Politechnika Poznańska

Wydział Automatyki, Robotyki i Elektrotechniki

Instytut Robotyki i Inteligencji Maszynowej

Zakład Sterowania i Elektroniki Przemysłowej

Praca Dyplomowa

Zastosowanie metod uczenia maszynowego do sterowania obiektem typu Aeropendulum (projekt zespołowy)

Michał Rojewski

Mikołaj Maciejewski

Marcin Sypniewski

Promotor:

dr hab. inż. Tomasz Pajchrowski

27.01.2022

Spis treści

1 Podstawy teoretyczne **6**
- 1.1 Uczenie przez wzmacnianie . . . 6
- 1.2 Sztuczne Sieci Neuronowe . . . 7
- 1.3 Proces Decyzyjny Markova . . . 8
- 1.4 Eksploracja a eksploatacja . . . 8
- 1.5 Podział algorytmów agentów . . . 9
  - 1.5.1 PG Agent (Policy Gradient) . . . 9
  - 1.5.2 PPO Agent (Proximal Policy Optimization) . . . 10
  - 1.5.3 DQN Agent (Deep Q-Network) . . . 11

2 Model matematyczny Aeropendulum **12**
- 2.1 Parametry modelu . . . 12
- 2.2 Równanie różniczkowe . . . 12

3 Wyniki badań symulacyjnych **14**
- 3.1 Przygotowanie wstępne . . . 14
- 3.2 Przygotowanie modelu symulacji . . . 14
  - 3.2.1 Środowisko . . . 15
  - 3.2.2 Nagroda . . . 17
  - 3.2.3 Obserwacje . . . 17
  - 3.2.4 Sygnał referencyjny . . . 18
- 3.3 Wyniki symulacji . . . 20
  - 3.3.1 Algorytm PPO . . . 20
  - 3.3.2 Algorytm DQN . . . 22
  - 3.3.3 Algorytm PG . . . 22
- 3.4 Wpływ parametrów na wynik regulacji . . . 24
  - 3.4.1 Sposób sterowania akcjami . . . 24
  - 3.4.2 Liczba i zakres akcji . . . 24

4 Wybór najlepszego algorytmu **25**
- 4.1 Proces działania algorytmu . . . 25
- 4.2 Skrypt algorytmu agenta . . . 26
  - 4.2.1 Wykorzystane sieci neuronowe . . . 27

5 Komunikacja **29**
- 5.1 Sprzęt fizyczny . . . 29
  - 5.1.1 Raspberry Pi . . . 29
  - 5.1.2 STEVAL-SPIN 3201 . . . 30
  - 5.1.3 Enkoder Magnetyczny AS5600 . . . 31

5.2 I^2C . . . 31
5.3 UDP . . . 32
5.4 TCP . . . 32
5.4.1 Three-way Handshake . . . 33
5.5 Implementacja . . . 34
5.5.1 Założenia UDP . . . 34
5.5.2 Wyniki testów UDP . . . 35
5.5.3 Założenia TCP . . . 39
5.5.4 Wyniki testów TCP dla Wi-Fi . . . 39
5.5.5 Programowa . . . 41
5.6 Podsumowanie rozdziału . . . 47

6 Praca ze stanowiskiem rzeczywistym 51
6.1 Wstęp . . . 51
6.2 Biblioteki zastosowane w skrypcie . . . 51
6.2.1 Tensorforce . . . 51
6.2.2 Gym . . . 51
6.3 Model Aeropendulum jako środowiska . . . 52
6.3.1 Inicjalizacja klasy . . . 52
6.3.2 Funkcja reset() . . . 53
6.3.3 Funkcja response() . . . 54
6.3.4 Funkcja reward_compute() . . . 56
6.3.5 Funkcja execute() . . . 57
6.4 Ewaluacja modelu . . . 58
6.4.1 Sygnał testowy . . . 58
6.4.2 Odpowiedz układów . . . 59
6.5 Skrypt uczenia ze wzmacnianiem . . . 60
6.5.1 Skrypt z tworzeniem nowego agenta . . . 60
6.5.2 Eksportowanie sieci neuronowej agenta . . . 62
6.6 Wyniki testów skryptu Python . . . 63

7 Podsumowanie pracy 67

Streszczenie

Praca polega na zastosowaniu jednej z metod uczenia maszynowego - uczenia przez wzmacnianie, do sterowania obiektem typu Aeropendulum. Aby wybrać najlepszy algorytm uczenia przez wzmacnianie zostało przeprowadzone wiele testów symulacyjnych w programach *Matlab* oraz *Simulink* sprawdzających skuteczność wybranych algorytmów, ich parametrów oraz wykorzystywanych sieci neuronowych w rozpatrywanym przez nas środowisku. Najlepszy algorytm z dającymi najlepsze wyniki parametrami i sieciami neuronowymi został zaimplementowany w języku *Python* aby przeprowadzić całą procedurę nauczania oraz test funkcjonalności. Aby zapewnić sprawną obustronną komunikację między skryptem nauczania maszynowego oraz stanowiskiem został stworzony kod oparty o protokół UDP i TCP.

Słowa kluczowe: Aeropendulum, uczenie maszynowe, uczenie przez wzmacnianie, UDP, TCP, Python, Matlab, Simulink, sterowanie.

Abstract

Title: Application of machine learning methods to the control of an Aeropendulum type object (team project).

The work consisted of using one of machine learning method's - reinforcement learning to control an Aeropendulum object. To achive the best reinforcement learning algorithm we ran series of simulations in *Matlab* and *Simulink* to determine the efficiency of selected algorithms, their parameters and used neural networks in the environment considered by us. The best algorithm, with most suitable parameters and neural networks was implemented using *Python* to recreate the whole learning process and to test it's functionality. To ensure quick and bilateral communication the code was made based on the UDP and TCP protocol.

Key words: Aeropendulum, machine learning, reinforcement learning, UDP, TCP, Python, Matlab, Simulink, control.

Wstęp

W pracy poruszamy zagadnienie uczenia przez wzmacnianie, jednej z metod nauczania maszynowego, do sterowania obiektem typu Aeropendulum. Do zrealizowania tego zagadnienia przygotowaliśmy model w programie *Matlab* oraz *Simulink* do symulacyjnego przetestowania algorytmów nauczania przez wzmacnianie. Najlepszy algorytm został zaimplementowany korzystając z języka *Python* w celu przetestowania wyników symulacji. Napisaliśmy także skrypty komunikacji aby umożliwić przesył informacji oraz przetestować nauczony algorytm sterowania w programie Matlab do interakcji ze stanowiskiem rzeczywistym.
W pierwszym rozdziale przedstawiamy informacje teoretyczne, które są podstawą wykorzystywanych przez nas zagadnień takich jak uczenie przez wzmacnianie czy sposoby działania wybranych przez nas algorytmów. Kolejny rozdział opisuje model matematyczny Aeropendulum w postaci równania różniczkowego, które zostało zaimplementowane w naszym modelu symulacyjnym. Trzeci rozdział został poświęcony części symulacyjnej. Opisuje budowę modelu symulacyjnego i objaśnia zastosowanie poszczególnych jego części. Na koniec tego rozdziału przedstawiamy wyniki przeprowadzonych testów dla wszystkich trzech algorytmów w celu porównania ich działania. Następny rozdział został poświęcony wybraniu najlepszego algorytmu na podstawie zebranych wyników i przedstawieniu dokładnego działania algorytmu. Piąty rozdział pokazuje za co odpowiedzialna jest komunikacja, w jaki sposób planujemy użyć wytrenowanego agenta w środowisku Matlab jako regulator, zajmiemy się zbadaniem i testowaniem potencjalnych rozwiązań mających doprowadzić do jak najlepszego użycia stworzonej w części symulacyjnej funkcji polityki dla realnych obiektów, w szczególności naszego Aeropendulum. Następny rozdział przedstawia jak zaimplementowaliśmy wiedzę uzyskaną z części symulacyjnej do pracy ze stanowiskiem rzeczywistym. Przedstawiamy napisane skrypty potrzebne do sterowania. Obejmują one zaimplementowanie środowiska, stworzenie agenta oraz przeprowadzony proces nauki. Kolejną częścią jest zbiór rozwiązań, które zostały zastosowane do przetestowania nauczonej sieci neuronowej najlepszego algorytmu. Ostatni rozdział to podsumowanie naszej całej pracy, wnioski jakie mogliśmy wyciągnąć podczas realizacji całego projektu oraz pomysły dalszego rozwoju.

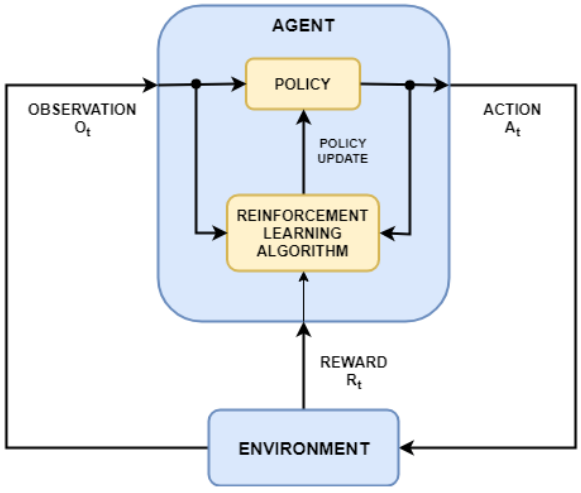
Rozdział 1

Podstawy teoretyczne

Autor: Marcin Sypniewski, Michał Rojewski, Mikołaj Maciejewski

1.1 Uczenie przez wzmacnianie

Uczenie przez wzmacnianie (w skrócie RL) można przetłumaczyć na *Reinforcement learning* i jest jednym z trzech głównych podejść uczenia maszynowego. Uczenie przez wzmacnianie w odróżnieniu od uczenia nadzorowanego czy nienadzorowanego nie wykorzystuje wcześniej przygotowanego zbioru danych lecz dokładnie opisane środowisko (ang. *environment*), czyli układ na który oddziałujemy akcjami (ang. *action*). Akcje powodują zmianę stanu środowiska i są generowane przez inteligentnego agenta (ang. *agent*), który na podstawie obserwacji (ang. *observations*) i nagrody (ang. *reward*) uczy się dobierać najlepsze akcje. Agent składa się z algorytmu uczącego (ang. *reinforcement learning algorithm*), który na podstawie nagrody, obserwacji i akcji w aktualnym stanie aktualizuje politykę (ang. *policy*) odpowiedzialną za dobieranie akcji. Obserwacje to informacje zebrane z aktualnego stanu w którym znajduje się nasze środowisko a nagrodą to sygnał skalarny który, określa jak dobrze zostały dobrane akcje. Agent dąży do zebrania jak największej nagrody w całym jednym epizodzie (ang. *episode*), czyli jednej iteracji całego procesu nauczania. [27].



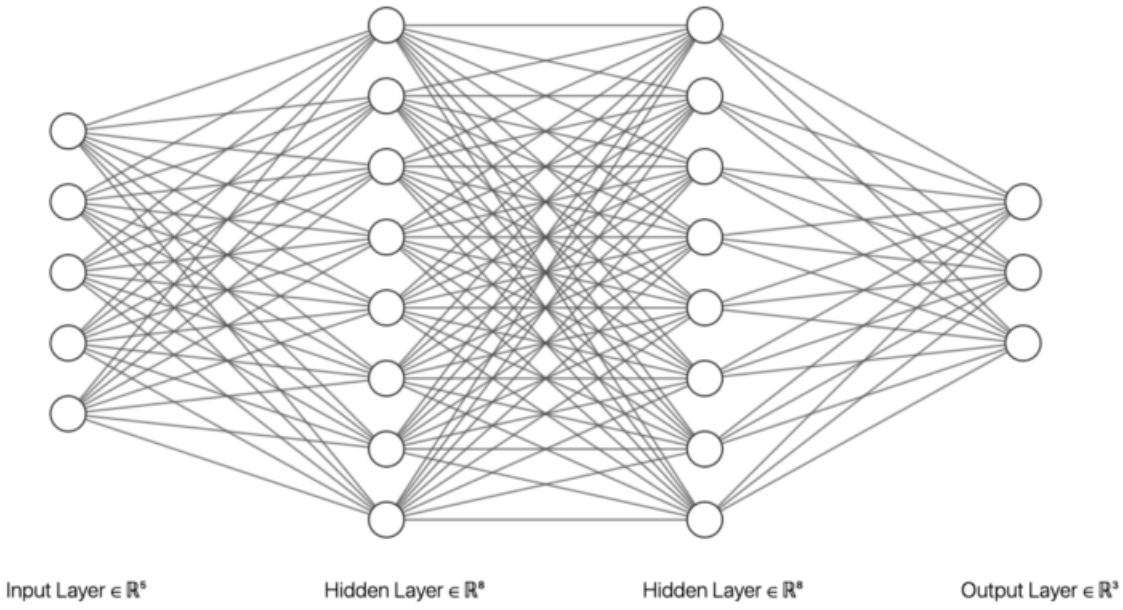
Rysunek 1.1. Diagram działania RL, źródło: www.mathworks.com

1.2 Sztuczne Sieci Neuronowe

Podczas nauczania przez wzmacnianie za aktualizację polityki (ang. policy) odpowiada zwykle sieć neuronowa (w postaci aktora i krytyka). *Sztuczna sieć neuronowa* to zbiór warstw złożonych z węzłów wzorowanych na cechach naturalnych komórek nerwowych [22], wszystkie neurony są ze sobą połączone w różny sposób i mogą przekazywać między sobą informacje. Możemy wyróżnić 3 rodzaje warstw

- Warstwa wejściowa (ang. *Input Layer*)
- Warstwa(y) ukryta (ang. *Hidden Layer*)
- Warstwa wyjściowa (ang. *Output Layer*)

Do węzłów warstwy wejściowej wprowadzane są dane które mają dostarczyć wszystkie informacje potrzebne do tego by sieć mogła rozwiązać problem. Warstw ukrytych może być nieskończenie wiele, a *sztuczne sieci neuronowe* posiadające więcej niż 2 warstwy ukryte klasyfikowane są jako metody *Uczenia głębokiego*, dane wyjściowe mogą być same w sobie rozwiązaniem problemu lub używane do dalszej analizy i obróbki.

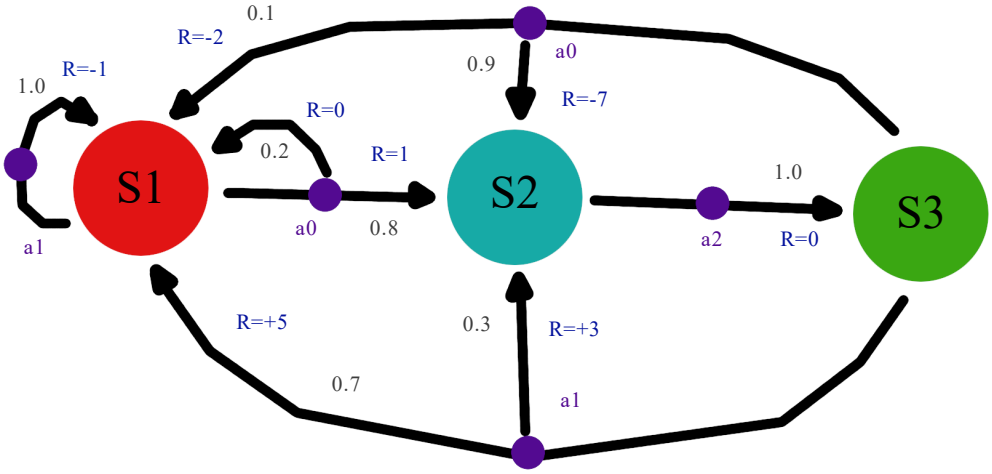


Rysunek 1.2. Schemat budowy sieci neuronowej, źródło: www.ichi.pro/pl/siec-neuronowa-klasyfikator-mnist-od-podstaw-przy-uzyciu-biblioteki-numpy-163534368243374

Dla pewnego uproszczenia możemy myśleć o węźle jak o modelu regresji liniowej. Posiada on dane wejściowe wagi, odchylenia, wartości progowe i dane wyjściowe. Po wprowadzeniu danych wejściowych nadajemy pewne wagi które podkreślają znaczenie zmiennych wchodzących na neurony. Tak przygotowane dane sumujemy przy jednoczesnym uwzględnieniu odchyleń lub wartości progowej i otrzymujemy pewne dane wyjściowe [6].

1.3 Proces Decyzyjny Markova

Algorytm który opisuje sposób w jaki nasz agent podejmuje decyzje i zmienia stan środowiska można nazwać Procesem Decyzyjnym Markova (ang. *Markov Decision Process)*. Można go opisać za pomocą zbioru akcji A, zbioru stanów S, macierzy prawdopodobieństwa tranzycji T oraz zebranej nagrody R. Gdy znajdujemy się w konkretnym stanie, mamy do dyspozycji akcje, które przeniosą nas do innych stanów. Prawdopodobieństwo przejścia z jednego stanu do drugiego jest opisane w macierzy T a za każdą wykonaną akcję przyznawana jest wcześniej już zdefiniowana nagroda, która jest sumowana. W ten sposób agent dąży do osiągnięcia największej nagrody [29]. Na poniższym diagramie widać przykład opisywanego procesu. Gdyby nasz agent znajdował się aktualnie w stanie *s3* może zdecydować aby podjąć akcje *a0* lub *a1*. W obu przypadkach przejdzie do innego stanu *s1* lub *s2* z określonym prawdopodobieństwem i zbierze określoną nagrodę. Gdyby wybrał akcje *a0* mógłby przejść do stanu *s1* z prawdopodobieństwem 0.1 oraz nagrodą równą $R = -2$ lub do stanu *s2* z prawdopodobieństwem 0.9 i nagrodą $R = -7$.



Rysunek 1.3. Przykład Procesu Decyzyjnego Markova, źródło: Opracowanie własne

1.4 Eksploracja a eksploatacja

Problem eksploracji a eksploatacji polega na zbalansowaniu w jakim stopniu agent stara się eksplorować na jakie różne sposoby może oddziaływać na środowisko (ilość powiązań akcji z konkretnym stanem w jakim znajduję się środowisko) oraz jak dużą nagrodę może uzyskać bazując na znanym mu na ten moment sposobie interakcji ze środowiskiem [13, 4].

1.5 Podział algorytmów agentów

Do testów wybraliśmy 3 różne algorytmy, które opisujemy poniżej. Każdy z nich ma pewne unikatowe zalety, które chcieliśmy wykorzystać. Wybrane algorytmy można podzielić ze względu na kilka czynników. [9, 24, 17, 15]

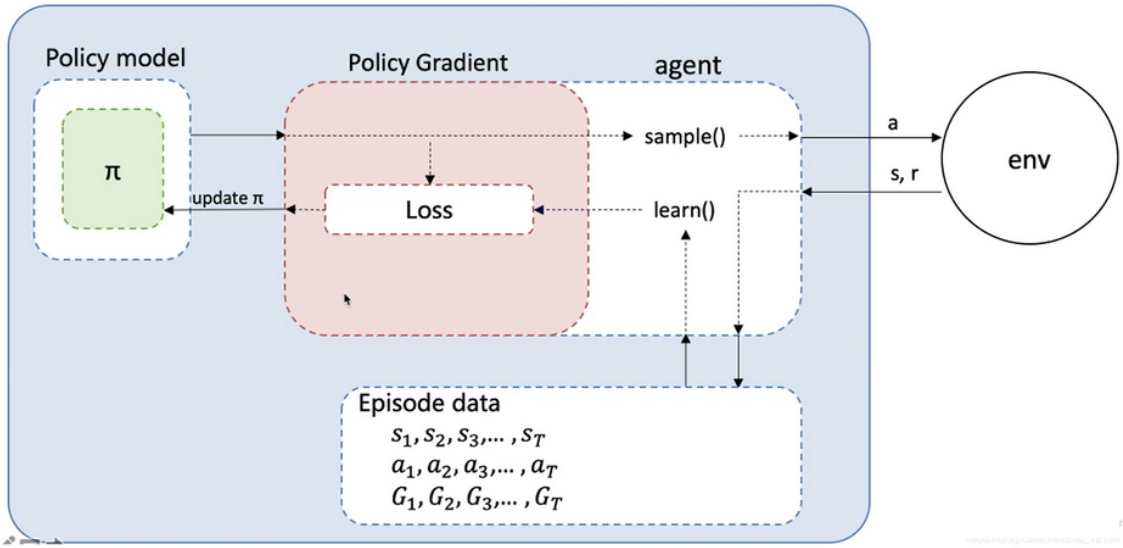
- **Bazujące lub niebazujące na modelu środowiska:** algorytm może podczas procesu uczenia stworzyć model środowiska z którym wchodzi w interakcje i na jego bazie wybierać kolejne akcje lub bazować je na polityce, która jest regularnie aktualizowana i na jej podstawie estymować jaką akcje powinien podjąć. Plusem algorytmów bazujących na modelu jest większa efektywność w konkretnie znanych zadaniach ponieważ możemy im zapewnić więcej informacji ale może to skutkować ograniczeniem ich działania tylko w tych przypadkach. Algorytmy które tego modelu nie tworzą, mogą mieć szersze zastosowanie.
- ***On-policy* lub *Off-policy*:** Algorytmy *On-policy* umożliwiają zmianę polityki bazując na danych zebranych aktualną wersją polityki a algorytmy *Off-policy* korzystają z większego zbioru danych który obejmuje dane zebrane także we wcześniejszych iteracjach. W tym przypadku bardzo ciężko definitywnie określić, który typ sprawdza się lepiej w jakich zastosowaniach.

1.5.1 PG Agent (Policy Gradient)

Algorytm PG nie bazuje na modelu środowiska oraz jest algorytmem *On-policy*. Zmiana polityki odbywa się poprzez obliczenie funkcji strat $L(\phi)$. Jest to wartość oczekiwana z logarytmu prawdopodobieństw uzyskanych z naszej polityki π_ϕ pomnożonej przez estymowany zysk dokonanej akcji.

$$L^{PG}(\phi) = \hat{E}_t[log\pi_\phi(a_t|s_t)\hat{A}_t] \tag{1.1}$$

Estymowany zysk dokonanej akcji kieruje zmianę naszej polityki w odpowiednim kierunku w zależności od znaku. Gdy jest ujemny znaczy, że przyniesie to negatywny efekt a gdy jest dodatni, przyniesie pozytywny. W tym algorytmie wykorzystujemy *baseline* jako odpowiednik krytyka co ma na celu zredukowanie wariancji zmiany polityki co powinno skutkować przyspieszeniem procesu uczenia. Ogólny przebieg algorytmu możemy zobaczyć na rysunku poniżej [Rysunek 1.4].



Rysunek 1.4. Przebieg algorytmu PG, źródło: https://programmer.group/pg-algorithm-based-on-policy-gradient.html

1.5.2 PPO Agent (Proximal Policy Optimization)

Z tej samej rodziny algorytmów co PG jest także algorytm PPO, czyli proksymalna optymalizacja polityki, nie bazuje na modelu środowiska oraz jest algorytmem *On-policy*. Jego cechą charakterystyczną jest to, że aktualizacja polityki odbywa się w sposób ograniczony co eliminuje jej niepewne drastyczne zmiany, które mogą wyniknąć na przykład z okazjonalnie bardziej nastawionych na eksplorację epizodów. Działa to podobnie jak w przypadku algorytmu PG jednak z ograniczeniem zmiany. Ograniczenie sprowadza się to do porównania polityki $\pi_0(\phi)$ z jej następną iteracją $\pi_1(\phi)$ a dokładniej wielkości zmiany jaka nastąpiłaby bez ograniczenia. Gdy stosunek zmiany polityki jest zbyt duży, czyli wychodzi poza zakres $(1-\epsilon, 1+\epsilon)$ zmiana jest odpowiednio ograniczana aby zmieścić się w przedziale. Funkcja strat $L^{CLIP}(\phi)$ obliczana jest jako wartość oczekiwana z minimalnej wartości zadania PG lub PG z ograniczeniem pomnożonej przez estymowany zysk dokonanej akcji.

$$L^{CLIP}(\phi) = \hat{E}_t[min(r_t(\phi)\hat{A}_t, clip(r_t(\phi), 1-\epsilon, 1+\epsilon)\hat{A}_t)] \tag{1.2}$$

1.5.3 DQN Agent (Deep Q-Network)

Kolejny algorytm także nie bazuje na modelu środowiska jednak tym razem mamy do czynienia z algorytmem *Off-policy*. Tworzy on optymalną politykę na podstawie tabeli, która określa nagrodę w konkretnym stanie przy określonej akcji. Tabela staje się pewnego rodzaju zbiorem instrukcji, który jest aktualizowany po każdym kroku. Na przykładowej tabeli [Tabela 1.1] możemy zauważyć, że mamy do dyspozycji trzy różne akcje (a1, a2, a3) oraz możemy znaleźć się w trzech różnych stanach (s1, s2, s3). Po wykonanej akcji tablica jest aktualizowana według wzoru:

$$Q_{new}(s,a) = Q(s,a) + \xi[R(s,a) + \varrho maxQ'(s',a') - Q(s,a)] \tag{1.3}$$

gdzie:
$Q_{new}(s,a)$ - nowa wartość w tabeli dla akcji a i stanu s
$Q(s,a)$ - obecna wartość w tabeli
ξ - częstotliwość nauczania
$R(s,a)$ - Nagroda za wykonanie akcji a w stanie s
ϱ - współczynnik skalowania długoterminowej nagrody
$maxQ'(s',a')$ - Maksymalna możliwa przyszła nagroda dla nowego stanu s'

	a1	a2	a3
s1	0	0	0
s2	0	0	0
s3	0	0	0

Tabela 1.1. Przykładowa tabela aktualizacji polityki

Rozdział 2

Model matematyczny Aeropendulum

Autor: Marcin Sypniewski, Michał Rojewski, Mikołaj Maciejewski

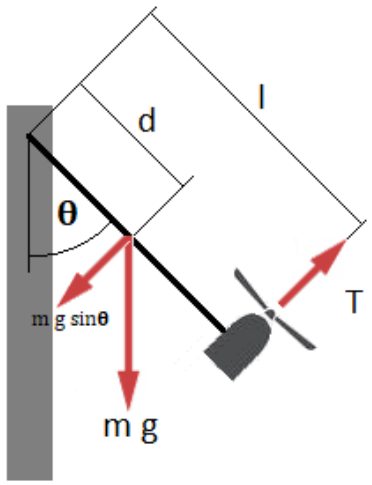
2.1 Parametry modelu

Parametry modelu zostały wcześniej obliczone i zdefiniowane przez studentów, którzy wykonali stanowiska laboratoryjne [19] i przedstawiają się następująco:

- masa: $m = 0.18kg$ (masa silnika i uchwytu razem)
- przyspieszenie grawitacyjne: $g = 9.81\frac{m}{s^2}$
- długość wahadła: $l = 0.25m$
- odległość od środka masy: $d = 0.25m$
- tarcie wiskotyczne na łożyskach: $c = 0.006\frac{Nms}{rad}$
- moment bezwładności: $J = m * l^2 = 0.01125kgm^2$

2.2 Równanie różniczkowe

Równanie różniczkowe opisujące działanie Aeropendulum także zostało wyznaczone przez wcześniej wspomnianych studentów na podstawie zasad dynamiki Newtona oraz rozkładu działających na wahadło sił. T to siła ciągu generowana przez śmigło.



Rysunek 2.1. Rozkład sił działających na Aeropendulum, źródło: [19]

Równanie różniczkowe opisujące układ:

$$J\ddot{\theta} + c\dot{\theta} + mgdsin\theta = lT \tag{2.1}$$

gdzie:
θ - kąt wychylenia wahadła,
$\dot{\theta}$ - prędkość wahadła,
$\ddot{\theta}$ - przyspieszenie wahadła

Do implementacji w Simulinku równanie różniczkowe zostało przekształcone do postaci:

$$\ddot{\theta} = -\frac{c}{J}\dot{\theta} - \frac{mgd}{J}sin\theta + \frac{l}{J}T \tag{2.2}$$

Aby uzależnić równanie od obrotów silnika, którymi chcemy sterować Aeropendulum dodany został współczynnik α, który zmienia obroty silnika na siłę ciągu:

$$T = \alpha * \omega_{RPM} \tag{2.3}$$

Rozdział 3

Wyniki badań symulacyjnych

Autor: Marcin Sypniewski

3.1 Przygotowanie wstępne

Do stworzenia modelu symulacji wykorzystaliśmy *Reinforcement Learning Toolbox* oferowany przez środowisko Matlab. Ta biblioteka pozwala na:

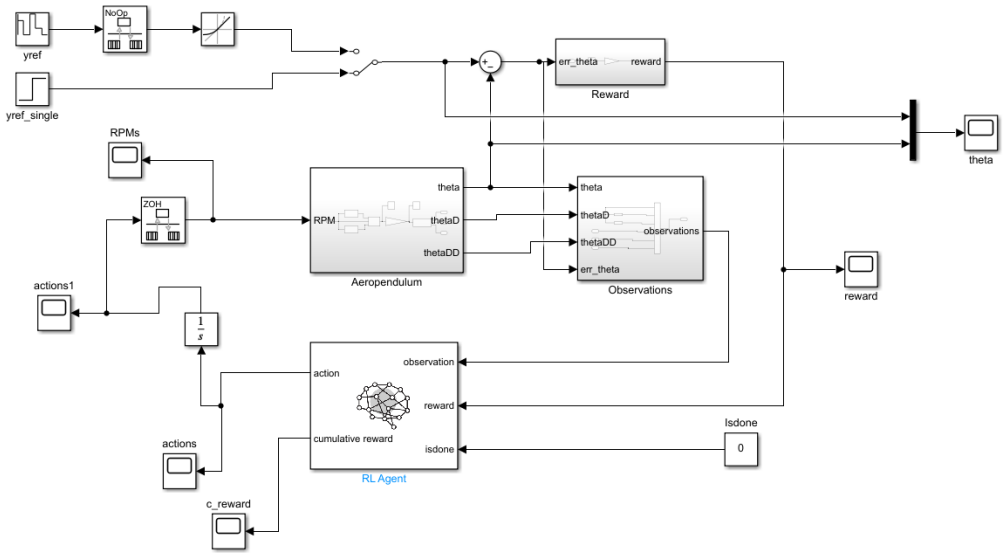
- Wykorzystanie gotowego bloku w Simulinku z inteligentnym agentem.
- Opisanie możliwych akcji i typu obserwacji
- Zastosowanie gotowych klas wybranych algorytmów agenta
- Przejrzystą edycje parametrów agenta czy wykorzystywanych sieci neuronowych
- Obserwacje na wykresach wartości nagrody przy każdym epizodzie w celu oceny skuteczności procesu uczenia

Przeprowadziliśmy także przegląd gotowych implementacji uczenia przez wzmacnianie w programie Simulink w przypadkach prostego sterowania. Pozwoliło nam to uzyskać dobry pogląd na sposób w jaki należy definiować kluczowe segmenty modelu takie jak na przykład wektor obserwacji oraz jak przyjąć przestrzeń i typ akcji aby uzyskać jak najbardziej naturalny sposób sterowania.

3.2 Przygotowanie modelu symulacji

Model symulacyjny ma posłużyć jako środek do znalezienia najlepszego algorytmu dla naszego agenta oraz dobrania odpowiednich parametrów i sieci neuronowych.
Do implementacji agenta z algorytmem w modelu służy blok *RL Agent*, który ma przygotowane wejścia na obserwacje, nagrodę oraz sygnał przerywający symulację (w naszym przypadku pominięty, gdyż koniec sesji nauczania wyznaczaliśmy w skrypcie odpowiadającym za całą procedurę nauczania) oraz wyjścia, którym jest aktualna akcja przekazywana do modelu Aeropendulum (środowiska) oraz podgląd przebiegu kumulowanej nagrody.

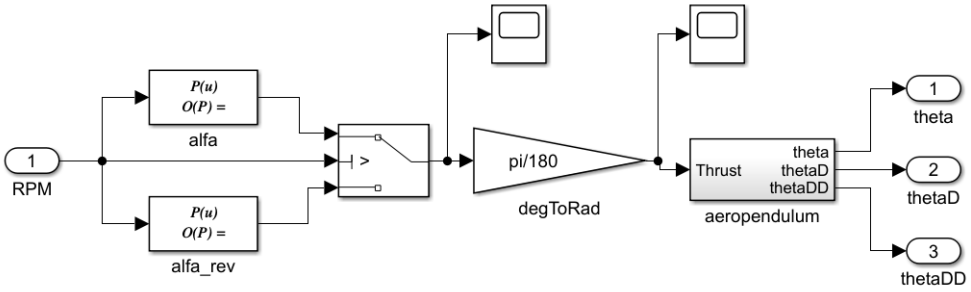
Akcje, które generuje agent są całkowane zanim zostaną przekazane do bloku z implementacją Aeropendulum. W ten sposób agent generuje sygnał, który odpowiada za zmianę sygnału sterującego a nie bezpośrednio zadawaną prędkość obrotową. Kumulowana nagroda jest powiększana o wartość zebranej nagrody w danym korku aktualnie rozpatrywanego epizodu.



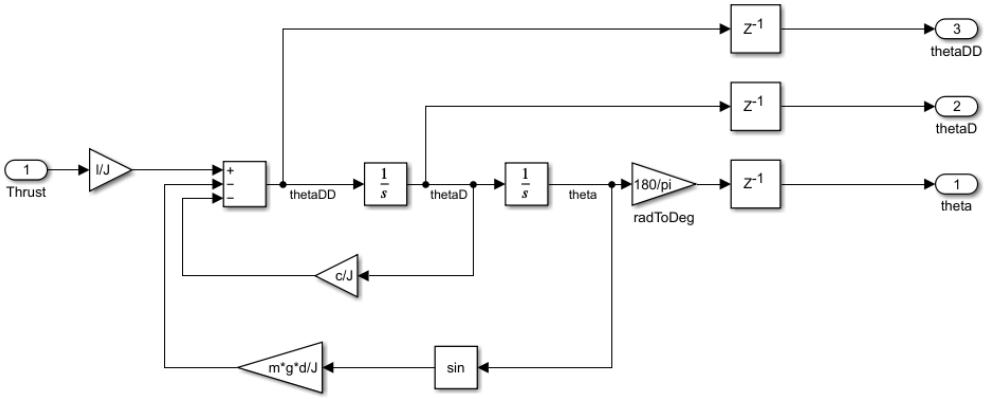
Rysunek 3.1. Model symulacyjny, źródło: Opracowanie własne

3.2.1 Środowisko

Model Aeropendulum został przygotowany przez studentów, którzy przygotowali rzeczywiste stanowiska z Aeropendulum [19] i to właśnie go używamy w naszej symulacji aby zachować zgodność z powstałym stanowiskiem. Model jest oparty o wcześniej zaprezentowany model matematyczny obiektu sterowania. Model został przez nas sprawdzony i dostosowany do celu w jakim chcemy go wykorzystać. Pozbyliśmy się bloku regulatora PID oraz dodaliśmy opóźnienie jednostkowe na wyjście sygnałów *theta, thetaD* oraz *thetaDD* aby zlikwidować pętlę algebraiczną, która pojawiła się po zamknięciu całego układu przez dodanie bloku agenta.
Model składa się z zewnętrznej części która zmienia prędkość obrotową silnika na ciąg, który jest potem przeliczany w części zewnętrznej na kąt wychylenia, prędkość oraz przyspieszenie wahadła.



Rysunek 3.2. Zewnętrzna implementacja Aeropendulum, źródło: Opracowanie własne



Rysunek 3.3. Wewnętrzna implementacja Aeropendulum, źródło: Opracowanie własne

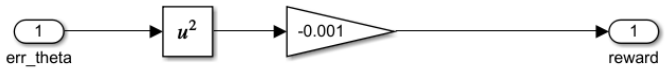
3.2.2 Nagroda

Sygnał nagrody składa się z tylko jednej części, mianowicie ujemnej nagrody od kwadratu uchybu kąta wychylenia. Ma to skutkować tym, że im wahadło będzie dalej od wartości zadanej tym mniejszą nagrodę dostanie.
Załóżmy, że wartość zadana to $yref = 30°$. Gdy wahadło zostanie wychylone na kąt równy $\theta = 15°$ to nagrodę w tym aktualnym stanie uzyskamy ze wzoru:

$$\begin{aligned} R &= -0.001(err\_theta^2) \\ R &= -0.001 * (yref - \theta)^2 \\ R &= -0.001 * (30 - 15)^2 \\ R &= -0.001 * 225 \\ R &= -0.225 \end{aligned} \tag{3.1}$$

gdzie, R - nagroda,
err_theta - uchyb kąta wychylenia
θ - kąt wychylenia
$yref$ - wartość zadana kąta wychylenia

Jak widać we wzorze (3.1), współczynnik -0.001 odpowiada za przeskalowanie wartości nagrody aby wartość nagrody zmniejszała się wraz ze wzrostem uchybu.
Próbowaliśmy rozszerzyć blok nagrody o dodatkowe sygnały (np. ujemnej nagrody od zbyt dużego przyspieszenia wahadła) jednak utrudniało to agentowi zrozumienie jego celu. Dlatego pozostawiliśmy sygnał nagrody obliczany tylko z wartości uchybu kąta wychylenia, ponieważ to właśnie kąt wychylenia wahadła jest wartością regulowaną.



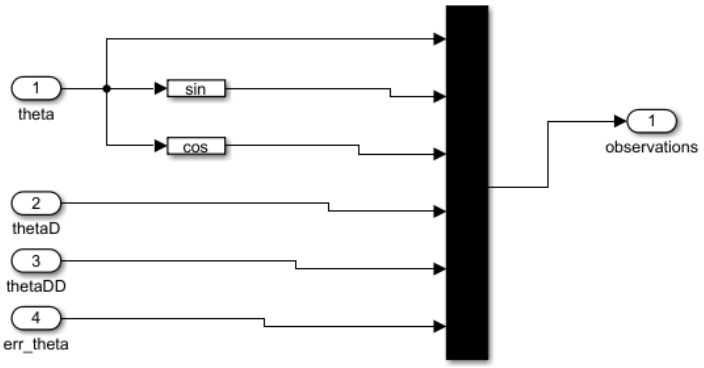
Rysunek 3.4. Funkcja nagrody, źródło: Opracowanie własne

3.2.3 Obserwacje

Aby dokładnie określić stan w jakim znajduje się Aeropendulum, jak zmienia się ten stan i dać agentowi jak najwięcej informacji nasze obserwacje składają się z:

- kąta wychylenia
- sinusa kąta wychylenia
- cosinusa kąta wychylenia
- prędkości kątowej
- przyspieszenia kątowego
- uchybu kąta wychylenia

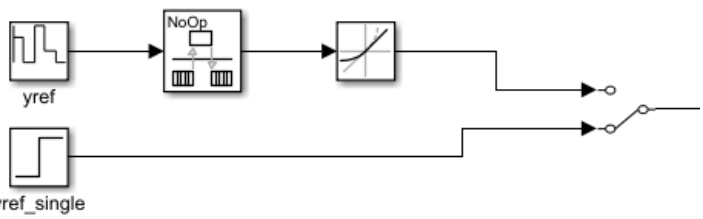
Przekazujemy kąt wychylenia oraz jego sinus i cosinus, ponieważ jest to najważniejsza wartość w kontekście naszej regulacji. Prędkość kątowa i przyspieszenie kątowe są pochodnymi kąta wychylenia i przekazanie ich jako już obliczone wartości do agenta przyspiesza jego działanie, ponieważ agent nie musi ich liczyć samemu podczas wybierania akcji. Uchyb kąta wychylenia nie spełnia tak kluczowej roli jak pozostałe obserwacje ale nadal jest czynnikiem, który poszerza wiedzę agenta o środowisku i jego aktualnym stanie.



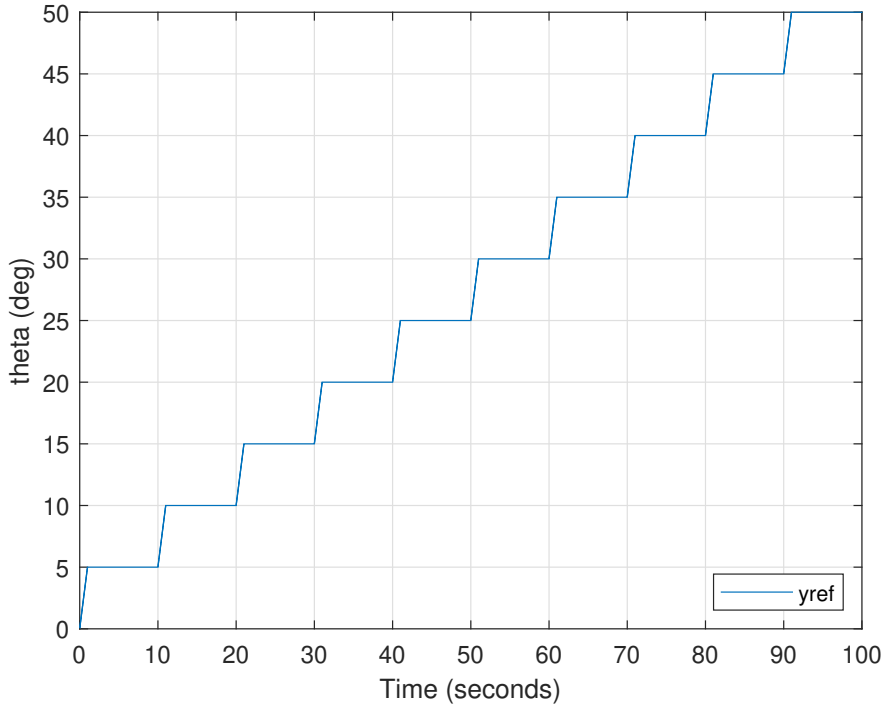
Rysunek 3.5. Implementacja obserwacji, źródło: Opracowanie własne

3.2.4 Sygnał referencyjny

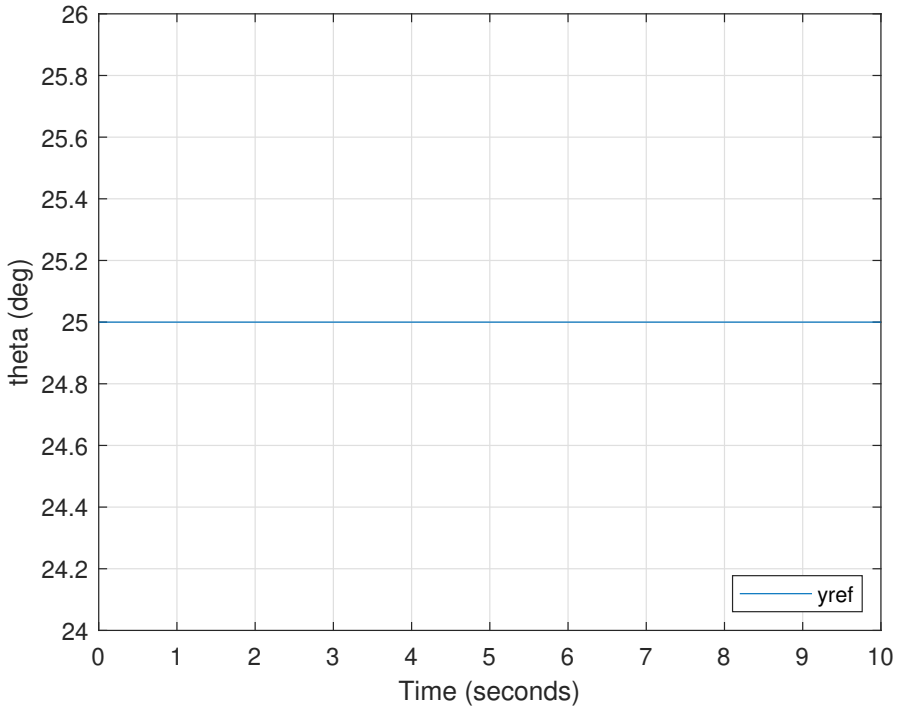
Zastosowaliśmy klasyczny, idealny skok jednostkowy jako sygnał zadanego kąta wychylenia, ponieważ mimo swojej prostoty dawał najlepsze rezultaty. Gdy próbowaliśmy upłynnić skok korzystając z bloku *Rate Limiter*, który ograniczał pierwszą pochodną sygnału agent dostawał wysoką nagrodę za pozostanie w bezruchu na samym początku co opóźniało jego działanie i utrudniało naukę. Kolejną zmianą, którą próbowaliśmy wprowadzić do naszego modelu był sygnał składający się z kilku kątów (3.7). To jednak powodowało, że agent nie podążał za sygnałem a ustalał się na jednej wartości i uzyskiwał w ten sposób zadowalającą go nagrodę uśredniając zyski z dobrej regulacji dla jednego kąta a straty przy reszcie zadanych kątów. Dlatego zdecydowaliśmy się na najprostsze rozwiązanie jakim jest właśnie skok jednostkowy.



Rysunek 3.6. Implementacja sygnałów referencyjnych, źródło: Opracowanie własne



Rysunek 3.7. Przykładowy przebieg zadanej sekwencji wartości kąta od czasu, źródło: Opracowanie własne



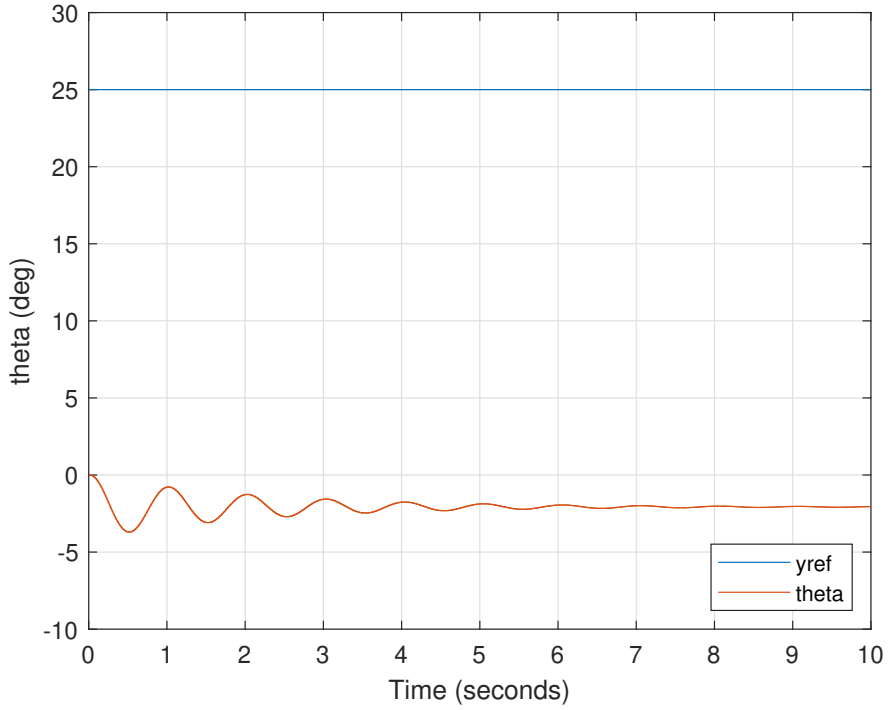
Rysunek 3.8. Przykładowy przebieg zadanej pojedynczej wartości kąta od czasu, źródło: Opracowanie własne

3.3 Wyniki symulacji

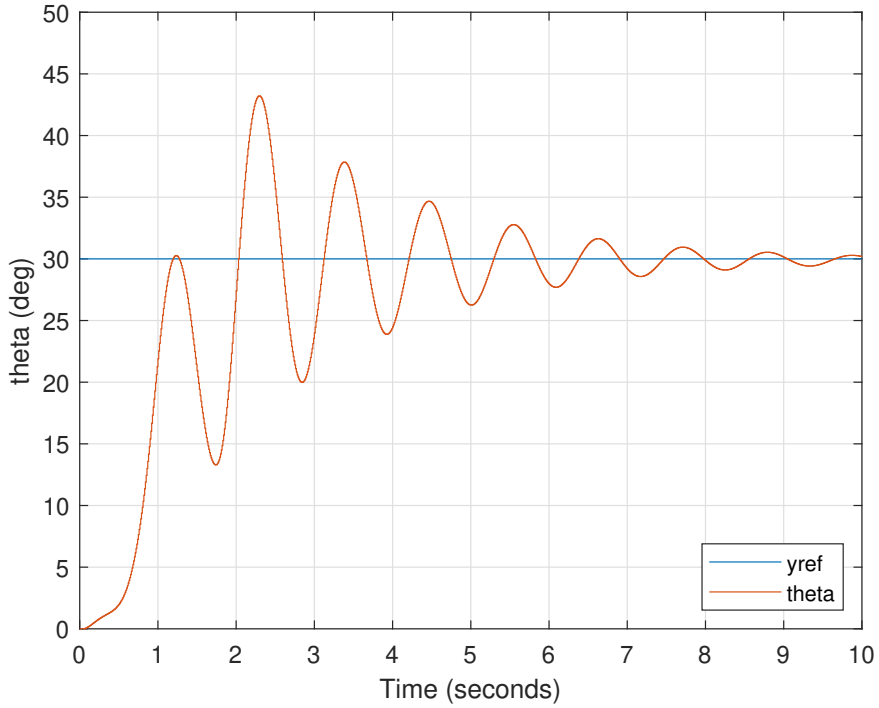
Wykonaliśmy wiele testów dla każdego ze sprawdzanych przez nas algorytmów przy zmianie: struktury modelu (np. ilości obserwacji), sygnału referencyjnego czy parametrów algorytmu. Dążyliśmy przede wszystkim do tego aby uzyskać szybką reakcje regulatora oraz jak najmniejszy uchyb ustalony.
Na wszystkich wykresach sygnał niebieski to sygnał referencyjny a czerwony to kąt wychylenia wahadła Aeropendulum.

3.3.1 Algorytm PPO

Był to algorytm, który spodziewaliśmy się, że da najlepsze rezultaty ze względu na to jak w ograniczony sposób aktualizował swoją politykę. Okazało się jednak, że powodowało to wpadanie algorytmu w długie okresy niepoprawnego działania i prób eksploatacji błędnych podejść [Rysunek 3.9]. Agent wpadał w tak zwane lokalne minima. Zdarzało się, że agent miał szczęście i udało się za pomocą wpływu *Entropy Loss Weight* (Opis parametru w sekcji 4.2) trafić w poprawny sposób działania przy danym kącie lecz przy próbie dalszego douczania ciężko było mu przenieść sposób regulacji na szerszy zakres kątów. Nawet gdy w końcu zaczął zbliżać się do poprawnego działania, reagował zbyt powoli oraz uchyb ustalony był zbyt duży.



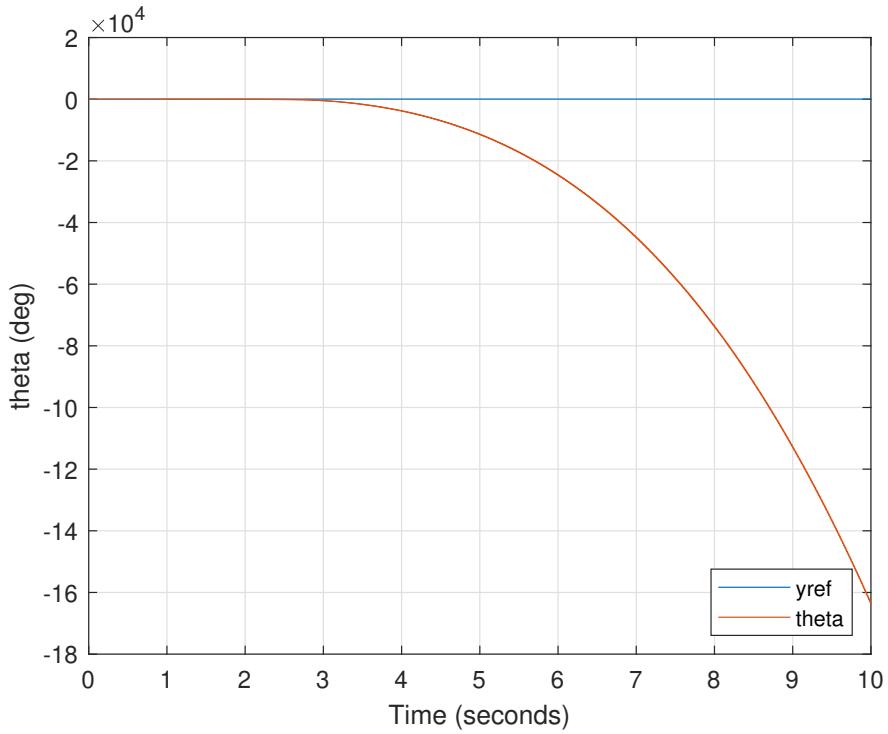
Rysunek 3.9. Lokalne minimum dla algorytmu PPO, źródło: Opracowanie własne



Rysunek 3.10. Regulacja algorytmem PPO dla theta = 30° po jednej sesji nauczania, źródło: Opracowanie własne

3.3.2 Algorytm DQN

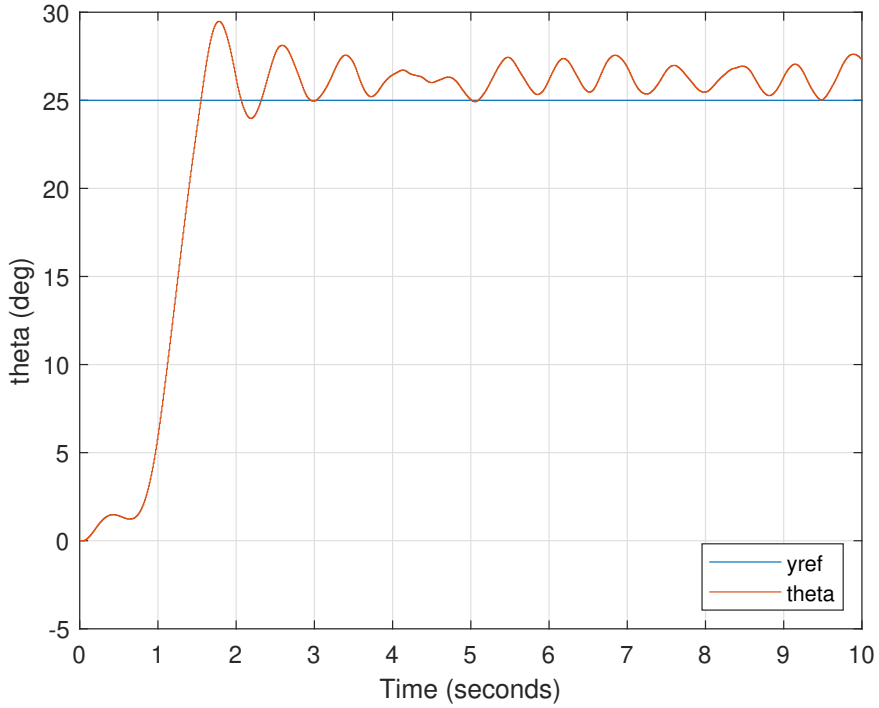
Algorytm najbardziej odstający w istocie działania od pozostałych, który pozwolił nam sprawdzić opcję *Off-policy* był bardzo niestabilny. Bardzo długo zajęło nauczenie go do uchwycenia mniej więcej zasady regulacji naszego wahadła. W tym przypadku wyniki były najgorsze ze wszystkich trzech algorytmów.



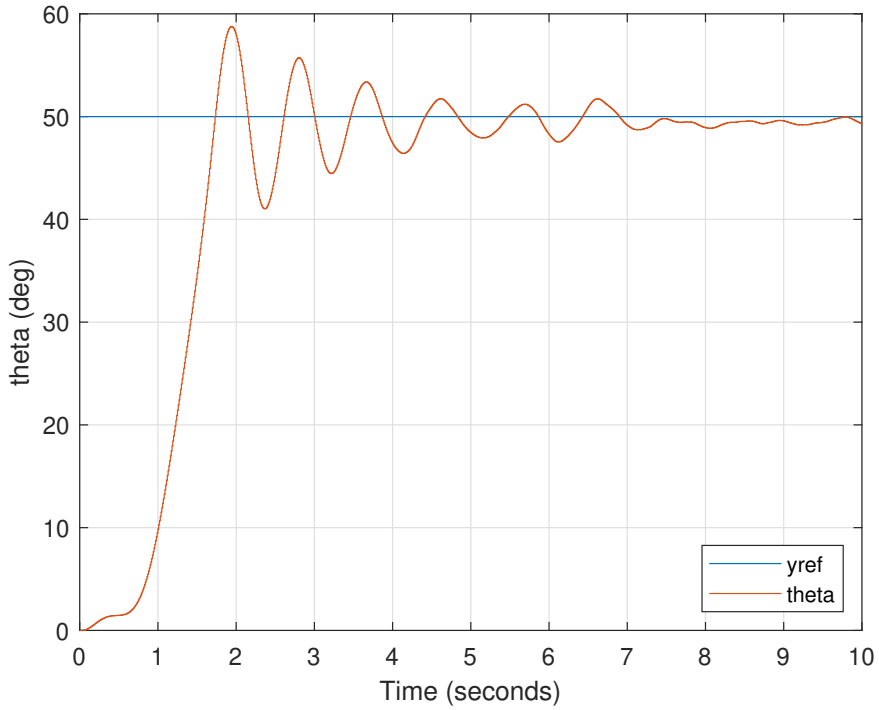
Rysunek 3.11. Nieudana regulacja algorytmem DQN dla theta $= 30°$ *po kilku sesjach nauczania, źródło: Opracowanie własne*

3.3.3 Algorytm PG

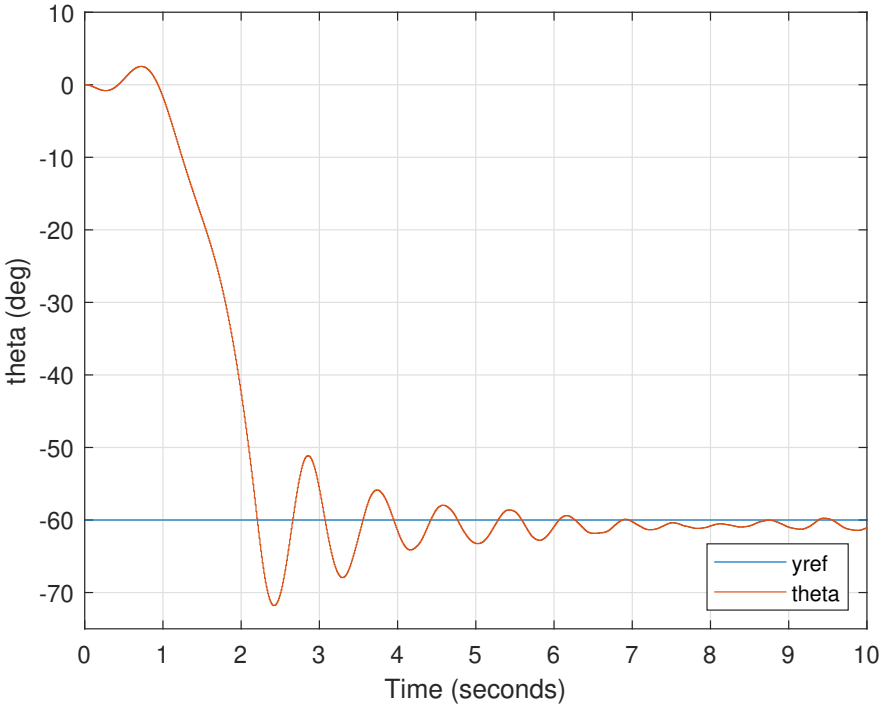
Ostatni algorytm, czyli PG okazał się bardzo skuteczny. Najszybciej ze wszystkich podczas procesu nauczania eksplorował środowisko i był w stanie eksploatować poprawny sposób działania. Już po około 6000-8000 epizodach był w stanie reagować wystarczająco szybko oraz osiągnąć uchyb ustalony około 2-3 stopni[Rysunek 3.12]. Każda następna sesja nauczania poprawiała jego działanie zmniejszając uchyb ustalony oraz poszerzając zakres kątów w którym jest w stanie dobrze regulować do przedziału $< -60°, 50° >$.



Rysunek 3.12. Regulacja algorytmem PG dla theta = 25° po pierwszej sesji nauczania, źródło: Opracowanie własne



Rysunek 3.13. Regulacja algorytmem PG dla theta = 50° po kilku sesjach nauczania, źródło: Opracowanie własne



Rysunek 3.14. Regulacja algorytmem PG dla theta = −60° po kilku sesjach nauczania, źródło: Opracowanie własne

3.4 Wpływ parametrów na wynik regulacji

3.4.1 Sposób sterowania akcjami

Podczas naszych testów korzystaliśmy z dwóch sposobów traktowania akcji w kontekście oddziaływania na nasze środowisko. Pierwszy z nich, który okazał się nieodpowiedni, to bezpośrednie zadawanie wartości prędkości silnika jako akcji do naszego Aeropendulum. Wadą tego rozwiązania były nagłe i ostre zmiany wartości prędkości silnika. Skutkowało to słabą regulacją z dużym uchybem. Kolejną wadą jest to, że ta koncepcja byłaby fizycznie nie do zrealizowania ponieważ sterownik, który jest użyty do zadawania określonej prędkości z jaką ma działać silnik posiada zabezpieczenie nadprądowe, które przerywałoby działanie Aeropendulum.

3.4.2 Liczba i zakres akcji

Liczba i zakres akcji może znacząco ułatwić lub utrudnić cały proces. Gdy liczba akcji jest mała (np. dwie lub trzy) agent szybko uczy się w jaki sposób z nich korzystać jednak jego możliwości są ograniczone. Zbyt duże zwiększenie liczby akcji powoduje znaczące wydłużenie całego procesu i utrudnienie agentowi znalezienie właściwego sposobu na uzyskanie postawionego mu celu. Zakres akcji ma podobne skutki. Gdy zbyt mały agent nie będzie w stanie spełnić niektórych wymagań, które mu narzucimy a gdy będzie zbyt duży agent straci sporo czasu na eksplorowanie akcji, które nie są mu potrzebne w osiągnięciu największej nagrody.

Rozdział 4

Wybór najlepszego algorytmu

Autor: Marcin Sypniewski

4.1 Proces działania algorytmu

Na podstawie zebranych przez nas wyników można z całą pewnością stwierdzić, że algorytm PG uzyskał najlepszą efektywność regulacji. Najlepiej spełniał postawione przez nas kluczowe założenia, czyli jak najmniejszy uchyb ustalony oraz szybkość reakcji.
Jako że algorytm PG okazał się najlepszym i zostanie przez nas wykorzystany przy implementacji na stanowisku rzeczywistym przedstawimy dokładniej jego działanie. Jak już pisaliśmy wcześniej jego celem jest estymacja prawdopodobieństwa wykonania poszczególnych akcji i wybranie jednej z nich na podstawie rozkładu prawdopodobieństwa, a aktualizacja polityki odbywa się po każdym ukończonym epizodzie. Algorytm PG bazuje na algorytmie *Monte Carlo Policy Gradient* [14], którego kroki przedstawiają się następująco:

1. Inicjalizacja aktora $\pi(S)$ z losowymi parametrami ϕ
2. Inicjalizacja krytyka $V(S)$ z losowymi parametrami Φ
3. Dla każdego epizodu generowany jest zestaw próbek zebrany przy użyciu aktualnej polityki $\pi(S)$. Zbiór próbek składa się z:
$$S_0, A_0, R_1.S_1, ..., S_{T-1}, A_{T-1}, R_T, S_T$$
4. Dla $t = 1, 2, 3, ..., T$: Obliczamy przeskalowaną przyszłą zebraną nagrodę
$$G_t = \sum_{k=t}^{T} \gamma^{k-t} R_k$$
Obliczamy funkcję δ_t korzystając z estymacji krytyka
$$\delta_t = G_t - V(S_t|\Phi)$$
5. Sumujemy gradienty dla krytyka
$$d\Phi = \sum_{t=1}^{T-1} \delta_t \nabla_\Phi V(S_t|\Phi)$$

6. Sumujemy gradienty dla aktor

$$d\phi = \sum_{t=1}^{T-1} \delta_t \nabla_\phi ln\pi(S_t|\phi)$$

7. Aktualizujemy parametry krytyka

$$\Phi = \Phi + \beta d\Phi$$

gdzie β - częstotliwość uczenia krytyka

8. Aktualizujemy parametry aktora

$$\phi = \phi + \alpha d\phi$$

gdzie α - częstotliwość uczenia aktora

9. Powtarzamy kroki od 3 do 9 dla każdego epizodu aż proces nauczania dobiegnie końca

4.2 Skrypt algorytmu agenta

W skrypcie naszego algorytmu agenta tworzymy sieci neuronowe używane przez krytyka i aktora. Na podstawie tych sieci i stworzonych reprezentacji aktora i krytyka tworzymy naszego agenta. Podczas definicji agenta określamy jego kluczowe parametry takie jak:

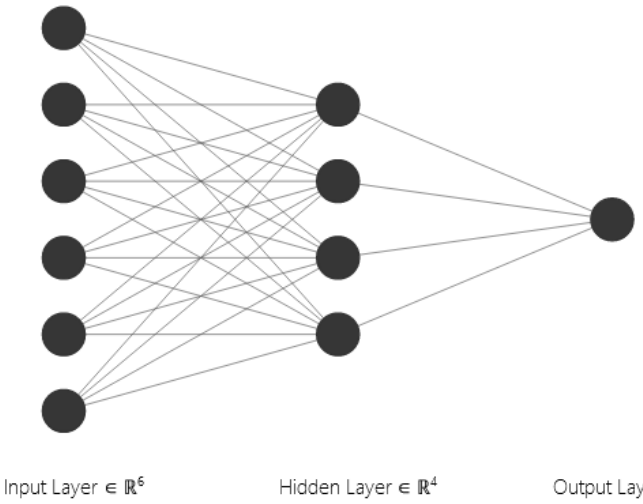
- *UseBaseline = true* - jest to flaga określająca czy używamy w algorytmie agenta krytyka w postaci *baseline*
- *DiscountFactor* - jest to wartość skalarna z przedziału od 0 do 1, która określa jak bardzo agent bierze pod uwagę przyszłą nagrodę. W naszym wypadku wynosi ona 0.95 aby agent skupił się na długoterminowej nagrodzie
- *SampleTime* $= T_s$ - Jest to czas próbkowania, który zdefiniowaliśmy tak jak dla całego układu czyli $T_s = 0.01s$
- *EntropyLossWeight* - ten parametr z przedziału od 0 do 1 odpowiada za balans pomiędzy eksploracją a eksploatacją. Podczas obliczania funkcji strat odejmujemy dodatkowy czynnik, który sprawia, że agent jest mniej pewny podjętej akcji. Może to pomóc wydostać się agentowi z minimów lokalnych (sytuacji gdy agent utknie z nie najbardziej optymalną polityką). Wartość 0.03 może wydawać się bardzo mała, jednak jest to jedna ze standardowych wartości stosowanych dla algorytmów uczenia przez wzmacnianie aby osiągnąć optymalną politykę.

listing 4.1. Skrypt algorytmu PG agenta

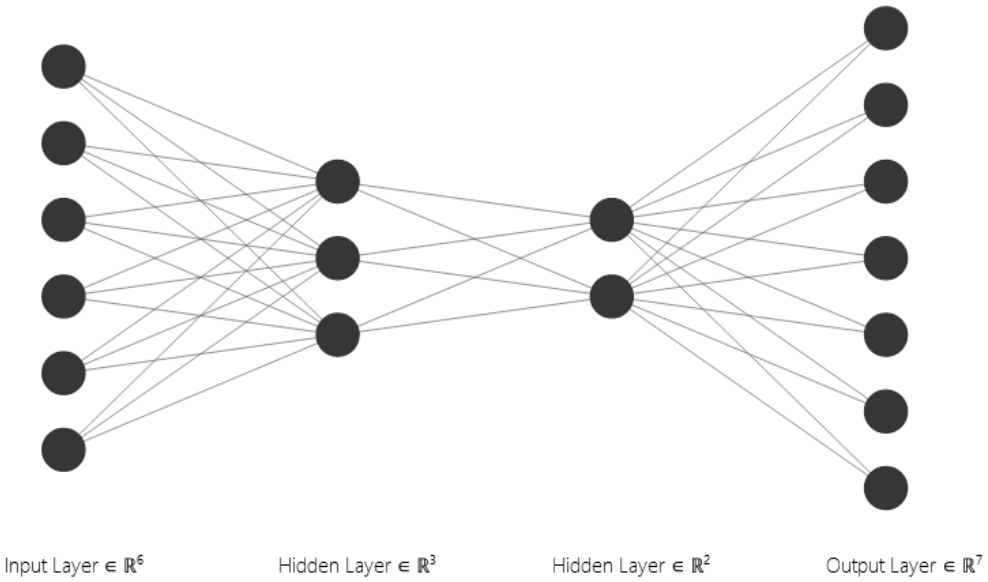
```
%% Agent

% create a network to be used as underlying critic approximator
baselineNetwork = [
    featureInputLayer(obsInfo.Dimension(1), 'Normalization', '
        none', 'Name', 'state')
    fullyConnectedLayer(4, 'Name', 'BaselineFC')
    fullyConnectedLayer(1, 'Name', 'BaselineFC2', '
        BiasLearnRateFactor', 0)];

% set some options for the critic
baselineOpts = rlRepresentationOptions('LearnRate',5e-3,'
    GradientThreshold',1);

% create the critic based on the network approximator
baseline = rlValueRepresentation(baselineNetwork,obsInfo,'
    Observation',{'state'},baselineOpts);

% create a network to be used as underlying actor approximator
actorNetwork = [
    featureInputLayer(obsInfo.Dimension(1), 'Normalization', '
        none', 'Name', 'state')
    fullyConnectedLayer(3,'Name','HL1')
    fullyConnectedLayer(2,'Name','HL2')
    fullyConnectedLayer(Num_of_actions, 'Name', 'action', '
        BiasLearnRateFactor', 0)];

% set some options for the actor
actorOpts = rlRepresentationOptions('LearnRate',5e-3,'
    GradientThreshold',1);

% create the actor based on the network approximator
actor = rlStochasticActorRepresentation(actorNetwork,obsInfo,
    actInfo,...
     'Observation',{'state'},actorOpts);

agentOpts = rlPGAgentOptions();
agentOpts.UseBaseline = true;
agentOpts.DiscountFactor = 0.95;
agentOpts.SampleTime = Ts;
agentOpts.EntropyLossWeight = 0.03;

agent = rlPGAgent(actor,baseline,agentOpts);
```

4.2.1 Wykorzystane sieci neuronowe

Staraliśmy się dobrać sieci z relatywnie prostą strukturą ponieważ pracujemy z prostym problemem regulacji. Sieci aktora oraz krytyka posiadają sześć neuronów w warstwie wejściowej. Odpowiada to liczbie obserwacji jakie mamy zdefiniowane w układzie. Sieć krytyka na wyjściu ma tylko jeden neuron jako, że jego zadaniem jest ocena działania aktora. Natomiast sieć aktora ma w ostatniej warstwie siedem neuronów co odpowiada liczbie możliwych akcji, które mogą zostać wykonane.



Rysunek 4.1. Sieć neuronowa krytyka, źródło: Opracowanie własne



Rysunek 4.2. Sieć neuronowa aktora, źródło: Opracowanie własne

Rozdział 5

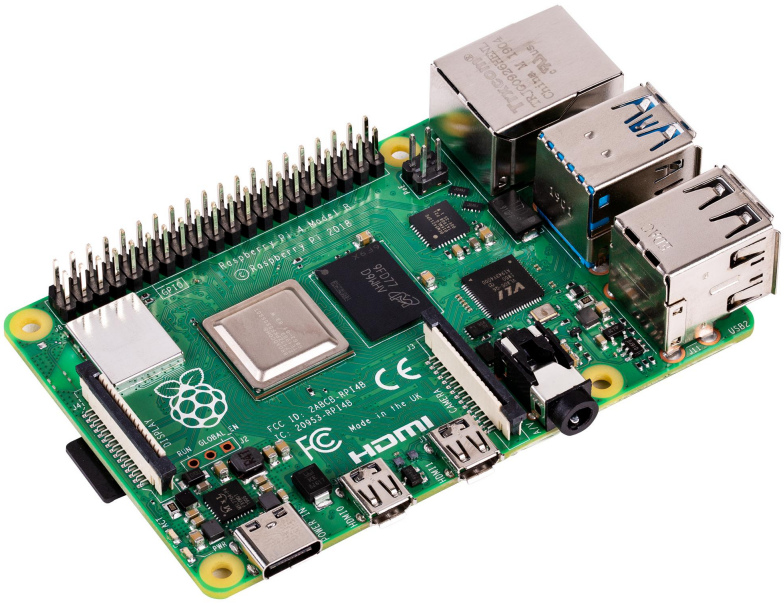
Komunikacja

Autor: Michał Rojewski

5.1 Sprzęt fizyczny

5.1.1 Raspberry Pi

Raspberry Pi jest najbardziej rozpowszechnionym komputerem jednopłytkowym wykorzystywanym głównie do celów edukacyjnych lub hobbystycznych. Tutaj pełni role zadajnika i serwera - odbiera wartość kąta z enkodera i wysyła ją do klienta *Matlab* który po przetworzeniu kąta na prędkość, zwraca ją tak by mogła zostać wysłana do sterownika silnika.



Rysunek 5.1. Komputer jednopłytkowy Raspberry Pi, źródło: https://pl.farnell.com/raspberry-pi/rpi4-modbp-2gb/raspberry-pi-4-model-b-2gb/dp/3051886

5.1.2 STEVAL-SPIN 3201

Do obsługi silnika korzystamy z sterownika z gotowymi rozwiązaniami *STEVAL-SPIN 3201* firmy *STMICROELECTRONICS*. Układ jest typu *Plug and Play*, dzięki czemu na nasze potrzeby praktycznie nie musimy w żaden sposób modyfikować instrukcji procesora. Wystarczy w odpowiednim formacie przesłać informację które sterownik zaimplementuje na silnik.

Rysunek 5.2. Sterownik Silnika STEVAL-SPIN3201, źródło: https://pl.farnell.com/stmicroelectronics/steval-spin3201/eval-board-bldc-controller/dp/2761527

MCP

Sterownik komunikuje się za pomocą *UART* dzięki specjalnie stworzonej przez producenta ramki komunikacyjnej *MCP Motor Control Protocol* Pozwala ona na odczyt oraz zapis danych w rejestrach kontrolera np. prędkość silnika, błędów lub też informację chociażby o wystartowaniu i zatrzymaniu silnika.

Frame_start	Payload_length	Payload_id	Payload[0]	...	Payload[n]	CRC

Rysunek 5.3. Konstrukcja ramki protokołu MCP, źródło: Opracowanie własne

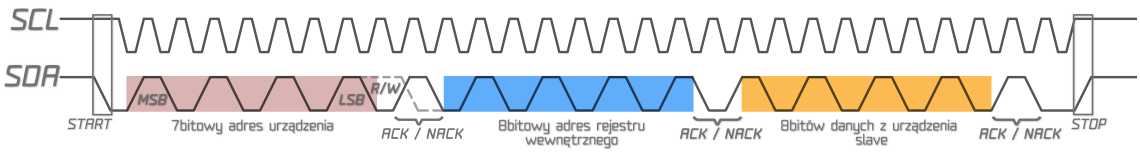
5.1.3 Enkoder Magnetyczny AS5600

Enkoder bezkontaktowy *Grove AS5600* firmy *Seeedstudio* o rozdzielczości 12-bit używany do zbadania kątu odchylenia ramienia Aeropendulum porozumiewający się z *RPi* za pomocą magistarli I^2C. Z uwagi na to, że oryginalne oprogramowanie dostarczone przez producenta było napisane dla środowisko Arduino, skorzystaliśmy po poprawkach z biblioteki napisanej na potrzeby innej pracy inżynierskiej [18].

Rysunek 5.4. Enkoder magnetyczny AS5600, źródło: https://kamami.pl/moduly-peryferyjne-grove-seeed-studio/580651-modul-enkodera-magnetycznego-z-ukladem-as5600-ze-zlaczem-grove-1010206912.html

5.2 I^2C

Interfejs I^2C korzysta z asymetrycznej komunikacji *Master-Slave*, gdzie w naszym przypadku *Raspberry Pi* to Master a enkoder *AS5600* to Slave. Protokół ten składa się z 2 linii - *SDA* czyli adresowej i *SCL* która jest zegarem taktującym, obie linie podłączamy poprzez rezystory podciągające (RPi jest natywnie przygotowane i posiada rezystory na płytce PCB).

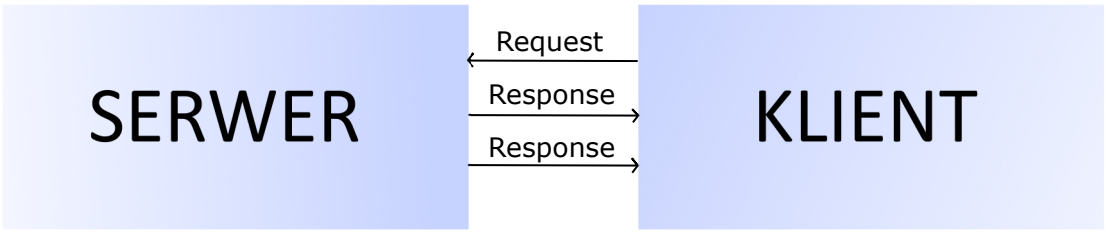


Rysunek 5.5. Przykładowy przebieg komunikacji I^2C, źródło: feriar-lab.pl/kurs-arduino-20-i2c/

Dane przesyłane są w bitach po 8 w formie *Big Endian*. Nie licząc bitu startu i stopu zmiana na linii SCL następuje tylko podczas wysokiego stanu zegara [Rysunek 5.5] [7] długość linii może wynosić do kilku metrów co definiowane jest jej pojemnością 400pF [16].

5.3 UDP

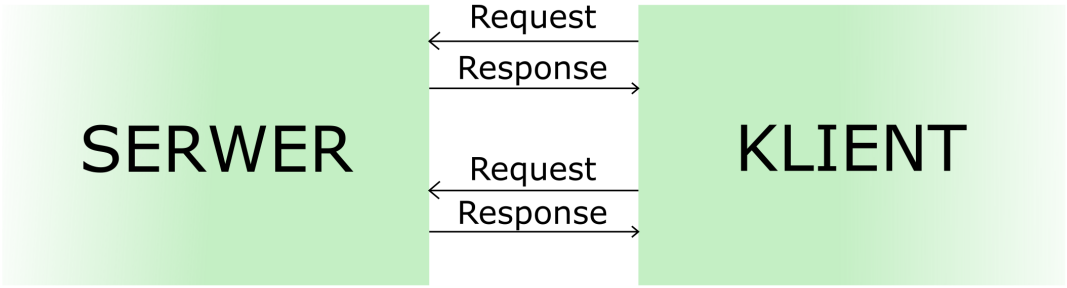
UDP - User Datagram Protocol co dosłownie tłumaczymy na protokół pakietów użytkownika, jest jednym z dwóch najbardziej znanych internetowych protokołów transportowych. Pakiety UDP mogą być wysyłane bezpołączeniowo tj. wysyłający nie potrzebuje potwierdzenia otrzymania Datagramu od odbierającego, co skutkuje szybkim przesyłaniem informacji ale bez gwarancji poprawnego odbioru. Dla naszych zastosowań *UDP* rozważane jest do zapewnienia komunikacji między jednopłytkowym komputerem *Raspberry Pi* zapewniającym odczyt z enkodera i zadawanie prędkości a programem *Matlab* służącym jako kontroler.



Rysunek 5.6. Zasada działania protokołu UDP, źródło: Opracowanie własne

5.4 TCP

TCP - Transmission Control Protocol co dosłownie tłumaczymy na protokół sterowania transmisją, jest drugim z wyżej wspominanych internetowych protokołów transportowych. Serwer nasłuchuje połączenia od klienta i po nawiązaniu go zaczyna wysyłać pakiety. Główną różnicą w stosunku do *UDP* jest fakt, że w tym protokole potrzebujemy potwierdzenia dotarcia pakietu do drugiej strony, zapewnia to bezpieczną transmisje bez utraty pakietów między procesami.

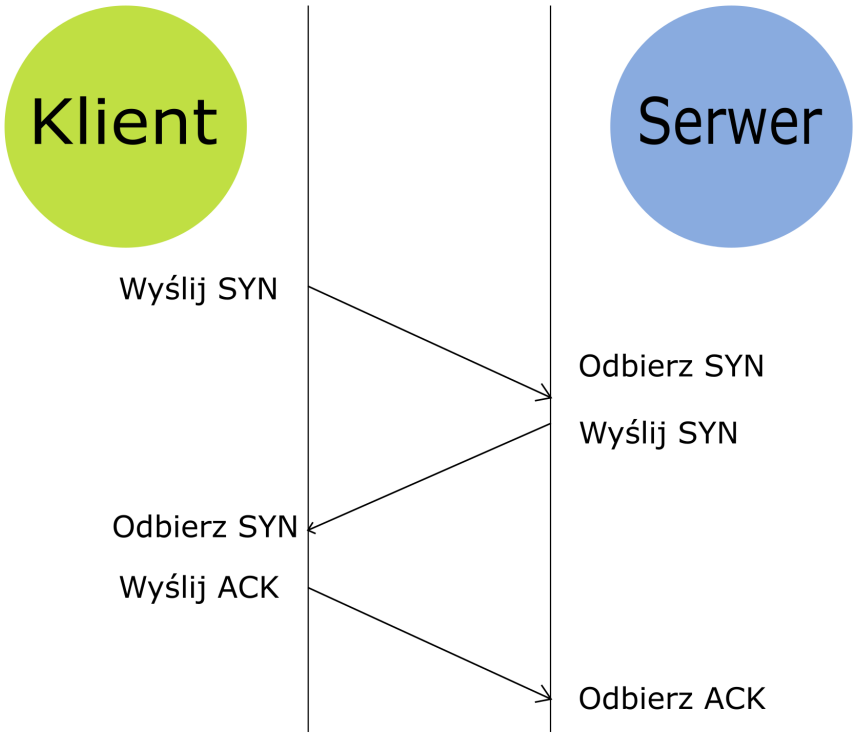


Rysunek 5.7. Zasada działania protokołu TCP, źródło: Opracowanie własne

5.4.1 Three-way Handshake

To procedura nawiązywania połączenia Klienta z Serwerem która odbywa się w 3 krokach. Polega na wysyłaniu flag bitowych i jest domyślna dla protokołu TCP/IP wg. standardu *RFC793* [21]:

- Krok pierwszy (SYN) Polega na wysłaniu przez klienta *SYN Synchronize Sequence Number* które informuje serwer o tym, że klient chce rozpocząć ciąg komunikacyjny.
- Krok drugi (SYN+ACK) Serwer odpowiada klientowi ustawionymi bitami *SYN-ACK*
- Krok trzeci (ACK) Klient potwierdza odpowiedź serwera za pomocą *ACK Acknowledgement* i oboje nawiązują *niezawodne* połączenie

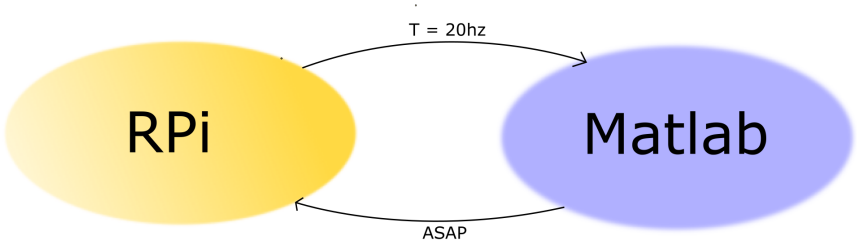


Rysunek 5.8. Zasada Three-way Handshake, źródło: Opracowanie własne

5.5 Implementacja

5.5.1 Założenia UDP

Serwer działający na Raspberry Pi domyślnie łączy się za pomocą kabla typu *8P8C* [11] w standardzie Ethernet by utrzymać możliwie jak najbardziej niezawodną (zwłaszcza zważywszy na protokół UDP) i szybką komunikacje. Serwer ma wysyłać dane co 50ms niezależnie od odpowiedzi klienta który powinien przesłać odpowiedź tak szybko jak jest w stanie tj. w momencie zakończenia obliczeń . Testy polegają na przypisaniu stanu wysokiego na czas przetwarzania przez *PC* w środowisku *Matlab* na porcie *GPIO* w *RPi*, [5] a następnie pomiaru poprawności wysyłania pakietów za pomocą oscyloskopu. Po takich próbach testowana jest sprawność działania układu przy przeniesieniu do standardu Wi-Fi przy pomocy routera.



Rysunek 5.9. Założenia pracy układu przy protokole UDP źródło: Opracowanie własne

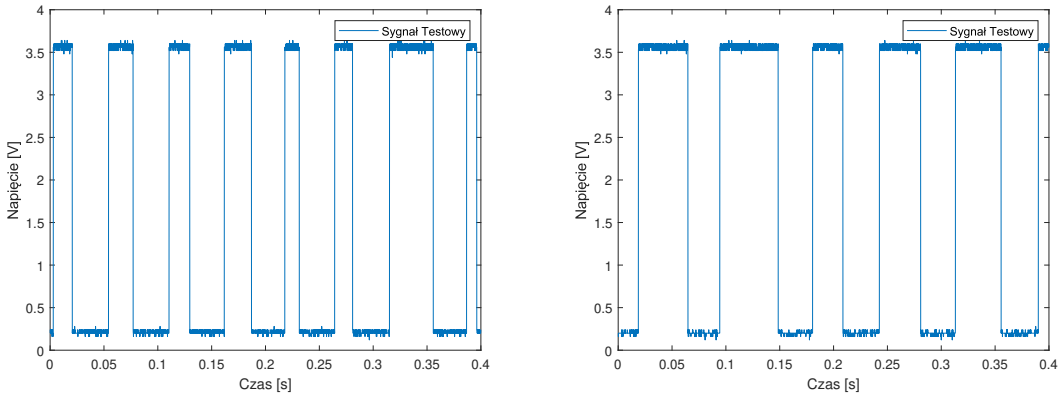
5.5.2 Wyniki testów UDP

Szybkość działania układu

Test wskazuje w stanie wysokim czas przetwarzania obliczeń w *Matlab* a w stanie niskim czas pracy serwera na *RPi*.

Ethernet

Dla ogółu okres wysyłania próbek wynosił zakładane 50Hz [Rysunek 5.10a] choć zdarzały się sytuacje gdzie spadał do wartości równymi nawet 80ms odstępu między wysłaniami [Rysunek 5.10b]. Średni czas przetwarzania obliczeń przez *Matlab* wyniósł $0.02967s \approx 30ms$ gdzie najszybszy czas był równy 13ms a najdłuższy około 50 ms przez co odrzucając duże odchyły możemy przyjąć, że standardowy czas procesowania przez PC w środowisku*Matlab* wynosi 20ms.



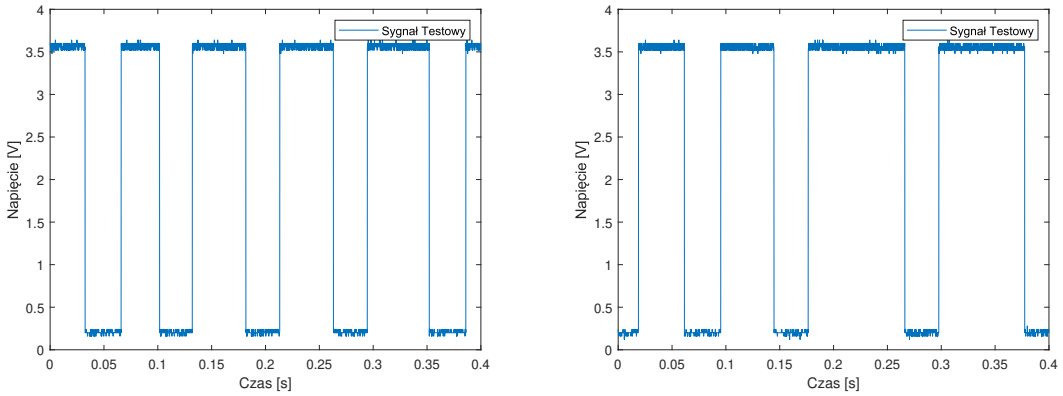
(a) Widok lepszych czasów przy pomiarze. *(b) Widok gorszych czasów przy pomiarze.*

Rysunek 5.10. Wyniki testu dla Ethernet/UDP

Wi-Fi

Dla WiFi czas przetwarzania odpowiednio się zwiększył do około 50ms [Rysunek 5.11a] w maksymalnej wartości 88ms [Rysunek 5.11b], minimalnej 36ms, może to wynikać chociażby z utraty pakietów lub, jakości przesyłu przez router. W celu wery-

fikacji procentu straconych pakietów należy wykonać następny test, który sprawdzi niezawodność protokołu przy komunikacji Wi-Fi.



(a) Widok lepszych czasów przy pomiarze. *(b) Widok gorszych czasów przy pomiarze.*

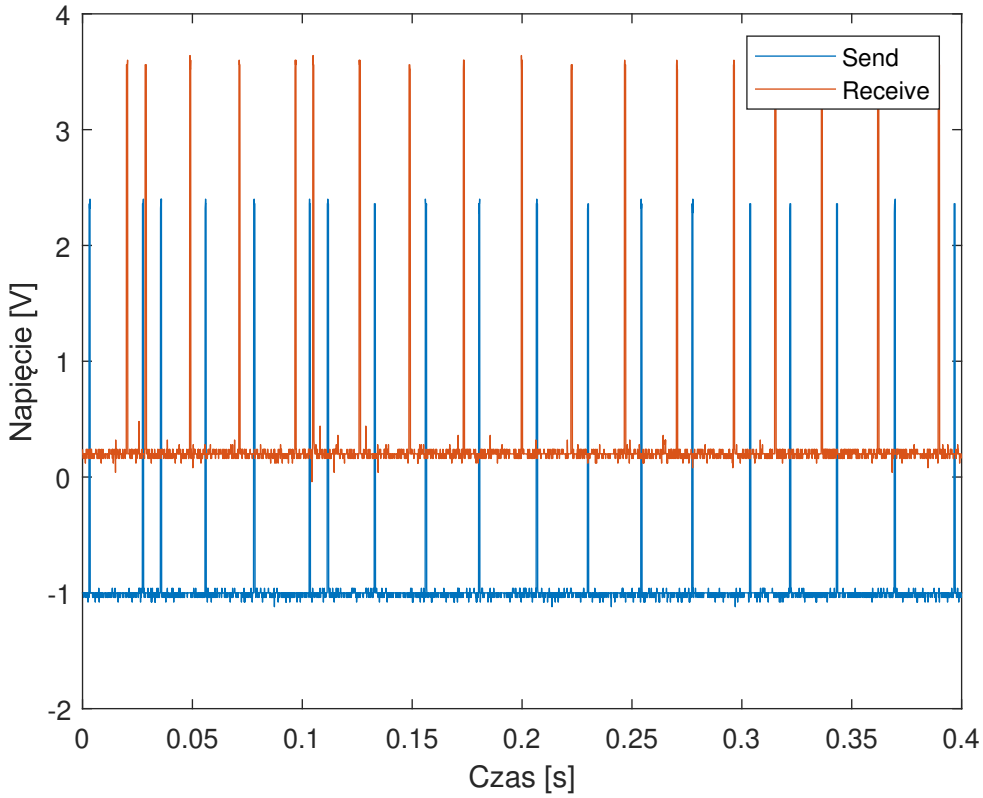
Rysunek 5.11. Wyniki testu dla Ethernet/Wi-Fi

Zgubione pakiety

Test miał polegać na przypisaniu wysokiego stanu na krótki czas na wyjście GPIO w momencie wysłania pakietu i na inne wyjście generalnego użytku przy przybyciu datagramu i zbadaniu wyjść przy użyciu dwóch kanałów oscyloskopu cyfrowego. W celu sprawdzenia poprawności przybycia pakietów układy miały działać szybciej niż w przypadku pierwszego testu by jeszcze bardziej sprawdzić granice i niezawodność pracy układu.

Ethernet

Przy wynikach testów uśredniony czas wysłania odpowiedzi wynosił 21.9ms [Tabela 5.1] co daje okres około 45Hz jednak przy odchyłach sięgających 5ms w górę i 14ms w dół. Kolejno przy czasach odpowiedzi okres również około 45Hz przy podobnych odchyłach [Tabela 5.1]. Datagramy spełniły warunek poprawnego dotarcia przed wysłaniem kolejnego pakietu, a jako, iż nie możemy jednoznacznie powiedzieć o braku straconych pakietów przy badaniu oscyloskopem możemy spodziewać się, że niedotarcie pakietów było niewielkie lub znikome, a układ działał poprawnie i był w stanie działać jako regulator.



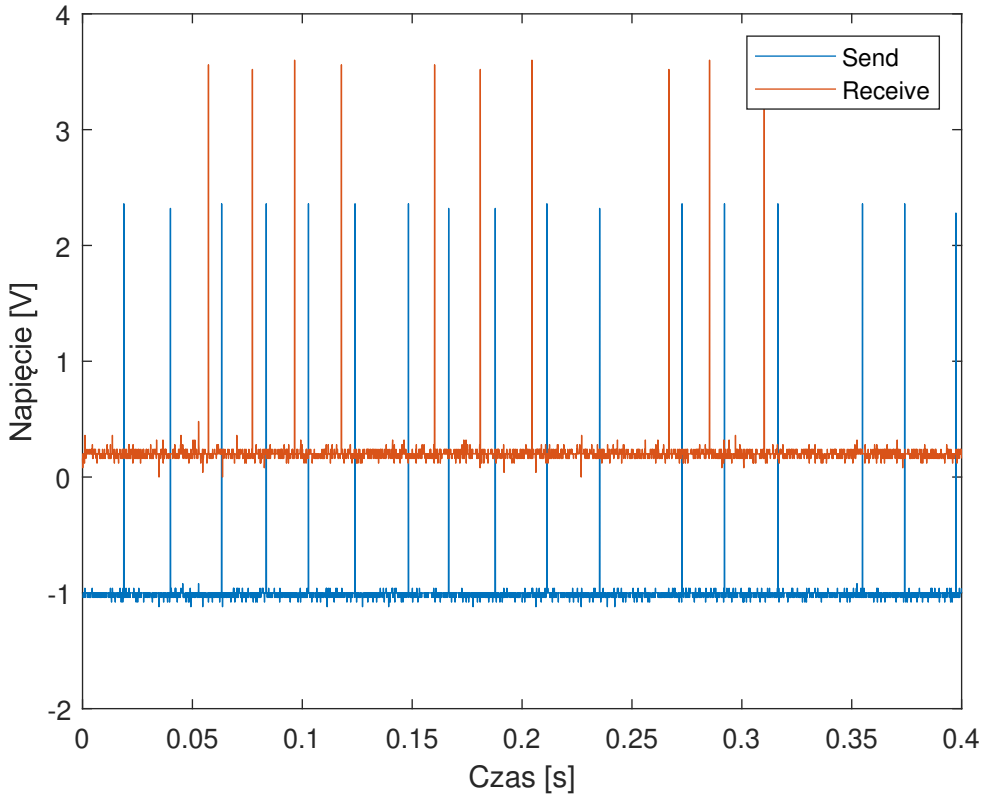
Rysunek 5.12. Przykładowy widok z oscyloskopu przy teście Ethernet źródło: [Opracowanie własne]

	Send	Receive
Średni czas [s]	0.0219	0,0217
Najdłuższy czas [s]	0,0272	0,0273
Najkrótszy czas [s]	0,008	0,0081

Tabela 5.1. Wartości z testów Ethernet

Wi-Fi

Przy testach z użyciem routera dla czasów wysyłania możemy zaobserwować znaczne zwiększenie najkrótszego czasu i niewielki wzrost uśrednionego czasu równego około 24ms [Tabela 5.2] co daje okres wartości 41Hz, najdłuższy czas: 38ms, najkrótszy czas: 18ms czyli odchyły rzędu 20ms, a dla czasów odpowiedzi kolejno: średni czas: 21ms, najdłuższy czas: 25ms, najkrótszy czas: 18.5ms. Niestety na widoku oscyloskopu [Rysunek 5.13] możemy zauważyć, że duża część pakietów (w tym przypadku z widocznych na 16 wysłanych 10 przybyło z powrotem do serwera) z odpowiedzią nie dotarła do *RPi*, co czyni go niezadowalającym przy pracy z Aeropendulum.



Rysunek 5.13. Przykładowy widok z oscyloskopu przy teście Wifi źródło: Opracowanie własne

	Send	Receive
Średni czas [s]	0.0237	0,0212
Najdłuższy czas [s]	0,0385	0,0249
Najkrótszy czas [s]	0,0183	0,0185

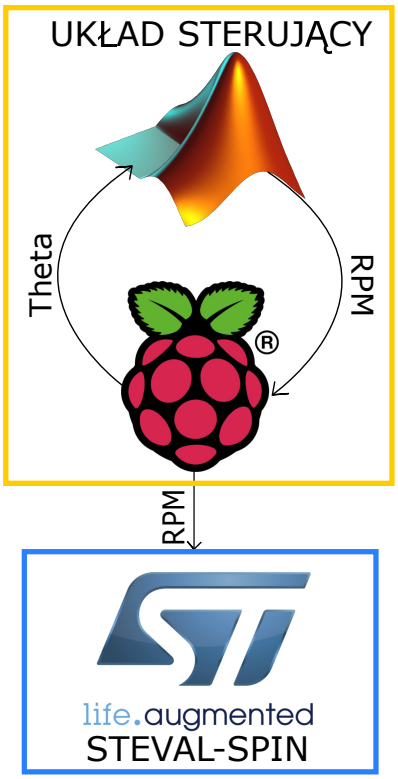
Tabela 5.2. Wartości z testów Wi-Fi

Podsumowanie Testów

O ile protokół UDP, może sukcesywnie być, używany przy połączeniu kablem *Ethernet*, to jego zastosowanie przy użyciu routera i *WiFi*, może być ryzykowne. Utrata pakietów wydaję się problemem ze względu na wydłużenie czasu regulacji, dlatego zostanie wykonany serwer działający z protokołem *TCP/IP* przeznaczony głównie do użycia z siecią bezprzewodową

5.5.3 Założenia TCP

W przypadku komunikacji z wykorzystaniem protokołu *TCP/IP* klient powinien wysłać wiadomość do nasłuchującego serwera *Raspberry Pi* o podjęciu komunikacji, następnie odebrać segment odczytu z enkodera, przetworzyć w środowisku *Matlab* i wysłać z powrotem do serwera. Na strony komunikacji nie zostanie narzucony żaden okres czasowy - Model uczony jest z częstotliwością 100 Hz - gdzie przy wykorzystaniu routera wydaje się do czas niemożliwy do osiągnięcia. Stąd proces powinien odbywać się tak szybko jak to możliwe. Następnie *RPi* wysyła pakiet do *PC* ten przetwarza go i od razu daję odpowiedź do przekazania dla sterownika. Główne powody implementacji takiego rozwiązania to zapobieganie stratom pakietów przy komunikacji.

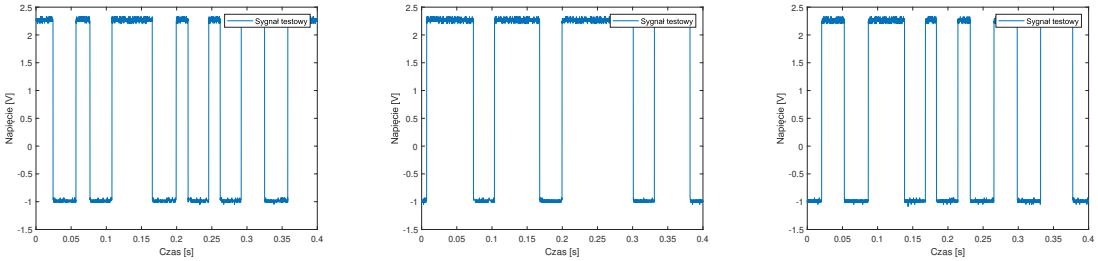


Rysunek 5.14. Plan sterowania układu, źródło: Opracowanie własne

5.5.4 Wyniki testów TCP dla Wi-Fi

Szybkość działania

Czas przetwarzania w stosunku do UDP/WiFi spadł do średnich 40ms [Rysunki 5.15a, 5.15c] a uśredniony okres wynosił 72ms czyli 14Hz gdzie najszybszy czas wysłania następnego segmentu wyniósł 45ms [Rysunek 5.15a] a najdłuższy 130ms [Rysunek 5.15b], co jest wyjątkowo dużym odchyleniem i nie zaobserwowano jego częstego powtarzania.

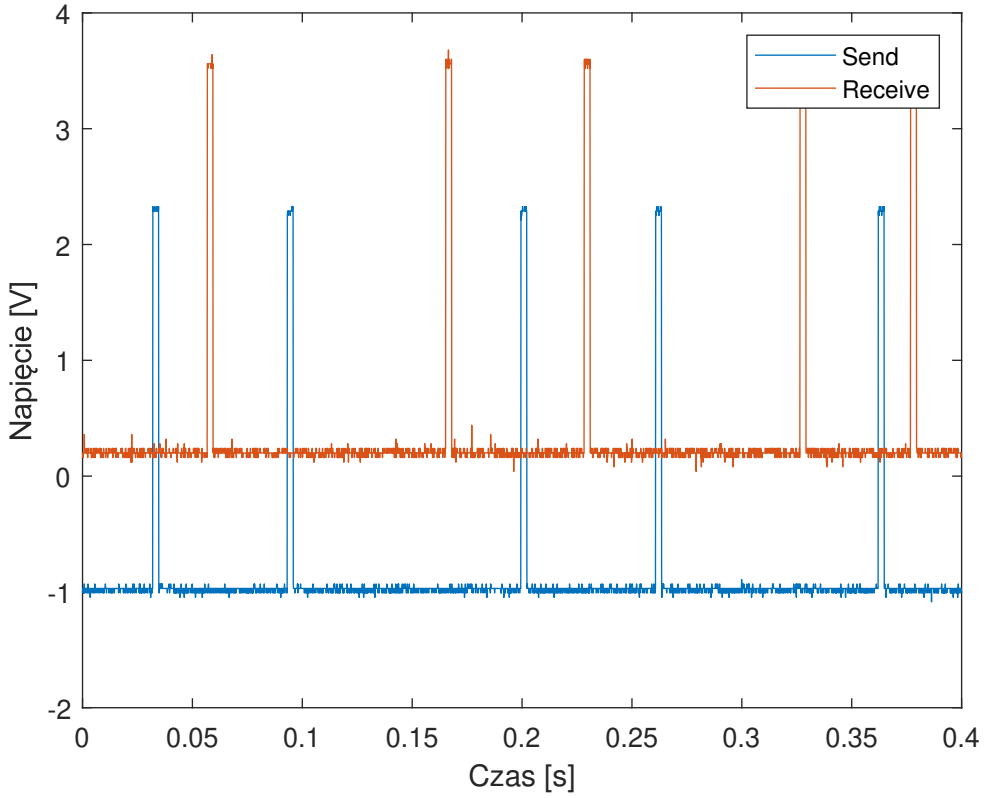


(a) Widok przeciętnych czasów przy pomiarze. *(b) Widok gorszych czasów przy pomiarze.* *(c) Widok przeciętnych czasów przy pomiarze.*

Rysunek 5.15. Wyniki testu dla Wi-Fi/TCP

Zgubione Pakiety

Zgodnie z oczekiwaniami nie zaobserwowano utraty pakietów. Średni czas między wysłaniami wynosił 74ms a między odpowiedziami 71ms [Tabela 5.3, Rysunek 5.16] co jest mniejszym taktowaniem niż zakładaliśmy, ale przy warunku poprawności dotarcia segmentów, może być wystarczające. Najdłuższy czas między wysyłaniami wyniósł 106ms, najkrótszy 58ms [Tabela 5.3], analogicznie dla odpowiedzi wartości te wynosiły: Max: 98ms Min: 52ms [Tabela 5.3].



Rysunek 5.16. Przykładowy widok z oscyloskopu przy teście Wi-Fi źródło: Opracowanie własne

	Send	Receive
Średni czas [s]	0,0744	0,0714
Najdłuższy czas [s]	0,1062	0,0982
Najkrótszy czas [s]	0,0.0581	0,0527

Tabela 5.3. Wartości z testów Wi-Fi

Podsumowanie Testów

Średnie czasy jak i odchyły [Tabela 5.3] pomiędzy operacjami *Send* a *Receive* są do siebie bardzo zbliżone, co ukazuje stabilność układu. Nie stwierdzono żadnej utraty pakietów co oznacza, że protokół *TCP/IP* może być zastosowany do komunikacji bezprzewodowej *Wi-Fi*.

5.5.5 Programowa

Serwer - Raspberry Pi

W początkowej wersji serwer był pisany w języku Python jednakże z powodu problemów z odbieraniem danych szyfrowanych przez *Matlab*, Kod programu pisany jest w języku C\C++ na systemie typu *UNIX - Raspberry Pi OS* z pomocą bibliotek *arpa/inet.h* [1] oraz *sys/socket.h* [28].
Aby użyć biblioteki inet odpowiednim podejściem jest zdefiniowanie struktury [Listing 5.1].

listing 5.1. Struktura adresu serwera

```
01. struct sockaddr_in server =
02.     {
03.         .sin_family = AF_INET,
04.         .sin_port = htons( SERVER_PORT )
05.     };
```

Następnie sprawdzana jest poprawność działania i brak błędów w programie poprzez wywoływanie instrukcji warunkowych zwracających błąd.

Startowanie serwera - Sprawdzenie ewentualnych błędów

- *inet_pton()* - Odpowiada za konwersje adresu z formy tekstowej na binarną [8]
- *socket()* - Funkcja tworzy węzeł końcowy (ang. endpoint) dla komunikacji i tworzy dla niego deskryptor pliku [26]
- *bind()* - Funkcja przypisuje adres do wcześniej stworzonego gniazda [**bind**] na podstawie struktury [5.1]
- *recvfrom()* - Wykorzystana w tym przypadku do odebrania sygnału od klienta o rozpoczęciu transmisji [23]

Startowanie serwera - Różnice przy użyciu TCP

Dodatkową operacją która należy wykonać aby wystartować serwer z użyciem *TCP* jest nasłuchiwanie za pomocą funkcji *listen()* [12]. Funkcja ta również jest wywoływana z poziomu instrukcji warunkowej mającej zwrócić ewentualny błąd.
Każda z powyższych funkcji zwraca 0 lub -1 w zależności od tego czy dane operacje udało się wykonać. Do kontroli poprawności połączenia wykorzystałem funkcję z biblioteki standardowej *perror()* [20] która wypisuje komunikat błędu i zostaje wywołana jeżeli wartość zwracanej funkcji jest mniejsza od zera, Jeśli żadna z powyższych funkcji (w podanej kolejności) nie zwróci, błędu następuje przejście do pętli głównej serwera.

Działanie serwera

Do obsługi samej istoty komunikacji używamy 2 funkcji zawartych w instrukcjach warunkowych(w celu obsługi błędów jak wyżej) send() oraz recv() [Listing 5.2] które przyjmują adres gniazda, wspólny bufor w którym zapisywane są przesłane bity oraz jego rozmiar. Funkcje te po nawiązaniu połączenia wykonują się w nieskończonej pętli programu zapewniając transfer danych między stronami.

listing 5.2. Funkcje send i recv()

```
01. if (send( clientSocket, buffer, strlen( buffer ), 0 ) <= 0)
02.     {
03.         perror("sendto() ERROR");
04.         exit(5);
05.     }
06.
07. if (recv( clientSocket, buffer2, sizeof( buffer2 ), 0 ) <= 0)
08.     {
09.         perror("recvfrom() ERROR");
10.         exit(4);
11.     }
```

WiringPi - Odczyt z enkodera

Do odczytu z enkodera używamy biblioteki z innej pracy [18] po poprawkach skalowania na podstawie biblioteki producenta przygotowanej dla *Arduino* [25]. Aby odczytać kąt z magistrali I^2C używam funkcji *GetRawAngle()* [Listing 5.3]

listing 5.3. Pobranie bitowej pozycji kąta

```
01. /*******************************************************
02. * Gets raw value of magnet position.
03. * start, end, and max angle settings do not apply
04. * @return value of raw angle register
05. *******************************************************/
06. uint16_t getRawAngle()
07. {
08.         uint16_t result;
09.         readTwoBytes(&result,_raw_ang);
10.         return result;
11. }
```

Otrzymany kąt należy pomnożyć przez 0.087 i używam do tego funkcji *convertRawAngleToDegrees()*[Listing 5.3]

listing 5.4. Konwersja do stopni

```
01. float convertRawAngleToDegrees(uint16_t newAngle)
02. {
03.   /* Raw data reports 0 - 4095 segments, which is conv - 0.087
        of a degree */
04.   float retVal = (float)(newAngle) * conv;
05.   return retVal;
06. }
```

Niezbędnym było przeprowadzenie operacji normalizacji. Ponieważ enkoder zwracał wartości w przedziale $< 0 : 360 >$ a model symulacyjny był uczony z wykorzystaniem zakresu $< -180, 180 >$. Do tego celu została wykorzystana funkcja przyjmująca wartość aktualną oraz skrajne [5.5].

listing 5.5. Funkcja normalizacji

```
01. float normalize( const float value, const float start, const
      float end )
02. {
03.   const float width       = end - start   ;
04.   const float offsetValue = value - start ;
05.
06.   return ( offsetValue - ( floor( offsetValue / width ) * width
        ) ) + start ;
07. }
```

RPi - obsługa ramki MCP

Aby zadać prędkość w obrotach/minutę wykorzystujemy sterownik *STEVAL-SPIN3201* który domyślnie jest w stanie wymusić na silniku działanie w określonych *RPM*. Komendy do mikroprocesora można wysłać za pomocą protokołu *MCP - Motor Control Protocol*. Wykorzystana do tego celu została funkcja napisana w języku C [18] do której możemy przekazać liczbę obrotów na minutę, efektem będzie przesłanie za pomocą *UART* instrukcji dla procesora sterownika. Funkcja ta wykorzystuje strukturę *Frame* zawierająca wszystkie niezbędne informacje o silniku i jego stanie, potrzebne do skonstruowania odpowiedniej ramki [5.6].

listing 5.6. Funkcja zadająca prędkość silnika

```
01. UART_STATUS SetMotorRefSpeed(int ref, int motorId, UART uart,
      Frame* f)
02. {
03.     Frame  cmd = Frame(1,FRAME_CODES::SET,STEVAL_REGISTERS::
          RAMP_FIN_SPEED,(int)STEVAL_REGISTERS_LEN::RAMP_FIN_SPEED
          , ref);
04.     return send(cmd,uart,f);
05. }
```

Matlab

Dane przesyłane przez serwer występują w postaci bajtów. Dla naszych potrzeb najprościej je zamienić na string który dalej będziemy procesować, dla tego celu napisana została funkcja [Listing 5.7].

listing 5.7. Funkcja przetwarzająca bajty na string

```
function s = decodeStringData(a)
a_t = a';
a_u = native2unicode(a_t);
s = str2double(a_u);
end
```

Aby obsłużyć naszego wytrenowanego agenta należy wygenerować funkcje *evaluatePolicy()* aby to zrobić musimy wpisać odpowiednie komendy [Listing 5.8] w *Command Window* środowiska *Matlab*.

listing 5.8. Operacje potrzebne do wygenerowania evaluatePolicy()

```
agent = load("Agent188.mat")
generatePolicyFunction(agent)
```

Wygenerowana funkcja [Listing 5.9] ma wszystkie niezbędne dane o naszym agencie to jest: zestaw akcji jakie może podjąć agent, informacje o potrzebnych obserwacjach itd. Po podaniu wektora obserwacji jako wartość zwracaną otrzymujemy akcję którą należy podać dalej do serwera.

listing 5.9. Funkcja pozwalająca przewidzieć akcję nauczonego agenta

```
function action1 = evaluatePolicy(observation1)
%#codegen

% Reinforcement Learning Toolbox
% Generated on: 25-Jan-2022 14:19:49

actionSet = [-2000;-1000;0;1000;2000;3000;4000];
% Select action from sampled probabilities
probabilities = localEvaluate(observation1);
% Normalize the probabilities
p = probabilities(:)'/sum(probabilities);
% Determine which action to take
edges = min([0 cumsum(p)],1);
edges(end) = 1;
[~,actionIndex] = histc(rand(1,1),edges); %#ok<HISTC>
action1 = actionSet(actionIndex);
end
%% Local Functions
function probabilities = localEvaluate(observation1)
persistent policy
if isempty(policy)
        policy = coder.loadDeepLearningNetwork('agentData.mat',
            'policy');
end
observation1 = observation1(:)';
probabilities = predict(policy, observation1);
end
```

Strona kliencka wykonana w środowisku *Matlab* powinna być maksymalnie prosta tak, aby jak najbardziej skrócić czas odpowiedzi do *RPi*. Przy obu protokołach kod wygląda analogicznie [Listing 5.10, 5.11] Klient czeka na wartość kąta wychylenia, przetwarza ją i odsyła do serwera. Różnicami jest deklaracja obiektu - *udp()* dla protokołu *UDP* i *tcpip()* oraz w tym, że dla opcji bezpołączeniowej musimy wysłać dodatkową wiadomość przed rozpoczęciem ciągłego nadawania, która zostanie odebrana przez drugą stronę jako sygnał do wystartowania.

listing 5.10. Skrypt klienta UDP

```
y = []; % Vector of theta's
yref = 35; % Ref Value
n = 2000000000; % nr of samples to take from server

u = udp('192.168.1.248',6789); %IP Server
fopen(u); %Open connection
fwrite(u,'Connection␣Succeed') %Establish connection

for i = 1:n
A = fread(u,100); %Receive data
y(i) = decodeStringData(A);
theta = y(i);

obs = [theta, sin(theta),cos(theta), yref-theta]; %Calculate
    Observation
Action = evaluatePolicy(obs);
Scaled_Action = Action/100;
fwrite(u,num2str(Scaled_Action)) %Send data
end

fclose(u) %Close connection
```

listing 5.11. Skrypt klienta TCP

```
y = []; % Vector of theta's
yref = 35; % Ref Value
n = 1000000; % nr of samples to take from server
t = tcpip('192.168.0.106',6789); %IP of Server

fopen(t); % Open and establish connection
A = fread(u,7); %Receive data
for i = 1:n

y(i) = decodeStringData(A);
theta = y(i);

obs = [theta, sin(theta),cos(theta), yref-theta]; %Calculate
    Observation
Action = evaluatePolicy(obs);
Scaled_Action = Action/100;
fwrite(u,num2str(Scaled_Action)) %Send data

end
fclose(t) %Close connection
```

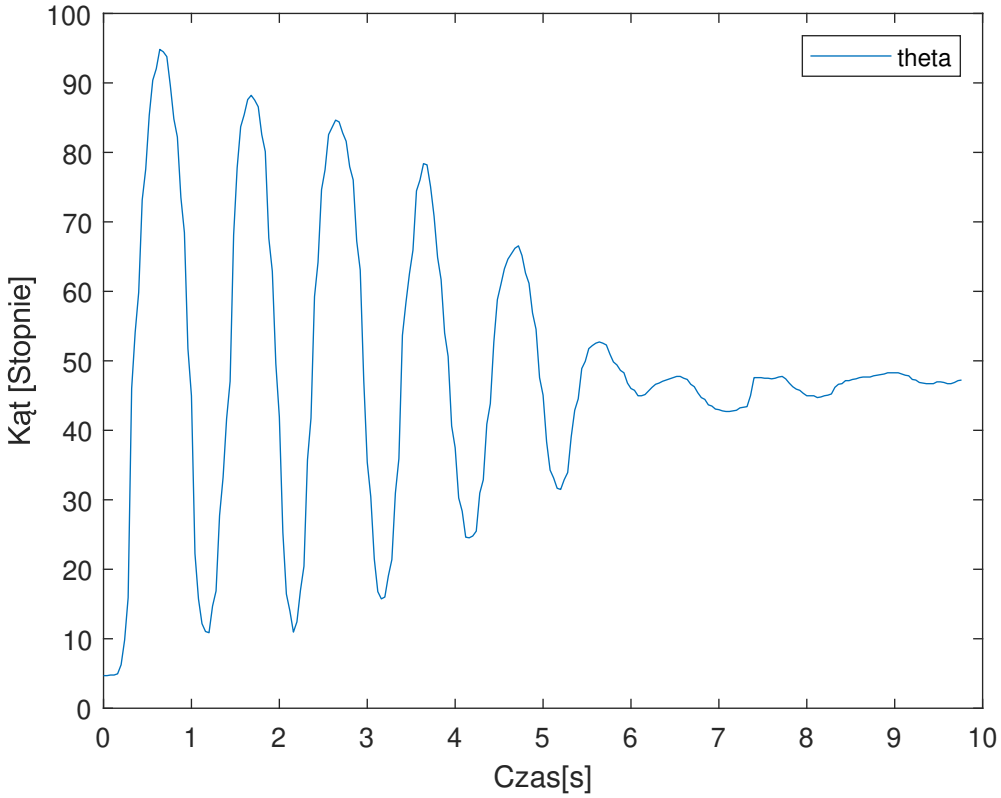
W związku z informacją o planowanym zakończeniu wsparcia starych bibliotek z obiektami *udp()* i *tcpip()* i zatwierdzeniem protokołu TCP/IP jako docelowego dla naszego projektu, stworzono wersję kodu [Listing 5.12] wykorzystująca nowe rozwiązania typu *tcpclient()*. Dodatkową zaletą tego obiektu jest to, że przy odbieraniu danych możemy od razu dokonać konwersji na interesujący nas typ co eliminuje konieczność użycia funkcji *decodeStringData()* [Listing 5.7].

listing 5.12. Uaktualniony skrypt klienta TCP

```
y = []; % Vector of theta's
yref = 35; % Ref Value
n = 1000000; % nr of samples to take from server

t = tcpclient("192.168.0.106",6789) %Open and establish
    connection
for i = 1:n
obs = [theta, sin(theta),cos(theta), yref-theta]; %Calculate
    Observation
Action = evaluatePolicy(obs);
Scaled_Action = Action/100;
write(t,num2str(Scaled_Action)); %Send data

end
clear t % Close connection
```

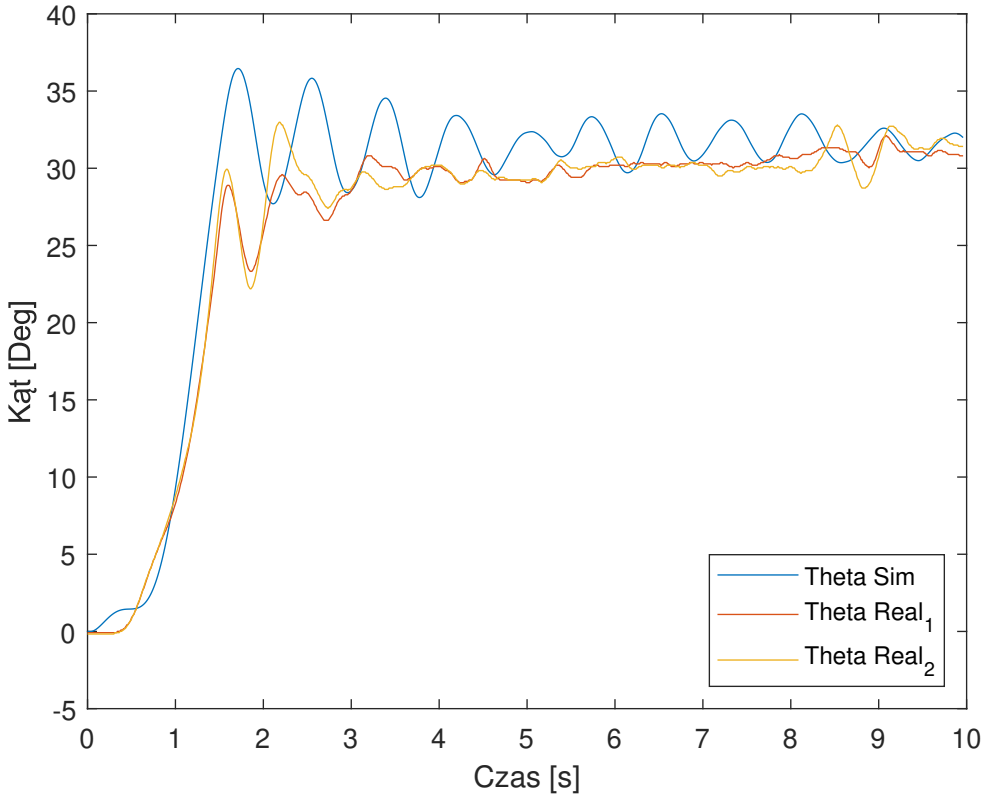
5.6 Podsumowanie rozdziału

Celem komunikacji było użycie modelu z badań symulacyjnych stworzonego dla środowiska *Matlab* jako regulatora prędkości. Po stworzeniu serwera, klienta i zaimplementowaniu predykcji agenta otrzymaliśmy odstające wyniki [Rysunek 5.17].



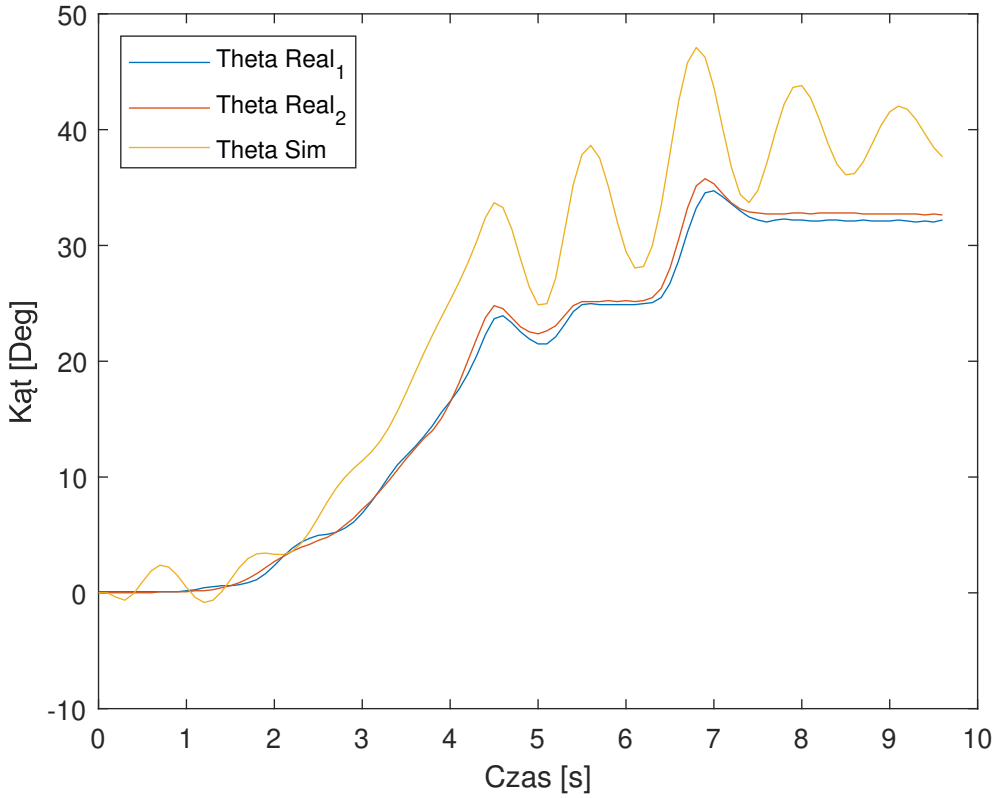
Rysunek 5.17. Zmiana kąta obrotu dla najlepszego Agenta symulacyjnego z pomocą Wi-Fi. źródło: Opracowanie własne

W celu porównania dla tej samej funkcji polityki zebrano wektor zadawanych prędkości i zadano za pomocą *RPi* z próbkowaniem 100Hz czyli częstotliwością odpowiadająca nauczaniu agenta, po wykonaniu kilku prób zebrano wyniki dla wartości referencyjnej 30 [Rysunek 5.18] z których wynika, że regulator na obiekcie rzeczywistym jest w stanie działać poprawnie i w miarę zgodnie z symulacyjną wersją.



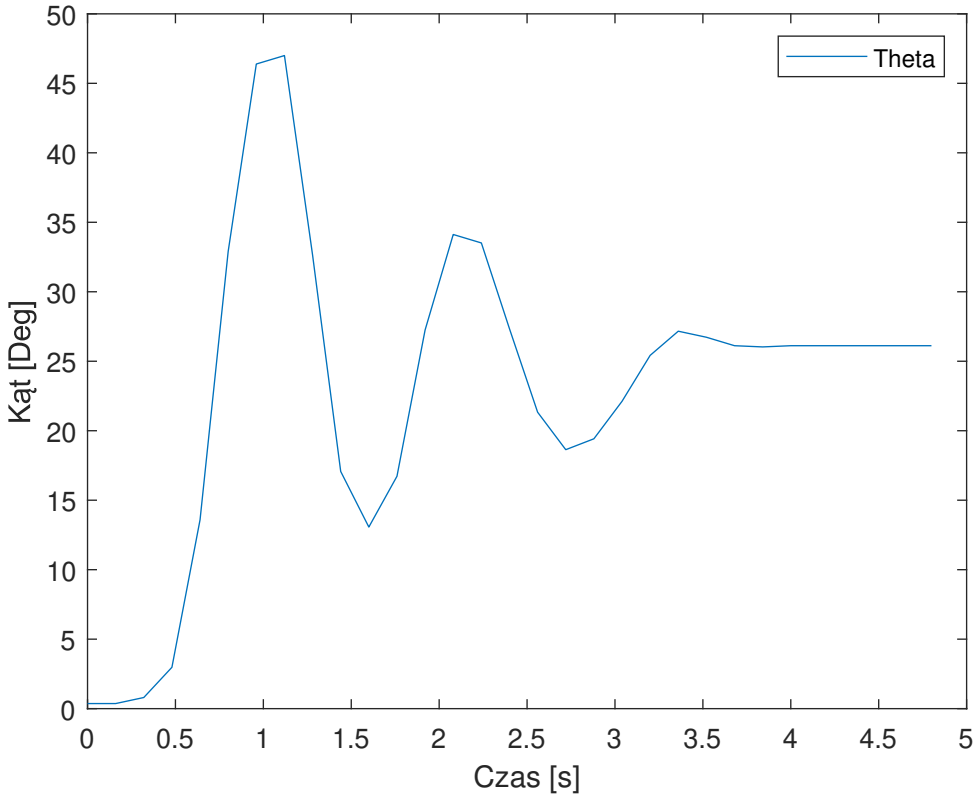
Rysunek 5.18. Zmiana kąta obrotu dla wektora prędkości najlepszego Agenta źródło: Opracowanie własne

Dla użytych protokołów komunikacyjnych wydaje się, że regulator uczony symulacyjnie przez opóźnienia odpowiada zbyt wolno (średnio 14 Hz) akcjami co prowadzi do reakcji układu podobnej do odpowiedzi skokowej na maksymalną prędkość, przy jednoczesnych zbyt dużych przeregulowaniach. Aby poprawić wyniki regulacji podjęto próbę nauczenia symulacyjnego agenta z próbkowaniem 0.1s co daje 10Hz. Rezultaty były słabsze - agent nie był w stanie zbliżyć do czasu regulacji gęściej próbkowanego odpowiednika natomiast osiągał wartość referencyjną. Przeprowadzono analogiczne testy dla wektora prędkości [Rysunek 5.19] i stwierdzono, że agent w obiekcie rzeczywistym daje nawet mniejsze przeregulowania niż symulacyjnym.



Rysunek 5.19. Zmiana kąta obrotu dla Agenta z wektora prędkości dla stworzonego agenta o niższym próbkowaniu. źródło: Opracowanie własne

Dla tak nauczonego agenta zaimplementowano komunikacje Wi-Fi i uruchomiono z przetwarzaniem kąta przez Matlab - wyniki były lepsze - agent poprawnie wyregulował układ dla 25 stopni lecz z dużymi przeregulowaniami [Rysunek 5.20]. Potencjalnym dalszym krokiem mogła by być implementacja biblioteki *Boost* dla Raspberry Pi dzięki której moglibyśmy stworzyć stabilniejszą czasowo komunikacje, następnie nauczyć model dla możliwego okresu Serwera i powtórzyć badania.



Rysunek 5.20. Zmiana kąta obrotu z pomocą Wi-Fi z dedykowanym agentem. źródło: Opracowanie własne

Rozdział 6

Praca ze stanowiskiem rzeczywistym

Autor: Mikołaj Maciejewski

6.1 Wstęp

Finalnym celem pracy jest sterowanie rzeczywistym obiektem Aeropendulum. W tym celu został opracowany skrypt w języku Python, posiadający środowisko odwzorowujące model Aeropendulum oraz algorytm uczenia ze wzmacnianiem. Wynikami przeprowadzonego uczenia będzie plik nauczonego algorytmu, który posłuży jako regulator obiektu. Następną częścią prac będzie zastąpienie symulacyjnego środowiska obiektem rzeczywistym. Kąt odczytany przez enkoder zostanie wysłany do agenta i na jego podstawienie zostanie obliczona wartość prędkości silnika i przesłaniu jej na sterownik.

6.2 Biblioteki zastosowane w skrypcie

6.2.1 Tensorforce

Tensorforce jest biblioteką, w której zaimplementowane zostały algorytmy uczenia ze wzmocnieniem. Bazuje ona na bibliotece *Tensorflow*. Biblioteka dostarcza prosty w obsłudze interfejs programistyczny, opisany w szerokiej dokumentacji. Ponadto, w swojej bazie posiada algorytm *Vanilla Policy Gradient*, który został wykorzystany jako finalny algorytm uczący. [10]

6.2.2 Gym

Biblioteka *Gym* dostarcza nam interfejs pozwalający do komunikację między algorytmem uczącym a środowiskiem, z którym pracuje. Posiada ona zbiór gotowych do zastosowania środowisk oraz możliwość stworzenia własnego środowiska. W końcowej wersji skryptu biblioteka nie została zastosowana ze względu na zmianę algorytmu uczącego, który wymusił porzucenie biblioteki *stablebaselines3* na rzecz *TensorForce*. Szybka zmiana biblioteki była możliwa dzięki doświadczeniu zebranemu na pracy z biblioteką *Gym*. [3]

6.3 Model Aeropendulum jako środowiska

6.3.1 Inicjalizacja klasy

listing 6.1. Inicjalizacja klasy

```
# integrate actions
def func0(y, t, a):
    dydt = a
    return dydt

def func1(y, t, a, b, c, u):
    dydt1 = y[1]
    dydt2 = -a * y[1] - b * math.sin(y[0]) + c * u
    dydt = [dydt1, dydt2]
    return dydt

class AeropendulumEnvironment(Environment):
    def __init__(self):
        self.rpm = 0
        self.theta = 0.0
        self.theta_old = 0.0
        self.thetaD = 0.0
        self.thetaD_old = 0.0
        self.thetaDD = 0.0

        self.y1 = 0.0
        self.y2 = 0.0, 0.0

        self.outdata = np.reshape([self.rpm, self.theta, self.
            theta_old, self.thetaD, self.thetaD_old, self.
            thetaDD, self.y1],(7,))

        # Parameters
        self.m = 0.120  # masa wahadla [kg]
        self.g = 9.81  # przyspieszenie grawitacyjne [m/s^2]
        self.c = 0.007  # tarcie wiskotyczne [Nms/rad]
        self.l = 0.25  # dlugosc wahadla [m]
        self.J = self.m * self.l * self.l  # moment
            bezwladnosci [kgm^2]
        self.d = 0.25  # odleglosc osi od srodka masy
        self.y = 0.0
        self.Ts = 0.01
        # Aero model coef
        self.a = self.c / self.J
        self.b = self.m * self.g * self.d / self.J
        self.c = self.l / self.J
        super().__init__()

    def states(self):
        return dict(type='float', shape=(7,), min_value
            =-800000.0, max_value=800000.0)

    def actions(self):
        #action space -2000:1000:4000
        return dict(type='int', num_values=7)

```

```
48.     def max_episode_timesteps(self):
49.         return super().max_episode_timesteps()
50.
51.     def close(self):
52.         super().close()
```

Listing 6.1 przedstawia nam początek skryptu, posiadający definicje 2 funkcji wykorzystanych w późniejszej części programu do rozwiązywania równań różniczkowych. Klasa *AeopendulumEnvironment* dziedziczy z klasy *Environment*, zdefiniowanej w bibliotece *Tensorforce*. Tworzy nam to zwięzłą strukturę modelu, której funkcje możemy zdefiniować w sposób odpowiadający działaniu naszego środowiska. Klasa została stworzona w osobnym pliku, aby umożliwić wykorzystanie środowiska w różnych skryptach uczących, bez potrzeby kopiowanie ich z skryptu wykorzystanego w naszej pracy.
Tworzenia klasy rozpoczynamy od zdefiniowania funkcji *__init__()*. Funkcja jest konstruktorem, który inicjalizuje podane parametry podczas stworzenia klasy.
Opis parametrów:

- Sekcja *Parametrs* definiuje parametry Aeropendulum, opisujące jego fizyczne cechy,
- Sekcja *Aero model coef* oblicza współczynniki równania różniczkowego, które zostaną wykorzystane w dalszej części skryptu. Zostały zdefiniowane w konstruktorze aby uniknąć powtarzania obliczeń za każdym wywołaniem funkcji step(),
- Pozostałe zmienne zostaną wykorzystane do obliczeń w funkcji *response()*. W związku ze specyfikacją obliczeń (inkrementacja wartości) niektórych parametrów, zostały one umieszczone na zewnątrz funkcji, w której dokonywane są obliczenia.
- Funkcja *actions()* definiuje liczbę akcji (7), jakie można dokonać w środowisku,
- Funkcja *states()* odpowiada za liczbę obserwacji (7), które będą przekazywane do agenta

6.3.2 Funkcja reset()

listing 6.2. Resetowanie środowiska

```
01.     def reset(self):
02.         self.timestep = 0
03.         self.rpm = 0.0
04.         self.theta = 0.0
05.         self.theta_old = 0.0
06.         self.thetaD = 0.0
07.         self.thetaD_old = 0.0
08.         self.thetaDD = 0.0
09.         self.y1 = 0.0
10.         self.y2 = 0.0, 0.0
11.
12.         output_states = [self.rpm, self.theta, self.theta_old,
                self.thetaD, self.thetaD_old, self.thetaDD, self.y1]
13.         out = np.reshape(output_states,(7,))
14.         return out
```

Listing 6.2 przedstawia nam funkcję *reset()*, odpowiadającą za przywrócenie środowiska do swojej początkowej wartości, sprzed ingerencji jakiejkolwiek akcji. Jest to przydatne po zakończonym epizodzie uczenia, po którym nowy epizod powinien zacząć się na świeżym modelu. W funkcji zerujemy wszystkie wartości obserwacji oraz wszystkie zmienne, które przechowują wartości poprzednich próbek. W przypadku pominięcia tego kroku, mimo rozpoczęcia kolejnego epizodu uczenia, model bazowałby na obliczeniach wykonanych w poprzednim epizodzie. W wcześniejszych wersjach skryptu niedopracowanie tej funkcji powodowało błędy w poprawnym przetwarzaniu danych ze względu na powyższy błąd. Rozwiązaniem było utworzenie wielu testów jednostkowych, które pomogły w zlokalizowaniu niezresetowanych zmiennych.

6.3.3 Funkcja response()

Najważniejszą i zarazem najbardziej rozbudowaną funkcją klasy *AeropendulumEnvironment* jest funkcja *response()*. Odpowiada ona za reakcję modelu na zadaną akcje oraz wygenerowanie tablicy obserwacji. Poprawne odwzorowanie modelu obiektu jest niezbędne do przeprowadzenia procesu uczenia, który zaowocuje dobrym agentem zdolnym do regulacji obiektu.
W celach omówienia funkcjonalności, funkcja step została podzielona na 2 listingi, które w finalnym skrypcie stanowią spójną całość.

listing 6.3. Funkcja obserwacji

```
01.     def response(self, action):
02.         # Calc rpm
03.         action_i = float((action-3) * 1000)
04.         #integrate actions
05.         x = odeint(func0, self.y1, [0.0, self.Ts], args=(
                action_i,))
06.         #sum actions to get rpms
07.         self.rpm += float(x[1])
08.         #polyvals to trust
09.         val = 0.0
10.         if self.rpm > 0.0:
11.             val = polyval([0.00000304986021927422,
                    -0.00476887717971010, 2.51670431281220], self.
                    rpm)
12.         else:
13.             val = polyval([-5.46944010985214e-06,
                    -0.00938361577873603, -2.43262552689868], self.
                    rpm)
14.         #deg to rad
15.         val = val * (math.pi / 180)
```

Listing 6.3 rozpoczyna się od zdefiniowania funkcji *response()*, do której przekazujemy argument *action*, będący akcjami generowanymi przez agenta. Zostają one przeliczone na prędkości silnika zdefiniowaną w obrotach na minutę. Zakres dopuszczalnych prędkości to <-2000,4000> z krokiem 1000. Z racji, że dostępne akcje to liczby całkowite z zakresu <0,6>, aby osiągnąć wymagane wartości stosujemy wzór:

$$action = (action - 2) * 1000 \tag{6.1}$$

Przykładowe obliczenia dla $action = 7$:

$$action = (6 - 2) * 1000 = 4000 \tag{6.2}$$

Otrzymana wartość jest przetwarzana za pomocą funkcji *odeint()* z biblioteki SciPy. Jest to funkcja służąca do rozwiązywania równań różniczkowych. Przyjmuje 3 wartości: funkcje będąca równaniem różniczkowym, warunki początkowe oraz wektor czasu. W tym przypadku naszym równaniem różniczkowym jest akcja obliczona w poprzednich krokach skryptu. Na wyjściu funkcji otrzymujemy natomiast zmianę wartości RPM (ang. revolutions per minute - obroty na minutę). Następnie otrzymana wartość jest dodawana do zmiennej RPM.
W następnych krokach wywoływana jest funkcja *polyval(W, x)*, zaimportowana z biblioteki NumPy. Oblicza ona wartość wartość wielomianu W dla dowolnej wartości bądź wektora wartości. W naszym przypadku posiadamy 2 takie wielomiany:

$$W_1(x) = 3.04986021927422x^2 * 10^{-6} - 0.00476887717971010x + 2.51670431281220 \tag{6.3}$$

$$W_2(x) = -5.46944010985214x^2 * 10^{-6} - 0.00938361577873603x - 2.43262552689868 \tag{6.4}$$

Danego wielomianu używamy w zależności od znaku zmiennej RPM - W_1 dla RPM większych od zera oraz W_2 dla pozostałych wartości. Otrzymaną wartość musimy przeskalować do jednostki $\frac{rad}{s}$ (radian na sekundę).

listing 6.4. Funkcja obserwacji

```
01. #thetaDD
02.         self.thetaDD = -self.a * self.thetaD_old - self.b *
               math.sin(self.theta_old) + self.c * val
03.         #calc thetaD and theta
04.         x1 = odeint(func1, self.y2, [0.0, self.Ts], args=(self.
               a, self.b, self.c, val))
05.         self.y2 = x1[1]
06.         #get thetaD
07.         self.thetaD = self.y2[1]
08.         self.thetaD_old = self.thetaD
09.         #get theta
10.         self.theta = self.y2[0]
11.         self.theta_old = self.theta
12.         #rad to deg
13.         self.theta = self.theta * (180 / math.pi)
14.
15.         output_states = [self.rpm, self.theta, self.theta_old,
               self.thetaD, self.thetaD_old, self.thetaDD, self.y1]
16.         out = np.reshape(output_states,(7,))
17.         return out
```

Listing 6.4 przedstawia sposób obliczania przyspieszenia kątowego *thetaDD*, prędkości kątowej *thetaD* i kąta wychylenia wahadła *theta*. Wszystkie zmienne zostaną wykorzystane jako obserwacje w późniejszej części skryptu.
Wartość *thetaDD* obliczana jest na podstawie sprzężenia zwrotnego, dostarczającego wartości zmiennych *thetaD* oraz *theta* obliczonych w poprzednim wywołaniu funkcji. Ponownie wykorzystujemy funkcje *odeint()*. Warto zauważyć dodatkowy parametr, w którym są przekazywane do funkcji argumenty (*args*). Trzy pierwsze stanowią współczynniki równania $\ddot{\theta}$. Ostatnim argumentem jest sygnał wejściowy, reprezentowany zmienną *val*.
Mamy do czynienia z układem drugiego stopnia. Aby móc taki układ rozwiązać, należy ułożyć równanie różniczkowe drugiego rzędu. Zostało ono przygotowane na podstawie modelu obiektu w Simulinku. Współczynniki a, b oraz c zostały obliczone we wcześniejszej części programu i odpowiadają fizycznym parametrom silnika.

$$\ddot{\theta} + a * \dot{\theta} + b * \theta - c * u = 0$$

Następnie przekształcamy je na układ dwóch równań pierwszego rzędu, które tworzą funkcje *function*.

$$\dot{\theta} = y'$$

$$\ddot{\theta} = -a * y' - b * y + c * u$$

Po rozwiązaniu układu otrzymujemy wartości dla zmiennych *thetaD* oraz *theta*. Na końcu następuje przeliczenie wartości *theta* z radianów na stopnie.

6.3.4 Funkcja reward_compute()

listing 6.5. Funkcja nagrody

```
01.     def reward_compute(self):
02.         theta_ref = 30
03.         theta_err = theta_ref - self.theta
04.         delta = -0.001*(theta_err*theta_err)
05.         return delta
```

Funkcja *reward_ compute()* służy do obliczania wartości funkcji nagrody. Tak jak w modelu symulacyjnym w Simulink, stanowi ona kwadrat uchybu kąta θ pomnożony o odpowiedni współczynnik. Ze względu na wcześniejsze wyniki symulacyjne funkcja nagrody pozostała w swojej prostej, niezmienionej formie.

6.3.5 Funkcja execute()

listing 6.6. Obliczenie wartości kąta wychylenia i jego pochodnych

```
01.     def execute(self, actions):
02.         assert actions == 0 or actions == 1 or actions == 2 or
                actions == 3 or actions == 4 or actions == 5 or
                actions == 6 or actions == 7
03.         ## Increment timestamp
04.         self.timestep += 1
05.         ## Update theta
06.         self.outdata = self.response(actions)
07.         ## Compute the reward
08.         reward = self.reward_compute()
09.         terminal = False
10.         return self.outdata, terminal, reward
```

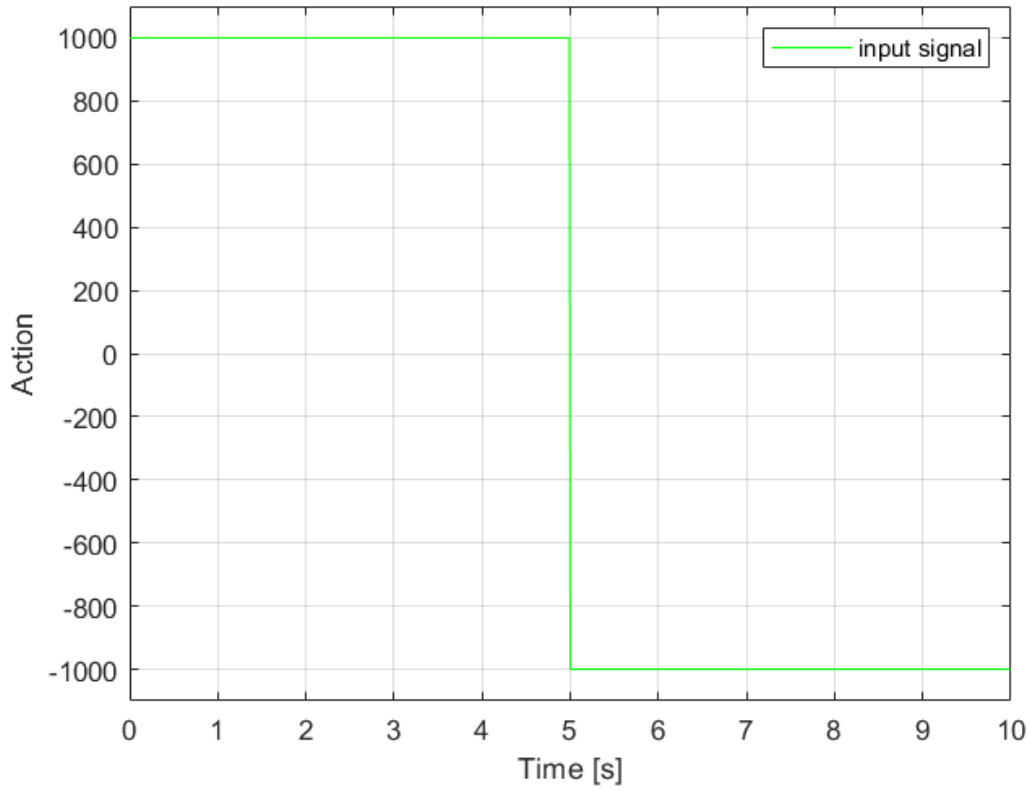
Funkcja *execute()* odpowiada za generowanie odpowiedzi obiektu na zadaną przez agenta akcje. Zwraca 3 wartości - obserwacje *outdata*, flagę skończenia procesu *terminal* i wartość nagrody *reward*. Tablica *outdata* została utworzona wcześniej w skrypcie i składa się z poniższych obserwacji:

- kąt wychylenia
- sinus kąta theta
- cosinus kąta theta
- prędkość kątową
- przyspieszenie kątowe
- uchyb kąta wychylenia

6.4 Ewaluacja modelu

Aby sprawdzić, czy utworzony przez nas model środowiska stanowi dobre odzwierciedlenie modelu w programie Matlab, a tym samym rzeczywistego obiektu, został on poddany testom. Do obu modeli został wprowadzony ten sam sygnał testowy, będący sygnałem prostokątnym. Wartość sygnału jest równa wartości akcji, dostępnej do wyboru przez algorytm uczący. Sygnał sprawdzi działanie modeli dla ujemnego i dodatniego zakresu prędkości. Sygnał prezentowany jest na osobnym wykresie ze względu na spory rozmiar osi y, który uniemożliwiłbym odczyt sygnałów wyjściowych.

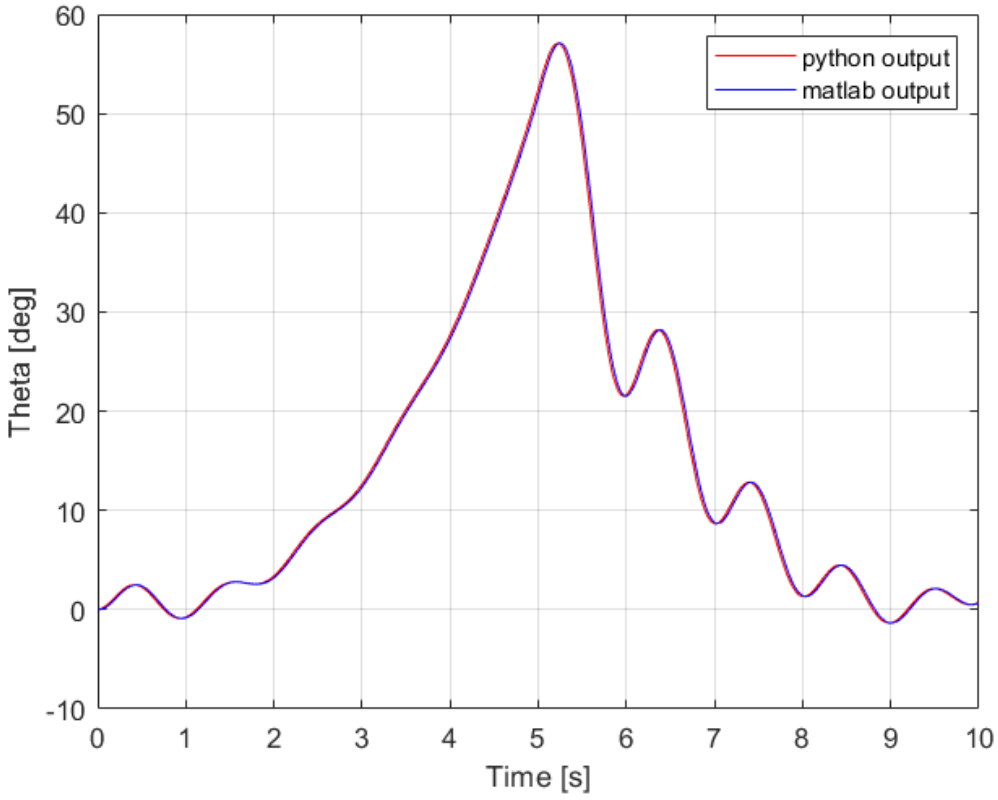
6.4.1 Sygnał testowy



Rysunek 6.1. Testowy sygnał prostokątny o dwóch wartościach skoku, źródło: Opracowanie własne

6.4.2 Odpowiedz układów

Sygnał testowy w postaci sygnału skokowego o dwóch wartościach 1000 i - 1000 został wyeksportowany do formatu .csv, po czym odczytany i użyty w skrypcie w ten sam sposób w jaki został użyty w programie Matlab w celu porównania środowisk. Odpowiedz modelu python została przekazana ponownie do Matlaba, skutkując poniższym wykresem.



Rysunek 6.2. Porównanie odpowiedzi układów, źródło: Opracowanie własne

Jak możemy zaobserwować, odpowiedzi obu układów są bardzo zbliżone, potwierdzając poprawność zaimplementowanego środowiska w programie. Aby dokładniej przedstawić różnicę między układami, został obliczony pierwiastek błędu średnio kwadratowego RMSE (od ang. Root-mean-square error).

$$RMSE(\hat{\theta}) = \sqrt{MSE(\hat{\theta})} = \sqrt{E(\hat{\theta} - \theta)^2} = 0.5377 \tag{6.5}$$

Otrzymana średnia wartość 0.5^o różnicy jest w pełni zadowalająca i nie powinna wpłynąć znacząco na dalsze wyniki prac.

6.5 Skrypt uczenia ze wzmacnianiem

6.5.1 Skrypt z tworzeniem nowego agenta

Drugą częścią implementacji w języku Python było stworzenie skryptu do nauczania ze wzmacnianiem. Wynikiem końcowym działania programu jest nauczony agent,który po odpowiedniej implementacji na mikroprocesorze, jest zdolny do sterowania realnym obiektem Aeropendulum.

listing 6.7. Struktura sieci i parametry agenta

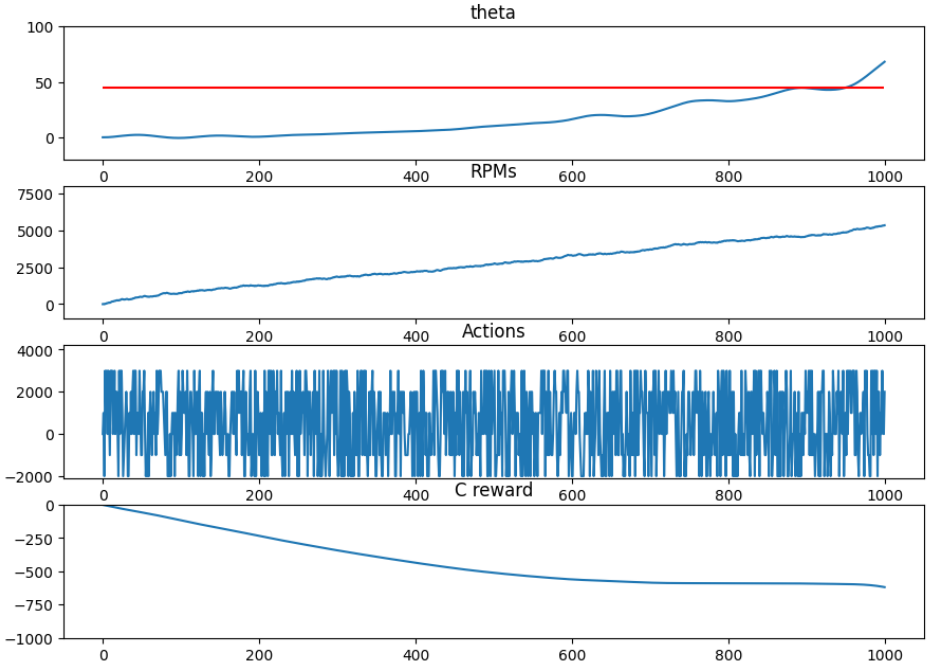
```
network_spec = [
    dict(type = 'dense', size=6),
    dict(type='dense', size=3),
    dict(type='dense', size=2),
]
baseline_spec = [
    dict(type = 'dense', size=6),
    dict(type='dense', size=4),
]

environment = Environment.create(
    environment=AEnv,
    max_episode_timesteps=1000)

agent = Agent.create(
    agent='vpg',
    environment=environment,
    batch_size=64,
    learning_rate=5e-3,
    discount=0.9,
    entropy_regularization=0.4,
    network=network_spec,
    baseline=baseline_spec,
    baseline_optimizer=dict(
        optimizer='adam', learning_rate=1e-3)
)
```

Ze względu na dobre wyniki części symulacyjnej przeprowadzonej w środowisku Matlab struktura agenta dla algorytmu Policy Gradient została dokładnie odwzorowana na podstawie architektury sieci oraz ustalonych parametrów. Za pomocą funkcji *create()* zostaje stworzone środowisko Aeropendulum.

listing 6.8. Skrypt uczenia agenta

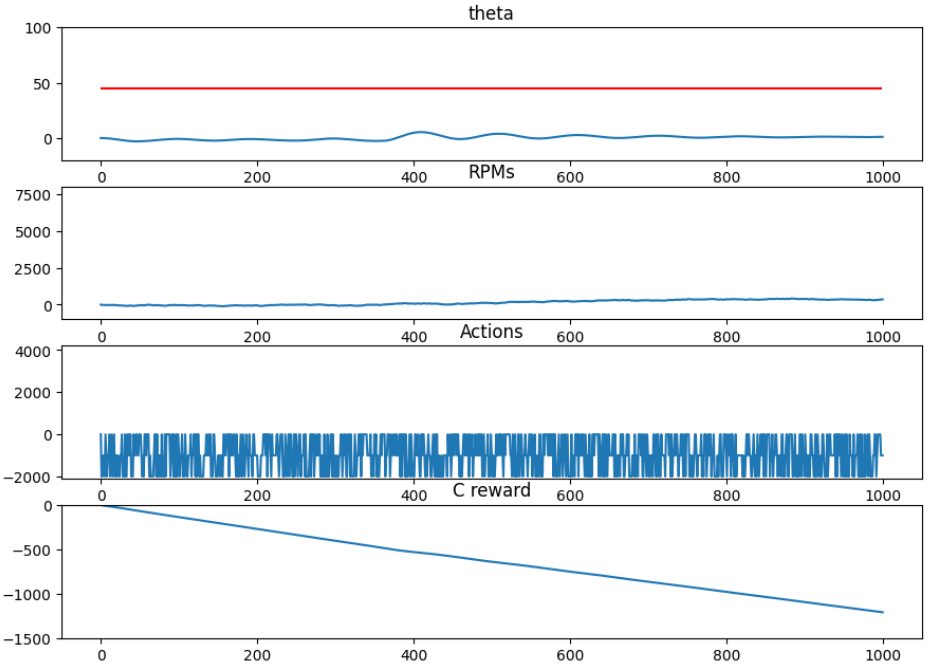
```
rg = 1500
temp2=[0.0]
for _ in range(rg):
    temp2=[0.0]
    mean_reward = 0.0
    rew_sum = 0.0
    states = environment.reset()
    print("States: ", states)
    print("Episodes: ", _+1, "/", rg)
    terminal = False
    while not terminal:
        actions = agent.act(states=states)
        states, terminal, reward = environment.execute(actions=
            actions)
        temp2 += [states[1]]
        rew_sum+=reward
        agent.observe(terminal=terminal, reward=reward)
    mean_reward = rew_sum/1000
```

Następnie przeprowadzony jest proces uczenia. Generujemy akcję agenta funkcją *act()*. Następnie odczytujemy obserwacje ze środowiska poddanego działaniu akcji z użyciem funkcji *execute()*. Dla zamieszczonego skryptu długość procesu uczenia to 1500 epizodów (zmienna rg).
W celu wygodnego odczytywania danych podczas procesu uczenia, został napisany kod generujący charakterystyki obserwacji dla czasu trwania jednego epizodu. Pomogło to w nadzorowaniu procesu uczenia oraz szybkie znalezienie ewentualnych nieprawidłowości.



Rysunek 6.3. Charakterystyki wybranych obserwacji, źródło: Opracowanie własne

Otrzymane wyniki były bardzo niezadowalające. Mimo poprawnie zdefiniowanego środowiska i tego samego algorytmu uczenia i parametrów, efekt procesu nauczania był znikomy. Brak efektów było obecny nawet podczas procesu trenowania zwiększonego do 8000 epizodów. Zostały podjęte próby modyfikacji modelu agenta, podobne do tych zastosowanych w części symulacyjnej. Zmiana liczby akcji lub zmiana ich zakresu nie prowadziła do osiągnięcia lepszych wyników. Na rysunku [Rysunek 6.4] prezentujemy odpowiedź układu dla 3 akcji w zakresie <-2000,2000>.



Rysunek 6.4. Charakterystyki wybranych obserwacji, źródło: Opracowanie własne

Po dłuższym okresie niepowodzeń postanowiliśmy wstrzymać pracę na skryptem uczącym, ponieważ nie prowadził on do osiągnięcia finalnego celu, jaki było sterowanie rzeczywistym obiektem. Możliwym powodem nikłego poziomu uczenia może być różnica w sposobie napisania algorytmu Policy Gradient w bibliotece Tensorforce w stosunku do algorytmu programu Matlab.
Zamiast tworzenia od podstaw modelu agenta postanowiliśmy zaimportować do skryptu gotowy plik algorytmu, który został nauczony w programie Matlab. Następnie zostałby on zastosowany w skrypcie sprawdzającym poprawność symulacyjnej regulacji Aeropendulum, pomijając niepotrzebny już proces uczenia.

6.5.2 Eksportowanie sieci neuronowej agenta

W celu wyeksportowania sieci neuronowej agenta skorzystaliśmy z funkcji *exportONNXNetwork()* z pakietu Deep Learning Toolbox Converter. Zdecydowaliśmy się na format .onnx ze względu na to, że jest on szeroko wspierany przez wiele narzędzi i platform programistycznych.[2] Znacznie ułatwi to przyszłą pracę nad

modelem agenta, szczególnie dla osób, które nie będą korzystały z tego samego zestawu bibliotek.
W celu wyeksportowania pliku należy przygotować sieć neuronową agenta programu Matlab. Ze względu na zmianę podejścia do rozwiązania problemu, nie potrzebujemy już eksportować struktury krytyka, ponieważ była ona wykorzystywana tylko podczas procesu uczenia. Wartościami, jakie zostaną przekazane do pliku .onnx są wszystkie dane potrzebne do odtworzenia sieci neuronowej.

listing 6.9. Skrypt odczytujący model agenta w formacie .onnx

```
01. onnx_model = onnx.load("C:/Users/User/Desktop/TF/actor_network.
        onnx")
02. k_model = onnx_to_keras(onnx_model, ['state'])
03. k_model.summary()
```

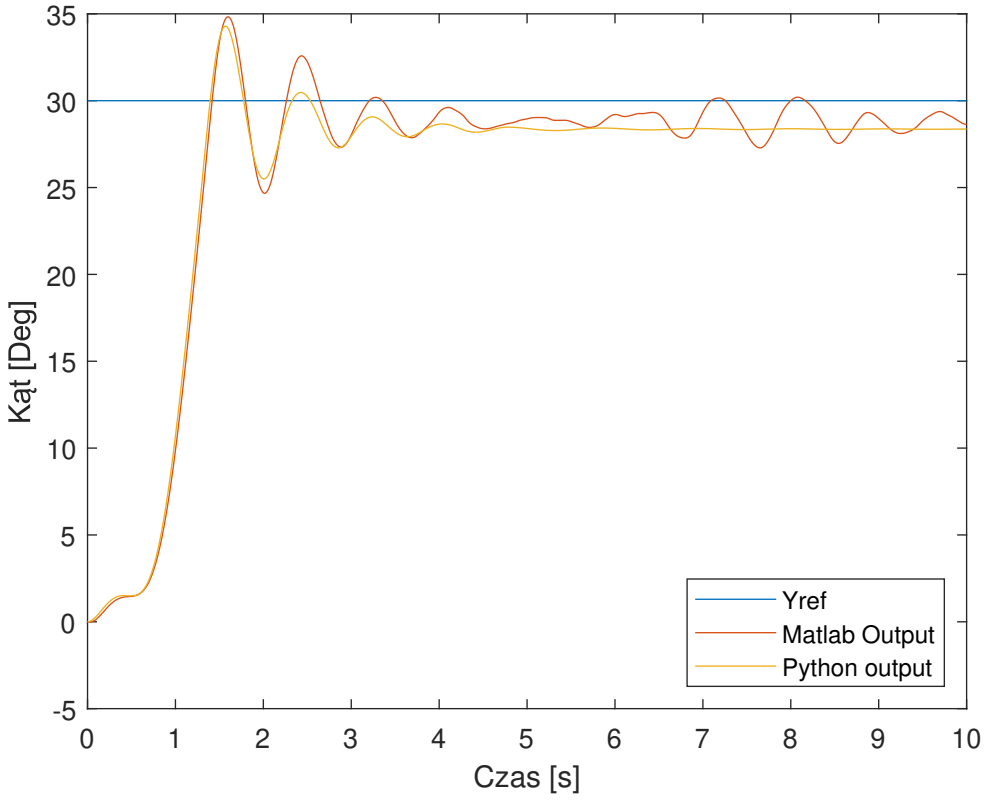
Następnie plik został wczytany do skryptu python i przekonwertowany na model keras przy użyciu funkcji *onnx_to_keras()*. Biblioteka Keras została zastosowana ze względu na rozległą dokumentację i wygodę obsługi. Otrzymaną strukturę sieci możemy zaprezentować za pomocą funkcji *summary()*.

Layer (type)	Output Shape	Param #	Connected to
state (InputLayer)	[(None, 6)]	0	
HL1_MatMul (Dense)	(None, 3)	18	state[0][0]
HL1_Add_const2 (Lambda)	(3,)	0	state[0][0]
HL1_Add (Lambda)	(None, 3)	0	HL1_MatMul[0][0] HL1_Add_const2[0][0]
HL2_MatMul (Dense)	(None, 2)	6	HL1_Add[0][0]
HL2_Add_const2 (Lambda)	(2,)	0	state[0][0]
HL2_Add (Lambda)	(None, 2)	0	HL2_MatMul[0][0] HL2_Add_const2[0][0]
action_MatMul (Dense)	(None, 7)	14	HL2_Add[0][0]
action_Add_const2 (Lambda)	(7,)	0	state[0][0]
action_Add (Lambda)	(None, 7)	0	action_MatMul[0][0] action_Add_const2[0][0]
RepresentationSoftMax (Lambda)	(None, 7)	0	action_Add[0][0]

Rysunek 6.5. Architektura sieci, źródło: Opracowanie własne

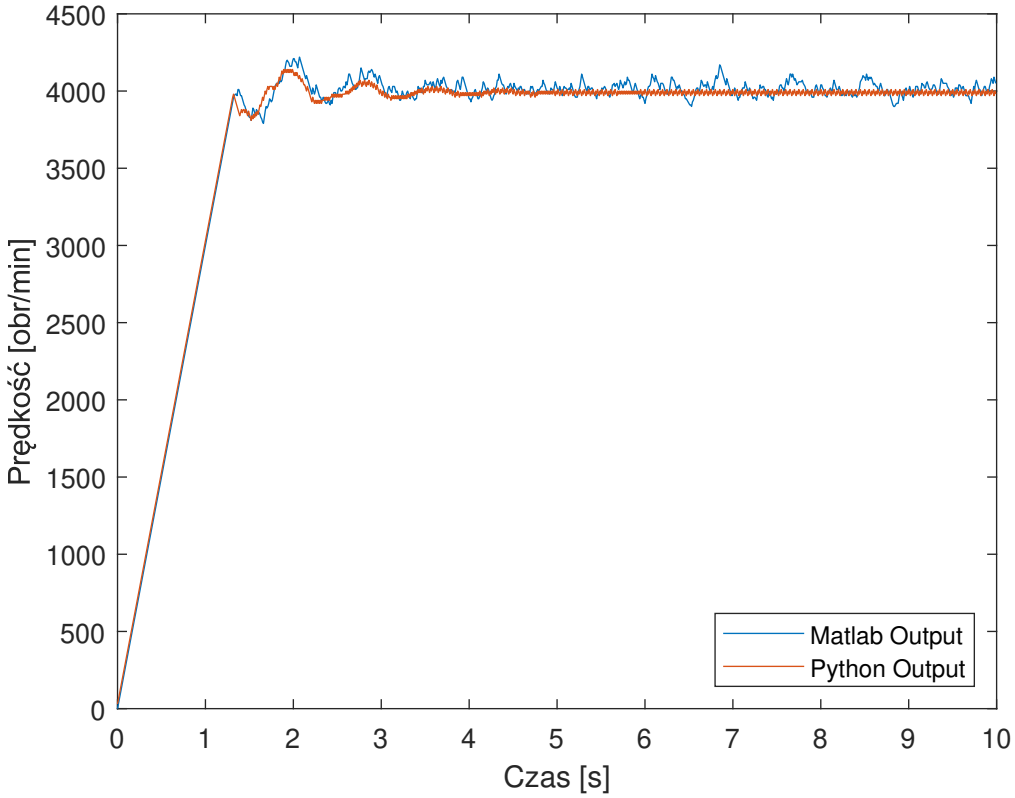
6.6 Wyniki testów skryptu Python

Sieć neuronowa agenta została przetestowana pod względem generowania poprawnych akcji z użyciem funkcji *predict()* z biblioteki Keras. W celu potwierdzenia zgodności dane z obu środowisk (Python i Matlab) zostały porównane na dwóch charakterystykach. [Rysunek 6.6 i Rysunek 6.7]



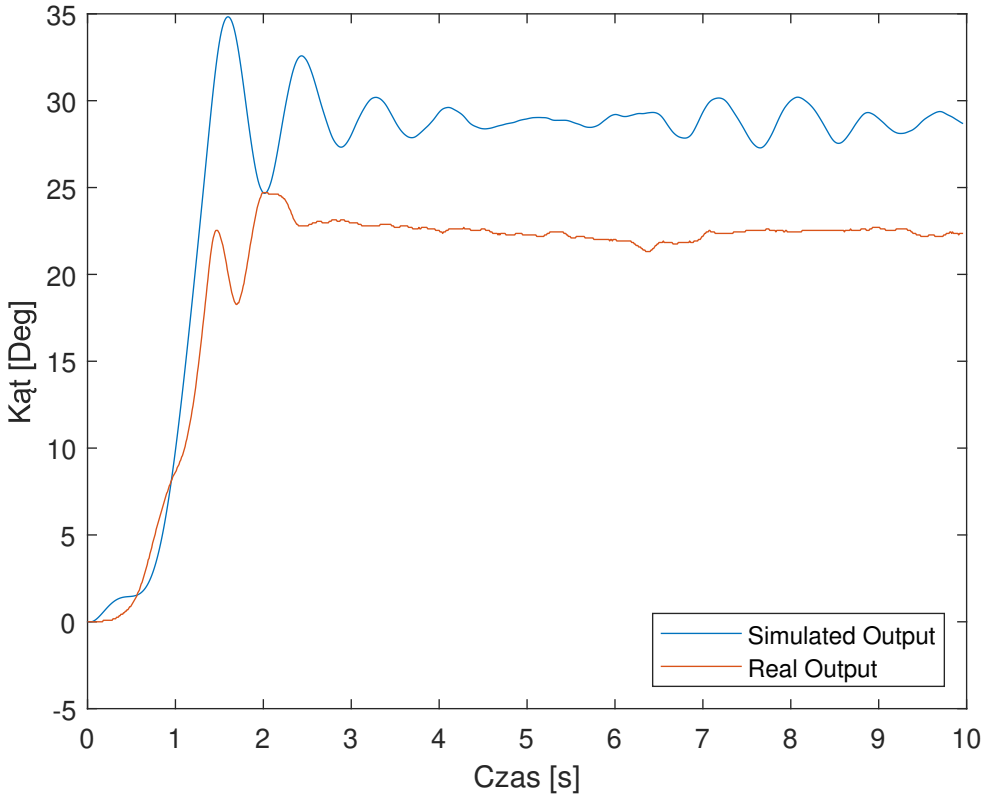
Rysunek 6.6. Charakterystyki kąta wychylenia wahadła, źródło: Opracowanie własne

Na powyższej charakterystyce możemy porównać jakość regulacji między modelami zaimplementowanymi w różnych środowiskach programowania (Matlab i Python). W przypadku charakterystyki sygnału Python output widzimy znacznie zmniejszone oscylacje i przeregulowanie, skutkujące znacznie płynniejszym przebiegiem charakterystyki. Od połowy charakterystyki widzimy też stanowczo ustaloną wartość kąta wychylenia, bez jakichkolwiek oscylacji. Widoczny uchyb ustalony może być spowodowany różnicą między definicją środowisk Aeropendulum, obliczoną w podrozdziale 6.4.2.



Rysunek 6.7. Charakterystyki akcji zadawanych przez agentów, źródło: Opracowanie własne

[Rysunek 6.7] prowadzi do podobnych konkluzji co [Rysunek 6.6]. Widzimy znacząco zmniejszone oscylacje i szybszą zmianę akcji w porównaniu do charakterystyki Matlab Output. Przebieg sygnału Python Output ulega ustaleniu, a oscylacja jest znikoma.



Rysunek 6.8. Charakterystyka , źródło: Opracowanie własne

Ostatnią charakterystyką jest porównanie symulacyjnej charakterystyki kąta wychylenia oraz realnej charakterystyki kąta wychylenia odczytanej z fizycznego obiektu Aeropendulum. Mimo obiecujących wyników części symulacyjnej możemy zauważyć, że nauczona sieć agenta nie osiąga wartości kąta referencyjnego. Jakoś regulacji nadal pozostaje na dobry poziomie, jednak ogólny uchyb ustalony jest dość duży. Mimo prób jak najwierniejszego odwzorowania wszystkich aspektów fizycznego środowiska, podczas realizacji kolejnych kroków nie da się uniknąć minimalnych przybliżeń, które nawarstwione mogą powodować znaczne odchylenie dla pracy na realnym Aeropendulum.

Rozdział 7

Podsumowanie pracy

Autor: Mikołaj Maciejewski, Marcin Sypniewski, Michał Rojewski

Przeprowadzone symulacje oraz liczne testy pozwoliły zaimplementowanie uczenia ze wzmocnieniem w celu stworzenia regulatora neuronowego dla nieliniowego obiektu typu Aeropendulum. Część symulacyjna dostarczyła nam sporo danych i wiedzy na temat działania różnych algorytmów jak i sposobu definicji ich poszczególnych komponentów, takich jak funkcja nagrody, akcji i obserwacji. Praca na modelu Aeropendulum w programie Matlab pozwoliła na wnikliwe i bezpieczne zapoznanie się ze środowiskiem oraz zrozumienie zasady implementacji własnych środowisk do wykorzystania z algorytmem uczenia przez wzmacnianie.
Rozdział zajmujący się istotą komunikacji wykazał iż model uczony w wyidealizowanych symulacyjnych warunkach odstaje regulacją dla obiektu rzeczywistego z powodu różnic w okresie zadawania wartości obserwacji dla agenta, które wynikały z ograniczeń wybranych przez nas rozwiązań to jest: wolnego środowiska obliczeniowego Matlab i podstawowych bibliotek działających na niemodyfikowanym Raspberry Pi. Próba nauczenie algorytmu na sprawdzonych parametrach przy pomocy biblioteki *Tensorforce* pokazała, że istnieją różnice między działaniem algorytmu w zależności od środowiska w którym została przeprowadzona. Implementacja wyeksportowanej nauczonej sieci neuronowej w języku *Python* pozwoliła na jej sprawdzenie i wstępne przygotowanie przykładowego skryptu, który można wykorzystać jako podstawę do dalszych badań. Możliwe jest rozwinięcie projektu aby działał on bezpośrednio na stanowisku rzeczywistym. Jednym z potencjalnych rozwiązań jest implementacja sieci neuronowej na *Raspberry Pi* w języku C.

Spis rysunków

1.1 Diagram działania RL, źródło: www.mathworks.com 6
1.2 Schemat budowy sieci neuronowej, źródło: www.ichi.pro/pl/siec-neuronowa-klasyfikator-mnist-od-podstaw-przy-uzyciu-biblioteki-numpy-163534368243374 7
1.3 Przykład Procesu Decyzyjnego Markova, źródło: Opracowanie własne 8
1.4 Przebieg algorytmu PG, źródło: https://programmer.group/pg-algorithm-based-on-policy-gradient.html 10

2.1 Rozkład sił działających na Aeropendulum, źródło: [19] 13

3.1 Model symulacyjny, źródło: Opracowanie własne 15
3.2 Zewnętrzna implementacja Aeropendulum, źródło: Opracowanie własne 16
3.3 Wewnętrzna implementacja Aeropendulum, źródło: Opracowanie własne 16
3.4 Funkcja nagrody, źródło: Opracowanie własne 17
3.5 Implementacja obserwacji, źródło: Opracowanie własne 18
3.6 Implementacja sygnałów referencyjnych, źródło: Opracowanie własne 18
3.7 Przykładowy przebieg zadanej sekwencji wartości kąta od czasu, źródło: Opracowanie własne 19
3.8 Przykładowy przebieg zadanej pojedynczej wartości kąta od czasu, źródło: Opracowanie własne 19
3.9 Lokalne minimum dla algorytmu PPO, źródło: Opracowanie własne . 20
3.10 Regulacja algorytmem PPO dla $theta = 30°$ po jednej sesji nauczania, źródło: Opracowanie własne 21
3.11 Nieudana regulacja algorytmem DQN dla $theta = 30°$ po kilku sesjach nauczania, źródło: Opracowanie własne 22
3.12 Regulacja algorytmem PG dla $theta = 25°$ po pierwszej sesji nauczania, źródło: Opracowanie własne 23
3.13 Regulacja algorytmem PG dla $theta = 50°$ po kilku sesjach nauczania, źródło: Opracowanie własne 23
3.14 Regulacja algorytmem PG dla $theta = -60°$ po kilku sesjach nauczania, źródło: Opracowanie własne 24

4.1 Sieć neuronowa krytyka, źródło: Opracowanie własne 28
4.2 Sieć neuronowa aktora, źródło: Opracowanie własne 28

5.1 Komputer jednopłytkowy Raspberry Pi, źródło: https://pl.farnell.com/raspberry-pi/rpi4-modbp-2gb/raspberry-pi-4-model-b-2gb/dp/3051886 29
5.2 Sterownik Silnika STEVAL-SPIN3201, źródło: https://pl.farnell.com/stmicroelectronics/steval-spin3201/eval-board-bldc-controller/dp/2761527 30

5.3 Konstrukcja ramki protokołu MCP, źródło: Opracowanie własne . . . 30
5.4 Enkoder magnetyczny AS5600, źródło: https://kamami.pl/moduly-peryferyjne-grove-seeed-studio/580651-modul-enkodera-magnetycznego-z-ukladem-as5600-ze-zlaczem-grove-101020692.html . . . 31
5.5 Przykładowy przebieg komunikacji I^2C, źródło: feriar-lab.pl/kurs-arduino-20-i2c/ . . . 31
5.6 Zasada działania protokołu UDP, źródło: Opracowanie własne . . . 32
5.7 Zasada działania protokołu TCP, źródło: Opracowanie własne . . . 33
5.8 Zasada Three-way Handshake, źródło: Opracowanie własne . . . 34
5.9 Założenia pracy układu przy protokole UDP źródło: Opracowanie własne 35
5.10 Wyniki testu dla Ethernet/UDP . . . 35
5.11 Wyniki testu dla Ethernet/Wi-Fi . . . 36
5.12 Przykładowy widok z oscyloskopu przy teście Ethernet źródło: [Opracowanie własne] . . . 37
5.13 Przykładowy widok z oscyloskopu przy teście Wifi źródło: Opracowanie własne . . . 38
5.14 Plan sterowania układu, źródło: Opracowanie własne . . . 39
5.15 Wyniki testu dla Wi-Fi/TCP . . . 40
5.16 Przykładowy widok z oscyloskopu przy teście Wi-Fi źródło: Opracowanie własne . . . 40
5.17 Zmiana kąta obrotu dla najlepszego Agenta symulacyjnego z pomocą Wi-Fi. źródło: Opracowanie własne . . . 47
5.18 Zmiana kąta obrotu dla wektora prędkości najlepszego Agenta źródło: Opracowanie własne . . . 48
5.19 Zmiana kąta obrotu dla Agenta z wektora prędkości dla stworzonego agenta o niższym próbkowaniu. źródło: Opracowanie własne . . . 49
5.20 Zmiana kąta obrotu z pomocą Wi-Fi z dedykowanym agentem. źródło: Opracowanie własne . . . 50

6.1 Testowy sygnał prostokątny o dwóch wartościach skoku, źródło: Opracowanie własne . . . 58
6.2 Porównanie odpowiedzi układów, źródło: Opracowanie własne . . . 59
6.3 Charakterystyki wybranych obserwacji, źródło: Opracowanie własne . 61
6.4 Charakterystyki wybranych obserwacji, źródło: Opracowanie własne . 62
6.5 Architektura sieci, źródło: Opracowanie własne . . . 63
6.6 Charakterystyki kąta wychylenia wahadła, źródło: Opracowanie własne 64
6.7 Charakterystyki akcji zadawanych przez agentów, źródło: Opracowanie własne . . . 65
6.8 Charakterystyka , źródło: Opracowanie własne . . . 66

Spis tablic

1.1 Przykładowa tabela aktualizacji polityki 11

5.1 Wartości z testów Ethernet 37
5.2 Wartości z testów Wi-Fi 38
5.3 Wartości z testów Wi-Fi 41

Spis listingów

4.1 Skrypt algorytmu PG agenta 27
5.1 Struktura adresu serwera 41
5.2 Funkcje send i recv() 42
5.3 Pobranie bitowej pozycji kąta 42
5.4 Konwersja do stopni 43
5.5 Funkcja normalizacji 43
5.6 Funkcja zadająca prędkość silnika 43
5.7 Funkcja przetwarzająca bajty na string 44
5.8 Operacje potrzebne do wygenerowania evaluatePolicy() 44
5.9 Funkcja pozwalająca przewidzieć akcję nauczonego agenta 44
5.10 Skrypt klienta UDP 45
5.11 Skrypt klienta TCP 45
5.12 Uaktualniony skrypt klienta TCP 46
6.1 Inicjalizacja klasy 52
6.2 Resetowanie środowiska 53
6.3 Funkcja obserwacji 54
6.4 Funkcja obserwacji 55
6.5 Funkcja nagrody 56
6.6 Obliczenie wartości kąta wychylenia i jego pochodnych 57
6.7 Struktura sieci i parametry agenta 60
6.8 Skrypt uczenia agenta 61
6.9 Skrypt odczytujący model agenta w formacie .onnx 63

Bibliografia

[1] *arpa/inet.h - definitions for internet operations*. URL: https://pubs.opengroup.org/onlinepubs/7908799/xns/arpainet.h.html (udostępniono 16. 01. 2022).

[2] Junjie Bai, Fang Lu, Ke Zhang i in. *ONNX: Open Neural Network Exchange*. https://github.com/onnx/onnx. 2019.

[3] Greg Brockman i in. *OpenAI Gym*. 2016.

[4] M. Coggan i D. Precup. *Exploration and Exploitation in Reinforcement Learning*. McGill University, 2004.

[5] *Core Functions*. URL: http://wiringpi.com/reference/core-functions/ (udostępniono 16. 01. 2022).

[6] IBM Cloud Education. *Sieci neuronowe*. URL: www.ibm.com/pl-pl/cloud/learn/neural-networks (udostępniono 17. 01. 2022).

[7] *I2C-bus specification and user manual*. 2021. URL: https://www.nxp.com/docs/en/user-guide/UM10204.pdf.

[8] $inet_pton(3)^-Linuxmanualpage$. URL: https://man7.org/linux/man-pages/man3/inet_pton.3.html (udostępniono 17. 01. 2022).

[9] Scott Jordan i in. „Evaluating the performance of reinforcement learning algorithms". W: *International Conference on Machine Learning*. PMLR. 2020, s. 4962–4973.

[10] Alexander Kuhnle, Michael Schaarschmidt i Kai Fricke. *Tensorforce: a TensorFlow library for applied reinforcement learning*. Web page. 2017. URL: https://github.com/tensorforce/tensorforce.

[11] *LAN Wiring Pinouts*. URL: http://www.zytrax.com/tech/layer_1/cables/tech_lan.html (udostępniono 16. 01. 2022).

[12] *listen(2) — Linux manual page*. URL: https://man7.org/linux/man-pages/man2/listen.2.html (udostępniono 25. 01. 2022).

[13] *Manifold - Exploration vs. Exploitation*. URL: https://www.manifold.ai/exploration-vs-exploitation-in-reinforcement-learning (udostępniono 09. 01. 2022).

[14] *Matlab Policy Gradient agents*. URL: https://www.mathworks.com/help/reinforcement-learning/ug/pg-agents.html (udostępniono 23. 01. 2022).

[15] *Matlab RL Agents*. URL: https://www.mathworks.com/help/reinforcement-learning/ug/create-agents-for-reinforcement-learning.html (udostępniono 09. 01. 2022).

[16] W. Mielczarek. *Szeregowe interfejsy cyfrowe*. Helion, 1993, s. 125.

[17] *On-policy vs. Off-policy*. URL: https://kowshikchilamkurthy.medium.com/off-policy-vs-on-policy-vs-offline-reinforcement-learning-demystified-f7f87e275b48 (udostępniono 09. 01. 2022).

[18] J. Walkowski P. Pilarski B. Podkański. *OPRACOWANIE I WYKONANIE STANOWISKA LABORATORYJNEGO Z TZW. WAHADŁEM LOTNICZYM*. 2021.

[19] J. Walkowski P. Pilarski B. Podkański. *Opracowanie i wykonanie stanowiska laboratoryjnego z tzw. wahadłem lotniczym (Aeropendulum)*. 2021.

[20] *perror(3) — Linux manual page*. URL: https://man7.org/linux/man-pages/man3/perror.3.html (udostępniono 17. 01. 2022).

[21] Jon Postel. *Transmission Control Protocol*. 1981.

[22] M.Szaleniec R.Tadeusiewicz. *Leksykon Sieci Neuronowych[Lexicon on Neural Networks]*. Projekt Nauka, 2015.

[23] *recvfrom(3p) — Linux manual page*. URL: https://man7.org/linux/man-pages/man3/recvfrom.3p.html (udostępniono 17. 01. 2022).

[24] *Reinforcement Learning algorithms - an intuitive overview*. URL: https://smartlabai.medium.com/reinforcement-learning-algorithms-an-intuitive-overview-904e2dff5bbc (udostępniono 09. 01. 2022).

[25] $Seeed_ArduinoAS5600$. URL: https://github.com/Seeed-Studio/Seeed_Arduino_AS5600 (udostępniono 17. 01. 2022).

[26] *socket(2) — Linux manual page*. URL: https://man7.org/linux/man-pages/man2/socket.2.html (udostępniono 17. 01. 2022).

[27] R. S. Sutton i A. G. Barto. *Reinforcement Learning an introduction*. The MIT Press, 2020.

[28] *sys/socket.h - Internet Protocol family*. URL: https://pubs.opengroup.org/onlinepubs/7908799/xns/syssocket.h.html (udostępniono 16. 01. 2022).

[29] Martijn Van Otterlo i Marco Wiering. „Reinforcement learning and markov decision processes". W: *Reinforcement learning*. Springer, 2012, s. 3–42.