Politechnika Wrocławska

Zakład Podstaw Cybernetyki i Robotyki

NARZĘDZIA KOMPUTEROWE W ROBOTYCE

Modelowanie kinematyki i dynamiki

Projekt przejściowy 2011/12
specjalności Robotyka
na Wydziale Elektroniki

Wrocław 2012

NARZĘDZIA KOMPUTEROWE W ROBOTYCE

Modelowanie kinematyki i dynamiki

Projekt przejściowy 2011/12
specjalności Robotyka
na Wydziale Elektroniki

Politechnika Wrocławska 2012

Redakcja techniczna

Janusz Jakubiak, Robert Muszyński

Skład raportu wykonano w systemie LaTeX

Spis treści

Wstęp

Oddajemy w ręce Czytelnika kompendium wiedzy dotyczące narzędzi komputerowych stosowanych w robotyce w zakresie modelowania kinematyki i dynamiki układów robotycznych. Jego autorami są uczestnicy Projektu przejściowego, realizowanego na specjalności Robotyka, na Wydziale Elektroniki Politechniki Wrocławskiej, w semestrze letnim roku akademickiego 2011/12. Ich zamiarem było zebranie w jednym miejscu informacji na temat aktulanie dostępnych narzędzi komputerowych, przydatnych we wspomnianym procesie modelowania, sposobu ich wykorzystania, a także porównanie własności i ocena przydatności tychże. Zdajemy sobie sprawę z tego, że prezentowana praca jest daleka od doskonałości i zapewne nie udzieli odpowiedzi na wiele palących kwestii tyczących modelowania, pozostajemy jednak w nadziei, że choć w pewnym stopniu ułatwi Czytelnikowi zapoznanie się z poruszaną tematyką.

Całość pracy została podzielona na dwie części: pierwszą, traktującą o samych narzędziach komputerowych, i drugą, dotyczącą ich zastosowań do modelowych przypadków. W części o narzędziach autorzy starali się przedstwić ich ogólną charakterystykę uzupełnioną prostymi przykładami aplikacji robotycznych. W części tej przedstawiono środowiska Wolfram Mathematica, Matlab, Microsoft Robotics Developer Studio, SimRobot, V-REP, Webots, Player/Stage, Autodesk Inventor, Open Dynamics Engine oraz JSBSim. W części drugiej zebrano pięć bardziej rozbudowanych zadań modelowania z zakresu robotyki, omówiono pokrótce teoretyczne podstawy ich rozwiązania oraz zilustrowano sposób implementacji w dwóch[1] środowiskach. Środowiska te wybrano spośród prezentowanych wcześniej tak, aby proces implementacji był możliwe najprostszy. I tak, omówiono tu problem modelowania sztywnego i elastycznego manipulatora stacjonarnego, sztywnego manipulatora mobilnego, sześcionożnego robota kroczącego i w końcu robota latającego typu quadrotor.

Janusz Jakubiak, Robert Muszyński
Wrocław, w maju 2012

[1] z jednym wyjątkiem

Część I

Narzędzia

1. Wolfram Mathematica

Marek Gulanowski

W pierwszym rozdziale pracy przedstawimy podstawy wykorzystania systemu obliczeń symbolicznych `Mathematica` do modelowania robotów na trzech przykładach:
— manipulatora sztywnego typu 2R,
— manipulatora typu 2R o elastycznych przegubach,
— robota mobilnego typu monocykl.
Celem jest ukazanie elementów funkcjonalności programu `Mathematica`, które mogą mieć zastosowanie w modelowaniu kinematyki i dynamiki robotów.

`Mathematica` jest programem obliczeniowym o szerokim wachlarzu zastosowań, rozwijanym przez firmę Wolfram od ponad 20 lat [1]. Wśród ogólnych zastosowań programu `Mathematica` można wyróżnić:
— obliczenia symboliczne:
  — rozwiązywanie równań i układów równań algebraicznych,
  — operacje na wektorach i macierzach,
  — przekształcanie wyrażeń matematycznych,
  — różniczkowanie i całkowanie,
  — rozwiązywanie równań różniczkowych.
— obliczenia numeryczne — odpowiedniki wyżej wymienionych obliczeń symbolicznych,
— obliczenia statystyczne i inne,
— importowanie i eksportowania kodu i danych, w tym eksport kodu w języku C,
— wizualizacji danych w formie:
  — zaawansowanych wykresów dwu- i trójwymiarowych,
  — animacji,
  — elementów interaktywnych (umożliwiających np. obserwację przebiegu wykresu w zależności od ręcznie ustawianego parametru).

1.1. Robotica — pakiet robotyczny dla programu Mathematica

W odróżnieniu od opisanego w rozdziale 2 środowiska MATLAB, `Mathematica` nie zawiera pakietów dedykowanych modelowaniu robotów. Został natomiast w tym celu stworzony w 1993 roku przez J. F. Nethery'ego i M. W. Sponga pakiet `Robotica` [3]. Umożliwia on między innymi:
— wprowadzenie kinematyki z użyciem parametrów Denavita-Hartenberga,
— wyliczenie na ich podstawie macierzy kolejnych przekształceń oraz kinematyki,
— wprowadzenie modelu dynamiki poprzez podanie mas, współrzędnych środka masy oraz macierzy inercji poszczególnych przegubów,
— wyliczenie na ich podstawie macierzy modelu dynamiki według formalizmu Eulera-Lagrange'a.
W niniejszym omówieniu pakiet `Robotica` będzie wykorzystywany do wykonywania podanych powyżej operacji.

1.2. Modelowanie kinematyki manipulatorów

Do modelowania kinematyki manipulatorów zostanie wykorzystany pakiet `Robotica`. W celu wykorzystania funkcji pakietu należy umieścić plik `robotica.m`[1] w katalogu roboczym[2] oraz zastosować komendę: `<< robotica.m`.

Zaobserwowano niepoprawne działanie funkcji `DataFile` z pakietu `Robotica` w programie `Mathematica` w wersji 8[3]. W celu zapewnienia prawidłowego działania tego pakietu, należy w pliku `robotica.m` usunąć linię 714 lub zamienić ją na komentarz, jak przedstawiono na wydruku 1.1.

Wydruk 1.1. Modyfikacja pakietu Robotica dla programu Mathematica w wersji 8: linia 714 przedstawiona wraz z otoczeniem

```
710
      If[StringLength[read] >7,
712     If[StringTake[read,8] == "DYNAMICS", skip=False]]];

714   (*Read[f, String];*)

716   read = Read[f, String];

718   If [read==EndOfFile, Print["Bad data file format."];
                  error=True;
720               Break[]];
```

Aby otrzymać kinematykę manipulatora należy wykonać następujące kroki:

1. załadowanie pakietu `Robotica` za pomocą komendy: `<< robotica.m`,
2. wprowadzenie danych dotyczących manipulatora określonych w wejściowym pliku tekstowym za pomocą funkcji `DataFile` (co omówiono powyżej),
3. obliczenie kinematyki: `FKin[]`.

W celu wprowadzenia modelu kinematyki, można skorzystać z funkcji `DataFile`, podając jako argument uprzednio przygotowany plik tekstowy, definiujący liczbę stopni swobody oraz rodzaj i parametry każdego przegubu manipulatora. Plik definiujący planarne dwuwahadło[4]. przedstawiono na wydruku 1.2.

Wydruk 1.2. Plik wejściowy funkcji DataFile

```
A Robotica input data file for
2 a two degree of freedom planar robot
  ---------------------------
4 DOF=2
  The Denavit-Hartenberg parameters
6 joint1=revolute
```

[1] Pakiet `Robotica` dostępny jest na stronie WWW University of Illinois: www-cvr.ai.uiuc.edu/~lab/ece470/robotica/robotica.m.

[2] Katalog roboczy można zidentyfikować za pomocą komendy `Directory`, a zmienić poleceniem `SetDirectory`. Alternatywnie, zamiast w katalogu roboczym, można umieścić plik `robotica.m` w katalogu instalacyjnym programu `Mathematica`, w podkatalogu `AddOns/Packages`.

[3] Jest ono najprawdopodobniej spowodowane odmiennym działaniem funkcji `Read` w wersji 8 programu `Mathematica` w stosunku do wersji 2, dla której napisany był pakiet `Robotica`.

[4] W przypadku, gdy nie ma potrzeby wyliczania modelu dynamiki, można określić jedynie kinematykę robota — nie należy wtedy umieszczać słowa kluczowego `DYNAMICS` ani kolejnych linii w pliku wejściowym.

```
a1=l1
alpha1=0
d1=0
theta1=q1
joint2=revolute
a2=l2
alpha2=0
d2=0
theta2=q2

DYNAMICS

gravity={0,g,0}
mass1=m1
center of mass={-(1/2)l1,0,0}
inertia matrix={1/3 m1 l1^2,0,0,0,0,0}
mass2=m2
center of mass={-(1/2)l2,0,0}
inertia matrix={1/3 m2 l2^2,0,0,0,0,0}
```

Plik ten określa zarówno kinematykę, jak i dynamikę manipulatora, która zostanie omówiona w sekcji 1.3. Kinematyka zdefiniowana jest w liniach 4–15. Konieczne jest podanie niżej opisanych parametrów.

DOF — liczba stopni swobody manipulatora (liczba przegubów): wartość minimalna: 2.
joint*n* — typ *n*-tego przegubu. Dopuszczalne wartości:
- — `revolute` (przegub obrotowy),
- — `prismatic` (przegub przesuwny),

a*n*, **alpha***n*, **d***n*, **theta***n* — parametry algorytmu Denavita-Hartenberga. Mogą one być podawane zarówno numerycznie, jak i symbolicznie, co widoczne jest w niniejszym przykładzie.

Tak przygotowany plik tekstowy należy przetworzyć za pomocą funkcji `DataFile`. Wywołanie funkcji `DataFile` (dla pliku wejściowego z wydruku 1.2) oraz komunikaty wygenerowane przez tę funkcję umieszczono na wydruku 1.3. Komunikaty te potwierdzają poprawność danych wejściowych.

Wydruk 1.3. Wywołanie funkcji `DataFile`

```
(* wczytanie pliku z definicja robota: *)
DataFile["Dokumenty/studia/PROP/robot.txt"]

(* komunikaty wypisane przez funkcje DataFile: *)
Kinematics Input Data
---------------------
Joint   Type            a       alpha   d       theta
1       revolute        l1      0       0       q1
2       revolute        l2      0       0       q2

Dynamics Input Data
-------------------
```

```
Gravity vector: [0, g, 0]
Link     mass
1        m1
2        m2
com vector
[-l1/2., 0, 0]
[-l2/2., 0, 0]
Inertia[1] =
 | (1/3)l1^2 m1 0 0 |
 | 0              0 0 |
 | 0              0 0 |
Inertia[2] =
 | (1/3)l2^2 m2 0 0 |
 | 0              0 0 |
 | 0              0 0 |
```

Aby możliwe było modelowanie kinematyki robota, konieczne jest wywołanie funkcji `ELDynamics[]`, która wyliczy macierze `A[i]`, będące odpowiednikami macierzy przekształceń A^i_{i-1} algorytmu Denavita-Hartenberga [4]. Wyznaczone także zostaną macierze `T[j,k]`, będące odpowiednikami macierzy przekształceń A^k_j (dla $j = 0 \ldots n-1$, $k = j+1 \ldots n$) oraz jakobian manipulatora `J`.

Macierz `T[0,2]`, której wygenerowanie opisano powyżej, umożliwia przekształcanie wektorów z przestrzeni przegubowej do przestrzeni zadaniowej. Polecenie

```
T[0,2]//MatrixForm
```

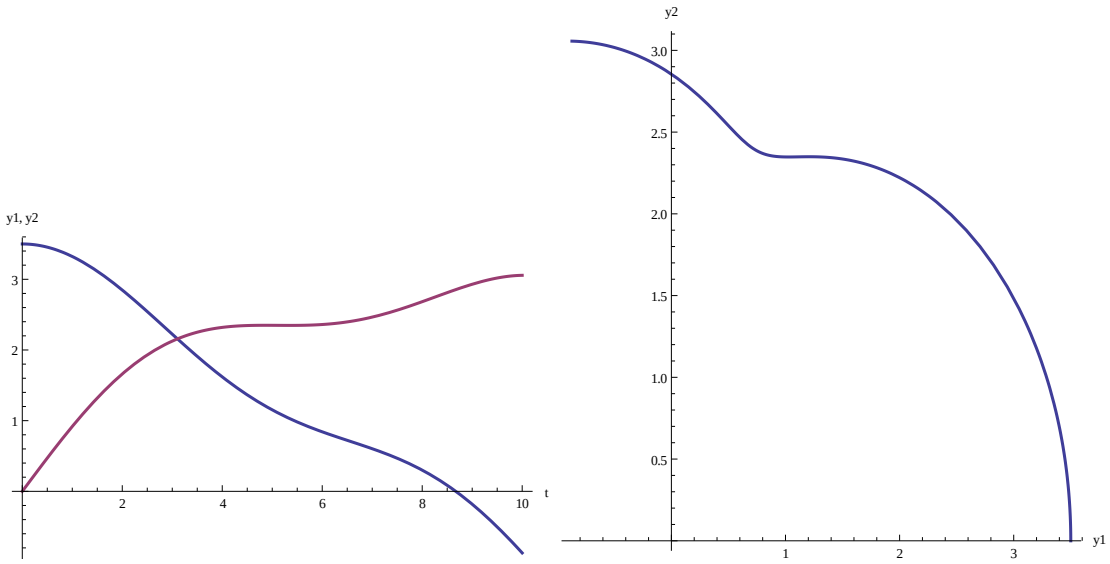
pozwala na obejrzenie tej macierzy. Jej postać dla rozpatrywanego robota podano poniżej

$$\begin{pmatrix} \text{Cos[q1+q2]} & -\text{Sin[q1+q2]} & 0 & \text{l1Cos[q1]}+\text{l2Cos[q1+q2]} \\ \text{Sin[q1+q2]} & \text{Cos[q1+q2]} & 0 & \text{l1Sin[q1]}+\text{l2Sin[q1+q2]} \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}. \tag{1.1}$$

Zatem dla zadanej trajektorii przegubowej $q(t)$ możliwe jest wyznaczenie trajektorii efektora $y(t)$. Sposób takiego przekształcenia trajektorii pokazano na wydruku 1.4.

Wydruk 1.4. Proste zadanie kinematyki dla dwuwahadła sztywnego

```
l1=3;l2=1/2;
q1 = 1/5 t; q2 =1/2t;
T[0,2][[;;,4]]//MatrixForm
Plot[{T[0,2][[1,4]],T[0,2][[2,4]]},{t,0,10},
     PlotStyle->Thick,AxesLabel->{"t","y1,␣y2"}]
ParametricPlot[{T[0,2][[1,4]],T[0,2][[2,4]]},
     {t,0,10},AxesOrigin->{0,0},PlotRange->All,
     AspectRatio->1,PlotStyle->Thick,AxesLabel->{"y1", "y2"}]
```



Rysunek 1.1. Wykresy współrzędnych zadaniowych efektora dla prostego zadania kinematyki

Wykonanie powyższych komend generuje trzy elementy wyjściowe — wektor współrzędnych zadaniowych efektora:

$$\begin{pmatrix} 3\mathrm{Cos}\left[\frac{t}{5}\right] + \frac{1}{2}\mathrm{Cos}\left[\frac{7t}{10}\right] \\ 3\mathrm{Sin}\left[\frac{t}{5}\right] + \frac{1}{2}\mathrm{Sin}\left[\frac{7t}{10}\right] \\ 0 \\ 1 \end{pmatrix} \tag{1.2}$$

oraz wykres dwóch pierwszych składowych wektora współrzędnych zadaniowych efektora i ich wykres parametryczny — oba przedstawione na rysunku 1.1.

1.3. Modelowanie dynamiki manipulatorów

Pakiet `Robotica` umożliwia także modelowanie dynamiki manipulatorów. Wymaga to wykonania następujący kroków (kroki 1–3 zostały omówione w sekcji 1.2):

1. załadowanie pakietu `Robotica` za pomocą komendy: `<< robotica.m`,
2. wprowadzenie danych dotyczących manipulatora określonych w wejściowym pliku tekstowym za pomocą funkcji `DataFile` (co omówiono w sekcji 1.2),
3. obliczenie kinematyki: `FKin[]`,
4. obliczenie dynamiki: `ELDynamics[]`,
5. zdefiniowanie równań dynamiki,
6. rozwiązanie równań dynamiki za pomocą funkcji `NDSolve`.

1.3.1. Dynamika manipulatora sztywnego

W celu wyprowadzenia modelu dynamiki manipulatora za pomocą funkcji `DataFile` należy umieścić odpowiednie parametry w pliku wejściowym, poprzedzone słowem kluczowym `DYNAMICS`. Na wydruku 1.2 podano je w liniach 17–25. Aby wyznaczenie macierzy modelu dynamiki było możliwe, należy podać dane opisane poniżej.

gravity — współrzędne wektora grawitacji,

mass$n$ — masa n-tego przegubu,
center of mass — współrzędne środka masy (dla każdego przegubu), wyrażone w lokalnym układzie współrzędnych, związanym z końcem danego przegubu,
inertia matrix — sześć nieredundantnych składowych macierzy inercji przegubu, także w lokalnym układzie współrzędnych. Należy przyjąć następującą, symetryczną postać macierzy inercji:

$$I = \begin{bmatrix} i_1 & i_2 & i_3 \\ i_2 & i_4 & i_5 \\ i_3 & i_5 & i_6 \end{bmatrix}. \tag{1.3}$$

Następnie należy wywołać funkcję `ELDynamics`, w ramach której wyznaczone zostają macierze do równań dynamiki:
MU — macierz sił bezwładności,
CM — macierz sił Coriolisa i odśrodkowych,
G — wektor sił grawitacji.
Macierze te pozwalają zdefiniować dynamikę manipulatora postaci

$$MU\ddot{q} + CM\dot{q} + G = u. \tag{1.4}$$

W opisany wyżej sposób dla dwuwahadła otrzymano następujące macierze:

$$MU = \begin{pmatrix} \frac{1}{4}\left(l_2^2 m_2 + l_1^2(m_1 + 4m_2) + 4l_1 l_2 m_2 \cos(q_2)\right) & \frac{1}{4} l_2 m_2 (l_2 + 2l_1 \cos(q_2)) \\ \frac{1}{4} l_2 m_2 (l_2 + 2l_1 \cos(q_2)) & \frac{l_2^2 m_2}{4} \end{pmatrix}, \tag{1.5}$$

$$CM = \begin{pmatrix} -\frac{1}{2} l_1 l_2 m_2 \sin(q_2)\dot{q}_2(t) & -\frac{1}{2} l_1 l_2 m_2 \sin(q_2)\left(\dot{q}_1(t) + \dot{q}_2(t)\right) \\ \frac{1}{2} l_1 l_2 m_2 \sin(q_2)\dot{q}_1(t) & 0 \end{pmatrix}, \tag{1.6}$$

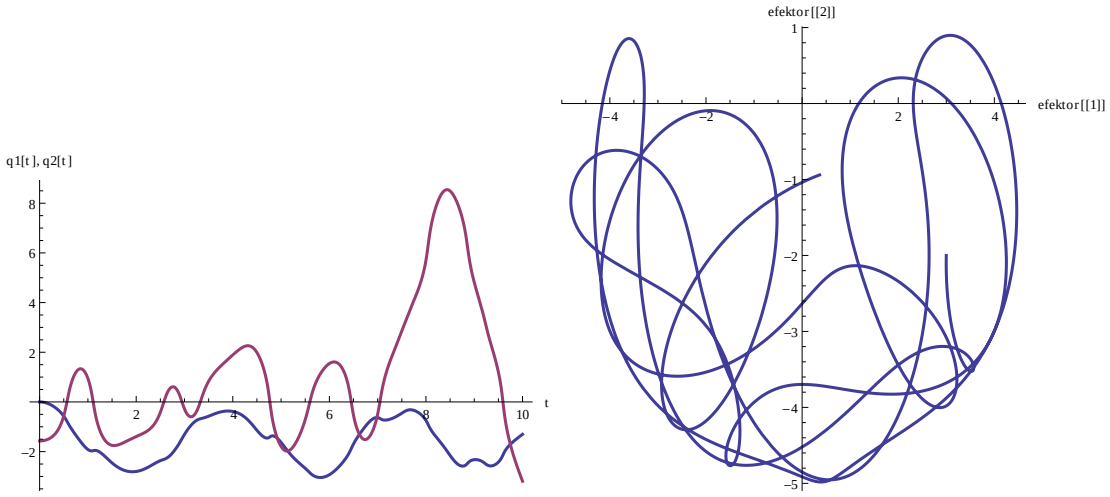
$$G = \begin{pmatrix} \frac{1}{2} g (l_1(m_1 + 2m_2)\cos(q_1) + l_2 m_2 \cos(q_1 + q_2)) \\ \frac{1}{2} g l_2 m_2 \cos(q_1 + q_2) \end{pmatrix}. \tag{1.7}$$

W zależności od przyjętego modelu dynamiki, możliwa jest modyfikacja wyliczonych macierzy, np. dodanie do elementów na przekątnej macierzy `MU` momentów bezwładności silników I_n.

Kolejnym etapem jest skonstruowanie równań różniczkowych dynamiki, które można numerycznie następnie rozwiązać za pomocą funkcji `NDSolve`. W celu poprawnego zinterpretowania równań przez tę funkcję, konieczna jest ich pewna modyfikacja, przedstawiona na wydruku poniżej. Następnie, po otrzymaniu rozwiązania równań dynamiki, możliwa jest ich prezentacja w formie wykresu od czasu, wykresu parametrycznego bądź animacji. Kod umożliwiający wykonanie opisanych powyżej operacji przedstawiony jest na wydruku 1.5.

Wydruk 1.5. Rozwiązanie równań dynamiki manipulatora sztywnego oraz wizualizacja rozwiązania

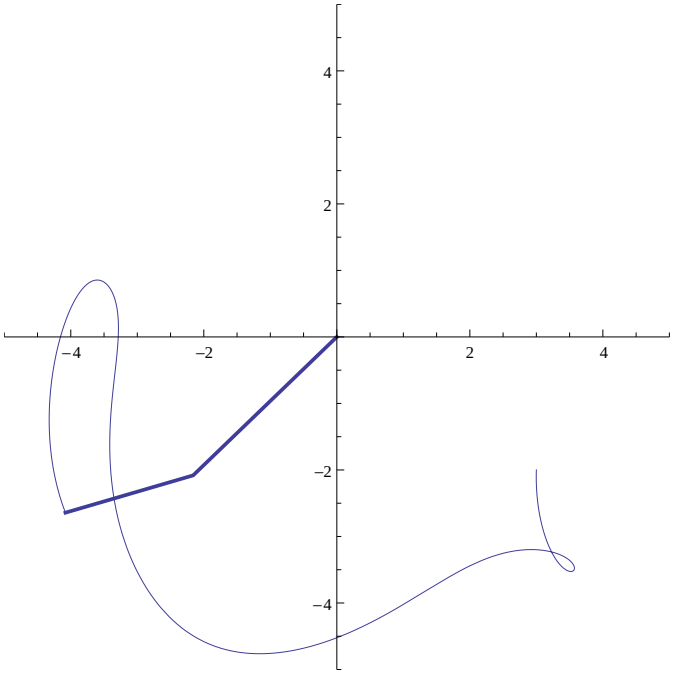
```
q[t_]:={q1[t],q2[t]};
u[t]={u1[t],u2[t]};
l1=3;l2=2;m1=10;m2=5;I1=0.5;I2=0.25;g=9.81;
MU[[1,1]]=MU[[1,1]]+I1;MU[[2,2]]=MU[[2,2]]+I2;
MU=MU/.{q1->q1[t],q2->q2[t]};
```



Rysunek 1.2. Wykresy rozwiązania równań dynamiki manipulatora sztywnego dla zerowych sterowań

```
G=G/.{q1->q1[t],q2->q2[t]};
CM=CM/.{q1->q1[t],q2->q2[t]}/.
  {q1[t]'[t]->q1'[t],q2[t]'[t]->q2'[t]};
dynamics=Apply[Equal,Transpose[
  {MU.q''[t]+CM.q'[t]+G,u[t]}],1];
init={q1[0]==0,q2[0]==-(\[Pi]/2),
  q1'[0]==0,q2'[0]==0};
u1[t]=0;u2[t]=0;
sol=NDSolve[Flatten[{dynamics,init}],
  {q1,q2},{t,0,10}];
Plot[{q1[t]/.sol,q2[t]/.sol},{t,0,10}(*...*)]
T[0,2]=T[0,2]/.{q1->q1[t],q2->q2[t]};
efektor=((T[0,2]/.sol).{0,0,0,1})[[1]];
ParametricPlot[{efektor[[1]],efektor[[2]]},
  {t,0,10},AspectRatio->1,AxesOrigin->{0,0},
  PlotRange->All]
T[0,1]=T[0,1]/.{q1->q1[t],q2->q2[t]};
ramie1=((T[0,1]/.sol).{0,0,0,1})[[1]];
Animate[
  Show[{
    ListPlot[{{0,0},{ramie1[[1]],ramie1[[2]]},
      {efektor[[1]],efektor[[2]]}}/.t->tt,Joined->True,
      PlotStyle->Thick,AspectRatio->1,
      PlotRange->{{-5,5},{-5,5}}],
ParametricPlot[{efektor[[1]],efektor[[2]]},{t,0,tt},
  AspectRatio->1,AxesOrigin->{0,0},PlotRange->All,
  PlotPoints->70]
  }],{tt,0,10},AnimationRate->.1]
```

Powyższy kod powoduje narysowanie wykresów, przedstawionych na rysunku 1.2. Następuje także odtworzenie animacji, ilustrującej zmiany konfiguracji manipulatora w czasie, z której pojedynczą klatkę umieszczono na rysunku 1.3.



Rysunek 1.3. Klatka animacji rozwiązania równań dynamiki manipulatora sztywnego dla zerowych sterowań

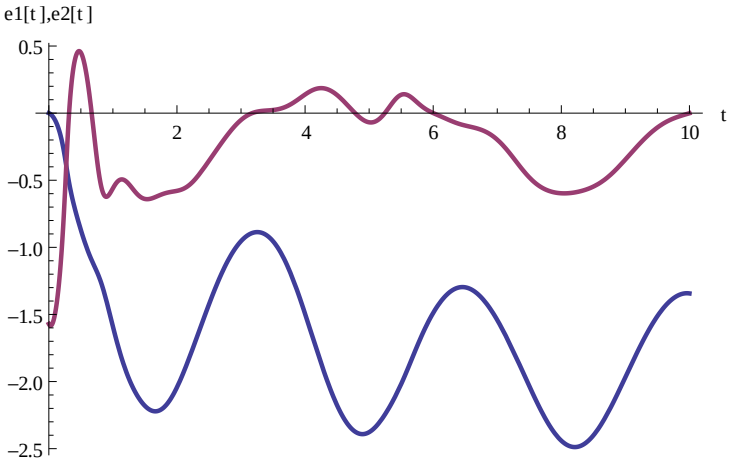
Alternatywny algorytm wyznaczenia równań dynamiki manipulatora o dowolnej liczbie przegubów w programie `Mathematica` przedstawiono w [2].

Sterowanie z wykorzystaniem algorytmu Qu i Dorseya

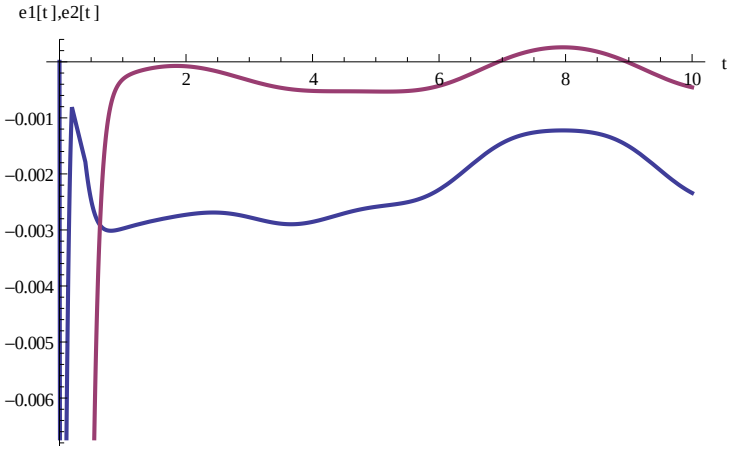
W celu zaprezentowania sposobu wykorzystania systemu `Mathematica` do symulacji algorytmów sterowania w tym podrozdziale przedstawimy implementację przykładowego algorytmu sterowania w pętli sprzężenia zwrotnego, w tym przypadku algorytm Qu i Dorseya, czyli sterownik PD [5]. Działania należy rozpocząć od przypisania niezerowej wartości zmiennym sterującym `u1[t]`, `u2[t]`, określenia trajektorii zadanej `qd[t]`, zdefiniowania sygnału błędu `e[t]`, a także nastaw regulacji `p` i `d`, odpowiednio dla członu proporcjonalnego i różniczkującego sterownika. Na wydruku 1.6 zamieszczono kod, służący do zdefiniowania i uruchomienia tego sterownika oraz wizualizacji efektów jego działania.

Wydruk 1.6. Implementacja algorytmu Qu i Dorseya dla manipulatora sztywnego

```
e[t_]:={e1[t],e2[t]};
e1[t_]:=q1[t]-q1d[t];e2[t_]:=q2[t]-q2d[t];
q1d[t_]:=Sin[t/5];q2d[t_]:=Sin[t];
p={100,1000,100000,1000000};
d={10,100,10000,100000};
contr={u1[t]->-p [[#]]e1[t]-d[[#]] e1'[t],
u2[t]->-p [[#]]e2[t]-d[[#]] e2'[t]
  }&/@Range[4]
solqd=NDSolve[Flatten[{dynamics,init}/.#],
  {q1,q2},{t,0,10},MaxSteps->{1000000}]&/@contr
Plot[{e1[t]/.solqd[[#]]/.t->tt,e2[t]/.
  solqd[[#]]/.t->tt},{tt,0,10},PlotStyle->Thick,
  AxesLabel->{"t","e1[t],e2[t]"}]&/@Range[4]
```



Rysunek 1.4. Sterowanie manipulatorem z wykorzystaniem algorytmu Qu i Dorseya dla nastaw $p = 100, d = 10$



Rysunek 1.5. Sterowanie manipulatorem z wykorzystaniem algorytmu Qu i Dorseya dla nastaw $p = 10000, d = 1000$

Na rysunkach 1.4–1.5 przedstawiono wyniki symulacji. Zaobserwować można zmniejszanie się wartości ustalonej sygnału błędu dla różnych wartości nastaw regulacji.

1.3.2. Dynamika manipulatora o elastycznych przegubach

Modelowanie dynamiki manipulatora o elastycznych przegubach wymaga uprzedniego wykonania tych samych kroków, które koniecznie były przy modelowaniu manipulatora sztywnego — zostały one omówione w sekcji 1.3.1. Na wydruku 1.7 przedstawiono sposób zdefiniowania dynamiki manipulatora o elastycznych przegubach.

Wydruk 1.7. Dynamika manipulatora o elastycznych przegubach

```
q[t_]:={q1[t],q2[t]};
qmot[t_]:={qmot1[t],qmot2[t]};
u[t_]:={u1[t],u2[t]};
l1=3;l2=2;m1=10;m2=5;I1=0.5;I2=0.25;g=9.81;
MU[[1,1]]=MU[[1,1]]+I1;MU[[2,2]]=MU[[2,2]]+I2;
MU=MU/.{q1->q1[t],q2->q2[t]};
```

```
G=G/.{q1->q1[t],q2->q2[t]};
CM=CM/.{q1->q1[t],q2->q2[t]}/.
  {q1[t]'[t]->q1'[t],q2[t]'[t]->q2'[t]};
K={{k1,0},{0,k2}};
i={{I1,0},{0,I2}};
dynElast=Flatten[{
  Apply[
    Equal,Transpose[{MU.q''[t]+CM.q'[t]+G,{0,0}}],1],
  Apply[
    Equal,Transpose[
      {i. qmot''[t]+(K.(qmot[t]-q[t])),u[t]}],1]
}];
```

1.4. Modelowanie dynamiki robota mobilnego typu monocykl

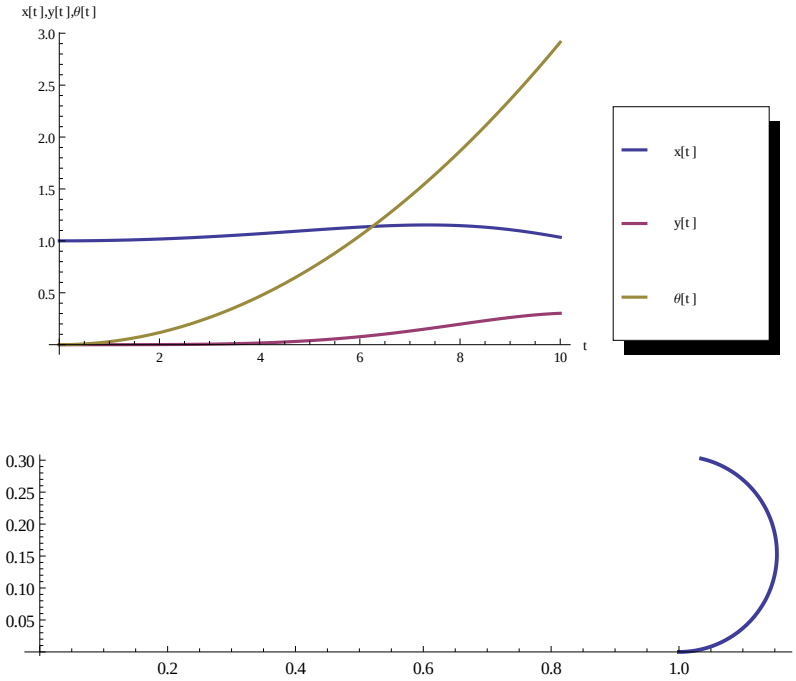
Program `Mathematica` umożliwia także modelowanie dynamiki układów nieholonomicznych, w tym robotów mobilnych. W tym przypadku skorzystanie z pakietu `Robotica` nie jest jednak możliwe. Konieczne jest zatem wyprowadzenie modelu dynamiki robota, np. z wykorzystaniem metody podanej w [2]. Przyjęto następującą postać modelu dynamiki monocykla:

$$\dot{q} = G\eta M\dot{\eta} = Bu \tag{1.8}$$

Na wydruku 1.8 przedstawiono implementację dynamiki robota typu monocykl [5] wraz z przykładowym sterowaniem.

Wydruk 1.8. Dynamika robota mobilnego typu monocykl

```
q[t_]:={x[t],y[t],th[t]};
eta[t]={v[t],w[t]};eta'[t]={v'[t],w'[t]};
u[t]={u1[t],u2[t]};
mmat={
  {mp+3mk,0},
  {0,iz+2mk+l^2+1/2 mk r^2+l^2 mk}};
bmat={{1,0},{0,1}};
gmat={{Cos[th[t]],0},{Sin[th[t]],0},{0,1}};
u[t]={1,1};
dynamics=
  Flatten[{
    Apply[Equal,Transpose[{q'[t],(gmat.eta[t])}],1],
    Apply[Equal,Transpose[{mmat.eta'[t],bmat.u[t]}],1]
  }];
mk=5;mp=97;iz=6.609;l=0.3;r=0.075;
init={x[0]==1,y[0]==0,th[0]==0,v[0]==0,w[0]==0};
sol=NDSolve[Flatten[{dynamics,init}],
  {x,y,th,v,w},{t,0,10}];
Needs["PlotLegends`"];
Plot[{x[t]/.sol,y[t]/.sol,th[t]/.sol},{t,0,10}(*...*)]
ParametricPlot[{x[t]/.sol[[1]],y[t]/.sol[[1]]},
  {t,0,10}(*...*)]
```



Rysunek 1.6. Wynik symulacji dynamiki robota mobilnego typu monocykl

Na rysunku 1.6 umieszczono wykresy w dziedzinie czasu oraz wykres parametryczny współrzędnych robota, wygenerowane za pomocą kodu z wydruku 1.8. W celu implementacji układów sterowania należy zastosować tę samą metodologię co przedstawiona w podrozdziale 1.3.1.

1.5. Podsumowanie

Opisane powyżej przykłady zastosowania programu `Mathematica` do modelowania konkretnych, prostych układów robotycznych mogą posłużyć za punkt wyjścia do modelowania bardziej złożonych robotów. Mogą być do tego wykorzystane zarówno pakiet `Robotica`, jak i metodyka opisana w [2].

Zalety programu `Mathematica` w modelowaniu dynamiki robotów to przede wszystkim możliwość wykorzystania zaawansowanego przetwarzania wyrażeń symbolicznych w celu wyprowadzenia równań dynamiki, numeryczne rozwiązanie tych równań dla zadanych sterowań (implementacja algorytmów sterowania), wizualizacja otrzymanego rozwiązania, a także eksport kodu w języku C. Program jest dostępny dla różnych systemów operacyjnych. Ograniczenia użyteczności systemu `Mathematica` wynikają z długiego czasu wykonywania bardziej złożonych obliczeń oraz ich złożoności pamięciowej, a szczególnie braku jakichkolwiek bardziej rozbudowanych pakietów przeznaczonych do modelowania układów robotycznych, wspieranych przez firmę Wolfram.

Bibliografia

[1] Wolfram Mathematica 8.

[2] R. Muszyński. Techniki komputerowe w robotyce. Mathematica – przykłady robotyczne. `http://rab.ict.pwr.wroc.pl/~mucha/TechKomp/index.html`, Wykłady na Wydziale Elektroniki Politechniki Wrocławskiej, 2011.

[3] J. F. Nethery, M. W. Spong. Robotica: A mathematica package for robot analysis. *IEEE Robotics & Automation Magazine*, 1994.
[4] K. Tchoń. Robotyka. Wykłady na Wydziale Elektroniki Politechniki Wrocławskiej, 2009.
[5] K. Tchoń, A. Mazur, R. Hossa, I. Dulęba, R. Muszyński. *Manipulatory i roboty mobilne*. Akademicka Oficyna Wydawnicza PLJ, 2000.

2. Matlab

Zuzanna Pietrowska, Tomasz Płatek

MATLAB [4] jest interaktywnym środowiskiem zorientowanym na zastosowania naukowe i inżynierskie rozwijanym przez firmę MathWorks. Zawiera własny język programowania wysokiego poziomu. Wśród głównych zastosowań można wymienić: obliczenia numeryczne, projektowanie algorytmów oraz analizę, przetwarzanie i wizualizację danych. W niniejszej pracy rozważana była wersja R2010b.

Ważną cechą MATLABa jest dostarczanie wielu gotowych rozwiązań z różnych dziedzin nauki. Biblioteki tej aplikacji zawierają gotowe funkcje z zakresu algebry, statystyki, metod optymalizacji oraz wielu innych. Dodatkową funkcjonalność wnoszą moduły zwane Toolboxami. Każdy taki moduł zawiera narzędzia (w postaci funkcji lub graficznych aplikacji) dostarczające rozwiązań w konkretnej dziedzinie nauki. Biblioteki wspierane przez firmę MathWroks są ciągle rozwijane i zawierają najnowocześniejsze rozwiązania.

Kolejną ważną funkcjonalnością MATLABa jest współpraca z innymi aplikacjami. Niezależni producenci oprogramowania dostarczają rozwiązań umożliwiających eksport danych do tego środowiska. MATLAB umożliwia również korzystanie z kodu napisanego w innym języku. Ważną zaletą tego oprogramowania jest również przenośność między różnymi platformami.

Gotowe rozwiązania mogą być kompilowane do wykonywalnych aplikacji, włączając graficzny interfejs użytkownika. MATLAB jest więc środowiskiem o bardzo szerokim zakresie zastosowań i wspierającym pracę użytkownika w wielu różnych aspektach.

Niniejszy raport rozpoczyna się od przeglądu pakietów rozszerzających funkcjonalność MATLABa, ze szczególnym naciskiem na Robotics Toolbox. W następnych rozdziałach zaprezentowano różne sposoby modelowania z użyciem omawianego środowiska. Kolejno jest to implementacja dwuwahadła z wykorzystaniem Robotics Toolbox, modelowanie monocykla w mplikach oraz manipulatora o elastycznych przegubach w Simulinku. Każdy opis zawiera podsumowanie, którego główny nacisk położony jest na wady i zalety omawianego rozwiązania.

2.1. Przegląd pakietów

Pakiety (ang. Toolboxes) są specjalistycznymi bibliotekami oprogramowania. Są one bardzo ważną częścią MATLABa rozszerzającą znacznie jego możliwości. W tym rozdziale przedstawiono wybrane pakiety, które mogą być szczególnie użyteczne w dziedzinie robotyki. Poniższe rozwiązania są wspierane przez producenta środowiska MATLAB. Oprócz nich można znaleźć oprogramowanie niezależnych twórców. Przykładem takiego pakietu jest Robotics Toolbox, który ze względu na szerokie możliwości zastosowania w robotyce został opisany w rozdziale 2.2. Przeglądu dokonano na podstawie dokumentacji MATLABa [4].

Control System Toolbox jest narzędziem wspierającym modelowanie systemów liniowych i ich sterowanie. Umożliwia zamodelowanie badanego obiektu m.in. przy użyciu transmitancji operatorowej oraz równań stanu. Obsługuje zarówno systemy SISO jak i MI-

MO. Przy jego pomocy można badać systemy ciągłe lub dyskretne. Pakiet ten posiada również wbudowane narzędzia do badania sterowania przy pomocy sterownika PID. Graficzny interfejs nie ogranicza się wyłącznie do prezentacji wyników lecz umożliwia również wprowadzanie zmian w badanym systemie.

Robust Control Toolbox jest pakietem wspierającym badania symulacyjne odpornych układów sterowania. Modele stworzone w tym narzędziu mogą uwzględniać niedokładności w modelu oraz zakłócenia zewnętrzne. Zapewnione są również procedury umożliwiające analizę najgorszego przypadku. Zaimplementowano w nim algorytmy sterowania odpornego m.in. H-nieskończoność, synteza μ. Ostatnią ważną funkcjonalnością są algorytmy redukcji modelu oraz jego weryfikacji po przeprowadzonym procesie upraszczania.

Fuzzy Logic Toolbox dostarcza gotowego zestawu narzędzi z zakresu logiki rozmytej. Umożliwia projektowanie systemów tego typu przy użyciu graficznego interfejsu użytkownika. Wspierane są również zaawansowane techniki, takie jak tworzenie systemów adaptacyjnych (ang. neurofuzzy) oraz metody klasyfikacji rozmytej (ang. fuzzy clustering).

Optimization Toolbox to pakiet wspierający obliczenia z dziedziny optymalizacji. Wspierane są metody programowania matematycznego m.in. programowanie liniowe oraz programowanie kwadratowe. Oprócz tego w rozwiązywanych problemach można stosować algorytmy optymalizacji nieliniowej: metody quasi-Newtonowskie, Neldera-Meada oraz wiele innych. Również w tym pakiecie otrzymujemy pełne wsparcie w zakresie wizualizacji otrzymanych wyników.

Computer Vision System Toolbox jest narzędziem wspierającym projektowanie algorytmów z dziedziny systemów wizyjnych. Najważniejszym jego elementem są rozwiązania dotyczące analizy obrazów wideo. W ty zakresie użytkownik ma do dyspozycji gotowe rozwiązania umożliwiające ekstrakcję cech, detekcję ruchu oraz śledzenie obiektów. Ważne są jednak również rozwiązania umożliwiające przechwycenie obrazu wideo oraz jego wyświetlenie. Gotowy algorytm może zostać wygenerowany do postaci kodu w języku C.

SimElectronics jest narzędziem wspierającym modelowanie i symulacje układów elektronicznych i mechatronicznych. Zawiera on bogatą bazę gotowych elementów z których możemy komponować nasz układ m.in. różnego rodzaju półprzewodniki, silniki, sterowniki czy sensory.

Wymienione pakiety nie wyczerpują wszystkich możliwości zastosowania MATLABa w robotyce. W opracowywaniu zebranych danych przydatny może okazać się pakiet Statistics Toolbox. W zastosowaniach związanych z przetwarzaniem sygnałów można skorzystać z Signal Processing Toolbox. Podsumowując, zakres wspieranych rozwiązań jest bardzo szeroki, warto więc przed przystąpieniem do pracy nad nowym zagadnieniem dokonać przeglądu metod dostarczanych przez MATLABa i jego pakiety.

2.2. Robotics Toolbox

Robotics Toolbox [2] jest darmowym pakietem narzędziowym środowiska MATLAB stworzonym i rozwijanym od 15 lat przez Peter'a Cork'a, dostępnym na zasadach GNU Lesser General Public License.

Robotics Toolbox jest to zestaw funkcji w mplikach, które są użyteczne w badaniach i symulacji układów dynamicznych w szczególności sztywnych manipulatorów przemysłowych. Dostarcza narzędzi do modelowania oraz obliczeń między innymi kinematyki prostej, kinematyki odwrotnej, dynamiki prostej, dynamiki odwrotnej, macierzy zręczności czy generowania dopuszczalnych trajektorii. W ramach pakietu zaimplementowane są przykładowe powszechnie znane i stosowane manipulatory takie jak Puma560, czy

Stanford. Robotics Toolbox zawiera także funkcje do manipulacji reprezentacjami modeli oraz konwersji pomiędzy różnymi typami danych takimi jak wektory, macierze obrotów, kwaterniony i inne.

Aktualnie dostępna jest wersja 9.0 Robotics Toolbox. Prócz obsługi sztywnych manipulatorów dodano w niej narzędzia do symulacji robotów mobilnych takie jak standardowe algorytmy planowania ścieżki, samolokalizacji czy budowy mapy otoczenia. Pakiet zawiera ponadto implementację nieholonomicznego robota mobilnego klasy(1,1) w postaci schematu blokowego w środowisku MATLAB/Simulink. Dostarcza również implementacji w postaci schematu blokowego MALAB/Simulink robota latającego quadcopter.

Robotic Toolbox umożliwia analizę wyników eksperymentów przeprowadzonych na rzeczywistych robotach.

2.2.1. Nota implementacyjna umożliwiająca szybki start

Aby przystąpić do implementacji modelu robota należy zapoznać się z dwoma podstawowymi klasami Robotics Toolbox.

Link — Klasa reprezentująca ogniwo manipulatora wraz z przegubem. Zawiera zarówno jego parametry kinematyczne jak i dynamiczne: tensor bezwładności, parametry silnika, współczynniki tarcia.

SerialLink — Jest to klasa reprezentująca sztywny manipulator utworzony z obiektów typu Link lub tablicy parametrów kolejnych transformacji algorytmu DH.

Implementacja pojedynczego ogniwa manipulatora `Link` wymaga szerszego komentarza:

1. Kinematyka ogniwa specyfikowana jest już w konstruktorze parametryzowanym:
   `Link(dh, options)`
   Obiekt jest tworzony w oparciu o wektor parametrów opisujących transformację Denavita-Hartenberga. Zapis parametrów transformacji jest następujący:
   $dh = [\theta \quad d \quad a \quad \alpha \quad \sigma]$, gdzie:
   — θ – zmienna stanu w przypadku przegubu rotacyjnego,
   — d – zmienna stanu w przypadku przegubu translacyjnego,
   — a – długość ogniwa,
   — α – stały kąt, opisujący skręt ogniwa względem poprzedniego ogniwa,
   — σ – parametr nie stanowiący części opisu transformacji DH, informuje o typie przegubu (przyjmuje wartość 0 jeżeli przegub jest obrotowy, 1 jeżeli przegub jest translacyjny).
   Powyższe można wyrazić w standardowej notacji DH (options='standard') lub przy użyciu zmodyfikowanej notacji DH (options='modified').
2. Przy implementacji ogniwa manipulatora należy podać następujące parametry dynamiki

   — m – masa ogniwa,
   — r – położenie środka masy ogniwa,
   — I – tensor bezwładności ogniwa postaci:

$$I = \begin{bmatrix} I_{xx} & I_{xy} & I_{xz} \\ I_{xy} & I_{yy} & I_{yz} \\ I_{xz} & I_{yz} & I_{zz} \end{bmatrix}, \tag{2.1}$$

   — J_m – bezwładność silnika,
   — G – przełożenie przekładni zębatej silnika,

— B – tarcie wiskotyczne dla silnika.

Parametry dynamiki umożliwiają późniejszą prawidłową symulację zachowania robota pod wpływem działających sił.

W celu zapoznania się z pozostałymi polami oraz metodami wymienionych klas odsyłamy do dokumentacji pakietu [2].

2.2.2. Implemetacja manipulatora typu dwuwahadło

Przed rozpoczęciem pracy z RoboticsToolbox należy każdorazowo uruchomić mplik `startup_rvc`, który inicjalizuje niezbędne zmienne.

Poniższy kod przedstawia sposób implementacji manipulatora typu dwuwahadło w Robotics Toolbox.

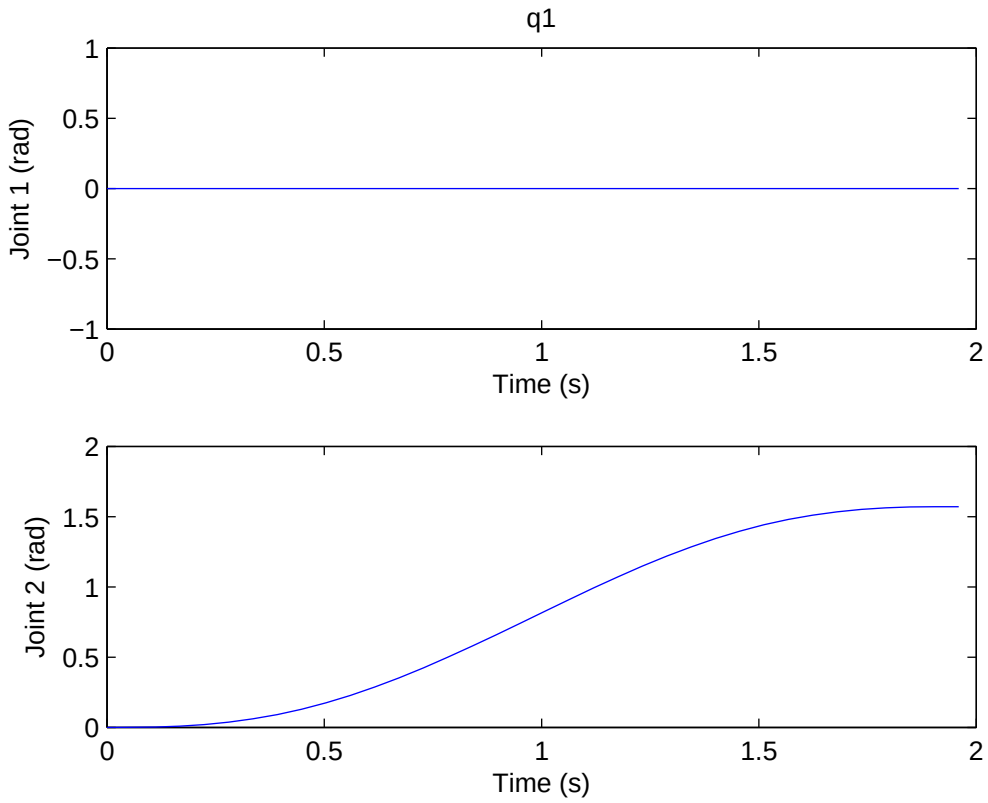
```
L(1) = Link([ 0 0 1 0 0], 'standard');
L(2) = Link([ 0 0 1 0 0], 'standard');

L(1).m = 1;
L(2).m = 1;

L(1).r = [ -0.5  0  0];
L(2).r = [ -0.5  0  0];

L(1).I = [ 0 0 0; 0 1/3 0; 0 0 1/3];
L(2).I = [ 0 0 0; 0 1/3 0; 0 0 1/3];

%bezwladnosc silnika
L(1).Jm =  1/3;
L(2).Jm =  1/3;
%przelozenie przekladni zebatej
L(1).G = 0;
L(2).G = 0;
% tarcie wiskotyczne
L(1).B =  0;
L(2).B =  0;

dwuwahadlo = SerialLink(L,'base',...
    troty(-pi/2), 'name', '2R');
```

Położenia środków ciężkości ogniw oraz ich tensory bezwładności zostały wyliczone przy założeniu, że ogniwa są to jednorodne cienkie pręty oraz, że odpowiednie lokalne układy współrzędnych są umieszczone na końcach przegubów w kierunku ujemnym osi OX. Opcja 'base' w konstruktorze robota 2R oznacza ustawienie bazy manipulatora względem podłoża. Wykorzystana funkcja `troty(kat)` to transformacja reprezentująca obrót względem osi OY.

Robota zdefiniowanego jak powyżej można wyświetlić w postaci reprezentacji graficznej, określając konfigurację w przestrzeni przegubowej

```
dwuwahadlo.plot([0 0]);
```

i następnie sterować nim ręcznie wykorzystując funkcję `teach()`.

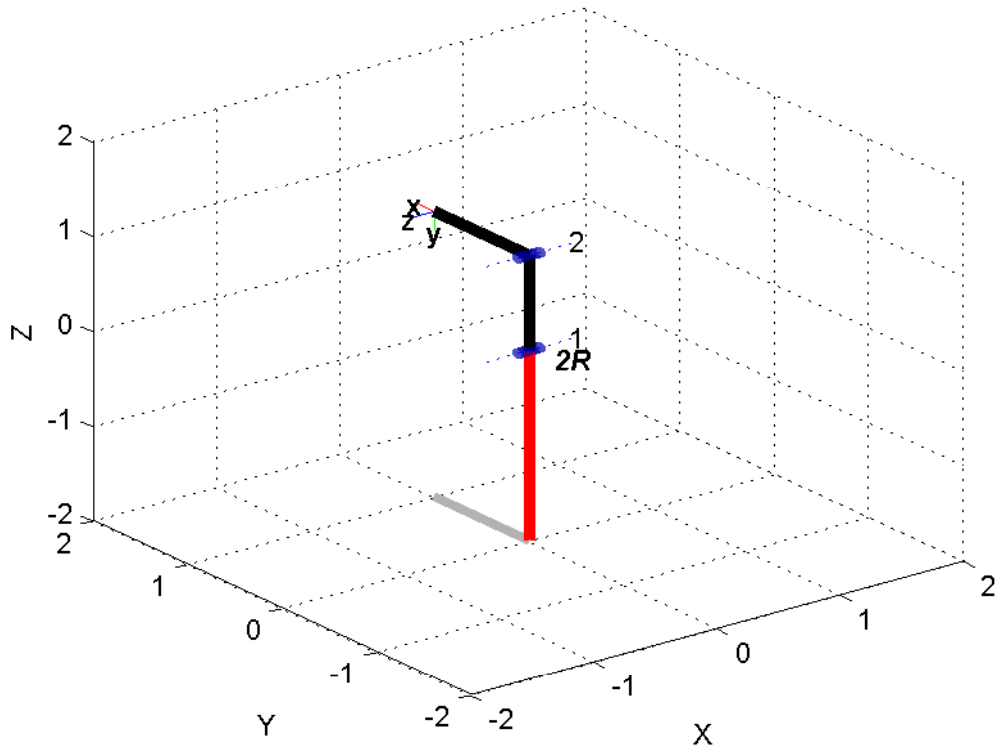


Rysunek 2.1. Trajektoria przegubowa wygenerowana przez funckję `jtraj(qz,qr,t)`

Można również wykorzystać funkcję `jtraj(qz,qr,t)`, która oblicza dopuszczalną trajektorię wielomianową ruchu robota w przestrzeni przegubowej i wyświetlić ruch realizowany przez poszczególne przeguby, co realizuje poniższy fragment kodu

```
t = [0:.056:2];

qz = [0 0];
qr = [0 pi/2];
q = jtraj(qz, qr, t);

figure;hold;
subplot(2,1,1)
plot(t,q(:,1))
title('q1')
xlabel('Time␣(s)');
ylabel('Joint␣1␣(rad)')
subplot(2,1,2)
plot(t,q(:,2))
xlabel('Time␣(s)');
ylabel('Joint␣2␣(rad)')
hold;
```

Uzyskane wykresy przedstawia rysunek 2.1.



Rysunek 2.2. Pojedyncza klatka animacji

Dysponując trajektorią przegubową można przeprowadzić animację ruchu manipulatora, wykorzystując odpowiednie przeciążenie metody `plot`

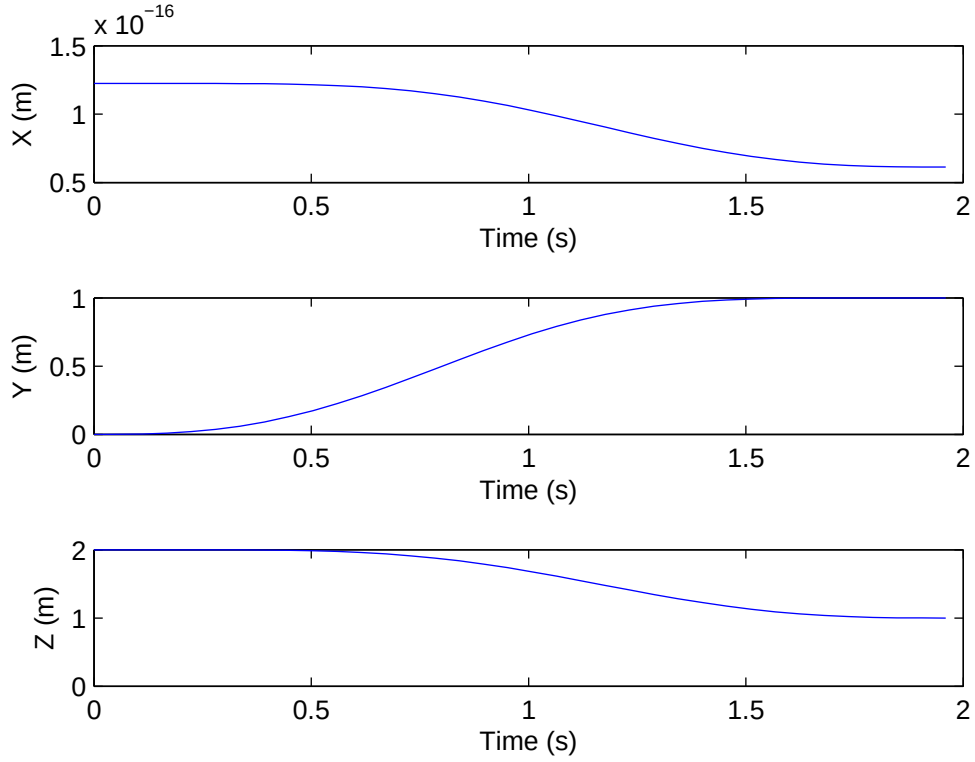
```
dwuwahadlo.plot(q);
```

Widoczne podczas animacji segmenty modelu nie są tożsame ogniwom robota, lecz łączą początki układów współrzędnych stowarzyszonych z poszczególnymi ogniwami manipulatora. Na końcu robota wyświetlany jest prawoskrętny układ współrzędnych, który ma za zadanie uwidocznić orientację efektora. Wyświetlany jest również cień manipulatora. Pojedyncza klatka symulacji przedstawiona została na rysunku 2.2.

Po zamodelowaniu dwuwahadła możliwe jest obliczenie kinematyki prostej robota w punkcie z wykorzystaneim funkcji `fkine(qz)`. Poniżej przedstawione jest odpowiednie wywołanie wraz z otrzymanym wynikiem.

```
dwuwahadlo.fkine(qz)

K =

    0.0000         0   -1.0000    0.0000
         0    1.0000         0         0
    1.0000         0    0.0000    2.0000
         0         0         0    1.0000
```

Kinematykę prostą można obliczyć również wzdłuż wygenerowanej wcześniej trajektorii

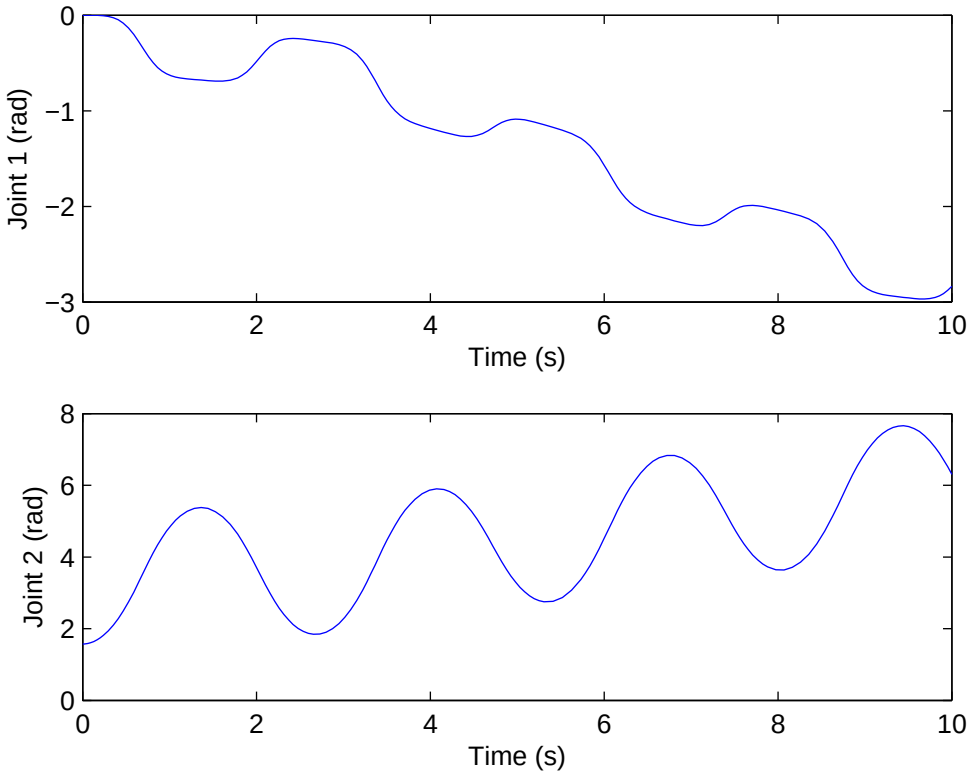


Rysunek 2.3. Ruch realizowany przez efektor w przypadku wyznaczonej trajektorii przegubowej

```
T = dwuwahadlo.fkine(q);

figure;hold;
    subplot(3,1,1)
    plot(t, squeeze(T(1,4,:)))
    xlabel('Time (s)');
    ylabel('X (m)')
    subplot(3,1,2)
    plot(t, squeeze(T(2,4,:)))
    xlabel('Time (s)');
    ylabel('Y (m)')
    subplot(3,1,3)
    plot(t, squeeze(T(3,4,:)))
    xlabel('Time (s)');
    ylabel('Z (m)')
 hold;
```

Ruch realizowany przez efektor w przypadku podanej trajektorii przegubowej przedstawiony jest na wykresie 2.3.

Klasa `SerialLink` wyposażona jest w metodę `fdyn`, która całkuje równania dynamiki manipulatora na danym interwale czasowym. Korzysta przy tym z funkcji numerycznego całkowania `ode45`. Zwraca wektor czasu oraz macierze położeń przegubów i prędkości przegubów. Poniższy kod realizuje symulację oraz animację ruchu manipulatora pod wpły-



Rysunek 2.4. Ruch realizowany pod wpływem siły grawitacji

wem siły grawitacji przy braku sterowania. Tworzy również wykresy przebiegu trajektorii przegubów przedstawione na rysunku 2.4.

```
[t1 q1 qd1] = dwuwahadlo.fdyn(10, [], [0 pi/2]);
figure;hold;
subplot(2,1,1)
    plot(t1,q1(:,1))
    xlabel('Time (s)');
    ylabel('Joint 1 (rad)')
    subplot(2,1,2)
    plot(t1,q1(:,2))
    xlabel('Time (s)');
    ylabel('Joint 2 (rad)')
    hold;
  figure;hold;
  clf;
  dwuwahadlo.plot(q1);
  hold;
```

Użytkownik Robotics Toolbox może zdefiniować siłę sterującą, działającą na przeguby poprzez implementację funkcji `torqfun`. Nazwa tej funkcji podawana jest jako argument dla metody `fdyn`. Niestety w wersji 9 pakietu metoda `fdyn` zawiera błąd, który uniemożliwia prawidłową realizację tego zadania [6]. Wspomniany błąd nie występował we wcześniejszych dystrybucjach Robotics Toolbox i ma zostać usunięty w wersji 9.1. Poniżej

zaprezentowano jak mogłaby wyglądać funkcja użytkownika realizująca prosty regulator PD oraz jak powinno zostać zrealizowane wywołanie jej w programie głównym.

Wydruk 2.1. Program główny

```
global P D qt;
P=[50 200];
D=[0 0];
t=t';
[q qd] = jtraj(qz, qr, t);
qt=[t q qd];
clear t;
clear q;
clear qd;
[T q1 qd1] = dwuwahadlo.fdyn(2,torqfun);
figure;hold;
subplot(2,1,1)
    plot(T,q1(:,1))
    xlabel('Time (s)');
    ylabel('Joint 1 (rad)')
    subplot(2,1,2)
    plot(T,q1(:,2))
    xlabel('Time (s)');
    ylabel('Joint 2 (rad)')
    hold;
  figure;hold;
  clf;
  dwuwahadlo.plot(q1);
  hold;
```

Wydruk 2.2. Funkcja torqfun

```
function  tau  = torqfun(T, q,qd )
global P D qt;
if(T>qt(length(qt),1))
    T=qt(length(qt),1);
end
%interpolacja wymaganych kątów dla tego punktu czasowego
q_dmd=interp1(qt(:,1),qt(:,2:3),T);
qprim_dmd=interp1(qt(:,1),qt(:,4:5),T);
%oblicz blad i moment sterujący
e=q_dmd-q;
eprim=qprim_dmd-qd;
tau=e*diag(P)+eprim*diag(D)
end
```

W celu zapoznania się z dokładnym opisem zaprezentowanych funkcji odsyłamy czytelnika do [1, 2].

2.2.3. Wady i zalety

Za zaletę Robotics Toolbox uznać należy fakt, że do przeprowadzania symulacji nie jest konieczne ręczne wyprowadzanie całego modelu manipulatora. Na korzyść pakietu działa również prostota implementacji manipulatora. Kod wykorzystywanych mplików jest ogólnodostępny, można zatem wprowadzać modyfikacje i usprawnienia, bądź czerpać inspirację z zastosowanych rozwiązań przy implementacji własnych plików.

Niestety w dokumentacji Robotics Toolbox pojawiają się błędy i niejasności. Przykładem może być opis do metody `jtraj` klasy `SerialLink`. Argumenty przyjmowane przez tą metodę to odpowiednio początkowe i końcowe położenia przegubów manipulatora, podczas gdy w dokumentacji błędnie napisano, że są to położenia początkowe i końcowe efektora.

Największą wadą Robotics Toolbox są pojawiające się błędy w kodzie. W podrozdziale 2.2.2 opisano problem z metodą `fdyn`. Pakiet nie jest produktem komercyjnym w związku z czym reakcja autora na zauważone niedociągnięcia przebiega ze znacznym opóźnieniem.

2.3. Implementacja własnych mplików

Modele dowolnych układów dynamicznych, w szczególności modele kinematyki oraz dynamiki kołowych robotów mobilnych, to układy równań różniczkowych. Jednym z możliwych sposobów rozwiązania/symulacji zachowania układu dynamicznego w środowisku MATLAB jest stworzenie funkcji, zawierającej zapis tego układu równań w notacji MATLABa, następnie rozwiązanie układu równań przy użyciu jednej z zaimplementowanych w MATLABie procedur `ode`. Procedury `ode` realizują cyfrowe algorytmy rozwiązywania układów równań różniczkowych pierwszego rzędu, w oparciu o metody całkowania numerycznego [3].

Przeznaczony do rozwiązania model obiektu należy przekształcić do następującej postaci:

$$\begin{bmatrix} \dot{x}_1 \\ . \\ . \\ . \\ \dot{x}_n \end{bmatrix} = \begin{bmatrix} f_1(t, x_1, .., x_n) \\ . \\ . \\ . \\ f_n(t, x_1, .., x_n) \end{bmatrix}, \tag{2.2}$$

gdzie:
— x_i – i-ta zmienna stanu,
— t – zmienna niezależna (czas), może oznaczać zależność od wymuszenia $u(t)$.
Następnie równania stanu w formie 2.2 należy zapisać w funkcji, zwracającej wektor pochodnych zmiennych stanu. W tym celu należy utworzyć mplik funkcyjny o nazwie takiej jak funkcja w nim zaimplementowana. Zawartość takiego pliku powinna wyglądać w następujący sposób:

```
function[dotx]=nazwa_funkcji(t,x)
2         dotx=[f1;...;fn];
end
```

Po zapisie równań obiektu w mpliku funkcyjnym można przystąpić do implementacji programu symulacyjnego. Kluczowym elementem tego programu jest zastosowanie jednej z procedur `ode`, których wywołanie ma następującą postać:

```
1 [T,X]=ode23('nazwa_funkcji',[tp,tk],x0,options)
```

Procedura `ode` zwraca w wyniku działania następujące zmienne:
— X – macierz zawierająca kolumny z wartościami rozwiązania,
— T – wektor kolumnowy zawierający chwile czasowe, w których określone są rozwiązania.
Argumenty procedury to odpowiednio:
— `'nazwa_funkcji'` – nazwa funkcji, która zawiera równania układu,
— $[t_p, t_k]$ – granice przedziału czasowego symulacji (lub tylko t_k jeżeli $t_p = 0$),
— x_0 – wektor warunków początkowych,
— *options* – nastawy parametrów procedury `ode` (opcjonalnie).
Alternatywnym sposobem wywołania funkcji `ode` jest:

```
2 [T,X]=ode113(@(t,x)nazwa_funkcji(t,x,inne_parametry),[tp,tk],
                                    x0,options)
```

Bieżące nastawy parametrów procedury `ode` można odczytać wpisując w oknie komend MATLABa polecenie `odeset`. Znaczenie poszczególnych parametrów opisane jest w dokumentacji programu MATLAB [4]. Aby zmienić wartość wybranych parametrów należy przed wywołaniem `ode` wykonać polecenie:

```
options=odeset('nazwa_parametru',nowa_wartość_parametru,..)
```

MATLAB dostarcza wiele procedur `ode` całkowania numerycznego, przykładowo:
1. Metody zmiennokrokowe:
   — `ode23`, `ode45` – zmodyfikowane metody Rungego-Kutty,
   — `ode113` – metoda Adamsa-Bashfortha-Moultona,
   — `ode15s`,`ode23s` – metoda NDFs i Rosenbrocka,
   — `ode23t` – metoda trapezowa,
   — `ode23tb` – metoda TR-BDF2 (połączenie metod trapezowej i wstecznego różniczkowania drugiego rzędu).
2. Metody stałokrokowe:
   — `ode1` – metoda Eulera,
   — `ode2` – metoda Heuna,
   — `ode3` – metoda Bogacki-Shampire,
   — `ode4` – metoda Rungego-Kutty 4. rzędu,
   — `ode5` – metoda Dormanda-Prince'a.

Opis działania wymienionych metod znaleźć można w dokumentacji pakietu [3].

Przed rozpoczęciem analizy przykładów warto jeszcze wspomnieć, że zmienne, z których korzysta zarówno program symulacyjny jak i mplik funkcyjny należy w obu plikach oznaczyć atrybutem `global`.

2.3.1. Implementacja robota mobilnego typu monocykl

Model kinematyki oraz dynamiki platformy mobilnej klasy $(2, 0)$

Pierwszym etapem pracy nad implementacją modelu platformy mobilnej klasy $(2, 0)$ w środowisku MATLAB jest wyprowadzenie równań kinematyki oraz dynamiki obiektu. Opis metodologii można znaleźć w [7].

Równania kinematyki w postaci bezdryfowego układu sterowania przyjmują postać

$$\begin{cases} \dot{x} = \cos\theta(\eta_1 + \eta_2), \\ \dot{y} = \sin\theta(\eta_1 + \eta_2), \\ \dot{\theta} = \frac{1}{l}(\eta_1 - \eta_2), \\ \dot{\varphi}_1 = \frac{2}{r}\eta_2, \\ \dot{\varphi}_2 = \frac{2}{r}\eta_1. \end{cases} \tag{2.3}$$

Równania dynamiki wyrażone we współrzędnych pomocniczych przedstawione są poniżej.

$$\begin{bmatrix} M_c + \frac{I_p}{l^2} + \frac{4I_{xx}}{r^2} & M_c - \frac{I_p}{l^2} \\ M_c - \frac{I_p}{l^2} & M_c + \frac{I_p}{l^2} + \frac{4I_{xx}}{r^2} \end{bmatrix} \cdot \dot{\eta} = \begin{bmatrix} 0 & \frac{2}{r} \\ \frac{2}{r} & 0 \end{bmatrix} \cdot u. \tag{2.4}$$

Wyjaśnienie oznaczeń w powyższych wzorach:
— l – połowa szerokości wózka,
— r – promień koła,
— M_c – masa całego obiektu,
— $(x, y, \theta, \varphi_1, \varphi_2)^T$ - wektor zmiennych stanu,
— $\eta = (\eta_1, \eta_2)^T$ – wektor sterowań kinematyką pojazdu,
— $u = (u_1, u_2)^T$ – wektor sterowań dynamiką pojazdu,
— I_p – moment bezwładności robota względem osi OZ,
— I_{xx} – moment bezwładności koła względem jego osi OX.

Implementacja modelu kinematyki

Kinematyka w postaci bezdryfowego układu sterowania jest zapisana zgodnie z 2.2 i nie wymaga żadnych przekształceń. Plik funkcyjny zawierający implementację modelu kinematyki robota mobilnego typu monocykl zamieszczony jest poniżej.

```
function qdot=monocykl1(t,q)

global l r;
eta=wymuszenie1(t);

qdot=zeros(5,1);
x=q(1);
y=q(2);
teta=q(3);
fi1=q(4);
fi2=q(5);
qdot(1)=cos(teta)*(eta(1)+eta(2));
qdot(2)=sin(teta)*(eta(1)+eta(2));
qdot(3)=1/l*(eta(1)-eta(2));
qdot(4)=2/r*eta(2);
qdot(5)=2/r*eta(1);
end
```

Sterowanie na poziomie kinematyki wyznaczane jest według prostej formuły zawartej w kolejnym pliku funkcyjnym. Treść pliku jest następująca:

```
function [eta] =wymuszenie1(t)
```

```
    eta=[t 1];
end
```

Plik symulacyjny zawierający wywołanie procedury `ode`, inicjuje zmienne oraz wizualizuje przebieg trajektorii ruchu robota. Kod pliku symulacyjnego wygląda jak następuje:

```
clear;

global l r;
l=1;
r=0.5;

t0=0;
tf=100;
q0=[0 0 0 0 0];
sol=ode113(@(t,q)monocykl1(t,q),[t0 tf],q0);
%oprócz t, q mogą być inne argumenty

x = 0:0.01:100;

q(:,1) = deval(sol,x,1);
q(:,2) = deval(sol,x,2);

plot(q(:,1),q(:,2));
grid;
xlabel('x');ylabel('y');
```

Funkcja `deval` pozwala na reprezentację uzyskanych wyników na wykresach ze zwiększoną dokładnością. Opis sposobu jej użycia znajduje się w dokumentacji MATLABa.

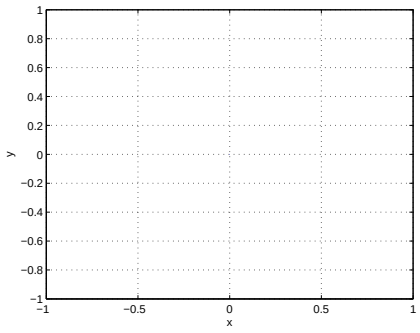
Aby sprawdzić poprawność implementacji modelu należy podać na wejścia określone wymuszenia, dla których łatwo przewidzieć sposób zachowania układu. Przykładowo wiadomo, że przy sterowaniu zerowym monocykl powinien pozostać nieruchomy, natomiast przy sterowaniu niezerowym na jednym z kół, robot powinien poruszać się po okręgu. Przebiegi trajektorii dla przeprowadzonych eksperymentów przedstawiono na wykresach 2.5–2.8. Wyniki te pokrywają się z oczekiwaniami, w związku z czym wykonany model należy uznać za prawidłowy.

Model dynamiki

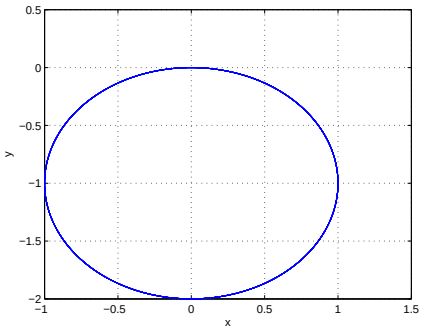
Równania dynamiki obiektu po przekształceniu do postaci 2.2 są następujące

$$\begin{cases} \dot{\eta_1} = \frac{1}{a-\frac{b^2}{a}} \cdot (\frac{2}{r}u_2 - \frac{2b}{ra}u_1), \\ \dot{\eta_2} = \frac{2}{ra}u_1 - \frac{b}{a^2-b_2}(\frac{2}{r}u_2 - \frac{2b}{ra}u_1), \end{cases} \tag{2.5}$$

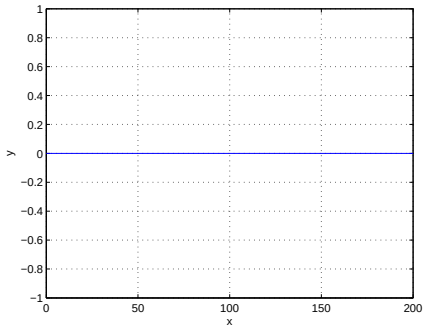
gdzie $a = M_c + \frac{I_p}{l^2} + \frac{4I_{xx}}{r^2}$, $b = M_c - \frac{I_p}{l^2}$. Dla sterowania dynamiką obiektu wejścia do systemu to $u = (u_1, u_2)^T$. Wejścia te kształtują przebiegi zmiennych $\eta = (\eta_1, \eta_2)^T$, które z kolei stanowią wejścia do kinematyki obiektu. Implementacja modelu dynamiki robota sprowadza się zatem do modyfikacji pliku funkcyjnego przez rozszerzenie zapisanego w nim układu równań o dwa dodatkowe równania różniczkowe wynikające z modelu dynamiki robota mobilnego typu monocykl.



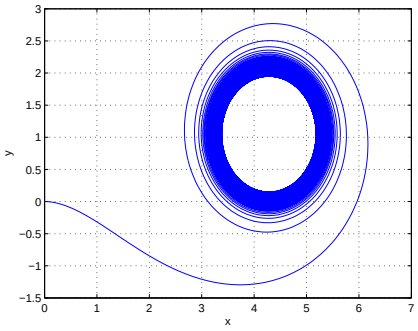
Rysunek 2.5. Ruch realizowany pod wpływem wymuszenia $\eta = [0 \quad 0]$ – robot nieruchomy



Rysunek 2.6. Ruch realizowany pod wpływem wymuszenia $\eta = [0 \quad 1]$



Rysunek 2.7. Ruch realizowany pod wpływem wymuszenia $\eta = [1 \quad 1]$



Rysunek 2.8. Ruch realizowany pod wpływem wymuszenia $\eta = [t \quad 1]$

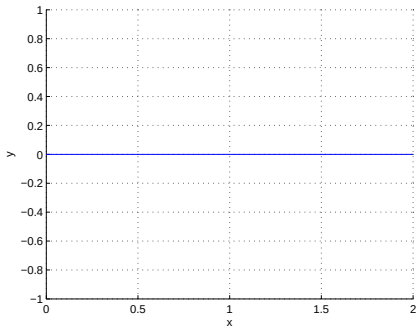
```
function zm_dot=monocykl(t,zm, u1,u2)

global l r a b;

qdot=zeros(5,1);
x=zm(1);
y=zm(2);
teta=zm(3);
fi1=zm(4);
fi2=zm(5);
eta1=zm(6);
eta2=zm(7);
eta_dot=zeros(2,1);
zm_dot=[qdot; eta_dot];

zm_dot(1)=cos(zm(3))*(zm(6)+zm(7));
zm_dot(2)=sin(zm(3))*(zm(6)+zm(7));
zm_dot(3)=1/l*(zm(6)-zm(7));
```

```
zm_dot(4)=2/r*zm(7);
zm_dot(5)=2/r*zm(6);
zm_dot(6)=1/(a-b^2/a)*(2/r*u2-2*b/(r*a)*u1);
zm_dot(7)=2/(r*a)*u1-b/(a^2-b^2)*(2/r*u2-2*b/(r*a)*u1);

end
```

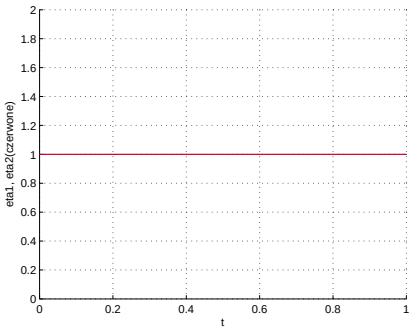
Plik symulacyjny przedstawiono niżej.

```
clear;

global l r a b;
l=1;
r=0.5;
Mc=10;
Ip=0.5;
Ixx=0.2;
a=Mc+Ip/(l*l)+4*Ixx/(r*r);
b=Mc-Ip/(l*l);
u1=0;
u2=10;

t0=0;
tf=1;
zm0=[0 0 0 0 0 1 1];
sol=ode113(@(t,zm)monocykl(t,zm,u1,u2),[t0 tf],zm0);
t = 0:0.001:tf;
zm(:,1) = deval(sol,t,1);
zm(:,2) = deval(sol,t,2);
zm(:,6) = deval(sol,t,6);
zm(:,7) = deval(sol,t,7);


figure;
hold;
plot(zm(:,1),zm(:,2));
grid;
xlabel('x');ylabel('y');
hold;

figure; hold;
plot(t,zm(:,6));
plot(t,zm(:,7),'r');
grid;
xlabel('t');ylabel('eta1, eta2(czerwone)');
hold;
```

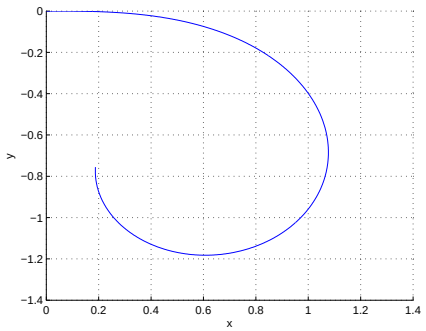
Sprawdzenie poprawności implementacji modelu dynamiki wymaga podobnie jak w przypadku modelu kinematyki podania odpowiedniego wymuszenia (tym razem jest nim wektor u). Przykładowe przebiegi trajektorii oraz wejść do kinematyki układu przedstawiono



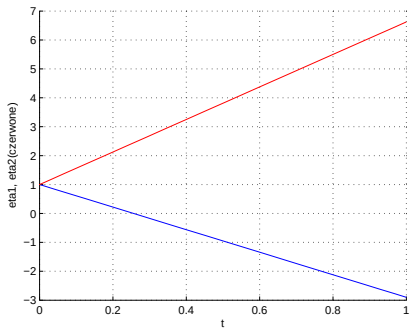
Rysunek 2.9. Ruch realizowany pod wpływem wymuszenia $u = [0 \quad 0]$



Rysunek 2.10. Zmiana η pod wpływem wymuszenia $u = [0 \quad 0]$



Rysunek 2.11. Ruch realizowany pod wpływem wymuszenia $u = [10 \quad 0]$



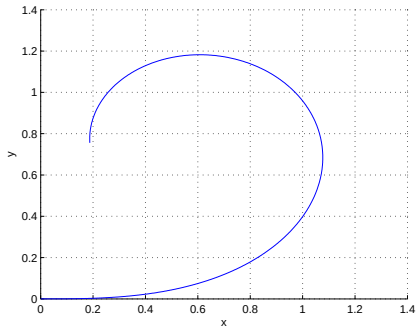
Rysunek 2.12. Zmiana η pod wpływem wymuszenia $u = [10 \quad 0]$

na wykresach 2.9–2.16. Wyniki pokrywają się z oczekiwaniami. W przypadku zastosowania sterowania $u = (0, 0)^T$ i przy warunku początkowym $\eta_1 = \eta_2 \neq 0$ robot porusza się wzdłuż linii prostej zgodnej z jego początkową orientacją. Przy sterowaniu niezerowym poprawność zachowania łatwiej jest określić analizując zachowanie zmiennych η_1, η_2. Na podstawie równań dynamiki można wywnioskować, że gdy u_2 jest większe niż u_1, η_2 powinno maleć, natomiast η_1 rosnąć. Sytuacja jest odwrotna gdy $u_2 < u_1$.

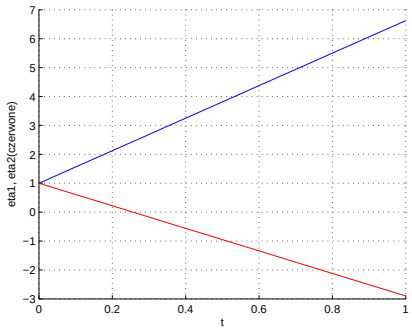
2.3.2. Wady i zalety

Za wadę modelowania obiektów przy użyciu mplików uznać można konieczność przekształcenia modelu do postaci 2.2. W przypadku braku odpowiednich zabezpieczeń może to doprowadzić do próby dzielenia przez zero. Ponadto w przypadku bardziej skomplikowanych modeli kod staje się mało czytelny.

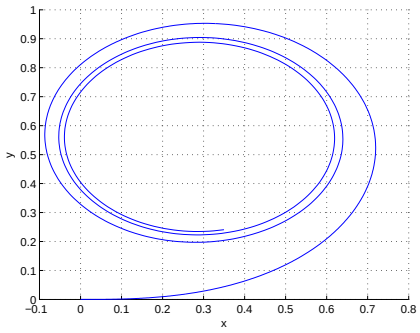
Ogromną zaletą jest możliwość prostej wizualizacji uzyskanych wyników. Co więcej ważna jest możliwość ingerencji w ustawienia funkcji `ode`. W razie potrzeby można bez problemu zwiększyć chociażby dokładność przeprowadzanych obliczeń.



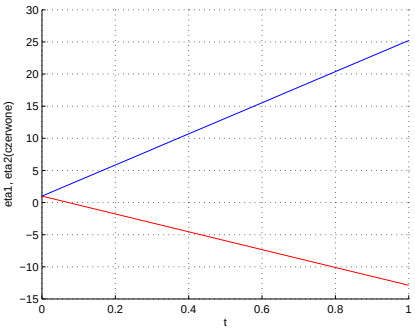
Rysunek 2.13. Ruch realizowany pod wpływem wymuszenia $u = [0 \quad 10]$



Rysunek 2.14. Zmiana η pod wpływem wymuszenia $u = [0 \quad 10]$



Rysunek 2.15. Ruch realizowany pod wpływem wymuszenia $u = [10 \quad 50]$



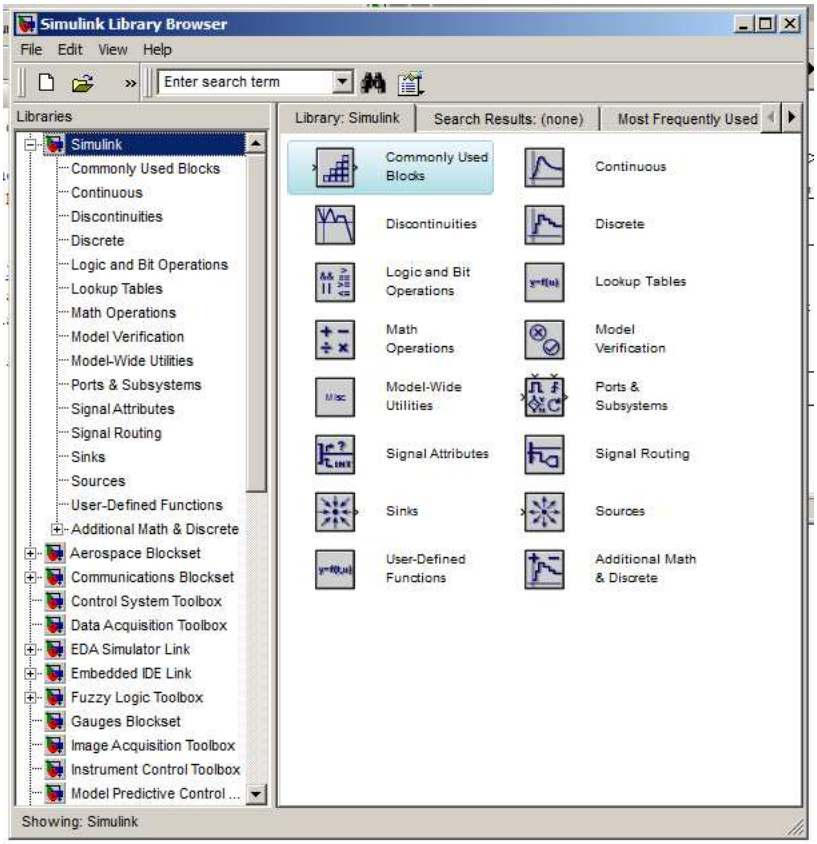
Rysunek 2.16. Zmiana η pod wpływem wymuszenia $u = [10 \quad 50]$

2.4. Simulink

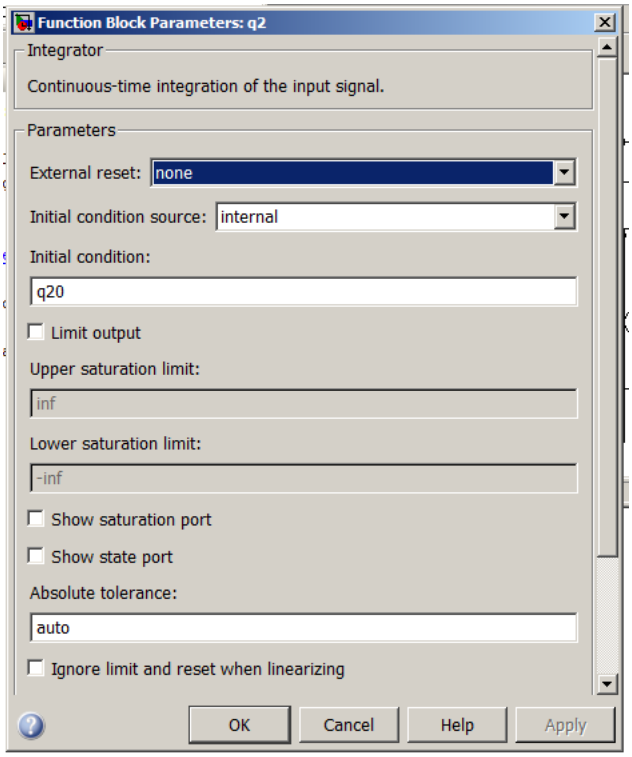
Jak napisano w [3] Simulink jest cyfrową realizacją koncepcji komputera analogowego. Informację o strukturze i parametrach układu dynamicznego wprowadza się do systemu liczącego w sposób graficzny, poprzez schemat blokowy wykonany w oknie edycyjnym Simulinka [3, 5]. Schemat blokowy składa się z pobranych z bibliotek Simulinka bloczków połączonych za pośrednictwem ich wejść i wyjść. Opis bloczków dostępny jest w dokumentacji MATLAB'a [4] oraz w wielu pozycjach wydawniczych [3, 5].

Simulink zapewnia możliwość implementacji zarówno obiektów opisanych równaniami liniowymi jak i nieliniowymi, w sposób ciągły, dyskretny oraz mieszany, układów deterministycznych i losowych.

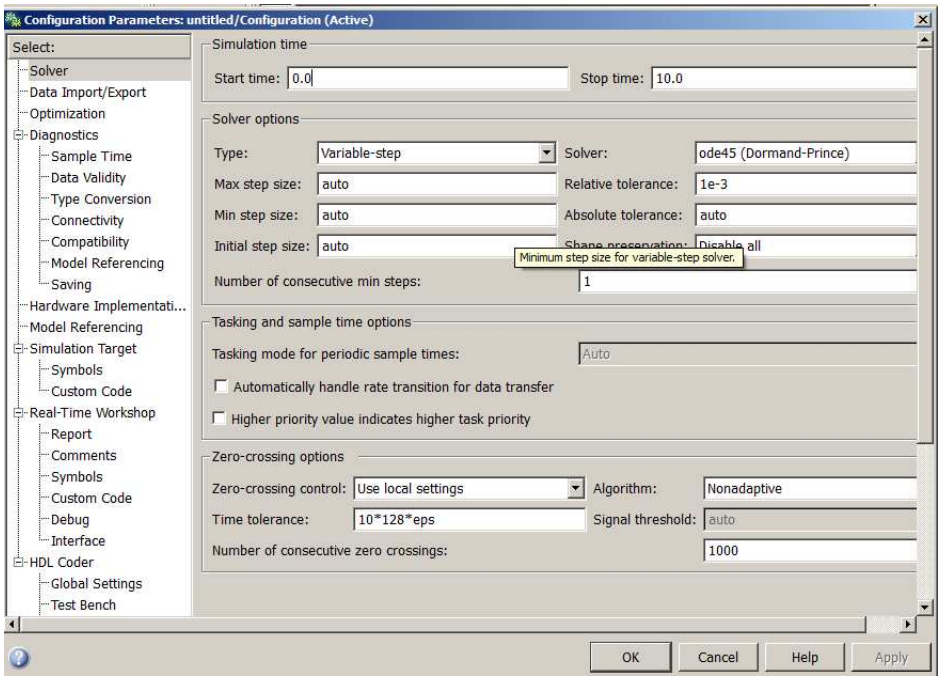
Dostępne biblioteki widoczne są w oknie `Simulink Library Browser` przedstawionym na rycinie 2.17. Bloczki wchodzące w skład danej biblioteki można dodawać do schematu przez przeciągnięcie ich przy użyciu myszy do okna edycji modelu. Każdy bloczek posiada okno dialogowe (`Block Parameters`), umożliwiające zmianę jego parametrów i ustawień. Przykładowe okno (`Block Parameters`) widoczne jest na rysunku 2.18. Wewnątrz schematu blokowego można używać zmiennych, przy czym każdorazowo przed uruchomieniem symulacji zmiennym tym należy przypisać wartości w oknie komend MATLAB'a. Istnieją bloki umożliwiające wprowadzanie oraz wyprowadzanie sygnałów z przeprowadzonych symulacji.



Rysunek 2.17. `Simulink Library Browser`



Rysunek 2.18. `Block Parameters`



Rysunek 2.19. `Configuration Parameters`

Rozpoczęcie obliczeń należy poprzedzić ustawieniem pożądanych parametrów symulacji w oknie `Configuration Parameters`, dostępnym w menu `Simulation` (rysunek 2.19).

Model wykonany w Simulinku można uruchomić z poziomu okna komend MATLAB'a lub z mpliku. W przypadku wykorzystania mpliku program steruje symulacją układu ze schematu blokowego, może definiować wymuszenia i przetwarzać uzyskane wyniki obliczeń. Uruchomienie symulacji realizuje poniższa linia kodu.

```
1 [T,X,Y]=sim('model',[tp,tk],options, ut)
```

Wyjaśnienie oznaczeń:
— $[T, X, Y]$ – wektor czasu T, macierz zmiennych stanu X, macierz sygnałów wyjściowych,
— $[t_p, t_k]$ – interwał czasowy, na którym poszukiwane jest rozwiązanie,
— `options` - nastawy parametrów symulacji, ustawiane tak jak w przypadku procedur `ode`,
— u_t – deklaracja sygnału wejściowego.

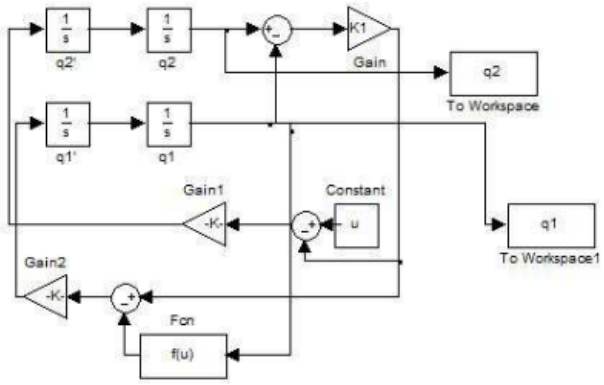
2.4.1. Implementacja manipulatora o pojedynczym elastycznym przegubie

Model dynamiki

Podobnie jak w przypadku implementacji modelu robota w mpliku, tworzenie modelu obiektu w Simulinku wymaga wyprowadzenia jego pełnego opisu matematycznego.

Równania dynamiki manipulatora o elastycznych przegubach z jednym ogniwem są następujące

$$\begin{aligned} \frac{1}{3}m_1 l_1^2 \ddot{q}_1 + \frac{1}{2}m_1 g l_1 \cos q_1 + (q_1 - q_2)K_1 &= 0 \\ I_1 \ddot{q}_2 + (q_2 - q_1)K_1 &= u, \end{aligned} \tag{2.6}$$



Rysunek 2.20. Schemat blokowy

gdzie:
— q_1 – położenie przegubu,
— q_2 – położenie silnika,
— m_1 – masa ogniwa,
— l_1 – długość ogniwa,
— K_1 – współczynnik sprężystości,
— I_1 – bezwładność silnika,
— u – sterowanie momentami napędowymi.

Schemat blokowy modelu

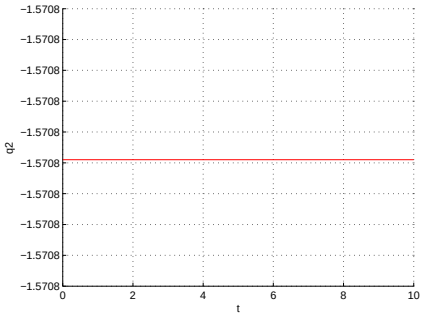
W celu implementacji równań dynamiki obiektu (2.6) w postaci schematu blokowego Simulinka należy zapisać je w następującej formie

$$\ddot{q}_1 = \frac{1}{\frac{1}{3}m_1 l_1^2}(-\frac{1}{2}m_1 g l_1 \cos q_1 + (q_2 - q_1)K_1) \tag{2.7}$$

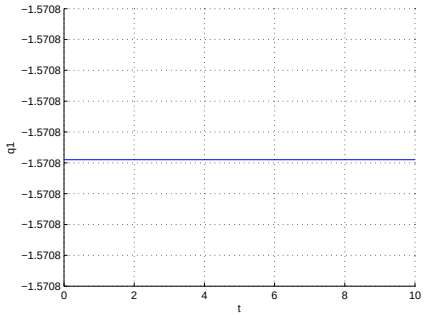
$$\ddot{q}_2 = \frac{1}{I_1}(u - (q_2 - q_1)K_1).$$

Następnie (2.7) wprost wyrażamy przez odpowiednie połączenie bloczków zgodnie z informacjami zawartymi w 2.4. Rezultat przedstawiony został na rysunku 2.20. Symulację modelu zrealizowano z wykorzystaniem mpliku.

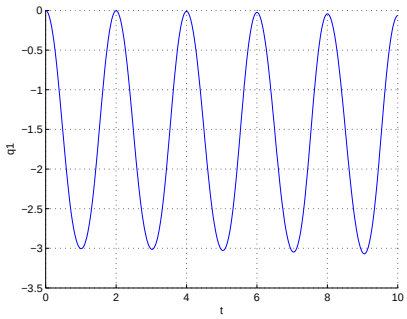
```
clear;

q20=-pi/2;
q10=-pi/2;

m1=10;
l1=1;
g=9.81;
I1=0.265;
K1=10;
u=0;
tk=10;

[t,Q]=sim('elastyczny', tk);
```



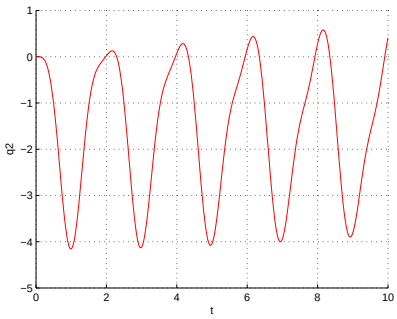
Rysunek 2.21. Ruch zmiennej q_1 przy warunku początkowym $q_1(0) = q_2(0) = -\frac{\pi}{2}$



Rysunek 2.22. Ruch zmiennej q_1 przy warunku początkowym $q_1(0) = q_2(0) = -\frac{\pi}{2}$



Rysunek 2.23. Ruch zmiennej q_1 przy warunku początkowym $q_1(0) = q_2(0) = 0$



Rysunek 2.24. Ruch zmiennej q_2 przy warunku początkowym $q_1(0) = q_2(0) = 0$

```
figure; hold;
plot(t, q1);
grid;
xlabel('t'); ylabel('q1');
hold;

figure; hold;
plot(t, q2,'r');
grid;
xlabel('t'); ylabel('q2');
hold;
```

Sprawdzenie poprawności modelu można zrealizować zmieniając warunki początkowe symulacji przy braku sterowania. Przykładowo gdy brak jest różnicy w położeniu przegubów q_1, q_2 i manipulator znajduje się w punkcie równowagi, obiekt powinien pozostawać w pełni nieruchomy (wykresy 2.22–2.21). Przy początkowym braku przesunięcia w przegubach, gdy manipulator znajduje się w punkcie przestrzeni przegubowej $q_1(0) = q_2(0) = 0$, spodziewamy się, że zmienna q_1 będzie oscylować między 0 a $-\pi$ i wystąpi przesunięcie w ruchu obu zmiennych stanu (wykresy 2.23–2.24).

2.4.2. Wady i zalety

Największą zaletą Simulinka jest przejrzysta i komunikatywna forma reprezentacji modelu matematycznego. Kod w języku MATLABa omówiony w poprzednim podrozdziale jest znacznie mniej czytelny niż schemat blokowy. Co więcej Simulink daje możliwość konwersji schematu do kodu w C/C++ oraz kodu w C/C++ do schematu w Simulinku. Mathworks zapewnia łatwą współpracę modeli Simulinkowych z innymi programami komercyjnym np. programami Adams i Inventor. Uzyskane wyniki podobnie jak w przypadku Robotics Toolbox oraz mplików mogą być wizualizowane w prosty sposób. Możliwa jest także praca z układem robotycznym w czasie rzeczywistym.

Rozwiązaniem wartym uwagi jest możliwość łączenia fragmentów kodu napisanych w języku programowania MATLABa ze schematami Simulinka (funkcje użytkownika można zaimplementować lub wywołać wewnątrz bloczku).

Konieczność ręcznego wyprowadzenia pełnego modelu opisywanego obiektu jest największą wadą Simulinka. Graficzna reprezentacja modelu matemtycznego, staję się wadą dla bardzo złożonych modeli. Model składa się z licznych bloków i połączeń między nimi, przez co traci ona na swojej przejrzystości i znacznie utrudnia wykrycie błędów implementacyjnych.

Bibliografia

[1] P. Corke. Robotic Toolbox for MATLAB Release 3, 1996.
[2] P. Corke. Robotic Toolbox for MATLAB Release 9, 2011.
[3] R. Klempka, A. Stankiewicz. *Modelowanie i symulacja układów dynamicznych*. AGH, 2006.
[4] Matlab documentation. http://www.mathworks.com/products/.
[5] W. Regel. *Przykłady i ćwiczenia w programie Simulink*. MIKOM, 2004.
[6] Forum robotics toolbox. http://groups.google.com/group/robotics-tool-box.
[7] K. Tchoń, A. Mazur, I. Dulęba, R. Hossa, R. Muszyński. *Manipulatory i roboty mobilne*. Akademiczna Oficyna Wydawnicza PLJ, 2000.

3. Microsoft Robotics Developer Studio

Microsoft Robotics Developer Studio R3 (MRDS R3) [2] jest środowiskiem programistycznym zorientowanym na zastosowania robotyczne. Umożliwia programowanie algorytmów oraz ich testowanie zarówno w środowisku symulacyjnym jak i na obiekcie rzeczywistym. Obiektem badań mogą być roboty i ich modele wykonane przez znanych producentów (np. Kuka) jak również przygotowane własnoręcznie.

Twórcy tego oprogramowania szczególną uwagę zwrócili na kwestię asynchroniczności działania systemów robotycznych. W omawianym środowisku, tak jak w rzeczywistych robotach, sygnały takie jak odczyty z sensorów mogą być przekazywane do sterownika w niezdefiniowanych chwilach czasu. Zaimplementowanie takiego podejścia wymagało stworzenia systemu opartego na współbieżnej, wielowątkowej pracy.

Pierwszą elementem środowiska MRDS wspierającym programowanie wielowątkowe jest biblioteka Concurrency and Coordination Runtime (CCR). Zapewnia ona możliwość uruchamiania pewnych fragmentów kodu niezależnie. Komunikację zapewnia mechanizm wiadomości i portów. Oznacza to, że nasza metoda wywołana niezależnie może wysłać wiadomość do portu (stanowiącego rodzaj kolejki). Będzie ona tam przechowywana aż do momentu odczytania jej przez metodę zwaną odbiorcą.

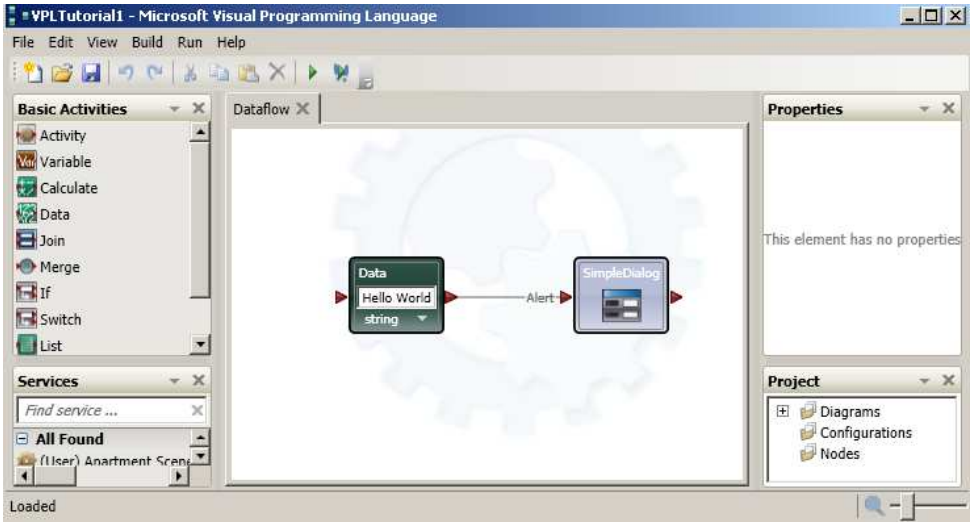
Decentralized Software Services (DSS) jest biblioteką implementującą mechanizm usług. Usługa posiada swój stan oraz można na niej wykonywać operacje. Może ona również wysyłać wiadomości do innych usług lub powiadomienia o zmianie stanu. Bardzo istotną cechą jest możliwość uruchomienia współpracujących ze sobą usług w różnych procesach lub nawet na różnych komputerach. Biblioteka CCR nie posiada takiej funkcjonalności, pozwala na komunikację wyłącznie wewnątrz procesu.

Elementem zapewniającym symulacje w czasie rzeczywistym jest Visual Simulation Environment (VSE). Jest on oparty na technologii NVIDIA PhysX, więc wykonywane przez niego obliczenia mogą być wspierane sprzętowo. Środowisko symulacyjne zapewnia również trójwymiarową wizualizację podczas której można ingerować w położenie obiektów oraz sprawdzić stan uruchomionych usług.

Ostatnim modułem jest Visual Programming Language. Zapewnia on możliwość graficznego tworzenia oprogramowania na podstawie gotowych usług. Jest to opcja skierowana głównie dla osób, które nie zajmują się robotyką profesjonalnie, lecz również w przypadku skomplikowanych projektów może okazać się bardzo przydatna, pomagając oszczędzić wiele monotonnej pracy. Przykładowa aplikacja napisana w VPL została przedstawiona na rysunku 3.1.

3.1. Zakres stosowalności

Środowisko Microsoft Robotics Developer Studio ze względu na swoją formę zestawu bibliotek posiada nieograniczoną wręcz możliwość rozbudowy, teoretycznie więc zakres zastosowań również jest szeroki. Jednak korzystanie z rozwiązań jawnie wspieranych jest o wiele prostsze, natomiast próby implementacji elementów nie przewidzianych przez twórców są znacznie utrudnione przez braki w dokumentacji.



Rysunek 3.1. Program "Hello World" w języku VPL

Zastosowaniem, w którym MRDS jest bardzo przystępnym narzędziem jest programowanie gotowych robotów zamodelowanych przez ich producentów. W tym przypadku wystarczy zaimplementować sterowanie robota, testując całość w środowisku symulacyjnym. Po osiągnięciu zamierzonych rezultatów można wgrać tak stworzony sterownik do pamięci robota lub sterować nim bezprzewodowo nie dokonując żadnej zmiany w kodzie.

Zadaniem, w którym omawiane środowisko byłoby niezwykle użyteczne jest badanie algorytmów planowania ścieżki, omijania przeszkód itp. Środowisko posiada zaimplementowane gotowe sensory, można również symulować szumy pomiarowe. Bardzo ciekawą opcją, która wspiera projektowanie systemów wizyjnych jest możliwość zamiany symulowanej kamery na rzeczywistą kamerę podłączoną do komputera. Warto podkreślić, że taka zamiana również nie wymaga żadnej ingerencji w kod programu.

Trudność tworzenia własnych modeli zależy od konkretnego przypadku. Dobrze udokumentowane i wspierane gotowymi rozwiązaniami są kołowe roboty mobilne oraz manipulatory modelujące jedynie kinematykę. Trudniejsze jest modelowanie manipulatorów uwzględniające dynamikę, natomiast aby otrzymać realistycznie zachowującego się robota latającego należałoby prawdopodobnie ręcznie opisać działające na niego siły. Niestety, dokumentacja części oprogramowania obejmująca bezpośrednią ingerencję w symulator fizyki nie jest opracowana.

Możliwości wykorzystania środowiska MRDS jako narzędzia do testowania algorytmów dynamiki są niewielkie, głównie ze względu na niską dokładność numeryczną wykonywanych obliczeń. Badanie np. błędu śledzenia trajektorii jest więc niemożliwe do wykonania. Można jednak taki algorytm zastosować w celu sprawdzenia poprawności modelu w sterowniku, lub aby skrócić procedurę testowania na rzeczywistym robocie.

Omówione środowisko nie posiada wbudowanej możliwości prezentowania danych z symulacji w postaci wykresów. Jednak z poziomu programisty mamy dostęp do wszystkich danych opisujących badane modele w dowolnym czasie. Dane te mogą być wyświetlane np. przy pomocy biblioteki `Microsoft Chart Controls` lub zapisywane do pliku i wyświetlane przez zewnętrzne oprogramowanie.

Podsumowując, Microsoft Robotics Developer Studio jest środowiskiem które bardzo ułatwia pracę nad apektami typowo związanymi ze sprzętem, potrafi realistycznie odwzorować peryferia robota. Może być to szczególnie ważne podczas prac projektowch (wybór i rozmieszczenie sensorów), a co ważne wspiera migrację kodu z symulatora do rzeczywitego

robta. Omawiane środowisko nie powinno być wykorzystywane w zastosowaniach typowo matematycznych głównie ze względu na niewystarczającą dokładność.

3.2. Implementacja własnych aplikacji

3.2.1. Konstrukcja aplikacji

Aby stworzyć model robota w środowisku MRDS należy napisać własną aplikację, korzystającą z bibliotek CCR oraz DSS. Realizacja tego zadania wymaga pewnej wiedzy dotyczącej założeń na temat funkcjonowania programów, zaszytych w Microsoft Robotics Developer Studio. W tym podrozdziale postaramy się przybliżyć czytelnikowi podstawowe intuicje.

Aplikacja w MRDS nazywana jest aplikacją DSS i składa się z wielu niezależnych usług (z angielskiego *services*). Każda usługa posiada swój stan powiązany z określonymi typami wiadomości, które otrzymuje, nazywanych operacjami (z angielskiego *operations*). Kiedy usługa otrzymuje wiadomość może zmienić swój stan i wysłać inne wiadomości i zawiadomienia (ang. *notifications*) do pozostałych usług. Stan usługi można uzyskać w programie przez wysłanie do niej wiadomości typu `Get`, można go też podejrzeć używając przeglądarki internetowej. Usługi mogą subskrybować (ang. *subscribe*) aby były zawiadomienie gdy zmieni się stan innej usługi lub kiedy w symulacji wystąpi jakieś inne zdarzenie. Usługi mogą też wchodzić w związki partnerskie (ang. *partner*), tak by móc wysyłać do siebie nawzajem wiadomości i otrzymywać odpowiedzi.

Usługa to klasa, która dziedziczy klasę bazową `DssServiceBase`. Składają się na nią cztery główne części:

Kontrakt (ang. *Contract*) definiuje jakie wiadomości można wysyłać do usługi oraz podaje unikalny identyfikator usługi (ang.*contract identifier*) w formie URI.

Stan (ang. *Internal State*) informacje, które usługa utrzymuje aby kontrolować swoje własne działanie.

Zachowania (ang. *Behaviors*) zestaw operacji (wiadomości), które usługa może obsłużyć i które są zaimplementowane w handlerach (funkcjach obsługi).

Kontekst wykonania (ang. *Execution Context*) związki partnerskie z innymi usługami oraz stan początkowy usługi.

Łączenie usług jako partnerów w aplikacje nazywane jest koordynacją (ang *orchestration*). Aplikacja DSS nie posiada pliku `main`. Usługi, które tworzą aplikację oraz wzajemne relacje między nimi podane są w pliku XML, nazywanym manifestem.

Microsoft Robotics Developer Studio instaluje wtyczkę do Microsoft Visual Studio. Tworzenie własnej usługi w środowisku Microsoft Visual Studio 2008 Professional Edition sprowadza się do wyboru jako typ projektu Dss Service. Automatycznie generowane są pliki, które zawierają szablon kodu do uzupełnienia.

Service.cs Główny plik usługi.

ServiceTypes.cs Plik typów usługi. Zawiera między innymi typy wiadomości jakie usługa może odbierać. Domyślnie są to:
- `Lookup` – podgląd stanu usługi,
- `Drop` – usunięcie usługi,
- `Get` – pobranie stanu usługi.

Zazwyczaj nie wymaga modyfikacji.

Service.manifest.xml Manifest usługi. W przypadku gdy na aplikację składa się więcej usług niż jedna tworzone są one w osobnych projektach. Jeden z projektów wybierany jest na projekt główny i to jego manifest pełni funkcję pliku startowego.

3.2.2. Modelowanie środowiska

Modelowanie systemów robotycznych w MRDS należy rozpocząć od zdefiniowania środowiska. W przykładach wykonanych na potrzeby projektu środowisko składa się z kamery, nieba oraz gruntu. Podrozdział ten opisuje w jaki sposób otrzymano taki efekt.

Miejscem, w którym należy deklarować poszczególne elementy jest przeciążona metoda `Start` nowo utworzonej usługi DSS. W opracowanych przykładach składa się ona z wywołania funkcji bazowej oraz metod odpowiedzialnych za utworzenie kamery oraz dodanie pozostałych elementów. Poniżej opisane kroki zostały zaimplementowane w pliku `DoublePendulum.cs`.

Metoda `SetupCamera` (wydruk 3.1) tworzy nową kamerę, ustawia jej położenie i orientacje, a następnie dodaje całość do środowiska symulacyjnego. Co ważne, można dodać wiele kamer np. śledzących różne roboty, a w trakcie symulacji przełączać między nimi. Metoda `AddSky` (wydruk 3.2) ustawia kształt (można wybrać sferę lub półsferę) oraz teksturę nieba. W drugiej części tej metody ustawiane jest oświetlenie sceny. Wszystkie te elementy również muszą zostać dodane do środowiska symulacyjnego przy użyciu metody `GlobalInstancePort.Insert`.

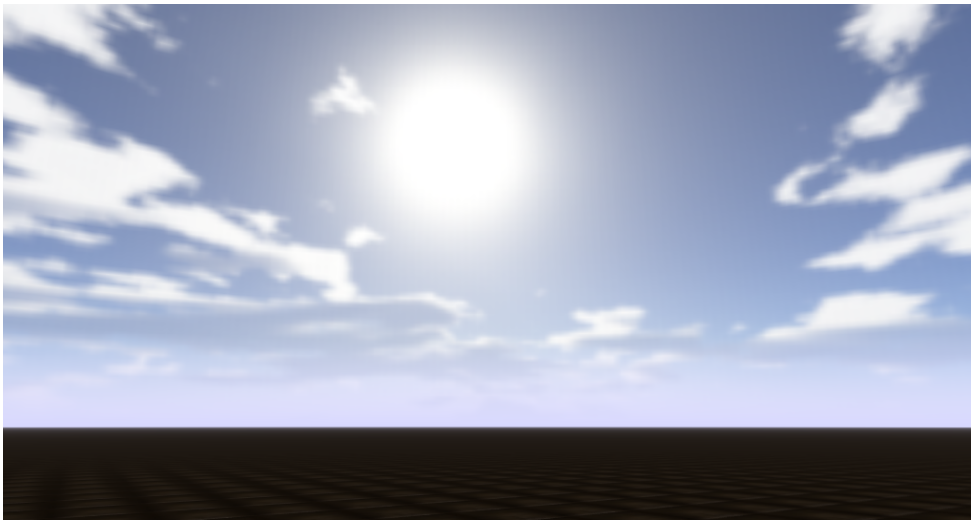
Wydruk 3.1. Metoda SetupCamera

```
  private void SetupCamera()
2       {
            CameraView view = new CameraView();
4           view.EyePosition = new Vector3(1.0f, 1.0f, 1.0f);
            view.LookAtPoint = new Vector3(0.0f, 0.0f, 0.0f);
6           SimulationEngine.GlobalInstancePort.Update(view);
        }
```

Wydruk 3.2. Metoda AddSky

```
1 void AddSky()
        {
3           SkyDomeEntity sky = new SkyDomeEntity(
                        "skydome.dds", "sky_diff.dds");
5           SimulationEngine.GlobalInstancePort.Insert(sky);

7           LightSourceEntity sun = new LightSourceEntity();
            sun.State.Name = "Sun";
9           sun.Type = LightSourceEntityType.Directional;
            sun.Color = new Vector4(0.8f, 0.8f, 0.8f, 1);
11          sun.Direction = new Vector3(0.5f, -.75f, 0.5f);
            SimulationEngine.GlobalInstancePort.Insert(sun);
13      }
```

W wykonanym przykładzie został wykorzystany najprostszy rodzaj gruntu: płaski, nieskończony dywan. Można jednak zastosować bardziej skomplikowane podłoża np. wczytane z pliku jako siatka trójkątów lub z pliku graficznego w którym odcień ze skali szarości będzie odpowiadał wysokości w danym punkcie. Zdefiniowano również materiał z którego stworzone jest podłoże. Całość została zaimplementowana w metodzie `AddGround` (wydruk 3.3).



Rysunek 3.2. Puste środowisko symulacyjne w MRDS

Wydruk 3.3. Metoda AddGround

```
HeightFieldEntity AddGround()
{
    HeightFieldEntity ground = new HeightFieldEntity(
        "simple ground",
        "floor.bmp",
        new MaterialProperties(...);
        );
    SimulationEngine.GlobalInstancePort.Insert(ground);

    return ground;
}
```

Uzyskane w ten sposób środowisko zostało zaprezentowane na rysunku 3.2.

3.2.3. Modelowanie dwuwahadła

W środowisku MRDS obiekty dodawane do symulacji muszą dziedziczyć po klasie `VisualEntity`. Z naszego punktu widzenia najistotniejsze są dwie metody: `Initialize` oraz `Update`. W pierwszej z nich inicjalizujemy nasz obiekt np. dodając do niego nowe elementy, zmieniając ich położenie itp. Metoda `Update` jest natomiast wywoływana w każdym kroku symulacji. To w niej możemy zadać siły, momenty, prędkości oraz położenia. Opisywany model znajduje się w pliku `DoublePendulumEntity.cs`.

Zamodelowane dwuwahadło składa się z następujących elementów: prostopadłościennej podstawy oraz dwóch ramion w kształcie kapsuł. Każde ramie zawiera również przegub. Tak skonstruowane ogniwo zaimplementowano w klasie `SingleShapesegmentEntity`. Oprócz wspomnianego przegubu klasa ta zawiera metody niezbędne do jego odtworzenia po procesie serializacji do pliku XML, które wywoływane są w metodzie `Initialize`. Sposób rozwiązania tego problemu zaczerpnięto z przykładu "TestBench" z książki [1].

Pierwszym elementem dodawanym do naszego dwuwahadła jest podstawa. Jest ona obiektem klasy `BoxShape` z przypisanymi jej właściwościami, takimi jak: nazwa, masa, położenie, rozmiar oraz materiał. Tak przygotowany element musi zostać dodany do stanu

naszego modelu (metoda `base.State.PhysicsPrimitives.Add`). Sposób dodawania tego elementu został przedstawiony na wydruku 3.4.

Wydruk 3.4. Podstawa dwuwahadła

```
BoxShapeProperties groundDesc = new BoxShapeProperties(
    "chassis",
    10000,
    new Pose(),
    new Vector3(1.0f,0.1f,1.0f));
    groundDesc.Material = new MaterialProperties(
                        "groundMaterial", 0.0f, 0.5f, 0.5f);
    BoxShape ground = new BoxShape(groundDesc);
    ground.State.Name = "GroundShape";
    base.State.PhysicsPrimitives.Add(ground);
```

Kolejnym etapem jest dodanie ogniw. W tym celu tworzymy obiekty klasy `SingleShapesegmentEntity`. W następnym etapie należy odpowiednio ustawić właściwości wbudowanego w każde ogniwo przegubu. Domyślnie wszystkie stopnie swobody są zablokowane. Naszym zadaniem jest odblokowanie stopnia swobody związanego z obrotem wokół jednej z osi. Robimy to tworząc obiekt klasy `JointAngularProperties`, a następnie ustawiając właściwość `TwistMode` na wartość `Free`. Proces ten został zaprezentowany na wydruku 3.5.

Wydruk 3.5. Ustawienie rodzaju przegubu

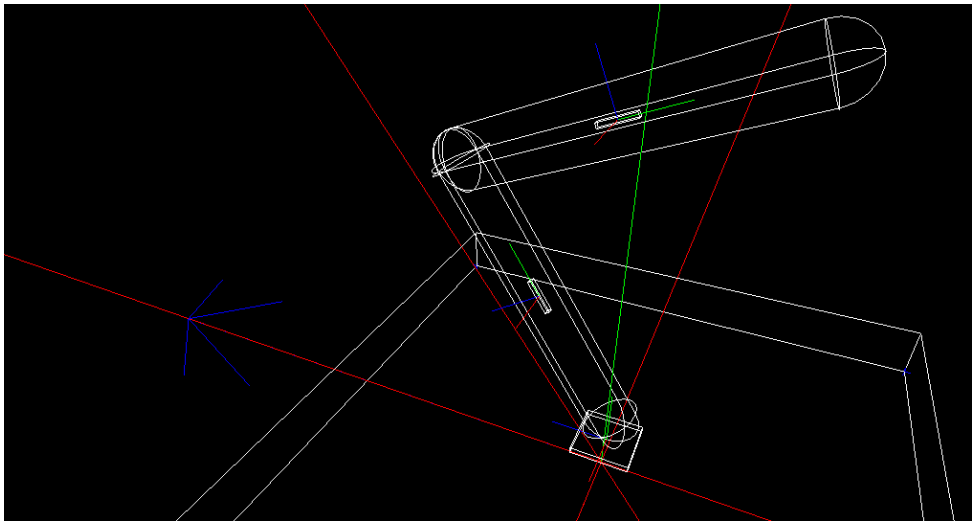
```
JointAngularProperties angular;
EntityJointConnector[] connectors;
name = "1DOF";
segment1 = Newsegment(position + new Vector3(0, 0, 0), name);
angular = new JointAngularProperties();
angular.TwistMode = JointDOFMode.Free;
```

Każdy przegub musi zostać skojarzony z dwoma bryłami które ma łączyć. W tym celu należy stworzyć dwuelementową tablicę obiektów `EntityJointConnector`. Każdy z elemtów tej tablicy przechowuje informacje o łączonym obiekcie, układzie współrzędnych połączenia oraz o punkcie połączenia (wydruk 3.6).

Wydruk 3.6. Definiowanie połączeń

```
connectors = new EntityJointConnector[2];
connectors[0] = new EntityJointConnector(
segment1,
new Vector3(0, 1, 0),
new Vector3(1, 0, 0),
new Vector3(0,
        -segment1.CapsuleShape.State.Dimensions.Y / 2.0f, 0));

 connectors[1] = new EntityJointConnector(...);
```

Ostatnią zmianą w domyślnych parametrach jest wyłączenie wykrywania kolizji w łączonych obiektach. W naszym przypadku oś obrotu znajduje się wewnątrz łączonych elementów, wykorzystanie tej opcji jest więc konieczne. Dodanie tak przygotowanego ogniwa



Rysunek 3.3. Model dwuwahadła w środowisku MRDS

odbywa się przy pomocy metody `InsertEntity`. Gotowy model został zaprezentowany na rysunku 3.3. Szczegółowy opis sposobu modelowania manipulatorów tego typu znajduje się w pozycji [1].

3.2.4. Modelowanie elastycznego przegubu

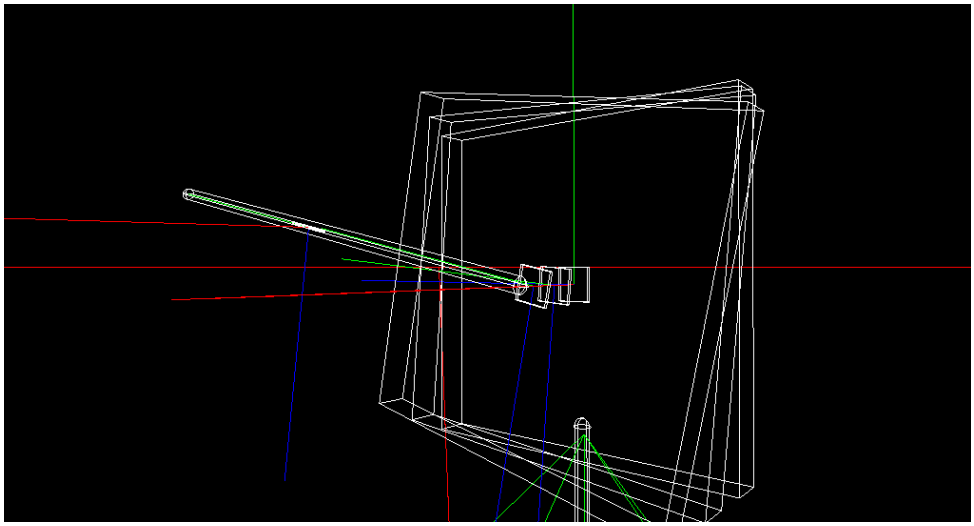
Modelowanie elastycznego przegubu okazało się znacznie trudniejsze niż dwuwahadła ponieważ twórcy środowiska MRDS nie dostarczają w tym przypadku gotowego rozwiązania. Aby zrealizować to zadanie konieczne okazało się znaczne skomplikowanie modelu oraz ingerencja w metodę `Update`.

Pierwszym elementem tego modelu jest podstawa do której na sztywno połączono pierwsze ramię. Następnie dodawane są trzy prostopadłościany: pierwszy jest na sztywno połączony z pierwszym ramieniem, drugi pełni rolę ramienia o zerowej długości, trzeci jest natomiast połączony na sztywno z drugim ramieniem. Zadeklarowano dwa połączenia obrotowe: pomiędzy pierwszym i środkowym prostopadłościanem oraz pomiędzy środkowym i trzecim prostopadłościanem. Sposób deklarowania elementów i połączeń między nimi jest analogiczny do opisanego w rozdziale 3.2.3.

Idea działania jest następująca: do środkowego prostopadłościanu mamy możliwość przyłożenia zewnętrznego momentu. Natomiast na trzeci prostopadłościan działa moment o wartości proporcjonalnej do różnicy położenia kątowego między nim a drugim prostopadłościanem. Na drugi prostopadłościan działa również moment o identycznej wartości lecz przeciwnym zwrocie. Powyższe rozumowanie zastało zaimplementowane w metodzie `Update` i przedstawione na wydruku 3.7. Przy niewielkich wychyleniach błędy numeryczne skutkowały drganiem całego obiektu, w takim przypadku elastyczność jest więc dezaktywowana.

Wydruk 3.7. Implementacja elastyczności w przegubie

```
1  public override void Update(FrameUpdate update)
{
3   float angle=MoveDegrees(auxiliarySegment.RotationAngles.X\
                          - flexibleSegment.RotationAngles.X);
5   if (angle> 2.0f)
     {
```



Rysunek 3.4. Model elastycznego przegubu w środowisku MRDS

```
7         if (auxiliarySegment!=null && flexibleSegment!=null)
           {
9            auxiliarySegment.PhysicsEntity.ApplyTorque(
                    new Vector3(torque- K * angle, 0, 0), true);
11           flexibleSegment.PhysicsEntity.ApplyTorque(
                    new Vector3((K * angle), 0, 0), true);
13         }
      }
15    else
         {
17         if (auxiliarySegment != null)
            {
19            auxiliarySegment.PhysicsEntity.ApplyTorque(
                    new Vector3(torque, 0, 0), true);
21          }
         }
23       PhysicsEntity.UpdateState(true);
         base.Update(update);
25  }
```

Gotowy model elastycznego przegubu został zaprezentowany na rysunku 3.4.

3.2.5. Implementacja robota typu monocykl

Wśród usług wyróżniamy tzw. usługi symulacyjne *simulation services*. Mogą one tworzyć obiekt w środowisku symulacyjnym, manipulować nim oraz odczytywać z niego dane. Usługa symulacyjna może zostać przypisana (subskrybować) do konkretnego obiektu za pomocą jego nazwy, wówczas otrzymuje powiadomienie kiedy obiekt ten zostanie wstawiony, zastąpiony lub usunięty. Simulation services muszą znajdować się w tym samym węźle DSS (procesie) co symulacja, wówczas w przypadku otrzymania powiadomienia o pojawieniu się obiektu w symulacji, odpowiedni port (standardowo nazywany `_notificationTarget`) zawiera referencję do wstawionego obiektu.

`SimulatedDifferentialDrive` to usługa symulacyjna dostarczana przez środowisko MRDS, która umożliwia interakcję z instancjami robotów dwukołowych sterowanych różnicowo w środowisku symulacyjnym. Wszystkie usługi, które chcą skomunikować się z napędem robota aby nim poruszyć czynią to poprzez usługę `SimulatedDifferentialDrive`. Przykładem jest Dashboard, okno dialogowe za pomocą którego można sterować ręcznie odpowiednio zaimplementowanym robotem.

Przez odpowiednio zaimplementowanego robota, rozumiemy obiekt klasy dziedziczącej po klasie `DifferentialDriveEntity` dostępnej w bibliotekach MRDS. Klasa ta zawiera implementacje wszystkich pól i metod reprezentujących napęd robota oraz umożliwiających kierowanie pojazdami dwukołowymi sterowanymi różnicowo. Warto zapoznać się z jej dokumentacją [2]. Dziedzicząc klasę `DifferentialDriveEntity` nie musimy nadpisywać wspomnianych metod. Musimy natomiast w naszej klasie pochodnej:

— zaimplementować konstruktor parametryzowany, określający wygląd robota oraz realizujący dodawanie czujników,
— nadpisać metodę `Initialize`.

Przykładową klasę, w szczególności treść konstruktora parametryzowanego przedstawia wydruk 3.8

Wydruk 3.8. Przykładowa klasa

```
[DataContract]
[DataMemberConstructor]
public class MojRobot : DifferentialDriveEntity
{
    public MojRobot() { }
    public MojRobot(Vector3 initialPos)
    {
        MASS = 9;
        CHASSIS_DIMENSIONS = new Vector3(0.393f, 0.18f, 0.40f);
        CHASSIS_CLEARANCE = 0.05f;
        FRONT_WHEEL_RADIUS = 0.08f;
        CASTER_WHEEL_RADIUS = 0.025f;
        FRONT_WHEEL_WIDTH = 4.74f;
        CASTER_WHEEL_WIDTH = 0.02f;
        FRONT_AXLE_DEPTH_OFFSET = -0.05f;

        base.State.Name = "ProstyRobocik";
        base.State.MassDensity.Mass = MASS;
        base.State.Pose.Position = initialPos;

        BoxShapeProperties motorBaseDesc =
        new BoxShapeProperties("chassis", MASS,
            new Pose(new Vector3(
            0,
            CHASSIS_CLEARANCE + CHASSIS_DIMENSIONS.Y / 2,
            0)),
            CHASSIS_DIMENSIONS);

        motorBaseDesc.Material = new MaterialProperties(
        "high_friction", 0.0f, 1.0f, 20.0f);
        motorBaseDesc.Name = "Chassis";
        ChassisShape = new BoxShape(motorBaseDesc);
```

```
34              CASTER_WHEEL_POSITION = new Vector3(0,
                    CASTER_WHEEL_RADIUS,
36                  CHASSIS_DIMENSIONS.Z / 2 - CASTER_WHEEL_RADIUS);

38              FRONT_WHEEL_MASS = 0.10f;

40              RIGHT_FRONT_WHEEL_POSITION = new Vector3(
                    CHASSIS_DIMENSIONS.X / 2 + 0.01f - 0.05f,
42                  FRONT_WHEEL_RADIUS,
                    FRONT_AXLE_DEPTH_OFFSET);
44
                LEFT_FRONT_WHEEL_POSITION = new Vector3(
46                  -CHASSIS_DIMENSIONS.X / 2 - 0.01f + 0.05f,
                    FRONT_WHEEL_RADIUS,
48                  FRONT_AXLE_DEPTH_OFFSET);

50              MotorTorqueScaling = 20;

52              //Robot wyposażony z przodu i z tyłu w dalmierze laserowe
                DalmierzLaserowyEntity ir1 =
54                new DalmierzLaserowyEntity(base.State.Name + "_rear",
                  new Pose(new  Vector3(initialPos.X,
56                CHASSIS_DIMENSIONS.Y / 2.0f
                      + CHASSIS_CLEARANCE + initialPos.Y,
58                    CHASSIS_DIMENSIONS.Z / 2.0f + initialPos.Z)));
                DalmierzLaserowyEntity ir2 =
60                new DalmierzLaserowyEntity(base.State.Name + "_front",
                  new Pose(new Vector3(initialPos.X,
62                CHASSIS_DIMENSIONS.Y / 2.0f
                      + CHASSIS_CLEARANCE + initialPos.Y,
64                    -CHASSIS_DIMENSIONS.Z / 2.0f + initialPos.Z),
                      TypeConversion.FromXNA(
66                  xna.Quaternion.CreateFromAxisAngle(
                            new xna.Vector3(0, 1, 0), (float)Math.PI))));
68              InsertEntityGlobal(ir1);
                InsertEntityGlobal(ir2);
70
                ConstructStateMembers();
72            }
    }
```

Ustawienia parametrów wyglądu robota (kształt, rozmiar) zostały zaczerpnięte z pliku `entities.cs` dostępnego w przykładach instalujących się wraz z MRDS.

Cała definicja klasy oznaczona jest dwoma atrybutami, które ułatwiają budowę infrastruktury dla usług. W ich miejsce kompilator MRDS generuje niezbędny kod ułatwiając tym samym pracę programiście.

— DataContract – stanowi informację, że obiekt rozważanej klasy to część stanu usługi, który musi podlegać serializacji przy przesyłaniu,

— DataMemberConstructor – informuje, że konstruktor parametryzowany tej klasy ma zostać zawarty w tworzonym pliku Proxy dll (szczegóły patrz podrozdział 3.3.3)

Wszelkie czujniki w jakie chcemy wyposażyć robota powinny być tworzone i dodawane

do monocykla na końcu konstruktora tak jak czujniki typu `DalmierzLaserowyEntity` w przykładzie 3.8.

Klasa `MojRobot` z wydruku 3.8 nie jest jeszcze kompletna - nadpisanie metody `Initialize` jest konieczne. Warto zaznaczyć, że można to zrealizować w klasie dziedziczącej z kolei po klasie `MojRobot`. `Initialize` dodaje instancję do środowiska fizycznego po kolejnym dodaniu podobiektów składowych. W ogólności można w niej również definiować wygląd robota. Poniżej przedstawiono treść procedury dla opisywanego monocykla.

```
using drive = Microsoft.Robotics.Services.Drive.Proxy;
//napęd rożnicowy
..
public override void Initialize(xnagrfx.GraphicsDevice device,
 PhysicsEngine physicsEngine)
        {
            try
            {
                base.ServiceContract = drive.Contract.Identifier;
                base.Initialize(device, physicsEngine);
            }
            catch (Exception e)
            {
                Console.WriteLine(e.ToString());
            }
        }
```

W powyższym fragmencie kodu istotna jest linia `base.ServiceContract = drive.Contract.Identifier;`, która sprawia, że robot ma możliwość jazdy.

Robota do środowiska symulacyjnego wstawiamy w metodzie `Start()` usługi zawierającej definicję klasy robota następującym poleceniem

```
MotorBaseWithDrive Rob =
  new MotorBaseWithDrive(
      new Vector3(6, 0.229f, 30f * 6f / 100f + 1f));
SimulationEngine.GlobalInstancePort.Insert(Rob);
```

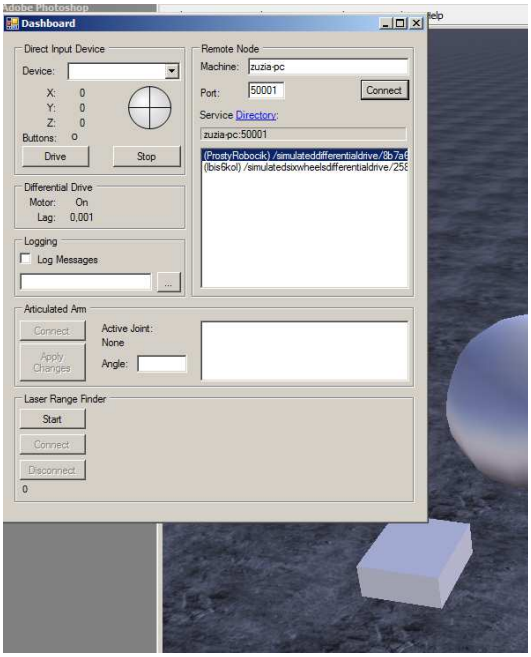
Aby mieć możliwość sterowania utworzonym monocyklem należy uruchomić `SimulatedDifferentialDrive` w głównym manifeście projektu, ustawiając go od razu na konkretny obiekt. Zrealizowano to w poniższym fragmencie kodu manifestu.

```
  <ServiceRecordType>
    <dssp:Contract>http://schemas.microsoft.com/.../simulateddifferentialdrive.html</dssp:Contract>
    <dssp:PartnerList>
      <dssp:Partner>
        <dssp:Service>http://localhost/ProstyRobocik</dssp:Service>
        <dssp:Name>simcommon:Entity</dssp:Name>
      </dssp:Partner>
    </dssp:PartnerList>
    <Name>this:SimulatedGenericDifferentialDrive</Name>
  </ServiceRecordType>
```

Adres URI obiektu zawsze przybiera postać `http://localhost/nazwa_obiektu`, gdzie nazwa obiektu to pole ustawiane w konstruktorze parametrycznym `base.State.Name = "ProstyRobocik";`.

Do nagłówka manifestu należy dodać linię:

```
<Manifest ...
    xmlns:simulateddifferentialdrive=
    "http://schemas.microsoft.com/robotics
____/simulation/services/2006/05
```



Rysunek 3.5. Widok symulacji

```
____/simulateddifferentialdrive.html"
6 .. >
```

Poprawność wykonania powyższych działań łatwo przetestować korzystając ze wspomnianego wcześniej okna dialogowego Dashboard. W tym celu do głównego manifestu projektu należy dodać poniższy fragment

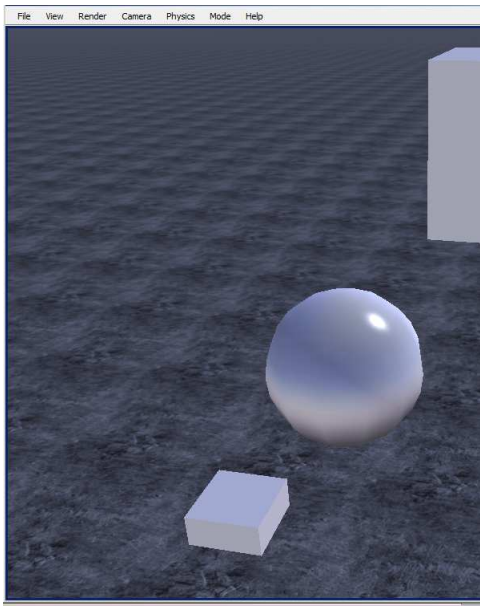
```
  <ServiceRecordType>
2   <dssp:Contract>
        http://schemas.microsoft.com/../simpledashboard.html
4   </dssp:Contract>
    <dssp:PartnerList />
6   <Name>this:SimpleDashboard</Name>
  </ServiceRecordType>
```

Jeżeli wszystko przebiegło prawidłowo po kompilacji i uruchomieniu programu powinno otworzyć się oprócz okna z symulacją, okno dialogowe Dashboard jak na rysunku 3.5. Po naciśnięciu przycisku Connect powinny pojawić się nazwy obiektów dostępnych do sterowania. Po dwukrotnym kliknięciu na wybranego robota oraz naciśnięciu przycisku Drive można sterować nim przy użyciu joysticka (lewa strona okna Dashboard).

3.2.6. Implementacja robota sterowanego różnicowo o większej liczbie kół

Robot sterowany różnicowo o większej liczbie kół zachowuje funkcjonalność `DifferentialDriveEntity`, lecz nie dziedziczy po niej z uwagi na występujące istotne różnice. Klasa robota tym razem dziedziczy klasę `VisualEntity` (opis w dokumentacji [3]), w związku z tym wszystkie niezbędne pola i metody do obsługi napędu pojazdu trzeba zaimplementować samodzielnie. W raporcie omówione i przytoczone zostaną jedynie zasady oraz najważniejsze fragmenty kodu, w celu poszerzenia wiedzy odsyłamy czytelnika do pozycji [1], gdzie przedstawione jest krok po kroku w jaki sposób zaimplementować robota o czterech kołach.

Rozważany jest sześciokołowy robot widoczny na rycinie 3.6 Na początek należy w za-



Rysunek 3.6. Widok symulacji

implantowanej klasie dodać i zainicjalizować pola statyczne, wykorzystywane przy określaniu wyglądu robota (jego masę, rozmiary nadwozia oraz kół, położenie kół, itp.). Koła robota to coś więcej niż zwykłe kształty. Są to konkretne instancje innej klasy, reprezentowane w symulacji fizyki środowiska. Dlatego dodajemy je jako pola składowe pojazdu.

```
WheelEntity _wheelFR;
WheelEntity _wheelFL;
WheelEntity _wheelRR;
WheelEntity _wheelRL;
WheelEntity _wheelMR;
WheelEntity _wheelML;
```

Dla każdego z nich w definicji klasy musi istnieć właściwość opatrzona oznaczeniami:
— Category("Wheels") – grupuje właściwości żeby były lepiej zorganizowane w oknie edycji symulatora,
— DataMember – informacja, że właściwość podlega serializacji przy przesyle do innych usług i zapisie na dysku.

```
[Category("Wheels")]
[DataMember]
public WheelEntity FrontRightWheel
{
    get { return _wheelFR; }
    set { _wheelFR = value; }
}
```

Koła oraz podwozie są tworzone w metodzie `Initialize`. Ustawienie obiektów podawane jest względem ustawienia instancji robota. Następnie w tej samej metodzie obiekty są dodawane do listy instancji fizycznych poleceniem
`base.State.PhysicsPrimitives.Add(chassis)`. Dla kół przebiega to nieco inaczej:

```
_wheelFR = new WheelEntity(wheelFRprop);
    _wheelFR.State.Name = base.State.Name +
                                "FrontRightWheel";
    _wheelFR.Parent = this;
    _wheelFR.Initialize(device, physicsEngine);
```

Obiekt nie jest jeszcze kompletny, ponieważ brak jest obsługi napędów. Konieczne należy dodać do klasy robota co następuje:

— `bool _isEnabled` – pole decyduje czy napęd robota jest włączony, inne usługi mogą zmieniać jego wartość za pośrednictwem odpowiedniej właściwości,
— `float _motorTorqueScaling` – skaluje prędkość obrotów do symulacji,
— `public float CurrentHeading` – rotacja pojazdu względem osi Y.

Dla każdego pola należy zaimplementować odpowiednią właściwość.

Należy zaimplementować konstruktor parametryzowany. W nim zadawane jest początkowe położenie robota, nadawana jest nazwa robota, dodawane są czujniki oraz zadawane jest skalowanie prędkości obrotów `_motorTorqueScaling`.

Pozostałe metody do zaimplementowania, odpowiadające odpowiednim metodom z `DifferentialDriveEntity`.

```
public void SetMotorTorque(float leftWheel,
 float rightWheel)
public void SetVelocity(float value)
public void SetVelocity(float left, float right)
private void SetAxleVelocity(float left, float right)
float ValidateWheelVelocity(float value)
public void DriveDistance(float distance,
 float power, SuccessFailurePort responsePort)
public void RotateDegrees(float degrees,
 float power, SuccessFailurePort responsePort)
void ResetRotationAndDistance()
```

Powyższe umożliwiają bezpośrednią kontrolę nad prędkością każdego z kół oraz przemieszczenie i zmianę orientacji robota (dwie ostatnie metody). Ich implementacja nie zostanie przytoczona ponieważ sprowadza się do rozszerzenia odpowiednich metod z `DifferentialDriveEntity`. Szczegóły można znaleźć w [1, 2].

Aktualizacja prędkości kół oraz położenia robota powinna być przeprowadzana w metodzie `Update`. W związku z tym należy nadpisać metodę dziedziczoną z `VisualEntity`. Istnieje wiele możliwych podejść do realizacji tego zadania [1].

Teraz można dodać robota do symulacji analogicznie jak było to realizowane w przypadku monocykla.

```
SimulationEngine.GlobalInstancePort.Insert(
new Ibis6kolEntity("Ibis6kol",
new Vector3(2.5f, 0f, 0.8f)));
```

Ważnym pojęciem które należy wprowadzić na tym etapie jest *Generic Contract*. Jest to typ podobny do abstrakcyjnej klasy. Różnica polega na tym, że nie posiada on własnej implementacji. Składa się tylko z definicji typów identyfikatora, stanu i głównego portu operacyjnego. Usługa implementująca *Generic Contract* nazywana jest usługą generic (ang. *generic service*). Usługa taka ma dwa porty operacyjne. Jeden z nich służy do obsługi tej usługi, drugi do otrzymywania komunikatów skierowanych do jej części związanej

z *Generic Contract*. Cel wprowadzenia typu *Generic Contract* to ujednolicenie obsługi podobnych obiektów oraz zapewnienie możliwości dynamicznego wiązania usług.

`SimulateDifferentialDrive` nie może zostać wykorzystane do sterowania stworzonym przez nas robotem sześciokołowym. Konieczne jest zaimplementowanie własnej usługi symulacyjnej, zapewniającej podobną funkcjonalność. Chcemy aby nowo tworzona usługa również mogła komunikować się z Dashboardem, w związku z czym nasza implementacja musi być zgodna z *Generic Contract* dla napędów, tak jak `SimulateDifferentialDrive`.

Dostęp do *Generic Contract* uzyskujemy dodając na początku pliku głównego oraz pliku typów usługi poniższą dyrektywę.

```
using pxdrive = Microsoft.Robotics.Services.Drive.Proxy;
```

W głównym pliku usługi musimy dodać port, który będzie otrzymywał ogólne komendy ruchu np. z Dashboard.

```
[AlternateServicePort(
            AllowMultipleInstances = true,
            AlternateContract = pxdrive.Contract.Identifier)]
private pxdrive.DriveOperations _diffDrivePort =
new Microsoft.Robotics.Services.Drive.Proxy.DriveOperations();
```

Ze względu na konieczność obsługi dodatkowych czterech kół należy zmodyfikować główny port oraz rozszerzyć stan:

```
[ServicePort("/SimulatedSixWheelsDifferentialDrive",
 AllowMultipleInstances = true)]
private SimulatedSixWheelsDifferentialDriveOperations
 _mainPort =
  new SimulatedSixWheelsDifferentialDriveOperations();

[InitialStatePartner(Optional = true,
ServiceUri =
"SimulatedSixWheelsDifferentialDriveService.Config.xml")]
private SimulatedSixWheelsDifferentialDriveState _state =
new SimulatedSixWheelsDifferentialDriveState();
```

Tworzona usługa symulacyjna wymaga ingerencji w plik typów. Jest tak z uwagi na konieczność obsługi stanu rozszerzonego do sześciu kół.

```
    [DataContract]
    public class SimulatedSixWheelsDifferentialDriveState
                : pxdrive.DriveDifferentialTwoWheelState
    {
        private pxmotor.WheeledMotorState _rearLeftWheel;
        private pxmotor.WheeledMotorState _rearRightWheel;
        private pxmotor.WheeledMotorState _middleLeftWheel;
        private pxmotor.WheeledMotorState _middleRightWheel;

        [DataMember]
        public pxmotor.WheeledMotorState RearLeftWheel
        {
```

```
            get { return _rearLeftWheel; }
            set { _rearLeftWheel = value; }
        }
        [DataMember]
        public pxmotor.WheeledMotorState RearRightWheel
        {
            get { return _rearRightWheel; }
            set { _rearRightWheel = value; }
        }
        [DataMember]
        public pxmotor.WheeledMotorState MiddleLeftWheel
        {
            get { return _middleLeftWheel; }
            set { _middleLeftWheel = value; }
        }
        [DataMember]
        public pxmotor.WheeledMotorState MiddleRightWheel
        {
            get { return _middleRightWheel; }
            set { _middleRightWheel = value; }
        }
    }
```

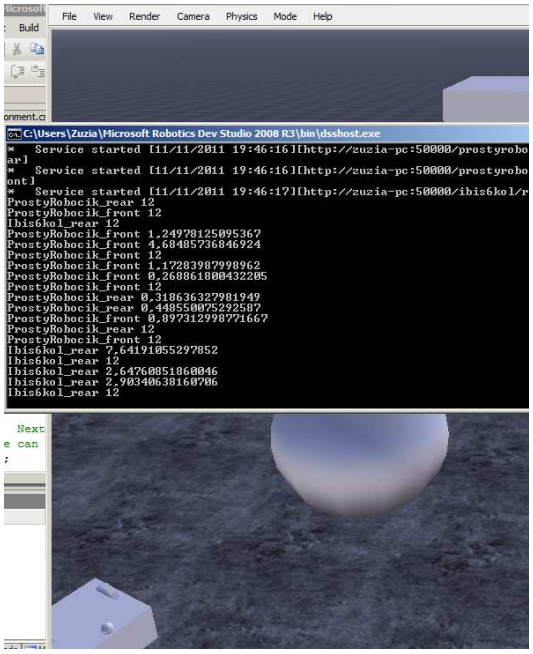
Funkcje sterujące, które trzeba zaimplementować w głównym pliku usługi:

```
public SimulatedSixWheelsDifferentialDriveService(DsspServiceCreationPort creationPort)
 : base(creationPort);
protected override void Start();
void CreateDefaultState();
void UpdateStateFromSimulation();
void InsertEntityNotificationHandlerFirstTime(simengine.InsertSimulationEntity ins);
void InsertEntityNotificationHandler(simengine.InsertSimulationEntity ins);
void DeleteEntityNotificationHandler(simengine.DeleteSimulationEntity del);
public IEnumerator<ITask> SubscribeHandler(pxdrive.Subscribe subscribe);
public IEnumerator<ITask> ReliableSubscribeHandler(pxdrive.ReliableSubscribe subscribe);
public IEnumerator<ITask> MainPortHttpGetHandler(dssphttp.HttpGet get);
public IEnumerator<ITask> MainPortGetHandler(Get get);
public void SetPoseHandler(SetPose setPose);
void SetPoseDeferred(corobot.Ibis6kolEntity entity, Pose pose);
public IEnumerator<ITask> HttpGetHandler(dssphttp.HttpGet get);
public IEnumerator<ITask> GetHandler(pxdrive.Get get);
public IEnumerator<ITask> DriveDistanceHandler(pxdrive.DriveDistance driveDistance);
public IEnumerator<ITask> RotateHandler(pxdrive.RotateDegrees rotate);
public IEnumerator<ITask> SetPowerHandler(pxdrive.SetDrivePower setPower);
public IEnumerator<ITask> SetSpeedHandler(pxdrive.SetDriveSpeed setSpeed);
public IEnumerator<ITask> EnableHandler(pxdrive.EnableDrive enable);
public IEnumerator<ITask> AllStopHandler(pxdrive.AllStop estop);
public IEnumerator<ITask> TestDriveDistanceAndRotateDegrees();
```

Szczegółowe informacje dotyczące realizacji zadania na przykładzie robota mobilnego czterokołowego znaleźć można w [1]. Kod zrealizowanej usługi dla robota sześciokołowego dostępny jest w dodatku 3.A.

Uruchomienie usługi w manifeście jest analogiczne do wcześniejszego przypadku:

```
<ServiceRecordType>
   <dssp:Contract>
            http://schemas.tempuri.org/../simulatedsixwheelsdifferentialdrive.html
   </dssp:Contract>
   <dssp:PartnerList>
     <dssp:Partner>
       <dssp:Service>http://localhost/Ibis6kol</dssp:Service>
       <dssp:Name>simcommon:Entity</dssp:Name>
     </dssp:Partner>
   </dssp:PartnerList>
   <Name>this:SixDifferentialDrive</Name>
 </ServiceRecordType>
```



Rysunek 3.7. Widok symulacji

3.2.7. Czujniki

Podobnie jak w przypadku robotów aby zaimplementować czujnik należy stworzyć klasę opisującą go oraz odpowiednią usługę symulacyjną. Można to łatwo zrealizować modyfikując przykłady instalujące się wraz z MRDS.

Do wcześniej opisanych robotów dodano sensory typu `DalmierzLaserowyEntity`. Za wzór do implementacji dalmierza posłużyły `CorobotIREntity` oraz `LaserRangeFiderEntity`. Działanie sensora widoczne jest na rycinie, przedstawiającej pracę środowiska symulacyjnego 3.7 . Czerwony punkt symbolizuje miejsce przecięcia promienia lasera podczerwonego z siatką fizyczna przedmiotów w symulacji. Warto zaznaczyć, ze siatka fizyczna przedmiotów to nie to samo co ich kształt w wizualizacji. Wynik pomiaru wypisywany jest na konsoli.

3.3. Podstawy programistyczne

3.3.1. Krótkie wprowadzenie do C#

Biblioteki składające się na Microsoft Robotics Developer Studio zostały napisane w języku C#, więc teoretycznie własne aplikacje można pisać w dowolnym języku wspieranym przez platformę NET. Praktycznie jednak inne języki nie posiadają wbudowanych mechanizmów znacznie ułatwiających pracę ze środowiskiem MRDS, wszystkie opracowane przez twórców przykłady również zostały napisane w C#. W rozdziale tym postanowiono więc przedstawić pewne techniki, które są przydatne w zrozumieniu i dalszym rozwijaniu aplikacji MRDS, a nie są znane osobom, które nigdy wcześniej nie pisały programów we wspomnianym języku.

Interfejsy

Interfejs w języku C# oznacza pewnego rodzaju „wzorzec" klasy. Zawiera on zbiór metod oraz właściwości. Mówimy, że klasa która ma zaimplementowane wszystkie metody oraz właściwości interfejsu implementuje dany interfejs. Przykładowy interfejs wraz z klasą go implementującą został przedstawiony na wydruku 3.9. Przed przystąpieniem do dalszej części warto zapoznać się z interfejsem IEnumerable.

Wydruk 3.9. Przykładowy interfejs i jego implementacja

```
public interface SampleInterface
    {
        int SomeProperty { get;}
        void SomeMethod(float someFloat);
    }
public class SampleClass : SampleInterface
    {
        public int SomeProperty
        {
            get{return 1;}
        }
        public void SomeMethod(float someFloat)
        {
            Console.WriteLine(someFloat.ToString())
        }
    }
```

Delegaty

Delegaty pełnią taką samą rolę jak wskaźniki na funkcje w C/C++ tzn. umożliwiają przekazywanie jednej funkcji jako argument do innej. Delegat również stanowi pewien „wzorzec", mianowicie ustala typ zwracanej wartości oraz argumenty przyjmowane przez funkcję. Przykładowy delegat został zdefiniowany i wykorzystany na wydruku 3.10. Metoda `SomeMethod` może zostać wywołana z metodami `method1` oraz `method2` ponieważ obie te metody zwracają wartość tego samego typu co delegeat `someDelegate` oraz przyjmują argumenty o takich samych typach.

Wydruk 3.10. Przykładowy delegat i jego zastosowanie

```
public class SomeClass
{
    public delegate int someDelegate(int arg1, int arg2);

    public SomeClass()
    {
        SomeMethod(method1);//wypisze 3
        SomeMethod(method2);//wypisze -1
    }

    public void SomeMethod(someDelegate method)
    {
        int value = method(1, 2);
```

```
14          Console.WriteLine(value.ToString());
        }
16
        public int method1(int arg1, int arg2)
18      {
            return arg1+arg2;
20      }
        public int method2(int arg1, int arg2)
22      {
            return arg1-arg2;
24      }
}
```

Anonimowe metody

Anonimowe metody pozwalają definiować funkcję w argumencie innej funkcji. Na wydruku 3.11 dwa pierwsze wywołania metody `SomeMethod` są równoważne wywołaniu z przykładu 3.10. Wywołanie trzecie pokazuje dodatkową funkcjonalność: wewnątrz anonimowej metody można wykorzystywać zmienne zadeklarowane poza tą metodą.

Wydruk 3.11. Przykładowy delegat i jego zastosowanie

```
1  public class SomeClass
   {
3      public delegate int someDelegate(int arg1, int arg2);

5      public SomeClass()
       {
7          SomeMethod(delegate(int arg1, int arg2)
           {
9              return arg1 + arg2;
           });//wypisze 3
11         SomeMethod(delegate(int arg1, int arg2)
           {
13             return arg1 - arg2;
           });//wypisze -1
15
           int outside = 1;
17         SomeMethod(delegate(int arg1, int arg2)
           {
19             return arg1 + arg2 + outside;
           });//wypisze 4
21     }

23     public void SomeMethod(someDelegate method)
       {
25         int value = method(1, 2);
           Console.WriteLine(value.ToString());
27     }
   }
```

3.3.2. Concurrency and Coordination Runtime

Concurrency and Coordination Runtime implementuje model programowania wielowątkowego, w którym poszczególne wątki komunikują się wyłącznie przy pomocy wiadomości. System ten zapewnia równoległe wykonywanie różnych zadań, ich koordynację oraz obsługę błędów.

Podstawą systemu komunikacji są wiadomości. Wiadomość może być dowolnego typu, lecz należy pamiętać, że do poprawnego działania aplikacji w sieci wymagana jest możliwość serializacji do formatu XML. Wiadomości mogą być wysyłane do portów. Port jest implementacja kolejki tj. wiadomość raz z niego odebrana jest usuwana. Każdy port może odbierać wiadomości tylko jednego typu. W przypadku potrzeby nasłuchiwania na wiadomości różnych typów, można zdefiniować zbiór portów (ang. PortSet).

Przetwarzać wiadomości mogą odbiorcy. Odbiorca jest implementowany jako metoda przyjmująca jako argument wiadomość o typie takim samym jak typ nasłuchiwanego portu. Nad prawidłowym przyporządkowaniem wiadomości i odbiorców czuwa struktura zwana arbitrem. Arbiter umożliwia łączenie portów i odbiorców na różne sposoby m.in. umożliwia ciągły nasłuch lub jednorazowe przesłanie wiadomości. Kolejną jego funkcjonalnością jest możliwość wykonywania operacji logicznych na portach. Możemy więc np. nakazać wykonanie danej metody dopiero gdy na dwa nasłuchiwane porty przyjdą wiadomości (logiczne "i") albo gdy wiadomość przyjdzie na jeden z nich (logiczne "lub").

Strukturą stojącą najwyżej w hierarchii jest zarządca. Zajmuje się on kolejkowaniem wszystkich wątków. Każda usługa DSS ma wbudowanego swojego zarządcę, możemy jednak stworzyć własnego. Ważną cechą zarządcy jest możliwość ustawienia liczby rdzeni, pod którą ma być optymalizowane kolejkowanie.

3.3.3. Decentralized Software Services

Biblioteki CCR dają możliwość aby niezależne fragmenty kodu przekazywały między sobą wiadomości w sposób asynchroniczny i były wykonywane równolegle w ramach jednego procesu. Biblioteki DSS rozszerzają tę koncepcję na wiele procesów, a nawet wiele maszyn.

Jak wynika z poprzednich podrozdziałów DSS odpowiada za podstawowe funkcje robotycznej aplikacji. Uruchamia i zatrzymuje usługi oraz obsługuje przepływ wiadomości między usługami przez porty usług (ang. *service forwarder ports*). DSS jest odpowiedzialny za:

— ładowanie ustawień usług,
— bezpieczeństwo (przekazywanie wiadomości pomiędzy usługami sprawia, że błąd może się pojawić w miejscu innym niż faktyczny problem, dlatego trzeba dokładnie nadzorować przepływ informacji w programie),
— pamiętanie jakie usługi są aktualnie uruchomione,
— kontrolę dostępu do plików i zasobów,
— interfejs użytkownika (np. dostęp do usług przez przeglądarkę internetową).

Za środowisko runtime dla MRDS odpowiada program nazywany DssHost.exe. Konkretna instancja działającego DssHost.exe nazywana jest węzłem DSS. Operacje na usługach są przeprowadzane przez wysyłanie wiadomości. Z uwagi na fakt, że komunikujące się ze sobą usługi mogą działać na różnych komputerach wiadomość musi podlegać przed przesłaniem serializacji do pliku XML i następnie przy odbiorze deserializacji. Realizację tego procesu umożliwiają pliki Proxy dll, tworzone automatycznie przy kompilacji przez program DssProxy.exe.

W celu szczegółowego zapoznania się z funkcjonalnościami zapewnianymi przez biblioteki DSS odsyłamy czytelnika do [1].

3.4. Wady i zalety

Microsoft Robotics Developer Studio wspiera wykonywanie projektów robotycznych w bardzo innowacyjny sposób. Środowisko to cechuje się bardzo wydajną, wielowątkową pracą, posada również możliwość rozproszenia obliczeń w sieci lokalnej. Bardzo praktyczna jest też możliwość wykonywania i obserwowania symulacji w czasie rzeczywistym. Wsród zalet wymienić należy również wsparcie producentów robotów, którzy udostępniają własne modele.

Niestety, oprogramowanie to jest ciągle w fazie wczesnego rozwoju. Wynikające z tego powodu braki w dokumentacji są szczególnie uciążliwe. Kolejną wadą, która wynika poniekąd z innowacyjności podejścia, jest problem z szybkim rozpoczęciem pracy z środowiskiem. Jest ono stosunkowo skomplikowane, a poznanie (chociaż powierzchowne) całej architektury jest niezbędne do rozpoczęcia pracy nad własnym projektem.

Sposób modelowania instancji fizycznych również należy uznać za wadę. Jest on niezwykle monotonny i pracochłonny, a twórcy oprogramowania nie udostępnili żadnego graficznego narzędzia, które ułatwiało by ten proces. Nie można też importować modeli z innego oprogramowania.

Mimo wymienionych wad środowisko MRDS jest warte polecenia, szczególnie w kontekście ciągłego jego rozwoju, a w związku z tym eliminacji niedociągnięć wynikających z wczesnej fazy rozwoju projektu. Pozwala ono modelować skomplikowane systemy robotyczne w realistycznie odwzorowywującym rzeczywistość środowisku symulacyjnym. Natomiast architektura, która początkowo wydaje się być zbyt skomplikowana, pomaga utrzymać logiczną strukturę nawet skomplikowanych projektów.

3.A. Dodatek: Wydruk programu

Wydruk 3.A. Usługa symulacyjna dla robota sześciokołowego (plik główny)

```
//SimulatedSixWheelsDifferentialDriveService umożliwia sterownaie instancją klasy Ibis6kolEntity
//między innymi za pomocą simple dashboard
using Microsoft.Ccr.Core;
using Microsoft.Dss.Core;
using Microsoft.Dss.Core.Attributes;
using Microsoft.Dss.ServiceModel.Dssp;
using Microsoft.Dss.ServiceModel.DsspServiceBase;
using Microsoft.Dss.Services.SubscriptionManager;
using W3C.Soap;

using System;
using System.Collections.Generic;
using System.ComponentModel;
using dssphttp = Microsoft.Dss.Core.DsspHttp;
using pxdrive = Microsoft.Robotics.Services.Drive.Proxy;
using xml = System.Xml;
using xna = Microsoft.Xna.Framework;
using submgr = Microsoft.Dss.Services.SubscriptionManager;
using simtypes = Microsoft.Robotics.Simulation;
using simengine = Microsoft.Robotics.Simulation.Engine;
using physics = Microsoft.Robotics.Simulation.Physics;
using corobot = Robot;
using Microsoft.Robotics.PhysicalModel;


namespace SimulatedSixWheelsDifferentialDrive
{
    [Contract(Contract.Identifier)]
    [AlternateContract(pxdrive.Contract.Identifier)]
    //zapewnia dostęp do simulated differential drive service
    [DisplayName("SimulatedSixWheelsDifferentialDrive")]
    [Description("SimulatedSixWheelsDifferentialDrive service (no description provided)")]
    //service obsługuje dwa porty identyfikowane różnymi kontraktami

    public class SimulatedSixWheelsDifferentialDriveService : DsspServiceBase
```

```
    {

        #region Simulation Variables
        corobot.Ibis6kolEntity _entity;
        //referencja do instancji robota sześciokołowego w środowisku symulacyjnym
        simengine.SimulationEnginePort _notificationTarget;//info czy pojawił się robot
        #endregion

        //subskrypcja
        [Partner("SubMgr", Contract = submgr.Contract.Identifier, CreationPolicy =
            PartnerCreationPolicy.CreateAlways, Optional = false)]
        private SubscriptionManagerPort _subMgrPort = new
            Microsoft.Dss.Services.SubscriptionManager.SubscriptionManagerPort();
        //otrzymuje komendy ruchu,identyczny jak z simulated differential drive
        [AlternateServicePort(
                    AllowMultipleInstances = true,
                    AlternateContract = pxdrive.Contract.Identifier)]

        private pxdrive.DriveOperations _diffDrivePort = new
            Microsoft.Robotics.Services.Drive.Proxy.DriveOperations();
        //główny port ma obsługiwać operacje rozszerzone do 6 kół, definicja w pliku typów
        [ServicePort("/SimulatedSixWheelsDifferentialDrive", AllowMultipleInstances = true)]
        private SimulatedSixWheelsDifferentialDriveOperations _mainPort = new
            SimulatedSixWheelsDifferentialDriveOperations();

        //rozszerzenie stanu service do 6 kół, definicja podana w pliku typów
        [InitialStatePartner(Optional = true, ServiceUri =
            "SimulatedSixWheelsDifferentialDriveService.Config.xml")]
        private SimulatedSixWheelsDifferentialDriveState _state = new
            SimulatedSixWheelsDifferentialDriveState();

        public SimulatedSixWheelsDifferentialDriveService(DsspServiceCreationPort creationPort)
            : base(creationPort)
        {
        }

        protected override void Start()
        {
            if (_state == null)
                CreateDefaultState();

            //simulation service działa na instancji klasy występującej w symulacji
            //do której referencja musi znaleźć się w _entity, w tym przypadku jest
            //to obiekt typu Ibis6kolEntity
            //w celu umożliwienia komunikacji service odpalany jest
            //z głównego manifestu projektu Simulator.manifest.xml
            //z wyszczególnionym partnerem o konkrenej nazwie (tutaj Ibis6kol). Adres
            //obiektu w symulacji przybiera postać http://localhost/Ibis6kol
            //w service dokonujemy subskrypcji oraz  sprawdzamy czy został
            //wstawiony obiekt o odpowiedniej nazwie (dokładnie takiej jak w manifeście).
            //Informacja, że pojawił się odpowiedni obiekt dostępna będzie poprzez port
            //_notificationTarget.

            //Dashboard i podobne obsługi zawsze komunikują się ze środowiskiem symulacyjnym
            //za pośrednictwem
            //simulation services
            //simulation services muszą być w tym samym węźle co symulacja
            //wówczas
            //_notificationTarget zawiera referencję do obieku

            _notificationTarget = new simengine.SimulationEnginePort();

            // PartnerType.Service is the entity instance name.
            simengine.SimulationEngine.GlobalInstancePort.Subscribe(
                ServiceInfo.PartnerList, _notificationTarget);

            //dopiero po uzyskaniu referencji do obiektu możliwe jest wydawanie polecen
            //wyjatek stanowi drop (zamknięcie usługi)
            Activate(new Interleave(
                new TeardownReceiverGroup
                (
                    Arbiter.Receive<simengine.InsertSimulationEntity>(false,
                    _notificationTarget, InsertEntityNotificationHandlerFirstTime),
                    Arbiter.Receive<DsspDefaultDrop>(false, _mainPort, DefaultDropHandler),
                    Arbiter.Receive<DsspDefaultDrop>(false, _diffDrivePort, DefaultDropHandler)
                ),
                new ExclusiveReceiverGroup(),
                new ConcurrentReceiverGroup()
            ));
        }

        void CreateDefaultState()
        {
        _state = new SimulatedSixWheelsDifferentialDriveState();
        _state.LeftWheel = new Microsoft.Robotics.Services.Motor.Proxy.WheeledMotorState();
        _state.RightWheel = new Microsoft.Robotics.Services.Motor.Proxy.WheeledMotorState();
        _state.LeftWheel.MotorState = new Microsoft.Robotics.Services.Motor.Proxy.MotorState();
        _state.RightWheel.MotorState = new Microsoft.Robotics.Services.Motor.Proxy.MotorState();
        _state.LeftWheel.EncoderState = new Microsoft.Robotics.Services.Encoder.Proxy.EncoderState();
        _state.RightWheel.EncoderState = new Microsoft.Robotics.Services.Encoder.Proxy.EncoderState();
        _state.RearLeftWheel = new Microsoft.Robotics.Services.Motor.Proxy.WheeledMotorState();
        _state.RearRightWheel = new Microsoft.Robotics.Services.Motor.Proxy.WheeledMotorState();
        _state.RearLeftWheel.MotorState = new Microsoft.Robotics.Services.Motor.Proxy.MotorState();
        _state.RearRightWheel.MotorState = new Microsoft.Robotics.Services.Motor.Proxy.MotorState();

        _state.MiddleLeftWheel = new Microsoft.Robotics.Services.Motor.Proxy.WheeledMotorState();
        _state.MiddleRightWheel = new Microsoft.Robotics.Services.Motor.Proxy.WheeledMotorState();
        _state.MiddleLeftWheel.MotorState = new Microsoft.Robotics.Services.Motor.Proxy.MotorState();
        _state.MiddleRightWheel.MotorState = new Microsoft.Robotics.Services.Motor.Proxy.MotorState();
        }
```

```
136
138         void UpdateStateFromSimulation()
            {
140         if (_entity != null)
            {
142         _state.TimeStamp = DateTime.Now;
            _state.LeftWheel.MotorState.CurrentPower = _entity.FrontLeftWheel.Wheel.MotorTorque;
144         _state.RightWheel.MotorState.CurrentPower = _entity.FrontRightWheel.Wheel.MotorTorque;
            _state.RearLeftWheel.MotorState.CurrentPower = _entity.RearLeftWheel.Wheel.MotorTorque;
146         _state.RearRightWheel.MotorState.CurrentPower = _entity.RearRightWheel.Wheel.MotorTorque;
            _state.MiddleLeftWheel.MotorState.CurrentPower = _entity.MiddleLeftWheel.Wheel.MotorTorque;
148         _state.MiddleRightWheel.MotorState.CurrentPower = _entity.MiddleRightWheel.Wheel.MotorTorque;
            _state.Position = _entity.State.Pose.Position;
150         }
            }

152         #region Entity Handlers
            void InsertEntityNotificationHandlerFirstTime(simengine.InsertSimulationEntity ins)
154         {
                // Publish the service to the local Node Directory
156             DirectoryInsert();

158             InsertEntityNotificationHandler(ins);


160
            }
162         //handle susequent  insert notifications
            void InsertEntityNotificationHandler(simengine.InsertSimulationEntity ins)
164         {
                _entity = (corobot.Ibis6kolEntity)ins.Body;//zapamiętaj referencję
166             _entity.ServiceContract = Contract.Identifier;
                //ustawienie identyfikatora obiektu na nasz (sygnalizacja kontroli)
168
                // create default state based on the physics entity
170             xna.Vector3 separation = _entity.FrontLeftWheel.Position - _entity.FrontRightWheel.Position;
                _state.DistanceBetweenWheels = separation.Length();
172
                _state.LeftWheel.MotorState.PowerScalingFactor = _entity.MotorTorqueScaling;
174             _state.RightWheel.MotorState.PowerScalingFactor = _entity.MotorTorqueScaling;

176             // enable other handlers now that we are connected
                Activate(new Interleave(
178                 new TeardownReceiverGroup
                    (
180                     Arbiter.Receive<DsspDefaultDrop>(false, _mainPort, DefaultDropHandler),
                        Arbiter.Receive<DsspDefaultDrop>(false, _diffDrivePort, DefaultDropHandler)
182                 ),
                    new ExclusiveReceiverGroup
184                 (
                        Arbiter.Receive<SetPose>(true, _mainPort, SetPoseHandler),
186                     Arbiter.ReceiveWithIterator<pxdrive.DriveDistance>(
                        true, _diffDrivePort, DriveDistanceHandler),
188                     Arbiter.ReceiveWithIterator<pxdrive.RotateDegrees>(
                        true, _diffDrivePort, RotateHandler),
190                     Arbiter.ReceiveWithIterator<pxdrive.SetDrivePower>(
                        true, _diffDrivePort, SetPowerHandler),
192                     Arbiter.ReceiveWithIterator<pxdrive.SetDriveSpeed>(
                        true, _diffDrivePort, SetSpeedHandler),
194                     Arbiter.ReceiveWithIterator<pxdrive.AllStop>(true, _diffDrivePort, AllStopHandler),
                        Arbiter.Receive<simengine.InsertSimulationEntity>(
196                     true, _notificationTarget, InsertEntityNotificationHandler),
                        Arbiter.Receive<simengine.DeleteSimulationEntity>(
198                     true, _notificationTarget, DeleteEntityNotificationHandler)
                    ),
200                 new ConcurrentReceiverGroup
                    (
202                     Arbiter.ReceiveWithIterator<dssphttp.HttpGet>(
                        true, _mainPort, MainPortHttpGetHandler),
204                     Arbiter.ReceiveWithIterator<Get>(
                        true, _mainPort, MainPortGetHandler),
206                     Arbiter.ReceiveWithIterator<dssphttp.HttpGet>(
                        true, _diffDrivePort, HttpGetHandler),
208                     Arbiter.ReceiveWithIterator<pxdrive.Get>(
                        true, _diffDrivePort, GetHandler),
210                     Arbiter.ReceiveWithIterator<pxdrive.Subscribe>(
                        true, _diffDrivePort, SubscribeHandler),
212                     Arbiter.ReceiveWithIterator<pxdrive.ReliableSubscribe>(
                        true, _diffDrivePort, ReliableSubscribeHandler),
214                     Arbiter.ReceiveWithIterator<pxdrive.EnableDrive>(
                        true, _diffDrivePort, EnableHandler)
216                 )
                ));
218         }

220         void DeleteEntityNotificationHandler(simengine.DeleteSimulationEntity del)
            {
222             _entity = null;

224             // disable other handlers now that we are no longer connected to the entity
                Activate(new Interleave(
226                 new TeardownReceiverGroup
                    (
228                     Arbiter.Receive<simengine.InsertSimulationEntity>(false,
                        _notificationTarget, InsertEntityNotificationHandlerFirstTime),
230                     Arbiter.Receive<DsspDefaultDrop>(false,
                        _mainPort, DefaultDropHandler),
232                     Arbiter.Receive<DsspDefaultDrop>(false,
                        _diffDrivePort, DefaultDropHandler)
234                 ),
                    new ExclusiveReceiverGroup(),
```

```
236                 new ConcurrentReceiverGroup()
                ));
238         }
            #endregion
240
            #region Subscribe Handling
242         public IEnumerator<ITask> SubscribeHandler(pxdrive.Subscribe subscribe)
            {
244             Activate(Arbiter.Choice(
                    SubscribeHelper(_subMgrPort, subscribe.Body, subscribe.ResponsePort),
246                 delegate(SuccessResult success)
                    {
248                     _subMgrPort.Post(new submgr.Submit(
                            subscribe.Body.Subscriber, DsspActions.UpdateRequest, _state, null));
250                 },
                    delegate(Exception ex) { LogError(ex); }
252             ));

254             yield break;
            }
256
            public IEnumerator<ITask> ReliableSubscribeHandler(pxdrive.ReliableSubscribe subscribe)
258         {
                Activate(Arbiter.Choice(
260                 SubscribeHelper(_subMgrPort, subscribe.Body, subscribe.ResponsePort),
                    delegate(SuccessResult success)
262                 {
                        _subMgrPort.Post(new submgr.Submit(
264                         subscribe.Body.Subscriber, DsspActions.UpdateRequest, _state, null));
                    },
266                 delegate(Exception ex) { LogError(ex); }
                ));
268             yield break;
            }
270         #endregion

272         #region _mainPort handlers
            public IEnumerator<ITask> MainPortHttpGetHandler(dssphttp.HttpGet get)
274         {
                UpdateStateFromSimulation();
276             get.ResponsePort.Post(new dssphttp.HttpResponseType(_state));
                yield break;
278         }

280         public IEnumerator<ITask> MainPortGetHandler(Get get)
            {
282             UpdateStateFromSimulation();
                get.ResponsePort.Post(_state);
284             yield break;
            }
286
            public void SetPoseHandler(SetPose setPose)
288         {
                if (_entity == null)
290                 throw new InvalidOperationException("Simulation entity not registered with service");

292             Task<corobot.Ibis6kolEntity, Pose> task = new
                    Task<corobot.Ibis6kolEntity, Pose>(_entity,
294                 setPose.Body.EntityPose, SetPoseDeferred);
                _entity.DeferredTaskQueue.Post(task);
296         }

298         void SetPoseDeferred(corobot.Ibis6kolEntity entity, Pose pose)
            {
300             entity.PhysicsEntity.SetPose(pose);
            }
302
            #endregion
304
            #region _diffDrivePort handlers
306         public IEnumerator<ITask> HttpGetHandler(dssphttp.HttpGet get)
            {
308             UpdateStateFromSimulation();
                pxdrive.DriveDifferentialTwoWheelState _twoWheelState =
310                 (pxdrive.DriveDifferentialTwoWheelState)((pxdrive.DriveDifferentialTwoWheelState)
                                                                    _state).Clone();
312             get.ResponsePort.Post(new dssphttp.HttpResponseType(_twoWheelState));
                yield break;
314         }//kompatybilność z usługami które oczekują stanu dwukołowego

316         public IEnumerator<ITask> GetHandler(pxdrive.Get get)
            {
318             UpdateStateFromSimulation();
                pxdrive.DriveDifferentialTwoWheelState _twoWheelState =
320                 (pxdrive.DriveDifferentialTwoWheelState)((pxdrive.DriveDifferentialTwoWheelState)
                                                                    _state).Clone();
322             get.ResponsePort.Post(_twoWheelState);
                yield break;
324         }

326         public IEnumerator<ITask> DriveDistanceHandler(pxdrive.DriveDistance driveDistance)
            {
328             SuccessFailurePort entityResponse = new SuccessFailurePort();
                _entity.DriveDistance(
330                 (float)driveDistance.Body.Distance,
                    (float)driveDistance.Body.Power,
332                 entityResponse);

334             yield return Arbiter.Choice(entityResponse,
                    delegate(SuccessResult s)
```

```
336                     { driveDistance.ResponsePort.Post(DefaultUpdateResponseType.Instance); },
                    delegate(Exception e)
338                 {
                        driveDistance.ResponsePort.Post(new W3C.Soap.Fault());
340                 });

342             yield break;
            }
344
            public IEnumerator<ITask> RotateHandler(pxdrive.RotateDegrees rotate)
346         {
                SuccessFailurePort entityResponse = new SuccessFailurePort();
348             _entity.RotateDegrees(
                    (float)rotate.Body.Degrees,
350                 (float)rotate.Body.Power,
                    entityResponse);
352
                yield return Arbiter.Choice(entityResponse,
354                 delegate(SuccessResult s) {
                            rotate.ResponsePort.Post(DefaultUpdateResponseType.Instance); },
356             delegate(Exception e)
                {
358                 rotate.ResponsePort.Post(new W3C.Soap.Fault());
                });
360
                yield break;
362         }

364         public IEnumerator<ITask> SetPowerHandler(pxdrive.SetDrivePower setPower)
            {
366             if (_entity == null)
                    throw new InvalidOperationException("Simulation entity not registered with service");
368
                // Call simulation entity method for setting wheel torque
370             _entity.SetMotorTorque(
                    (float)(setPower.Body.LeftWheelPower),
372                 (float)(setPower.Body.RightWheelPower));

374             UpdateStateFromSimulation();
                setPower.ResponsePort.Post(DefaultUpdateResponseType.Instance);
376
                // send update notification for entire state
378             _subMgrPort.Post(new submgr.Submit(_state, DsspActions.UpdateRequest));
                yield break;
380         }

382         public IEnumerator<ITask> SetSpeedHandler(pxdrive.SetDriveSpeed setSpeed)
            {
384             if (_entity == null)
                    throw new InvalidOperationException("Simulation entity not registered with service");
386
                _entity.SetVelocity(
388                 (float)setSpeed.Body.LeftWheelSpeed,
                    (float)setSpeed.Body.RightWheelSpeed);
390
                UpdateStateFromSimulation();
392             setSpeed.ResponsePort.Post(DefaultUpdateResponseType.Instance);

394             // send update notification for entire state
                _subMgrPort.Post(new submgr.Submit(_state, DsspActions.UpdateRequest));
396
                yield break;
398         }

400         public IEnumerator<ITask> EnableHandler(pxdrive.EnableDrive enable)
            {
402             if (_entity == null)
                    throw new InvalidOperationException("Simulation entity not registered with service");
404
                _state.IsEnabled = enable.Body.Enable;
406             _entity.IsEnabled = _state.IsEnabled;

408             UpdateStateFromSimulation();
                enable.ResponsePort.Post(DefaultUpdateResponseType.Instance);
410
                // send update for entire state
412             _subMgrPort.Post(new submgr.Submit(_state, DsspActions.UpdateRequest));
                yield break;
414         }

416         public IEnumerator<ITask> AllStopHandler(pxdrive.AllStop estop)
            {
418             if (_entity == null)
                    throw new InvalidOperationException("Simulation entity not registered with service");
420
                _entity.SetMotorTorque(0, 0);
422             _entity.SetVelocity(0);

424             UpdateStateFromSimulation();
                estop.ResponsePort.Post(DefaultUpdateResponseType.Instance);
426
                // send update for entire state
428             _subMgrPort.Post(new submgr.Submit(_state, DsspActions.UpdateRequest));
                yield break;
430         }
            #endregion
432
            // Test the DriveDistance and RotateDegrees messages
434         public IEnumerator<ITask> TestDriveDistanceAndRotateDegrees()
            {
```

```
436             Random rnd = new Random();
                bool success = true;
438
                // drive in circles
440             while (success)
                {
442                 double distance = rnd.NextDouble() * 1 + 0.5;
                    double angle = rnd.NextDouble() * 90 - 45;
444
                    // first leg
446                 yield return Arbiter.Choice(
                        _diffDrivePort.RotateDegrees(angle, 0.2),
448                     delegate(DefaultUpdateResponseType response) { },
                        delegate(Fault f) { success = false; }
450                 );

452                 yield return Arbiter.Choice(
                        _diffDrivePort.DriveDistance(distance, 0.2),
454                     delegate(DefaultUpdateResponseType response) { },
                        delegate(Fault f) { success = false; }
456                 );

458                 // return
                    yield return Arbiter.Choice(
460                     _diffDrivePort.RotateDegrees(180, 0.2),
                        delegate(DefaultUpdateResponseType response) { },
462                     delegate(Fault f) { success = false; }
                    );
464
                    yield return Arbiter.Choice(
466                     _diffDrivePort.DriveDistance(distance, 0.2),
                        delegate(DefaultUpdateResponseType response) { },
468                     delegate(Fault f) { success = false; }
                    );
470
                    // reset position
472                 yield return Arbiter.Choice(
                        _diffDrivePort.RotateDegrees(180 - angle, 0.2),
474                     delegate(DefaultUpdateResponseType response) { },
                        delegate(Fault f) { success = false; }
476                 );
                }
478         }
        }
480 }
```

Bibliografia

[1] K. Johns, T. Taylor. *Professional Microsoft Robotics Developer Studio*. Wiley Publishing, 2008.

[2] Dokumentacja Microsoft Robotics Developer Studio. http://msdn.microsoft.com/en-us/library/microsoft.robotics.simulation.engine.differentialdriveentity_members.aspx.

[3] Dokumentacja Microsoft Robotics Developer Studio. http://msdn.microsoft.com/en-us/library/microsoft.robotics.simulation.engine.visualentity.aspx.

4. SimRobot

Michał Wcisło

Projekt SimRobot powstał w roku 1990 jako ogólny, kinematyczny symulator dla pojedynczego użytkownika [2]. Za jego rozwój odpowiedzialne są Universität Bremen i German Research Center for Artificial Intelligence. Zamysłem twórców było stworzenie aplikacji pozwalającej w wygodny sposób pracować na modelach rzeczywistych obiektów w środowisku 3D. Aplikacja wykorzystuje bibliotekę graficzną (OpenGL) i fizyczną (ODE). Projekt jest silnie związany z zawodami RoboCup, gdzie wykorzystywany jest do symulacji przez niemiecki zespół. Niniejszy opis oprogramowania opiera się na wersji SimRobot 2011 (B-Human edition).

Aplikację SimRobot można zainstalować pod systemami Windows, Linux i Mac OS X. Instalacja w systemie Linux może być problematyczna z uwagi na konieczność instalacji dodatkowych bibliotek. Katalog z programem zawiera projekt Visual Studio, stąd instalacja z wykorzystaniem tego pakietu wydaje się najprostsza.

4.1. Opis aplikacji

Sama aplikacja ma stosunkowo prosty interfejs (rysunek 4.1), który ogranicza się do wczytania sceny, podglądu jej struktury, sterowania symulacją i manipulacją widokiem. Aby rozpocząć symulację konieczne jest stworzenie dwóch plików. Jeden z nich opisuje scenę, drugi jest kontrolerem obiektów na scenie i jego zadaniem jest opis zachowań modeli.

Opis sceny ma postać dokumentu XML w schemacie RoSiML (Robot Simulation Markup Language) [1]. Umieszcza się w nim parametry oświetlenia, symulacji (krok symulacji), fizyczne (siłę grawitacji) oraz opis modeli znajdujących się na scenie. Obiekty tworzy się wykorzystując proste bryły tj. sfery, walce oraz prostopadłościany. Wszystkim tworzonym obiektom nadaje się masę i definiuje punkt ciężkości. Do łączenia elementów wykorzystuje się różne rodzaje przegubów, do których można dodawać napędy. Dodatkowo istnieje możliwość umieszczania sensorów położenia, odległości lub sensorów zderzakowych.

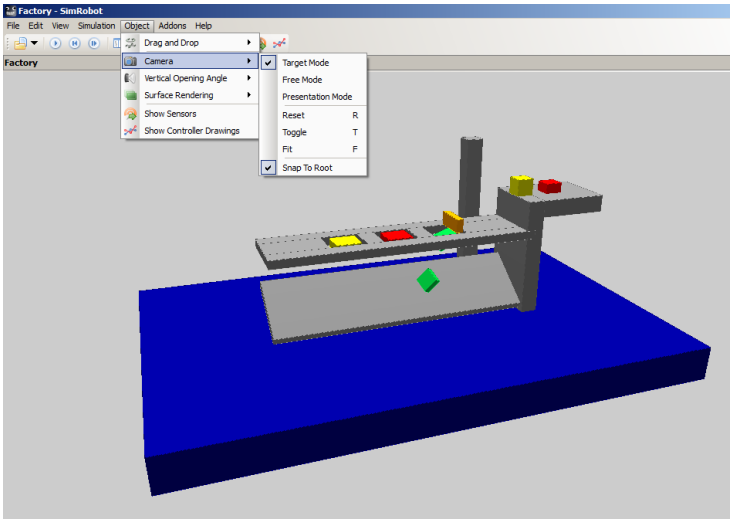
Plik kontrolera jest programem napisanym w języku C++ implementującym klasę modelowanego obiektu. W ciele klasy definiuje się funkcję *update*, która wykonywana jest w każdym cyklu symulacji, przez co pozwala na opisanie w niej zachowań obiektu.

Dobrym punktem wyjściowym do tworzenia własnej sceny i kontrolera są dwa przykłady dostarczane wraz z aplikacją. Przy tworzeniu własnego kontrolera należy pamiętać o przeniesieniu wygenerowanej biblioteki z rozszerzeniem .dll do folderu SimRobot w katalogu Build.

4.2. Modelowanie prostych obiektów

Aby przetestować działanie aplikacji należało stworzyć trzy obiekty dynamiczne: dwuwahadło, monocykl oraz manipulator z elastycznościami.

Obiekt dwuwahadła powstał z prostych prostopadłościanów ze zdefiniowanymi przegubami i napędami umieszczonych na nieruchomej bryle na środku sceny. Obraz sceny



Rysunek 4.1. Interfejs programu SimRobot

został umieszczony na rysunku 4.3. Modelowanie kinematyki zostało przeprowadzone na podstawie schematu z rysunku 4.2, na którym umieszczono lokalne układy współrzędnych zgodne z notacją Denavita-Hartenberga. Następnie wyprowadzono macierze przejścia między kolejnymi elementami łańcucha kinematycznego A_0^1 i A_1^2 oraz macierz kinematyki manipulatora (równania (4.1)–(4.4)). Dane zawierające wymiary zostały zamieszczone poniżej.

$$l1 = 0,5[m], \qquad l2 = 0,5[m], \qquad h = 0,7[m],$$

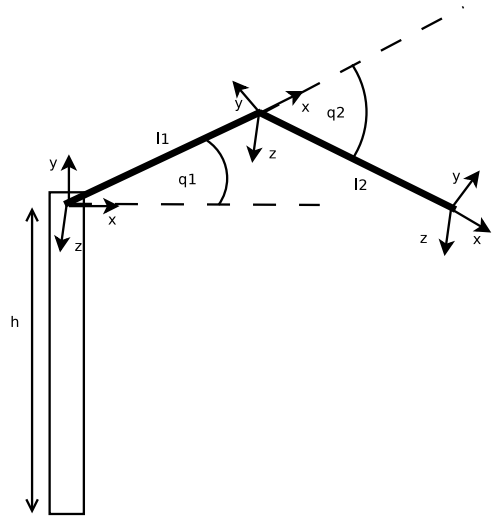
$$K(q) = A_0^1 A_1^2 \tag{4.1}$$

$$\mathbf{A_0^1} = Rot(z, q_1)Trans(x, l_1) = \begin{pmatrix} \cos q_1 & -\sin q_1 & 0 & l_1 \cos q_1 \\ \sin q_1 & \cos q_1 & 0 & l_1 \sin q_1 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{4.2}$$

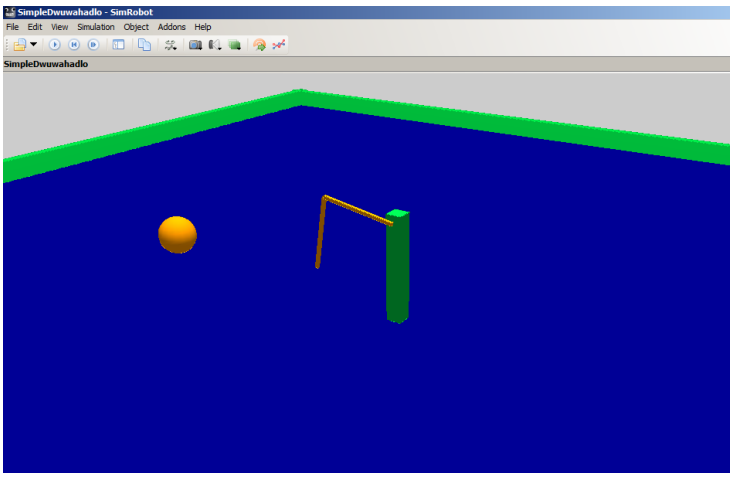
$$\mathbf{A_1^2} = Rot(z, q_2)Trans(x, l_2) = \begin{pmatrix} \cos q_2 & -\sin q_2 & 0 & l_2 \cos q_2 \\ \sin q_2 & \cos q_2 & 0 & l_2 \sin q_2 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{4.3}$$

$$\mathbf{K(q)} = \begin{pmatrix} \cos(q_1 + q_2) & -\sin(q_1 + q_2) & 0 & l_2 \cos(q_1 + q_2) + l_1 \cos q_1 \\ \sin(q_1 + q_2) & \cos(q_1 + q_2) & 0 & l2 \sin(q_1 + q_2) + l_1 \sin q_1 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{4.4}$$

Monocykl zamodelowano jako prostopadłościan z kołami w formie sfer. Model monocykla został umieszczony na rysunku 4.5. Równania ograniczeń nieholonomicznych



Rysunek 4.2. Schemat dwuwahadła



Rysunek 4.3. Obiekt dwuwahadła

(4.5)–(4.8) zostały wyprowadzone na podstawie rysunku 4.4. Ograniczenia w postaci Pfaffa zostały przedstawione w równaniu (4.9). Następnie wyprowadzono kinematykę monocykla (równanie (4.10)).

Ograniczenia na brak poślizgu bocznego:

$$.\dot{x}_A \sin\Theta - \dot{y}_A \cos\Theta = 0, \tag{4.5}$$

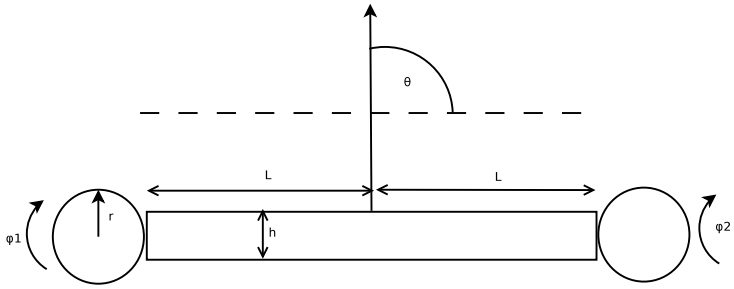
$$\dot{x}_B \sin\Theta - \dot{y}_B \cos\Theta = 0. \tag{4.6}$$

Ograniczenia na brak poślizgu wzdłużnego:

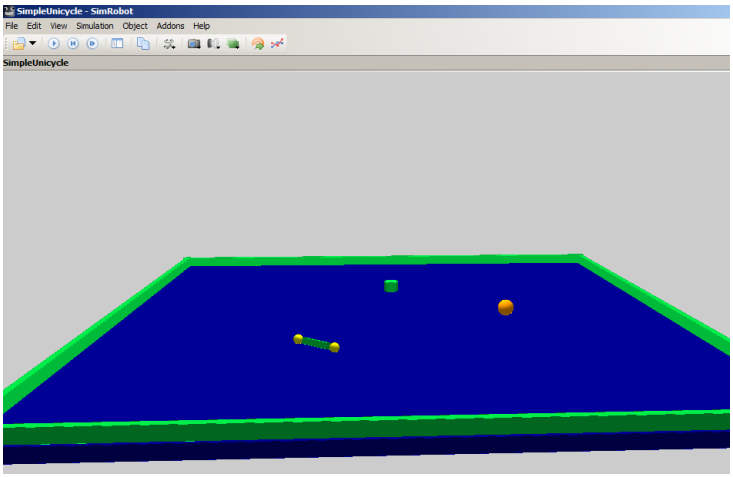
$$.\dot{x}_A \cos\Theta - \dot{y}_A \sin\Theta - r\phi_1 = 0, \tag{4.7}$$

$$\dot{x}_B \cos\Theta - \dot{y}_B \sin\Theta - r\phi_2 = 0. \tag{4.8}$$

$$\begin{pmatrix} \sin\Theta & -\cos\Theta & 0 & 0 & 0 \\ \cos\Theta & \sin\Theta & -L & -r & 0 \\ \cos\Theta & \sin\Theta & L & 0 & -r \end{pmatrix} \begin{pmatrix} \dot{x} \\ \dot{y} \\ \dot{\Theta} \\ \dot{\phi}_1 \\ \dot{\phi}_2 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}. \tag{4.9}$$



Rysunek 4.4. Schemat monocykla



Rysunek 4.5. Obiekt monocykla

$$\begin{pmatrix} \dot{x} \\ \dot{y} \\ \dot{\Theta} \\ \dot{\phi}_1 \\ \dot{\phi}_2 \end{pmatrix} = \begin{pmatrix} \cos\Theta & \cos\Theta \\ \sin\Theta & \sin\Theta \\ \frac{1}{L} & -\frac{1}{L} \\ 0 & \frac{2}{r} \\ \frac{2}{r} & 0 \end{pmatrix} \begin{pmatrix} u_1 \\ u_2 \end{pmatrix}. \tag{4.10}$$

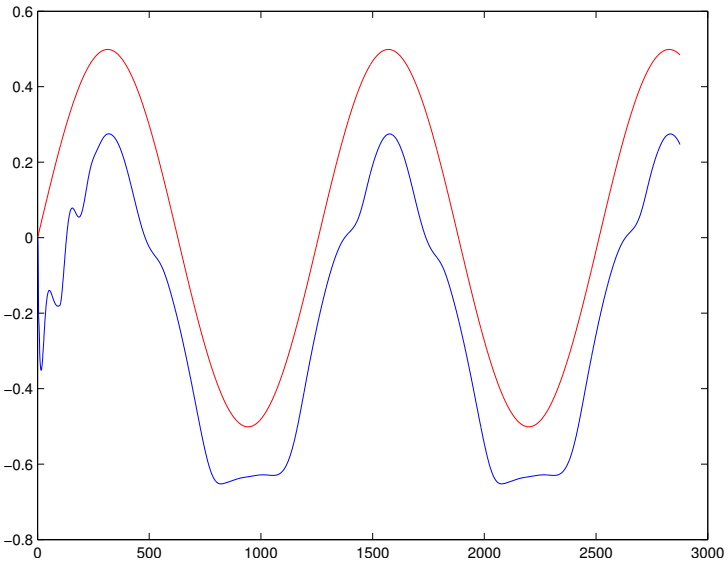
Interpretacja fizyczna sterowań ma następującą postać:
$\omega = \frac{1}{L}(u_1 - u_2)$ – prędkość kątowa monocykla,
$v = u_1 + u_2$ – prędkość liniowa monocykla.

Model manipulatora z elastycznościami nie został stworzony, ponieważ schemat RoSiML nie udostępnia możliwości stworzenia przegubu elastycznego.

4.3. Symulacje obiektów

W celu przetestowania stworzonych modeli i wypróbowania środowiska symulacyjnego zaimplementowano algorytm sterowania Qu i Dorsey'a dla dwuwahadła (śledzenie trajektorii przegubowej) i algorytm Samsona (śledzenie trajektorii) dla monocykla [3]. Oba algorytmy zostały zaimplementowane w formie programu kontrolera.

Przy implementacji algorytmu Qu i Dorsey'a w modelu dynamiki dwuwahadła zostały uwzględnione parametry dynamiczne modelu umieszczonego w symulatorze. Wyliczane



Rysunek 4.6. Wykres położenia (niebieski) i trajektorii przegubowej (czerwony) przegubu $q1$

położenia przegubów były bezpośrednio przekazywane na napędy stworzonego manipulatora, dlatego przykład ten miał charakter jedynie demonstracyjny. Na rysunkach 4.6 i 4.7 przedstawiono wykresy położenia (niebieski) i trajektorii (czerwony) przegubu odpowiednio $q1$ i $q2$. Pewne niedoskonałości sterowania mogą być spowodowane stosunkowo prostą implementacją elementów składowych algorytmu w szczególności obiektu integratora.

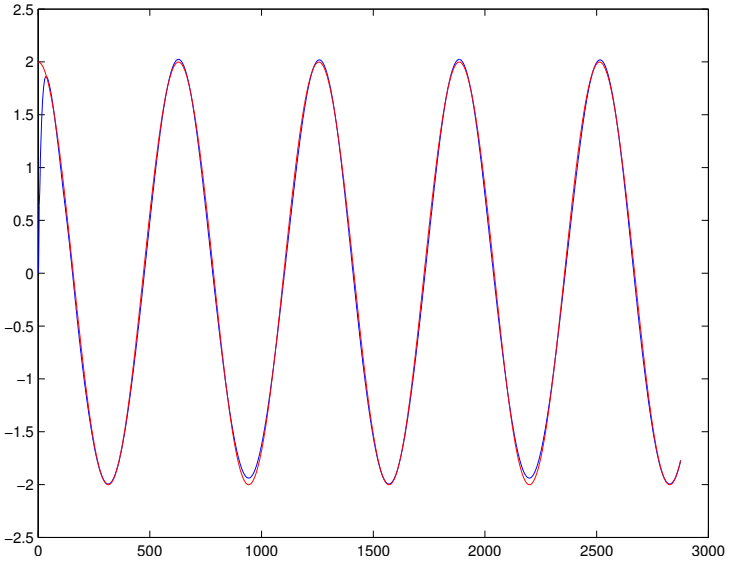
W algorytmie Samsona także uwzględniono parametry dynamiczne modelu w szczególności koła w formie sfer co ma swoje odzwierciedlenie w implementacji kontrolera. Prędkości kół ustawiane są na podstawie prędkości liniowej i kątowej wyliczanej w algorytmie. Na rysunkach 4.8 i 4.9 zamieszczono śledzoną trajektorię jak i położenie oraz kąt obrotu monocykla.

4.4. Podsumowanie

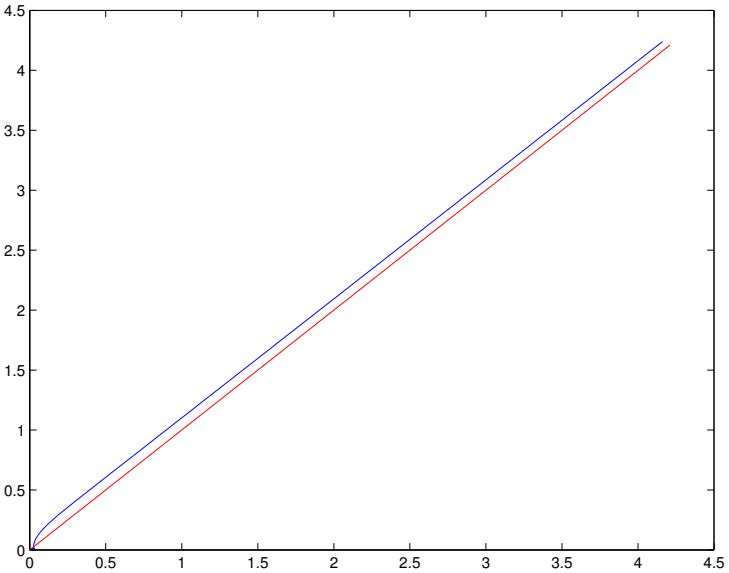
Aplikacja SimRobot zgodnie z założeniami twórców jest symulatorem kinematycznym i stosunkowo trudno realizować w niej algorytmy dynamiczne. Sam program nie dostarcza żadnych struktur przydatnych w sterowaniu modelowanych obiektów. Wszelkie kontrolery należy tworzyć od początku, co przemawia za małą uniwersalnością tego oprogramowania. Sam fakt tworzenia własnych struktur potwierdza fakt, że nie jest to aplikacja przeznaczona do symulacji, w których istotną rolę odgrywają wyniki liczbowe. Należy także wspomnieć, że dokumentacja SimRobot jest raczej skąpa i trudno znaleźć przykłady lub wsparcie ze strony użytkowników, lub twórców tego oprogramowania.

Z drugiej strony sama struktura tworzonych symulacji jest bardzo wygodna. Opis sceny w XML jest łatwo skalowalny i pozwala na efektywne i szybkie generowanie środowiska symulacji. Także zastosowanie interfejsu programistycznego opartego na języku C++ pozwala na tworzenie i testowanie własnych algorytmów sterowania.

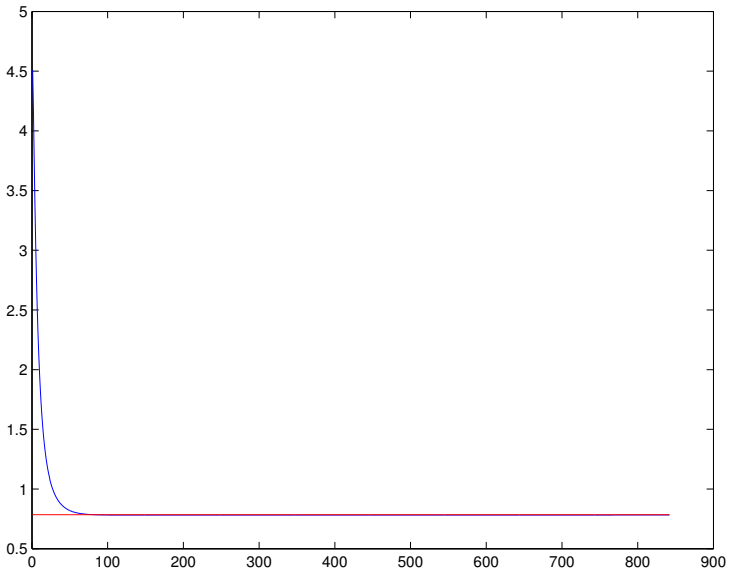
Podsumowując SimRobot jest dobrym narzędziem do wizualizacji już zaimplementowanych systemów sterowania i symulowania interakcji pomiędzy modelami i otoczeniem.



Rysunek 4.7. Wykres położenia (niebieski) i trajektorii przegubowej (czerwony) przegubu $q2$



Rysunek 4.8. Wykres położenia (niebieski) i śledzonej trajektorii (czerwony) monocykla



Rysunek 4.9. Wykres kąta obrotu (niebieski) i śledzonego kąta obrotu (czerwony) monocykla

Bibliografia

[1] K. Ghazi-Zahedi, T. Laue, T. Röfer, P. Schöll, K. Spiess, A. Twickel, S. Wischmann. RoSiML schema. http://www.informatik.uni-bremen.de/spprobocup/RoSiML.html.

[2] T. Laue, T. Röfer, redaktorzy. *SimRobot – Development and Applications*. Deutsches Forschungszentrum für Künstliche Intelligenz GmbH, Sichere Kognitive Systeme, Bremen, Germany, 2008.

[3] A. Mazur. Projekt specjalnościowy, zajęcia w roku akademickim 2010/2011.

5. V-REP

Dawid Powązka

Virtual Robot Evaluation Platform jest systemem symulacji robotów z wbudowanym interfejsem edytorskim. V-REP jest wykorzystywany do symulacji, testowania, ewaulacji prostych i złożonych systemów zrobotyzowanych lub robotycznych podzespołów. Producentem tego oprogramowania jest dr. Marc Andreas Freese. Przy opracowaniu tego dokumentu korzystano z wersji V-REPa 2.5.9.

5.1. Cechy V-REP

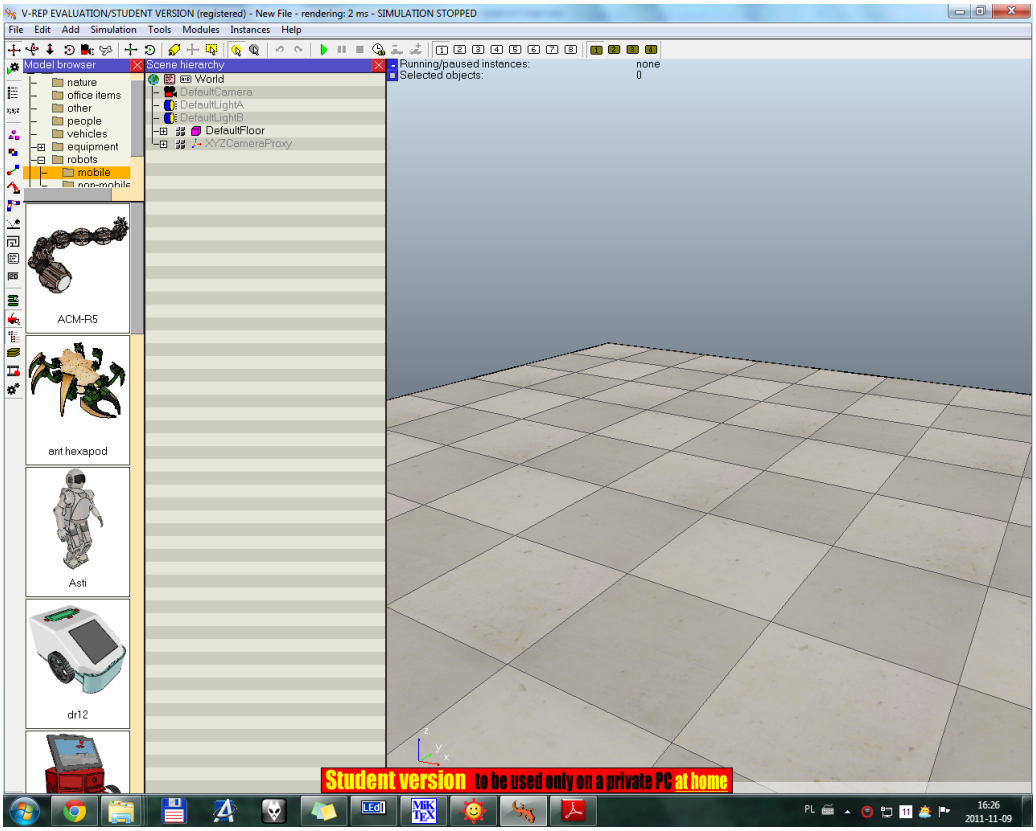
System V-REP jest wykorzystywany w dziedzinie robotyki do:
— symulacji zespołów robotycznych,
— wizualizacji procesów przemysłowych,
— szybkiego prototypowania nowych podzespołów,
— badania i budowy systemów sterowania robotów,
— prezentacji możliwości robotów.

Przedstawione powyżej zastosowania wymagają od oprogramowania pewnej elastyczności oraz dużej gamy możliwości. Rozpoczynając pracę z jakimkolwiek oprogramowaniem symulacyjnym powinniśmy mieć świadomość jego ma ono możliwości a jakie stawia wymagania czy ograniczenia. Główne cechy V-REPa zostały przedstawione poniżej. W dalszej części dokumentu zostaną opisane dokładniej elementy związane bezpośrednio z modelowaniem dynamiki i kinematyki robotów. System V-REP:
— umożliwia symulację i modelowanie dynamiki oraz kinematyki robotów,
— umożliwia symulację działania sensorów,
— posiada moduły wykrywania kolizji, obliczania odległości od obiektów czy generowania ścieżek dla robotów,
— zezwala na pisanie własnych skryptów w języku LUA,
— umożliwia dołączenie zewnętrznego sterownika,
— dostarcza API do jezyka C/C++.

System V-REP można pobrać ze strony producenta [6]. Dla celów testowych mamy do dyspozycji wersję aktywną na 1-2 miesiące. Dla studentów po rejestracji oprogramowanie nie ulegnie dezaktywacji. Ponadto, można pobrać program V-REP Player, który umożliwia odgrywanie wcześniej utworzonych symulacji.

5.2. Interfejs graficzny V-REP

Oprogramowanie V-REP dostarcza wbudowane środowisko edytorskie, które pozwala użytkownikowi na tworzenie nowych modeli robotów. Graficzny interfejs programu został przedstawiony na rysunku 5.1. Jest on zbudowany przejrzyście i intuicyjnie, co pozwala na szybki dostęp do właściwości edytowanego elementu lub innego rodzaju ustawień modułów obliczeniowych np. dynamiki. W V-REPie możemy zdefiniować 7 głównych widoków, co jest pomocne jeśli co jakiś czas chcemy obserwować symulację z pewnej perspektywy.

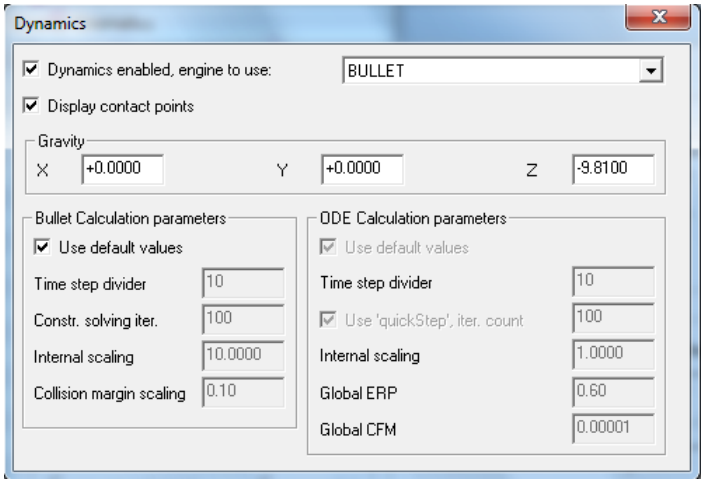


Rysunek 5.1. Graficzny interfejs systemu V-REP

Ciekawym rozwiązaniem jest praca na warstwach. Mamy do dyspozycji 16 warstw, na których mogą być umieszczane różne elementy. Domyślnie aktywnych jest pierwszych 8 warstw, co oznacza, że widzimy przedmioty znajdujące się na tych warstwach. Chcąc ukryć pewne przedmioty, np. przeguby, przesuwamy je na dalszą warstwę. Do manipulacji używamy głównie myszki. Za pomocą odpowiedniego narzędzia możemy zmieniać położenie przedmiotu, lub jego orientację. W razie bardziej dokładnych operacji możemy wprowadzić odpowiednie wartości liczbowe. W celu dobrego zaznajomienia się z poruszaniem po programie warto przeczytać rozdział *User interface* w dokumentacji V-REPa [5].

5.3. Dynamika w V-REP

Środowisko V-REP umożliwia symulację dynamiczną obiektów prostych oraz złożonych jak roboty. W tym celu wykorzystywane są dwa systemy modelujące dynamikę i fizykę obiektów. Są do Open Dynamics Engine oraz Bullet Physics. Są to dwa silniki na licencji open source. Więcej informacji o tych narzędziach można znaleźć na stronach internetowych [4] i [1]. Okno wyboru silnika dynamiki i jego ustawienia jest przedstawione na rysunku 5.2. Wszystkie przedmioty znajdujące się na scenie mogą być statyczne lub dynamicznie symulowane. Elementy statyczne nie podlegają działaniu żadnych sił i są obiektami stałymi, nieoddziałującymi z żadnymi innymi elementami sceny. Natomiast obiekty symulowane dynamicznie są elementami oddziałującymi z otoczeniem. Tego typu obiekty wymagają ustawienia dodatkowych parametrów jak masa, momenty bezwładności itp. Jednakże należy tutaj wspomnieć, że oprogramowanie pozwala na wprowadzenie tylko podstawowych, najpotrzebniejszych własności obiektów. Okno własności obiektu



Rysunek 5.2. Ustawienia dynamiki

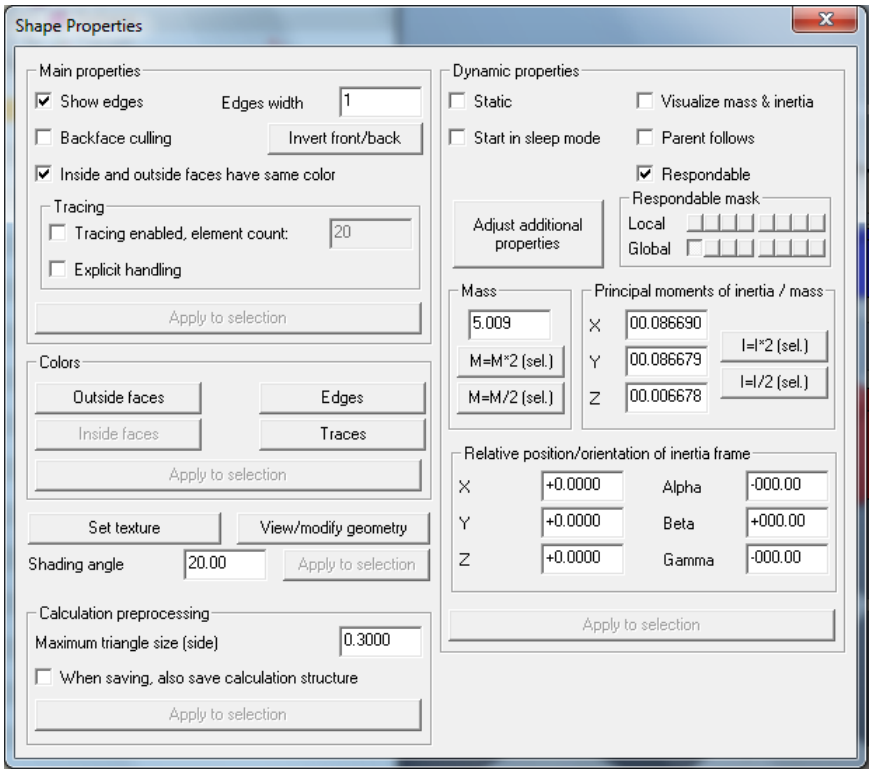
przedstawiono na rysunku 5.3. Należy również zaznaczyć, że producent zaleca stosowanie dynamiki tylko jeśli jest ona niezbędna. W sytuacjach kiedy nasze roboty nie wchodzą w interakcje z otoczeniem zalecane jest stosowanie wyłącznie kinematyki do sterowania. Do jak najlepszego wykorzystania dynamiki we własnych symulacjach producent przedstawił 7 zasad przygotowywania symulacji z wykorzystaniem dynamiki. Są one następujące:

1. Używaj *Pure Shapes*.
2. Używaj prostej hierarchicznej struktury.
3. Starannie wybieraj bazę modelu.
4. Używaj rozsądnych wymiarów.
5. Używaj rzeczywistych mas, lecz nie za lekkich.
6. Utrzymuj dostatecznie duże momenty bezwładności.
7. Przenieś wszystkie dynamiczne elementy na 9 warstwę.

Przedstawione powyżej reguły zostaną bardziej przybliżone w podrozdziale 5.7 o symulacji dwuwahadła.

5.4. Kinematyka

W systemie V-REP możliwe jest sterowanie robotów w trybie kinematyki prostej lub odwrotnej. Kinematyka odwrotna jest zaszyta w programie w postaci algorytmów pseudo inverse lub DLS. Oprogramowanie pracuje na zdefiniowanych przez użytkownika łańcuchach kinematycznych, które w bardzo łątwy sposób można utworzyć. Do tego celu wystarczy odpowiednio zbudowanej hierarchicznej struktury robota. Okno parametrów przedstawione jest na rysunku 5.4. Można definiować wiele łańcuchów kinematycznych z różnymi parametrami oraz algorytmami rozwiązywania. Ciekawą rzeczą jest definiowanie bliźniaczych łańcuchów kinematycznych z różnymi algorytmami rozwiązywania kinematyki odwrotnej. W takim przypadku możemy ustawić, że gdy pierwszy łańcuch z algorytmem mało dokładnym pseudo inverse nie zwróci rozwiązania, możemy przełączyć na dużo bardziej dokładny algorytm DLS. Pozwala to na ponowne przeliczenie zadania kinematyki w miejscach wrażliwych na błędy. W dolnej części okna 5.4 wybieramy końcówkę roboczą naszego łańcucha kinematycznego oraz bazę. Możliwe jest również ustawienie ograniczeń. Szczegóły implementacyjne zostaną przedstawione w podrozdziale 5.7 o budowie dwuwahadła.



Rysunek 5.3. Parametry obiektu

5.5. Sensory

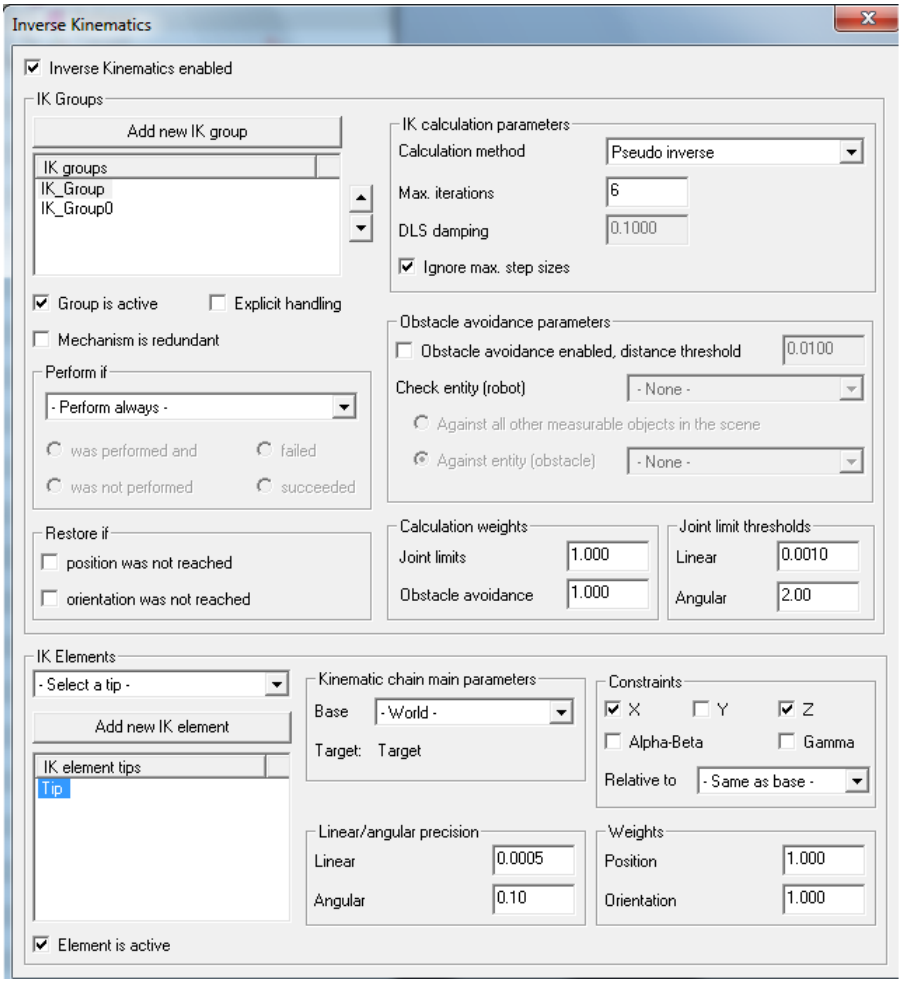
Dużą zaletą korzystania z oprogramowania V-REP jest dość rozbudowana i łatwa w obsłudze sensoryka. Możliwe jest korzystanie z:
— sensorów odległości,
— kamer,
— czujników nacisku.

5.5.1. Czujniki odległości

W programie V-REP dostępnych jest 5 typów czujników odległości. Są to:
— ultradźwiękowe,
— podczerwieni,
— laserowe,
— indukcyjne,
— pojemnościowe.
Ponadto, użytkownik może wybrać jeden z pięciu kształtów rozchodzenia się promienia danego typu sensora. Wszystkie rodzaje zostały przedstawione na rysunku 5.5. Każdy dodany czujnik może skonfigurować według jego rzeczywistych parametrów. Możemy zmieniać zasięg czujnika, kąt rozchodzenia się jego promieni, strefę nieczułości itp.

5.5.2. Kamery

W omawianym oprogramowaniu możemy dodać również kamery. Są one reprezentowane jak pokazano na rysunku 5.6. Tutaj również możemy dostosować parametry do rzeczywistej kamery. Można zmieniać rozdzielczość, kąt projekcji, dystans na jaki widzi nasza kamera itp. Obraz z kamery może być na bieżąco wyświetlany podczas symulacji.



Rysunek 5.4. Parametry kinematyki

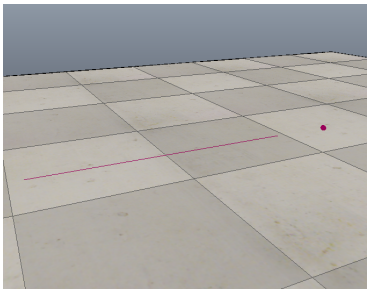
Ciekawą rzeczą w systemie V-REP jest możliwość zastosowania prostego przetwarzania obrazów. Takie przetwarzanie odbywa się poprzez dodawanie w odpowiedniej kolejności interesujących nas operacji na obrazie. Można używać np. progowania czy wykrywania danego koloru. Dzięki temu można budować proste systemy wizyjne dla naszych robotów.

5.5.3. Czujniki nacisku

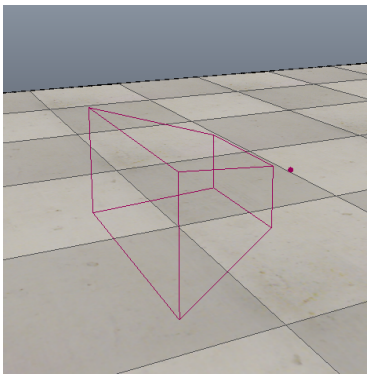
Czujnik nacisku jest istotny w sytuacjach gdy nasze modele wchodzą w interakcje z innymi przedmiotami na scenie. V-REP umożliwia nam korzystanie z takiego typu czujników. Mamy możliwość ustawienia ilości próbek, z których później może być zwracana wartość średnia lub mediana. Można również ustawić próg czułości naszego sensora.

5.6. Skrypty

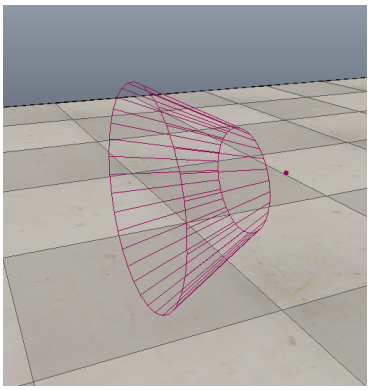
V-REP umożliwia pisanie własnych skryptów w języku skryptowym LUA. Język ten przypomina pseudokod w C, także opanowanie go nie przyprawia większych problemów. Więcej na temat samego języka można znaleźć w dokumentacji na stronie [3]. W systemie V-REP każdy skrypt jest identyfikowany z obiektem np. robotem. Wyjątkiem jest skrypt odpowiadający za symulację otoczenia. Takie zastosowanie umożliwia użytkownikowi pełną elastyczność w tworzeniu modeli, korzystania z odczytów sensorów, sterowania



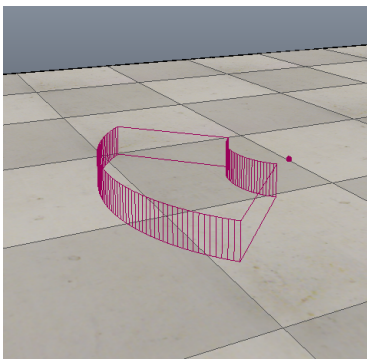
(a) Promień.



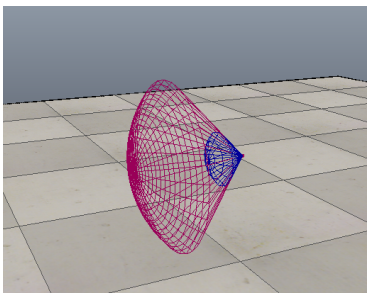
(b) Piramida.



(c) Cylinder.

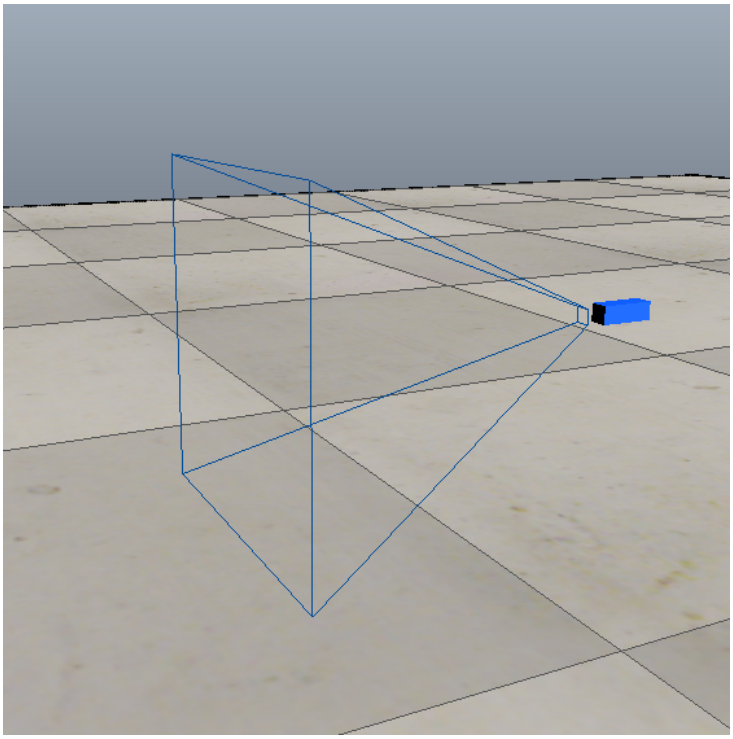


(d) Dysk.



(e) Stożek.

Rysunek 5.5. Rodzaje czujników odległości



Rysunek 5.6. Kamera

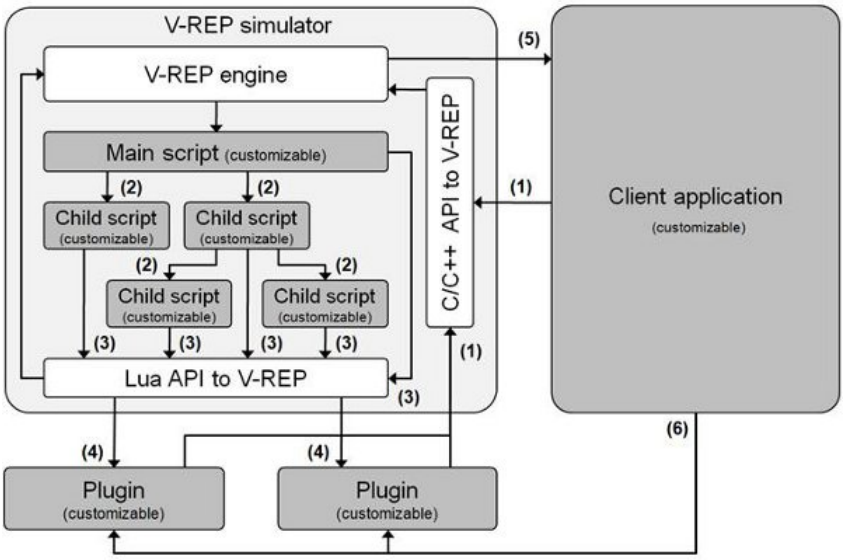
przegubami. W efekcie daje to możliwość budowania systemów sterowania i testowania różnych algorytmów sterowania. Na rysunku 5.7 przedstawiono przebieg sterowania pomiędzy skryptami w V-REP. Ważne uwagi jest, że skrypt główny (MainScript) uruchamia po kolejne skrypty utożsamione z naszymi modelami (ChildScript), które są wykonywane w trybie wątkowym lub bez wątkowym. W trybie wątkowym skrypty te pracują przez cały czas w tle, natomiast bez wątkowe są wywoływane w każdym kroku symulacji, podejmują pewne akcje i kończą swoje działanie. Więcej informacji o całej architekturze sterowania w V-REPie można przeczytać w dokumentacji technicznej na stronie systemu [6].

5.7. Modelowanie dynamiki i kinematyki dwuwahadła

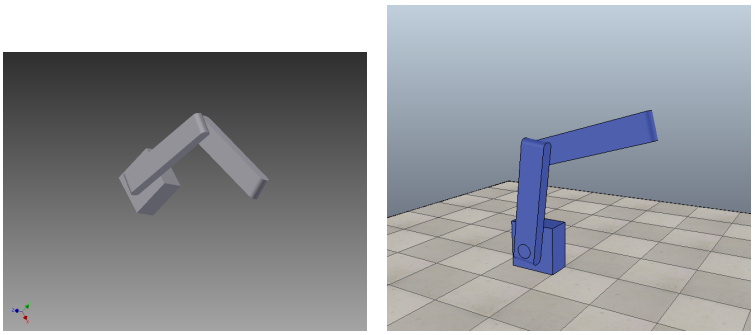
W tym rozdziale opisane zostały etapy tworzenia modelu manipulatora o dwóch przegubach obrotowych. W opisie pominięto mniej istotne szczegóły, zwracając uwagę na rzeczy związane bezpośrednio z modelowaniem dynamiki i kinematyki manipulatorów. W szczególności nie będzie tutaj podawany sposób bezpośredniej implementacji w środowisku V-REP.

5.7.1. Trójwymiarowy model manipulatora

V-REP dostarcza środowisko edytorskie w którym użytkownik może tworzyć własne modele robotów. Aczkolwiek dostarczany interfejs jest bardzo nieefektywny. Prostszym i mniej czasochłonnym rozwiązaniem, przedstawionym w tym rozdziale, jest import gotowego modelu z innego programu edytorskiego. Możemy dodawać obiekty zapisane w plikach *.dxf, *.obj, *.stl oraz *.3ds. W przedstawianym przykładzie model manipulatora stworzono w programie Autodesk Inventor Professional 2011. Jest to dużo bardziej kompleksowy program dostarczający intuicyjny interfejs użytkownika. Tworzenie w nim



Rysunek 5.7. Skrypty.



(a) Model w programie Inventor (b) Zaimportowany model do programu V-REP

Rysunek 5.8. Model dwuwahadła

modelu jest dużo bardziej wydajne i dokładne. Więcej informacji o tym oprogramowaniu można znaleźć na stronie producenta [2]. Na rysunku 5.8 przedstawiono model w programie Inventor oraz zaimportowany model w programie V-REP. Niestety, nie są importowane połączenia ruchome ani złożone człony. Obiekt zostaje wprowadzony w postaci jednej sztywnej bryły. Wymusza to na użytkowniku podzielenie modelu na mniejsze części. Po wykonaniu takiej operacji w niektórych przypadkach bryła zostanie podzielona na dużo więcej części niż jest to wymagane. W takim wypadku należy pogrupować odpowiednie części w celu uzyskania jednolitego członu manipulatora. Cała wymieniona tutaj procedura jest szczegółowo opisana w *Inverse kinematics tutorial* zamieszczonego w instrukcji do systemu [5].

5.7.2. Dynamika

Przygotowania do symulacji dynamicznej

W celu symulacji zachowań dynamicznych musimy utworzyć dodatkowe elementy nazwane w programie *Pure Shapes*. Producent systemu V-REP zaleca stosowanie wyłącznie tego typu elementów do symulacji dynamiki obiektów. Wynikać to może z dobrej optyma-

lizacji lub opisu obiektów *Pure Shapes* w symulatorze dynamiki. Jest to również pierwszy punkt z zaleceń producenta odnośnie tworzenia symulacji dynamicznych, przedstawionych w punkcie 5.3. W tym celu zaznaczamy interesujący nas przegub i edytujemy go w trybie *Triangle edit mode*. Po zaznaczeniu odpowiednich płaszczyzn tworzymy nowy obiekt typu *Pure Shape*. Cała procedura jest szczegółowo opisane w *Importing and preparing rigid bodies tutorial* w dokumentacji [5]. Tak utworzone elementy przesuwamy na warstwę numer 9, aby nie były widoczne podczas symulacji. We właściwościach stworzonych brył odznaczmy pozycję *static*. Dzięki temu nasze obiekty będą dynamicznie symulowane. Wstawiamy również odpowiednie wartości mas i momentów bezwładności, które bardzo łatwo można odczytać z programu Inventor.

Przeguby

Następnym etap tworzenia modelu jest wstawienie w odpowiednie miejsca przegubów. Po wykonaniu tej operacji należy wybrać odpowiedni tryb sterowania z tych przedstawionych w punkcie 5.3. W naszym przypadku wybieramy tryb *IK mode* i zaznaczamy opcję *hybrid mode*. Dzięki temu nasze przeguby będą mogły pracować w trybie odwrotnej kinematyki i ponad to będą również dynamicznie symulowane. Bardzo ważne jest ustawienie odpowiedniego momentu obrotowego. Za mała wartość skutkuje tym, że ramie obróci się w stronę podłoża lub w pewnym ustawieniu moment będzie za mały aby udźwignąć ramię.

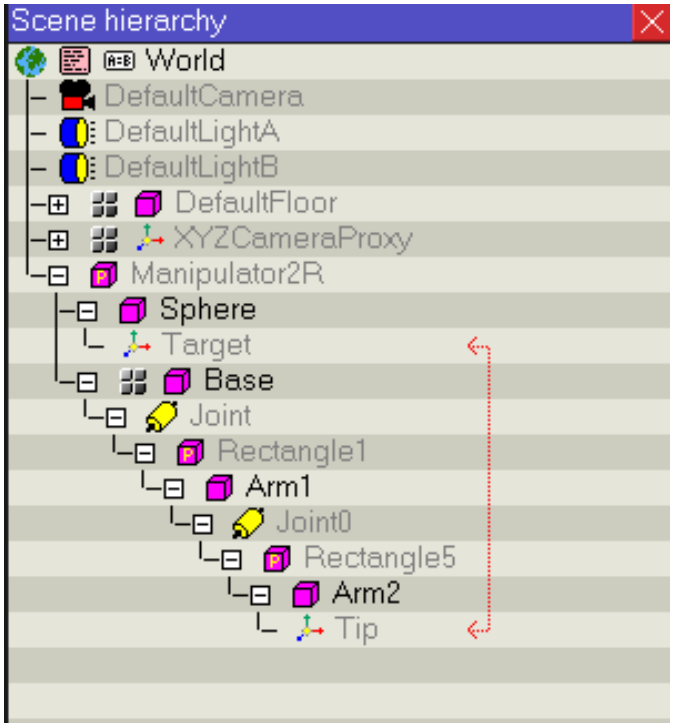
5.7.3. Kinematyka

Budowa łańcucha kinematycznego

Po wykonaniu czynności z poprzednich punktów możemy przejść do budowy łańcucha kinematycznego. W tym celu dodajemy dwa elementy typu *Dummy*. Jeden z nich ustawiamy na końcu naszego efektora. Będzie to końcowy punkt łańcucha kinematycznego. Nazywamy go *TIP*. Drugi element może pozostać w innym miejscu, aczkolwiek dla porządku ustawiamy go w tym samym miejscu. Nazywamy go *TARGET*. Następnie tworzymy połączenie pomiędzy dodanymi obiektami *Dummy*. Po tej operacji przechodzimy do tworzenia łańcucha kinematycznego. Zaczynamy od końcówki *TIP*. Następnie zaznaczamy człon efektora (statyczny) i tworzymy ten drugi rodzicem pierwszego. Potem zaznaczamy człon efektora (statyczny) i utworzony poprzez *Pure Shapes* człon efektora dynamiczny i tworzymy ten drugi rodzicem pierwszego. Kolejno dodajemy następne elementy modelu, włączając w to przeguby. Pilnujemy, aby przegub miał tylko jednego potomka. Po dodaniu wszystkich elementów powinniśmy otrzymać hierarchiczną strukturę, którą przedstawiono na rysunku 5.9. Szczegółowy opis przytoczonej procedury jest dostępny w dokumentacji [5].

Ustawienia kinematyki

Następnym etapem jest dodanie nowej grupy kinematycznej w oknie parametrów kinematyki 5.4. Dodajemy element *TIP* jako końcówkę łańcucha. Co możemy tutaj jeszcze osiągnąć to wybór algorytmu rozwiązywania zadania odwrotnej kinematyki. Zaleca się stosowanie szybkiego sposobu rozwiązywania *pseudo inverse*, a w przypadku nie uzyskania rozwiązania przejście na algorytm DLS. Można to osiągnąć przez dodanie bliźniaczej grupy kinematycznej, tak jak to zostało opisane w sekcji 5.4. W ten sposób przygotowany model jest gotowy do symulacji. Za każdym razem przesunięcie elementu *TARGET* spowoduje podążanie elementu *TIP*, a co za tym idzie rozwiązanie zadania odwrotnej kinematyki i podążanie, w ramach możliwości i ograniczeń, całego ramienia. Działanie całego modelu



Rysunek 5.9. Struktura hierarchiczna modelu

można ocenić podczas symulacji gotowego przykładu, dołączonego do niniejszego sprawozdania. W celu dokładnego zrozumienia rzeczy przytoczonych w powyższym opisie należy przestudiować dokmentację do systemu V-REP [5]. W szczególności *Inverse kinematic tutorial* i *Importing and preparing rigid bodies tutorial*.

5.8. Roboty mobilne i sensory

V-REP pozwala na symulację chyba każdego rodzaju robota mobilnego. Możemy znaleźć w nim gotowe modele robotów kołowych, kroczących czy latających. Zasady tworzenia symulacji dla takiego typu robotów są niemalże identyczne jak dla manipulatorów.

5.8.1. Tworzenie robotów mobilnych

Podobnie jak w przypadku manipulatora 2R, mamy możliwość stworzenia modelu od podstaw. Jednakże, również tutaj zalecane jest utworzenia modelu w innym programie wspierającym projektowanie. Inną możliwością jest skorzystanie z jednego z gotowych modeli dostepnych w V-REPie, co uczyniono w poniższym przykładzie.

5.8.2. Dynamika

W projektowaniu symulacji robotów mobilnych obowiązują te same zasady co dla manipulatorów. Stworzenie ukrytych elementów *Pure Shapes* jest wymagane. W dołączonym do niniejszego sprawozdania przykładzie można zaobserwować omawiane elementy na warstwie dziewiątej. Takie wymagania dotyczące hierarchicznej struktury są w mocy. Różniącą się kwestią jest tryb sterowania przegubów. Stosujemy tutaj jako tryb pracy przegubu (*Torque/Force mode*). W skryptach zadajemy, za pomocą odpowiedniej funkcji, prędkości obrotowe.

5.8.3. Sensory i skrypty

Bardzo dużym atutem korzystania z V-REPa jest możliwość dodania i symulacji działania sensorów. W połączeniu z robotem daje to możliwość budowy i testowania algorytmów sterowania robotami. W omawianym przykładzie dodano do robota dwa sensory: odległości i kamerę. Czujnik nacisku był wbudowany z przykład, aczkolwiek nie jest on w ogóle wykorzystywany. W zamieszczonym przykładzie został zaimplementowany przykładowy algorytm sterowania. Kamera reaguje jedynie na czerwone walce umieszczone na scenie. Po wykryciu takiego elementu robot skręca w prawo. W przypadku zielonych przeszkód działa czujnik odległości. Po wykryciu przeszkody robot skręca w lewo. Poniżej przedstawiono fragment skryptu napisanego na potrzeby tego prostego algorytmu.

```
result, distance=simReadProximitySensor(proximity_sensor) -- odczytujemy
czujnik odległości
render_result, table1 = simReadRenderingSensor(rendering_sensor)
-- odczytujemy dane z kamery
if (result>0) then backUntilTime=simGetSimulationTime()+4 end
-- Jeśli czujnik odległości coś wykrył
if(render_result==1) then time = simGetSimulationTime()+4 end
if(render_result == -1) then
simSetJointTargetVelocity(leftJointHandle,speed/8)
simSetJointTargetVelocity(rightJointHandle,-speed/8)
end

if (backUntilTime<simGetSimulationTime() and time <simGetSimulationTime())
then
-- Nic nie zostało wykryte jedziemy do przodu
simSetJointTargetVelocity(leftJointHandle,speed)
simSetJointTargetVelocity(rightJointHandle,speed)
elseif(result>0) then
-- Wykryto przeszkodę przez czujnik odległości, skręcamy w lewo
simSetJointTargetVelocity(leftJointHandle,-speed/8)
simSetJointTargetVelocity(rightJointHandle,speed/8)
end

if (table1[12]>0.3) then -- wykryto czerwony kolor
simSetJointTargetVelocity(leftJointHandle,speed/8)
simSetJointTargetVelocity(rightJointHandle,-speed/8)
end
```

Powyższy skrypt jest wywoływany w trybie bez wątkowym, co oznacza że jest on wykonywany w każdym kroku symulacji. Na początku odczytywane są dane z czujników, a następnie wykonywane odpowiednie czynności. Działanie algorytmu można ocenić podczas działania.

5.9. Podsumowanie

Podsumowując, V-REP jest niewątpliwie oprogramowaniem wartym uwagi. Jego hybrydowa natura pozwala na symulację zarówno dynamiki jak i kinematyki robotów stacjo-

narnych i mobilnych. Środowisko edytorskie pozwala na konstruowanie własnych modeli lub import gotowych, wykonanych w innych programach.

Dużą zaletą jest możliwość korzystania z sensorów. To w połączeniu z językiem skryptowym LUA umożliwia wyposażenie robotów w algorytmy sterowania. Udostępnianie API do języka C/C++ pozwala na dołączenie aplikacji klienckiej, mającej cechy nadrzędnego sterownika robotów. Niestety w V-REPie nie doszukano się możliwości symulacji przegubów lub członów elastycznych. Ponadto, można obserwować i rejestrować przebiegi różnych wartości podczas symulacji na wykresach.

Proces tworzenia modeli 3D robotów w wbudowanym interfejsie jest nieefektywny i sprawiający wrażenie niezrozumiałego dla początkującego użytkownika. Poniekąd wadą jest również brak możliwości debbugowania napisanych skryptów. Brakuje także w dokumentacji przykładów użycia funkcji z API. Jednakże, V-REP jest systemem zdecydowanie godnym polecenia. Jego uniwersalność przyciąga użytkowników zainteresowanych manipulatorami i wieloma rodzajami robotów mobilnych.

Bibliografia

[1] Bullet Physics. www.bulletphysics.org.
[2] Autodesk Inventor Professional 2011. www.autodesk.pl.
[3] Lua. www.lua.org.
[4] Open Dynamics Engine. www.ode.org.
[5] Dokumentacja systemu v-rep. www.v-rep.eu/helpFiles/index.html.
[6] V-rep. www.v-rep.eu.

6. Webots

Bartosz Kułak

Środowisko Webots jest narzędziem stworzonym do modelowania, programowania i symulowania działania robotów stacjonarnych i mobilnych. Przy jego pomocy użytkownik może projektować złożone układy robotyczne, wykorzystując dużą bazę elementów takich jak bryły sztywne, serwomechanizmy czy sensory. Wszystkie zaprojektowane roboty i zespoły robotyczne użytkownik może zaprogramować wykorzystując wbudowane środowisko programistyczne [3].

6.1. Możliwości środowiska Webots

Środowisko Webots zostało opracowane z myślą o symulacji działania robotów. Użytkownik ma możliwość projektowania mechaniki robotów oraz ich kinematyki. Wynik symulacji jest wstępem do zaprojektowania robotów rzeczywistych, gdyż środowisko umożliwia zdobycie wiedzy o zachowywaniu się symulowanych maszyn. Dodatkowo środowisko oferuje możliwość programowania robotów przy pomocy wbudowanego środowiska programistycznego. Programowanie robotów odbywa się w języku C/C++. Webots zawiera także wykrywanie kolizji obiektów. Wizualizacja projektowanych robotów odbywa się w głównym oknie programu. Sam program nie zawiera opcji rejestracji danych liczbowych czy generacji wykresów (np. przebiegu zmiennych przegubowych). Użytkownik może w programowanym sterowniku skorzystać z odpowiednich funkcji umożliwiających zapis do pliku zmiennych liczbowych. Dzięki temu otrzymane dane można poddać dalszej analizie w programach zewnętrznych (np. w arkuszu kalkulacyjnym).

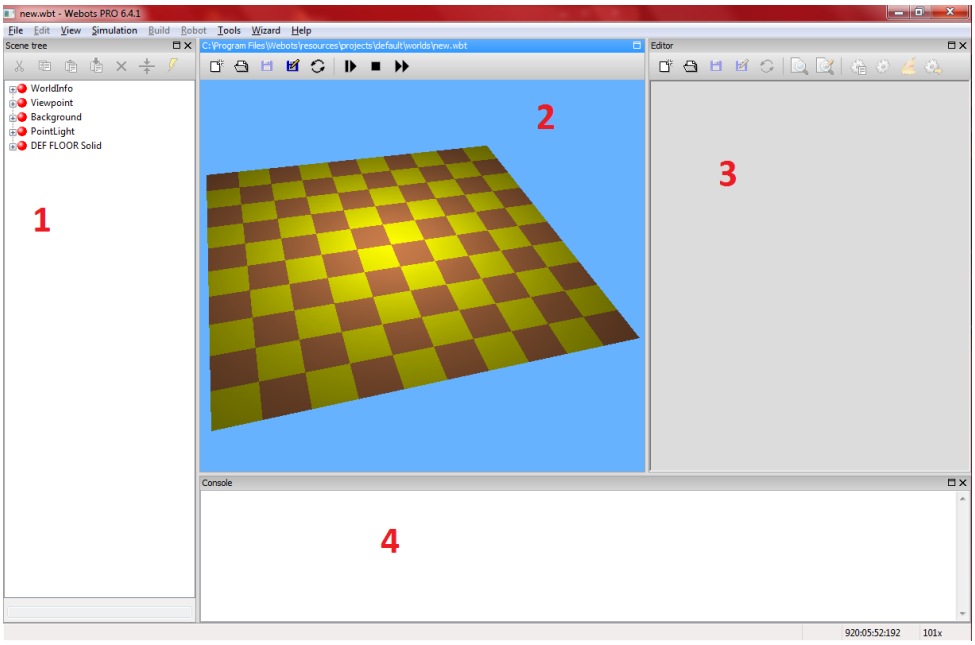
6.2. Instalacja

Program Webots w wersji 6.4.1 jest środowiskiem komercyjnym dostępny na zasadach darmowej licencji z pewnymi ograniczeniami. Użytkownik może projektować roboty i zespoły robotyczne, jednak nie ma możliwości kompilacji sterowników. Ponad to po rejestracji na stronie producenta [2] istnieje możliwość uzyskania 30-dniowego okresu próbnego, w czasie którego użytkownik ma pełny dostęp do wszystkich funkcji programu. Producent nie udostępnia wersji studenckiej programu.

6.3. Uruchomienie programu

Po uruchomieniu programu pojawia się okno główne pokazane na rysunku 6.1. W programie możemy wyróżnić 4 następujące obszary:

1. sekcja `Scene Tree`, w której znajdują się deklaracje wszystkich obiektów (podłoże, bryły sztywne, roboty, a także punkty oświetlenia i definicje działających na scenie sił),
2. okno symulacji (w nim wyświetlany jest efekt modelowania i programowania),



Rysunek 6.1. Okno główne programu Webots

3. sekcja `Editor`, która umożliwia pisanie, kompilowanie i uruchamianie sterowników,
4. sekcja `Console` (wyświetla wszelkie komunikaty o błędach i ostrzeżenia).

Poruszanie się wewnątrz środowiska Webots jest intuicyjne i nie powinno sprawiać większych trudności po krótkim zapoznaniu się z jego budową.

6.4. Pierwsze kroki w programie Webots

W celu rozpoczęcia pracy ze środowiskiem Webots należy z paska menu wybrać opcję `File -> New world`. Następnie należy zapisać projekt w osobnym folderze.

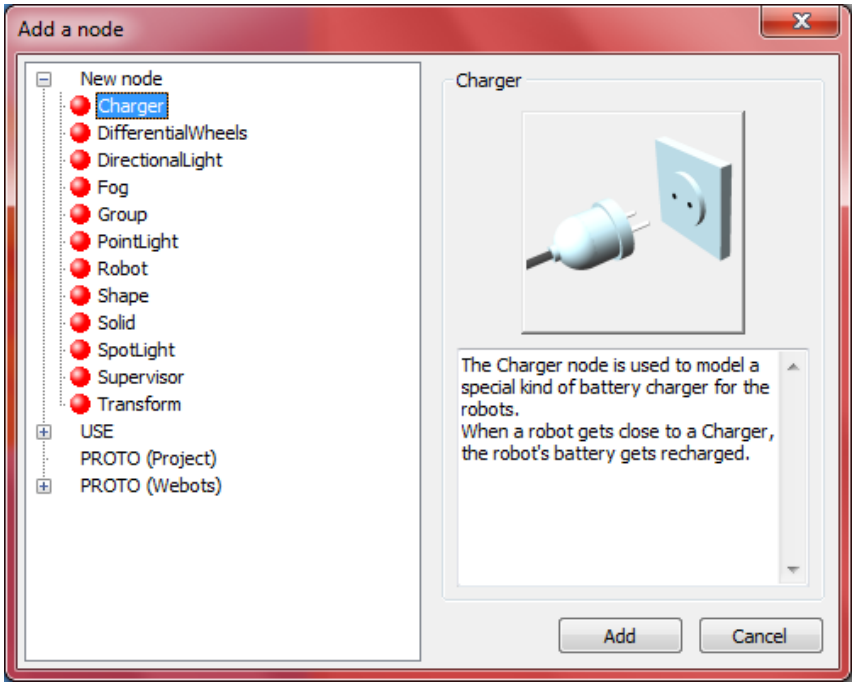
6.4.1. Projektowanie robotów

Po stworzeniu nowej sceny w sekcji `Scene tree` ukazują się pola opisujące obiekty znajdujące się na scenie:

— `WorldInfo` – tutaj znajdują się definicje działających sił. Możemy także wpisać informacje o autorze projektu.
— `Viewpoint` – określa położenie punktu obserwacji sceny.
— `Background` – określa kolor tła w oknie symulacji
— `PointLight` – definiuje położenie punktu oświetlenia sceny (użytkownik może sam tworzyć punkty oświetlenia sceny w dowolnej liczbie).
— `DEF FLOOR Solid` - definicja podłoża sceny (możemy tutaj dowolnie zmieniać rozmiar podłoża, jego położenie i obrót oraz kolor).

W celu dodania obiektu należy kliknąć przycisk `Add new` znajdujący się po prawej stronie sekcji `Scene tree`. Po kliknięciu na przycisk pojawia się nowe okno przedstawione na rysunku 6.2. Użytkownik ma do wyboru opcje, z których warto wyróżnić:

— `DifferentialWheels` – napęd różnicowy stosowany w robotach jeżdżących (np. minisumo),
— `Group` – opcja umożliwia tworzenie skomplikowanych układów złożonych z kilku elementów, które będą się zachowywały jak jeden element,



Rysunek 6.2. Dodawanie nowego obiektu

— `Robot` – tworzenie nowego robota,
— `Shape` – dodawanie obiektów widocznych na scenie,
— `Solid` – dodawanie brył sztywnych na scenę
— `Transform` – opcja definiująca translację i rotację jednego obiektu względem drugiego.

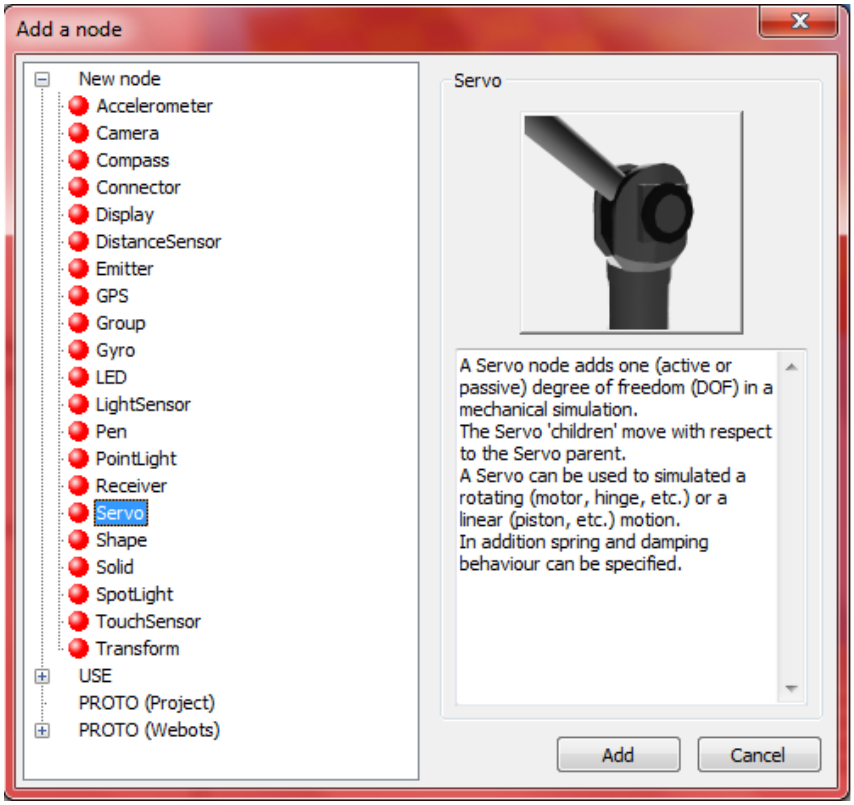
Tworzenie nowego robota zaczynamy od wyboru `DifferentialWheels` dla robotów mobilnych jeżdżących lub `Robot` dla robotów stacjonarnych, latających czy kroczących.

Opcja `DifferentialWheels` pozwala na symulację robotów napędzanych mechanizmem różnicowym. Użytkownik może zdefiniować tutaj maksymalną prędkość, przyspieszenie i wielkość kół. Napęd różnicowy potrzebuje do działania zdefiniowania dwóch brył, które będą pełniły role kół. W tym celu w sekcji `Children` należy dodać dwa obiekty typu `Solid` i nadać im odpowiednie kształty (`Shape`). Należy pamiętać, aby rozmiar kół deklarowany w sekcji `Shape` był taki sam jak zdefiniowany w `DifferentialWheels`.

Po wyborze opcji `Robot` użytkownik może zaprojektować praktycznie każdy typ robota. Użytkownik musi jednak modelowanie zacząć od podstaw, gdyż program nie oferuje żadnych schematów do wykorzystania (można jedynie wzorować się na gotowych projektach, co w większości przypadków skróci czas pracy do minimum). Wewnątrz sekcji `Robot` możemy definiować połączenia pomiędzy obiektami (połączenia sztywne lub przeguby przesuwne i obrotowe, przy wykorzystaniu `Servo`), wykorzystywać takie elementy jak akcelerometry, kamery, GPS, żyroskopy czy czujniki odległości (rysunek 6.3).

6.4.2. Programowanie sterowników

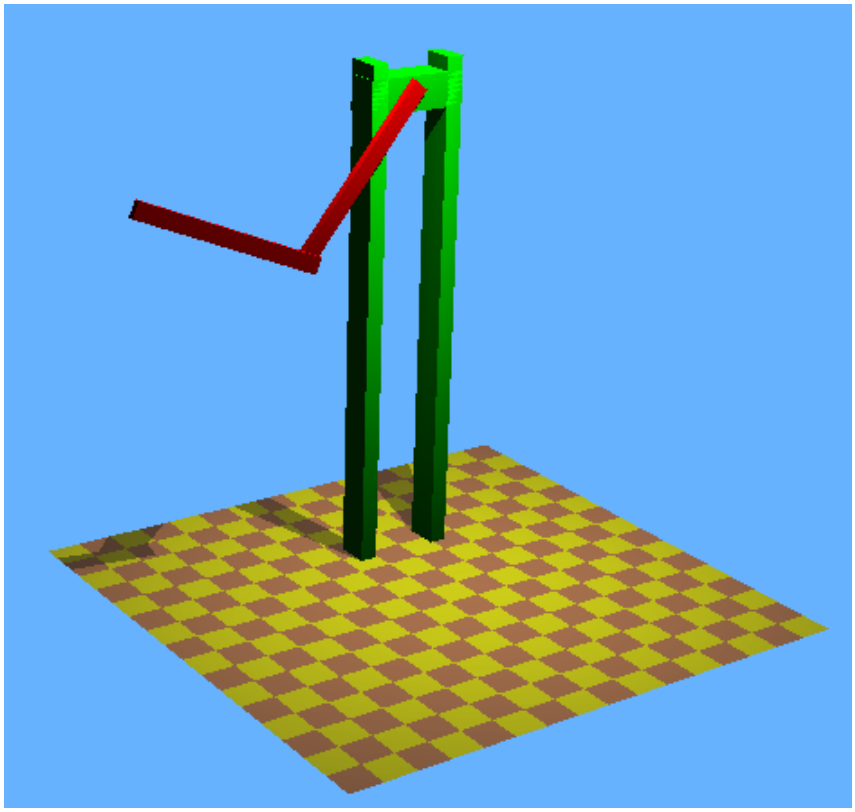
Zaprojektowane roboty lub układy robotyczne mogą zostać zaprogramowane przez użytkownika. Programowanie sterowników odbywa się w sekcji `Editor`. W celu omówienia sposobu programowania przyjrzyjmy się przykładowemu fragmentowi kodu z wydruku 6.1.



Rysunek 6.3. Wybór elementów podczas projektowania robota

Wydruk 6.1. Przykładowy sterownik

```
#include <webots/robot.h>
#include <webots/differential_wheels.h>
#include <webots/distance_sensor.h>

#define SPEED 60
#define TIME_STEP 64

int main()
{
  wb_robot_init();

  WbDeviceTag ir0 = wb_robot_get_device("ir0");
  WbDeviceTag ir1 = wb_robot_get_device("ir1");
  wb_distance_sensor_enable(ir0, TIME_STEP);
  wb_distance_sensor_enable(ir1, TIME_STEP);

  while(wb_robot_step(TIME_STEP)!=-1) {

    double ir0_value = wb_distance_sensor_get_value(ir0);
    double ir1_value = wb_distance_sensor_get_value(ir1);
    double left_speed, right_speed;
    left_speed = -SPEED;
    right_speed = -SPEED / 2;
```



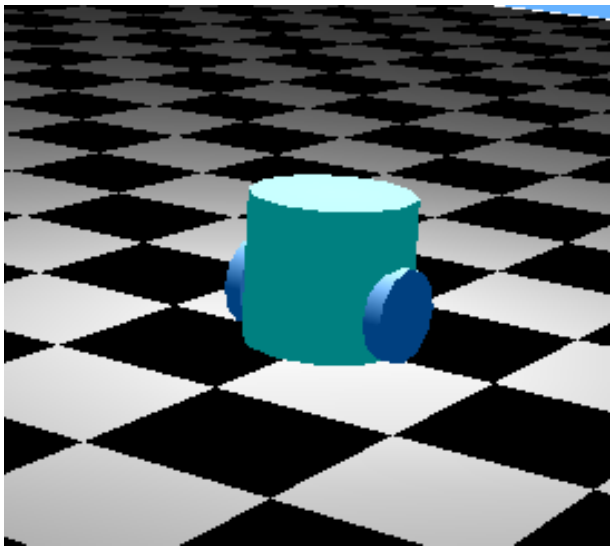
Rysunek 6.4. Przykładowy model podwójnego wahadła

```
24
      wb_differential_wheels_set_speed(left_speed, right_speed);
26  }
    return 0;
28 }
```

Na początku każdego sterownika użytkownik powinien dołączyć biblioteki `Webots`, które umożliwiają synchronizację i komunikację sterowników z symulacją w oknie programu. Dodatkowo, jeśli sterownik tego wymaga, użytkownik może dołączyć dowolne biblioteki C++. Jako pierwsze w funkcji `main` znajduje się wywołanie `wb_robot_init()`. Jest to funkcja, która inicjuje narzędzia programistyczne związane z symulacją. Dalej widzimy szereg funkcji typowych dla sterowników pisanych w programie Webots (takich jak funkcje `wb_robot_get_device`, `wb_distance_sensor_enable`, czy w końcu funkcję `wb_differential_wheels_set_speed`). Wykorzystywanie takich funkcji jest konieczne przy pisaniu sterowników, a przykłady ich zastosowania są dostępne w dołączonych przez producenta sterownikach, a także w podręczniku użytkownika Webots [1].

6.5. Modelowanie podwójnego wahadła

W programie Webots zaprojektowano przykładowy model podwójnego wahadła (rysunek 6.4). Podstawa wahadła została wykonana przy pomocy narzędzia `Group`. Natomiast samo wahadło składa się z dwóch elementów typu `Solid` z przypisanymi odpowiednimi kształtami i masami. Przeguby obrotowe zostały zrealizowane za pomocą narzędzia `Servo`. Zamodelowanie w taki sposób umożliwia obserwację zachowania się podwójnego wahadła



Rysunek 6.5. Przykładowy monocykl

podczas swobodnego ruchu, z zadanej pozycji początkowej (użytkownik może sam zadać pozycję początkową zmieniając odpowiednie wartości w narzędziu `Servo`).

6.6. Monocykl

Przykładowy model monocykla znajduje się na rysunku 6.5 Monocykl został zaprojektowany w oparciu o narzędzie `DifferentialWheels`. Ciało robota jest obiektem typu `Solid` o kształcie walca.

6.7. Podsumowanie

Środowisko Webots jest rozbudowanym środowiskiem symulującym działanie robotów. Z powodzeniem może być wykorzystywane przy projektowaniu i modelowaniu robotów. Jego niewątpliwą zaletą jest dość prosta i intuicyjna obsługa oraz możliwość pisania zaawansowanych sterowników w powszechnie znanym języku C/C++. Dostępność dużej ilości opcji podczas projektowania pozwala na projektowanie złożonych układów robotycznych.

Wśród wad programu można wymienić niedoskonały system wykrywania kolizji (przenikanie się obiektów) oraz mimo wszystko konieczność żmudnej pracy podczas projektowania robotów. Program nie oferuje opcji importu gotowych obiektów z innych środowisk, ani też nie pozwala na eksport tworzonych projektów do otwartych formatów.

Bibliografia

[1] Webots Guide. http://www.cyberbotics.com/guide.pdf.
[2] Download Webots. http://www.cyberbotics.com/download.
[3] Webots. http://www.cyberbotics.com/overview.

7. Player/Stage

Łukasz Czyż

Player i Stage to darmowe programy umożliwiające symulację zarówno jednego, jak również kilku robotów mobilnych jednocześnie. Pierwszy z nich działa na zasadzie klient-serwer stanowiąc abstrakcyjną warstwę fizyczną od sprzętu. Umożliwia to pisanie własnych, przenośnych programów. Stage pozwala na przeprowadzenie dwuwymiarowej symulacji. Do poprawnej pracy niezbędne jest uruchomienie w pierwszej kolejności Playera, a następnie pluginu Stage'a. Wywołanie samego Playera jest możliwe tylko w przypadku przeprowadzenia symulacji na rzeczywistym obiekcie.

Poniżej przedstawiono zakres możliwości oferowanych przez wyżej wymienione aplikacje na przykładzie robota mobilnego Pioneer wraz z czujnikiem laserowym LMS200 firmy Sick. Poradnik ten zawiera instrukcje dla osób, które dopiero zaczynają swoją przygodę z Playerem/Stagem. Dodatkową pomoc można uzyskać na stronie domowej projektu pod adresem `http://playerstage.sourceforge.net`

7.1. Player

Player to oprogramowanie typu serwer. Stanowi on warstwę obsługi sprzętowej. Rolę klienta pełni program, który komunikuje się z serwerem przez protokół TCP/IP. Oficjalnie wspierane są biblioteki dla języków C, C++, Python [2]. Zadaniem Playera jest komunikacja z sensorami czy napędami robota. Pozwala to na łatwe zarządzanie czujnikami i innymi urządzeniami, jak na przykład kamerą zamontowaną na robocie. Komunikacja między nimi a Playerem jest możliwa dzięki wykorzystaniu driverów. Dużym ułatwieniem jest możliwość skorzystania z gotowych driverów jak na przykład:

— Garcia firmyAcroname,
— Khephera firmy K-Team,
— Obot d100 firmy Botrics,
— Clodbuster firmy UPenn GRASP,
— 914 PC-BOT firmy White Box Robotics,
— NOMAD200 firmy Nomadics,
— Platforma mobilna Erratic firmy Videre Design[1].

Każdy z nich udostępnia zbiór interfejsów, które można wykorzystać pisząc własny program sterujący pracą robota. Pozwala to na tworzenie przenośnych skryptów działających na każdym urządzeniu, które używa tego samego drivera. Niewątpliwą zaletą Playera jest możliwość przeprowadzenia symulacji na rzeczywistym obiekcie z pominięciem środowiska symulacyjnego.

7.1.1. Instalacja

Najnowszą wersję Playera można pobrać ze strony producenta. Instalacja możliwa jest na Linuksie, Solarisie lub BSD. Niestety nie ma możliwości korzystania z Playera/Stage'a

[1] Listę wszystkich obsługiwanych driverów można znaleźć na stronie producenta Playera/Stage'a [1].

w systemach MS Windows. Obecnie najnowszą wersją Playera jest 3.0.2 [3]. Po rozpakowaniu archiwum, należy wejść do katalogu playera, utworzyć w nim folder `build`, przejść do niego i wykonać instrukcje:

```
cmake ..
make
make install
```

Typowym powodem niepomyślnego przebiegu instalacji jest brak wymaganych pakietów. Informacje o tym, które z nich wymagają doinstalowania zostaną wyświetlone w komunikacie o błędzie. Innym problemem napotykanym podczas instalacji jest brak zgodności wersji instalowanych pakietów z instalowaną wersją Playera lub Stage'a, o czym nie wspomniano w dokumentacji znajdującej się na stronie producenta.

7.2. Stage

Stage stanowi dwuwymiarowe[2] środowisko symulacyjne robotów mobilnych[3]. Umożliwia ono symulację zarówno pojedyńczego robota jak i grupy robotów poruszających się w wybranym środowisku. Ponadto pozwala na wybór map otoczenia oraz czujników, których chcemy użyć podczas przeprowadzania symulacji.

7.2.1. Instalacja

Podobnie jak Player tak i Stage może zostać zainstalowany na Linuksie, BSD lub Solarisie. Najnowszą wersją jest Stage v3.2.2 [4]. Przed przystąpieniem do instalacji należy zainstalować następujące biblioteki:
— pkg-config,
— FLTK 1.1.x,
— OpenGL,
— libpng,
— ltdl (Libtool).
Po zainstalowaniu powyższych bibliotek i rozpakowaniu archiwum Stage'a, można przejść do kompilacji i instalacji. W tym celu należy w katalogu Stage'a wykonać instrukcje:

```
cmake ..
make
make install
```

Jeśli instalacja przebiegła pomyślnie, można przejść do pierwszego uruchomienia Stage'a. W przypadku wystąpienia błędów może okazać się konieczne doinstalowanie dodatkowych bibliotek/pakietów.

7.2.2. Konfiguracja

Konfiguracja odbywa się w kilku etapach poprzez edytowanie wybranych plików. Należą do nich:

[2] Modelowane obiekty są trójwymiarowe jednak podczas symulacji zmieniają się tylko dwie współrzędne. Możliwość oglądania symulacji w perspektywie ilustruje rysunek 7.2. Twórcy nazwali ją jako przestrzeń 2,5-wymiarową.

[3] Autorzy Playera oraz Stage'a stworzyli także Gazebo - środowisko umożliwiające symulację robotów w przestrzeni trójwymiarowej.

— plik *.cfg zawierający informacje o udostępnionych interfejsach oraz o nazwie pliku opisującego symulowany świat,
— plik *.world definiujący rozmiar i położenie robota oraz okna symulacji; pozwala także na wybór obrazu sceny w formacie .png co ilustruje wydruk 7.2,
— plik *.inc opisujący wygląd robota lub innego urządzenia wykorzystywanego w symulacji.

Zawartość pliku simple.cfg przedstawia wydruk 7.1. Zawiera on informację o udostępnionych interfejsach m.in.:
— position2d – odpowiada za sterowanie napędami robota,
— laser – odczytuje dane ze skanerów laserowych,
— graphics2d – odpowiada za rysowanie figur na mapie,
— sonar – odczytuje dane z sonarów,
— camera – przekazuje obraz z kamery.

Każdy czujnik może zostać użyty podczas symulacji więcej niż jeden raz. W tym celu zostały wprowadzone w pliku `simple.cfg` liczebniki porządkowe określające ich numerację. Konfiguracja przykładowego robota została zapisana w dwóch plikach: `pioneer.inc` (opis robota Pioneer) oraz `sick.inc` (konfiguracja dalmierza laserowego LMS200 firmy Sick, zamontowanego na robocie).

Wydruk 7.1. Zawartość pliku simple.cfg

```
# Desc: Player sample configuration file [...]

# load the Stage plugin simulation driver
driver
(
  name "stage"
  provides [ "simulation:0" ]
  plugin "stageplugin"

  # load the named file into the simulator
  worldfile "simple.world"
)

# Create a Stage driver [...]
driver
(
  name "stage"
  provides [ "position2d:0" "laser:0" "speech:0"
  "graphics2d:0" "graphics3d:0" ]
  model "r0"
)
```

Wydruk 7.2. Zawartość pliku simple.world

```
[...]

window
(
```

```
  size [ 635.000 666.000 ] # in pixels
  scale 37.481   # pixels per meter
  center [ -0.019  -0.282 ]
  rotate [ 0  0 ]

  show_data 1              # 1=on 0=off
)

# load an environment bitmap
floorplan
(
  name "cave"
  size [16.000 16.000 0.800]
  pose [0 0 0 0]
  bitmap "bitmaps/cave.png"
)

pioneer2dx
(
  # can refer to the robot by this name
  name "r0"
  pose [ -7 -7 0 45 ]

  [...]
)
```

7.3. Przykład

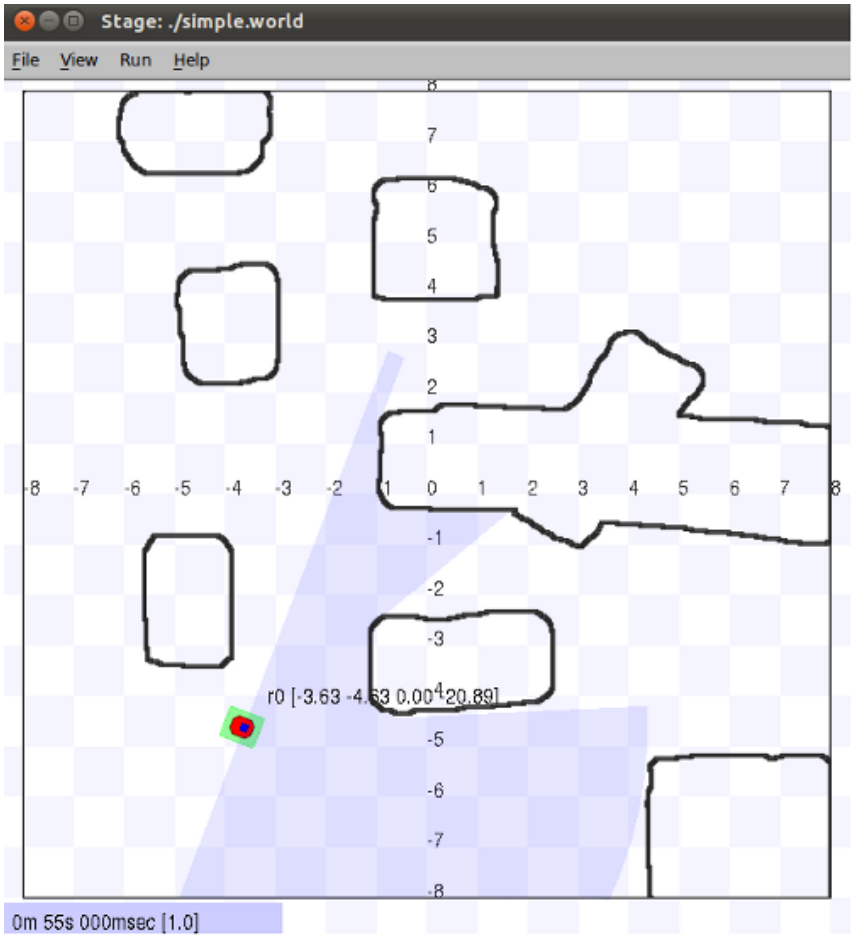
Aby uruchomić przykładową symulację należy przejść do katalogu Stage/worlds i wykonać polecenie:

```
player simple.cfg
```

co powinno przynieść rezultat pokazany na wydruku 7.3.

Wydruk 7.3. Uruchomienie Player/Stage

```
Registering driver
Player v.3.0.2

* Part of the Player/Stage/Gazebo Project [...]

invoking player_driver_init()...
 Stage driver plugin init

 ** Stage plugin v3.2.2 ** [...]

success
 Stage plugin:  6665.simulation.0 is a Stage world
 [Loading ./simple.world][Include pioneer.inc]
```



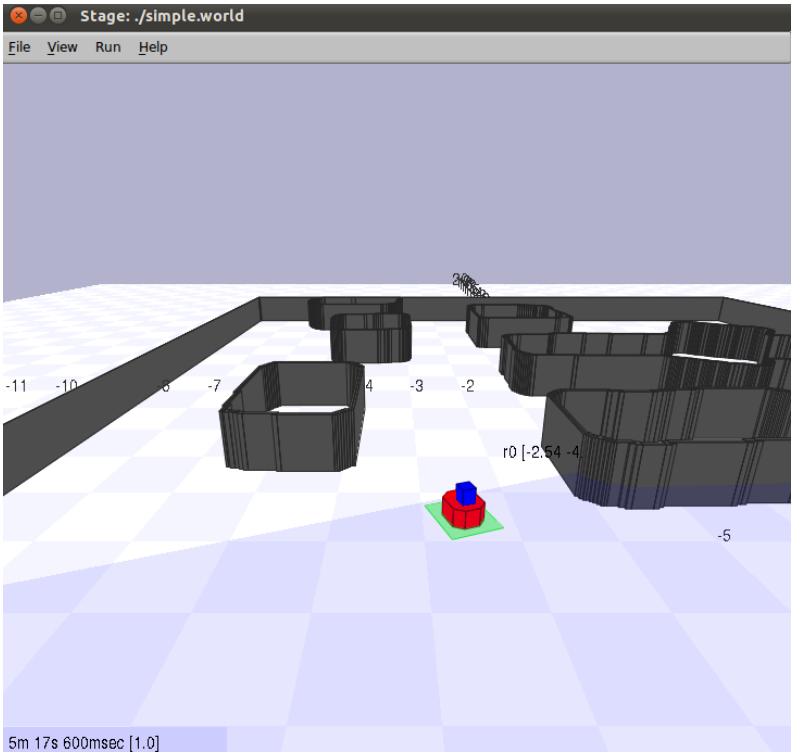
Rysunek 7.1. Okienko symulacji Stage

```
 [Include map.inc][Include sick.inc]

 Stage plugin:  6665.position2d.0 is "r0"
 Stage plugin:  6665.laser.0 is "r0.laser:0"
 Stage plugin:  6665.speech.0 is "r0"
 Stage plugin:  6665.graphics2d.0 is "r0"
 Stage plugin:  6665.graphics3d.0 is "r0"
listening on 6665
Listening on ports: 6665
```

Wynika z niego, iż najpierw uruchomiony zostaje serwer Playera a następnie plugin Stage'a. W efekcie otrzymujemy okienko z symulacją widoczne na rysunku 7.1, przedstawiające symulację robota Pioneer wraz z czujnikiem laserowym LMS200 firmy Sick. Z menu programu Stage'a można zmienić widok z dwuwymiarowego na 2,5-wymiarowy, co ilustruje rysunek 7.2.

7.4. Podsumowanie

Player/Stage posiadają zarówno zalety jak i wady. Do pierwszej grupy zaliczyć można m.in.:



Rysunek 7.2. Zmiana perspektywy okna symulacji

— fakt, iż są to środowiska bezpłatne, rozpowszechniane na zasadzie GNU General Public License,
— symulacja zachowania wielu robotów mobilnych,
— możliwość pisania własnych programów,
— udostępnienie wielu gotowych driverów,
— opcja wgrania programu do rzeczywistego obiektu.

Wśród wad wymienić można m.in.:

— typowe błędy podczas instalacji związane z brakującymi pakietami lub ich niezgodnością z wersją Playera/Stage'a,
— brak możliwości przeprowadzenia symulacji dla pozostałej grupy robotów, w tym manipulatorów,
— ograniczenie systemów operacyjnych do Linuksa, BSD oraz Solarisa.

Po pomyślnym etapie instalacji, Player oraz Stage stwarzają ogromne możliwości w zakresie symulacji robotów mobilnych. Twórcy cały czas pracują nad nowymi wersjami programów poprawiając ich funkcjonalność.

Bibliografia

[1] Zbiór driverów obsługiwanych przez Playera. http://playerstage.sourceforge.net/doc/Player-3.0.2/player/supported_hardware.html.
[2] Player – wspierane biblioteki. http://playerstage.sourceforge.net/doc/Player-3.0.2/player/group__clientlibs.html.
[3] Player w wersji 3.0.2. http://sourceforge.net/projects/playerstage/files/Player/3.0.2/.
[4] Stage w wersji 3.2.2. http://sourceforge.net/projects/playerstage/files/Stage/3.2.2/.

8. Autodesk Inventor

Michał Kot

Produkty CAD 3D Autodesk Inventor zawierają pełny i elastyczny zestaw oprogramowania do projektowania elementów mechanicznych 3D, symulowania produktów, tworzenia narzędzi i prezentacji projektów. Program Inventor rozszerza możliwości projektowania 3D o cyfrowe prototypowanie, umożliwiając tworzenie dokładnego modelu trójwymiarowego, który ułatwia projektowanie, wizualizowanie i symulowanie produktów przed ich wykonaniem. Cyfrowe prototypowanie przy użyciu programu Inventor pomaga projektować lepsze produkty, zmniejszać koszty projektowe i szybciej wprowadzać produkty na rynek [1].

8.1. Zastosowania Autodesk Inventor

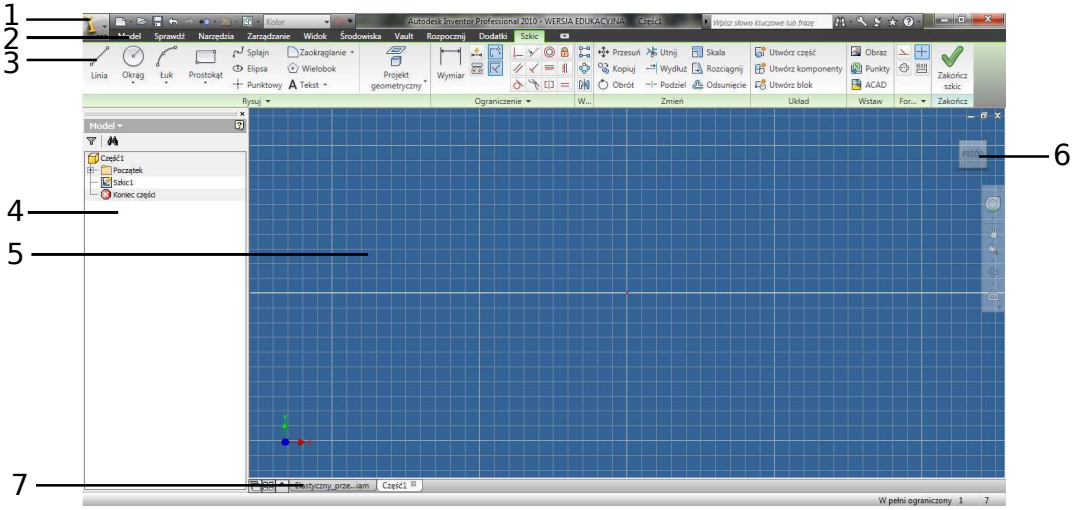
Autodesk Inventor jest środowiskiem konstrukcyjnym, które zostało opracowane przede wszystkim do zastosowań praktycznych. Modelowanie konkretnych obiektów z rzeczywistych materiałów, posiadających masę i bezwładność prowadzi do stworzenia rysunków złożeniowych. Umożliwia to przygotowanie kompletnej charakterystyki elementów pozwalającej na fizyczną realizację każdego z komponentów. Warto również wspomnieć, że Inventor posiada zaimplementowany moduł wykrywania kolizji. Mimo iż służy on jedynie do wskazywania już powstałych kolizji (nie zapobiega im w czasie rzeczywistym) bardzo przydaje się w procesie modelowania, gdyż pozwala użytkownikowi zauważyć nachodzące na siebie elementy konstruowanego obiektu.

Kolejnym istotnym (szczególnie z punktu widzenia robotyka) narzędziem jest symulacja dynamiczna, która służy do prostej analizy i weryfikacji układu poddanego działaniu zewnętrznych sił i momentów obrotowych (rozdział 8.5). Moduł ten pozwala na zarejestrowanie animacji, która może być zapisana do oddzielnego pliku, co pozwala na zaprezentować obiekt w atrakcyjny sposób. Inventor posiada także możliwość eksportowania własnych modeli do formatu XML, który z kolei może być wykorzystany przez bardziej złożone środowiska symulacyjne, takie jak Matlab (szerzej w rozdziale 8.7).

8.2. Instalacja

Niniejszy dokument został stworzony na podstawie wersji Autodesk Inventor 2010. Na użytek własny dostępna jest wersja studencka, którą można pobrać ze strony producenta [1]. W ramach instalacji warto zaopatrzyć się w oprogramowanie dodatkowe, którym jest Autodesk Vault. Jest to odpowiednik systemów kontroli wersji, stworzony przez grupę Autodesk i dostosowany do funkcjonalności programu AI.

Po instalacji gotowi jesteśmy do pracy z programem. Co ważne, należy uzbroić się w cierpliwość przy uruchamianiu programu — dwukrotne kliknięcie ikony programu z pozoru nie przynosi efektu — program po prostu uruchamia się dość długo.



Rysunek 8.1. Okno główne programu

8.3. Okno główne programu

Przykładowe okno główne, już po rozpoczęciu pracy z nową częścią, zostało przedstawione na rysunku 8.1. Najważniejsze komponenty, które należy w nim wyróżnić to:

1. ikonka I_{PRO} służąca do operacji na plikach,
2. zakładki odpowiadające różnym funkcjom programu,
3. wstążki – konkretne opcje wykorzystywane w modelowaniu,
4. przeglądarka modelu przedstawiająca strukturę modelu i pozwalająca na dostęp do podzespołów,
5. okno modelowania programu, w którym wykonujemy operacje bezpośrednio na modelu,
6. narzędzia służące do kontroli widoku,
7. zakładki odpowiadające otwartym aktualnie plikom.

Przy pracy z modelem bardzo istotne są narzędzia do kontroli widoku umieszczone po prawej stronie — m.in. trójwymiarowa kostka, która pozwala na bardzo wygodne obracanie modelu — klikając w jej wierzchołki, boki lub krawędzie mamy możliwość wybrania płaszczyzny modelu, której chcemy się przyjrzeć. Warto także wspomnieć, że podczas pracy z konkretną funkcjonalnością (wiązania, wymiarowanie, rysowanie itp.) każdą operację anulujemy za pomocą klawisza *ESC* - domyślnie po wykonaniu czynności raz wykonujemy ją aż do momentu anulowania polecenia.

8.4. Rodzaje plików

Przed przystąpieniem do pracy warto zapoznać się z rodzajami i rozszerzeniami plików na których pracuje Autodesk Inventor. Najważniejsze z nich to:

— pliki części (z rozszerzeniem *.ipt, opisane w rozdziale 8.4.1),
— pliki zespołu (z rozszerzeniem *.iam, opisane w rozdziale 8.4.2,
— pliki rysunku (z rozszerzeniem *.idw, zawierające rysunki złożeniowe naszego modelu),
— pliki prezentacji (z rozszerzeniem *.ipn, zawierające prezentację naszego modelu np. w postaci animacji).

8.4.1. Plik części

Plik części (z rozszerzeniem *.ipt) zawiera opis pojedynczego elementu, z których to budowane są bardziej złożone konstrukcje (rozdział 8.4.2) . Jako części definiuje się możliwie najprostsze podzespoły (śrubka, belka itp.), które mogą być wyprodukowane niezależnie. Część sama w sobie nie posiada elementów ruchomych i jako całość stanowi sztywną bryłę.

Szkic dwuwymiarowy

Konstruowanie części zawsze rozpoczyna się od utworzenia szkicu (modelu 2D), który będzie odpowiadać jednej ze ścian. Ważne jest, aby rysując jakąkolwiek część pamiętać o wymiarowaniu (ustawianiu konkretnych wartości długości i kątów) przy pomocy ikonki z menu. Po ukończeniu pracy nad szkicem wybieramy ikonkę *Zakończ szkic dwuwymiarowy*.

Wyciągnięcie proste

Dwuwymiarowemu szkicowi można teraz nadać trzeci wymiar – poprzez wykorzystanie ikonki *Wyciągnięcie proste*. W nowo otwartym menu ustawiamy głębokość wyciągnięcia, a następnie wybieramy wyciąganą płaszczyznę - na początku jest to zawsze pierwszy szkic. W tym momencie mamy gotową pierwszą trójwymiarową część, którą można poddawać dalszym modyfikacjom.

Modyfikacja części

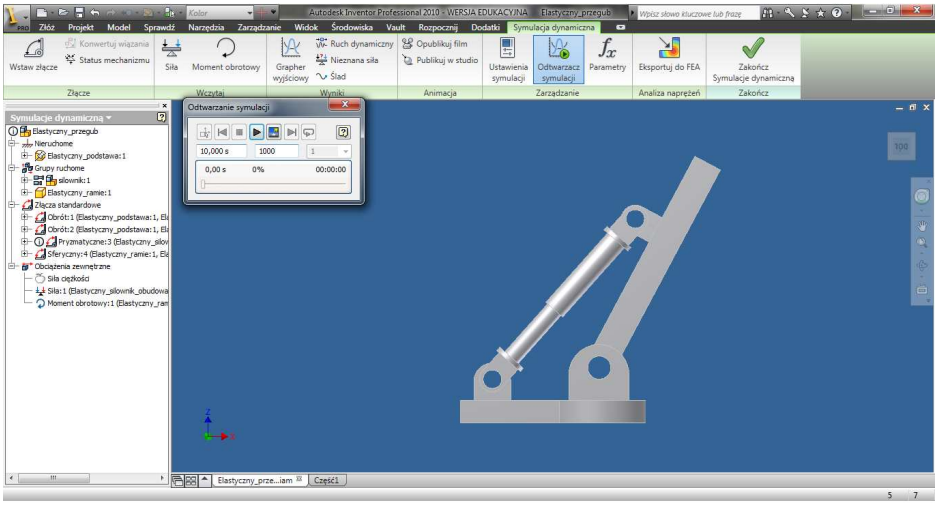
W celu dalszej edycji części należy tworzyć nowe szkice dwuwymiarowe na jednej z płaszczyzn projektowanego elementu (poprzez wybór odpowiedniej ikonki w menu, a następnie płaszczyzny). Po narysowaniu na wybranej płaszczyźnie zadanego kształtu ponownie można wykonać wyciągnięcie proste bądź inne operacje – np. wycięcie otworu. I tak aby wykonać otwór w szkicu dwuwymiarowym należy narysować okrąg, a następnie po zakończeniu szkicu wybrać z menu opcję *Otwory*, ustawić parametry i wybrać środek okręgu jako środek otworu.

8.4.2. Plik zespołu

Plik zespołu (z rozszerzeniem *.iam) składa się z wielu elementów (części), które mogą występować wielokrotnie w ramach jednego zespołu, a pomiędzy sobą są połączone tzw. wiązaniami. Rodzaje wiązań (ograniczeń) dzielimy na:

- nieruchome
  - wiązanie zestawiające (ustawia części w jednej płaszczyźnie),
  - wiązanie kątowe (stały kąt pomiędzy płaszczyznami),
  - wiązanie styczne (styczność okręgu do płaszczyzny),
  - wiązanie wstawiające (zestawia dwie okrągłe części ustawiając środki okręgów współliniowo).
- ruchome
  - obrotowe (obrót jednego okręgu powoduje obrót drugiego),
  - obrotowo-przesuwny (obrót okręgu powoduje przesunięcie stycznego elementu).

Przy dodawaniu pierwszej części do zespołu należy zachować szczególną ostrożność – będzie to część nieruchoma, na stałe przytwierdzona do otoczenia. Kolejno dodawane części będą już mogły się poruszać. Elementami zespołu mogą być także inne zespoły – między nimi można także definiować wiązania (tak jak pomiędzy częściami). Ustawiając ograniczenia należy wybrać odpowiednią ikonkę z menu, a także płaszczyzny elementów, które chcemy ze sobą powiązać.



Rysunek 8.2. Symulacja dynamiczna

8.5. Symulacja dynamiczna

Symulacja dynamiczna środowiska Autodesk Inventor może być wykorzystana jako pierwsza symulacyjna weryfikacja poprawności zaprojektowanego modelu. Uruchomienie tego podsystemu możliwe jest w dowolnym momencie pracy z modelem, co czyni go przydatnym przez całą fazę projektowania. W zakładce *Środowiska* znajduje się ikonka *Symulacja dynamiczna*. Po jej uruchomieniu pojawia się okno z *Odtwarzaczem symulacji* (tak jak na rysunku 8.5), w którym można ustawiać parametry symulacji – czas, liczbę wszystkich klatek i liczbę klatek pomijanych przy animacji. Przy pracy z symulacją warto wyróżnić dwa tryby:

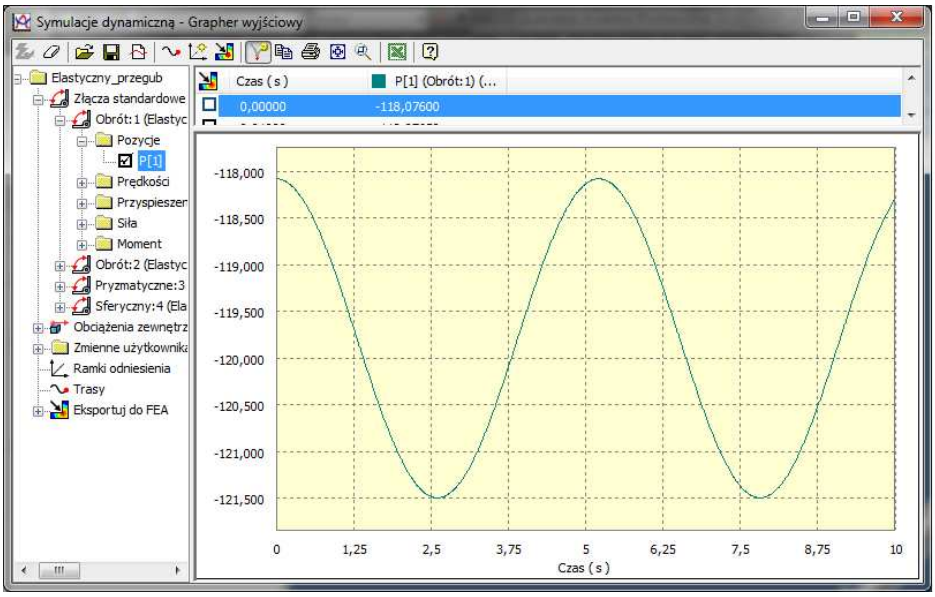
— tryb konstrukcji, w którym możemy edytować nasz obiekt dodając do niego siły i momenty,
— tryb symulacji, w którym jakakolwiek edycja obiektu jest niedostępna.

Zmiana trybu symulacji na tryb konstrukcji odbywa się poprzez pierwszą z lewej ikonkę w odtwarzaczu symulacji, natomiast w drugą stronę przechodzimy wciskając po prostu przycisk *Play*.

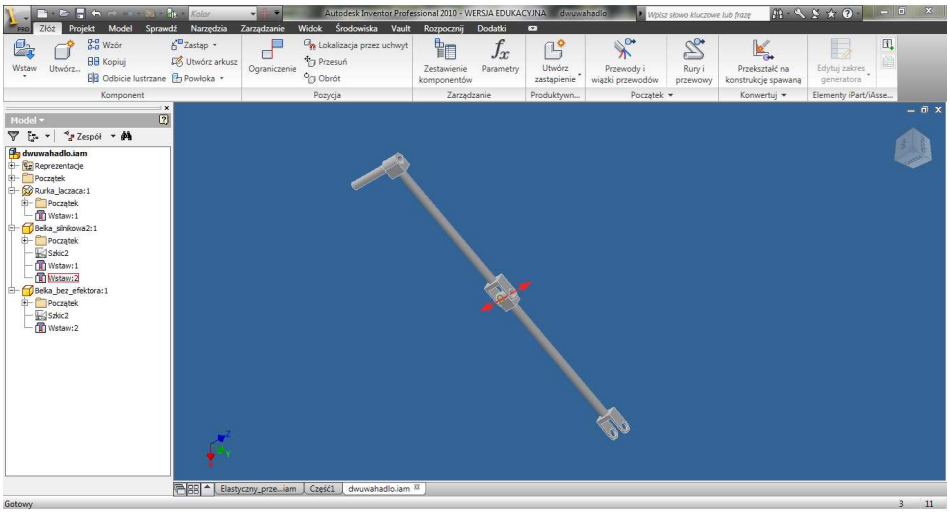
Tryb konstrukcji daje nam możliwość przykładania do naszego modelu sił i momentów zewnętrznych (ikonki na górze okna), a także definiowanie siły grawitacji (dostępna z przeglądarki modelu). Każda z sił musi być przyłożona w konkretnym punkcie obiektu, a także niezbędne jest zdefiniowanie jej kierunku i stałej wartości. Podobnie sytuacja wygląda z momentem, tylko że w tym przypadku wybieramy oś obrotu. Warto jest także pamiętać o poprawnym definiowaniu jednostek, które w przypadku momentu domyślnie wprowadzane są w Niutonomilimetrach.

8.5.1. Grapher wyjściowy

Po wykonaniu symulacji mamy możliwość analizy różnorakich wartości fizycznych (położeń, prędkości, przyspieszeń, sił, momentów itp.) za pomocą `Graphera wyjściowego`, którego ikonka znajduje się obok ikonki *Momentu obrotowego*. Dostarcza on prostego interfejsu, w którym dla każdego elementu modelu można włączyć wykres różnych wartości fizycznych, a następnie porównać je, wydrukować bądź zapisać do pliku. Przykładowe okno graphera zawiera rysunek 8.3.



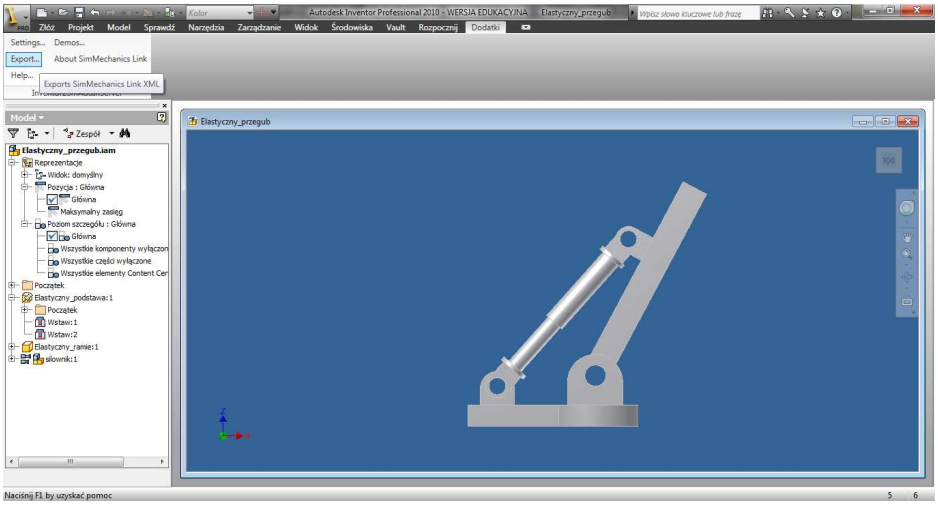
Rysunek 8.3. Grapher wyjściowy



Rysunek 8.4. Przykładowy model dwuwahadła

8.6. Przykładowy model dwuwahadła

Autodesk Inventor bardzo dobrze sprawdza się również w projektach z dziedziny robotyki. Za pomocą wiązań pomiędzy elementami możliwe jest modelowanie przegubów manipulatora, zarówno tych obrotowych jak i przesuwnych. Widok gotowego projektu dwuwahadła pokazano na rysunku 8.4. Składa się ono z nieruchomej rurki, użytej jako podstawy oraz dwóch ramion połączonych ze sobą przegubami obrotowymi. Przeguby obrotowe zostały zrealizowane poprzez łącza wstawiające – jedno z nich łączy pierwsze ramię z podstawą, natomiast drugie łączy oba ramiona ze sobą, co jest uwidaczniane w przeglądarce modelu po lewej stronie.



Rysunek 8.5. Eksport modelu do formatu XML

8.7. Eksport i import modelu

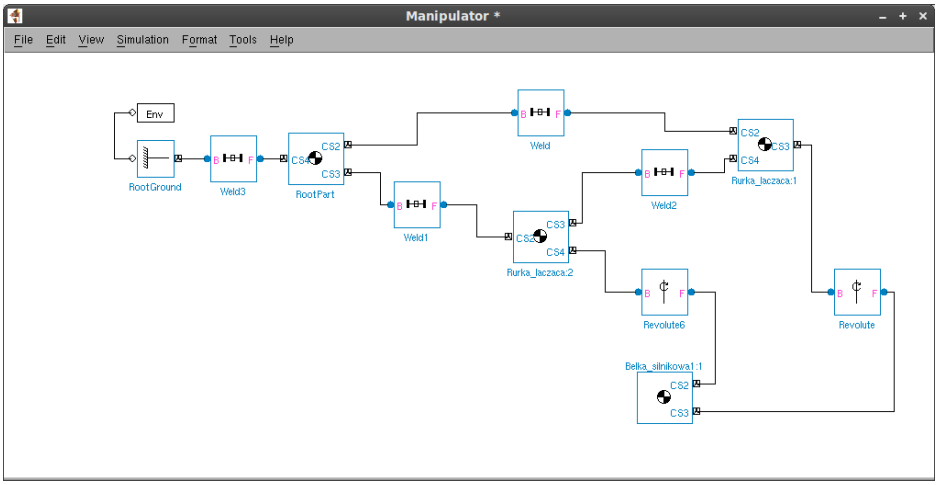
Autodesk Inventor umożliwia eksport utworzonego modelu w formacie XML. Niezbędne jest w tym przypadku wykorzystanie środowiska Matlab, które udostępnia pakiet `SimMechanics Link`. Instalacja odbywa się przez odpowiedni skrypt w Matlabie. Wszystkie informacje, jak i sam pakiet, można znaleźć na stronie [2].

Poza samymi elementami konstrukcji, ich wagą, rodzajem i rozmiarami, do formatu XML zapisywane są także połączenia (wiązania) pomiędzy nimi. Cała operacja odbywa się poprzez wejście w menu dodatki i wybranie polecenia *eksport*, a następnie zapisanie modelu do pliku XML (tak jak zostało to zaprezentowane na rysunku 8.5). Poniższy wydruk zawiera fragment pliku XML wygenerowanego dla przykładowego dwuwahadła:

```
(...)
      <Body>
        <name>"Belka_silnikowa:1"</name>
        <nodeID>"Belka_silnikowa:1"</nodeID>
        <status>""</status>
        <mass>0.0314159</mass>
        <massUnits>"kg"</massUnits>
(...)
```

Aby zaimportować plik w formacie XML do Simulinka, należy wykorzystać polecenie `mech_import('nazwa_pliku.xml')`, które zamieni kod XMLa na bloki Simulinka. Przykład tak zaimportowanego modelu został przedstawiony na rysunku 8.6. Znaczenie poszczególnych bloków modelu:

— `Env` – blok reprezentujący środowisko zewnętrzne,
— `RootGround` – blok reprezentujący nieruchomy punkt odniesienia,
— `Weld` – połączenie spawane, czyli nieruchome połączenie,
— `RootPark, RurkaLaczaca, BelkaOstatnia` – element modelu,
— `Revolute` – blok reprezentujący połączenie obrotowe (wiązanie wstawiające).



Rysunek 8.6. Model z Inventora przeniesiony do Simulinka

8.8. Podsumowanie

Autodesk Inventor jest rozbudowanym i złożonym środowiskiem do modelowania obiektów fizycznych. Do jego największych zalet z pewnością zalicza się prosty interfejs użytkownika, mnogość zastosowań, a także rozbudowane możliwości współpracy z zewnętrznymi programami. Wrażenie robi także liczba podsystemów włączonych w skład Autodesk Inventora – rozpoczynając od *Symulacji dynamicznej*, przez *Analizę naprężeń*, aż po *Autodesk Vault*, który jest odpowiednikiem systemu kontroli wersji dla projektów związanych z Autodeskiem.

Największą wadą systemu jest niewątpliwie brak rozbudowanej analizy dynamicznej, która z punktu widzenia robotyka ma ogromne znaczenie. Ponadto wciąż rozbudowy wymaga moduł analizy kolizji, który jest bardzo prosty i nie posiada możliwości bieżącego śledzenia kolizji.

Bibliografia

[1] Autodesk. Autodesk inventor. http://www.autodesk.pl/.
[2] Mathworks. SimMechanics Link. http://www.mathworks.com/help/toolbox/physmod/smlink/ug/brpyzn7-1.html.

9. Open Dynamics Engine

Tomasz Jordanek

Biblioteka ODE jest przykładem środowiska do symulacji i wizualizacji modeli fizycznych (brył sztywnych), opartym na licencji LGPL (free software). ODE pozwala na symulacje w czasie rzeczywistym m.in. dzięki temu została już z powodzeniem wykorzystana w grach komputerowych (co można uznać za duży sukces i dowód na realizm odwzorowania symulowanych obiektów) [1].

9.1. Instalacja

Instalacja biblioteki ODE jest przebiega w 90% zgodnie z procesem przedstawionym w dokumentacji (w przypadku instalacji na systemie LINUX). Z jednym wyjątkiem. W przypadku funkcji odpowiedzialnych za samą grafikę i wizualizację symulacji odpowiednie biblioteki nie zostaną automatycznie dołączone do katalogów `/usr/local/include` oraz `/usr/local/lib`. O czym w dokumentacji nie jest użytkownik poinformowany. W tym przypadku należy samemu przenieść odpowiednie pliki (z katalogu `drawstuff`) do wyżej wymienionych lokalizacji, i przy podczas kompilacji ręcznie je linkować (np. definiując odpowiednią zmienną w pliku `Makefile`). Zwykle wystarczy to zrobić tylko dla 4 plików:

— `drawstuff.o`,
— `drawstuff.h`,
— `version.h`,
— `x11.o`.

Dostarczone wraz z biblioteką przykłady demonstracyjne posiadają odpowiednio skonfigurowany plik `Makefile`, więc nie będzie problemu z ich kompilacją. Jednak jeśli użytkownik chciałby skompilować demo w innej lokalizacji niż domyślna, bądź też własny program, to działanie zakończy się niepowodzeniem bez odpowiedniego dołączenia tychże plików (`drawstuff.h` w pliku nagłówkowym oraz reszty plików podczas kompilacji).

9.2. Tworzenie programu

W celu pokazania podstawowych elementow w bibliotece ODE zaimplementowano prosty program `Hello World`. Program typu `Hello World` napisany z wykorzystaniem biblioteki ODE (oraz Drawstuff - wizualizacja) ma za zadanie umożliwić zapoznanie się z poszczególnymi etapami procesu tworzenia symulacji, kolizji oraz wizualizacji. Proces tworzenia symulacji możemy w tym wypadku podzielić na 3 główne etapy (warstwy):

— warstwa obiektów,
— warstwa geometrii (kolizji),
— warstwa wizualizacji.

Z pewnym przybliżeniem można powiedzieć, że każda wyższa warstwa do prawidłowej pracy potrzebuje wcześniejszej. Przedstawiony program pokazuje zachowanie się sfery, w przypadku nadania jej prędkości kątowej bądź też przypadku gdy działa na nią określone siła. Na tym przykładzie zostaną przedstawione poszczególne 3 warstwy tworzenia całej aplikacji, oraz specyfika tworzenia sceny i obiektów z wykorzystaniem tej biblioteki (ODE).

W 90% aplikacji będą występowały funkcje takie jak:

1. main,
2. start,
3. simLoop,
4. drawEverything,
5. nearCallback,
6. command.

W nich to zostają zwykle zrealizowane wcześniej wymienione warstwy (czasem na siebie nachodząc). Najlepszym sposobem przedstawienia realizacji symulacji w środowisku ODE będzie omówienie poszczególnych funkcji.

9.2.1. Plik `makefile`

Najbardziej istotnym elementem pliku `Makefile` jest lista obiektów i plików nagłówkowych związanych z modułem `Drawstuff` (odpowiedzialnym za wizualizację symulacji). W związku z faktem iż jego elementy nie są automatycznie umieszczane w odpowiednim kartotekach systemu (nie są tworzone także pliki typu .a) wymagane jest każdorazowe wskazywanie ich podczas kompilacji.

9.2.2. Funkcja `main`

W tym miejscu zostaje zainicjalizowana oraz scharakteryzowana scena. W funkcji main zostają przekazane także wskaźniki dla funkcji z modułu Drawstuff odpowiedzialnych za wizualizację naszej symulacji. Inicjalizacja świata przedstawiona została na wydruku 9.1.

Wydruk 9.1. Inicjalizacja świata

```
dInitODE();
swiat = dWorldCreate();
space = dHashSpaceCreate (0);
grupazderzen = dJointGroupCreate (0);
dWorldSetGravity (swiat,0,0,-0.05);
dCreatePlane (space,0,0,1,0);
```

W tej części programu zostaje zainicjalizowany `solver` (jest on odpowiedzialny za całe obliczanie fizyki, tzn. zachowanie się ciał na które działa siła, posiadających prędkość), cała scena (zmienna `swiat`), podłoże (`space`), oraz obiekt potrzebny podczas obsługi ew. zderzeń obiektów (`grupazderzen` nasza sfera będzie zderzała się z podłożem). Przekazanie odpowiednich wskaźników do nazw funkcji dla modułu `Drawstuff` odbywa się także w funkcji `main`: wydruk 9.2.

Wydruk 9.2. Przekazanie wskaźników

```
dsFunctions fn;
fn.version = DS_VERSION;
fn.start = &start;
```

```
fn.step = &Symuluj;
fn.command = &command;
fn.stop = 0;
fn.path_to_textures = DRAWSTUFF_TEXTURE_PATH;
```

Zostają tutaj przekazane wskaźniki do funkcji inicjalizującej (`start`), funkcji pętli symulacji simLoop (tutaj nazwanej `Symuluj`), funkcję obsługi przerwań od klawiatury (`command`) oraz ścieżka do wykorzystywanych tekstur. W funkcji main zwykle inicjalizuje się także większość (jeśli nie wszystkie) obiekty występujące w symulacji. W tym programie (wydruk 9.3) stworzony zostanie tylko jeden obiekt, sfera (zostaną jeszcze stworzone obiekty takie jak `world` oraz `space` jednak są one zawsze tworzone jeśli ma mieć sens tworzenie jakichkolwiek innych).

Wydruk 9.3. Inicjacja obiektu oraz geometrii

```
cialo[0] = dBodyCreate (swiat);
sfera[0] = dCreateSphere (space,0.5);
dMassSetSphere (&m,1,0.5);
dBodySetMass (cialo[0],&m);
dGeomSetBody (sfera[0],cialo[0]);
dBodySetPosition(cialo[0],0,0,2);
```

W tym miejscu zostaje zainicjalizowany obiekt (`cialo[0]`) oraz geometria (`sfera[0]`). Jest to podejście dosyć nietypowe. Obiekt nie musi mieć jakiejkolwiek geometrii. Jeśli jednak wymaga jest jakakolwiek możliwość interakcji danego obiektu z otoczeniem należy danemu obiektowi przypisać jakąś geometrię (w tym wypadku sferę). Jeśli np. użytkownik chciałby badań jak zachowuje się obiekt z prędkością początkową wyrzucony w ziemskim polu grawitacyjnym to nie ma potrzeby deklaracji geometrii tak długo jak długo użytkownika zainteresowany jest ruchem aż to zderzenia (które tu nie wystąpi) z ziemią. W przypadku obiektu któremu nie przypisano żadnej geometrii nie będzie możliwa obsługa zderzeń. Najistotniejszym elementem jest tu funkcja

```
dGeomSetBody (sfera[0],cialo[0]);
```

dzięki której obiekt zostanie połączony z geometrią. Dzięki temu wszelkie zmiany (dotyczy np. położenia) zastosowane względem jednego, zostaną automatycznie zastosowane względem drugiego. Po zainicjalizowaniu i opisaniu poszczególnych obiektów zostaje wywołana pętla symulacji

```
dsSimulationLoop (argc,argv,480,320,&fn);
```

gdzie zostają przekazane parametry wywołania programu, wymiary okna (w przypadku wizualizacji) i adres zmiennej przechowującej wskaźniki do pozostałych funkcji symulacji.

9.2.3. Funkcja start

Jest to funkcja wywoływana zwykle tylko raz i odpowiedzialna za alokację pamięci, ustawianie początkowe kamery (jeśli wykonywana jest wizualizacja). W tym programie zawiera ona tylko dwie wyżej wymienione instrukcje

```
dAllocateODEDataForThread(dAllocateMaskAll);
dsSetViewpoint(PosCam,AngCamUs);
```

Parametrami drugiej funkcji są pozycja i orientacja kamery.

9.2.4. Funkcja simLoop

Funkcja simLoop stanowi główną pętlę symulacji. To w niej znajduje się odpowiedzialna za „upływ czasu" w świecie (scenie) funkcja

```
dWorldStep(swiat,0.05);
```

Dzięki niej zostają ponownie obliczone nowe pozycje, orientacje obiektów. Twórcy zalecają by krok symulacji (w tym wypadku wartość 0.05) był stały. Związane jest to z ew. błędami które będą się pojawiały jeśli krok symulacji będzie zmienny (tzn. drugi argument tej funkcji nie musi mieć argumentu const, mimo wszystko powinien być jednak stały). Dla małych symulacji (jak ta, jest tu tylko jeden obiekt) możliwe jest (nie widać zbytnich zaburzeń) zmienianie tego parametru w trakcie działania programu (można np. zwalniać/przyśpieszać symulację).

W przypadku wielu obiektów krok symulacji powinien być jednak mały, związane jest to z obliczaniem pozycji i sił w przypadku ew. zderzeń obiektów. W przypadku dużych prędkości i dużego kroku symulacji obiekty mogą wpierw „zajść na siebie", po czym dopiero zostaną wprowadzone siły odpowiedzialne mające przeciwdziałać nachodzeniu się obiektów na siebie (tzn. siły te pojawiają się gdy dwa obiekty są wystarczająco blisko siebie).

Drugim ważnym elementem jest funkcja `nearCallback`, zostanie ona dokładnie opisana w innym miejscu, lecz jest odpowiedzialna za obsługę wszelkiego rodzaju kolizji między obiektami. Powinna być wywoływana przed funkcją `dWorldStep()`.

Dla poprawnego działania na pod koniec tej pętli powinno nastąpić wywołanie funkcji `drawEverything`, dzięki czemu wybrane obiekty zostaną przerysowane w swych nowych położeniach. W programie hello.cpp funkcja `drawEverything()` nie została wyszczególniona, i jej zawartość stanowi po prostu część funkcji `simLoop` (w przypadku dużych modeli tzn. ze skomplikowanymi obiektami takimi jak TriMesh, bądź też dużą liczbą obiektów wygodniejsze jest napisanie osobnej funkcji `drawEverything` odpowiedzialnej za ponowne przerysowanie obiektów).

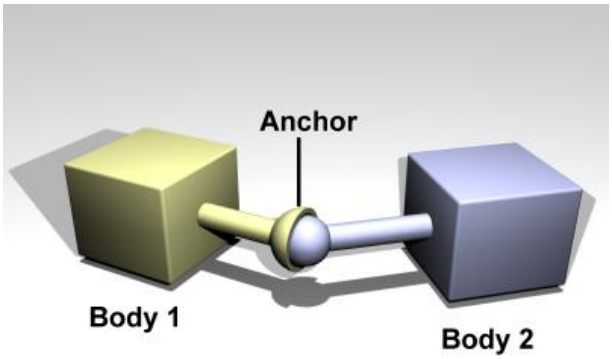
9.2.5. Funkcja drawEverything

Zwykle w tej funkcji zawiera się cała obsługa rysowania sceny, i obiektów. Przydatna jest szczególnie w przypadku rysowania obiektów takich jak TriMesh (siatka trójkątna), bądź też gdy obiektów jest dużo. W przypadku tego dema nie została zaimplementowana. Całość obsługi rysowania znajduje się w funkcji `simLoop` (gdzie ew. funkcja `drawEverything` powinna zostać wywołana). W tym przypadku rysowania dla naszej sfery zostaje wywołana w postaci pokazanej na wydruku 9.4.

Wydruk 9.4. Wywołanie funkcji drawEverything

```
1 dsSetColor (0,1,1);
  dsSetTexture (DS_WOOD);
3 dReal sides[3] = {0.5,0.5,0.5};
  dsDrawBox (dBodyGetPosition(cialo[0]),
5 dBodyGetRotation(cialo[0]),sides);
```

Zostają tu zadeklarowany kolor i tekstura użyta do narysowania sześcianu. Pobierana jest aktualna pozycja i orientacja sfery a następnie rysowany jest sześcian. Dlaczego sześcian? Funkcje odpowiedzialne za obliczenia kolizji, przyśpieszeń oraz położenia są zupełnie niezależne od modułu Drawstuff odpowiedzialnego za wizualizację. Tak więc



Rysunek 9.1. Typ przegubu: ball and socket

`solver` od ODE „widzi" sferę natomiast moduł Drawstuff może swobodnie rysować sześcian, cylinder, itp. Gdyby użytkownik chciał jednak narysować kulę należałoby zmienić ostatnie wywołanie funkcji na pokazane na wydruku 9.5.

Wydruk 9.5. Zmiana wywołania funkcji drawEverything

```
1 dsDrawSphere(dBodyGetPosition(cialo[0]),
  dBodyGetRotation(cialo[0]),RADIUS);
```

9.2.6. Funkcja nearCallback

Najprawdopodobniej najbardziej złożona i skomplikowana funkcja z dotychczas wymienonych. Jej zadaniem jest obsługa wszelkiego rodzaju zderzeń (oraz sytuacji bliskich zderzeniu) pomiędzy dwoma geometriami (nie obiektami, obiekt może być bez geometrii). Dlatego też aby obiekt był dynamiczny (mógł zderzać się z innymi) musi mieć przypisaną swoją geometrię, która jest przekazywana do tej właśnie funkcji. Wszelka obsługa ew. zderzeń musi zostać zaimplementowana przez użytkownika (także tych najbardziej podstawowych jak np. kontakt z podłożem). W tym celu wykorzystywany jest obiekt typu dJoinGroupID w którym przechowywane są wszystkie stawy (joints) pojawiające się kiedy dwa obiekty(geometrie) potencjalnie ze sobą kolidują (dokładny opis idei obiektów typu „joint" zostanie podany w innym miejscu).

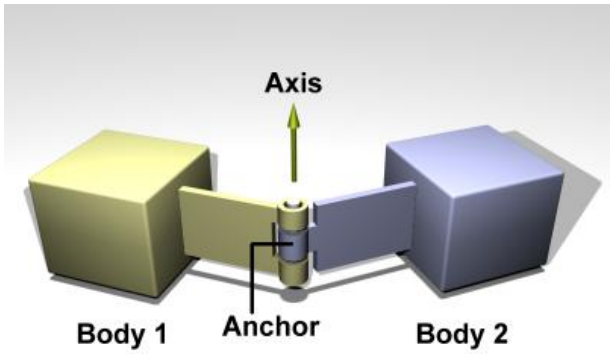
9.2.7. Funkcja command

W tej funkcji realizowana jest obsługa zdarzeń z klawiatury. Zwykle opiera się na funkcji `switch()` z odpowiednimi parametrami. Niektóre sygnały są obsługiwane w sposób domyślny przez moduł Drawstuff (takie jak np. kombinacje klawiszy Ctrl+X czyli zakończenie programu).

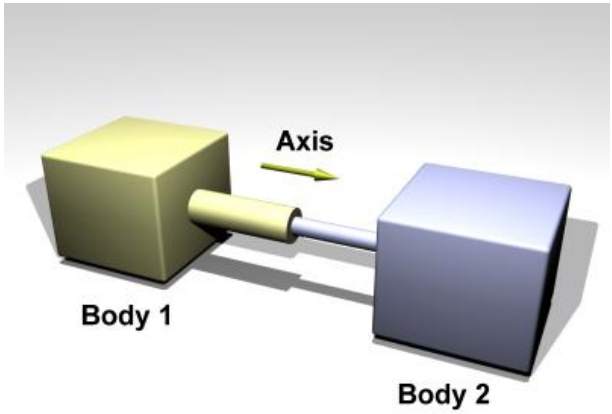
9.2.8. Przeguby

Poszczególne przeguby zostały umieszczone i opisane na rysunkach 9.1–9.6. Jak można zauważyć ODE posiada dużą liczbę różnych przegubów (w większości będących złożeniem paru podstawowych).

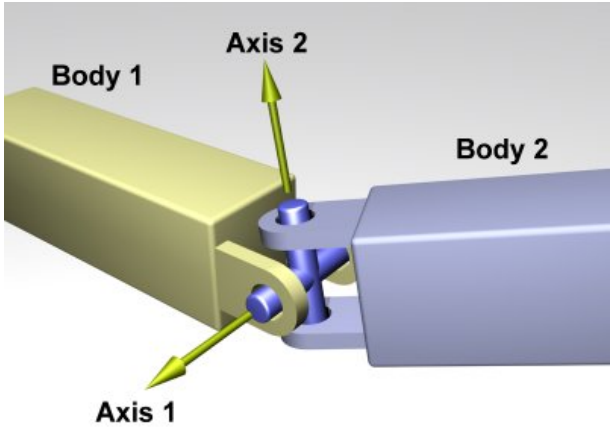
Połączenie dwóch ciał odpowiednim przegubem odbywa się w sposób bardzo prosty i intuicyjny, zaprezentowany na wydruku 9.6.



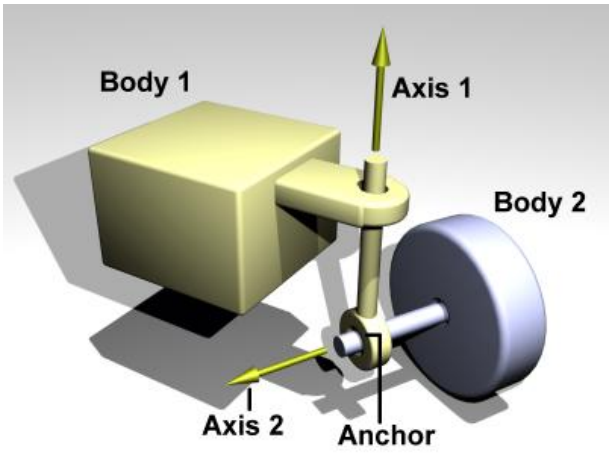
Rysunek 9.2. Typ przegubu: hinge



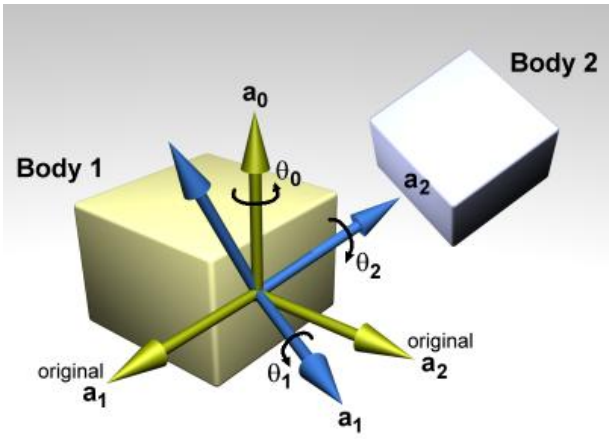
Rysunek 9.3. Typ przegubu: slider



Rysunek 9.4. Typ przegubu: universal



Rysunek 9.5. Typ przegubu: hinge 2



Rysunek 9.6. angular motor

Wydruk 9.6. Połączenie dwóch ciał

```
static dJointID hinge;
hinge = dJointCreateHinge (world,0);
dJointAttach (hinge,body[0],body[1]);
dJointSetHingeAnchor (hinge,0,0,1);
dJointSetHingeAxis (hinge,1,-1,1.56);
```

W pierwszej linii następuje zadeklarowanie zmiennej będącej przegubem a następnie przypisanie go do sceny (świata). `dJointHinge` tworzy przegub typu wahadło. Funkcja `dJointAttach` łączy dwa ciała za pomocą podanego jako argument przegubu. Funkcje `dJointSetHingeAnchor` oraz `dJointSetHingeAxis` są już typowe dla tego określonego typu przegubu i odpowiadają za wprowadzenie odpowiednich wartości nastaw.

9.3. Wizualizacja

W przypadku ODE użytkownik ma do wyboru dwa główne środowiska. Są nimi Drawstuff oraz OpenGL. Niewątpliwą zaletą modułu Drawstuff jest jego prostota. Rysowanie świata oraz poszczególnych obiektów nie sprawia poważnych trudności. Problemem zwią-

zanym z ową prostotą jest nie tak szeroki zakres możliwości tego modułu. Porównanie możliwości modułu Drawstuff z osobną biblioteką do wizualizacji (OpenGL) wypada dla tego pierwszego bardzo niekorzystnie. Jednak jak zostało to wspomniane Drawstuff pozwala na bardzo szybkie uzyskanie podglądu stworzonego świata, oraz obserwacji zachowań poszczególnych obiektów, do tego nie są potrzebne specjalistyczne efekty graficzne jakie może dostarczyć OpenGL. Wizualizacja jest także bardzo istotnym elementem procesu debagowania programu. W przypadku świata z wieloma obiektami użytkownik nie jest w stanie na poziomie debugera tekstowego (wartości położeń, orientacji itp.) stwierdzić czy symulacja przebiega w sposób właściwy.

9.4. Obsługa zderzeń

Wprzypadku ODE obsługa zderzeń nie jest rzeczą trywialną. Użytkownik jest zmuszony do implementacji obsługi zderzeń pomiędzy każdą parą obiektów (a właściwie geometrii). Jest to szczególnie uciążliwe w przypadku występowania bardzo dużej ilości elementów na scenie.

9.5. API

O ile zaznajomienie się z API występującym w ODE jest mocno ułatwione poprzez fakt implementacji w języku C, o tyle niektóre jego elementy (np. brak struktur od razu grupujących w jednym miejscu obiekt i geometrię) utrudniają pracę. Modułowa budowa programów pozwala na bardzo szybki start (w tworzeniu aplikacji), jednak zaczyna przeszkadzać wraz z rozwojem i wzrostem objętości kodu. Użytkownik od samego początku panuje nad poszczególnymi etapami tworzenia sceny, obiektów, obsługi zdarzeń. Jednak zdecydowanie mniej wygodna jest praca nad dużym objętościowo projektem z wykorzystaniem ODE.

9.6. Podsumowanie

Biblioteka ODE pozwala na symulację dynamiki brył sztywnych oraz ich zderzeń. Całość funkcjonalności ODE można tutaj skrócić do słów: „symulacja dynamiki oraz zderzeń brył sztywnych". Bardzo dobrym obrazem różnic w funkcjonalności jest porównanie różnego rodzaju efektów, sytuacji, eksperymentów przedstawionych w programach demonstracyjnych dołączonych do biblioteki.

Biblioteka dostarcza dużą ilość przykładowych programów demonstracyjnych, wraz z ich kodem źródłowym. Stanowi to bardzo dobrą bazę do zapoznania się z API oraz rozwiązaniami najpopularniejszych problemów powstających przy tworzeniu programu. Dokumentacja w obydwu przypadkach jest też wykonana przy użyciu takiego samego narzędzia - Doxygen. W przypadku ODE wyżej wymieniony problem nie występuje, natomiast sama dokumentacja (2-3 zdaniowe opisy funkcji) prezentuje dość wysoki poziom.

Poniżej przykładowa lista wymagań i cech projektu który miałby być wykonywany z użyciem ODE.

— krótki czas życia projektu,
— proste narzędzie do debagowania,
— nieskomplikowana jakość wizualizacji,
— praca w środowisku C.

Bibliografia

[1] R. Smith. Open Dynamics Engine User Guide. http://www.scribd.com/doc/38543171/Ode-Latest-Userguide.

10. JSBSim

Przemysław Synak

Biblioteka `JSBSim` udostępniana na zasadach licencji LGPL jest zestawem funkcji napisanych prawie w całości w języku `C++` (jednakże drobna ich część została zaimplementowana w `C`), w celu symulacji zachowania różnych obiektów latających (takich jak samoloty, helikoptery, quadrotory, UAV, etc.) pod wpływem różnych zjawisk aerodynamicznych [2]. Sposób jej implementacji pozwala na dołączanie jej jako zewnętrznej bibliotek do projektu, lub wykorzystanie jako samodzielnego środowiska symulacyjnego[1]. `JSBSim` nie jest graficznym symulatorem lotu, tzn. nie umożliwia obserwacji zachowania się obiektu w trakcie symulacji, w tym celu należy wykorzystać inne biblioteki czy programy (np. zob. projekt FlightGear[2]). Biblioteka jest przenośna dzięki czemu jest z dużym powodzeniem wykorzystywana zarówno przez osoby pracujące w środowisku *MS Windows* jak i *Mac OS* oraz *Linux*. Celem niniejszej pracy jest omówienie budowy, możliwości oraz wykorzystania biblioteki języka `C++ JSBSim`.

10.0.1. Instalacja

Instalację należy przeprowadzić zgodnie z instrukcjami, warto jedynie zwrócić uwagę, że samego jej opisu brak jest w dokumentacji[3].

10.1. Przygotowanie projekt z JSBSim

10.1.1. Ogólna budowa projektu w JSBSim

W przypadku wykorzystania `JSBSim` jako samodzielnego środowiska symulacyjnego użytkownik musi dostarczyć informacje dotyczące zarówno symulowanego obiektu jak i otaczającego go środowiska (podłoże, zjawiska aeorodynamiczne etc.). Wszystkie te informacje muszą być zapisane w plikach typu **xml**. Potrzebne jest więc utworzenie 3 plików w których będą zapisane dane projektu, a mianowicie[4]:

aircraft.xml – plik ten zawiera wszystkie informacje dotyczące modelu obiektu, tzn. pełny opis symulowanego obiektu (np. helikoptera),

initialpar.xml – w pliku tym zapisane są informacje opisujące warunki początkowe symulacji takie jak: położenie początkowe obiektu, początkowe prędkości etc.,

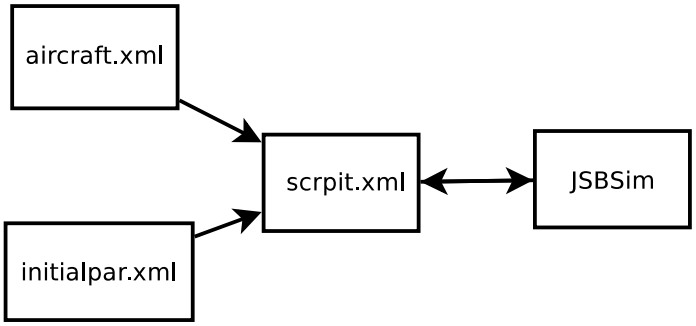
script.xml plik ten zawiera wszelkie informacje dotyczące procesu symulacji, takie jak: początkowy czas symulacji, czas końca symulacji, krok symulacji. Dodatkowo informa-

[1] Dlatego też w dalszej części będzie ona nazywana środowiskiem symulacyjnym.

[2] Jest to projekt typu *open source* który wykorzystuje bibliotekę `JSBSim`, jednocześnie tworząc obraz 3D symulowanego obiektu (http://www.flightgear.org/).

[3] Przeprowadzona została instalacja dla systemu *Windows 7* oraz *Ubuntu 10.10*, sprawdzenie instalacji na *Mac OS* nie było niestety możliwe.

[4] Nazwy plików oczywiście mogą być inne nż podane tu przykładowo dobrane tak by odpowiadały swej zawartości.



Rysunek 10.1. Zależności plików **xml** w środowisku JSBSim

cje o wszystkich zdarzeniach warunkowych (które zostaną dokładnie opisane później) powinny być zawarte w tym pliku.

Plik **script.xml** zawiera dodatkowo informacje o tym, które pliki typu opisujące obiekt i otoczenie będzie wykorzystywał. Dzięki takiej strukturze symulacja takiego samego scenariusza dla innej maszyny wymaga jedynie zadeklarowania innego pliku **xml** który tą maszynę opisuje. Zależności pomiędzy poszczególnymi plikami zostały przedstawione na rysunku 10.1.

10.1.2. Przygotowanie plików xml

Składnia plików **xml** powoduje, że ich bezpośrednie przygotowanie nie jest łatwe – wygodniej jest korzystać w tym celu z dedykowanych narzędzi. Niestety `JSBSim` nie dostarcza żadnych narzędzi wspierających tworzenie plików **xml** zgodnych z jego składnią[5].

10.1.3. Wymagane dane wstępne – pomocne programy

W przypadku `JSBSim` od użytkownika nie jest wymagana wiedza na temat dynamiki symulowanego obiektu. Potrzebna jest jednak pełna identyfikacja mas i bezwładności poszczególnych elementów obiektu. `JSBSim` zmusza do identyfikacji pełnych macierzy bezwładności, których wyznaczenie nie jest zadaniem trywialnym. W celu identyfikacji wyżej wymienionych parametrów można wykorzystać program *Autodesk Inventor* [1], który to służy do projektowania elementów mechaniki. Dzięki jego rozległej bazie danych materiałów (aluminium, stal etc.) oraz możliwości eksportu projektu (jako plik **xml**) możliwe jest szybkie zidentyfikowanie wspomnianych parametrów obiektu potrzebnych do zainicjowania symulacji.

Drugim zestawem potrzebnych danych jest wiedza na temat teoretyczna na temat modelowanych zjawisk aerodynamicznych. `JSBSim` w przypadku zjawisk takich jak wiatr, siła nośna etc. wykorzystywana jest postać tzn. „współczynników”[6]. Niestety składnia plików **xml** eksportowanych przez `Autodesk Inventor` nie jest zgodna ze składnią `JSBSim`– wymagane jest więc żmudne przekopiowywanie poszczególnych wartości z pliku do pliku.

[5] Istnieją wprawdzie internetowe aplety do tworzenia dość prostych podstawowych schematów symulacji, takie jak (aeromatic [4]) , lecz nie są one kompletne i dysponują ograniczoną ilością opcji do wyboru (typu kliknij/zaznacz).

[6] Pojęcie współczynników w `JSBSim` definiowane jest jako wielkość, której iloczyn z inną wielkością daje w wyniku siłę, zob rozdz. 2.6 [2].

10.1.4. JSBSim a MATLAB

Dla MATLABa został opracowany interfejs GUI [3] będący niejako „wtyczką" biblioteki `JSBSim`. Niestety projekt ten nie jest oficjalną częścią projektu `JSBSim` (nie jest też aktywnie wspierany przez `MathWorks`), przez co nie jest rozwijany odpowiednio w stosunku do zmian w głównej gałęzi `JSBSim`, co skutkuje tym, że aktualna wersja zestawu funkcji do środowiska MATLAB nie jest zgodna z najnowszą wersją `JSBSim`.

Niestety podczas testów nie udało się uruchomić rzeczonego GUI na systemie *Linux*. W przypadku systemu *Windows 7* możliwe było uruchomienie GUI, jednak błędy w poszczególnych funkcjach uniemożliwiły inicjalizację samej symulacji. Na chwilę obecną używanie tego zestawu funkcji jest praktycznie niemożliwe (wykorzystana wersja systemu MATLAB to 2010a).

10.1.5. Symulacja

Sam proces symulacji, jego warunki (czas początkowy symulacji, czas końcowy, krok symulacji, zdarzenia warunkowe, dane wyjściowe etc.) powinny dla wygody użytkownika być umieszczone we wspomnianym wcześniej pliku skryptu, dzięki czemu uruchomienie symulacji następuje po wydaniu polecenia:

```
jsbsim --script=(ścieżka_do_pliku/)nazwa_skryptu.xml
```

Folderem domyślnym dla polecenia `jsbsim` jest folder instalacyjny środowiska. Tam też domyślnie zostanie umieszczony plik **csv** z wynikami symulacji. Istnieje też możliwość wysłania wyników pod określony adres `IP` (dokładny opis znajduje się w rozdziale 3.1.10 [2]).

10.1.6. Budowa projektu rakiety w JSBSim

Budowę projektu rakiety[7] w `JSBSim` należy rozpocząć od wspomnianego już opisu symulowanego obiektu, przyjmijmy że plik ów będzie się nazywał **aircraft.xml**. Dokument musi się rozpoczynać od opisu jego zawartości tzn.

```
<fdm_config name="aircraft" version="0.9" release="ALPHA">
...
</fdm_config>
```

Wszystkie dane zapisane w tym pliku muszą się znajdować wewnątrz tego głównego znacznika. Następnie można już rozpocząć opis właściwy obiektu.

```
<metrics>
        <wingarea unit="M2"> 1 </wingarea>
        <wingspan unit="M"> 1 </wingspan>
        <chord unit="M"> 1 </chord>
        <htailarea unit="M2"> 0 </htailarea>
        <htailarm unit="M"> 0 </htailarm>
        <vtailarea unit="M2"> 0 </vtailarea>
        <vtailarm unit="M"> 0 </vtailarm>
        <location name="AERORP" unit="IN">
            <x> 0 </x>
            <y> 0 </y>
```

[7] Przez rakietę rozumie się obiekt o regularnych kształtach, z jednym silnikiem umiejscowionym u dołu a zorientowanym początkowo wzdłuż ujemnej półosi X.

```
            <z> 0 </z>
        </location>
        <location name="EYEPOINT" unit="IN">
            <x> 0 </x>
            <y> 0 </y>
            <z> 0 </z>
        </location>
    </metrics>
```

Metryka ma za zadanie opisać wymiary obiektu czyli jego rozpiętość przestrzenną. Znacznik <*location name=AERORP*> określa centrum oporów aerodynamicznych natomiast *unit="IN"* oznacza cale. Zmienne M,$M2$ określają jednostki w jakich zostały podane poszczególne liczby w tym wypadku są to metry i metry kwadratowe. W `JSBSim` można swobodnie posługiwać się angielskim (imperialnym) systemem miar i wag (stopy oznaczane przez FT, oraz funty oznaczane jako LBS)[8]. Kolejnym krokiem powinno być zdefiniowanie rozmieszczenia masy w obiekcie.

```
<mass_balance>
        <ixx unit="KG*M2"> 15 </ixx>
        <iyy unit="KG*M2"> 15 </iyy>
        <izz unit="KG*M2"> 15 </izz>
        <ixy unit="KG*M2"> -0 </ixy>
        <ixz unit="KG*M2"> -0 </ixz>
        <iyz unit="KG*M2"> -0 </iyz>
        <emptywt unit="KG"> 100 </emptywt>
        <location name="CG" unit="IN">
            <x> 0 </x>
            <y> 0 </y>
            <z> 0 </z>
        </location>
    </mass_balance>
```

Jak widać wykorzystywane jest tu zwyczajny zapis macierzy inercji z momentami głównymi i dewiacyjnymi. Jednostkami wykorzystanymi jest masowy moment bezwładności czyli $kg * m^2$. W znaczniku <*emptywt*> podana jest jednostka oraz całkowita waga obiektu. Następnie zdefiniowany jest punkt środka masy ("CG") poprzez podanie jego współrzędnych (we współrzędnych związanych z obiektem). W przypadku większości symulacji należy także zdefiniować reakcje pomiędzy obiektem a podłożem.

```
<ground_reactions>
        <contact type="BOGEY" name="CONTACT">
            <location unit="IN">
                <x> -15 </x>
                <y> 10 </y>
                <z> 3 </z>
            </location>
            <static_friction> 0 </static_friction>
            <dynamic_friction> 0 </dynamic_friction>
            <rolling_friction> 0 </rolling_friction>
```

[8] Pełny wykaz jednostek wykorzystywanych przez `JSBSim` znajduje się na stronie 11 dokumentacji [2].

```
            <spring_coeff unit="LBS/FT"> 10000 </spring_coeff>
            <damping_coeff unit="LBS/FT/SEC"> 200000 </damping_coeff>
            <max_steer unit="DEG"> 0.0 </max_steer>
            <brake_group> NONE </brake_group>
            <retractable>0</retractable>
        </contact>
    </ground_reactions>
```

Użyty tutaj typ kontaktu **BOGEY** jest wykorzystywany do kontaktów „miękkich" tzn. takich jak pomiędzy kołami samolotu a pasem lotniska. Jego położenie określane jest zaraz później (w układzie związanym z obiektem). Następnie definiowane są poszczególne właściwości takie jak tarcie statyczne i dynamiczne, sterowalność koła (stałe, sterowalne, swobodne), grupa (dotyczy zagadnienia hamowania)[9]. W przypadku nawet najprostszego sterowania istotna jest wiedza na temat obiektu, której to mogą dostarczyć sensory. W przypadku opisywanego środowiska użytkownik ma bezpośredni dostęp do wszystkich wielkości charakteryzujących obiekt[10], lecz w celu zaimplementowania pewnych ograniczeń, zdarzeń związanych z sensorami i pomiarem wielkości (niedokładność, wpływ warunków atmosferycznych, opóźnienie) należy wykorzystać znacznik **<sensor>** np. tak jak poniżej.

```
<sensor name= predkosciomierz  >
<input> velocity/u-aero-fps </input>
<lag> 0.1 </lag>
<noise variation={ PERCENT }
distribution={ UNIFORM }> 0.03 </noise>
<bias> 0.1 </bias>
</sensor>
```

Podany sensor należy wpierw nazwać a następnie podać jego własności związane z cechami wpływającymi na pogarszanie jakości mierzonej wielkości (niedokładność, rozkład niedokładności, błąd stały, opóźnienie). Sensor idealny miałby więc postać

```
<sensor name= predkosciomierz  >
<input> velocity/u-aero-fps </input>
</sensor>
```

czyli podawałby dokładnie wielkość zadaną mu do pomiaru (w tym wypadku **velocityu-aero-fps** czyli prędkość liniową wzdłuż osi X podaną w kontekście kroków symulacji na sekundę). Następnie można już wykorzystać zdefiniowany sensor w kontrolerze PID. Niech celem sterowania będzie utrzymanie wspomnianej prędkości w okolicach 10. `JSBSim` posiada zaimplementowaną strukturę opisującą sterownik PID.

```
<pid name= sterownik_PID >
  <input>
   <sum>
    <property> predkosciomierz </property>
    <value> -10 </value>
   </sum>
```

[9] Pełny opis kontaktu koło-podłoże został przedstawiony w dokumentacji [2] strona 40.

[10] Ich długa lista znajduje się na stronie 141 dokumentacji [2]. Większość z nich nie jest opisana co znacznie utrudnia posługiwanie się tymi parametrami przez osoby polskojęzyczne i nie obeznane z terminologią lotniczą.

```
  </input>
  <kp> 100 </kp>
  <ki> 10 </ki>
  <kd> 50 </kd>
  [<trigger> property </trigger>]
  <output> silnik_ciag </output>
</pid>
```

Wpierw zdefiniowana jest konkretnej instancji sterownika (przez co możliwe jest odwoływanie się do niego), a następnie jego wielkości wejściowej (w naszym przypadku jest to sygnał z sensora czyli wcześniej zdefiniowanego prędkościomierza). Kolejnym krokiem jest zadanie parametrów kontrolera. Opcjonalna część związana ze znacznikiem **<trigger>** dotyczy wyzwalacza który to w przypadku niezerowej wielkości zeruje blok integratora. W **<output>** zapisana jest wielkość wyjściowa sterownika. Niestety sekcja silników nie jest opisana w dokumentacji, przez co koniecznym stało się wykorzystanie już zaimplementowanych przez twórców (w wielu przykładach) silników. W tym celu należy wykorzystać znacznik **<propulsion>**.

```
<propulsion>
        <engine file="XLR99">
            <location unit="IN">
                <x> -5 </x>
                <y> 0 </y>
                <z> 0 </z>
            </location>
            <orient unit="DEG">
                <roll> 0.0 </roll>
                <pitch> 0 </pitch>
                <yaw> 0 </yaw>
            </orient>
            <feed>0</feed>
            <feed>1</feed>
            <thruster file="xlr99_nozzle">
                <location unit="IN">
                    <x> -5 </x>
                    <y> 0 </y>
                    <z> 0 </z>
                </location>
                <orient unit="DEG">
                    <roll> 0.0 </roll>
                    <pitch> -90 </pitch>
                    <yaw> 0.0 </yaw>
                </orient>
            </thruster>
        </engine>
        <tank type="zbiornik_paliwowy">
            <location unit="IN">
                <x> 1 </x>
                <y> 0 </y>
                <z> 0 </z>
```

```
            </location>
            <capacity unit="LBS"> 10 </capacity>
            <contents unit="LBS"> 10 </contents>
        </tank>
</propulsion>
```

Powyższe otoczenie łączy wcześniej zdefiniowany silnik (plik o nazwie „xlr99.xml" znajdujący się w domyślnym domyślnie w folderze „/$JSBSim1.0/engine$") a następnie umiejscawia go w obiekcie według podanych wartości. Do silnika przypisany jest tutaj zbiornik na paliwo, gdzie oprócz jego położenia należy zdefiniować wagę paliwa które może się w nim zmieścić oraz ilość „jednostek" paliwa w nim się mieszczącą. Ostatnim krokiem będzie zdefiniowanie parametrów które to mają być zapisane w pliku wyjściowym symulacji

```
<output name="ladownik.csv" type="CSV" rate="1">
        <property> predkosciomierz </property>
        <rates> ON </rates>
        <velocities> ON </velocities>
        <forces> ON </forces>
        <moments> ON </moments>
        <position> ON </position>
</output>
```

Trzeba zdefiniować zarówno nazwę jak i typ pliku wyjściowego (najlepiej jest używać rozszerzenia csv jest ono bowiem najłatwiejsze do późniejszego przetwarzania), a następnie wszystkie potrzebne dane. Warto zauważyć, że prócz jednej zdefiniowanej przez użytkownika wielkości („predkosciomierz" odpowiada wartościom zmierzonym przez sensor), która to musi być zdefiniowana w otoczeniu **<property>**, wybrane wielkości (np. **<forces>**) są specjalnymi znacznikami i wyrażają zestaw wszystkich sił (tzn. sił wzdłuż wszystkich osi współrzędnych). Kolejnym plikiem jest plik zawierający wszystkie wartości początkowe symulacji jest on zwykle dość krótki i w tym wypadku ma postać

```
<initialize name="war_pocz">
 <ubody> unit="M/SEC">    3.1  </ubody>
  <vbody unit="M/SEC">     0.0  </vbody>
  <wbody unit="M/SEC">     0.0  </wbody>
  <latitude unit="DEG">    90.6  </latitude>
  <longitude unit="DEG">  -90.0  </longitude>
  <phi unit="DEG">          0.0  </phi>
  <theta unit="DEG">        0.0  </theta>
  <psi unit="DEG">         90.0  </psi>
  <altitude unit="M">    8000.0  </altitude>
</initialize>
```

Wielkości tu użyte nie wymagają dalszego komentarza (są zgodne z nazwami).

Po zdefiniowaniu obiektu w pliku **aircraft.xml**, silnika **engine.xml** oraz pliku **warpocz.xml** dla ułatwienia symulacji należy zapisać skrypt (także jako plik typu **xml**).

```
<runscript  name="landing">
  <description>Skrypt automatyzujący symulacje</description>
  <use aircraft="aircraft" initialize="warpocz"/>
  <run start="0.0" end="300" dt="0.001">
</runscript>
```

W znaczniku tym zdefiniowane są następujące parametry:
— wykorzystywany obiekt, w tym wypadku plik **aircraft.xml**,
— zestaw wartości początkowych zapisanych w pliku **warpocz.xml**,
— początkowy i końcowy czas symulacji (odpowiednio 0 i 300),
— krok symulacji, równy w tym wypadku $0.001s$.
W celu przeprowadzenia symulacji wystarczy podać wcześniej podaną komendę

```
jsbsim --script=(ścieżka_do_pliku/)nazwa_skryptu.xml
```

10.1.7. Opracowanie wyników – pomocne programy

Wynikiem symulacji są pliki typu **csv** zawierające wartości wcześniej zdefiniowanych wielkości (parametrów) w kolejnych chwilach. Do wygodnej analizy tych wyników może zostać wykorzystane każde środowisko rachunkowe (np. *Excel*, *OpenOffice*). Jednakże najwygodniejszym narzędziem do wizualizacji wyników wydaje się być *Gnuplot*, a to ze względu na możliwość szybkiego tworzenia skryptów znacznie przyśpieszających generowanie poszczególnych wykresów.

10.2. Podsumowanie

Środowisko `JSBSim` nie jest łatwe do oceny. Z jednej strony zaawansowane techniki modelowania zjawisk aerodynamicznych pozwalają na jego pełne i profesjonalne zastosowania[11], z drugiej strony przygotowanie nawet projektów prostych obiektów jest bardzo pracochłonne z powodu plików **xml**[12]. Poniżej zostały wyszczególnione główne zalety i wady środowiska `JSBSim`.
Zalety:
— jasny podział na definicje obiektów, otoczenia i warunków symulacji,
— szeroka gama modelowanych zjawisk aerodynamicznych[13],
— szeroka możliwość integracji z innymi środowiskami (*Python*, *FlightGear*, aplikacje własne z `JSBSim` jako biblioteką zewnętrzną),
— przenośność (środowisko jest kompatybilne z większością używanych systemów operacyjnych),
— dokumentacja obejmuje prawie wszystkie istotne zagadnienia.
— Bogaty zbiór gotowych przykładowych projektów (dość duża różnorodność),
— nie jest wymagana znajomość dynamiki obiektu.
Wady:
— brak środowiska ułatwiającego tworzenie plików wejściowych (**xml**), przez co bardzo utrudnia utworzenie pierwszego własnego projektu,
— niedokładne objaśnienia i niejasne stwierdzenia w dokumentacji (bądź też błędy[14]). Dokumentacja wydaje się być pisana bardzo chaotycznie. Nieuniknione jest niestety zapoznanie się z jej całością przed przystąpieniem do pracy (ok. 175 stron). Natomiast „studia przypadku" nie stanowią oddzielnej części dokumentacji (także wymagają zapoznania się z całością dokumentacji),

[11] http://jsbsim.sourceforge.net/documentation.html poniżej listy podstawowych instrukcji znajduje się lista udokumentowanych aplikacji środowiska `JSBSim`.
[12] Opis prostej kulki ze spadochronem zajmuje ok. 3 stron samego pliku typu *aircraft.xml*- zob. [2] str. 112.
[13] Wykorzystnie współczynników „coefficients" pozwala na dokładną reprezentację zjawisk aerodynamicznych.
[14] Wykres prędkości i położenia w [2] na str. 116 jest błędny

— wymagana jest duża wiedza na temat postaci zjawisk aerodynamicznych[15],
— brak implementacji prostego parsera ułatwiającego import do projektu, danych dotyczących masy i wymiarów,
— wymagana jest duża liczba dodatkowych środowisk/programów potrzebnych do utworzenia projektu lub opracowania wyników symulacji.

Bibliografia

[1] Autodesk. Autodesk Inventor. http://www.autodesk.pl/adsk/servlet/pc/index?siteID=553660&id=14578074.
[2] JSBSim. JSBSIM Manual. http://jsbsim.sourceforge.net/Building_JSBSim_with_Visual_Cpp.pdf.
[3] JSBSim. JSBSim-Matlab GUI. http://jsbsim.sourceforge.net/matlab.html.
[4] JSBSim. JSBSim xml php script. http://jsbsim.sourceforge.net/aeromatic2.html.

[15] Jest to wynik uboczny reprezentacji sił jako wcześniej wspomnianych „coefficients".

Część II

Zastosowania

11. Sztywny manipulator stacjonarny

Tomasz Jordanek, Łukasz Czyż

W tym rozdziale przedstawiony zostanie sposób modelowania sztywnych manipulatorów stacjonarnych. Przyjęto, że manipulator zbudowany jest z jednorodnych prętów tak, że środek ciężkości znajduje się dokładnie w połowie długości ramienia.

Do zamodelowania powyższego obiektu i przeprowadzenia symulacji wybrano dwa środowiska: Webots oraz Autodesk Inventor. Umożliwiło to porównanie zakresu ich możliwości i stosowalności. Głównym czynnikiem mającym wpływ na dokonany wybór było porównanie szybkości i złożoności implementacji modelu.

11.1. Metody modelowania sztywnych manipulatorów stacjonarnych

11.1.1. Model kinematyki

W celu wyznaczenia kinematyki manipulatora sztywnego skorzystano z notacji Denavita-Hartenberga. Składa się ona z trzech etapów:
— związanie z każdym przegubem manipulatora lokalnego układu współrzędnych,
— określenie ciągu transformacji sąsiednich układów współrzędnych:

$$A_{i-1}^{i}(q_i) = Rot(Z,\theta_i)Trans(Z,d_i)Trans(X,a_i)Rot(X,\alpha_i), \tag{11.1}$$

— wyznaczenie kinematyki manipulatora jako złożenia transformacji:

$$K(q) = \prod_{i=1}^{n} A_{i-1}^{i}(q_i) = \begin{bmatrix} R_0^n(q) & T_0^n(q) \\ 0 & 1 \end{bmatrix}. \tag{11.2}$$

11.1.2. Model dynamiki

Pierwszym krokiem do otrzymania dynamiki manipulatora jest wyznaczenie lagranżjanu, jako różnicy energii kinetycznej i potencjalnej ramion oraz układów napędowych. To wyprowadzenie przedstawia wzór

$$L(q,\dot{q}) = K(q,\dot{q}) - V(q). \tag{11.3}$$

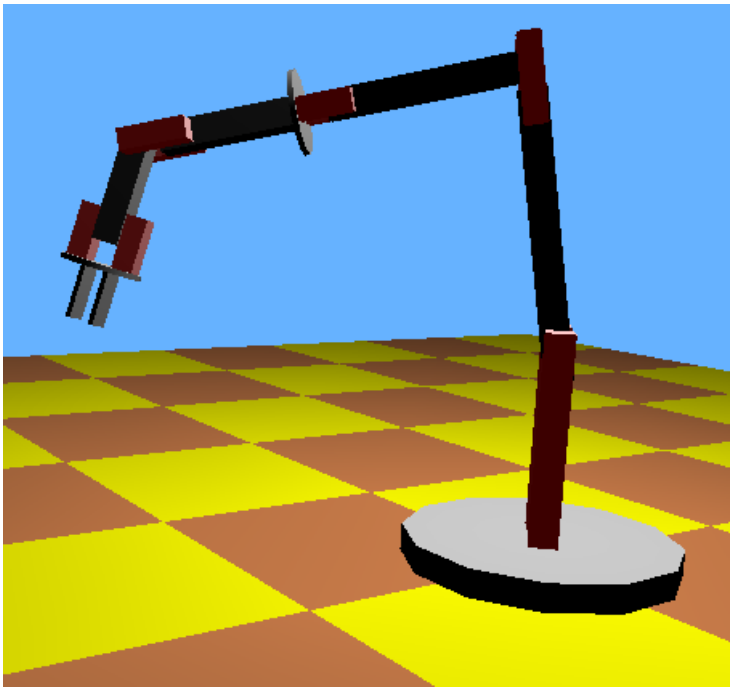
Następnie wykorzystano równanie Eulera-Lagrange'a:

$$\frac{d}{dt}\frac{\partial L(q,\dot{q})}{\partial \dot{q}} - \frac{\partial L(q,\dot{q})}{\partial q} = u. \tag{11.4}$$

Model dynamiki manipulatora sztywnego opisuje równanie

$$M(q)\ddot{q} + C(q,\dot{q})\dot{q} + D(q) = u. \tag{11.5}$$

Przyjęto następujące oznaczenia:
— M – macierz sił bezwładności,
— C – macierz sił Coriolisa i sił odśrodkowych,
— D – wektor sił grawitacji,
— u – wektor sterowania.



Rysunek 11.1. Model manipulatora sztywnego

11.2. Webots

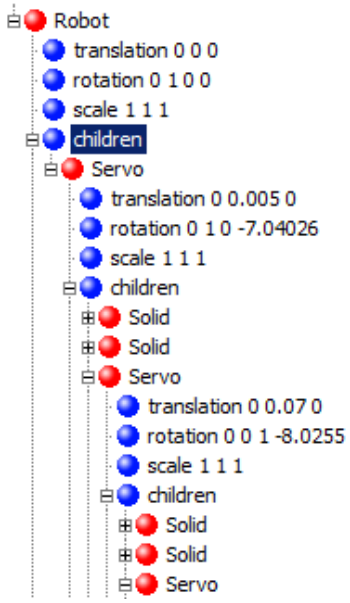
Środowisko Webots pozwala na szybkie modelowanie zarówno manipulatorów jak i robotów mobilnych. Główną zaletą programu jest możliwość przeprowadzenia prostych symulacji dynamicznych, pozwalających na badanie zachowania robota. Program jest bardzo intuicyjny a do jego obsługi nie jest wymagana żadna specjalistyczna wiedza.

11.2.1. Model wieloprzegubowego manipulatora sztywnego

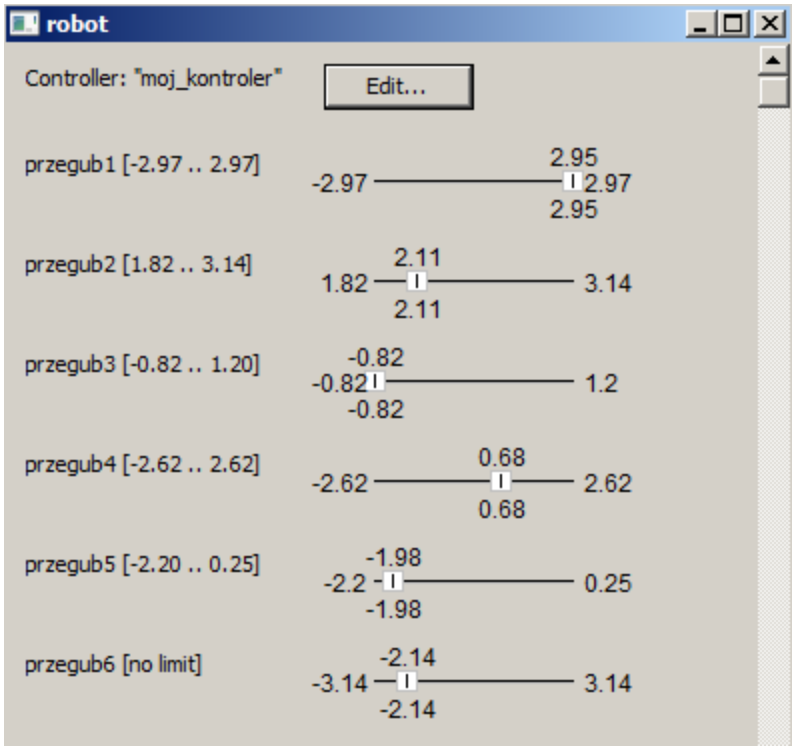
Modelowany obiekt to stacjonarny manipulator bez elastyczności o sześciu przegubach obrotowych. Model tak określonego manipulatora wprowadzony w środowisku Webots przedstawia rysunek 11.1. Przeguby obrotowe manipulatora w programie są realizowane poprzez obiekty typu `Servo`. Ich opcje pozwalają m. in. na wybór położenia przegubu na scenie, osi obrotu czy zdefiniowanie zakresu ruchu. Dodatkowo zdefiniowano elementy typu `Solid` reprezentujące ramiona robota. Ich kształt, wymiary czy kolor można zmienić przez dodanie obiektu `Shape`. Przy projektowaniu własnego modelu w środowisku Webots należy pamiętać, iż każdy nowy element stanowi gałąź drzewa wszystkich elementów robota. Stąd jako pierwszy powstaje obiekt typu `Robot`, który w grupie `Children` posiada jedno `Servo`. Jest to pierwszy przegub zdefiniowany w podstawie manipulatora, realizujący obrót całego robota. Kolejny przegub zostaje wstawiony do grupy `Children` pierwszego `Serva`. W analogiczny sposób dodawane są kolejne przeguby robota. Schemat budowy manipulatora w postaci drzewa komponentów ilustruje rysunek 11.2.

11.2.2. Symulacja zachowania obiektu

Środowisko Webots umożliwia przeprowadzenie prostej symulacji dynamicznej. Gotowy model można wprawić w ruch przez zadanie wartości położenia każdego z przegubów. W tym celu należy wybrać opcję `Robot` → `Robot Window`, co powinno przywołać okno pokazane na rysunku 11.3. W celu uzyskania zadowalających rezultatów symulacji, należy



Rysunek 11.2. Fragment drzewa elementów składowych manipulatora



Rysunek 11.3. Wybór zadanej pozycji przegubów

wprowadzić ograniczenia zakresu ruchów każdego z przegubów poprzez nadanie im odpowiednich wartości parametrów `minPosition` oraz `maxPosition` wyrażonych w radianach. Umożliwi to przeprowadzenie symulacji z uwzględnieniem zakresu ruchów realnych do osiągnięcia przez rzeczywisty manipulator.

11.2.3. Kontroler robota

Niewątpliwą zaletą oprogramowania Webots jest fakt, iż pozwala ono na pisanie własnych skryptów sterujących pracą manipulatora. Nowy kontroler generujemy wybierając z menu `Kreator` → `Nowy kontroler robota...` . Przykładowy skrypt sterownika przedstawia wydruk 11.1.

Wydruk 11.1. Przykładowy sterownik manipulatora

```
#include <webots/robot.h>
#include <stdio.h>
#include <webots/servo.h>
#include <math.h>
#define TIME_STEP 64

int main(int argc, char **argv)
{
  wb_robot_init();
  WbDeviceTag servo1 = wb_robot_get_device("przegub1");
  wb_servo_enable_position(servo1,TIME_STEP);
  do {
    wb_robot_step(TIME_STEP);
    double poz1 = wb_servo_get_position(servo1);
    printf("Przeguby: %.1f",poz1*180/M_PI);
    wb_servo_set_velocity(servo1,0.1);
  } while (wb_robot_step(TIME_STEP) != -1);
  wb_robot_cleanup();
  return 0;
}
```

Dla uproszczenia założono w nim sterowanie jedynie pierwszym przegubem manipulatora. W celu umożliwienia odczytywania położenia przegubu należy wywołać funkcje `wb_robot_get_device()` oraz `wb_servo_enable_position()`. Sam odczyt realizuje funkcja `wb_servo_get_position()`. Otrzymany wynik można wypisać do konsoli Webotsów w celu ciągłej weryfikacji parametrów symulacji robota. Po zapisaniu, skompilowaniu i zbudowaniu sterownika, można go przetestować na własnym modelu. W tym celu należy wybrać kontroler zmieniając wartość parametru `controller` w opcjach obiektu `Robot` w obszarze okna `Scene tree`. Warto dodać, że w środowisku Webots dostępnych jest szereg funkcji poszerzających możliwości sterownika. Ich szczegółowy opis można znaleźć w dokumentacji oprogramowania na stronie producenta [4].

11.3. Autodesk Inventor

Autodesk Inventor to zaawansowany modelator bryłowy, który służy do zamodelowania projektowanego urządzenia jako modelu 3D, a następnie wygenerowania na podsta-

wie modelu rysunków złożeniowych, wykonawczych, ofertowych, poglądowych i innych. Jest to narzędzie, które standardowo posiada wszystkie narzędzie niezbędne do tego aby poprawnie wykonać w zasadzie dowolny projekt w branży elektromaszynowej. Został on zaprojektowany specjalnie do projektowania obiektów o skomplikowanej strukturze, złożonych nawet z kilkunastu tysięcy komponentów. Modelowanie części w programie Inventor realizowane jest w oparciu o jądro ShapeManager. Wspomaganie tworzenia skomplikowanych konstrukcji zapewnia między innymi technika modelowania parametrycznego oraz projektowania adaptacyjnego. Nie należy pomijać tutaj bardzo rozbudowanych możliwości graficznych tego programu.

Funkcjonalne narzędzie projektowe pod nazwa Inventor 2010, w którym został wykonany projekt manipulatora Irb-6 jest kolejną wersją pakietu trójwymiarowego projektowania parametrycznego firmy AutoDesk Inc. Tworzone projekty składają sie z obiektów, które mają jak najwierniej odzwierciedlać przyszłą konstrukcję rzeczywistą. Projektowanie elementy mogą mieć nadane własności rzeczywistych materiałów konstrukcyjnych. W programie jest zintegrowana analiza ruchu oraz analiza naprężeń. Ogólna filozofia programu jest podobna do innych narzędzi tej klasy takich jak Solid Edge, Solid Works lub wyższej klasy takich jak Catia. Istotna zaleta programu jest podobieństwo poleceń i interfejsu z programem AutoCad 2010. Program jest pakietem zawierającym moduły przeznaczone do tworzenia części, zespołów i rysunków.

Ze względu na skomplikowanie wykonania symulacji tworzonego obiektu w programie Autodesk Inventor wykorzystano do tego celu środowisko Solid Edge with Synchronous Technology 4. System Solid Edge ST4 jest głównym składnikiem portfolio Velocity Series, która stanowi rodzinę w pełni zintegrowanych, prekonfigurowanych rozwiązań mających na celu zapewnienie kompleksowej kontroli cyfrowego procesu opracowywania produktu.

11.3.1. Model wieloprzegubowego manipulatora sztywnego

Zadanie ujęte w projekcie polegało na stworzeniu teoretcznego modelu stacjonarnego manipulatora o wielu stopniach swobody i przedstawieniu wyników porównujących ten model do obiektu rzeczywistego. Należało wziąć pod uwagę następujące czynniki:
— sposób tworzenia części wchodzących w skład robota,
— jakość i szczegółowość elementów użytych do budowy ramion i przegubów,
— odzwierciedlenie przegubów,
— fizyczne ograniczenia kątowe przy poruszaniu ramionami względem siebie,
— precyzja działania,
— sposób monitorowania parametrów czy nastaw.

W celu uniknięcia zbyt dużego skomplikowania modelu przyjęto następujące założenia:
— ramiona wykonane są z jednolitego materiału o środku ciężkości w połowie jego długości,
— brak tarcia statycznego i dynamicznego,
— przyjęto sześć stopni swobody.

Dla uproszczenia przyjęto również, że model będzie wzorowany na manipulatorze IRB-6.

11.3.2. Poruszanie się w Autodesk Inventor

Idea powstawania projektu w programie Inventor 2010 polega na tworzeniu obiektów opisywanych przez zbior polecen, który jest przypisany do każdej stworzonej figury. Podstawowym obiektem jest spójna pojedyncza bryła, uzyskana na podstawie operacji tworzenia bryły (najczęściej ze szkicu płaskiego) oraz szeregu operacji przetwarzania tej bryły w postać końcową przez operacje takie jak wyciaganie, wykonywanie otworów, dodawanie

fazowań czy zaokragleń. Tworzenie wiekszych jednostek (zespołów) odbywa sie na drodze składania ich z części. Utworzony zespół może być analizowany pod kątem ewentualnych konfliktów w ruchu poszczególnych mechanizmów jak i możliwa jest symulacja naprezeń wynikajaca z pracy ukladu pod zadanymi obciażeniami. Więcej informacji można znaleźć w [1].

11.3.3. Tworzenie modelu w Autodesk Inventor

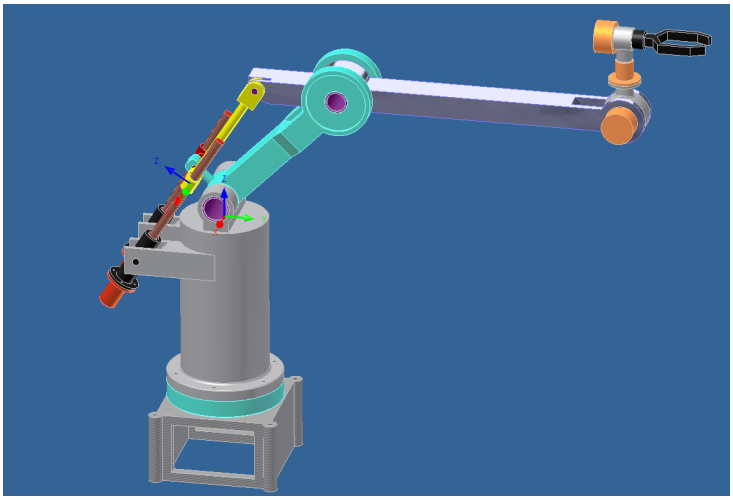
Projektowanie zaczyna sie od zdefiniowania katalogu (projektu), w którym będą zapisywane części oraz zespoły związane z opracowywaną konstrukcją [3]. Kolejny etap polega na wyborze odpowiedniego typu pliku przez polecenie `Nowy` w menu głównym okna. Rysowanie rozpoczyna sie w szkicowniku `Szkic 2D`, następnie poprzez możliwość tworzenia elementów typu linia, koło, prostokąt oraz wykonywaniu operacji takich jak fazuj, utnij itp. powstaje szkic, który należy ograniczyć odpowiednimi więzami geometrycznymi oraz wymiarami bazowymi podanymi na płaszczyźnie. Po zakończeniu szkicowania `koniec szkicu`, należy stworzyć trzecią oś elementu `Wyciagnij` w celu stworzenia bryły w 3D, która jest podstawowym elementem podczas projektowania. Na utworzonej tak bryle poprzez dowolny wybór płaszczyzny lub boku poleceniem `Szkic 2D` tworzony jest kolejny szkic i tym samym czynność zostaje powtorzona. Gdy utworzony element jest gotowy należy zapisać go w folderze projektu i przystąpić do tworzenia kolejnych części, które będą wchodziły w skład przyszłego zespołu, ponieważ grupa kilku cześci tworzy zespół. Tworzenie zespołu odbywa sie poprzez zainicjowanie nowego pustego dokumentu zespołu, nastepnie wstawienie istniejących i zapisanych na dysku cześci lub zespołów. Po wstawieniu cześci lub zespołu należy ustanowić pewne zalezności pomiedzy poszczególnymi elementami zespołu. Zależności te okreslają wzajemne położenie poszczególnych elementów, umożliwiając ich wzajemne dopasowanie. Każde wiazanie ogranicza stopień swobody układu, dlatego podczas wykonywania symulacji należy pamietać o wyłaczeniu cześci ograniczeń uniemożliwiajacych ruch elementów wzgledem siebie. Istnieje możliwość wstawienia do zespołu znormalizowanych części zawartych w bibliotece. Model wyprowadzonego robota pokazano na rysunku 1.4.

11.3.4. Eksport modelu do Solid Edge ST4

W celu utworzenia animacji gotowy zespół został wyeksportowany do programu Solid Edge ST4 gdzie nastepnie zdefiniowano momenty obrotowe elementów `Silnik` podając predkość oraz zakreślany kąt lub w przypadku posuwu liniowego odległość końcową. Poprzez polecenie `Symulacja` należy otworzyć dodatkowe okno w którym ustawiamy czas animacji oraz definiujemy momenty rozpoczęcia i zatrzymania ruchu każdego elementu wcześniej zdefiniowanego. Poleceniem `Film` tworzona jest animacja. Więcej informacji znajduje się w [2].

11.3.5. Symulacja zachowania obiektu

Etapy tworzenia modelu:
— zwymiarowanie całego manipulatora aby jego działania miały fizyczny sens,
— odpowiedni dobór tulei na przegubach,
— odpowiedni dobór przegubów,
— zintegrowanie wszystkich części w całość.



Rysunek 11.4. Robot stworzony za pomocą Autodesk Inventor

Problemy napotkane przy realizacji zadania

W programie Autodesk Inventor główną przeszkodą w wykonaniu zadania jest zbyt skomplikowany interfejs do przeprowadzenia prawidłowej symulacji. Z problemem poradzono sobie stosując eksport modelu do innej platformy i wykonanie symulacji.

11.4. Podsumowanie

Webots to wszechstronna aplikacja umożliwiająca w prosty i szybki sposób modelowanie zarówno manipulatorów jak i robotów mobilnych. Do jego największych zalet należy zaliczyć intuicyjną obsługę oraz możliwość pisania własnych kontrolerów, które następnie można wgrać do rzeczywistego robota. W pracy z tym środowiskiem można wyróżnić następujące etapy:
— budowa modelu,
— opracowanie skryptu sterującego robotem,
— przeprowadzenie symulacji,
— wgranie sterownika do rzeczywistego obiektu.

Niestety aplikacja nie jest pozbawiona wad. Pierwszą z nich jest fakt, iż w pełni funkcjonalna wersja programu bez ograniczeń dostępna jest tylko w wersji 30-dniowej, po uprzedniej rejestracji na stronie producenta. Do słabych stron można również zaliczyć brak możliwości dokładnego odwzorowania modelowanego obiektu. Można skorzystać jedynie z prostych brył geometrycznych bez możliwości precyzyjnej obróbki jakiej możemy wykonać projektując np. w środowisku Autodesk Inventor, jednak należy pamiętać, że Webots powstał w celu szybkiego prototypowania obiektów. Osoby bardziej wymagające, dla których priorytetem jest bardziej realistyczne odwzorowanie rzeczywistego robota wybiorą zapewne inne środowisko.

Środowisko Autodesk Inventor jest bardzo dobrym narzędziem do modelowania manipulatorów sztywnych. Dzięki wysokiej precyzji doboru elementów można bardzo dokładnie odwzorowac obiekt rzeczywisty. Pozwala on na łatwa weryfikację modeli teoretycznych. Należy zauważyć, że środowisko to nie pozwala na proste wykonanie symulacji. W związku z tym zalecane jest wykorzystanie w tym celu innego środowiska jakim jest na przykład Solid Edge.

11.4.1. Porównanie własności środowisk

Inventor:
1. Zalety:
   a) bogata biblioteka,
   b) tworzenie własnych obiektów i łączenie ich ze sobą,
   c) export modelu do innych środowisk,
   d) możliwość realizacji projektów rzeczywistych obiektów,
   e) standaryzacja części,
   f) możliwość tworzenia bardzo złożonych struktur.
2. Wady:
   a) duże wymagania sprzętowe,
   b) czasochłonne opracowanie modelu,
   c) obszerna instrukcja,
   d) złożoność programu.

Webots:
1. Zalety:
   a) małe wymagania sprzętowe,
   b) prostota obsługi,
   c) szybkość tworzenia modelu,
   d) prosta konfiguracja modelu,
   e) intuicyjny interfejs.
2. Wady:
   a) niska dokładność komponentów modelu,
   b) ogranicznone możliwości projektowe,
   c) brak możliwości eksportu modelu,
   d) skromna biblioteka podzespołów,
   e) brak standaryzacji części,
   f) brak wsparcia modeli z innych programów.

Bibliografia

[1] Autodesk Inventor. http://www.aplikom.com.pl/index/produkty/oprogramowanie/opis/164_Autodesk_Inventor.
[2] Solid Edge. http://www.gmsystem.pl/ProduktySolidEdge.php.
[3] Tutorial Inventor. http://www.simr.pw.edu.pl/~jbo/cad/inventor_cw4_zespol.pdf.
[4] Webots tutorial. http://www.cyberbotics.com/reference.pdf.

12. Elastyczny manipulator stacjonarny

Michał Kot, Michał Wcisło

Manipulatory niezaliczające się do grona manipulatorów sztywnych nazywa się elastycznymi. Elastyczności mogą występować zarówno w ogniwach (ramionach), jak i przegubach manipulatora. Oba przypadki komplikują proces modelowania obiektu, jednakże elastyczności ogniw są bardziej złożonym problemem, który zostanie w niniejszym rozdziale pominięty.

12.1. Elastyczne przeguby

Zjawisko elastyczności pomiędzy napędami, a sterowanymi przegubami powoduje obecność elementów przekładniowych w nowoczesnych robotach przemysłowych. Celem przekładni jest przeniesienie ruchu w inne miejsce, zamiana ruchu obrotowego na liniowy (lub odwrotnie), a także zwiększenie mechanicznej podatności robota przy jednoczesnej dokładności pozycjonowania przegubów między innymi przy współpracy z człowiekiem. Możliwa staje się kontrolę sił, którymi manipulator oddziałuje na otoczenie, dzięki czemu zmniejszone jest prawdopodobieństwo destrukcyjnych efektów pracy robota. Modelowanie elastycznych przegubów prezentowane w niniejszym rozdziale zakłada całkowitą sztywność ogniw manipulatora.

12.1.1. Konfiguracja manipulatora

Liczba zmiennych potrzebnych do zapisu bieżącej konfiguracji manipulatora ulega podwojeniu w stosunku do manipulatorów sztywnych. Występujący w manipulatorach sztywnych wektor zmiennych przegubowych oznaczanych jako $q \in \mathbb{R}^{\mathbb{N}}$ zapisywany jest jako $q^1 \in \mathbb{R}^{\mathbb{N}}$, natomiast dodatkowo pojawia się wektor zmiennych napędowych $q^2 \in \mathbb{R}^{\mathbb{N}}$. Sterowane napędy działają bezpośrednio na zmienną q^2, natomiast różnice pomiędzy q^1 i q^2 wynikają z elastyczności przegubów występujących w obiekcie, modelowanych jako liniowa sprężyna umieszczona pomiędzy napędem, a przegubem.

12.1.2. Podstawowe założenia

Usprawnienia procesu modelowania manipulatorów elastycznych uzyskuje się poprzez określenie kilka podstawowych założeń [2], które eliminują niechciane zjawiska fizyczne:

1. niewielkie, a w konsekwencji liniowe przemieszczenie q^1 względem q^2,
2. środki mas napędów umieszczone w osiach obrotów przegubów,
3. każdy napęd znajduje się przed napędzanym ogniwem,
4. kątowa energia kinetyczna każdego z napędów nie zależy od pozostałych napędów.

Powyższe założenia prowadzą do uproszczenia modelu dynamiki manipulatora. Analiza przypadku, w którym ostatnie z założeń zostało pominięte znajduje się w rozdziale 12.1.9.

12.1.3. Pomiar aktualnej konfiguracji

Pomiar pełnej konfiguracji manipulatora wymaga dwukrotnie większej liczby czujników, ponieważ posiadamy dwukrotnie więcej zmiennych stanu. Dla robota składającego się z N przegubów konieczne jest wykorzystanie 4N czujników do pomiaru położeń i prędkości: $q^1, q^2, \dot{q}^1, \dot{q}^2$. Pomiar aktualnych położeń i prędkości napędów można mierzyć podobnie jak w manipulatorach sztywnych z wykorzystaniem enkoderów i tachometrów. Sposób pomiaru położeń i prędkości przegubów jest bardziej skomplikowany. Istnieją jednakże złożone systemy czujników, umożliwiające jednoczesny pomiar konfiguracji zarówno napędów jak i ogniw manipulatora [1].

12.1.4. Model kinematyki

Model kinematyki manipulatorów elastycznych wyznacza się z wykorzystaniem konfiguracji q^1 analogicznie do manipulatorów sztywnych, np. z wykorzystaniem notacji Denevita-Hartenberga. Kinematykę macierzową przedstawia równanie

$$K(q^1) = A_0^n(q^1) = \prod_{i=1}^{n} A_{i-1}^i(q_i^1) = \begin{bmatrix} R_0^n(q^1) & d_0^n(q^1) \\ 0 & 1 \end{bmatrix} \in SE(3). \tag{12.1}$$

12.1.5. Model dynamiki

Model dynamiki manipulatora elastycznego można wyprowadzić korzystając z zasady najmniejszego działania Eulera-Lagrange'a. W przypadku braku tarcia i sił zewnętrznych działających na manipulator (poza sterowaniem, czyli momentami napędów u), a także wykorzystując założenia z rozdziału 12.1.2 model dynamiki manipulatora elastycznego można zapisać za pomocą następujących równań:

$$\begin{cases} M(q^1)\ddot{q}^1 + C(q^1, \dot{q}^1)\dot{q}^1 + D(q^1) + K(q^1 - q^2) = 0, \\ I_s\ddot{q}^2 + K(q^2 - q^1) = u, \end{cases} \tag{12.2}$$

gdzie M jest macierzą bezwładności, C jest macierzą oddziaływań Coriolisa i sił odśrodkowych, D jest wektorem grawitacji, $I_s = diag\{I_1, I_2, ..., I_n\}$ jest macierzą bezwładności wirników silników, $K = diag\{k_1, k_2, ..., k_n\}$ jest macierzą współczynników elastyczności przegubów,u jest sterowaniem momentami silników.

Gdy wartości macierzy K dążą do nieskończoności wektory zmiennych przegubowych i napędowych stają się równe, dzięki czemu układ równań (12.2) po zsumowaniu upraszcza się do standardowego modelu dynamiki sztywnego manipulatora

$$(M(q^1) + I_s)\ddot{q}^1 + C(q^1, \dot{q}^1)\dot{q}^1 + D(q^1) = u. \tag{12.3}$$

12.1.6. Odwrotna dynamika

Zadaniem odwrotnej dynamiki jest wyznaczenie momentów w silnikach $\tau_d(t)$, które zapewnią pożądane trajektorie przegubów $q_d^1(t)$. W manipulatorach elastycznych nie ma możliwości bezpośredniego sterowania przegubami $q^1(t)$, w związku z czym konieczne staje się obliczenie trajektorii zadanych położeń napędów $q_d^2(t)$ według wzoru

$$q_d^2 = q_d^1 + K^{-1}(M(q_d^1)\ddot{q}_d^1 + C(q_d^1, \dot{q}_d^1)\dot{q}_d^1 + D(q_d^1)). \tag{12.4}$$

Prędkości $\dot{q}_d^2(t)$ i przyspieszenia $\ddot{q}_d^2(t)$ oblicza się różniczkując równanie (12.4). Wymusza to zadanie trajektorii przegubów wraz z pochodnymi do czwartego rzędu. Z wykorzystaniem zadanych przyspieszeń silników możliwe jest obliczenie pożądanych momentów ze wzoru

$$\tau_d = I_s \ddot{q}_d^2 + K(q_d^2 - q_d^1). \tag{12.5}$$

12.1.7. Zadania sterowania

Zadania sterowania dla manipulatorów elastycznych są takie same jak dla manipulatorów sztywnych. Podstawową różnicą dla algorytmów sterowania jest potrzeba przeliczenia zadanych konfiguracji ogniw $q_d^1(t)$ na konfiguracje napędów $q_d^2(t)$, tak jak zostało to przedstawione w rozdziele 12.1.6. Do powszechnie realizowanych zadań sterowania dla manipulatorów elastycznych należą:

— sterowanie do punktu $(q^1, \dot{q}^1) = (q_d^1, 0)$,
— śledzenie zadanej trajektorii przegubowej $(q^1(t) = q_d^1(t))$,
— śledzenie zadanej trajektorii efektora w przestrzeni kartezjańskiej $(y(t) = y_d(t))$,
— sterowanie siłą przy kontakcie manipulatora z otoczeniem.

We wszystkich przypadkach trajektoria $q_d^2(t)$ nie jest zadawana, lecz może być wyliczona ze wzoru (12.4).

12.1.8. Różne warianty sprzężenia zwrotnego

Sterowanie manipulatorem elastycznym z wykorzystaniem regulatora PD wymaga zastosowania sprzężenia zwrotnego. Ze względu na podwójną liczbę zmiennych stanu możliwe jest modelowanie sprzężenia na cztery różne sposoby. Każdy z nich wybiera dwie zmienne stanu, które są podłączane do części proporcjonalnej i różniczkującej regulatora:
1. $\tau = u - (K_P q^1 + K_D \dot{q}^1)$ – sprzężenie zwrotne położeń i prędkości ogniw,
2. $\tau = u - (K_P q^2 + K_D \dot{q}^2)$ – sprzężenie zwrotne położeń i prędkości napędów,
3. $\tau = u - (K_P q^1 + K_D \dot{q}^2)$ – sprzężenie zwrotne położeń ogniw i prędkości napędów,
4. $\tau = u - (K_P q^2 + K_D \dot{q}^1)$ – sprzężenie zwrotne położeń napędów i prędkości ogniw.

Zostało udowodnione [2], że jedynie drugie i trzecie sprzężenie jest stabilne dla wartości $K_P > 0$ i $K_D > 0$. Ponadto w trzecim sprzężeniu wartość K_P nie może przekroczyć pewnej konkretnej wartości (zależnej od samego manipulatora), w związku z czym mamy do czynienia z ograniczonym wzmocnieniem proporcjonalnym.

12.1.9. Zależność kątowej energii kinetycznej

W literaturze bardzo często [4], [2], [3] spotyka się model dynamiki manipulatora z pominiętym założeniem dotyczącym braku wzajemnego wpływu kątowych energii kinetycznych napędów. Powoduje to pojawienie się górno-trójkątnej macierzy S reprezentującej bezwładność pomiędzy silnikami, a przyspieszeniami ogniw. Rozbudowany model zawiera układ równań

$$\begin{cases} M(q^1)\ddot{q}^1 + S\ddot{q}^1 + C(q^1, \dot{q}^1)\dot{q}^1 + D(q^1) + K(q^1 - q^2) = 0, \\ S^T\ddot{q}^1 + I_s\ddot{q}^2 + K(q^2 - q^1) = u. \end{cases} \tag{12.6}$$

12.2. Modelowanie manipulatorów elastycznych w środowisku MATLAB

MATLAB jako jedno z najczęściej stosowanych środowisk symulacyjnych jest niezwykle przydatnym narzędziem przy modelowaniu manipulatorów elastycznych. Umożliwia on użytkownikowi realizację złożonych modeli manipulatorów zarówno poprzez wbudowany język skryptowy jak i z wykorzystaniem pakietu Simulink, w którym preferowana jest graficzna reprezentacja modelu układu. Dla gotowych modeli dostępne są zaawansowane techniki symulacyjne, które pozwalają na pełną analizę dynamiczną manipulatora.

12.2.1. Przygotowanie modelu matematycznego manipulatora

Przed przystąpieniem do modelowania manipulatora w środowisku MATLAB niezbędne jest przygotowanie matematycznego modelu dynamiki robota w postaci przedstawionej układem równań (12.2). W tym celu można skorzystać z not katalogowych producenta, w których czasem znajdują się informacje umożliwiające obliczenie wartości macierzy bezwładności, Coriolisa i bezwładności silników, a także wektora grawitacji.

Gotowa dynamika w postaci (12.2) jest gotowa do implementacji z wykorzystaniem pakietu Simulink. Na początku należy przemnożyć z lewej strony drugie równanie przez odwrotną macierz bezwładności silników, co po podwójnym scałkowaniu pozwoli nam otrzymać położenia napędów q^2. Następnie otrzymana zmienna podstawiana jest do równania pierwszego (różnica z położeniem przegubów q^1 jest mnożona przez macierz sztywności K). Otrzymana w ten sposób zależność przypomina model dynamiki manipulatora sztywnego z dodatkowym członem zależnym od położeń napędów, co umożliwia zastosowanie analogicznego toku postępowania przy implementacji. Przykładowy proces modelowania elastycznego realizujący przedstawioną koncepcję znajduje się w rozdziale 12.2.2.

12.2.2. Model elastycznego trójwahadła z wykorzystaniem toolboxa Simulink

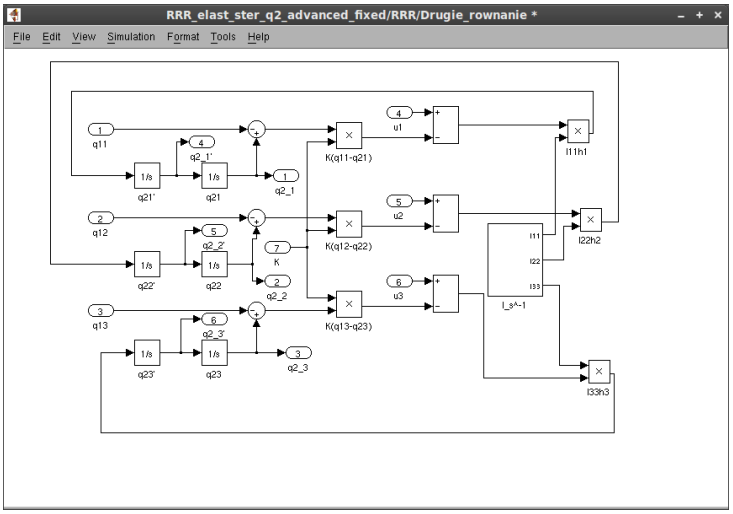
W celu zaprezentowania procesu konstruowania modelu manipulatora elastycznego w pakiecie Simulink środowiska MATLAB zrealizowany został model manipulatora RRR (potrójne wahadło) o trzech przegubach obrotowych. Macierze M, C, I_s i K (opisane w rozdziale 12.1.5) posiadają wymiar 3x3 ze względu na trzy stopnie swobody robota, natomiast wektor grawitacji posiada trzy wiersze.

Drugie równanie modelu dynamiki

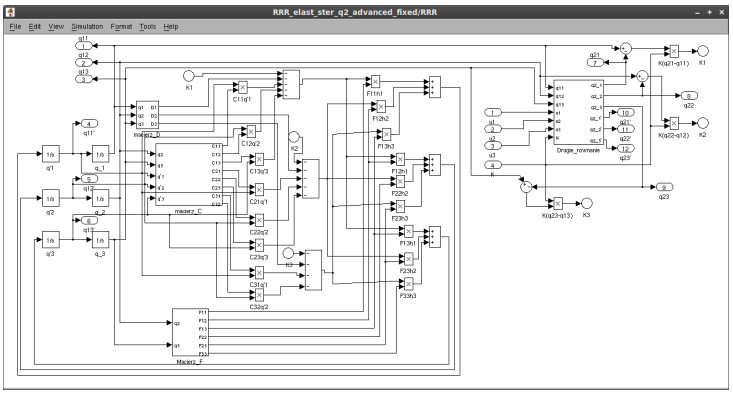
Budowę obiektu z bloczków Simulinka rozpoczynamy od zrealizowania drugiego równania modelu dynamiki (12.2), tak jak zostało to przedstawione na rysunku 12.1. Wejściami (bloki `Inport`) do podsystemu są aktualne położenia przegubów i ewentualnie sterowania (zostanie to sprecyzowane w rozdziale 12.2.3), natomiast na wyjściu (bloki `Outport`) znajdują się położenia napędów manipulatora. Różnica pomiędzy położeniami napędów i przegubów mnożona jest przez macierz sztywności, odejmowana od sterowania, a następnie mnożona przez odwrotność macierzy I_s. Tak otrzymane przyspieszenia napędów $\ddot{q}^2$ są podwójnie całkowane.

Pierwsze równanie modelu dynamiki

Wynikowe wartości położeń q^2 należy podstawić do pierwszego równania. Otrzymany w ten sposób układ został zaprezentowany na rysunku 12.2. Przypomina on standardowy model dynamiki dla manipulatorów sztywnych, jednakże sterowanie jest ukryte w części odpowiadającej drugiemu równaniu, natomiast macierze M, C i D, przemnożone przez



Rysunek 12.1. Implementacja drugiego rownania modelu dynamiki



Rysunek 12.2. Model dynamiki manipulatora RRR

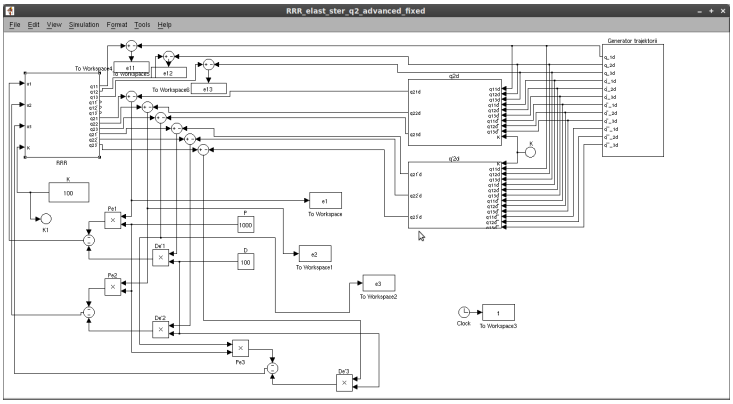
odpowiednie pochodne zmiennych stanu, sumują się z z dodatkowym wyrazem $K(q^2 - q^1)$ dając w rezultacie przyspieszenia przegubów $\ddot{q}^1$. Po podwójnym scałkowaniu otrzymujemy aktualne położenia przegubów.

12.2.3. Zadanie sterowania elastycznego trójwahadła

Algorytmy sterowania zadanej trajektorii przegubowej dla manipulatorów elastycznych są bardziej skomplikowane od algorytmów dla manipulatorów sztywnych. Jednym ze skuteczniejszych jest algorytm całkowania wstecznego (ang. backstepping), którego opis znajduje się w [5]. Innym podejściem jest zastosowanie statycznego regulatora PD (algorytm Qu i Dorsey'a), którego warianty zostały opisane w rozdziale 12.1.7. Na rysunku 12.3 znajduje się przykład zastosowania regulatora PD dla zamodelowanego trójwahadła. Wybrany wariant trzeci charakteryzuje się potrzebą obliczenia błędów położeń i prędkości napędów w celu zastosowania sprzężenia zwrotnego i otrzymania sterowania zgodnie ze wzorem

$$u = -K_P e^2 - K_D \dot{e}^2. \tag{12.7}$$

Błędy położeń e^2 i prędkości $\dot{e}^2$ wymagają zadanych trajektorii napędów. Wyznaczenie położeń $q_d^2(t)$ z wykorzystaniem wzoru (12.4) jest stosunkowo proste. Zadanie komplikuje się przy obliczaniu zadanych prędkości $\dot{q}_d^2(t)$, ponieważ konieczne jest zróżniczkowanie



Rysunek 12.3. Algorytm Qu i Dorsey'a

równania (12.4). Wiąże się to z koniecznością policzenia pochodnych po czasie wektora grawitacji, a także macierzy bezwładności i Coriolisa, co dla skomplikowanych manipulatorów jest zadaniem pracochłonnym. Wzór na zadane prędkości napędów dany jest równaniem

$$\dot{q}_d^2 = \dot{q}_d^1 + K^{-1}(M(q_d^1)\dddot{q}_d^1 + \dot{M}(q_d^1)\ddot{q}_d^1 + \dot{C}(q_d^1, \dot{q}_d^1, \ddot{q}_d^1)\dot{q}_d^1 + C(q_d^1, \dot{q}_d^1)\ddot{q}_d^1 + \dot{D}(q_d^1)). \qquad (12.8)$$

Jak nietrudno zauważyć, do obliczenia zadanych prędkości napędów potrzebujemy także trzeciej pochodnej $\dddot{q}_d^1$, co wymusza odpowiednio gładką trajektorię zadaną.

Trajektoria zadana

Wykorzystane w symulacji trajektorie zadane przegubów zostały przedstawione na rysunku 12.4. Przebiegi czasowe położeń przegubów w założeniu mają śledzić następujące funkcje:
— $q_{1d}^1(t) = 0.5 \cdot sin(t)$,
— $q_{2d}^1(t) = e^{-\frac{t}{2}}$,
— $q_{3d}^1(t) = cos(\frac{t}{2})$.

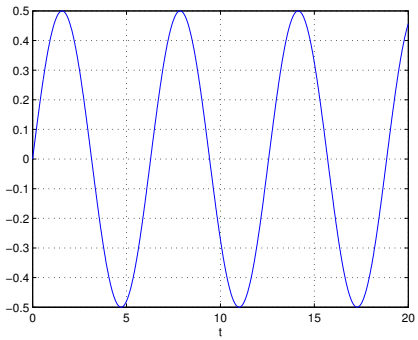
Parametry symulacji

Wszystkie symulacje zostały uruchomione w tym samym środowisku, z parametrami ustawionymi w następujący sposób:
— czas symulacji: 100 sekund,
— metoda całkowania: ode23 (Bogacki-Shampine),
— dokładność obliczeń: 10^{-6}.

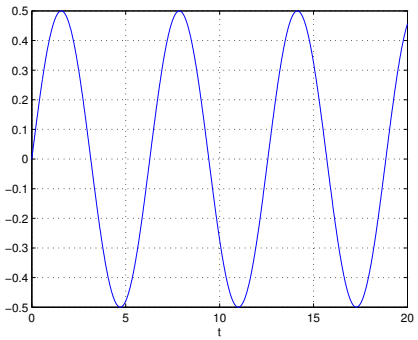
Błąd śledzenia trajektorii zadanej

Na rysunkach 12.5, 12.6 i 12.7 zamieszczono wykresy błędów śledzenia położeń przegubów w funkcji czasu dla trzech różnych nastaw regulatora. W obu przypadkach system zachowuje się stabilnie, jednakże występujące oscylacje i szybkość zbieżności błędu do zera zależą w dużej mierze od parametrów P i D. Nastawy regulacji powinny być dobrane odpowiednio dla manipulatora, ponieważ zarówno zbyt duże jak i zbyt małe wartości mogą prowadzić do niestabilności algorytmu.

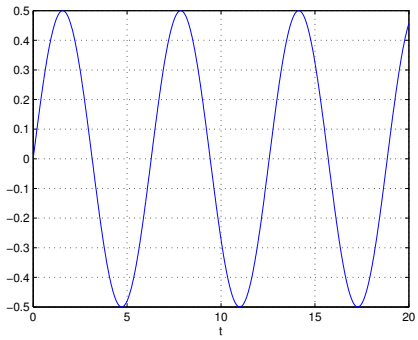
Lepsza zbieżność błędu otrzymywana jest w przegubach bliżej efektora. Wynika to z wartości macierzy bezwładności silników I_s, które maleją dla kolejnych przegubów ze względu na mniejszą wymaganą moc silników.



(a) $q^1_{1d}(t)$

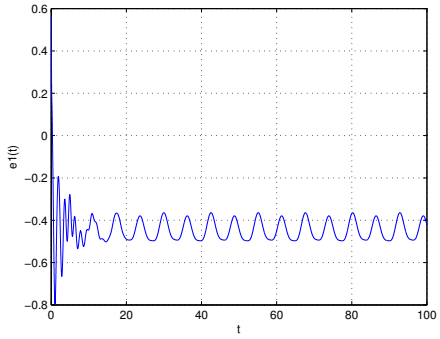


(b) $q^1_{2d}(t)$

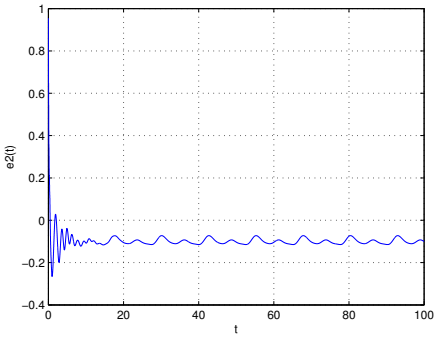


(c) $q^1_{3d}(t)$

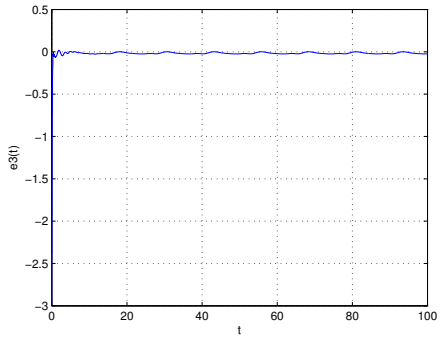
Rysunek 12.4. Trajektorie zadane przegubów



(a) $e_1^1(t)$ dla P=100, D=10

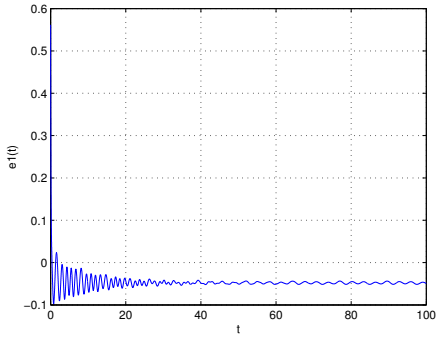


(b) $e_2^1(t)$ dla P=100, D=10

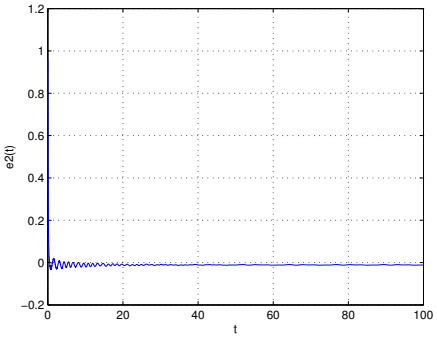


(c) $e_3^1(t)$ dla P=100, D=10

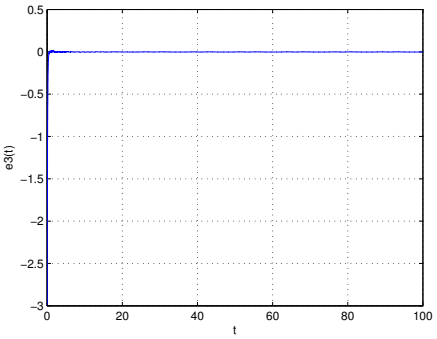
Rysunek 12.5. Błędy śledzenia położeń przegubów w funkcji czasu dla nastaw P=100, D=10



(a) $e_1^1(t)$ dla P=1000, D=100

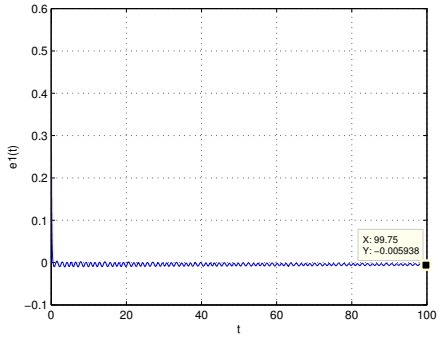


(b) $e_2^1(t)$ dla P=1000, D=100

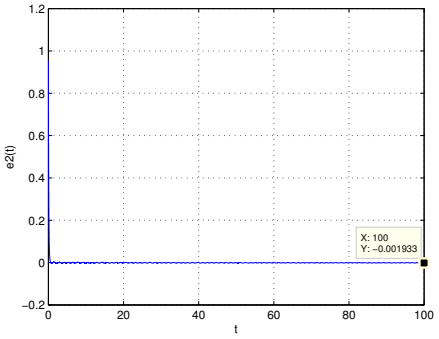


(c) $e_3^1(t)$ dla P=1000, D=100

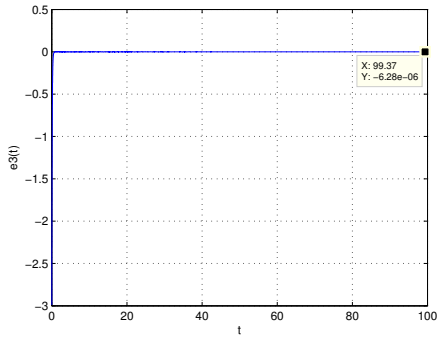
Rysunek 12.6. Błędy śledzenia położeń przegubów w funkcji czasu dla nastaw P=1000, D=100



(a) $e_1^1(t)$ dla P=10000, D=1000



(b) $e_2^1(t)$ dla P=10000, D=1000



(c) $e_3^1(t)$ dla P=10000, D=1000

Rysunek 12.7. Błędy śledzenia położeń przegubów w funkcji czasu dla nastaw P=10000, D=1000

12.3. Modelowanie manipulatorów elastycznych w środowisku Mathematica

Jako drugie narzędzie do modelowania manipulatora z elastycznościami w przegubie wykorzystano środowisko Mathematica. Jest to bardzo rozbudowana aplikacja stworzona na potrzeby obliczeń symbolicznych, ale obecnie wykorzystywana jest w wielu dziedzinach nauki.

12.3.1. Implementacja matematycznego modelu dynamiki badanego manipulatora

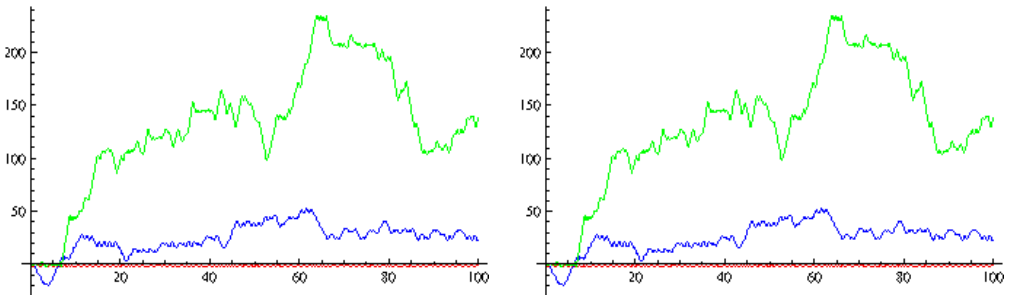
Model badanego manipulatora można wprowadzić do środowiska Mathematica bezpośrednio wykorzystując układ równań 12.2. Po wprowadzeniu macierzy sił bezwładności, Coriolisa i grawitacji wymienione równanie wyprowadza się w sposób przedstawiony poniżej:

```
dynamics = Flatten[{
   Thread[(mmat.q''[t])[[1]] + (cmat.q'[t])[[1]] +
      dmat[[1]] + (kmat.(q[t] - qq[t]))[[1]] == 0],
   Thread[(mmat.q''[t])[[2]] + (cmat.q'[t])[[2]] +
      dmat[[2]] + (kmat.(q[t] - qq[t]))[[2]] == 0],
   Thread[(mmat.q''[t])[[3]] + (cmat.q'[t])[[3]] +
      dmat[[3]] + (kmat.(q[t] - qq[t]))[[3]] == 0],
   Thread[(imat.qq''[t])[[1]] + (kmat.(qq[t] - q[t]))[[1]] ==
     u[t][[1]]],
   Thread[(imat.qq''[t])[[2]] + (kmat.(qq[t] - q[t]))[[2]] ==
     u[t][[2]]],
   Thread[(imat.qq''[t])[[3]] + (kmat.(qq[t] - q[t]))[[3]] ==
     u[t][[3]]]

   }];
init = {q1[0] == 0, q2[0] == -\[Pi]/4, q3[0] == \[Pi]/4, q1'[0] == 0,
   q2'[0] == 0, q3'[0] == 0, qq1[0] == 0, qq2[0] == -\[Pi]/4,
   qq3[0] == \[Pi]/4, qq1'[0] == 0, qq2'[0] == 0, qq3'[0] == 0};
```

Następnie należy rozwiązać układ równań różniczkowych opisujących dynamikę modelowanego trójwahadła wykorzystując polecenie NDSolve oraz wyświetlić rozwiązanie:

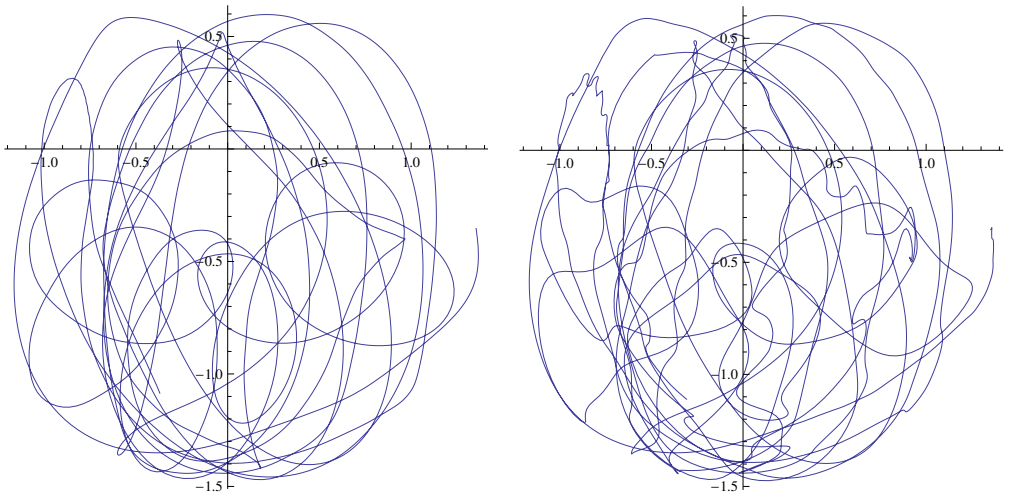
```
sol = NDSolve[
   Join[Flatten[{dynamics, init}]], {q1, q2, q3, qq1, qq2, qq3}, {t,
    0, 100}, MaxSteps -> Infinity] /. {u1[t] -> 0, u2[t] -> 0,
   u3[t] -> 0, i1 -> 0.2, i2 -> 0.05, i3 -> 0.005}

Plot[{q1[t] /. sol, q2[t] /. sol, q3[t] /. sol}, {t, 0, 100},
 PlotStyle -> {{Red}, {Blue}, {Green}}, PlotRange -> All]

Plot[{qq1[t] /. sol, qq2[t] /. sol, qq3[t] /. sol}, {t, 0, 100},
 PlotStyle -> {{Red}, {Blue}, {Green}}, PlotRange -> All]
```

Wynik przedstawionych poleceń w formie wykresu został przedstawiony na rysunku 12.8. Trajektorię efektora we współrzędnych przegubowych (q_1) i we współrzędnych napędów (q_2) przedstawiono na rysunku 12.9.



(a) q_1 (b) $q2$

Rysunek 12.8. Rozwiązanie układu równań różniczkowych opisujących trójwahadło z elastycznościami w przegubach (bez sił sterowania)



(a) q_1 (b) $q2$

Rysunek 12.9. Trajektoria efektora trójwahadło z elastycznościami w przegubach (bez sił sterowania)

12.3.2. Zadanie sterowania elastycznego trójwahadła

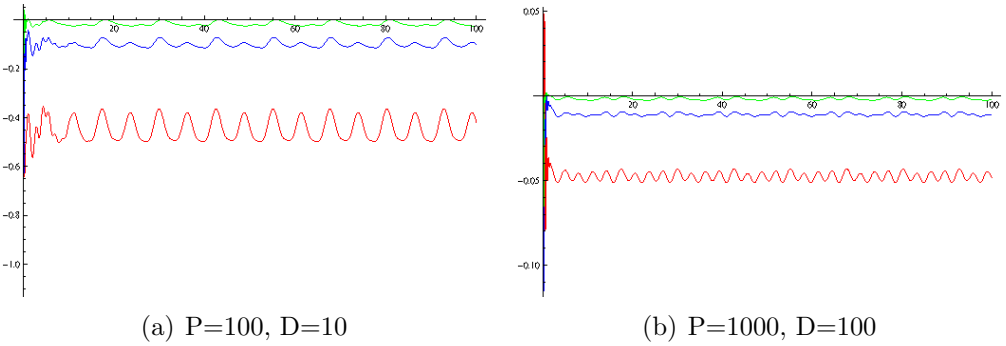
W celu weryfikacji stworzonego modelu zaimplementowano algorytm Qu i Dorsey'a (statyczna regulacja PD). Zadana trajektoria została przedstawiona w poprzedniej części (rysunek 12.4). Zgodnie z 12.1.6 aby wyliczyć zadaną trajektorię w we współrzędnych napędowych. Można to wykonać wykorzystując kod zamieszczony poniżej:

```
dynamicsMan = Flatten[{
    Thread[
     q1d[t] + (Inverse[kmat].mmat.qd''[t])[[
         1]] + (Inverse[kmat].cmat.qd'[t])[[
         1]] + (Inverse[kmat].dmat)[[1]] == qq1d[t]],
    Thread[
     q2d[t] + (Inverse[kmat].mmat.qd''[t])[[
         2]] + (Inverse[kmat].cmat.qd'[t])[[
         2]] + (Inverse[kmat].dmat)[[2]] == qq2d[t]],
    Thread[
     q3d[t] + (Inverse[kmat].mmat.qd''[t])[[
         3]] + (Inverse[kmat].cmat.qd'[t])[[
         3]] + (Inverse[kmat].dmat)[[3]] == qq3d[t]]
    }];

sold = NSolve[
    Join[Flatten[{dynamicsMan}]], {qq1d[t], qq2d[t],
     qq3d[t]}] /. {q1[t] -> q1d[t], q2[t] -> q2d[t], q3[t] -> q3d[t],
    q1'[t] -> q1d'[t], q2'[t] -> q2d'[t], q3'[t] -> q3d'[t],
    i1 -> 0.2, i2 -> 0.05, i3 -> 0.005};
```

Następnie należy już tylko rozwiązać układ równań różniczkowych

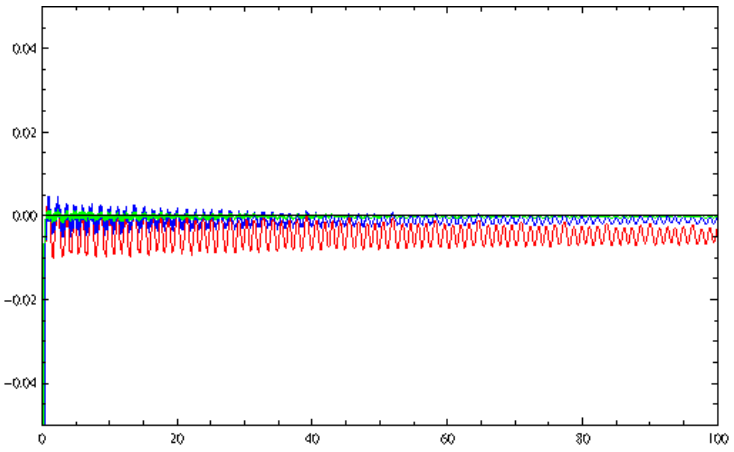
```
e1[t] = qq1[t] - qq1dd[t]; e2[t] = qq2[t] - qq2dd[t];
e3[t] = qq3[t] - qq3dd[t]; e1'[t] = qq1'[t] - qq1dd'[t];
e2'[t] = qq2'[t] - qq2dd'[t]; e3'[t] = qq3'[t] - qq3dd'[t];
qq1dd[t] = (qq1d[t] /. sold)[[1]];
qq2dd[t] = (qq2d[t] /. sold)[[1]];
qq3dd[t] = (qq3d[t] /. sold)[[1]]
qq1dd'[t] = D[qq1dd[t], t]; qq2dd'[t] = D[qq2dd[t], t];
qq3dd'[t] = D[qq3dd[t], t]; qq1dd''[t] = D[qq1dd'[t], t];

p = 100; d = 10;

sol2 = NDSolve[
    Join[Flatten[{dynamics, init}]], {q1, q2, q3, qq1, qq2, qq3}, {t,
     0, 100},
    MaxSteps -> 1000000000] /. {u1[t] -> -p e1[t] - d e1'[t],
    u2[t] -> -p e2[t] - d e2'[t], u3[t] -> -p e3[t] - d e3'[t],
    i1 -> 0.2, i2 -> 0.05, i3 -> 0.005};
```

Należy zwrócić uwagę, że jako błąd prędkości i położenia wykorzystano współrzędne napędów. Przykładowe wykresy błędów położenia dla różnych nastaw regulatora przedstawiono na rysunku 12.10. Duża ilość oscylacji, które są szczególnie widoczne dla dużych nastaw regulatora jest spowodowana elastycznością przegubową modelowanego manipulatora. Można także zauważyć charakterystyczne dla algorytmu Qu i Dorsey'a zmniejszanie się błędów proporcjonalnie do zwiększania nastaw regulatora.



(a) P=100, D=10

(b) P=1000, D=100



(c) P=10000, D=1000

Rysunek 12.10. Błędy położenia dla wszystkich przegubów dla różnych nastaw w algorytmie Qu i Dorsey'a

12.4. Porównanie środowisk MATLAB i Mathematica

MATLAB jest bardzo dobrym narzędziem do modelowania manipulatorów z elastycznymi przegubami.Poznanie środowiska MATLAB przebiega bez większych problemów, uzyskanie żądanego wyniku polega na odnalezieniu odpowiedniej funkcji. Pakiet Simulink pozwala na tworzenie intuicyjnych schematów obiektów opisanych równaniami dynamiki. Podwójna liczba zmiennych stanu komplikuje problem sterowania, jednakże zastosowanie odpowiednich algorytmów pozwala w pewnym zakresie wymusić żądane zachowanie układu. Przy zastosowaniu statycznego regulatora PD należy koniecznie pamiętać o poprawnym doborze sprzężenia zwrotnego i nastaw regulacji.

Mathematica jest bardzo wszechstronnym narzędziem, pozwalającym uzyskiwać wyniki właściwie po bezpośrednim wprowadzeniu interesującego użytkownika równania. Modelowanie manipulatora z elastycznościami było bardzo intuicyjne, ponieważ model dynamiki, a także algorytm statycznej regulacji PD mógł zostać zaimplementowany w oparciu o teorię bezpośrednio zaczerpniętą z książek. Nie potrzebne było także wprowadzanie dodatkowej struktury elementów składowych modelu (np. sprzężenia zwrotnego). Możliwości wyświetlania i zapisu uzyskanych wyników są bardzo szerokie. Dodatkowo sama aplikacja jest zbudowana uniwersalnie, na stosunkowo wysokim poziomie abstrakcji, co pozwala na wykorzystanie tych samych funkcji do obiektów tekstu, równań czy rysunku. Sporym minusem aplikacji Mathematica jest jej składnia, której znajomość jest niezbędna, jako że potencjał środowiska skupia się właśnie w tej części. Poznanie struktury zapisu

równań może sprawiać kłopoty początkującym, ale po pewnym obyciu pozwala uzyskać zaskakujące wyniki w krótkim czasie.

Porównywanie obu aplikacji pod kątem, która z nich jest lepsza jest niemożliwe, ponieważ oba środowiska zapewniają możliwości satysfakcjonujące większość użytkowników. Bardziej na miejscu wydaje się ocena ich przydatności dla konkretnych zastosowań. MATLAB wydaje się bardziej praktycznym narzędziem, szczególnie z uwagi na szereg narzędzi do projektowania (szybkie prototypowanie, możliwość interakcji z warstwą sprzętową np. National Instruments). Z kolei Mathematica częściej wykorzystywana jest do zastosowań teoretycznych i stała się standardem w środowiskach naukowych, gdzie często używa się jej do przedstawienia nowych teorii.

Bibliografia

[1] DLR. DLR light-weight robot (LWR). http://www.dlr.de/rm/desktopdefault.aspx/tabid-3803/6175_read-8963/.

[2] A. D. Luca. Part 1: Modeling and control of robots with elastic joints. http://www.dis.uniroma1.it/~deluca/ADL_Dottorato2011_Part1_slides.pdf, 2011.

[3] J. B. Michael Thummel, Martin Otter. Control of robots with elastic joints based on automatic generation of inverse dynamics models. Raport instytutowy, Institut fur Robotik und Mechatronik DLR Oberpfaffenhofen, Wessling, Germany, 2011.

[4] S. Moberg. *Modeling and Control of Flexible Manipulators*. Praca doktorska, Department of Electrical Engineering Linköping University, Sweden, 2010.

[5] K. Tchon, A. Mazur. Algorytm całkowania wstecznego dla robota o elastycznych przegubach. Raport instytutowy, Instytut Cybernetyki Technicznej PWR, 1997.

13. Sztywny manipulator mobilny

Zuzanna Pietrowska, Tomasz Płatek

W bieżącym rozdziale zajmiemy się zagadnieniem modelowania manipulatora mobilnego, rozumianego jako manipulator sztywny zamontowany na platformie mobilnej. Efektem takiej konfiguracji układu jest zwiększenie przestrzeni roboczej manipulatora, jednak skutkuje to znacznym utrudnieniem problemu sterowania. Komplikacje te wynikają między innymi z obecności sprzężeń dynamicznych pomiędzy składowymi systemu (np. ruch manipulatora może powodować ruch całej platformy) oraz z konieczności uwzględnienia ograniczeń jednego podsystemu podczas sterowania całym układem.

O modelowanym układzie zakładamy zazwyczaj, że ruch platformy odbywa się bez poślizgów kół (wzdłużnego i/lub bocznego), oraz że porusza się ona po płaszczyźnie ekwipotencjalnej. Ponadto zakładamy, że modelowane koła nie podlegają deformacjom. Z założenia o braku poślizgu kół mogą wynikać ograniczenia nieholonomiczne. Ograniczenia tego typu nie zmniejszają przestrzeni stanu układu, mogą jednak utrudniać osiągnięcie pewnych konfiguracji.

Ze względu na rodzaj ograniczeń nałożonych na wchodzące w skład manipulatora mobilnego podsystemy, możemy dokonać następującego podziału [4]:
— typ (h; h) – holonomiczny manipulator na holonomicznej platformie,
— typ (h; nh) – holonomiczna platforma z nieholonomicznym manipulatorem,
— typ (nh; h) – nieholonomiczna platforma z holonomicznym manipulatorem,
— typ (nh; nh) – nieholonomiczny manipulator na nieholonomicznej platformie.
W dalszej części pracy rozważane będą układy typu (nh; h) oraz (nh; nh).

13.1. Kinematyka i równania ruchu manipulatora mobilnego

Kinematyką manipulatora mobilnego nazywamy funkcję, która wyznacza położenie i orientację efektora manipulatora względem przyjętego bazowego układu współrzędnych w zależności od położenia platformy oraz przegubów manipulatora [7]. Przez równania ruchu manipulatora mobilnego rozumiemy jego model dynamiki wraz z równaniami ograniczeń nieholonomicznych nałożonych na układ (jeśli występują) [7]. Równania ograniczeń nieholonomiczych wyrażone są odpowiednio w postaci Pfaffa (gdy interesuje nas opis obiektu we współrzędnych uogólnionych) lub w postaci bezdryfowego układu sterowania (gdy interesuje nas opis obiektu wyrażony we współrzędnych pomocniczych).

13.2. Dynamika manipulatora mobilnego

W niniejszej pracy zdecydowano się na opisanie dynamiki manipulatora mobilnego bazując na formaliźmie Lagrange'a [7]. Rozważany jest układ mechaniczny, którego zachowanie jest opisane przez współrzędne uogólnione $q \in R^n$ i prędkości $\dot{q} \in R^n$, spełniające l ($l < n$) niezależnych ograniczeń fazowych, mających postać Pfaffa

$$A(q)\dot{q} = 0. \tag{13.1}$$

W celu uwzględnienia ograniczeń nieholonomicznych postaci (13.1) w modelu układu należy zastosować zasadę d'Alemberta, mówiącą, że

Twierdzenie 1. *Siły więzów F_w, wymuszające spełnienie ograniczeń nieholonomicznych, nie wykonują pracy na dopuszczalnych trajektoriach układu.*

Powyższą zasadę można wyrazić równaniem

$$F_w^T dq = 0. \tag{13.2}$$

Równanie ograniczeń 13.1 można wyrazić w równoważnej postaci

$$A(q)dq = 0. \tag{13.3}$$

Z równań (13.2) oraz (13.3) wynika, że

$$F_w^T = \lambda^T A(q), \tag{13.4}$$

gdzie $\lambda \in R^l$ jest wektorem mnożników Lagrange'a.

W celu opisania dynamiki układu z ograniczeniami (13.1) należy zdefiniowaniać lagranżian układu swobodngo (bez ograniczeń) postaci

$$L(q, \dot{q}) = E_k(q, \dot{q}) - E_p(q), \tag{13.5}$$

gdzie $E_k(q, \dot{q}) = \frac{1}{2}\dot{q}^T Q(q)\dot{q}$ jest energią kinetyczną układu swobodnego, $Q(q)$ to symetryczna, dodatnio określona macierz inercji, zaś $E_p(q)$ to energia potencjalna układu. Wówczas równania ruchu układu przyjmują postać

$$\frac{d}{dt}\frac{\partial L(q,\dot{q})}{\partial \dot{q}} - \frac{\partial L(q,\dot{q})}{\partial q} = F, \tag{13.6}$$

gdzie wektor F symbolizuje uogólnione siły niepotencjalne działające na układ, w tym siły więzów F_w. Przy założeniu, że na układ działają jedynie siły wymuszające spełnienie ograniczeń (13.1)

$$\frac{d}{dt}\frac{\partial L(q,\dot{q})}{\partial \dot{q}} - \frac{\partial L(q,\dot{q})}{\partial q} = F_w, \tag{13.7}$$

co po podstawieniu zależności (13.4) daje

$$\frac{d}{dt}\frac{\partial L(q,\dot{q})}{\partial \dot{q}} - \frac{\partial L(q,\dot{q})}{\partial q} = A^T(q)\lambda. \tag{13.8}$$

Ostatecznie, model dynamiki obiektu wyrażony we współrzędnych uogólnionych, uwzględniający siły więzów nieholonomicznych oraz siły sterujące, opisywany jest równaniem

$$\frac{d}{dt}\frac{\partial L(q,\dot{q})}{\partial \dot{q}} - \frac{\partial L(q,\dot{q})}{\partial q} = A^T(q)\lambda + Bu, \tag{13.9}$$

gdzie u to m-wymiarowy wektor sił sterujących, zaś B to macierz rozmiaru $n \times m$, wskazująca sposób, w jaki siły sterujące wpływają na stan układu.

13.2.1. Model dynamiki we współrzędnych uogólnionych

Oznaczmy wektor współrzędnych uogólnionych manipulatora mobilnego przez $q = (q_m^T, q_r^T)^T$, gdzie q_m jest wektorem współrzędnych platformy mobilnej

$$q_m = (x, y, \theta, \phi_1, ..., \phi_k, \beta_1, ...\beta_k)^T, \tag{13.10}$$

gdzie współrzędne x i y opisują położenie środka ciężkości platformy względem układu nieruchomego, θ opisuje orientację platformy względem układu nieruchomego, k określa liczbę jej kół, zaś ϕ_i oraz β_i określają odpowiednio kąt obrotu oraz kąt skrętu i-tego koła. Wektor q_r jest wektorem współrzędnych przegubowych manipulatora

$$q_r = (\theta_1, ..., \theta_p)^T.$$

Załóżmy, że układ porusza się po płaskiej powierzchni prostopadłej do wektora sił grawitacji oraz że koła platformy są jednorodne, dzięki czemu energia potencjalna platformy jest stała i nie wpływa na analizowane równania ruchu. Wówczas funkcja Lagrange'a (13.5) dla platformy mobilnej z umieszczonym na niej manipulatorem ma postać

$$L(q, \dot{q}) = E_{k_m}(q_m, \dot{q}_m) + E_{k_r}(q, \dot{q}) - E_{p_r}(q), \tag{13.11}$$

gdzie $E_{k_m}(q_m, \dot{q}_m) = \frac{1}{2}\dot{q}_m^T Q_m(q_m)\dot{q}_m$– energia kinetyczna platformy mobilnej, $E_{k_r}(q, \dot{q}) = \frac{1}{2}\dot{q}^T Q_r(q)\dot{q}$ – energia kinetyczna manipulatora, $E_{p_r}(q)$ – energia potencjalna manipulatora.

Model manipulatora mobilnego (nh,h) we współrzędnych uogólnionych

Po zróżniczkowaniu funkcji Lagrange'a (13.11) zgodnie z równaniem (13.9) otrzymujemy równania dynamiki manipulatora mobilnego w postaci

$$Q_r(q)\ddot{q} + Q_m(q_m)\ddot{q}_m + C_r(q, \dot{q})\dot{q} + C_m(q_m, \dot{q}_m)\dot{q}_m + D(q_r) = A^T(q)\lambda + B(q)u, \tag{13.12}$$

lub w bardziej szczegółowej postaci

$$\begin{bmatrix} Q_{r11} & Q_{r12} \\ Q_{r21} & Q_{r22} \end{bmatrix} \begin{pmatrix} \ddot{q}_m \\ \ddot{q}_r \end{pmatrix} + \begin{bmatrix} Q_m & 0 \\ 0 & 0 \end{bmatrix} \begin{pmatrix} \ddot{q}_m \\ \ddot{q}_r \end{pmatrix} + \begin{bmatrix} C_{r11} & C_{r12} \\ C_{r21} & C_{r22} \end{bmatrix} \begin{pmatrix} \dot{q}_m \\ \dot{q}_r \end{pmatrix} \begin{bmatrix} C_m & 0 \\ 0 & 0 \end{bmatrix} \begin{pmatrix} \dot{q}_m \\ \dot{q}_r \end{pmatrix} + \begin{pmatrix} 0 \\ D \end{pmatrix} = \begin{pmatrix} A^T\lambda \\ 0 \end{pmatrix} + \begin{bmatrix} B & 0 \\ 0 & I \end{bmatrix} \begin{pmatrix} u_m \\ u_r \end{pmatrix}, \tag{13.13}$$

gdzie:
- Q_r – macierz bezwładności manipulatora,
- C_r – macierz oddziaływań Coriolisa i odśrodkowych manipulatora,
- Q_m – macierz bezwładności platformy mobilnej,
- C_m – macierz oddziaływań Coriolisa i odśrodkowych platformy jezdnej,
- $D = \frac{\partial E_{p_r}}{\partial q}$ – wektor grawitacji manipulatora,
- A – macierz Pfaffa,
- B – macierz rozmiaru $n \times m$, wskazująca sposób w jaki siły sterujące wpływają na stan układu,
- u_m – wektor sterowań platformy mobilnej,
- u_r – wektor sterowań manipulatora.

Model manipulatora mobilnego (nh,nh) we współrzędnych uogólnionych

Uwzględnienie ograniczeń nieholonomicznych w manipulatorze poprzez zastosowanie zasady d'Alemberta prowadzi do następujących równań dynamiki modelujących manipulator mobilny

$$Q_r(q)\ddot{q} + Q_m(q_m)\ddot{q}_m + C_r(q,\dot{q})\dot{q} + C_m(q_m,\dot{q}_m)\dot{q}_m + D(q_r) = A_{11}^T(q_m)\lambda_1 + A_{21}^T(q_r)\lambda_2 + B(q)u, \quad (13.14)$$

gdzie:
— Q_r – macierz bezwładności manipulatora,
— C_r – macierz oddziaływań Coriolisa i odśrodkowych manipulatora,
— Q_m – macierz bezwładności platformy mobilnej,
— C_m – macierz oddziaływań Coriolisa i odśrodkowych platformy jezdnej,
— $D = \frac{\partial E_{p_r}}{\partial q}$ – wektor grawitacji manipulatora,
— u_m – wektor sterowań platformy mobilnej,
— u_r – wektor sterowań manipulatora,
— A_{11} – macierz ograniczeń Pfaffa dla platformy mobilnej,
— λ_1 – wektor mnożników Lagrange'a dla platformy mobilnej,
— A_{21} – macierz ograniczeń Pfaffa dla części manipulacyjnej,
— λ_2 – wektor mnożników Lagrange'a dla części manipulacyjnej,
— $B(q)$ – macierz wejściowa,
— u – wektor sterowań.

Na podstawie postaci powyższych równań można stwierdzić, że w tym przypadku ograniczenia są nałożone na każdy z podsystemów osobno. Macierze $A_{11}(q_m)$, $A_{21}(q_r)$ oraz $B(q)$ mają postać

$$A_{11} = \begin{bmatrix} A_1^T(q_m) \\ 0 \end{bmatrix}, \quad A_{21} = \begin{bmatrix} 0 \\ A_2^T(q_r) \end{bmatrix}, \quad B(q) = \begin{bmatrix} B_{11}(q_m) & 0 \\ 0 & B_{22}(q) \end{bmatrix}. \quad (13.15)$$

Podmacierze macierzy wejścia B określają, które współrzędne są bezpośrednio napędzane przez silniki.

13.2.2. Model dynamiki we współrzędnych pomocniczych

Modele przedstawione w rozdziale 13.2.1 wymagają znajomości mnożników Lagrange'a. Jednak wiedza ta nie jest potrzebna do sterowania omawianym obiektem. Z postaci ograniczeń (13.1) wynika, że istnieją pewne pomocnicze prędkości η spełniające warunek

$$\dot{q} = G(q)\eta, \quad (13.16)$$

gdzie G jest macierzą, dla której $A(q)G(q) = 0$. Zastosowanie tak zdefiniowanych współrzędnych pomocniczych pozwala na wyeliminowanie mnożników Lagrange'a z wyprowadzonych powyżej modeli.

Model manipulatora mobilnego (nh,h) we współrzędnych pomocniczych

Równania ograniczeń manipulatora mobilnego typu (nh,h) przyjmują postać

$$A(q_m)\dot{q}_m = 0. \quad (13.17)$$

Zapisując te ograniczenia w formie bezdryfowego układu sterowania otrzymujemy

$$\dot{q}_m = G(q_m)\eta. \quad (13.18)$$

Mnożąc lewostronnie pierwszy wiersz macierzowych równań (13.13) przez $G^T(q_m)$ otrzymujemy model dynamiki w postaci

$$\begin{bmatrix} G^T(Q_{r11}+Q_{m11})G & G^TQ_{r12} \\ Q_{r21}G & Q_{r22} \end{bmatrix} \begin{pmatrix} \dot{\eta} \\ \ddot{q}_r \end{pmatrix} + \\ + \begin{bmatrix} G^T(C_{r11}+C_{m11})G + G^T(Q_{r11}+Q_{m11})\dot{G} & G^TC_{r12} \\ Q_{r21}\dot{G}+C_{r21}G & C_{r22} \end{bmatrix} \begin{pmatrix} \eta \\ \dot{q}_r \end{pmatrix} + \\ + \begin{pmatrix} 0 \\ D \end{pmatrix} = \begin{bmatrix} G^TB & 0 \\ 0 & I \end{bmatrix} \begin{pmatrix} u_m \\ u_r \end{pmatrix}. \quad (13.19)$$

Model manipulatora mobilnego (nh,nh) we współrzędnych pomocniczych

Ograniczenia nieholonomiczne w przypadku platformy mobilnej typu (nh,nh) mają postać

$$A_1(q_m)\dot{q}_m = 0, A_2(q_r)\dot{q}_r = 0, \quad (13.20)$$

gdzie A_1 jest macierzą ograniczeń Pfaffa dla platformy, natomiast A_2 jest macierzą ograniczeń Pfaffa dla manipulatora. Ograniczenia te można wyrazić w postaci bezdryfowego układu sterowania

$$\begin{pmatrix} \dot{q}_m \\ \dot{q}_r \end{pmatrix} = \begin{bmatrix} G_1 & 0 \\ 0 & G_2 \end{bmatrix} \begin{pmatrix} \eta \\ \tau \end{pmatrix} = G\xi, \xi = \begin{pmatrix} \eta \\ \tau \end{pmatrix}, \quad (13.21)$$

gdzie ξ oznacza wektor prędkości pomocniczych. Na podstawie równań (13.21) oraz (13.14) otrzymujemy

$$Q^*\dot{\xi} + C^*\xi + D^* = B^*u, \quad (13.22)$$

gdzie:
— $D^*(q) = G^TD$,
— $Q^* = G^T(q)Q(q)G(q)$,
— $C^* = G^T(q)(C(q, G(q)\eta)G(q) + Q(q)\dot{G}(q, G(q)\eta))$,
— $Q(q) = Q_r(q) + Q_m(q_m)$,
— $C(q, \dot{q}) = C_r(q, \dot{q}) + C_m(q_m, \dot{q}_m)$,
— $B^* = G^T(q)B(q)$.

13.2.3. Energia kinetyczna i potencjalna układu

Z rozważań uzyskanych w poprzednim podrozdziale wynika, że do wyprowadzenia modelu dynamiki potrzebna jest znajomość energii potencjalnej i kinetycznej układu. Z przyjętego założenia o ekwipotencjalnej płaszczyźnie ruchu platformy wynika, że jej energia potencjalna jest stała. Należy zatem wyznaczyć:
— energię kinetyczną platformy mobilnej (skrzyni oraz kół),
— energię potencjalną manipulatora,
— energię kinetyczną manipulatora.

Energia kinetyczna platformy mobilnej

Energia kinetyczna skrzyni wózka względem układu bazowego wyraża się zależnością [1]

$$E_{kp} = \frac{1}{2}I_z\dot{\theta} + \frac{1}{2}M_p(\dot{x}^2 + \dot{y}^2) + P_1(\dot{y}\cos\theta - \dot{x}\sin\theta)\dot{\theta} - P_2(\dot{y}\sin\theta + \dot{x}\cos\theta)\dot{\theta},$$

gdzie I_z – moment bezwładności ciała względem osi Z układu lokalnego, M_p – masa ciała, P_1,P_2 – momenty rzędu pierwszego ciała w lokalnym układzie współrzędnych ($P_1 = P_2 = 0$, jeżeli początek lokalnego układu współrzędnych znajduje się w środku ciężkości).

Pierwszym krokiem przy wyznaczaniu energii kinetycznej koła jest wyznaczenie transformacji układu koła w układ bazowy postaci

$$A_0^{k_i} = A_0^P Trans(z,-e)Rot(z,\alpha_i)Trans(x,l_i)Rot(z,\beta_i) \\ Trans(x,d_i)Rot(z,\gamma_i)Rot(x,\varphi_i), \quad (13.23)$$

gdzie:
— A_0^P – transformacja układu lokalnego poruszającego się wraz z robotem w układ bazowy,
— $\alpha_i, \beta_i, \gamma_i$ – kąty opisujące geometrię i-tego koła względem układu odniesienia związanego ze skrzynią robota,
— l_i, d_i – parametry opisujące umiejscowienie i-tego koła względem układu odniesienia związanego ze skrzynią robota.

Energia kinetyczna pojedynczego koła wyraża się wzorem [1]

$$E_{Ki} = \frac{1}{2}I_{xxi}\dot{\phi}_i^2 + \frac{1}{2}I_{zzi}\dot{\Delta}_i^2 + \frac{1}{2}M_{Ki}\{\dot{x}^2 + \dot{y}^2 + d_i^2\dot{\Delta}_i^2 + l_i^2\dot{\theta}^2 + 2l_i d_i\dot{\Delta}_i\theta_i cos\beta_i + \\ + 2d_i\dot{\Delta}_i\left[\dot{y}cos(\alpha_i+\beta_i+\theta) - \dot{x}sin(\alpha_i+\beta_i+\theta)\right] + 2l_i\dot{\theta}\left[\dot{y}cos(\alpha_i+\theta)\dot{x}sin(\alpha_i+\theta)\right]\},$$

gdzie:
— $\alpha_i, \beta_i, l_i, d_i, \varphi_i, \theta$ – parametry transformacji,
— $Ixx_i = \frac{1}{2}MR^2$ – moment bezwładności liczony względem osi X,
— $Izz_i = \frac{1}{12}M\omega^2 + \frac{1}{4}MR^2$ – moment bezwładności liczony względem osi Z,
— M – masa koła,
— R – promień koła,
— ω – grubość koła,
— $\Delta_i = \beta_i + \theta$.

Energia kinetyczna manipulatora na platformie mobilnej

W celu wyznaczenia energii kinetycznej manipulatora na początku należy wyznaczyć transformacje kinematyczne jego przegubów [3]. Transformacja opisująca położenie i orientację podstawy manipulatora względem bazowego układu odniesienia ma postać

$$A_0^b(x,y,\theta) = Trans(X,x)Trans(Y,y)Rot(Z,\theta)Trans(X,a),$$

gdzie x, y, θ – położenie i orientacja platformy mobilnej, a – położenie podstawy manipulatora względem środka platformy. Następnym krokiem jest wyznaczenie transformacji poszczególnych ogniw względem podstawy manipulatora z wykorzystaniem algorytmu Denevita-Hartenberga. Transformacje opisujące położenie poszczególnych ogniw względem bazowego układu odniesienia mają postać

$$T_0^i(x,y,\theta,\theta_1,...,\theta_i) = A_0^b(x,y,\theta)A_b^1(\theta_1)\cdot ... \cdot A_{i-1}^i(\theta_i), \quad (13.24)$$

gdzie θ_i oznacza współrzędne przegubowe kolejnych stopni swobody manipulatora. Do wyznaczenia energii kinetycznej ogniw manipulatora potrzebne są jeszcze macierze pseudoinercji ogniw

$$J_i = \begin{bmatrix} \int\limits_{l_i} x_i^2 dm & \int\limits_{l_i} x_i y_i dm & \int\limits_{l_i} x_i z_i dm & \int\limits_{l_i} x_i dm \\ \int\limits_{l_i} y_i x_i dm & \int\limits_{l_i} y_i^2 dm & \int\limits_{l_i} y_i z_i dm & \int\limits_{l_i} y_i dm \\ \int\limits_{l_i} z_i x_i dm & \int\limits_{l_i} z_i y_i dm & \int\limits_{l_i} z_i^2 dm & \int\limits_{l_i} z_i dm \\ \int\limits_{l_i} x_i dm & \int\limits_{l_i} y_i dm & \int\limits_{l_i} z_i dm & \int\limits_{l_i} dm \end{bmatrix}. \tag{13.25}$$

Energię kinetyczną i-tego ogniwa można wyznaczyć z zależności

$$K_i = \frac{1}{2} tr\left(\dot{T}_0^i J_i \dot{T}_0^{iT}\right).$$

Całkowita energia kinetyczna manipulatora jest sumą energii kinetycznych pojedynczych ogniw.

Energia potencjalna manipulatora na platformie mobilnej

Energia potencjalna manipulatora jest sumą energii potencjalnych poszczególnych ogniw liczoną według wzoru

$$E_{p_r} = -\sum_i m_i \left\langle g, T_0^i R_i \right\rangle, \tag{13.26}$$

gdzie R_i opisuje położenie środka masy i-tego ogniwa we współrzędnych jednorodnych względem układu lokalnego stowarzyszonego z tym ogniwem manipulatora. Wektor grawitacji g wyrażony jest w układzie podstawowym.

Macierz bezwładności oraz macierz Coriolisa

Macierz bezwładności $Q_r(q)$ części manipulacyjnej obiektu można obliczyć korzystając ze wzoru

$$Q_{ij} = trS(i,j) = tr\sum_{k=1}^{n} \frac{\partial A_0^k}{\partial q_i} J_k \left(\frac{\partial A_0^k}{\partial q_j}\right)^T. \tag{13.27}$$

Macierz Coriolisa cześci manipulacyjnej $C_r(q)$ można obliczyć na podstawie $Q_r(q)$ ze wzoru

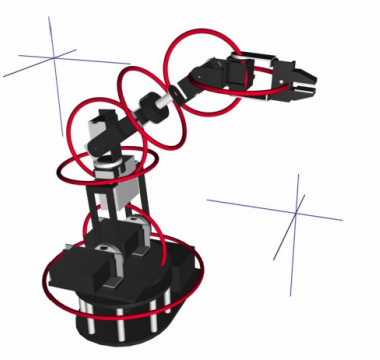
$$C^{ij} = \sum_k C_{kj}^i \dot{q}_k. \tag{13.28}$$

Wyrażenia C_{kj}^i to tak zwane symbole Christoffela I rodzaju wyznaczane w następujący sposób

$$C_{kj}^i = \frac{1}{2}\left(\frac{\partial Q^{ij}}{\partial q_k} + \frac{\partial Q^{ik}}{\partial q_j} - \frac{\partial Q^{jk}}{\partial q_i}\right).$$

13.3. Wykorzystanie programu Mathematica

Mathematica jest wygodnym narzędziem, które umożliwia proste wyprowadzanie modeli matematycznych nawet bardzo skomplikowanych manipulatorów mobilnych. Ponadto



Rysunek 13.1. CYTON ALPHA 7D 1G

Rysunek 13.2. Pioneer3-DX

pozwala ona na przeprowadzanie symulacji układów tego typu oraz implementację i testowanie algorytmów sterowania.

W tym rozdziale przedstawiony zostanie sposób w jaki można napisać skrypt, który automatycznie obliczy model matematyczny manipulatora mobilnego oraz zaprezentowane zostaną dwa wyprowadzone przy jego użyciu modele manipulatorów mobilnych typu (nh,h). Następnie opisane zostanie jak zrealizować symulację działania urządzenia, wykorzystując uzyskany w poprzednim kroku model.

13.3.1. Modelowane obiekty

Manipulator CYTHON ALPHA 7D 1G na platformie Pioneer 3-DX

Pierwszy z rozważanych obiektów to manipulator CYTON ALPHA 7D 1G (rysunek 13.1) zamontowany na platformie klasy $(2,0)$ Pioneer3-DX (rysunek 13.2). Schemat łańcucha kinematycznego rozważanego manipulatora przedstawiono na rysunku 13.3. Transformacja opisująca położenie i orientację układu znajdującego się u podstawy manipulatora względem układu podstawowego jest następująca:

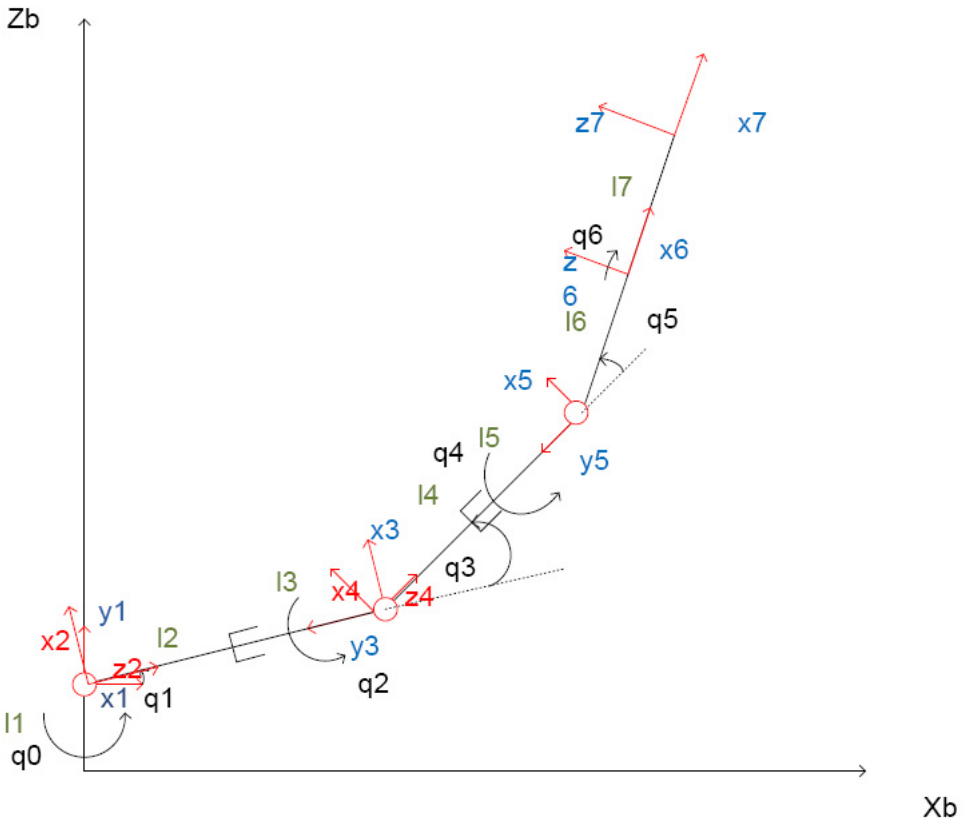
$$A_0^b = Tran(X,x)Trans(Y,y)Rot(Z,\theta)Trans(X,a) = A_0^p \cdot Trans(X,a),$$

gdzie:

$$A_0^p = Tran(X,x)Trans(Y,y)Rot(Z,\theta).$$

Transformacje opisujące położenie ogniw względem podstawy manipulatora można wyrazić w notacji Denavita-Hartenberga

$$\begin{aligned}
A_b^1(q_0) &= Rot(Z,q_0)Trans(z,l_1)Rot\left(X,\tfrac{\pi}{2}\right),\\
A_1^2(q_1) &= Rot\left(Z,q_1+\tfrac{\pi}{2}\right)Rot\left(X,\tfrac{\pi}{2}\right),\\
A_2^3(q_2) &= Rot(Z,q_2)Trans(z,l_2+l_3)Rot\left(X,-\tfrac{\pi}{2}\right),\\
A_3^4(q_3) &= Rot(Z,q_3)Rot\left(X,\tfrac{\pi}{2}\right),\\
A_4^5(q_4) &= Rot(Z,q_4)Trans(z,l_4+l_5)Rot\left(X,-\tfrac{\pi}{2}\right),\\
A_5^6(q_5) &= Rot\left(Z,q_5-\tfrac{\pi}{2}\right)Trans(x,l_6)Rot\left(X,-\tfrac{\pi}{2}\right),\\
A_6^7(q_6) &= Rot(Z,q_6)Trans(x,l_7).
\end{aligned} \tag{13.29}$$



Rysunek 13.3. Schemat kinematyki ramienia

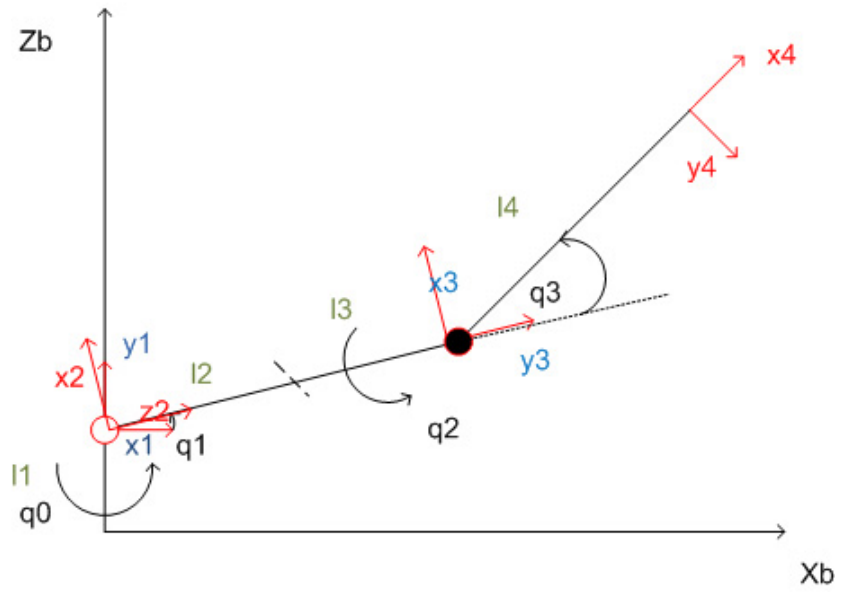
Ponieważ model tak zdefiniowanego manipulatora jest bardzo skomplikowany, poniżej przedstawiono rozważania dla manipulatora złożonego z czterech pierwszych ogniw. Zmodyfikowany schemat kinematyki manipulatora przedstawiono na rysunku 13.4.

Transformacje opisujące położenie układów lokalnych ogniw służące do wyznaczenia ich położenia względem podstawy manipulatora w notacji Denavita-Hartenberga są następujące

$$
\begin{aligned}
A_b^1(q_0) &= Rot(Z,q_0)Trans(z,l_1)Rot\left(X,\tfrac{\pi}{2}\right),\\
A_1^2(q_1) &= Rot\left(Z,q_1+\tfrac{\pi}{2}\right)Rot\left(X,\tfrac{\pi}{2}\right),\\
A_2^3(q_2) &= Rot(Z,q_2)Trans(z,l_2+l_3)Rot\left(X,\tfrac{\pi}{2}\right),\\
A_3^4(q_3) &= Rot\left(Z,-q_3+\tfrac{\pi}{2}\right)Trans(x,l_4).
\end{aligned}
\tag{13.30}
$$

Przyjęto, że ogniwa manipulatora są cylindryczne. Wobec tego dla układów stowarzyszonych z ogniwami przyjętych jak na rysunku 13.4 macierze pseudoinercji wynoszą odpowiednio:

$$
J_1 = \begin{bmatrix}
\frac{m_1 r^2}{4} & 0 & 0 & 0\\
0 & \frac{l_1^2 m_1}{3} & 0 & -\frac{l_1 m_1}{2}\\
0 & 0 & \frac{m_1 r^2}{4} & 0\\
0 & -\frac{l_1 m_1}{2} & 0 & m_1
\end{bmatrix},
\tag{13.31}
$$



Rysunek 13.4. Schemat kinematyki ramienia po uproszczeniu

$$J_2 = \begin{bmatrix} \frac{m_2 r^2}{4} & 0 & 0 & 0 \\ 0 & \frac{m_2 r^2}{4} & 0 & 0 \\ 0 & 0 & \frac{l_2^2 m2}{3} & \frac{l_2 m_2}{2} \\ 0 & 0 & \frac{l_2 m_2}{2} & m_2 \end{bmatrix}, \tag{13.32}$$

$$J_3 = \begin{bmatrix} \frac{m_3 r^2}{4} & 0 & 0 & 0 \\ 0 & \frac{l_3^2 m_3}{3} & 0 & -\frac{l_3 m_3}{2} \\ 0 & 0 & \frac{m_3 r^2}{4} & 0 \\ 0 & -\frac{l_3 m_3}{2} & 0 & m_3 \end{bmatrix}, \tag{13.33}$$

$$J_4 = \begin{bmatrix} \frac{l_4^2 m_4}{3} & 0 & 0 & -\frac{l_4 m_4}{2} \\ 0 & \frac{m_4 r^2}{4} & 0 & 0 \\ 0 & 0 & \frac{m_4 r^2}{4} & 0 \\ -\frac{l_4 m_4}{2} & 0 & 0 & m_4 \end{bmatrix}. \tag{13.34}$$

Manipulator RTR na monocyklu

Poprzedni obiekt opisany jest skomplikowanymi równaniami ruchu, w związku z tym nie udało się wykorzystać funkcji **NDSolve** do jego symulacji. Aby sprawdzić poprawność działania skrytpu wyprowadzono model manipulatora mobilnego RTR na platformie klasy $(2, 0)$. Dla tego obiektu rozwiązano równania ruchu funkcją **NDSolve** oraz przedstawiono uzysane wyniki na wykresach.

Model kinematyki rozważanego manipulatora przedstawiono w [3]. Transformacja opisująca położenie i orientację układu znajdującego się u podstawy manipulatora względem układu podstawowego jest następująca:

$$A_0^b = Tran(X, x)Trans(Y, y)Rot(Z, \theta)Trans(X, a) = A_0^p \cdot Trans(X, a).$$

Transformacje opisujące położenie ogniw względem podstawy manipulatora można wyrazić w notacji Denavita-Hartenberga

$$\begin{aligned} A_b^1(q_1) &= Rot(Z, q_1), \\ A_1^2(q_2) &= Trans(z, q_2)Trans(x, l_2)Rot\left(X, \tfrac{\pi}{2}\right), \\ A_2^3(q_3) &= Rot(Z, q_3)Trans(x, l_3). \end{aligned} \tag{13.35}$$

Przyjęte macierze pseudoinercji ogniw manipulatora (pierwsze ogniwo nieważkie, kolejne ogniwa to jednorodne cienkie pręty):

$$J_1 = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}, \tag{13.36}$$

$$J_2 = \begin{bmatrix} \frac{l_2^2 m_2}{3} & 0 & 0 & -\frac{l_2 m_2}{2} \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ -\frac{l_2 m_2}{2} & 0 & 0 & m_2 \end{bmatrix}, \tag{13.37}$$

$$J_3 = \begin{bmatrix} \frac{l_3^2 m_3}{3} & 0 & 0 & -\frac{l_3 m_3}{2} \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ -\frac{l_3 m_3}{2} & 0 & 0 & m_3 \end{bmatrix}. \tag{13.38}$$

13.3.2. Wyprowadzenie modeli matematycznych

Model we współrzędnych uogólnionych

Tworzenie w Mathematice skryptu obliczającego model matematyczny obiektu należy rozpocząć od zdefiniowania macierzy przekształceń elementarnych.

```
TransX[d_] :=
                    {{ 1, 0, 0, d},
                       { 0, 1, 0, 0},
                       { 0, 0, 1, 0},
                       { 0, 0,  0, 1}};
TransY[d_] :=
                    {{ 1, 0, 0, 0},
                       { 0, 1, 0, d},
                       { 0, 0, 1, 0},
                       { 0, 0,  0, 1}};
TransZ[d_] :=
                    {{ 1, 0, 0, 0},
                       { 0, 1, 0, 0},
                       { 0, 0, 1, d},
                       { 0, 0,  0, 1}};
RotZ[q_] :=
                    {{ Cos[q], -Sin[q], 0, 0},
                      { Sin[q],      Cos[q], 0, 0},
```

```
                    {      0,          0, 1, 0},
                    {      0,          0, 0, 1}};
RotX[q_] :=
                  {{ 1,       0,         0, 0},
                    { 0, Cos[q], -Sin[q], 0},
                    { 0, Sin[q],  Cos[q], 0},
                    { 0,       0,         0, 1}};
```

Następnie należy określić zmienne stanu rozważanego obiektu oraz wprowadzić parametry transformacji Denavita-Hartenberga, które zostaną wykorzystane przy wyznaczaniu modelu kinematyki manipulatora mobilnego. W tym miejscu należy zaznaczyć, że kinematyka manipulatora mobilnego zależy nie tylko od jego kątów przegubowych lecz również współrzędnych położenia i orientacji platformy jezdnej. Dla obiektu pierwszego opisanego w podrozdziale 13.3.1 mamy:

```
qplatformy[t_] = {x[t], y[t], theta[t], fi1[t], fi2[t]};
q[t_] = {q0[t], q1[t], q2[t], q3[t]};
(* RotZ TransZ TransX RotX *)
pomo = {x[t], y[t], theta[t]};
DHParameters = {{q0[t], l1, 0, Pi/2}, {q1[t] + Pi/2, 0, 0,
Pi/2}, {q2[t], l2 + l3, 0, Pi/2}, {-q3[t] + Pi/2, 0, l4, 0}};
```

Dla obiektu drugiego opisanego w podrozdziale 13.3.1:

```
qplatformy[t_] = {x[t], y[t], theta[t], fi1[t], fi2[t]};
q[t_] = {q1[t], q2[t], q3[t]};
(* RotZ TransZ TransX RotX *)
pomo = {x[t], y[t], theta[t]};
DHParameters = {{q1[t], 0, 0, 0}, {0, q2[t], l2, Pi/2},
 {q3[t], 0, l3, 0}};
```

Wykorzystując powyższe definicje można zapisać transformacje opisujące położenie układów lokalnych stowarzyszonych z ogniwami manipulatora oraz transformację opisującą położenie i orientację układu znajdującego się w podstawie manipulatora względem układu podstawowego.

```
(* Transformacje opisujące położenie ogniw*)
A0b = TransX[ pomo[[1]] ] .TransY[pomo[[2]]]
        .RotZ[pomo[[3]]].TransX[a];
A0b // MatrixForm
Ab1 = RotZ[ DHParameters[[1 ,1]]].TransZ[DHParameters[[1 ,2]]].
   TransX[ DHParameters[[1 ,3]]].RotX[DHParameters[[1 ,4]]];
Ab1 // MatrixForm
A12 = RotZ[DHParameters[[2 ,1]]].TransZ[DHParameters[[2 ,2]]] .
   TransX[DHParameters[[2 ,3]]].RotX[DHParameters[[2 ,4]]];
A12 // MatrixForm
A23 = RotZ[DHParameters[[3 ,1]]].TransZ[DHParameters[[3 ,2]]] .
   TransX[DHParameters[[3 ,3]]].RotX[DHParameters[[3 ,4]]];
A23 // MatrixForm
A34 = RotZ[DHParameters[[4 ,1]]].TransZ[DHParameters[[4 ,2]]] .
   TransX[DHParameters[[4 ,3]]].RotX[DHParameters[[4 ,4]]];
A34 // MatrixForm
```

W przypadku manipulatora RTR na monocyklu należy pominąć macierz A34 (ponieważ manipulator posiada w tym przypadku 3 stopnie swobody). Wynik realizacji tego kodu dla obiektu pierwszego

$$A0b = \begin{bmatrix} \text{Cos[theta[}t\text{]]} & -\text{Sin[theta[}t\text{]]} & 0 & a\text{Cos[theta[}t\text{]]} + x[t] \\ \text{Sin[theta[}t\text{]]} & \text{Cos[theta[}t\text{]]} & 0 & a\text{Sin[theta[}t\text{]]} + y[t] \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, \tag{13.39}$$

$$Ab1 = \begin{bmatrix} \text{Cos[q0[}t\text{]]} & 0 & \text{Sin[q0[}t\text{]]} & 0 \\ \text{Sin[q0[}t\text{]]} & 0 & -\text{Cos[q0[}t\text{]]} & 0 \\ 0 & 1 & 0 & \text{l1} \\ 0 & 0 & 0 & 1 \end{bmatrix}, \tag{13.40}$$

$$A12 = \begin{bmatrix} -\text{Sin[q1[}t\text{]]} & 0 & \text{Cos[q1[}t\text{]]} & 0 \\ \text{Cos[q1[}t\text{]]} & 0 & \text{Sin[q1[}t\text{]]} & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, \tag{13.41}$$

$$A23 = \begin{bmatrix} \text{Cos[q2[}t\text{]]} & 0 & \text{Sin[q2[}t\text{]]} & 0 \\ \text{Sin[q2[}t\text{]]} & 0 & -\text{Cos[q2[}t\text{]]} & 0 \\ 0 & 1 & 0 & \text{l2} + \text{l3} \\ 0 & 0 & 0 & 1 \end{bmatrix}, \tag{13.42}$$

$$A34 = \begin{bmatrix} \text{Sin[q3[}t\text{]]} & -\text{Cos[q3[}t\text{]]} & 0 & \text{l4Sin[q3[}t\text{]]} \\ \text{Cos[q3[}t\text{]]} & \text{Sin[q3[}t\text{]]} & 0 & \text{l4Cos[q3[}t\text{]]} \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}. \tag{13.43}$$

Transformacje wyrażające położenie ogniw manipulatora obliczamy wykorzystując transformacje (13.24) między układami lokalnymi stowarzyszonymi z poszczególnymi ogniwami.

```
(* Transformacje dla ogniw *)
T01a = A0b.Ab1;
(T01 = % // Simplify);
T01 // MatrixForm

T02a = T01.A12;
(T02 = % // Simplify);
T02 // MatrixForm

T03a = T02.A23;
(T03 = % // Simplify);
T03 // MatrixForm

T04a = T03.A34;
(T04 = % // Simplify);
T04 // MatrixForm
```

Dla obiektu drugiego nie obliczamy T04.

Aby obliczyć energię kinetyczną oraz potencjalną dowolnego manipulatora należy znać jego macierze pseudoinercji (13.25). Należy zatem wprowadzić do Mathematici równania (13.31)–(13.34) lub (13.36)–(13.38). Następnie można obliczyć macierz bezwładności części manipulacyjnej obiektu $Q(q)$, korzystając ze wzoru (13.27). Uzyskana macierz będzie zależna od zmiennych stanu opisujących zachowanie platformy oraz manipulatora. Implementację wzoru dla pierwszego obiektu przedstawiono poniżej.

```
QQ =  {{0, 0, 0, 0, 0, 0, 0, 0, 0},
       {0, 0, 0, 0, 0, 0, 0, 0, 0},
       {0, 0, 0, 0, 0, 0, 0, 0, 0},
           {0, 0, 0, 0, 0, 0, 0, 0, 0},
       {0, 0, 0, 0, 0, 0, 0, 0, 0},
       {0, 0, 0, 0, 0, 0, 0, 0, 0},
       {0, 0, 0, 0, 0, 0, 0, 0, 0},
       {0, 0, 0, 0, 0, 0, 0, 0, 0},
       {0, 0, 0, 0, 0, 0, 0, 0, 0}
      };


tmp = {x, y, theta, fi1, fi2, q0, q1, q2, q3};
  For[i = 1, i <= 9, i++,
            For[j = 1, j <= 9, j++,
  SS = D[T01, tmp[[i]][t]].J1.Transpose[D[T01, tmp[[j]][t]]]+
      D[T02, tmp[[i]][t]].J2.Transpose[D[T02, tmp[[j]][t]]] +
      D[T03, tmp[[i]][t]].J3.Transpose[D[T03, tmp[[j]][t]]] +
      D[T04, tmp[[i]][t]].J4.Transpose[D[T04, tmp[[j]][t]]];

    QQ[[i, j]] = Tr[SS];
    ];];
QQ = Simplify[QQ];

QQ // MatrixForm
```

Dla drugiego obiektu:

```
QQ =  {{0, 0, 0, 0, 0, 0, 0, 0},
       {0, 0, 0, 0, 0, 0, 0, 0},
       {0, 0, 0, 0, 0, 0, 0, 0},
       {0, 0, 0, 0, 0, 0, 0, 0},
       {0, 0, 0, 0, 0, 0, 0, 0},
       {0, 0, 0, 0, 0, 0, 0, 0},
       {0, 0, 0, 0, 0, 0, 0, 0},
       {0, 0, 0, 0, 0, 0, 0, 0}
      };


tmp = {x, y, theta, fi1, fi2, q1, q2, q3};
  For[i = 1, i <= 8, i++,
            For[j = 1, j <= 8, j++,
  SS = D[T01, tmp[[i]][t]].J1.Transpose[D[T01, tmp[[j]][t]]]+
      D[T02, tmp[[i]][t]].J2.Transpose[D[T02, tmp[[j]][t]]] +
      D[T03, tmp[[i]][t]].J3.Transpose[D[T03, tmp[[j]][t]]];
```

```
    QQ[[i, j]] = Tr[SS];
    ];];
QQ = Simplify[QQ];

QQ // MatrixForm
```

Jak widać zmiana wynikająca z mniejszej liczby zmiennych stanu jest minimalna. Wykorzystując komendę QQ == Transpose[QQ] można sprawdzić, że uzyskana macierz jest symetryczna.

Aby obliczyć macierz grawitacji $D(q)$ należy zdefiniować wektor grawitacji

$$g = (0, 0, -9.81, 0)$$

oraz położenia środków ciężkości ogniw R_i, gdzie i to numer ogniwa. Następnie należy zaimplementować wzór na energię potencjalną manipulatora (13.26) oraz na sam wektor D w następujący sposób:

```
Ep1 = m1*(g.(T01.R1));
Ep2 = m2*(g.(T02.R2));
Ep3 = m3*(g.(T03.R3));
Ep4 = m4*(g.(T04.R4));

Ep = -(Ep1 + Ep2 + Ep3 + Ep4);
Epc = % // Simplify

(* Wektor D *)
DD = {D[Epc, q0[t]], D[Epc, q1[t]], D[Epc, q2[t]],
 D[Epc, q3[t]]};
Column[DD]
```

W energii potencjalnej obiektu drugiego nie ma czwartego ogniwa w związku z czym należy usunąć człon $Ep4$ oraz usunąć pochodną po q_0.

Macierz Coriolisa części manipulacyjnej $C(q)$ można obliczyć na podstawie $Q(q)$ implementując wprost wzór (13.28). Dla obiektu pierwszego:

```
CC = {{0, 0, 0, 0, 0, 0, 0, 0, 0},
      {0, 0, 0, 0, 0, 0, 0, 0, 0},
      {0, 0, 0, 0, 0, 0, 0, 0, 0},
      {0, 0, 0, 0, 0, 0, 0, 0, 0},
      {0, 0, 0, 0, 0, 0, 0, 0, 0},
      {0, 0, 0, 0, 0, 0, 0, 0, 0},
      {0, 0, 0, 0, 0, 0, 0, 0, 0},
      {0, 0, 0, 0, 0, 0, 0, 0, 0},
      {0, 0, 0, 0, 0, 0, 0, 0, 0}
     };
tmp = {x, y, theta, fi1, fi2, q0, q1, q2, q3};
    For[i = 1, i <= 9, i++,
          For[j = 1, j <= 9, j++,
     For[k = 1, k <= 9, k++,

      CC[[i, j]] =
       CC[[i, j]] +
```

```
        1/2*tmp[[k]]'[
          t]*(D[QQQ[[i, j]], tmp[[k]][t]] +
           D[QQQ[[i, k]], tmp[[j]][t]] -
           D[QQQ[[j, k]], tmp[[i]][t]])
        ] ;];];

CCC = Simplify[CC];

CCC // MatrixForm
```

Dla obiektu drugiego:

```
CC =  {{0, 0, 0, 0, 0, 0, 0, 0},
       {0, 0, 0, 0, 0, 0, 0, 0},
       {0, 0, 0, 0, 0, 0, 0, 0},
       {0, 0, 0, 0, 0, 0, 0, 0},
       {0, 0, 0, 0, 0, 0, 0, 0},
       {0, 0, 0, 0, 0, 0, 0, 0},
       {0, 0, 0, 0, 0, 0, 0, 0},
       {0, 0, 0, 0, 0, 0, 0, 0}
      };
tmp = {x, y, theta, fi1, fi2, q1, q2, q3};
    For[i = 1, i <= 8, i++,
          For[j = 1, j <= 8, j++,
     For[k = 1, k <= 8, k++,

      CC[[i, j]] =
       CC[[i, j]] +
        1/2*tmp[[k]]'[
          t]*(D[QQQ[[i, j]], tmp[[k]][t]] +
           D[QQQ[[i, k]], tmp[[j]][t]] -
           D[QQQ[[j, k]], tmp[[i]][t]])
        ] ;];];

CCC = Simplify[CC];

CCC // MatrixForm
```

Model monocykla we współrzędnych uogólnionych wprowadzono ręcznie na podstawie przeprowadzonych wcześniej obliczeń. Macierz Coriolisa C_m dla monocykla jest zerowa.

```
(* Macierz bezwladnosci monocykla *)
Qm = {{Mc, 0, 0, 0, 0, 0, 0, 0},
      { 0, Mc, 0, 0, 0, 0, 0, 0},
      { 0, 0, Ip, 0, 0, 0, 0, 0},
      { 0, 0, 0, Ixx, 0, 0, 0, 0},
      { 0, 0, 0, 0, Ixx, 0, 0, 0},
      { 0, 0, 0, 0, 0, 0, 0, 0},
      { 0, 0, 0, 0, 0, 0, 0, 0},
      { 0, 0, 0, 0, 0, 0, 0, 0},
      { 0, 0, 0, 0, 0, 0, 0, 0}
    };
(* Macierz Pfaffa monocykla *)
```

```
Aq = {{Sin[theta], -Cos[theta], 0, 0, 0},
      {Cos[theta], Sin[theta], L, 0, -R},
      {Cos[theta], Sin[theta], -L, -R, 0}
   };
(* Macierz Sterowań monocykla *)
BB =    {{ 0, 0},
         { 0, 0},
         { 0, 0},
         { 1, 0},
            { 0, 1}
   };
```

Na tym etapie skrypt zawiera już wszystkie elementy potrzebne do obliczenia równań manipulatora mobilnego we współrzędnych uogólnionych, czyli równań dynamiki (13.12) oraz kinematyki części nieholonomicznej. Model we współrzędnych uogólnionych nie jest jednak satysfakcjonujący z uwagi na występowanie w nim mnożników Lagranża λ. Część modelu związaną z obiektem nieholonomicznym jakim jest monocykl można przekształcić do współrzędnych pomocniczych [4]. Sposób realizacji tego zadania zostanie przedstawiony w następnym podrozdziale.

Model we współrzędnych pomocniczych

Macierz $G(q)$ bezdryfowego układu sterowania monocykla została wprowadzona w gotowej formie.

```
GGG =  {{ Cos[theta[t]], Cos[theta[t]]},
        { Sin[theta[t]], Sin[theta[t]]},
        { 1/L, -1/L},
        { 0, 2/R},
           { 2/R, 0}
   };
Gprim={{-Sin[theta[t]]*theta'[t], -Sin[theta[t]]*theta'[t]},
        { Cos[theta[t]]*theta'[t], Cos[theta[t]]*theta'[t]},
        { 0, 0},
        { 0, 0},
           { 0, 0}
   };
```

Teraz należy znaleźć postać macierzy bezwładności dla części manipulacyjnej, macierzy bezwładności dla monocykla oraz macierzy Coriolisa dla obu podsystemów składowych i macierzy sterowań. Realizuje się to poprzez lewostronne pomnożenie równań platformy mobilnej (pierwszego wiersza macierzowego równań (13.12)) przez macierz G^T. Implementację dla manipulatora posiadającego cztery stopnie swobody na monocyklu podano poniżej. W przypadku zmiany na obiekt posiadający manipulator o mniejszej liczbie stopni swobody należy odpowiednio zmodyfikować wymiary poszczególnych macierzy.

```
tmp1 = {{QQQ[[1, 1]], QQQ[[1, 2]], QQQ[[1, 3]], QQQ[[1, 4]],
    QQQ[[1, 5]]},
   {QQQ[[2, 1]], QQQ[[2, 2]], QQQ[[2, 3]],
           QQQ[[2, 4]], QQQ[[2, 5]]},
   {QQQ[[3, 1]], QQQ[[3, 2]], QQQ[[3, 3]],
           QQQ[[3, 4]], QQQ[[3, 5]]},
   {QQQ[[4, 1]], QQQ[[4, 2]], QQQ[[4, 3]],
```

```
            QQQ[[4, 4]], QQQ[[4, 5]]},
   {QQQ[[5, 1]], QQQ[[5, 2]], QQQ[[5, 3]],
            QQQ[[5, 4]], QQQ[[5, 5]]}
   };
tmp2 = Transpose[GGG].tmp1.GGG;
tmp3 = {{QQQ[[1, 6]], QQQ[[1, 7]], QQQ[[1, 8]], QQQ[[1, 9]]},
   {QQQ[[2, 6]], QQQ[[2, 7]], QQQ[[2, 8]], QQQ[[2, 9]]},
   {QQQ[[3, 6]], QQQ[[3, 7]], QQQ[[3, 8]], QQQ[[3, 9]]},
   {QQQ[[4, 6]], QQQ[[4, 7]], QQQ[[4, 8]], QQQ[[4, 9]]},
   {QQQ[[5, 6]], QQQ[[5, 7]], QQQ[[5, 8]], QQQ[[5, 9]]}
   };
tmp4 = Transpose[GGG].tmp3;
tmp5 = {{QQQ[[6, 1]], QQQ[[6, 2]], QQQ[[6, 3]], QQQ[[6, 4]],
    QQQ[[6, 5]]},
   {QQQ[[7, 1]], QQQ[[7, 2]], QQQ[[7, 3]],
               QQQ[[7, 4]], QQQ[[7, 5]]},
   {QQQ[[8, 1]], QQQ[[8, 2]], QQQ[[8, 3]],
               QQQ[[8, 4]], QQQ[[8, 5]]},
   {QQQ[[9, 1]], QQQ[[9, 2]], QQQ[[9, 3]],
               QQQ[[9, 4]], QQQ[[9, 5]]}
   };
tmp6 = tmp5.GGG;
tmp7 = {{QQQ[[6, 6]], QQQ[[6, 7]], QQQ[[6, 8]], QQQ[[6, 9]]},
   {QQQ[[7, 6]], QQQ[[7, 7]], QQQ[[7, 8]], QQQ[[7, 9]]},
   {QQQ[[8, 6]], QQQ[[8, 7]], QQQ[[8, 8]], QQQ[[8, 9]]},
   {QQQ[[9, 6]], QQQ[[9, 7]], QQQ[[9, 8]], QQQ[[9, 9]]}
   };
NowaQ = ArrayFlatten[{{tmp2, tmp4}, {tmp6, tmp7}}];

NowaQ // MatrixForm

NowaQm = Transpose[GGG].{{Mc, 0, 0, 0, 0},
                         { 0, Mc, 0, 0, 0},
                   { 0, 0, Ip, 0, 0},
                   { 0, 0, 0, Ixx, 0},
                   { 0, 0, 0, 0, Ixx}
   }.GGG

ttmp1 = {{CCC[[1, 1]], CCC[[1, 2]], CCC[[1, 3]], CCC[[1, 4]],
    CCC[[1, 5]]},
   {CCC[[2, 1]], CCC[[2, 2]], CCC[[2, 3]],
               CCC[[2, 4]], CCC[[2, 5]]},
   {CCC[[3, 1]], CCC[[3, 2]], CCC[[3, 3]],
               CCC[[3, 4]], CCC[[3, 5]]},
   {CCC[[4, 1]], CCC[[4, 2]], CCC[[4, 3]],
               CCC[[4, 4]], CCC[[4, 5]]},
   {CCC[[5, 1]], CCC[[5, 2]], CCC[[5, 3]],
               CCC[[5, 4]], CCC[[5, 5]]}
   };
ttmp2 = Transpose[GGG].ttmp1.GGG + Transpose[GGG].tmp1.Gprim;
ttmp3 = {{CCC[[1, 6]], CCC[[1, 7]], CCC[[1, 8]], CCC[[1, 9]]},
   {CCC[[2, 6]], CCC[[2, 7]], CCC[[2, 8]], CCC[[2, 9]]},
```

```
60    {CCC[[3, 6]], CCC[[3, 7]], CCC[[3, 8]], CCC[[3, 9]]},
      {CCC[[4, 6]], CCC[[4, 7]], CCC[[4, 8]], CCC[[4, 9]]},
62    {CCC[[5, 6]], CCC[[5, 7]], CCC[[5, 8]], CCC[[5, 9]]}
      };
64 ttmp4 = Transpose[GGG].ttmp3;
   ttmp5 = {{CCC[[6, 1]], CCC[[6, 2]], CCC[[6, 3]], CCC[[6, 4]],
66     CCC[[6, 5]]},
      {CCC[[7, 1]], CCC[[7, 2]], CCC[[7, 3]],
68                 CCC[[7, 4]], CCC[[7, 5]]},
      {CCC[[8, 1]], CCC[[8, 2]], CCC[[8, 3]],
70                 CCC[[8, 4]], CCC[[8, 5]]},
      {CCC[[9, 1]], CCC[[9, 2]], CCC[[9, 3]],
72                 CCC[[9, 4]], CCC[[9, 5]]}
      };
74 ttmp6 = tmp5.Gprim + ttmp5.GGG;
   ttmp7 = {{CCC[[6, 6]], CCC[[6, 7]], CCC[[6, 8]], CCC[[6, 9]]},
76    {CCC[[7, 6]], CCC[[7, 7]], CCC[[7, 8]], CCC[[7, 9]]},
      {CCC[[8, 6]], CCC[[8, 7]], CCC[[8, 8]], CCC[[8, 9]]},
78    {CCC[[9, 6]], CCC[[9, 7]], CCC[[9, 8]], CCC[[9, 9]]}
      };
80
   NowaC = ArrayFlatten[{{ttmp2, ttmp4}, {ttmp6, ttmp7}}];
82 NowaC // MatrixForm

84 NowaCm = Transpose[GGG].{{Mc, 0, 0, 0, 0},
                            { 0, Mc, 0, 0, 0},
86                         { 0, 0, Ip, 0, 0},
                           { 0, 0, 0, Ixx, 0},
88                         { 0, 0, 0, 0, Ixx}
      }.Gprim
90 (* D bez zmian *)
   NowaB = Transpose[GGG].BB
92
   NowaQm // MatrixForm
94 zero = {{0, 0, 0, 0},
          {0, 0, 0, 0}};
96 zero2 = { { 0, 0},
            { 0, 0},
98          { 0, 0},
            { 0, 0}
100        };
   zero3 = {{0, 0, 0, 0},
102         {0, 0, 0, 0},
            {0, 0, 0, 0},
104         {0, 0, 0, 0}
       };
106 NowaQm = ArrayFlatten[{{NowaQm, zero}, {zero2, zero3}}];
   NowaQm // MatrixForm
108


110 NowaCm // MatrixForm
```

```
NowaCm = ArrayFlatten[{{NowaCm, zero}, {zero2, zero3}}];
NowaCm // MatrixForm

NowaD = Flatten[{0, 0, DD}]
```

Przedstawiony tok rozumowania pozwala wyprowadzić wszystkie macierze składowe modelu we współrzędnych pomocniczych (13.19). Ostatnim krokiem jaki należy zrealizować jest zebranie uzyskanych wyników i przekształcenie ich do postaci układu równań różniczkowych.

```
eta[t_] = {eta1[t], eta2[t]};
qr[t_] = {q0[t], q1[t], q2[t], q3[t]};
taum[t_] = {u1[t], u2[t]};
taur[t_] = {tau1[t], tau2[t], tau3[t], tau4[t]};
(*eta'[t_]={eta1'[t], eta2'[t]}
qr'[t_]={q0'[t], q1'[t], q2'[t], q3'[t], q4'[t], q5'[t], q6'[t]}*)
prawastrona[t_] = {2/R*u2[t], 2/R*u1[t], tau1[t], tau2[t],
 tau3[t], tau4[t]}
qq[t_] = Flatten[{eta[t], qr[t]}]
qq1[t_] = Flatten[{eta'[t], qr''[t]}]
qq2[t_] = Flatten[{eta[t], qr'[t]}]
qplatformy[t_] = {x[t], y[t], theta[t], fi1[t], fi2[t]};


model = Flatten[{
 Apply[Equal, Transpose[{qplatformy'[t], (GGG.eta[t])}], 1],
 Apply[Equal,
   Transpose[{NowaQ.qq1[t] + NowaQm.qq1[t] + NowaC.qq2[t] +
      NowaCm.qq2[t] + NowaD, prawastrona[t]}], 1]
  }];
model // MatrixForm
```

Dla drugiego obiektu, o ośmiu stopniach swobody należy wyeliminować q_0 oraz $tau4$.

Wyprowadzone modele stanowią załącznik w wersji elektronicznej. Równania uzyskane dla obiektu drugiego pokrywają się z modelem w [3].

13.3.3. Symulacje

Jak wspomniano wcześniej ramię Cyton na Pioneerze jest nawet w wersji uproszczonej zbyt skomplikowane aby możliwe było przeprowadzenie symulacji jego zachowania przy użyciu funkcji **NDSolve**. Dlatego badania symulacyjne przeprowadzone zostały jedynie na manipulatorze RTR na monocyklu.

W modelu matematycznym, który został wyprowadzony przez wykonanie kroków opisanych w poprzednim rozdziale przyjęto następujące parametry obiektu

```
m2 = 1; l2 = 1;
m3 = 1; l3 = 1;

L = 1; R = 1;
Mc = 1;
Ip = 1;
Ixx = 1;
a = 0;
```

oraz wybrano niezerowe warunki początkowe

```
init = {x[0] == 0, y[0] == 0, theta[0] == Pi/3, fi1[0] == 0,
    fi2[0] == 0,
    eta1[0] == 0, eta2[0] == 0,
    q1[0] == 0, q2[0] == 0, q3[0] == Pi/12,
    q1'[0] == 0, q2'[0] == 0, q3'[0] == 0
    };
u1[t] = 0;
u2[t] = 0;
tau1[t] = 0; tau2[t] = 0; tau3[t] = 0;
```

Równania ruchu manipulatora RTR na monocyklu zostały rozwiązane przy użyciu metody **NDSolve**

```
model = Simplify[model];
model // MatrixForm
sol = NDSolve[
  Flatten[{model, init}],
   {x, y, theta, fi1, fi2, eta1, eta2, q1, q2,
   q3}, {t, 0, 10}, Method -> RungeKutta, MaxSteps -> 100000]
```

Na koniec zwizualizowano wyniki przy użyciu zestawu poleceń

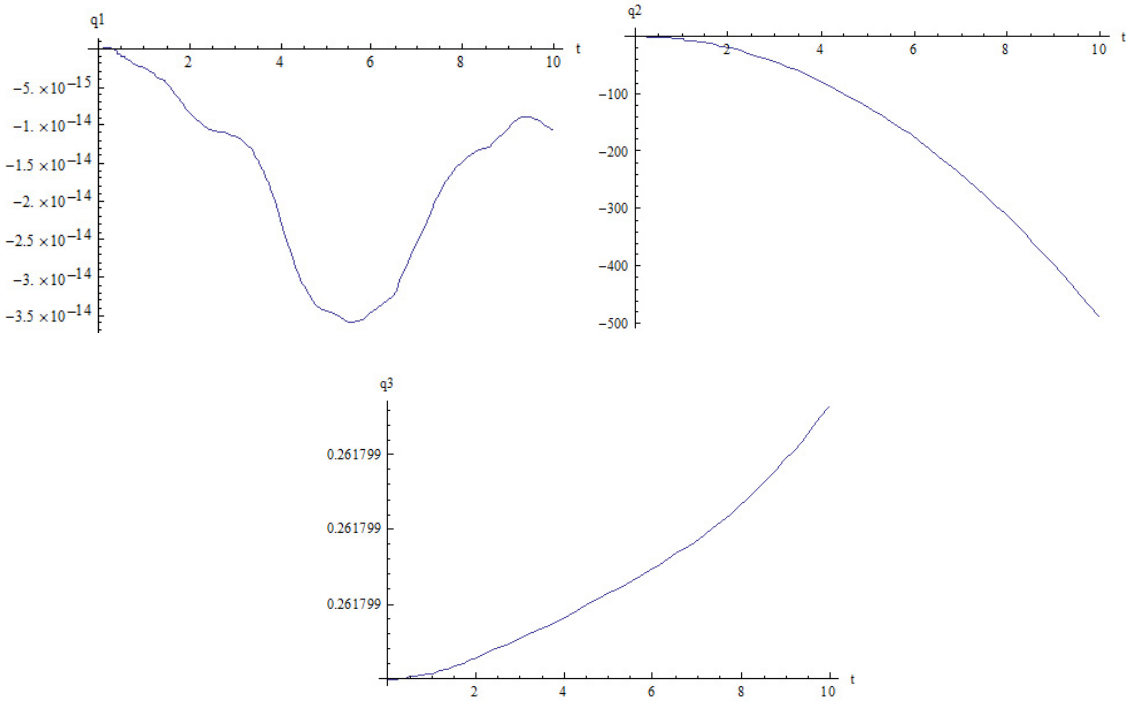
```
tmax = 10;
Plot[Evaluate[q1[t] /.sol], {t, 0, tmax}, PlotRange -> All,
 AxesLabel -> {"t", "q1"}]
Plot[Evaluate[q2[t] /.sol], {t, 0, tmax}, PlotRange -> All,
 AxesLabel -> {"t", "q2"}]
Plot[Evaluate[q3[t] /.sol], {t, 0, tmax},
 AxesLabel -> {"t", "q3"}]
ParametricPlot[{x[t] /. sol[[1]], y[t] /. sol[[1]]}, {t, 0, 10},
 AxesOrigin -> {0, 0}, PlotStyle -> Thick]
Plot[Evaluate[{fi1[t] /.sol, fi2[t] /. sol}], {t, 0, tmax},
 AxesLabel -> {"t", "fi1 , fi2"}]
```

Uzyskane wykresy przedstawione zostały na rysunkach 13.5–13.6. Na wykresie 13.6 zauważyć można, że działanie siły grawitacji na manipulator przy braku sterowania powoduje delikatny ruch platformy co jest zgodne z oczekiwaniami. Ponadto jak pokazuje wykres 13.5, gdy działa sama siła grawitacji cały obiekt spada (wskazuje na to zmiana wartości q_2).

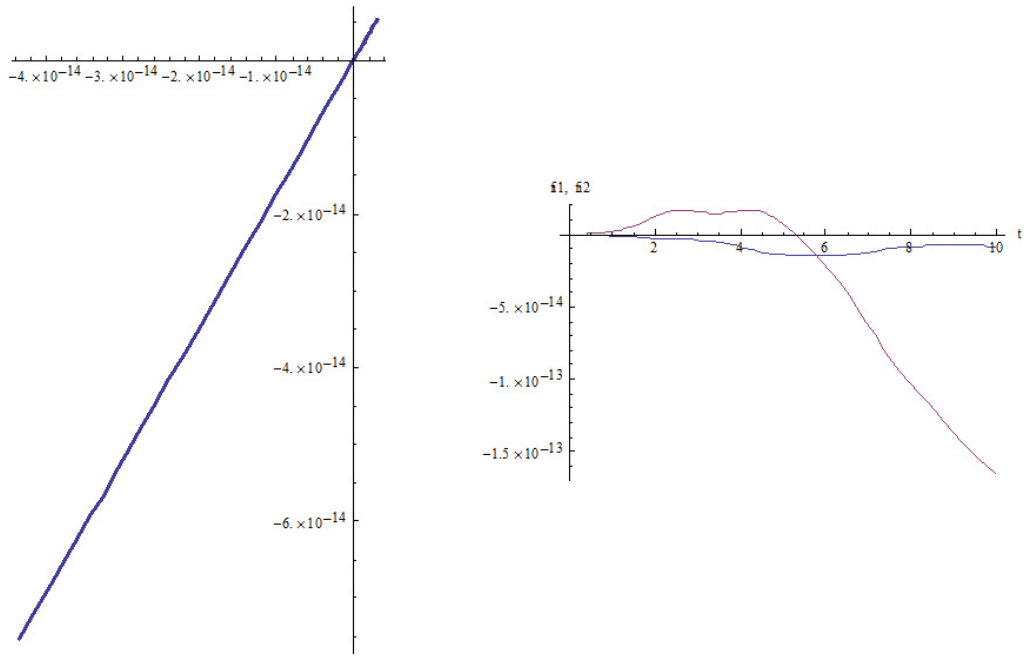
13.4. Wykorzystanie środowiska MRDS

Projekt tworzony w Microsoft Robotics Developer Studio R3 do pracy wymaga zainstalowania zintegrowanego środowiska programistycznego tej samej firmy [6]. Z tego względu wykorzystano program Microsoft Visual Studio 2008 Proffesional. Każdy projekt MRDS składa się z niezależnie działających usług (Decentralized Software Services) komunikujących się między sobą niezależnie. Każda z usług projektu odpowiada za inne aspekty symulacji, jest też zaimplementowana jako osobny projekt Microsoft Visual Studio.

Usługa *Enviroment* dodaje do symulacji jej elementy składowe. Oprócz niezbędnych elementów, takich jak podłoże, kamera oraz niebo, środowisko symulacyjne zawiera obiekty z którymi robot może wchodzić w interakcje: prostopadlościenne bloki oraz piłkę.



Rysunek 13.5. Wyniki symulacji



Rysunek 13.6. Wyniki symulacji

Głównym zadaniem usługi *Simulator* jest uruchomienie całego środowiska symulacyjnego. Najważniejszym plikiem tego projektu jest manifest. Określa on jakie usługi powinny zostać uruchomone oraz jakie są związki między nimi. Jest to też projekt, którego uruchomienie powoduje rozpoczęcie symulacji.

Ostatnią usługą wchodzącą w skład projektu jest *Robot*. Zawiera ona implementację manipulatora mobilnego oraz graficznego interfejsu użytkownika do sterowania manipulatorem. Działanie usług *Enviroment* oraz *Simulator* zostało dokładnie opisane w poprzedniej części raportu, natomiast zasada działania usługi *Robot* jest tematem następnych rozdziałów.

Zaimplemetowany model składa się z manipulatora i platformy mobilnej. Opis tworzenia obiektów tego typu znajduje się w [2]. Na podstawie tej publikacji zaimplementowano poszczególne obiekty wchodzące w skład projektu. W ramach projektu zaimplementowano model manipulatora CYTON ALPHA 7D 1G zamontowanego na platformie klasy $(2, 0)$ Pioneer3-DX.

13.4.1. Implementacja manipulatora mobilnego w środowisku MRDS

Podejście do modelowania manipulatora mobilnego w środowisku MRDS jest bardzo intuicyjne. Mianowicie, należy dodać do środowiska symulacyjnego platformę mobilną, a następnie w jej podwoziu umieścić manipulator. Takie podejście determinuje strukturę klas projektu.

Manipulator składa się z obiektów typu `SingleShapeSegmentEntity`. Reprezentują one jego ogniwa, które są łączone w ramię w klasie `Arm`. Klasą reprezentującą monocykl jest `MobileManipulator`, która zgodnie z nazwą dodaje również do niego wcześniej zaimplementowane ramię. Dodatkowo, zaimplementowany został graficzny interfejs użytkownika (klasa `RobotArmUI`). Szczegółowy diagram klas znajduje się w załączniku w wersji elektronicznej.

Klasa `SingleShapeSegmentEntity`

Klasą implementującą ogniwo manipulatora jest `SingleShapeSegmentEntity`. Dziedziczy ona po klasie `SingleShapeEntity`, a więc reprezetuje obiekt środowiska symulacyjnego składający się z jednego, prostego kształtu. Dodatkową funkcjonalnością jest wbudowany przegub (wydruk 13.1).

Wydruk 13.1. Właściwość `Joint` klasy `SingleShapeSegmentEntity`

```
private Joint _customJoint;

        [DataMember]
        public Joint CustomJoint
        {
            get { return _customJoint; }
            set { _customJoint = value; }
        }
```

Ze względu na fakt, że w przypdaku pracy w sieci połączenia definiowane przez klasę `Joint` nie są poprawnie przenoszone, w metodzie `Initialize` są one wyszukiwane, następnie poprawnie odnawiane (wydruk 13.2).

Wydruk 13.2. Metoda `Initialize` klasy `SingleShapeSegmentEntity`

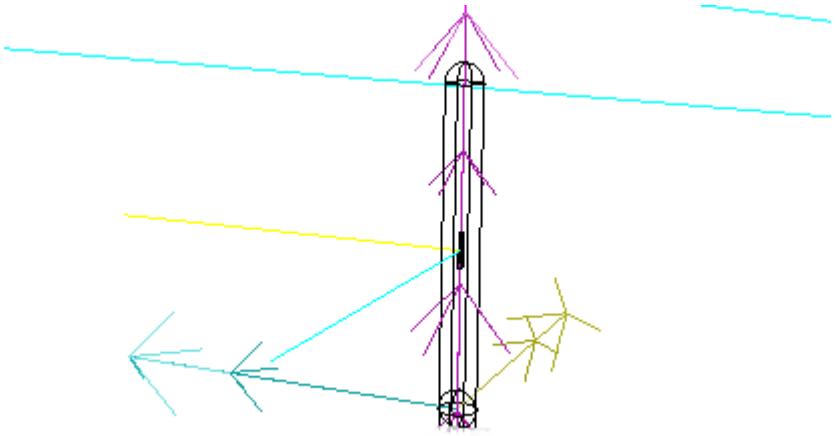
```
public override void Initialize(Microsoft.Xna.Framework.
        Graphics.GraphicsDevice device,
```

```
         PhysicsEngine physicsEngine)
        {
            base.Initialize(device, physicsEngine);

            if (_customJoint != null)
            {
                if (ParentJoint != null)
                    PhysicsEngine.DeleteJoint(
                        (PhysicsJoint)ParentJoint);

                if (_customJoint.State.Connectors[0].Entity
                                                    == null)
                    _customJoint.State.Connectors[0].Entity =
FindConnectedEntity(
        _customJoint.State.Connectors[0].EntityName,
         this);

                if (_customJoint.State.Connectors[1].Entity
                                                    == null)
                    _customJoint.State.Connectors[1].Entity =
FindConnectedEntity(
        _customJoint.State.Connectors[1].EntityName,
         this);

                ParentJoint = _customJoint;
                PhysicsEngine.InsertJoint(
                        (PhysicsJoint)ParentJoint);
            }
        }
```

Operacja wyszukiwania połączeń wykonywana jest w metodzie `FindConnectedEntity`, która zawiera rekurencyjne przeszukiwanie drzewa obiektów sceny przy pomocy metody `FindConnectedEntityHelper` (wydruk 13.3). Procedura ta została zaczerpnięta z pozycji [2], tam też znajduje się szczegółowy opis procedury odnawiania połączeń.

Wydruk 13.3. Metoda `VisualEntity` klasy `SingleShapeSegmentEntity`

```
VisualEntity FindConnectedEntity(string name,
                                 VisualEntity me)
        {
            while (me.Parent != null)
                me = me.Parent;
            return FindConnectedEntityHelper(name, me);
        }

        VisualEntity FindConnectedEntityHelper(string name,
                                 VisualEntity me)
        {
            if (me.State.Name == name)
                return me;
            foreach (VisualEntity child in me.Children)
            {
                VisualEntity result =
```



Rysunek 13.7. Instancja fizyczna ogniwa

```
17                  FindConnectedEntityHelper(name, child);
                    if (result != null)
19                      return result;
                }
21              return null;
            }
23          }
```

Wygląd gotowego ogniwa został zaprezentowany na rysunku 13.7.

Klasa `Arm`

Klasa `Arm` reprezentuje robotyczne ramię. Tak jak w przypadku klasy `SingleShapeSegmentEntity` dziedziczy ona po klasie `SingleShapeEntity`, obiekt tej klasy może być więc bezpośrednio dodany do środowiska symulacyjnego. Ciało tej klasy składa się m.in z listy parametrów tworzonego ramienia (wydruk 13.4), gdzie zmienne `L` oznaczają długości ramion, `LxRadius` opisują promień przekroju poprzecznego, natomiast `H` i `G` odpowidają za wymiary podstawy.

Wydruk 13.4. Parametry manipulatora

```
1  static float L1 = 0.047f;
       static float L2 = 0.1148f;
3      static float L3 =0.0395f;
       static float L4 = 0.09525f;
5      static float L5 = 0.064f;
       static float L6 = 0.067f;
7      static float L7 = 0.083f;
       static float H = Conversions.InchesToMeters(3f);
9      static float G = Conversions.InchesToMeters(2f);
       static float L1Radius = 0.005f;
11     static float L2Radius = 0.005f;
       static float L3Radius = 0.005f;
13     static float L4Radius = 0.005f;
       static float L5Radius = 0.005f;
15     static float L6Radius = 0.005f;
       static float L7Radius = 0.005f;
```

Kolejnym elementem jest klasa `JointDesc`, która odpowiada za przechowywanie informacji o przegubach, takich jak ich nazwa, zakres ruchu, instancja w symulacji, położenie (docelowe oraz aktualne) oraz prędkość (wydruk 13.5). Metedy tej klasy odpowiadają za sprawdzenie czy żądane położenie znajduje się w zakresie ruchu (`ValidTarget`), sprawdzenie czy należy wykonać ruch (`NeedToMove`) oraz wykonanie ruchu (`UpdateCurrent`).

Wydruk 13.5. Klasa `JointDesc`

```
class JointDesc
        {
            public string Name;
            public float Min;
            public float Max;
            public PhysicsJoint Joint;
            public PhysicsJoint Joint2;
            public float Target;
            public float Current;
            public float Speed;
            public JointDesc(string name,
                             float min,
                             float max)
            {
                Name = name; Min = min; Max = max;
                Joint = null;
                Joint2 = null;
                Current = Target = 0;
                Speed = 30;
            }
            public bool ValidTarget(float target)
            {
                return ((target >= Min) && (target <= Max));
            }
            public bool NeedToMove(float epsilon)
            {
                if (Joint == null) return false;
                return (Math.Abs(Target - Current) > epsilon);
            }
            public void UpdateCurrent(double time)
            {
                float delta = (float)(time * Speed);
                if (Target > Current)
                    Current = Math.Min(Current + delta, Target);
                else
                    Current = Math.Max(Current - delta, Target);
            }
        }
```

Ciało klasy `Arm` tworzone jest również przez tablicę elementów typu `JointDesc`. Jej inicjalizacja została zaprezentowana na wydruku 13.6.

Wydruk 13.6. Tablica opisująca przeguby

```
JointDesc[] _joints = new JointDesc[]
        {
```

```
            new JointDesc("Shoulder Base", -180, 180),
            new JointDesc("Shoulder Pitch", -180, 180),
            new JointDesc("Shoulder Yaw", -180, 180),
            new JointDesc("Elbow Pitch", -180, 180),
            new JointDesc("Wrist Roll", -180, 180),
            new JointDesc("Wrist Yaw", -180, 180),
            new JointDesc("Wrist Pitch", -180, 180),
        };
```

Elementy `file` oraz `writer` są niezbędne do zapisu ruchu ramienia do pliku (wydruk 13.7).

Wydruk 13.7. Obiekty zapisujące dane do pliku

```
FileStream file=null;
StreamWriter writer=null;
```

Głównym zadaniem konstruktora klasy `Arm` (wydruk 13.8) jest inicjalizacja instancji fizycznej modelowanego ramienia. W pierwszych dwóch liniach tworzone są obiekty niezbędne do zapisu danych otrzymanych z symulacji do pliku. Modelowanie samego manipulatora rozpoczyna się od ustawienia ramienia w odpowiednim miejscu oraz nadania właśnie tworzonemu obiektowi kształtu prostopadłościanu, który będzie pełnił rolę podstawy. W dalszej części tej metody tworzone są kolejne ogniwa. Proces ten zaczyna się od stworzenia kształtu kapsuły o wymiarach odpowiednich dla tworzonego ogniwa. Następnie definiowany jest obiekt klasy `SingleShapeSegmentEntity` o zadanym kształcie. Kolejnym etapem jest skonfigutowanie przegubu stowarzyszonego z nowo utworzonym ogniwem. Wybierana jest oś obrotu, łączone obiekty oraz punkty tych połączeń. Ostatecznie ogniwo dodawane jest do środowiska symulacyjnego. Proces ten powtarzany jest dla każdego ogniwa.

Wydruk 13.8. Konstruktor klasy `Arm`

```
public Arm(string name,Vector3 position)
        {
        file = new FileStream(@"C:\Users\...\output.txt",
                                        FileMode.Create);
            writer = new StreamWriter(file);

            SpringProperties spring = new SpringProperties(
                                Int32.MaxValue, 1000, 0);

            float baseHeight = H - L1Radius - 0.001f;
            State.Name = name;
            State.Pose.Position = position;
            State.Pose.Orientation = new Quaternion(0, 0,
                                                0, 1);

            // build the base
            BoxShape = new BoxShape(new BoxShapeProperties(
                "Base",
                50, // mass
                new Pose(new Vector3(0, baseHeight / 2, 0),
                                new Quaternion(0, 0, 0, 1)),
```

```
                new Vector3(G * 2, baseHeight, G * 2)));
            // build and position L0 (top of the base)
            SphereShape L0Sphere = new SphereShape(
        new SphereShapeProperties(
                "L0Sphere",
                50,
                new Pose(new Vector3(0, 0, 0),
                        new Quaternion(0, 0, 0, 1)),
                L1Radius));

        SingleShapeSegmentEntity L0Entity =
                new SingleShapeSegmentEntity(L0Sphere,
                        position + new Vector3(0, H, 0));
            L0Entity.State.Pose.Orientation =
                                new Quaternion(0, 0, 0, 1);
            L0Entity.State.Name = name + "_L0";
            JointAngularProperties L0Angular =
                        new JointAngularProperties();
            L0Angular.Swing1Mode = JointDOFMode.Free;
            L0Angular.SwingDrive = new JointDriveProperties(
                               JointDriveMode.Position,
                                spring, 100000000);
            EntityJointConnector[] L0Connectors =
                        new EntityJointConnector[2]
            {
                new EntityJointConnector(L0Entity,
                                new Vector3(0,1,0),
                                new Vector3(1,0,0),
                                new Vector3(0, 0, 0)),
                new EntityJointConnector(this,
                                new Vector3(0,1,0),
                                new Vector3(1,0,0),
                                new Vector3(0, H, 0))
            };
            L0Entity.CustomJoint = new Joint();
            L0Entity.CustomJoint.State =
        new JointProperties(L0Angular, L0Connectors);
            L0Entity.CustomJoint.State.Name = "BaseJoint";

            this.InsertEntityGlobal(L0Entity);   {...}
        }
```

Kolejne dwie metody klasy `Arm` pozwalają na odczytanie zadanej konfiguracji ramienia (`GetTargetJointAngles`) oraz jego konfiguracji aktualnej (`GetCurrentJointAngles`). Zostały one zaprezentowane na wydruku 13.9.

Wydruk 13.9. Metody dostępu do konfiguracji

```
public void GetTargetJointAngles(
                    out float shoulderBaseVal,
                    out float shoulderPitchVal,
                    out float shoulderYawVal,
                    out float elbowPitchVal,
```

```
                    out float wristRollVal,
                    out float wristYawVal,
                    out float wristPitchVal)
        {
            shoulderBaseVal = _joints[0].Target;
            shoulderPitchVal = _joints[1].Target;
            shoulderYawVal = _joints[2].Target;
            elbowPitchVal = _joints[3].Target;
            wristRollVal = _joints[4].Target;
            wristYawVal = _joints[5].Target;
            wristPitchVal = _joints[6].Target;
         }
        public void GetCurrentJointAngles(
                    out float shoulderBaseVal,
                    out float shoulderPitchVal,
                    out float shoulderYawVal,
                    out float elbowPitchVal,
                    out float wristRollVal,
                    out float wristYawVal,
                    out float wristPitchVal)
        {
            shoulderBaseVal = _joints[0].Current;
            shoulderPitchVal = _joints[1].Current;
            shoulderYawVal = _joints[2].Current;
            elbowPitchVal = _joints[3].Current;
            wristRollVal = _joints[4].Current;
            wristYawVal = _joints[5].Current;
            wristPitchVal = _joints[6].Current;
        }
```

Pola zaprezentowane na wydruku 13.10 są niezbędne do realizacji ruchu. Zmienna `_moveToActive` determinuje, czy obecnie wykonywany jest ruch, `_epsilon` ustala jego dokładnośc, natomiast port `_moveToResponsePort` odpowiada za odbieranie informacji o żądanym położeniach i wysłanie informacji zwrotnej o sukcesie bądź porażce.

Wydruk 13.10. Realizacja ruchu

```
bool _moveToActive = false;
        const float _epsilon = 0.01f;
        SuccessFailurePort _moveToResponsePort = null;
```

Bardzo ważną role pełni funkcja `MoveTo` (wydruk 13.11). Jest ona odpowiedzialna za ustawienie docelowej konfiguracji przegubów, po uprzednim zweryfikowaniu ich poprawności. Metoda ta również wylicza z jaką prędkością powinno poruszać się każde ogniwo aby osiągnąc cel w zadanym czasie.

Wydruk 13.11. Metoda `MoveTo` klasy `Arm`

```
public SuccessFailurePort MoveTo(
            float shoulderBaseVal,
            float shoulderPitchVal,
            float shoulderYawVal,
            float elbowPitchVal,
            float wristRollVal,
```

```
7           float wristYawVal,
            float wristPitchVal,
9           float time)
        {
11          SuccessFailurePort responsePort =
                        new SuccessFailurePort();
13
            if (_moveToActive)
15          {
                responsePort.Post(
17       new Exception("Previous MoveTo still active."));
                return responsePort;
19          }

21          if (!_joints[0].ValidTarget(shoulderBaseVal))
            {
23              responsePort.Post(
                 new Exception(_joints[0].Name +
25               "Joint set to invalid value: " +
                 shoulderBaseVal.ToString()));
27              return responsePort;
            }
29
            if (!_joints[1].ValidTarget(shoulderPitchVal))
31          {
                responsePort.Post(
33               new Exception(_joints[1].Name +
                 "Joint set to invalid value: " +
35              shoulderPitchVal.ToString()));
                return responsePort;
37          }

39      ...
            // set the target values on the joint descriptors
41          _joints[0].Target = shoulderBaseVal;
            _joints[1].Target = shoulderPitchVal;
43          _joints[2].Target = shoulderYawVal;
            _joints[3].Target = elbowPitchVal;
45          _joints[4].Target = wristRollVal;
            _joints[5].Target = wristYawVal;
47          _joints[6].Target = wristPitchVal;

49          for (int i = 0; i < _joints.Length; i++)
                _joints[i].Speed = Math.Abs(
51       _joints[i].Target - _joints[i].Current) / time;

53          _moveToActive = true;

55          _moveToResponsePort = responsePort;

57          return responsePort;
        }
```

Metoda `Update` (wydruk 13.12) wywoływana jest w każdej klatce symulacji. Z tego względu własnie w niej realizowany jest ruch na podstawie zebranego wcześniej opisu. Na początku tej metody sprawdzane jest czy tablica zawierająca opisy przegubów zawiera już informacje ich symulowanych instancjach. Następnie sprawdzane jest, czy którykolwiek z przegubów wymaga przesunięcia. Jeśli tak, to ta operacja jest wykonywana. Ostatnim zadaniem tej metody jest zapis aktualnej pozycji do pliku.

Wydruk 13.12. Metoda `Update` klasy `Arm`

```
    public override void Update(FrameUpdate update)
    {
        if (_joints[0].Joint == null)
        {
            VisualEntity entity = this;
            if (entity.Children.Count > 0)
            {
                entity = entity.Children[0];
                _joints[0].Joint = (PhysicsJoint)entity.
                                    ParentJoint;
                if (entity.Children.Count > 0)
                {
                    entity = entity.Children[0];
                    _joints[1].Joint =
             (PhysicsJoint)entity.ParentJoint;
            ...
                }
            }
        }

        base.Update(update);

        // update joints if necessary
        if (_moveToActive)
        {
            bool done = true;
           // Check each joint and update it if necessary.
            if (_joints[0].NeedToMove(_epsilon))
            {
                done = false;

                Vector3 normal =
    _joints[0].Joint.State.Connectors[0].JointNormal;
                _joints[0].UpdateCurrent(_prevTime);
                _joints[0].Joint.
            SetAngularDriveOrientation(
                    Quaternion.FromAxisAngle(normal.X,
             normal.Y, normal.Z,
             Conversions.DegreesToRadians(
                                _joints[0].Current)));
            }
      ...
            if (_joints[6].NeedToMove(_epsilon))
            {
```

```
                    done = false;

                    Vector3 axis =
        _joints[6].Joint.State.Connectors[0].JointNormal;
            _joints[6].UpdateCurrent(_prevTime);
            _joints[6].Joint.SetAngularDriveOrientation(
            Quaternion.FromAxisAngle(axis.X, axis.Y,
        axis.Z, Conversions.DegreesToRadians(
                                _joints[6].Current)));
                }
                if (done)
                {
                        _moveToActive = false;
                        _moveToResponsePort.Post(
                                new SuccessResult());

                }
            }
            _prevTime = update.ElapsedTime;

            if (writer != null && L7Entity != null)
            {
                writer.WriteLine(
        update.ApplicationTime.ToString() + " "
            + L7Entity.Position.X.ToString() + " "
            + L7Entity.Position.Y.ToString() + " "
            + L7Entity.Position.Z.ToString() + " "
            + L7Entity.RotationAngles.X.ToString() + " "
            + L7Entity.RotationAngles.Y.ToString() + " "
            + L7Entity.RotationAngles.Z.ToString() + " ");
            }
        }
```

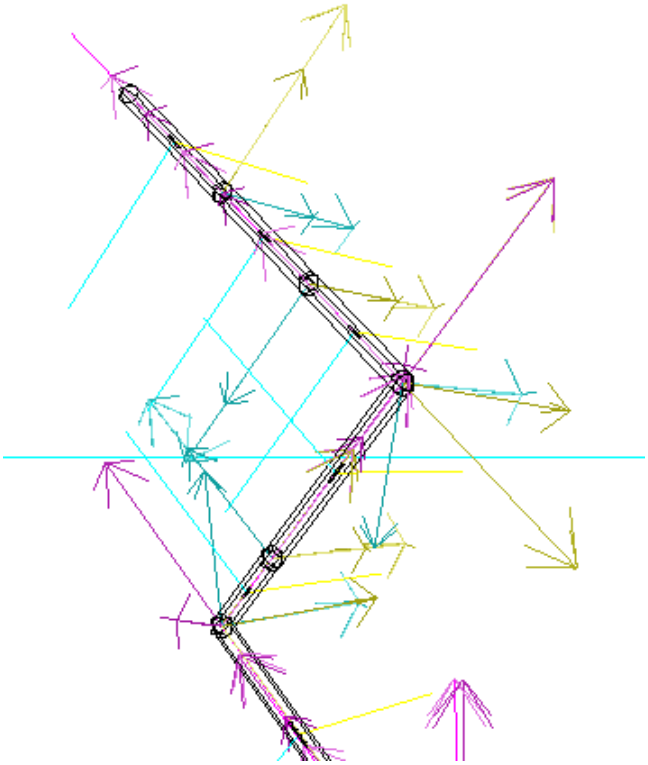
Gotowe ramię zostało zaprezentowane na rysunku 13.8. Możliwe jest dalsze zwiększenie funkcjonalności ramienia m.in. o zaimplementowanie odwrotnej kinematyki. Wiele przydatnych wskazówek na ten temat znajduje się w [2].

Klasa `MobileManipulator`

Klasa `MobileManipuator` implementuje plaftormę mobilną oraz dodaje do jej podwozia manipulator. Dzięki temu, że dziedziczy ona po klasie `DifferentialDriveEntity` (zdokumentowanej w [5]) zaimplementowanie monocykla jest bardzo uproszczone. Ogranicza się ono jedynie do określenia kształtu podwozia i kół w konstruktorze klasy (wydruk 13.13). W ostatnich liniach omawianej metody tworzony jest manipulator.

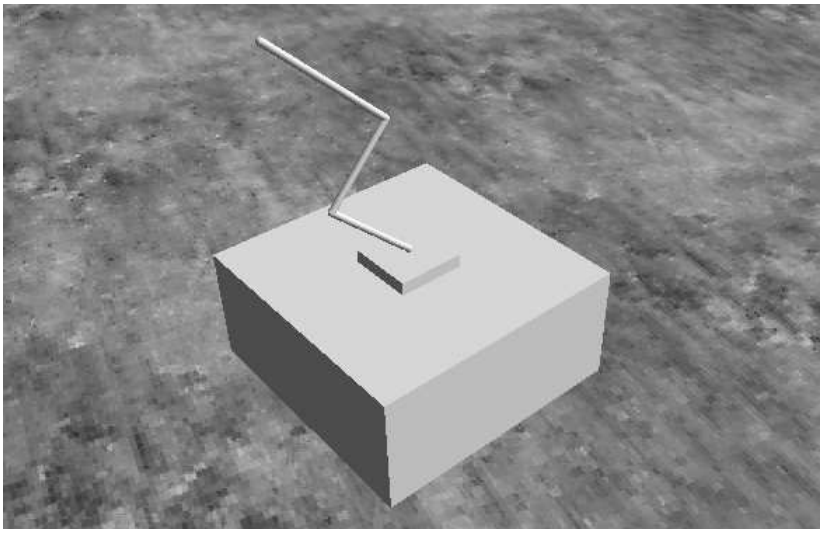
Wydruk 13.13. Klasa `MobileManipulator`

```
public MobileManipulator(Vector3 initialPos)
        {
            MASS = 200;
            CHASSIS_DIMENSIONS = new Vector3(
                                0.393f, 0.18f, 0.40f);
            CHASSIS_CLEARANCE = 0.05f;
            FRONT_WHEEL_RADIUS = 0.08f;
```



Rysunek 13.8. Instancja fizyczna ramienia

```
8           CASTER_WHEEL_RADIUS = 0.025f;
            FRONT_WHEEL_WIDTH = 4.74f;
10          CASTER_WHEEL_WIDTH = 0.02f;
            FRONT_AXLE_DEPTH_OFFSET = -0.05f;
12
            base.State.Name = "MobileManipulator";
14          base.State.MassDensity.Mass = MASS;
            base.State.Pose.Position = initialPos;
16
            BoxShapeProperties motorBaseDesc =
18      new BoxShapeProperties("chassis", MASS,
                new Pose(new Vector3(
20              0,
                CHASSIS_CLEARANCE + CHASSIS_DIMENSIONS.Y / 2,
22              0)),
                CHASSIS_DIMENSIONS);
24
            motorBaseDesc.Material =
26      new MaterialProperties("high_friction",
                                        0.0f, 1.0f, 20.0f);
28          motorBaseDesc.Name = "Chassis";
            ChassisShape = new BoxShape(motorBaseDesc);
30
            CASTER_WHEEL_POSITION = new Vector3(0,
32              CASTER_WHEEL_RADIUS, // distance from ground
                CHASSIS_DIMENSIONS.Z / 2 -
34                              CASTER_WHEEL_RADIUS);
```



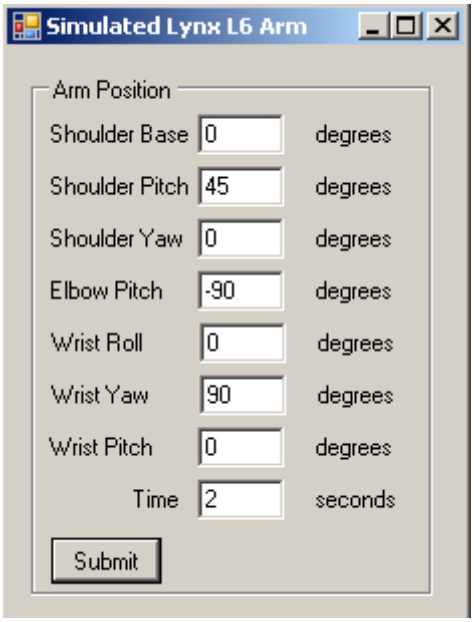
Rysunek 13.9. Manipulator mobilny w środowisku MRDS

```
36              FRONT_WHEEL_MASS = 0.10f;

38              RIGHT_FRONT_WHEEL_POSITION = new Vector3(
                    CHASSIS_DIMENSIONS.X / 2 + 0.01f - 0.05f,
40                  FRONT_WHEEL_RADIUS,
                    FRONT_AXLE_DEPTH_OFFSET);
42
                LEFT_FRONT_WHEEL_POSITION = new Vector3(
44                  -CHASSIS_DIMENSIONS.X / 2 - 0.01f + 0.05f,
                    FRONT_WHEEL_RADIUS,
46                  FRONT_AXLE_DEPTH_OFFSET);

48              MotorTorqueScaling = 20;

50              ConstructStateMembers();

52              arm = new Arm("Arm", new Vector3(0,
                                CHASSIS_DIMENSIONS.Y, 0));
54
                this.InsertEntity(arm);
56          }
```

Postać całego robota została zaprezentowana na rysunku 13.9.

Graficzny interfejs użytkownika

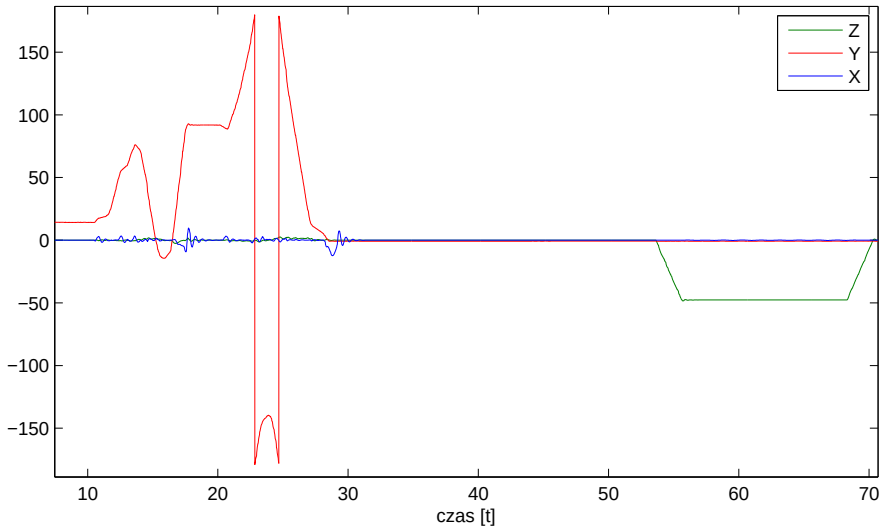
Graficzny interfejs użytkownika zaprezentowany został na rysunku 13.10. Pozwala on na ustalenie konfiguracji manipulatora we współrzędnych przegubowych. Czas trwania ruchu określony jest przez ostatni parametr. Ponieważ robot działa w innym wątku niż interfejs użytkowanika komunikacja musi odbywać się przy pomocy interfejsu portów i wiadomości. W tym celu klasa `RobotArmUI` zawiera port `_fromWinformPort`. Na wydruku 13.14 zaprezentowana została metoda uruchamiana po naciśnięciu przycisku `Submit`. Jej zadaniem jest odczytanie parametrów z okna i wysłanie ich do manipulatora.



Rysunek 13.10. Graficzny interfejs użytkownika

Wydruk 13.14. Metoda `Submit` klasy `RobotArmUI`

```
private void _submitButton_Click(object sender, EventArgs e)
        {
            try
            {
                MoveToPositionParameters moveParams =
                        new MoveToPositionParameters();

                _errorLabel.Text = string.Empty;

                moveParams.shoulderBaseVal =
                 Single.Parse(shoulderBaseText.Text);
                moveParams.shoulderPitchVal =
                Single.Parse(shoulderPitchText.Text);
                moveParams.shoulderYawVal =
                 Single.Parse(shoulderYawText.Text);
                moveParams.elbowPitchVal =
                 Single.Parse(elbowPitchText.Text);
                moveParams.wristRollVal =
                 Single.Parse(wristRollText.Text);
                moveParams.wristYawVal =
                 Single.Parse(wristYawText.Text);
                moveParams.wristPitchVal =
                 Single.Parse(wristPitchText.Text);
                moveParams.time =
                 Single.Parse(_timeText.Text);

                _fromWinformPort.Post(
                 new FromWinformMsg(
        FromWinformMsg.MsgEnum.MoveToPosition,
                                    null, moveParams));
```



Rysunek 13.11. Przykład prezentacji wyników

```
            }
32          catch
            {
34              _errorLabel.Text = "Invalid Value";
            }
36      }
```

13.4.2. Prezentacja wyników

Przeprowadzana symulacja wizualizowana jest w czesie rzeczywistym w interaktywnym, trójwymiarowym środowisku. Oprócz tego zaimplementowano możliwość zapisu uzyskanych danych do pliku. Sposób realizacji tego procesu został opisany w rozdziale 13.4.1. Rezultat zostaje zapisywany w postaci tabeli w pliku tekstowym. W rozważanym przypadku zapisano położenie efektora we współrzędnych zewnętrznych. Tak otrzymane dane mogą zostać wkorzytasne w innym programie. Na rysunku 13.11 został przedstawiony wykres stworzony w MATLABie.

Inną możliwościa prezentowania danych jest wyświetlanie ich przy pomocy narzędzi dostarczanych wraz ze środowiskiem `.Net`, a dokladnie biblioteki `Microsoft Chart Controls`. W tym przypadku możliwe jest generowanie wykresów w czasie rzeczywistym. Aby otrzymać taką funkcjonalność należy stworzyć nowe okno aplikacji z wbudowanych wykresem, a następnie skomunikować je z symulacją analogicznie jak w przypadku graficznego interfejsu użytkownika.

13.5. Podsumowanie

Wykorzystywane w tym rozdziale narzędzia, Mathematica oraz Microsoft Robotics Developer Studio prezentują zupełnie różny sposób podejścia do modelowania obiektów robotycznych. Opis manipulatora mobilnego wykonany w programie Mathematica jest analogiczny do opisu w postaci równań matematycznych. Natomiast model w środowisku MRDS jest bardzo specyficznym programem napisanym obiektowo.

Różne podejścia determinują różne wymagania stawiane potencjalnemu użytkownikowi. O ile w przypadku programu Mathematica najważniejsza jest umiejetność modelowania matematycznego lub fakt posiadania wyprowadzonych równań ruchu obiektu, to w przypadku środowiska MRDS wymagana jest umiejętność programowania obiektowego.

Samo wprowadzanie modelu w programie Wolfram Mathematica po poznaniu podstaw jego obsługi jest bardzo intuicyjne, pozwala na bardzo łatwe przeniesienie równań z kartki papieru do postaci rozpoznawanej przez aplikację. Nie możemy natomiast liczyć na rozbudowane wsparcie samego pocesu tworzenia modelu, co z jednej strony wymaga znacznej wiedzy z zakresu matematyki i fizyki, z drugiej strony pozwala na zaimplementowanie rozwiązań niewspieranych przez producenta.

Twórcy środowiska MRDS zaprezentowali zupełnie inne podejście do modelowania. W tym przypadku otrzymujemy narzędzie ukierunkowane wyłącznie na zastosowania robotyczne. Efektem jest środowisko, które wymaga jedynie podstawowej wiedzy o prawach fizyki opisujących zachowanie opisywanych układów. Niestety okupione jest to węższym zakresem zastosowań. Również innowacyne podejście do programowania współbierznego implikuje pewne trudności, a płynne posługiwanie się tym narzędziem wymaga swobody w programowaniu obiektowym oraz wielu godzin spędzonych z samym środowiskiem.

Podsumowując, opisywane narzędzia mają zupełnie różną funkcjonalność, więc nie można ocenić, które z nich jest lepsze. Można natomiast polecić je do różnych zastosowań. Program Wolfram Mathematica znakomicie sprawdzi się np. przy testowaniu algorytmów sterowania najniższego poziomu tj. śledzenia trajektorii czy ścieżki. Jest niezastąpiony, gdy konieczne jest wyprowadzenie modelu matematycznego skomplikowanego obiektu. Natomiast MRDS jest bardzo pomocnym narzędzniem przy projektowaniu i testowaniu algorymów planowania ścieżki czy omijania przeszkód. Oba narzędzia są warte polecenia i doskonale sprawdzą się w zagadnienach pod których kątem zostały stworzone. Implementacja gotowych rozwiązań na istniejącym obiekcie fizycznym mogłaby składać się z syntezy algorytmów niskiego poziomu zaprojektowanych w Mathematice oraz algorytmów wysokiego poziomu przetestowanych w MRDS.

Bibliografia

[1] R. Hossa. *Modele i algorytmy sterowania kołowych robotów mobilnych.* Praca doktorska, Politechnika Wrocławska, 1996.

[2] K. Johns, T. Taylor. *Professional Microsoft Robotics Developer Studio.* Wiley Publishing, 2008.

[3] A. Mazur. Algorytmy sterowania manipulatorów mobilnych - analiza jakości. Raport instytutowy, Politechnika Wrocławska, 2000, 2000.

[4] A. Mazur. *Sterowanie oparte na modelu dla nieholonomicznych manipulatorów mobilnych.* Oficyna wydawnicza Politechniki Wrocławskiej, Wrocław, 2009.

[5] Dokumentacja microsoft robotics developer studio. http://msdn.microsoft.com/en-us/library/dd145240.aspx.

[6] Wymagania microsoft robotics developer studio. http://www.microsoft.com/robotics/#GetStartedStep2.

[7] K. Tchoń, A. Mazur, I. Dulęba, R. Hossa, R. Muszyński. *Manipulatory i roboty mobilne.* Akademiczna Oficyna Wydawnicza PLJ, 2000.

14. Sześcionożny robot kroczący

Dawid Powązka

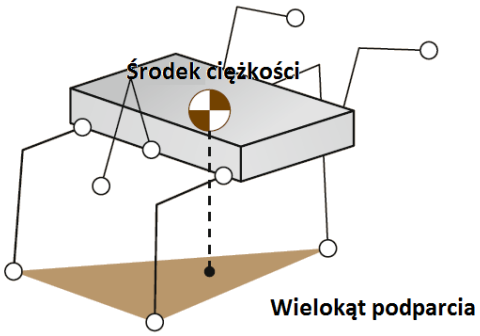
Roboty kroczące zawdzięczają swoją atrakcyjność systemowi lokomocji zbliżonemu do sposobu poruszania się zwierząt. Wykazują przez to możliwości adaptacji i poruszania się w bardzo zróżnicowanych i trudnych środowiskach. Poniższe rozdziały skupiają się na podstawach modelowania sześcionożnych robotów kroczących oraz implementacji podanych metod w systemie V-REP.

14.1. Budowa robota

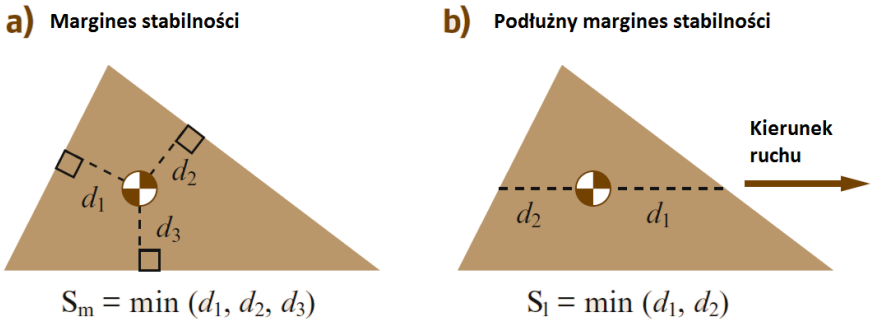
Sześcionożny robot kroczący jest maszyną kroczącą o 6 nogach. Roboty te są głównie inspirowane owadami oraz imitują ich sposób chodu. Każda z nóg robota posiada 3 stopnie swobody. Maszyny sześcionożne realizują chód statycznie stabilny. Oznacza to, że robot zachowuje stabilność w każdym punkcie ruchu oraz, że jego środek ciężkości nie wychodzi poza wielokąt podparcia. Wielokątem podparcia nazywamy wielokąt powstały z połączenia wierzchołków nóg opartych na podłożu (rysunek 14.1). Definiuje się również pojęcie marginesu stabilności jako odległość środka ciężkości maszyny do krawędzi wielokąta podparcia (rysunek 14.2). Ruch robota generuje się w taki sposób, aby nie przekroczyć pewnej ustalonej doświadczalnie granicy stabilności. Dzięki temu do opisu ruchu robota można stosować wyłącznie metody kinematyki. Maszyny sześcionożne realizują 3 sposoby chodu:
— trójpodporowy,
— czteropodporowy,
— pięciopodporowy.
Noga znajduje się w fazie podporowej wtedy gdy znajduje się na podłożu. Natomiast faza przenoszenia oznacza nogę znajdującą się w powietrzu. Realizacja chodu trójpodporowego odbywa się poprzez naprzemienne przenoszenie odpowiednich trójek nóg. Jest to najszybszy sposób poruszania się sześcionożnego robota kroczącego. W chodzie pięciopodporowym, w jednym czasie przenoszona jest jedynie jedna noga, podczas gdy pozostałe 5 znajduje się na podłożu. Jest to najwolniejszy sposób kroczenia. Chód czteropodporowy jest chodem pośrednim, w którym jednocześnie przenoszone są 2 nogi. W opisie chodu robotów stosuje się diagramy chodu (rysunek 14.3). W pozycji [1] zdefiniowany jest współczynnik obciążenia nogi β (*duty factor*), który jest czasem styku nogi z podłożem znormalizowanym w stosunku do okresu chodu. Dla chodu trójpodporowego współczynnik ten wynosi $\beta = 0.5$.

14.2. Virtual Robot Experimentation Platform (V-REP)

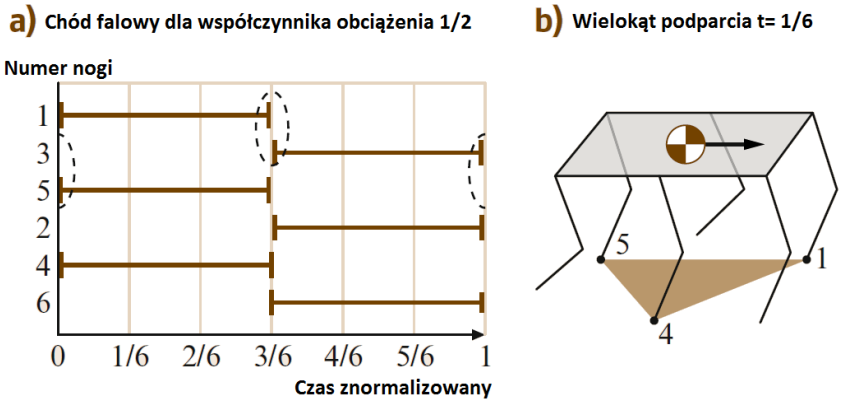
Virtual Robot Evaluation Platform jest systemem symulacji robotów z wbudowanym interfejsem edytorskim. V-REP jest wykorzystywany do symulacji, testowania, ewaulacji prostych i złożonych systemów zrobotyzowanych lub robotycznych podzespołów. System V-REP jest wykorzystywany w dziedzinie robotyki do:
— symulacji zespołów robotycznych,



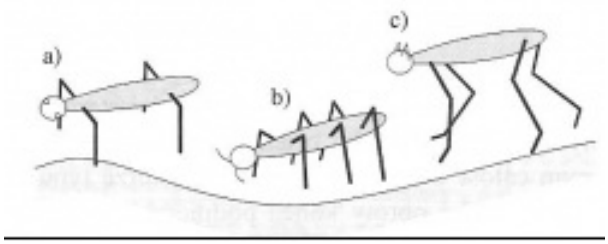
Rysunek 14.1. Wielokąt podparcia [1]



Rysunek 14.2. Margines stabilności [1]



Rysunek 14.3. Chód trójpodporowy [1]



Rysunek 14.4. Rodzaje postury nóg

— wizualizacji procesów przemysłowych,
— szybkiego prototypowania nowych podzespołów,
— badania i budowy systemów sterowania robotów,
— prezentacji możliwości robotów.

Atrakcyjność użycia środowiska V-REP do modelowania sześcionożnego robota kroczącego polegała na wykorzystaniu istniejącego w nim modelu, opisanego w podrozdziale14.4.1. Więcej informacji na temat samego systemu można znaleźć w rozdziale 5 oraz w pozycji [2].

14.3. Modelowanie sześcionożnego robota kroczącego

Modelowanie sześcionożnych robotów kroczących odbywa się głównie z wykorzystaniem metod kinematyki. Poniższy rozdział opisuje podstawowe zagadnienia związane z tą tematyką, natomiast szczegółowe informacje można znaleźć w publikacji [3].

14.3.1. Kinematyka nogi robota

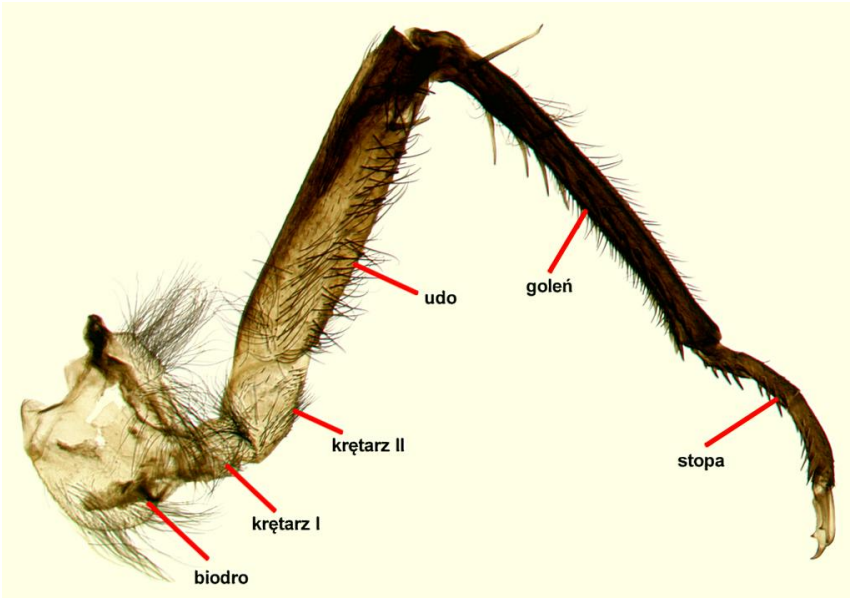
Robot kroczący może mieć jedną z postur nóg przedstawionych na rysunku 14.4. Wyróżnione zostały tutaj kolejno od prawej postura nóg:
— płaza,
— owada,
— kręgowca.

W dalszych rozważaniach będziemy się zajmować posturą nogi owada, która jest typowa dla robotów sześcionożnych. Szczegółowy schemat budowy został przedstawiony na rysunku 14.5. Noga taka posiada 3 stopnie swobody. W biodrze znajdują się 2 przeguby. Pierwszy przenosi nogę w zakrok i wykrok, a drugi odpowiada za podniesie nogi do góry lub opuszczenie w dół. Trzeci przegub znajdujący się w kolanie odpowiada za zgięcie nogi. Warto tutaj wspomnieć, że owady utrzymują podczas ruchu zawsze kąt ostry pomiędzy udem a goleniem. Na rysunku 14.6 zostało przedstawione rozmieszczenie układów współrzędnych stowarzyszonych z kolejnymi stawami nogi.

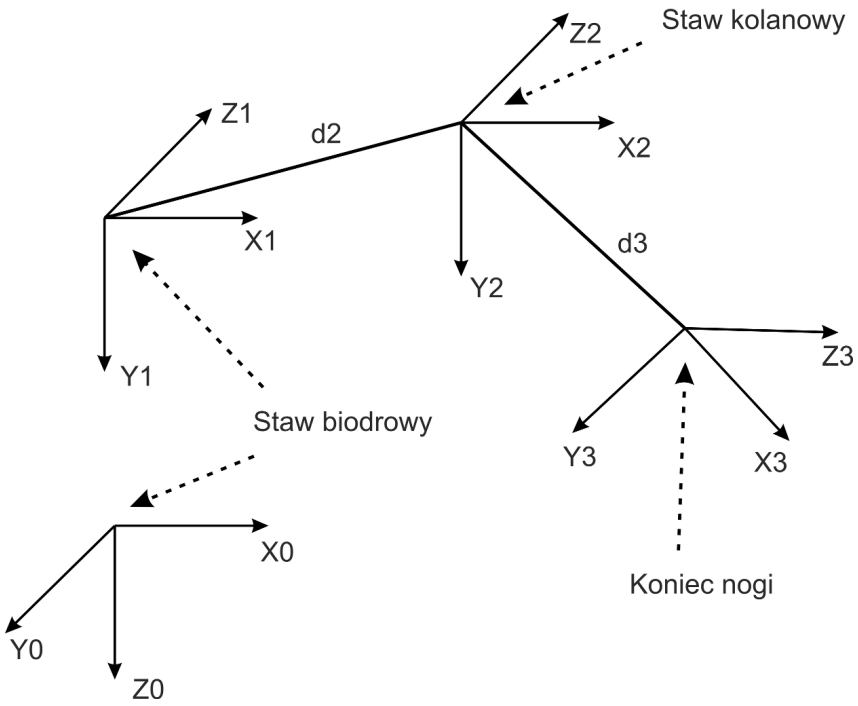
Kinematyka prosta

Kinematykę opisywanej nogi w notacji Denavita-Hartenberga modelują transformacje

$$A_0^1 = Rot(Z,\theta_1)Rot(X,\frac{\pi}{2}), \tag{14.1}$$
$$A_1^2 = Rot(Z,\theta_2)Trans(X,d_2), \tag{14.2}$$
$$A_2^3 = Rot(Z,\theta_3)Trans(X,d_3). \tag{14.3}$$



Rysunek 14.5. Noga owada



Rysunek 14.6. Kinematyka nogi owada

Wynikiem złożenia podstawowych macierzy obrotu i translacji jest macierz kinematyki

$$A_0^3 = \begin{bmatrix} c_1c_{23} & -c_1s_{23} & s_1 & c_1c_{23}d_3 + c_1c_2d_2 \\ s_1c_{23} & -s_1s_{23} & -c_1 & s_1c_{23}d_3 + s_1c_2d_2 \\ s_{23} & c_{23} & 0 & s_{23}d_3 + s_2d_2 \\ 0 & 0 & 0 & 1 \end{bmatrix}, \tag{14.4}$$

gdzie $s_i = \sin(q_i)$, $s_ij = \sin(q_i + q_j)$ oraz tak samo dla $\cos(q_i)$. Współrzędne kartezjańskie końca nogi w układzie bazowym biodra są następujące:

$$P = \begin{bmatrix} p_x \\ p_y \\ p_z \end{bmatrix} = \begin{bmatrix} c_1c_{23}d_3 + c_1c_2d_2 \\ s_1c_{23}d_3 + s_1c_2d_2 \\ s_{23}d_3 + s_2d_2 \end{bmatrix}. \tag{14.5}$$

Zadanie odwrotne kinematyki

Zadanie odwrotne kinematyki polega na wyliczeniu wartości kątów, które ustawią efektor w zadanym położeniu. W rozważanym przypadku, nogi o 3 stopniach swobody, wygodnie jest się posłużyć metodą algebraiczną.

Metoda algebraiczna jest łatwa do implementacji w różnego rodzaju językach programowania. Wykorzystuje funkcję atan2, która jest dwuargumentową funkcją arcusa tangensa. Kąt θ_1 otrzymujemy z równania

$$\theta_1 = atan2(p_y, p_x). \tag{14.6}$$

Kąt θ_3 otrzymujemy w podobny sposób

$$\theta_3 = atan2(\pm s_3, c_3), \tag{14.7}$$

wyliczając uprzednio wartości $\cos(q_3)$ oraz $\sin(q_3)$.

$$c_3 = \frac{(\frac{p_x}{c_1})^2 + p_z^2 - d_2^2 - d_3^2}{2d_2d_3}, \tag{14.8}$$

$$s_3 = \sqrt{1 - c_3^2}. \tag{14.9}$$

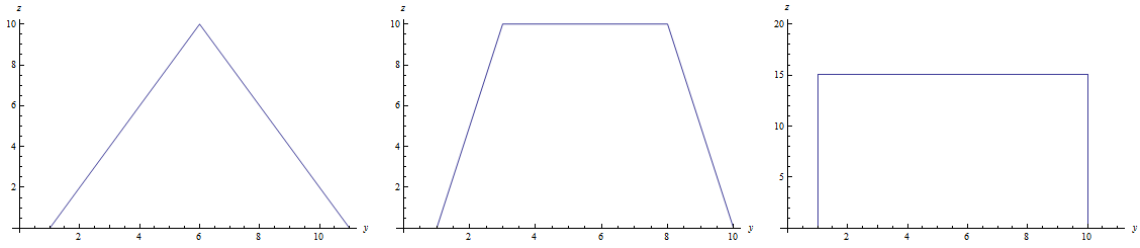
Wyliczenie kąta θ_2 jest nieco bardziej skomplikowane i zawiera dwa przypadki. Dla $\theta_1 \neq 0$ wyliczenia są następujące

$$\theta_2 = atan2(s_2, c_2), \tag{14.10}$$

gdzie $s_2 = \frac{C - p_y}{D + B}$, $c_2 = \frac{c_1s_2s_3l_2}{c_1A}$, $A = c_3d_3 + d_2$, $B = s_1s_3d_3$, $C = \frac{s_1Ap_z}{s_3d_3}$, $D = \frac{s_1A^2}{s_3d_3}$. Wyliczenie kąta θ_2 dla przypadku $\theta_1 = 0$

$$\theta_2 = atan2(p_z, p_x) - atan2(b, A), \tag{14.11}$$

gdzie $b = d_3s_3$. Bezpośrednie wyprowadzenie metody znajduje się w publikacji [3], rozdział 4.1.3 i nie będzie tutaj przytaczane.



(a) Trajektoria trójkątna (b) Trajektoria trapezowa (c) Trajektoria prostokątna

Rysunek 14.7. Trajektorie ruchu końca nogi robota

14.3.2. Generowanie trajektorii końca nogi

Sterowanie robotem kroczącym odbywa się poprzez generowanie odpowiednich trajektorii dla końców nóg robota. Trajektorie te muszą być dobrze zdefiniowane i oczekujemy ich dokładnej realizacji. Odpowiednie sekwencje przestawień nóg powodują ruch postępowy całego robota. W ogólnym przypadku koniec nogi robota, której kinematyka została opisana w podrozdziale 14.3.1, porusza się zwykle po jednej z 3 podstawowych trajektorii, przedstawionych na rysunku 14.7.

Teoretycznie chcemy, aby noga była przemieszczana w fazie podparcia i przenoszenia jak najszybciej jest to możliwe. W rzeczywistości noga nie może przyspieszyć do pewnej ustalonej prędkości w nieskończenie krótkim czasie. Zatem generowana trajektoria musi odpowiadać możliwościom układu napędowego oraz spełniać 4 ograniczenia na ruch: dwa na prędkość początkową i końcową i dwa na położenie początkowe i końcowe.

Trajektoria wielomianowa

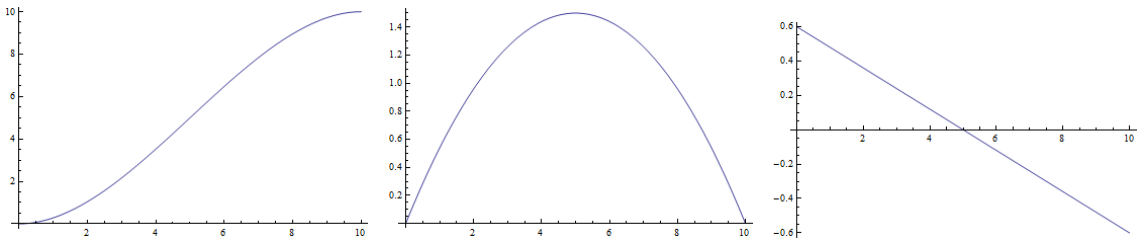
Dla robotów kroczących stosuje się zazwyczaj trajektorie wielomianowe. Trajektorie taką w ogólnym przypadku opisuje równanie (14.12). W podanym wzorze $a = [a_x, a_y, a_z]^T$, $b = [b_x, b_y, b_z]^T$ i $c = [c_x, c_y, c_z]^T \cdots$ wynikają z ograniczeń

$$[x(t), y(t), z(t)]^T = a + bt + ct^2 \cdots \tag{14.12}$$

Dla robotów kroczących używa się głównie trajektorii wielomianowych trzeciego stopnia. Po rozwiązaniu dodatkowych równań ograniczeń nałożonych na ruch, otrzymujemy wzór (14.13) opisujący szukaną trajektorie, gdzie x_0, y_0, z_0 określa warunek początkowy ruchu, a $\Delta x, \Delta y, \Delta z$ całkowitą zmianę danej współrzędnej w czasie T.

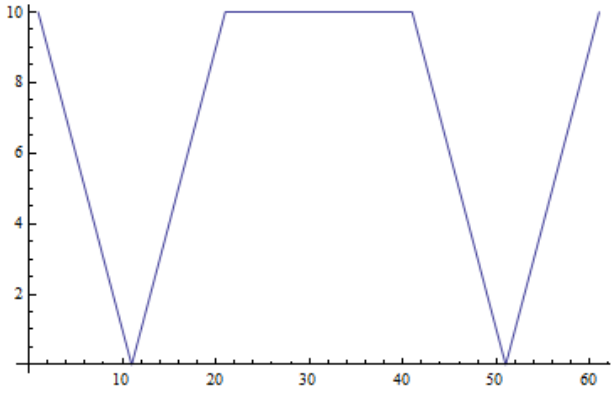
$$\begin{bmatrix} x(t) \\ y(t) \\ z(t) \end{bmatrix} = \begin{bmatrix} x_0 + 3\frac{\Delta x}{T^2}t^2 - 2\frac{\Delta x}{T^3}t^3 \\ y_0 + 3\frac{\Delta y}{T^2}t^2 - 2\frac{\Delta y}{T^3}t^3 \\ z_0 + 3\frac{\Delta z}{T^2}t^2 - 2\frac{\Delta z}{T^3}t^3 \end{bmatrix} \tag{14.13}$$

Na rysunku 14.8 przedstawione zostały wykresy przesunięcia, prędkości i przyspieszenia dla współrzędnej z(t), wygenerowane z zastosowaniem trajektorii (14.13). Na rysunku 14.9 przedstawiono wykresy przesunięcia, prędkości i przyspieszenia dla współrzędnej z(t), w przypadku generowania trajektorii trójkątnej. Użyto tutaj jednostajnej zmiany współrzędnej. Jak widać na wykresie przyspieszenia 14.9(c) pojawiły się niekorzystne impulsy przyspieszenia związane z gwałtownym wzrostem prędkości (wykres 14.9(b)) do pewnej ustalonej wartości. Na rysunku 14.10 przedstawiono przebiegi przy zastosowaniu wielomianowej trajektorii. Przemieszczenie współrzędnej jest gładkie i wyeliminowano wcześniej wspomniane impulsy przyspieszenia.

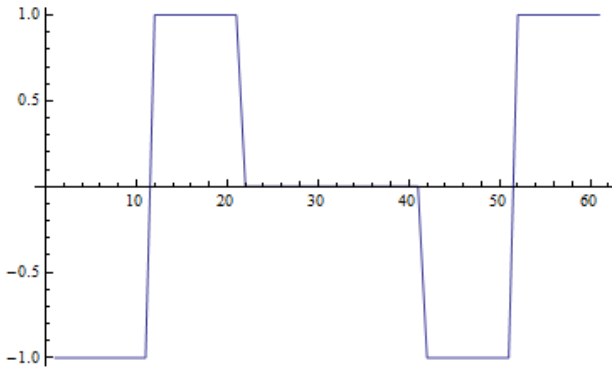


(a) Przesunięcie z(t) (b) Prędkość $\dot{z}(t)$ (c) Przyspieszenie $\ddot{z}(t)$

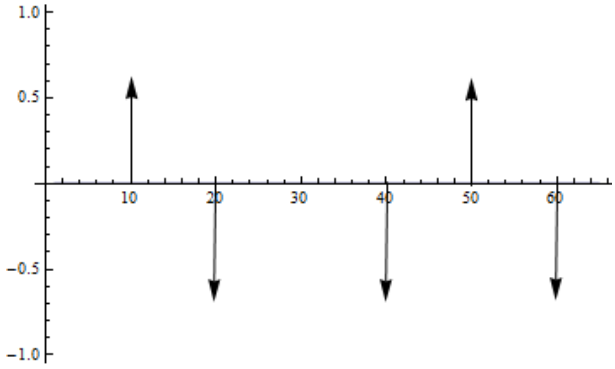
Rysunek 14.8. Przebieg trajektorii dla $\Delta z = 10$ i czasu T=10s



(a) Przemieszczenie z(t)

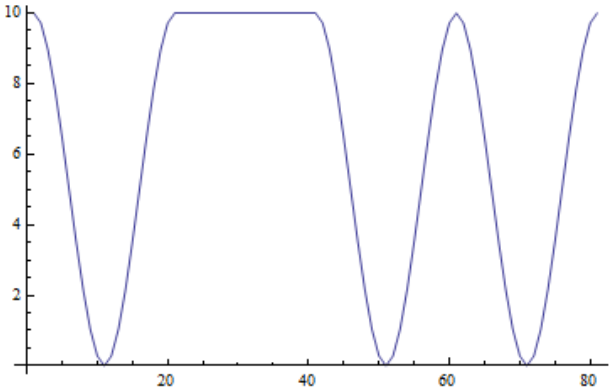


(b) Prędkość z(t)

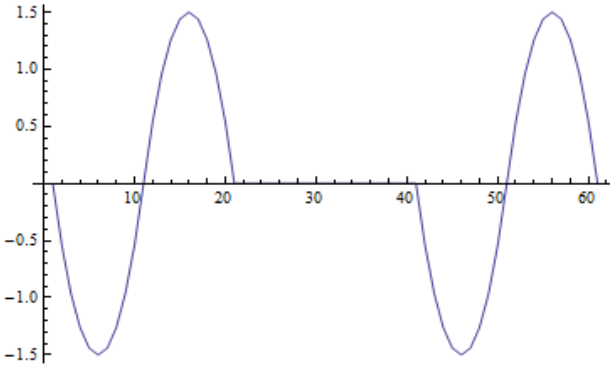


(c) Przyspieszenie z(t)

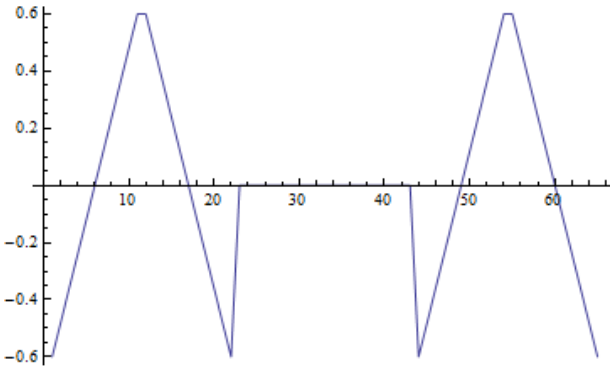
Rysunek 14.9. Przebiegi trajektorii końca nogi – przypadek dla jednostajnej zmiany współrzędnych



(a) Przemieszczenie z(t)



(b) Prędkość z(t)



(c) Przyspieszenie z(t)

Rysunek 14.10. Przebiegi trajektorii końca nogi – trajektoria wielomianowa

14.3.3. Generowanie ruchu robota

Generowanie ruchu robota polega na zadawaniu odpowiednich sekwencji ruchów robota. W każdej sekwencji zadawane są przesunięcia dla każdej ze współrzędnej końca nogi. Przesunięcia te są związane bezpośrednio z etapem ruchu, w którym znajduje się maszyna krocząca. Wyróżnić można 3 podstawowe rodzaje ruchu:
1. Ruch postępowy w zadanym kierunku.
2. Ruch postępowy ze skrętem.
3. Ruchy korpusu.

Poniżej zostały opisane zagadnienia związane z generowaniem poszczególnego rodzaju chodu. Szczegóły implementacyjne zostały pominięte i będą przytoczone w rozdziale 14.4.

Ruch postępowy w zadanym kierunku

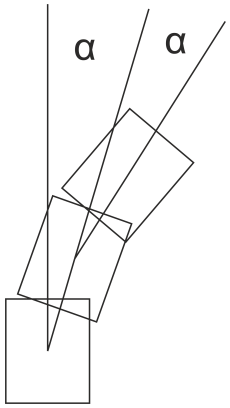
Chód postępowy można generować poprzez wyliczanie odpowiednich przesunięć dla efektora każdej z nóg. Zmiana położenia zależeć będzie od fazy w której znajdować będzie się robot. Dla chodu trójpodporowego fazy mogą być następujące:
1. podnieś nogę 2, 4, 6,
2. przenieś nogi 2, 4, 6 na początek fazy podporowej, nogi 1, 3, 5 na koniec fazy podporowej,
3. postaw nogi 2, 4, 6,
4. podnieś nogi 1, 3, 5,
5. przenieś nogi 1, 3, 5 na początek fazy podporowej, nogi 2, 4, 6 na koniec fazy podporowej,
6. postaw nogi 1, 3, 5.

Nogi 1, 3, 5 określają kolejno nogę lewą przednią, lewą tylnią i prawą środkową. Natomiast nogi 2, 4, 6 określają nogę lewą środkową, prawą tylnią i prawą przednią. Powtarzanie wyżej wymienionych sekwencji będzie powodować ruch postępowy całej maszyny. Odpowiednia modyfikacja współrzędnych początku i końca fazy podporowej będzie skutkowało poruszaniem się robota w wybranym kierunku. Kierunek poruszania się może być określony poprzez kąt ruchu. W takim przypadku początek fazy podporowej może być zmieniony poprzez wykorzystanie funkcji sin i cos.

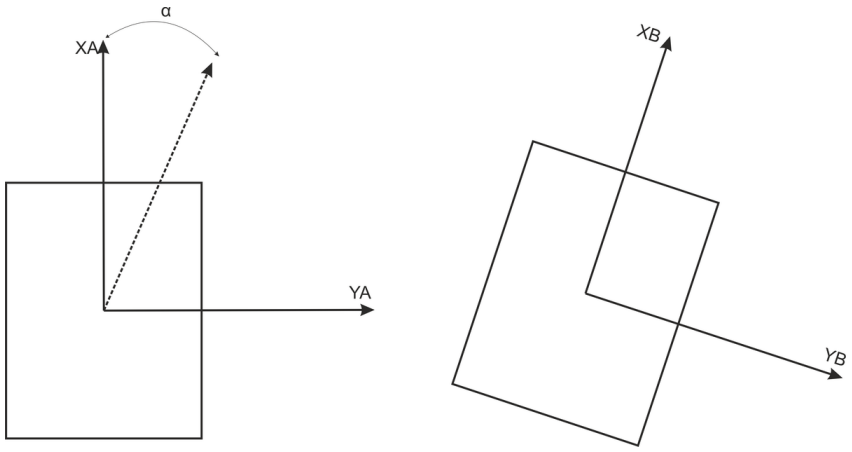
Generowanie chodu pięciopodporowego realizowany jest w podobnej formie do tego podanego powyżej. W tym samym czasie w fazie przenoszenia znajduje się tylko jedna noga. Natomiast pozostałe 5 nóg, znajdujących się w fazie podporowej przemieszczane jest o $\frac{1}{6}$ długości kroku w kierunku końca fazy podporowej.

Ruch postępowy ze skrętem

W przypadku ruchu postępowego ze skrętem trajektorie nóg w fazie podporowej powinny być odpowiednimi łukami okręgu. Długość łuku powinna odpowiadać realizowanej długości kroku. Zmusiłoby to nas do zmiany sposobu generowania trajektorii nogi. Łuki skrętu są zatem przybliżane odpowiednimi odcinkami prostymi, z których każdy jest realizowany w ciągu jednej fazy podporowej. Metoda ta jest bardzo cenna, ponieważ nie wymaga zmiany kształtu trajektorii końca nogi. Modyfikacji ulega jedynie współrzędna początku fazy podporowej nogi. Podczas styku z podłożem noga przemieszcza się po linii prostej. Współrzędne końca fazy podporowej zostają takie same jak w przypadku ruchu postępowego prostoliniowego. Wartość kąta skrętu zależy od kinematyki nogi, rozmiarów maszyny oraz musi spełniać kryteria stabilności chodu. Zwyczajowo maksymalną wartość kąta skrętu wyznacza się eksperymentalnie. Stwierdza się jednak, że w przypadku proporcji geometrycznych nogi i korpusu jak u owadów, może być realizowany obrót o $\pm 5^\circ$ w jednej fazie podporowej. Zatem jeśli w ciągu jednej fazy podporowej korpus obraca się



Rysunek 14.11. Obrót korpusu w ruchu postępowym



Rysunek 14.12. Początkowy i końcowy układ współrzędnych

o kąt α to w n fazach podporowych obróci się o kąt $n\alpha$. Cały proces dobrze ilustruje rysunek 14.11. Stowarzyszmy zatem układ współrzędnych A z pozycją korpusu przed wykonaniem skrętu, czyli na początku fazy podporowej oraz układ B z pozycją korpusu po wykonaniu skrętu, tak jak przedstawiono na rysunku 14.12. Pomiędzy układem A i B występuje następująca transformacja

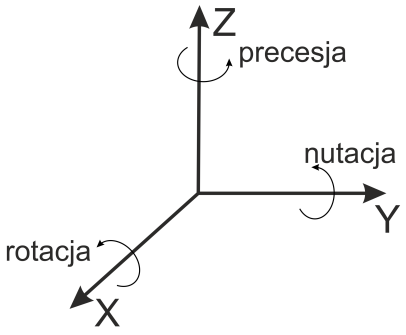
$$T_B^A = Rot(Z, \alpha)Trans(X, l), \tag{14.14}$$

gdzie l oznacza długość kroku, a α kąt skrętu. Zatem macierz T_B^A ma następującą postać

$$T_B^A = \begin{bmatrix} c_\alpha & -s_\alpha & 0 & lc_\alpha \\ s_\alpha & c_\alpha & 0 & ls_\alpha \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}. \tag{14.15}$$

Jeśli następnie zdefiniujemy wektor X_i^B określający współrzędne nogi i-tej w układzie współrzędnych B taki, że

$$X_i^B = \begin{bmatrix} x_i^B \\ y_i^B \\ z_i^B \\ 1 \end{bmatrix}, \tag{14.16}$$



Rysunek 14.13. Kąty obrotu

otrzymamy, że współrzędne nogi i-tej X_i^A w układzie współrzędnych A jest dany następująco

$$X_i^A = T_B^A X_i^B, \tag{14.17}$$

czyli na początku fazy podporowej

$$x_i^A = c_\alpha x_i^B - s_\alpha y_i^B + l c_\alpha \tag{14.18}$$
$$y_i^A = s_\alpha x_i^B + c_\alpha y_i^B + l s_\alpha. \tag{14.19}$$

Powyższe współrzędne należy wyrazić w układzie odniesienia biodra, co jest już zadaniem prostym.

Ruchy korpusu

Robot kroczący może poruszać korpusem stojąc w miejscu lub podczas ruchu postępowego. Bezpośrednio za ruchy korpusy odpowiadają ruchy nóg. Odpowiednie ich przesunięcie pozwoli nam na odchylenie korpusu o pewien kąt lub jego przesunięcie według któreś ze współrzędnych. Na rysunku 14.13 przedstawiono kąty obrotu według osi X, Y i Z, czyli kolejno rotacja, nutacja i precesja. Do tego dochodzi również translacja wzdłuż każdej z osi. Jeśli stowarzyszymy zatem układ współrzędnych B z korpusem maszyny kroczącej, a układ W z nieruchomym układem odniesienia to transformację układu korpusu do układu odniesienia T_B^W jest równa

$$T_B^W = \begin{bmatrix} c_\alpha c_\beta & c_\alpha s_\beta s_\gamma - s_\alpha c_\gamma & c_\alpha s_\beta c_\gamma + s_\alpha s_\gamma & p_x \\ s_\alpha c_\beta & s_\alpha s_\beta s_\gamma + c_\alpha c_\gamma & s_\alpha s_\beta c_\gamma - c_\alpha s_\gamma & p_y \\ -s_\beta & c_\beta s_\gamma & c_\beta c_\gamma & p_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{14.20}$$

Kąt α oznacza obrót wokół osi Z, kąt β obrót wokół osi Y, a kat γ obrót wokół osi X. Wektor $[p_x, p_y, p_z]$ opisuje przesunięcie początku układu współrzędnych korpusu. Jeśli wektor X_i^B będzie określał współrzędne nogi w układzie korpusu B. To położenie końca nogi X_i^W w układzie odniesienia

$$X_i^W = T_B^W X_i^B. \tag{14.21}$$

14.3.4. Planowanie ruchu

Planowanie ruchu robotów kroczących w terenie z przeszkodami polega na wybieraniu punktów stąpnięć tak, aby zachowana była stabilność a przeszkody ominięte. Trajektoria

korpusu powinna być tak dobrana, by nogi zawsze mogły znaleźć podparcie. Zagadnienie to jest bardzo trudne i nie zostało dotychczas w pełni rozwiązane. Wykorzystuje się tutaj metody:
— wnioskowania,
— przeszukiwania grafów,
— drzewa ruchów możliwych,
— funkcji potencjałowych,
— statystyki,
— analizy ograniczeń kinematycznych itd.
Każde z powyżej wymienionych rozwiązań wykorzystuje uproszczenia i nie daje uniwersalnego sposobu na rozwiązanie sformułowanego problemu. W publikacji [3] została przedstawiona metoda przeszukiwania drzewa ruchów możliwych.

14.4. Implementacja modelu w systemie V-REP

Rozdział ten zawiera opis modelowania sześcionożnego robota kroczącego w systemie V-REP. Przedstawione zostały tutaj użyte rozwiązania oraz szczegóły ich implementacji w podanym środowisku. Wyniki działania można zobaczyć wykonując symulację ze sceny dołączonej jako załącznik do tego dokumentu.

14.4.1. Model

W celu implementacji chodu użyto model robota kroczącego dołączonego do systemu V-REP. Został on przedstawiony na rysunku 14.14. Kinematyka robota została zbudowana zgodnie z zaleceniami producenta oprogramowania, umieszczonych w [2]. W środku ciężkości robota zostały umieszczone dwa układy współrzędnych. Pierwszy układ nazywany *Ant_legBase* jest układem odniesienia dla nóg. Natomiast drugi *Ant_base* jest nieruchomym układem współrzędnych powiązanym z korpusem robota. Dzięki temu rozwiązaniu możemy bezpośrednio ustawiać położenie końca nogi w określonym układzie współrzędnych nóg. Dodatkowo układ ten może być przemieszczany względem nieruchomego układu współrzędnych. Następną ważną rzeczą związaną z modelem jest dynamika. W omawianym robocie została ona zaimplementowana, zgodnie z 7 zasadami używania dynamiki w środowisku V-REP. Dzięki dynamicznej symulacji jesteśmy w stanie zaobserwować jakikolwiek ruch postępowy robota, co w przypadku nie używania dynamiki byłoby nie możliwe. Dynamicznie symulowane nogi są w stanie oddziaływać z innymi obiektami znajdującymi się na scenie jak z samym podłożem sceny. Dodatkowo wyeliminowane zostało negatywne zjawisko przenikania przez siebie nóg. Poprzez to model jest bardziej realistyczny, ponieważ błędne sterowanie może doprowadzić do kolizji pomiędzy nogami lub wykroczenia poza wielomian podparcia, co najprawdopodobniej poskutkuje wywróceniem się robota. Omówione powyżej cechy sprawiają, że użyty model w bardzo dobrym stopniu oddaje rzeczywistość i nadaje się do implementacji metod ruchu.

14.4.2. Implementacja chodu

Poniżej są przedstawione szczegółowe informacje dotyczące implementacji podstawowych sposobów ruchu sześcionożnego robota kroczącego. Opisane tam informacje dotyczą prostokątnej trajektorii końca nogi. Rzeczywiście napisany kod można zobaczyć edytując skrypty zamieszczone w pliku sceny dołączonej do tego dokumentu.

Rysunek 14.14. Model robota sześcionożnego

Generator trajektorii

Generator trajektorii w V-REP został zaimplementowany z wykorzystaniem architektury systemu, opisanej w [2]. Zgodnie z tym V-REP posiada możliwość dołączenia do modelów skryptów w dwóch trybach: wątkowym i bez wątkowym. Skrypty w trybie wątkowym wykonywane są w osobnym wątku. Posiadają część inicjalizacyjną, a następnie są egzekwowane od początku do końca. Natomiast skrypty w trybie bez wątkowym wykonywane są co krok symulacji.

Generator trajektorii został umieszczony w skrypcie bez wątkowym. Co krok symulacji wyliczane są nowe współrzędne dla końców wszystkich nóg. Współrzędne wyliczane są używając trajektorii wielomianowej opisanej wzorem (14.13), który w przypadku dyskretnym przyjmie formę

$$\begin{bmatrix} x(\Delta tn) \\ y(\Delta tn) \\ z(\Delta tn) \end{bmatrix} = \begin{bmatrix} x_0 + 3\frac{\Delta x}{T^2}(\Delta tn)^2 - 2\frac{\Delta x}{T^3}(\Delta tn)^3 \\ y_0 + 3\frac{\Delta y}{T^2}(\Delta tn)^2 - 2\frac{\Delta y}{T^3}(\Delta tn)^3 \\ z_0 + 3\frac{\Delta z}{T^2}(\Delta tn)^2 - 2\frac{\Delta z}{T^3}(\Delta tn)^3 \end{bmatrix}, \tag{14.22}$$

gdzie Δt oznacza krok sterowania oraz $n = 12 \cdots \frac{T}{\Delta t}$. Krok sterowania w naszym wypadku wynosi 50ms. Czas ruchu T musi być tak dobrany aby iloraz $\frac{T}{\Delta t}$ był liczną naturalną. Przesuwanie nóg odbywa się za pomocą funkcji *simSetObjectPosition*, po wykonaniu której kinematyka jest automatycznie wyliczana i następują odpowiednie przesunięcia nóg lub korpusu. Jest to duża zaleta systemu V-REP, który zwalnia programistę z konieczności implementowaniu własnych metod obliczania kinematyki odwrotnej.

Druga część generatora trajektorii znajduję się w skrypcie wątkowym w postaci funkcji *MoveLegs*, zadaniem której jest przesłanie odpowiednich wartości translacji dla nóg

i położeń początkowych przez interfejs parametrów skryptów do skryptu bez wątkowego. Następnie funkcja oczekuje na zgłoszenie wykonania trajektorii przez skrypt bez wątkowy.

Ruch postępowy w wybranym kierunku

Ruch postępowy robota w wybranym kierunku został zaimplementowany za pomocą funkcji *MoveHex*. Funkcja ta przeprowadza robota poprzez zbiór sekwencji opisanych w 14.3.3. Przyjmuje ona 4 argumenty:
1. długość kroku,
2. wysokość kroku,
3. kierunek ruchu,
4. czas ruchu.

Kierunek ruchu określony jest przez kąt w kierunku którego ma się poruszać robot. Czas ruchu określa czas przeprowadzenia nogi przez pełną trajektorię prostokątną. Rysunek 14.15 przedstawia kolejne fazy wykonywania trajektorii prostokątnej. Faza 1 jest początkiem fazy podporowej. Współrzędne tego punktu są wyliczane w następujący sposób

$$X1 = (initX - stepAmplitude/2)\cos(\alpha), \tag{14.23}$$
$$Y1 = (initY - stepAmplitude/2)\sin(\alpha), \tag{14.24}$$

gdzie initX i initY określa współrzędną X i Y końca nogi w fazie 2, a α kierunek ruchu. Zatem w fazie 3, która jest końcem fazy podporowej, współrzędne wyliczane są następująco

$$X3 = (initX + stepAmplitude/2)\cos(\alpha), \tag{14.25}$$
$$Y3 = (initY + stepAmplitude/2)\sin(\alpha). \tag{14.26}$$

Następnie są wyliczane zmiany współrzędnych np.

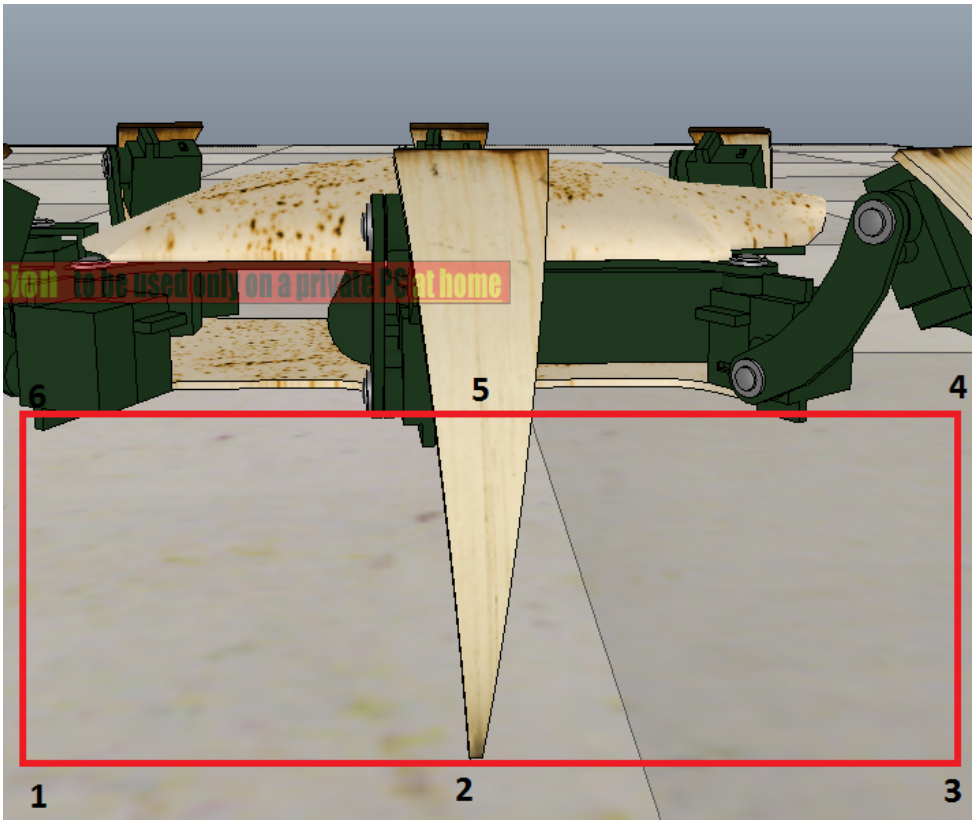
$$\Delta X = X1 - PosX, \tag{14.27}$$
$$\Delta Y = Y1 - PosY, \tag{14.28}$$

gdzie PosX i PosY oznaczają aktualne położenie nogi. Zmiany dla współrzędnych Z odbywa się w podobny sposób. Brana pod uwagę jest różnica pomiędzy podążaną końcową wysokością nogi a aktualną. Po obliczeniu zmiany współrzędnych dla każdej ze współrzędnych wszystkich nóg wywoływana jest funkcja *MoveLegs*, która to przeprowadza nogi przez zadane trajektorie. Następnie przechodzimy do następnej fazy ruchu i powtarzamy wyżej wymienione operacje. Jedno wywołanie funkcji wykonuje jeden okres ruchu, czyli dwie fazy podporowe. Zapętlenie wywołania tej funkcji spowoduje rytmiczny chód maszyny.

Ruch postępowy ze skrętem

Ruch postępowy ze skrętem jest realizowany za pomocą funkcji *MoveRotateHex*, która to przyjmuje 4 argumenty:
1. długość kroku,
2. wysokość kroku,
3. kąt skrętu,
4. czas ruchu.

Sekwencje ruchu są tutaj takie same jak w przypadku funkcji *MoveHex*. Modyfikacją ulegają tylko wartości współrzędnych na początku fazy podporowej, jak to zostało opisane w sekcji 14.3.3. Nowe współrzędne obliczane są za pomocą funkcji *GetXinA* oraz *GetYinA*,



Rysunek 14.15. Fazy trajektorii prostokątnej

które wyliczają współrzędne końca nogi według wzorów (14.18) i (14.19). Współrzędne na końcu fazy podporowej są niezmienne. W tym przypadku zmiana współrzędnych dla początku fazy podporowej jest określona następująco

$$\Delta X = GetXinA - PosX, \tag{14.29}$$
$$\Delta Y = GetYinA - PosY. \tag{14.30}$$

Natomiast dla końca fazy podporowej

$$\Delta X = (initX + stepLength/2) - PosX, \tag{14.31}$$
$$\Delta Y = (initY + stepLength/2) - PosY. \tag{14.32}$$

Po wyliczeniu wszystkich przesunięć wywoływana jest funkcja *MoveLegs* i następuje przejście do kolejnej fazy ruchu. Jedno wywołanie funkcji *MoveRotateHex* realizuje jeden cykl chodu. Zapętlenie tej funkcji spowoduje rytmiczne kroczenie ze skręcaniem. W przypadku podania długości kroku jako 0, nastąpi obrót w miejscu o zadany kąt.

Ruchy korpusu

Ruchy korpusu są realizowane przez funkcję *MoveBodyHex*. Realizuje ona transformację (14.21). Przyjmuje 7 parametrów:

1. kąt rotacji,
2. kąt nutacji,
3. kąt precesji,
4. translacja wzdłuż osi X,
5. translacja wzdłuż osi Y,

6. translacja wzdłuż osi Z,
7. czas ruchu.

Wyliczanie zmiany współrzędnych odbywa się w sposób podany w powyższych sekcjach. Jedno wykonanie funkcji realizuje obrót korpusu o podane kąty oraz przesunięcie o podany wektor translacji.

14.5. Podsumowanie

W modelowaniu sześcionożnych robotów kroczących stosowane są zazwyczaj wyłącznie metody kinematyki opisane w niniejszym rozdziale. Wymagana jest zatem podstawowa wiedza na temat podstawowych macierzy transformacji, notacji Denavita-Hartenberga czy metodach rozwiązywania odwrotnego zadania kinematyki.

Aby móc skutecznie symulować działanie własnego robota w środowisku V-REP potrzebujemy stworzyć wiarygodny model. Proces tworzenia modeli jest bardzo czasochłonny, a dostępny w systemie interfejs niewygodny. Można jednak posłużyć się konstrukcją wykonaną w innym środowisku projektowym i importować taki model bezpośrednio do V-REP. Szczegółowe informacje można znaleźć w rozdziale 5 lub dokumentacji środowiska [2]. System pozwala nam na tworzenie łańcuchów kinematycznych. Możemy przemieszczać końcówki robocze w wybranym przez nas układzie odniesienia. Sterowanie jest możliwe w trybie kinematyki prostej i odwrotnej. Dodatkowo jesteśmy w stanie wybrać odpowiedni algorytm rozwiązywania odwrotnego zadania kinematyki oraz jego dokładność.

W rozdziale 14.4 zostały przedstawione szczegóły implementacyjne modelowania robota sześcionożnego w środowisku V-REP. Użyte tam rozwiązania są odzwierciedleniem teoretycznych rozważań z rozdziału 14.3. Dzięki dobremu modelowi robota udało się uzyskać wiarygodne wyniki symulacyjne. Z powodzeniem udało się zaimplementować podstawowe rodzaje ruchów maszyn kroczących, bazujących wyłącznie na metodach kinematyki.

Model w środowisku V-REP można z powodzeniem rozszerzyć o różnego rodzaju sensory. Najbardziej pożądane są czujniku kontaktu stopy z podłożem. Można to uzyskać poprzez wykorzystanie sensorów nacisku i modyfikację stopy robota. Dodatkowo wyposażenie robota w dalmierze, akcelerometry i żyroskopy pozwoliłoby na implementację i testowanie różnego rodzaju metod planowania ruchu, opisanych w 14.3.4.

Struktura implementacyjna metod ruchu w skryptach V-REP nadaje się, po uprzedniej modyfikacji, do bezpośredniej implementacji tych funkcji w sterowniku rzeczywistego robota. System V-REP może zatem służyć jako środowisko do testowania i ewaulacji metod generowania i planowania ruchów. Popełnienie błędów podczas modelowania nie niesie za sobą poważniejszych konsekwencji, niż gdyby to było w przypadku np. wywrócenia się rzeczywistego robota.

System V-REP dzięki swojej hybrydowej budowie i szerokiej gamie możliwości w pełni sprostał wymaganiom modelowania sześcionożnych robotów kroczących Jest to środowisko godne polecenia przy modelowaniu i symulowaniu różnego rodzaju układów robotycznych.

Bibliografia

[1] B. Siciliano, O. Khatib. *Springer Handbook of Robotics*. Springer, 2008.
[2] Dokumentacja systemu V-REP. `www.v-rep.eu/helpFiles/index.html`.
[3] T. Zielińska. *Maszyny kroczące*. Wydawnictwo naukowe PWN, 2003.

15. Robot latający typu quadrotor

Bartosz Kułak, Marek Gulanowski, Przemysław Synak

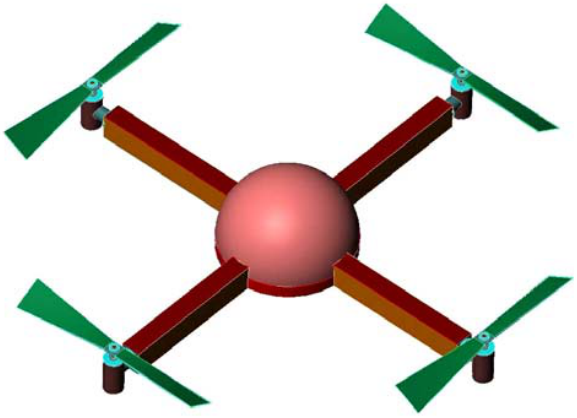
Roboty latające typu quadrotor zdobywają obecnie coraz większą popularność, zarówno w środowisku badaczy, wśród producentów urządzeń elektronicznych, jak i wśród odbiorców komercyjnych. W literaturze rozpatruje się róże modele dynamiki tego typu pojazdu oraz różne strategie i algorytmy sterowania.

Quadrotor jest to helikopter czterośmigłowy, w którym typowo wszystkie śmigła znajdują się w jednej płaszczyźnie, jak przedstawiono na rysunku 15.1. Quadrotor posiada 6 stopni swobody: 3 związane z przemieszczeniem liniowym, 3 – z rotacją. Można zatem określić wektor współrzędnych quadrotora jako

$$q = \begin{pmatrix} x \\ y \\ z \\ \phi \\ \theta \\ \psi \end{pmatrix} = \begin{pmatrix} \xi \\ \eta \end{pmatrix}, \tag{15.1}$$

gdzie $\xi = (x, y, z)^T$ – część translacyjna wektora współrzędnych, $\eta = (\phi, \theta, \psi)^T$ – część rotacyjna.

Zaletą pojazdów typu quadrotor z punktu widzenia implementacji układu sterowania to między innymi fakt, iż jest to najłatwiejsza do sterowania konstrukcja spośród helikopterów, głownie z powodu znajdowania się wszystkich rotorów w jednej płaszczyźnie. Zastosowanie dwóch śmigieł prawo- i dwóch lewoskrętnych (śmigła leżące naprzeciwko siebie obracają się w tym samym kierunku) powoduje wzajemne znoszenie się niepożądanych momentów generowanych przez ich ruch. Natomiast trudności modelowania tego typu układów wynikają z nieliniowej postaci modelu dynamiki – wymaga zastosowania nieliniowego sterownika (co oznacza dużą złożoność obliczeniową wyznaczania sygnałów



Rysunek 15.1. Helikopter typu quadrotor [2]

sterujących) albo wyznaczenia przybliżenia liniowego w punkcie równowagi i zastosowanie liniowego algorytmu sterowania.

W niniejszym rozdziale wykorzystano trzy środowiska komputerowe w celu modelowania takiego układu: `Autodesk Inventor`, `Wolfram Mathematica` oraz `Mathworks` MATLAB. W programie `Inventor` utworzono trójwymiarowy model istniejącej konstrukcji quadrotora, co pozwoliło wyliczyć jego parametry inercyjne. W programach `Mathematica` oraz MATLAB wprowadzono model dynamiki obiektu oraz wykonano symulacje działania układów sterowania.

15.1. Model dynamiki i algorytmy sterowania układu typu quadrotor

W literaturze przyjmuje się różne strategie sterowania i dla nich wybiera się właściwą postać modelu dynamiki. Różnice wynikają głównie z wyboru współrzędnych, dla których ma być realizowane śledzenie trajektorii. W wielu pracach ograniczono się tylko do części rotacyjnej (η), a przedstawione tam modele i algorytmy sterowania nie uwzględniają przemieszczenia liniowego. Może się zatem okazać, że podczas gdy osiągana jest zerowa wartość błędu współrzędnych η, pozostałe współrzędne mają trajektorię rozbieżną. Pomimo tego mankamentu uważa się, iż sama stabilizacja orientacji quadrotora jest zagadnieniem wartym uwagi. W przypadku przyjęcia takiej strategii sterowania model dynamiki układu może być następujący [3]:

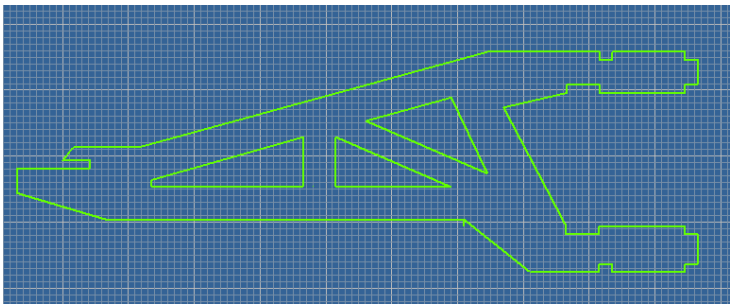
$$I_{xx}\ddot{\phi} = \dot{\theta}\dot{\psi}(I_{yy} - I_{zz}) + J_r\dot{\theta} + u_1, \tag{15.2}$$
$$I_{yy}\ddot{\theta} = \dot{\phi}\dot{\psi}(I_{zz} - I_{xx}) + J_r\dot{\phi} + u_2, \tag{15.3}$$
$$I_{zz}\ddot{\psi} = \dot{\phi}\dot{\theta}(I_{xx} - I_{yy}) + u_3, \tag{15.4}$$

gdzie I_{xx}, I_{yy}, I_{zz} – główne momenty bezwładności pojazdu, J_r – bezwładność silników, $(u_1,\, u_2,\, u_3)^T = u$ – wektor sygnałów sterujących (momentów sił wynikających z siły ciągu silników).

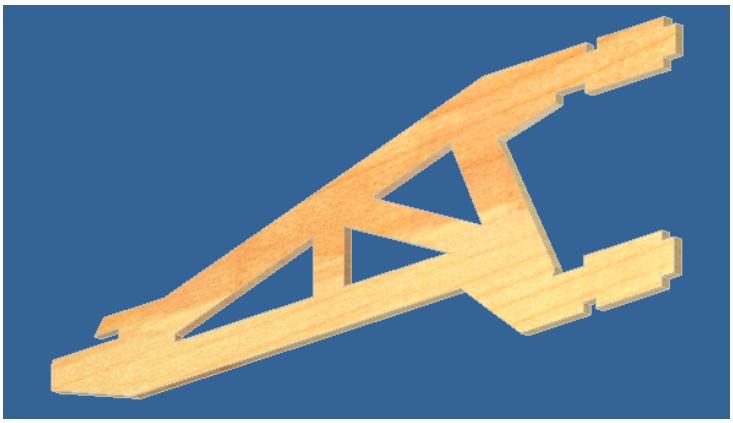
Można także przyjąć inny model, w którym weźmie się pod uwagę wszystkich sześć współrzędnych. Model dynamiki może mieć wtedy następującą postać [1]:

$$m\ddot{x} = u\sin(\theta), \tag{15.5}$$
$$m\ddot{y} = u\cos(\theta)\sin(\phi), \tag{15.6}$$
$$m\ddot{z} = u\cos(\theta)\cos(\phi), \tag{15.7}$$
$$\ddot{\phi} = u1, \tag{15.8}$$
$$\ddot{\theta} = u2, \tag{15.9}$$
$$\ddot{\psi} = u3, \tag{15.10}$$

gdzie u – łączna siła ciągu silników, przyjmowana jako w przybliżeniu stała. Jak widać jest to model znacznie uproszczony w części rotacyjnej, bierze natomiast pod uwagę wpływ orientacji pojazdu na jego prędkości liniowe. Oba ukazane wyżej modele zostały wykorzystane w symulacjach.



Rysunek 15.2. Rysunek techniczny przykładowej części (rama)



Rysunek 15.3. Widok 3D przykładowej części (rama)

15.2. Modelowanie parametrów inercyjnych w programie Inventor

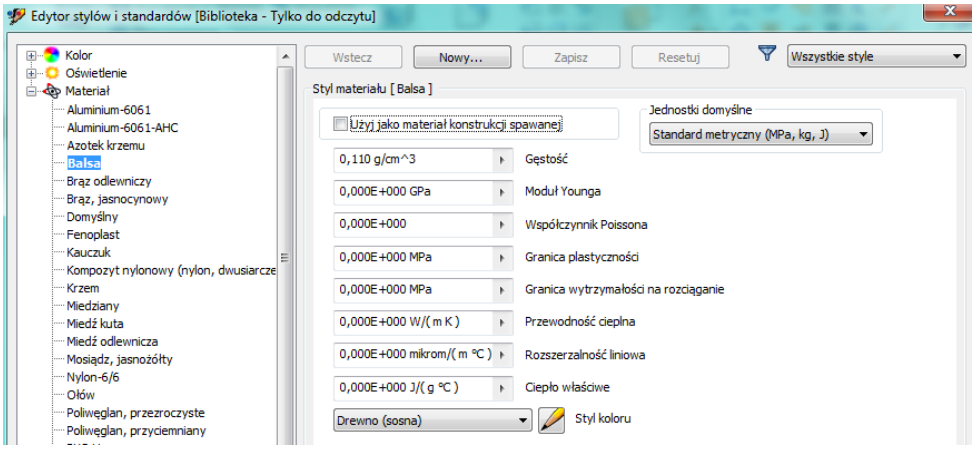
Jednym z postawionych zadań było zaprojektowanie fizycznego robota typu quadrotor i wyznaczenie jego parametrów inercyjnych. Z tego względu wybór padł na oprogramowanie firmy Autodesk, które umożliwia projektowanie robota na poziome pojedynczych części. Złożenie wszystkich części w jeden zespół jest proste i intuicyjne, co znacznie usprawnia pracę. Ostatecznie oprogramowanie potrafi samo wyznaczyć parametry inercyjne zaprojektowanego zespołu. W wykonywanym projekcie założeniem było wyznaczenie macierzy inercji.

15.2.1. Główne założenia

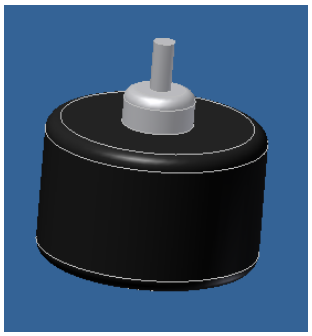
Wykonany quadrotor [4] nie posiada odpowiedniego sterownika, przez co nie może być jeszcze wykorzystywany w badaniach laboratoryjnych. Głównym zadaniem projektowym było przeniesienie robota do programu Autodesk Inventor, co w dalszej perspektywie może umożliwić zaprojektowanie odpowiedniego sterowania.

15.2.2. Projektowanie części

Projekt robota rozpoczęto od stworzenia wszystkich potrzebnych do jego wykonania części, które powstały na podstawie rysunków zawartych w pracy inżynierskiej [4]. Na początku należało wykonać rysunek techniczny każdej części (rys. 15.2), a następnie, przy wykorzystaniu opcji `Wyciągnięcie proste` zostały stworzone elementy (rys. 15.3). Kolejnym krokiem było zadeklarowanie materiału, z którego zostały wykonane części.



Rysunek 15.4. Okno edytora stylów i standardów



Rysunek 15.5. Silnik BLDC

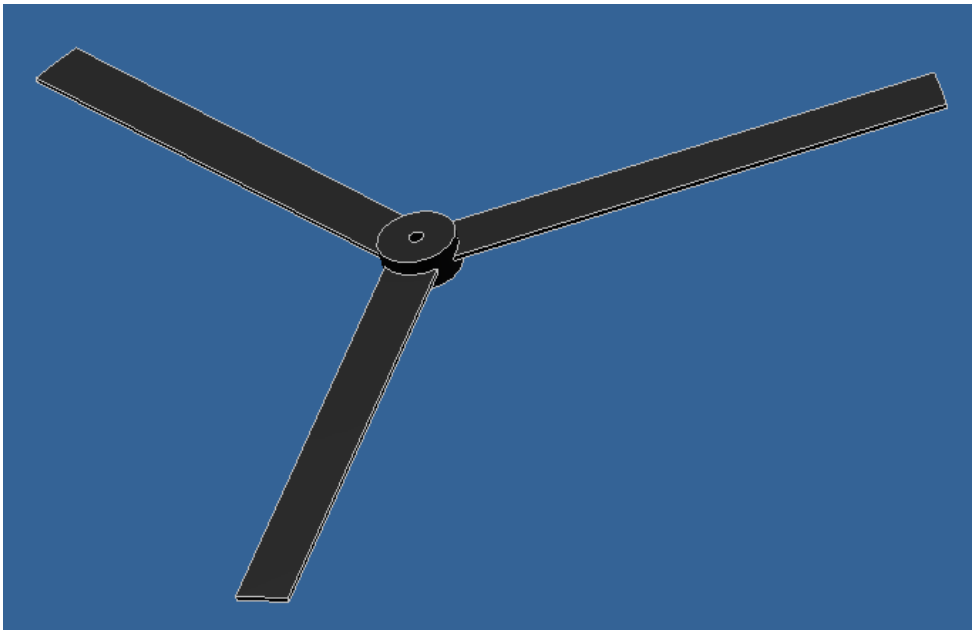
Aby dodać nowy materiał należy otworzyć `Edytor stylów i standardów` (rys. 15.4). Tam możemy dodać nowy materiał, w naszym przypadku `drewno balsowe`, które jest szeroko wykorzystywanym materiałem w modelarstwie ze względu na niską gęstość i stosunkowo dużą wytrzymałość na działające siły. Z punktu widzenia założeń zadania, w `Edytorze stylów i standardów` wystarczy podać gęstość materiału $(0,11\frac{g}{cm^3})$. Dodatkowo został wybrany `styl koloru materiału` przypominający drewno. Tak przygotowane części można wykorzystać w dalszych etapach modelowania robota.

15.2.3. Projektowanie jednostek napędowych

W projektowanym quadrotorze znajdują się cztery jednostki napędowe. Są to silniki BLDC wraz ze sterownikami. Na potrzeby zadania nie było konieczności wiernego odwzorowywania napędów, dlatego zostały one wykonane przy użyciu prostych narzędzi. Każdy silnik składa się z trzech walców o odpowiednich wymiarach i masach oraz wygładzonych krawędziach (rys. 15.5). Zamontowane na wałach silników śmigła (rys. 15.6) zostały wycięte z bryły o kształcie walca. Sterowniki silników to natomiast proste bryły prostopadłościenne z zaokrąglonymi krawędziami.

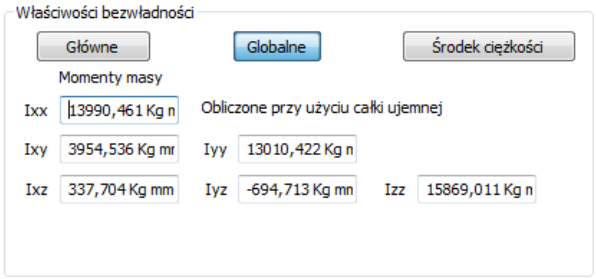
15.2.4. Składanie części

Ze wszystkich zaprojektowanych części został złożony model quadrotora (rys. 15.7). Program Autodesk Inventor pozwala na składanie zespołów składających się z wielu części w prosty i intuicyjny sposób (przez deklarowanie ograniczeń i wiązań). Po złożeniu qu-



Rysunek 15.6. Trójłopatowe śmigło

Rysunek 15.7. Model zaprojektowanego quadrotora



Rysunek 15.8. Parametry inercyjne robota

adrotora możemy odczytać jego parametry inercyjne, które znajdują się we właściwościach fizycznych (rys. 15.8).

15.2.5. Otrzymane wyniki

Głównym założeniem modelowania robota typu quadrotor w środowisku Autodesk Inventor było wyznaczenie parametrów inercyjnych obiektu. Program sam wyznacza macierz bezwładności zaprojektowanego układu, jednak w wersji 2010 użytkownik otrzymuje parametry w nieużywanych jednostkach ($\frac{kg}{mm^2}$), dlatego konieczne jest ich przeliczenie na jednostki zgodne z układem SI ($\frac{kg}{m^2}$). Po przeliczeniu macierz bezwładności ma postać:

$$M(q) = \begin{bmatrix} 13,99 \cdot 10^{-3} & 3,95 \cdot 10^{-3} & 0,34 \cdot 10^{-3} \\ 3,95 \cdot 10^{-3} & 13,10 \cdot 10^{-3} & -0,69 \cdot 10^{-3} \\ 0,34 \cdot 10^{-3} & -0,69 \cdot 10^{-3} & 15,87 \cdot 10^{-3} \end{bmatrix} \tag{15.11}$$

Projekt quadrotora w programie Autodesk Inventor daje także możliwość zbudowania dowolnej liczby rzeczywistych obiektów o identycznych parametrach. W dłuższej perspektywie pozwala to na przeprowadzanie zaawansowanych badań laboratoryjnych.

15.3. Modelowanie dynamiki i implementacja układów sterowania w programie Mathematica

W niniejszym podrozdziale przedstawiono sposób modelowania obiektu typu quadrotor w programie `Mathematica`. Poniższy opis można traktować także jako przedstawienie ogólnej metodologii modelowania dynamiki i symulacji układów sterowania obiektów robotycznych. Poniżej opisano ogólny algorytm postępowania w celu symulacji układu, a następnie szczegółowo omówiono poszczególne jego kroki.

15.3.1. Ogólny schemat modelowania

Symulacja układu sterowania obiektu dynamicznego w systemie `Mathematica` sprowadza się do numerycznego rozwiązania układu równań różniczkowych. Wymiar układu odpowiadać musi wymiarowi obiektu, tj. liczbie współrzędnych, które określają jego stan. Ogólny algorytm postępowania przedstawia się następująco:

1. Wprowadzenie modelu dynamiki obiektu w postaci układu skalarnych równań różniczkowych (przy czym możliwa jest konwersja równań w postaci macierzowej do równań skalarnych).

2. Zdefiniowanie algorytmu sterowania – wyliczania wartości zmiennych sterujących będących wejściem obiektu.
3. Przeprowadzenie symulacji – numeryczne rozwiązanie układu równań różniczkowych.
4. Wizualizacja wyników w formie wykresów lub animacji.

15.3.2. Wprowadzanie modelu dynamiki obiektu

Model dynamiki powinien zostać wprowadzony jako układ równań różniczkowych, które następnie można rozwiązać numerycznie. Taki układ równań można w systemie `Mathematica` w momencie definiowania przypisać do zmiennej, którą można będzie następnie przekazać do funkcji realizującej całkowanie numeryczne. Wydruk 15.1 przedstawia sposób wprowadzenia modelu dynamiki i przypisania go do zmiennej `dynamics`. Warto zwrócić uwagę na to, że wszystkie zmienne, występujące w równaniach dynamiki, które nie są współrzędnymi, dla których równania te będą rozwiązywane, muszą mieć przypisane wartości liczbowe – w przeciwnym przypadku próba ich rozwiązania numerycznego spowoduje błąd. Przypisanie tych wartości może nastąpić zarówno przed wykonaniem kodu z wydruku 15.1, jak i po nim, ale koniecznie przed próbą rozwiązania układu równań.

Wydruk 15.1. Określenie modelu dynamiki quadrotora

```
dynamics={
  ixx phi''[t]==theta'[t]psi'[t](iyy-izz)+jr theta'[t]+u1,
  iyy theta''[t]==phi'[t]psi'[t](izz-ixx)+jr phi'[t]+u2,
  izz psi''[t]==phi'[t]theta'[t](ixx-iyy)+u3
 };
```

15.3.3. Definiowanie sygnałów sterujących

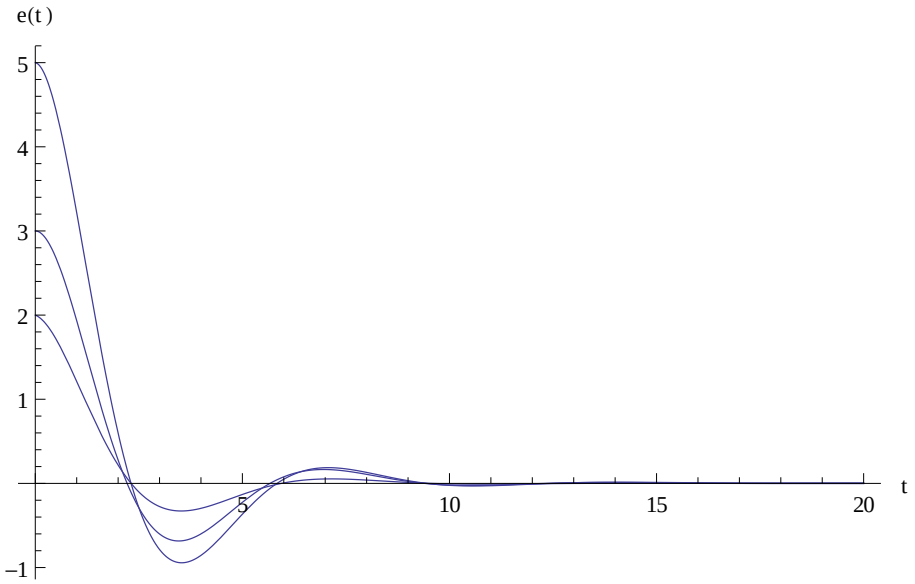
Dla obranego modelu dynamiki należy określić algorytm sterowania, którego działanie ma być symulowane. Tak jak w przypadku równań dynamiki, równania określające sterowanie zapisać możemy do zmiennej w celu późniejszego użycia. Nie definiuje się ich jednak jako dodatkowe równania – wygodnym sposobem jest określenie sterowania w formie reguł (ang. *rules*, za pomocą operatora „`->`"), co przedstawiono na wydruku 15.2. Funkcje `phid[t]`, `thetad[t]`, `psid[t]` oznaczają tu trajektorie zadane: odpowiednio ϕ_d, θ_d oraz ψ_d.

Wydruk 15.2. Określenie sygnałów sterujących

```
controller= {
  u1->p1(phid[t]-phi[t])+d1(phid'[t]-phi'[t]),
  u2->p2(thetad[t]-theta[t])+d2(thetad'[t]-theta'[t]),
  u3->p3(psid[t]-psi[t])+d2(psid'[t]-psi'[t])
  };
```

15.3.4. Przeprowadzanie symulacji

Do wykonania symulacji układu sterowania należy posłużyć się funkcją `NDSolve`. Przed jej wywołaniem należy jednak określić warunki początkowe. Dla każdej współrzędnej liczba warunków początkowych na tę współrzędną i jej kolejne pochodne musi być równa najwyższemu rzędowi pochodnej, jaki występuje w równaniach dynamiki. Deklarację warunków początkowych dla rozważanego przypadku umieszczono na wydruku 15.3.



Rysunek 15.9. Wynik symulacji

Wydruk 15.3. Określenie warunków początkowych dla współrzędnych i ich pochodnych

```
init={
  phi[0]==5,phi'[0]==0,
  theta[0]==4,theta'[0]==0,
  psi[0]==2,psi'[0]==0
  };
```

Kiedy określone zostały już równania dynamiki, algorytm sterowania oraz warunki początkowe, należy przekazać je do funkcji `NDSolve`, która zwróci wektor rozwiązań równań dla każdej współrzędnej. Wywołanie przedstawiono na wydruku 15.4. Równania dynamiki oraz warunki początkowe przekazywane są jako wektor (połączenie dwóch wektorów w jeden za pomocą funkcji `Flatten`), w których pod zmienne sterujące podstawione zostają formuły określone w wektorze reguł `controller` (następuje podstawienie zgodnie z zadeklarowanymi regułami za sprawą operatora „`/.`”).

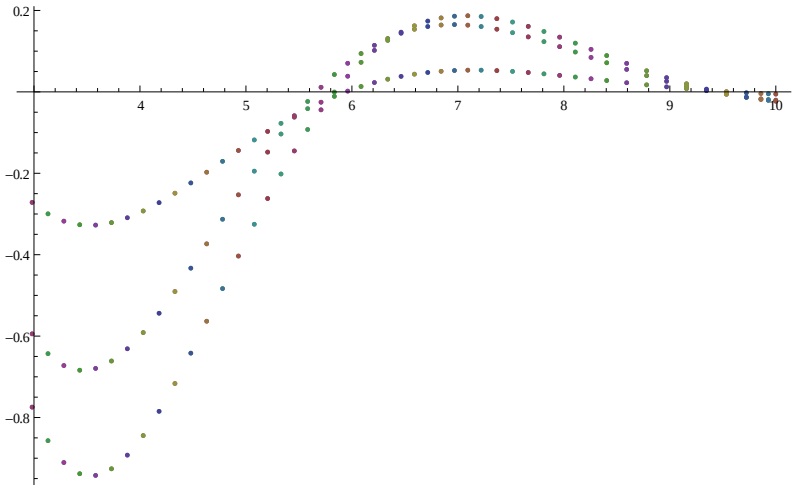
Wydruk 15.4. Wywołanie funkcji NDSolve

```
tmax=10;
solpd=NDSolve[Flatten[{dynamics,init}/.controller]
  ,{phi,theta,psi},{t,0,tmax},MaxSteps->1000000];
```

15.3.5. Wizualizacja wyników

Do wizualizacji wyników symulacji wykorzystać należy funkcję `Plot`. Wywołanie tej funkcji przedstawiono na wydruku 15.5, natomiast wygenerowany wykres na rysunku 15.9. Wykres ten przedstawia przebiegi sygnału błędu:

$$e = \begin{pmatrix} e_1 \\ e_2 \\ e_3 \end{pmatrix}^T = \begin{pmatrix} \phi_d - \phi \\ \theta_d - \theta \\ \psi_d - \psi \end{pmatrix}. \tag{15.12}$$



Rysunek 15.10. Wizualizacja ostatnich wyliczonych wartości w trakcie symulacji

Wydruk 15.5. Wizualizacja wyników symulacji

```
Plot[e[t]/.solpd,{t,0,tmax},PlotRange->All,
  AxesLabel->{"t","e(t)"}]
```

15.3.6. Postępowanie w przypadku błędów

Często zdarza się, że wynik uzyskany po uruchomieniu symulacji w powyższy sposób nie jest zgodny z oczekiwaniami. Może pojawić się kilka rodzajów błędów.

1. Funkcja `NDSolve` zgłasza, że przekazany układ równań nie jest prawidłowo określonym układem równań różniczkowych. W tym przypadku należy upewnić się, czy nie popełniono błędów składniowych (zdarza się np. postawienie przecinka po ostatnim elemencie wektora, co powoduje dołączenie wartości `Null`).
2. Funkcja `NDSolve` zgłasza, że wystąpiły parametry nieposiadające wartości liczbowej (ang. *non-numeric values*) – należy upewnić się, że zostają im przypisane konkretne wartości.
3. Układ jest niestabilny pomimo istnienia dowodu zbieżności. Taka sytuacja najczęściej wynika z błędnego wprowadzenia równań dynamiki bądź algorytmu sterowania.

W przypadku niestabilności systemu czas symulacji często staje się bardzo długi, co jest kłopotliwe w przypadku gdy istotne jest szybkie otrzymanie wyniku. Przydatnym sposobem, aby szybko stwierdzić, czy zachowanie obiektu jest zgodne z pożądanym, jest obserwacja wyników symulacji w jej trakcie, a nie dopiero po jej zakończeniu. Jest to możliwe dzięki zastosowaniu funkcji `Monitor`, co zaprezentowano na wydruku 15.6. Przykładowy wynik działania kodu przedstawiono na rysunku 15.10. Zaprezentowany kod powoduje przedstawianie na wykresie ostatnich 100 próbek dla każdej symulowanej współrzędnej w miarę ich wyliczania. Można w ten sposób wcześnie zdiagnozować niewłaściwe działanie układu sterowania i zatrzymać symulację.

Wydruk 15.6. Wizualizacja wyników symulacji na bieżąco

```
tmax=10;
values={};
Monitor[solpd=NDSolve[Flatten[{dynamics,init}/.controller],
  {\[Phi],\[Theta],\[Psi]},{t,0,tmax},MaxSteps->1000000,
  EvaluationMonitor:>(AppendTo[values,{t,#}&/@e[t]];
```

```
6     Pause[0.01];)],
    Quiet@ListPlot[values[[
8     If[Length[values]>=100,-100,-Length[values]];;]],
  PlotRange->All]]
```

15.4. Modelowanie dynamiki i implementacja układów sterowania w programie MATLAB

W tym podrozdziale przedstawiony zostanie sposób wykorzystania środowiska MATLAB w celu syntezy sterownika oraz badań symulacyjnych modelu quadrotora. Główną zaletą tego środowiska jest bardzo rozbudowana biblioteka specjalistycznych funkcji, pogrupowanych ze względu na zastosowanie w poszczególne „toolboxy". Jak zostanie to pokazane poniżej, wykorzystanie poszczególnych „toolboxów" pozwala na szybką i sprawną syntezę sterowników dla modelowanego obiektu (jakim w tym wypadku jest quadrotor).

15.4.1. Model dynamiki quadrotora w środowisku MATLAB

Do symulacji wykorzystany został model dynamiki przedstawiony w równaniach (15.5), natomiast wartości parametryczne zostały pobrane ze środowiska **AutoDesk Inventor** gdzie został zaimplementowany model struktury quadrotora (macierz bezwładności (15.11)).

Równania stanu obiektu mogą być zapisane w środowisku MATLAB za pomocą struktury „state-space". Pozwala ona na wygodny (w późniejszym etapie syntezy sterownika) i jednoznaczny zapis równań opisujących model i ma następującą postać:

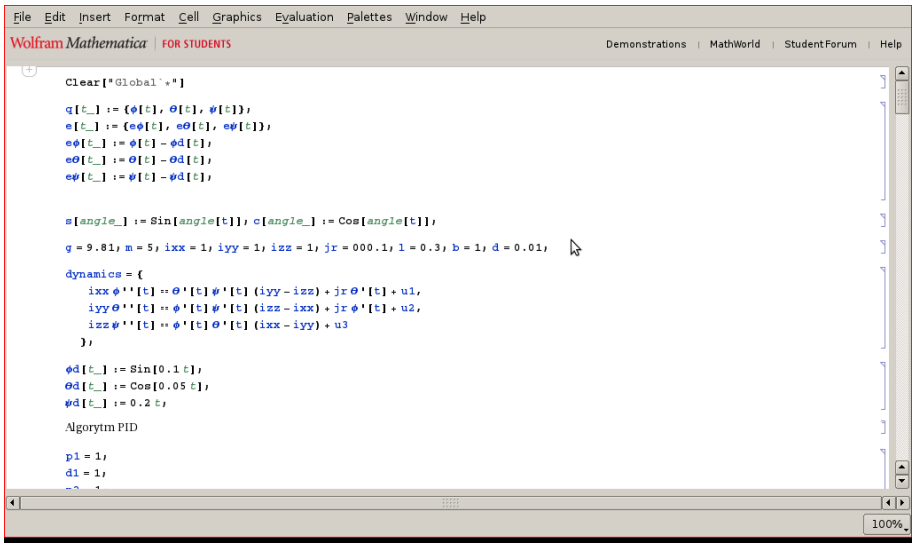
$$\dot{q} = Aq + Bu, \tag{15.13}$$

$$y = Cq + Du. \tag{15.14}$$

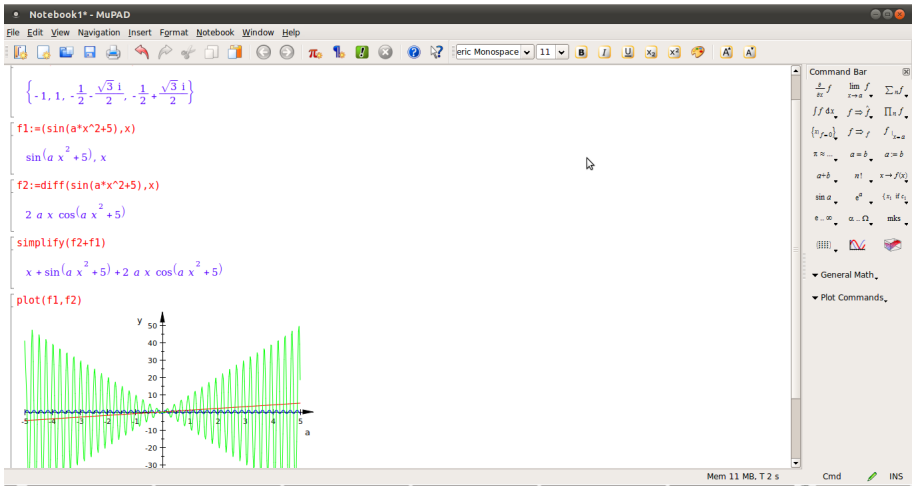
Macierze A i B to zlinearyzowane równania stanu (15.5) w pobliżu punktów pracy. Sam proces linearyzacji wokół punktów pracy powinien być przeprowadzony w środowisku do obliczeń symbolicznych. MATLAB mimo faktu bycia środowiskiem wykorzystywanym głównie do obliczeń numerycznych dysponuje specjalistycznym toolboxem („Symbolic Matlab Toolbox") wykorzystywanym do obliczeń symbolicznych. Dzięki tej bibliotece użytkownik wyposażony jest w dynamiczny notatnik[1] wyglądem i użytecznością bardzo zbliżony do znanego z programu **Mathematica**. Na rysunkach 15.11 oraz 15.12 zostały przedstawione GUI w obydwu aplikacjach. Obliczenie macierzy A i B może zostać wykonane w środowisku MATLABMUPAD funkcji *diff* oraz *solve* których wywołania w tym wypadku będą miały postać ($r1$, $r2$ oraz $r3$ to równania (15.2)):

```
r1:= tt  *pp *(I_y-I_z)+J_r*tt +u1
r2:=ff*pp*(I_z-I_x)+J_r*ff+u2
r3:=ff*tt*(I_x-I_y)+u3
A:=matrix([[diff(r1,ff),diff(r1,tt),diff(r1,pp)],[diff(r2,ff), ...
   diff(r2,tt),diff(r2,pp)],[diff(r3,ff),diff(r3,tt),diff(r3,pp)]])
B:=matrix([[diff(r1,u1),diff(r1,u2),diff(r1,u3)],[diff(r2,u1), ...
   diff(r2,u2),diff(r2,u3)],[diff(r3,u1),diff(r3,u2),diff(r3,u3)]])
solve([r1=0,r2=0,r3=0],[ff,tt,pp]).
```

[1] W celu zainicjowania pracy w tym programie należy wywołać komendę „mupad".



Rysunek 15.11. Widok na notatnik użytkownika w środowisku **Mathematica**



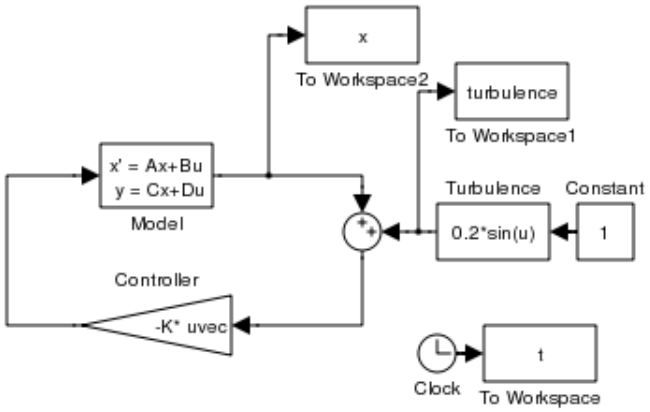
Rysunek 15.12. Widok na notatnik użytkownika do obliczeń symbolicznych w środowisku MATLAB

Macierz C jest macierzą jednostkową, natomiast macierz „feedforward” D została przyjęta jako 0.

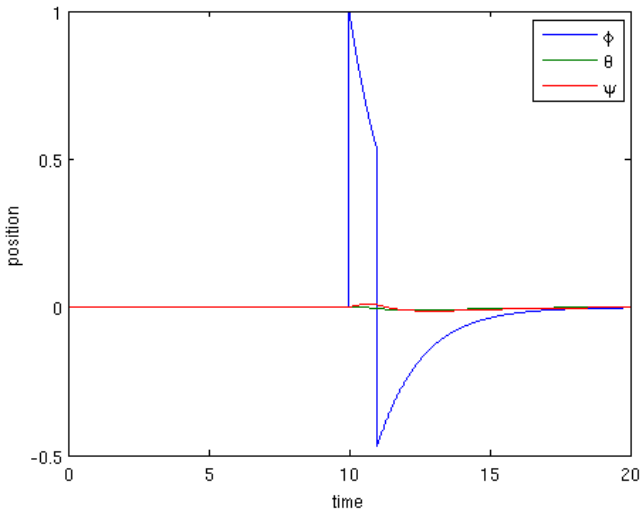
15.4.2. Synteza sterowników w oparciu o metody rozmieszczania biegunów i kryterium minimalno-kwadratowe

Kolejnym krokiem jest synteza sterowników dla układu zapisanego w postaci równań stanu zlinearyzowanych wokół punktu pracy. W przypadku sterownika opartego na metodzie rozmieszczania biegunów, sterownika minimalno-kwadratowego schemat układu w **Simulink** został przedstawiony na rysunku 15.13. W obydwu przypadkach należy również wyliczyć odpowiednią macierz wzmocnień sprzężenia zwrotnego K, która to pozwoli na stabilizację układu wokół punktu równowagi. W tym celu należy wykorzystać odpowiednio następujące polecenia

```
>> Kpp=place(A,B,P)
```



Rysunek 15.13. Schemat układu w Simulinku



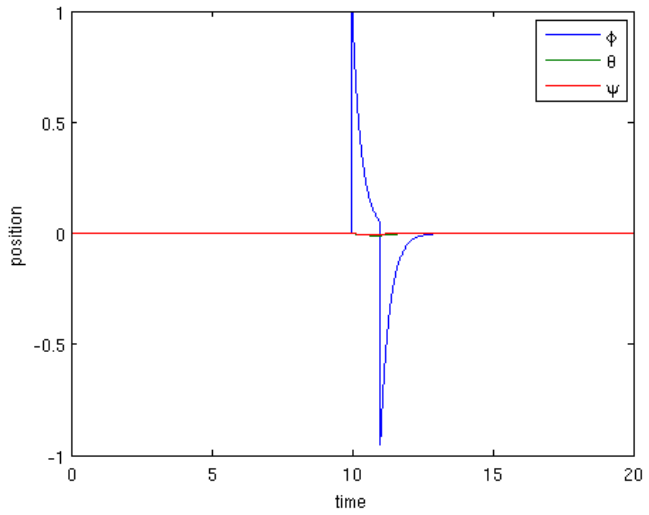
Rysunek 15.14. Zachowanie się modelu ze sterownikiem LQ podczas wytrącenia z punktu równowagi

```
>> model=ss(A,B,C,D)
>> Klq=lqr(system,Q,R)
```

gdzie P jest wektorem docelowych biegunów systemu wybranych przez projektanta, natomiast macierze Q i R są parametrami sterownika LQ. Na wykresach 15.14 oraz 15.15 przedstawiono odpowiedzi obiektów z sterownikami na wytrącenie (zaburzenie) z punktu równowagi.

15.4.3. Wykorzystanie Model Predictive Control Toolbox oraz Aerospace Blockset

Główną i niewątpliwą zaletą środowiska MATLAB jest bogaty zestaw specjalistycznych funkcji i skryptów. W poprzedniej sekcji został przedstawiony dość prosty sposób na reprezentację modelu quadrotora, jego symulację i syntezę sterownika z zastosowaniem pewnych ogólnych metod. W tej sekcji zostanie przedstawiony sposób wykorzystania bardziej zaawansowanych narzędzi dostępnych w środowisku MATLAB.



Rysunek 15.15. Zachowanie się modelu ze sterownikiem PP podczas wytrącenia z punktu równowagi

Model Predictive Control udostępnia użytkownikowi bardzo zaawansowany blok służący do projektowania sterownika opartego na metodzie sterowania predykcyjnego, które to jest szeroko wykorzystywane w dziedzinie sterowania obiektami latającymi.

Aerospace Blockset dostarcza wygodny sposób reprezentacji modelu w zależności od wybranego formalizmu oraz ilości stopni swobody. Użytkownik może wybierać pomiędzy zapisem przy użyciu kątów Eulera lub kwaternionów. Obiekt może posiadać 3 bądź 6 stopni swobody. Rysunek 15.16 przedstawia część struktur danych (bloków simulinka) z których użytkownik może wybrać najbardziej mu odpowiadający.

Na schemacie 15.17 został przedstawiony układ wykorzystujący elementy z obydwu wcześniej wspomnianych modułów. Podsystemy $linear_{acc}$ oraz $angular_{acc}$ reprezentują równania (15.5) zapisane przy użyciu bloków funkcyjnych środowiska MATLAB Simulink.

15.5. Ocena użyteczności opisanych środowisk do modelowania

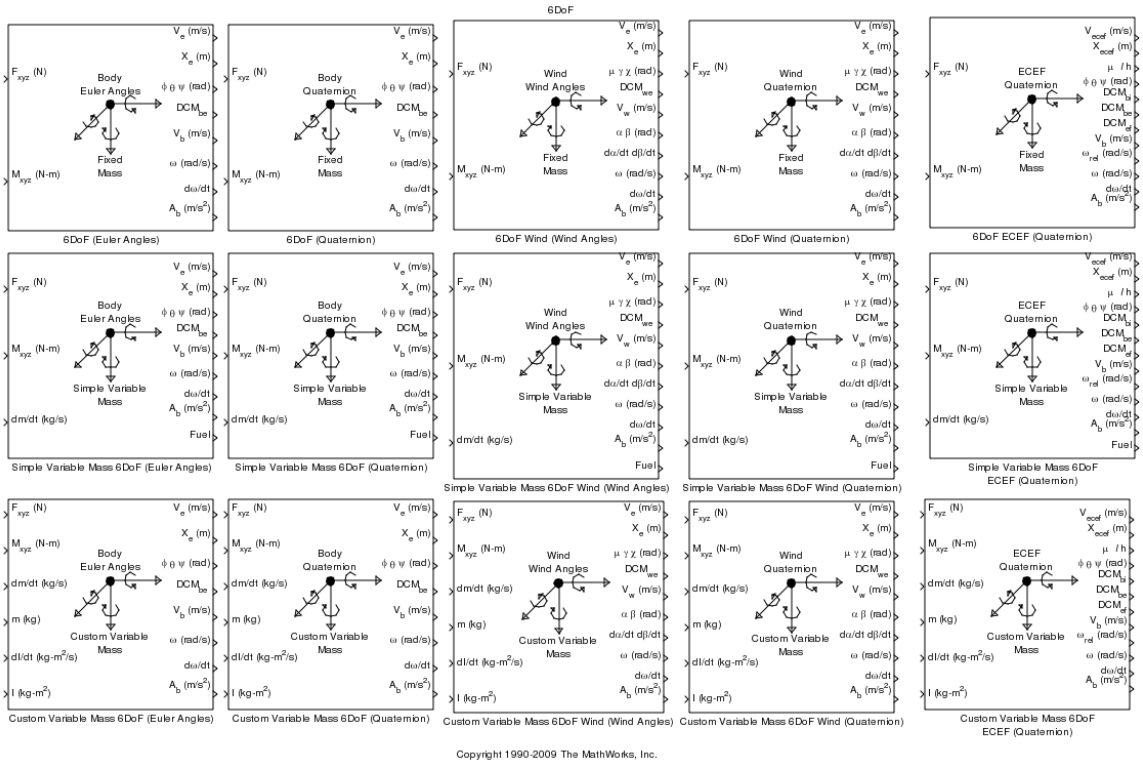
Poniżej przedstawiono uwagi, sformułowane na podstawie doświadczenia w pracy z opisywanym oprogramowaniem podczas modelowania wyżej omówionych układów. Rozpatrzono na ile dany program może być użyteczny w modelowaniu układów robotycznych oraz jakich kompetencji i nakładu pracy wymaga od użytkownika.

15.5.1. Autodesk Inventor

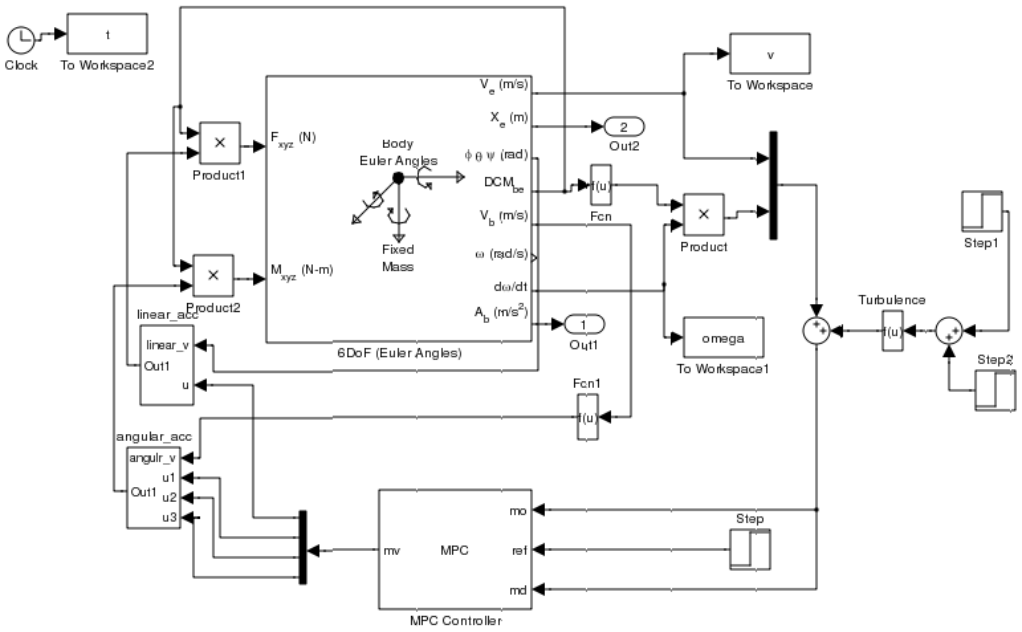
Oprogramowanie firmy Autodesk jest środowiskiem projektowym i z założenia pozwala na modelowanie fizyczne. Zaprojektowane części można łączyć w zespoły, a za pomocą wbudowanych w program narzędzi, sprawdzić, czy projekt jest poprawny.

Obsługa programu jest łatwa i intuicyjna, początkujący użytkownik nie powinien mieć z nią większych problemów. Zawarte w nim narzędzia ułatwiają tworzenie złożonych projektów, a z poszczególnych części można wygenerować rysunki techniczne, które są niezbędne do poprawnego wykonania fizycznego robota.

Przy projektowaniu quadrotora w środowisku Autodesk Inventor zostały postawione konkretne cele, które zostały w pełni zrealizowane. Program Autodesk Inventor jest ideal-



Rysunek 15.16. Menu wyboru bloków reprezentujących obiekt latający



Rysunek 15.17. Układ model-sterownik wykorzystujący bloki z **Model Predictive Control** oraz **Aerospace Blockset**

nym narzędziem do wyznaczania fizycznych parametrów obiektów (macierz inercji, środek ciężkości, itp.). Możliwości programu na tym się niestety kończą, ale otrzymane wyniki mogą być z powodzeniem wykorzystywane do dalszych badań w innych środowiskach.

15.5.2. MATLAB i Mathematica

Zarówno MATLAB, jak i **Mathematica** są bardzo rozbudowanymi narzędziami o szerokim zastosowaniu, które w pewnym stopniu pokrywa się. W obu środowiskach można z powodzeniem przeprowadzić symulacje układów regulacji nieliniowych obiektów dynamicznych takich jak quadrotor. Oba programy zapewniają różne algorytmy numerycznego rozwiązywania równań różniczkowych o parametrach, na które można wpływać, co umożliwia strojenie symulacji (jeśli jest to potrzebne) lub też automatyczny dobór algorytmów i parametrów. Istotną różnicą jest natomiast dostępność wyspecjalizowanych pakietów, wspierających modelowanie konkretnych klas obiektów. Istnieje wiele takich pakietów dla środowiska MATLAB, natomiast nie są one zapewnione dla programu **Mathematica**. Co więcej, MATLAB posiada duże możliwości w zakresie tzw. szybkiego prototypowania i generowania kodu dla mikrokontrolerów. MATLAB zdaje się więc bardziej przydatny w kontekście późniejszej implementacji symulowanych układów w rzeczywistych urządzeniach. Kolejną zaletą środowiska MATLAB jest bardzo bogaty zbiór różnych bibliotek funkcyjnych. Jak to zostało przedstawione we wcześniejszych rozdziałach, możliwe jest wykorzystanie dość prostych struktur danych w celu przeprowadzania symulacji, jak i użycie specjalistycznych modułów oraz struktur danych (które to są nietrywialne w przypadku implementacji ich po raz pierwszy). Ostatnim istotnym faktem jest obecność środowiska do obliczeń symbolicznych MATLABMUPAD, które to mimo tego, że ustępuję pod względem rozbudowania i użyteczności programowi firmy Wolfram może być śmiało wykorzystywane do ograniczonych obliczeń i symulacji symbolicznych. Zaletą środowiska **Mathematica** jest natomiast pełen aparat obliczeń symbolicznych oraz elastyczna składnia, zawierająca m.in. elementy programowania funkcyjnego [5].

Należy zauważyć, że praca w każdym z tych środowisk wymaga uprzedniego zapoznania się z ich składnią algorytmami postępowania, co jest pewnym utrudnieniem ze względu na dużą złożoność i szeroką funkcjonalność obu programów. Zatem modelowanie układów robotycznych wymaga wstępnej wiedzy w zakresie obsługi programu, a także przygotowania w zakresie modelowania dynamiki i teorii sterowania.

Nakład pracy potrzebny do wykonania symulacji zależy od konkretnego układu i przyjętego podejścia. W przypadku posiadania gotowych równań dynamiki i algorytmu sterowania, można dość szybko dokonać symulacji działania układu poprzez wprowadzenie równań w programie **Mathematica**, także jeśli wymagane są matematyczne przekształcenia równań w postaci symbolicznej. Wymaga to wtedy oczywiście dodatkowej pracy i być może użycia zaawansowanej składni. W przypadku posiadania schematu układu w postaci schematu blokowego najwygodniejszym sposobem symulacji (o ile schemat ten nie jest zbyt rozbudowany) jest wprowadzenie go w środowisku **Simulink** programu MATLAB.

Bibliografia

[1] P. Castillo, R. Lozano, A. Dzul. Stabilization of a mini-rotorcraft having four rotors. *Intelligent Robots and Systems, 2004. (IROS 2004). Proceedings. 2004 IEEE/RSJ International Conference on*, wolumen 3, strony 2693 – 2698, 2004.

[2] M. Chen, M. Huzmezan. A combined MBPC/2DOF H-Infinity controller for a quad rotor UAV. *Proceedings AIAA Guidance, Navigation, and Control Conference*, Austin, TX, 2003.

[3] I. Dikmen, A. Arisoy, H. Temeltas. Attitude control of a quadrotor. *Recent Advances in Space Technologies, 2009. RAST '09. 4th International Conference on*, strony 722 –727, 2009.
[4] W. Kotowicz. Stabilizacja lotu robota typu quadrotor. 2010. Praca inżynierska.
[5] Wolfram Mathematica 8.