tychże chwilach. Klasa ta udostępnia również możliwość łączenia dwóch trajektorii za pomocą metody `addTrajectory()`.

Klasa `CBHDProcedure` posiada pola określające parametry algorytmu CBHD oraz metodę `CBHDExecute()` wykonującą ów algorytm zgodnie z parametrami algorytmu, zachowanie trajektorii oraz powrót robota do pozycji początkowej. Metoda `designateExpectedPosition()` działa natomiast na robota strumieniem pola wektorowego $[g_1, g_2]$, zachowuje trajektorię takiego ruchu i również wraca do pozycji początkowej.

Dyskretyzacja i rozwiązanie równania stanu

Głównym problemem algorytmicznym rozwiązanym w toku prac nad aplikacją było rozwiązanie różniczkowego równania stanu. W tym celu posłużono się algorytmem Rungego-Kutty IV rzędu, przy czym krok tego algorytmu jest określony w jednym z pól `N` klasy `CBHDProcedure` określającym liczbę przedziałów, na które dzieli się jeden segment. Krok ten równy jest długości pojedynczego przedziału. Pole `s0` zaś jest liczbą całkowitą z przedziału $[0, 4N-1]$ określa numer przedziału, w którym rozpoczyna się cykl algorytmu.

3 Interfejs użytkownika



Użytkownik ma możliwość wyboru:

- klasy robota: monocyklu i samochodu kinematycznego
- czasu trwania jednego segmentu t
- liczby przedziałów N na którą dzielimy pojedynczy segment. Długość jednego przedziału stanowi długość kroku algorytmu Rungego-Kutty.
- $s_0 \in [0, 4N - 1]$ – numer przedziału rozpoczynającego cykl algorytmu
- dla samochodu kinematycznego, długości l

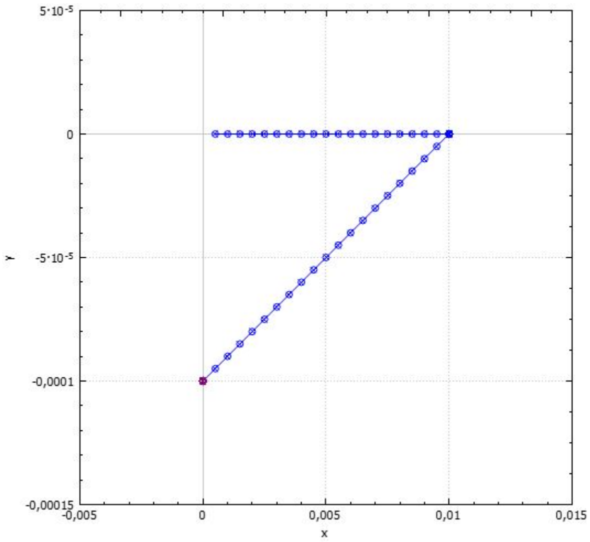
Program dostarcza skalowalnego okna wykresu. Po rozpoczęciu obliczeń zostaje wyświetlony wynik oraz tabela z wynikami ostatecznej pozycji w osiach ruchu, rzeczywistej pozycji oraz błędu ich położenia.

4 Wyniki symulacji

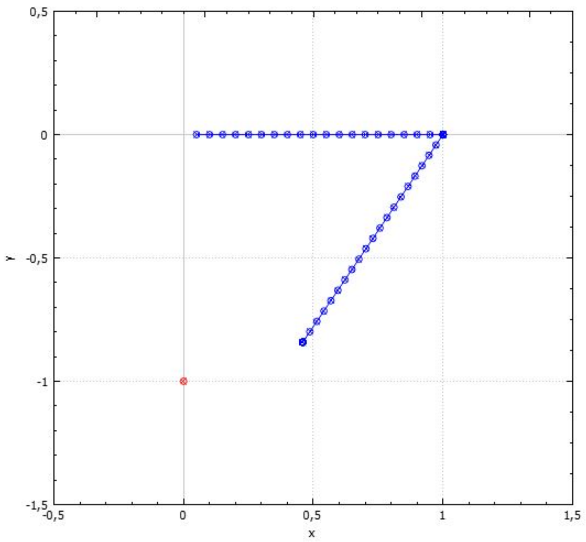
W celu sprawdzenia poprawności działania programu przeprowadzono testy dla monocykla i samochodu kinematycznego. Testy polegały na wykonaniu algorytmu CBHD dla małego i dużego czasu t, oraz dla różnych punktów początkowych s_0.

4.1 Monocykl

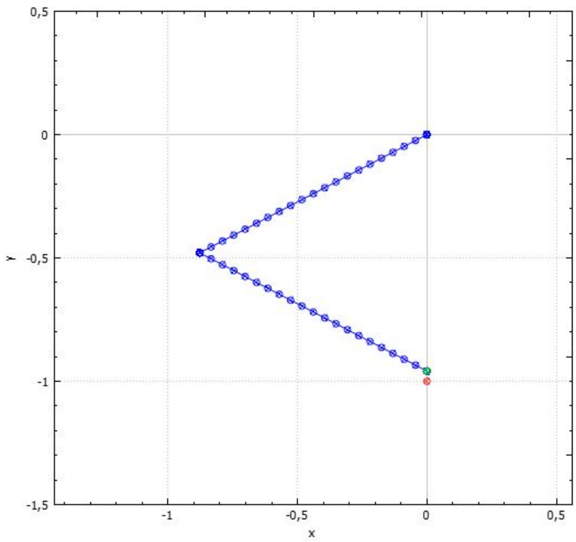
Dla monocykla przeprowadzono testy dla $t = 0.01s$ i $t = 1.00s$, oraz dla $s_0 = 10$ przy $N = 20$.



Rysunek 1: t=0.01 N=20 s_0=0, błąd=0.0$m/0rad$



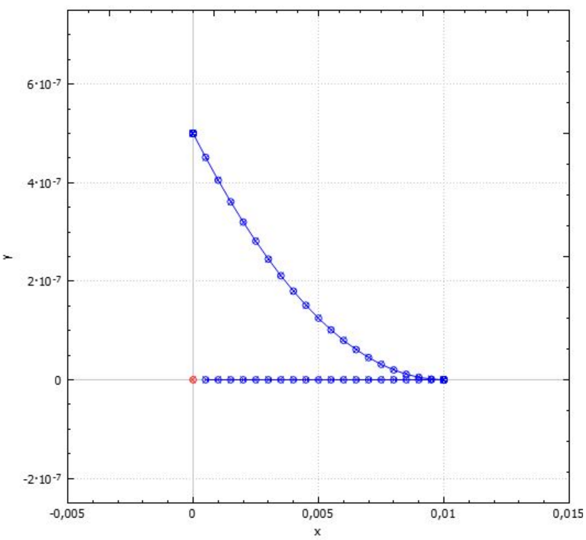
Rysunek 2: t=1.00 N=20 s_0=0, błąd=0.486$m/0rad$



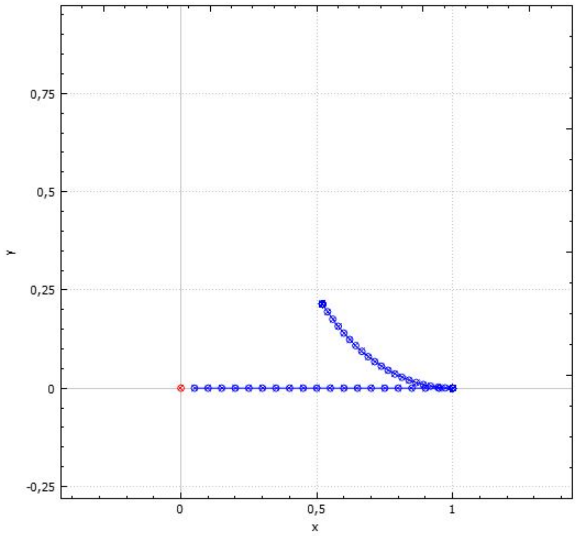
Rysunek 3: t=1.00 N=20 s_0=30, błąd=0.041$m/0rad$

4.2 Samochód kinematyczny

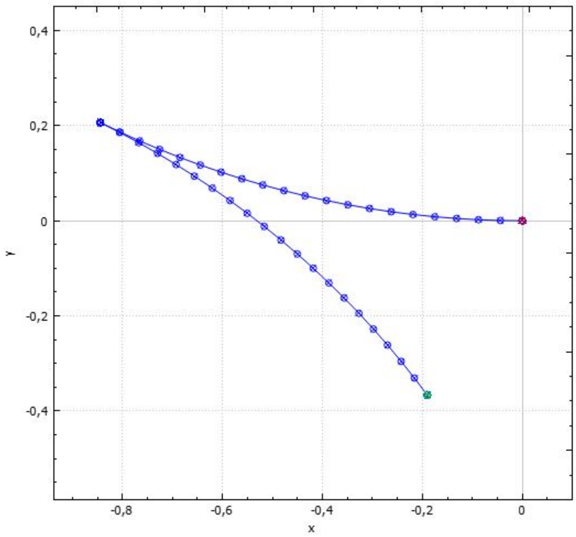
Dla samochodu kinematycznego przeprowadzono analogiczne testy jak w poprzednim przypadku



Rysunek 4: t=0.01 N=20 s_0=0, błąd=0.0$m/0rad$



Rysunek 5: t=1.00 N=20 s_0=0, błąd=0.563m/0.159rad



Rysunek 6: t=1.00 N=20 s_0=30, błąd=0.412m/0.041rad

Można zauważyć, że dla obu przypadków program zwraca spodziewane wyniki, to znaczy błąd algorytmu CBHD rośnie wraz ze wzrostem wartości czasu t.

Dzięki regulacji parametru s_0 możliwa jest minimalizacja błędu stanu końcowego robota.

5 Bibliografia

1. Dulęba, Ignacy. 1998. *Algorithms of Motion Planning for Nonholonomic Robots*. Oficyna Wydawnicza Politechniki Wrocławskiej

A Instrukcja rozwoju oprogramowania

Sposób implementacji algorytmu pozwala na łatwe dodawanie własnych klas robotów do programu. W tym celu należy rozszerzyć klasę bazową `MobileRobot`, implementując we własnej klasie metody `dq()` oraz `dqLie()` zawierające równanie stanu robota, oraz pole wektorowe wygenerowane przez nawias Liego.

Dodatkowo w oknie głównym programu należy dodać nową klasę robota do listy wyboru, oraz w klasie `MainWindow` dodać obsługę wyboru.

Repozytorium zawierające kod źródłowy programu znaleźć można pod adresem: `www.github.com/jedrek1993/PlanowanieRuchu`