Wydział Elektroniki i Technik Informacyjnych
Politechnika Warszawska

SZAU

Sprawozdanie z projektu drugiego

Mikołaj Sendybył, Wojciech Pobocha

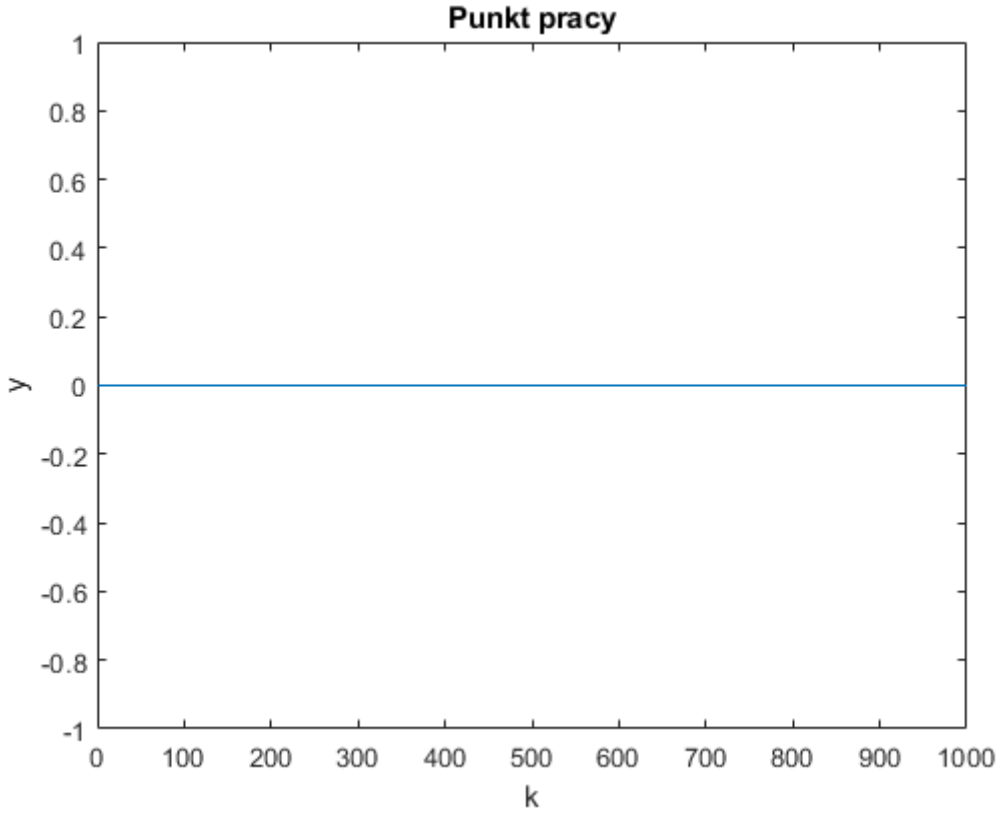
Warszawa, 2024

Spis treści

1. Symulacja procesu

1.1. Sprawdzenie punktu pracy obiektu

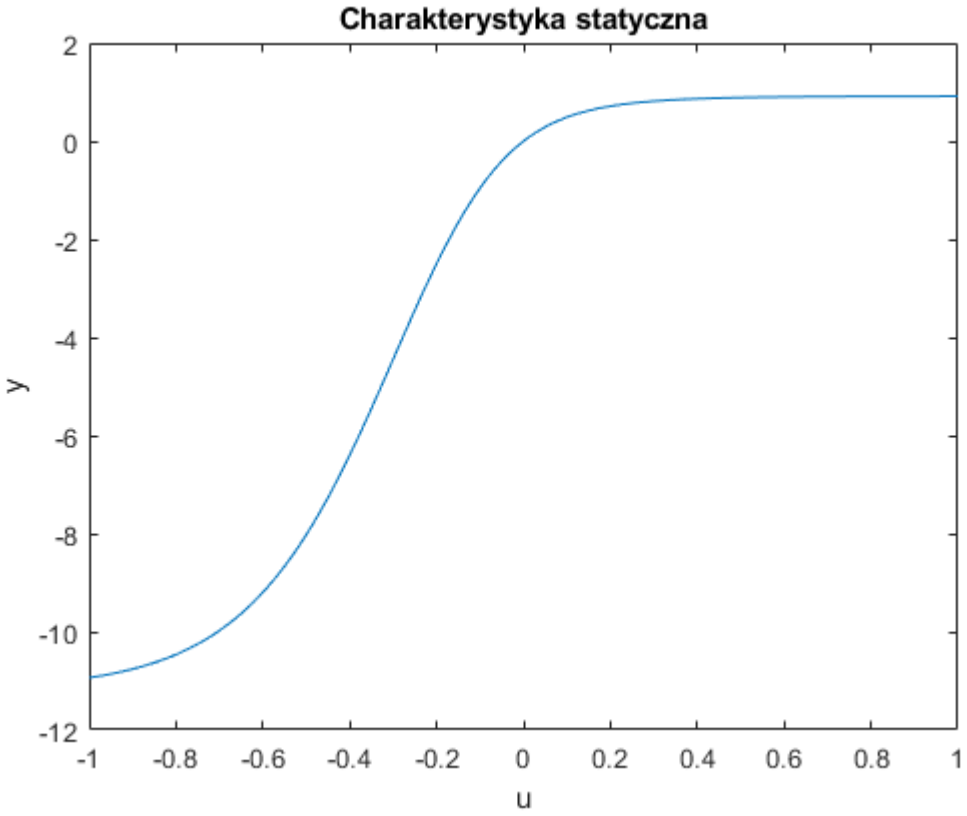
Przed wyznaczeniem charakterystyki statycznej sprawdzono poprawność podanego punktu pracy $y = u = x_1 = x_2 = 0$. Wyniki przedstawiono na wykresie 1.1 - punkt pracy się zgadza.



Rys. 1.1. Sprawdzenie podanego punktu pracy

1.2. Charakterystyka statyczna

Charakterystykę statyczną wyznaczono metodą eksperymentalną - zasymulowano obiekt dla $u \in [-1, 1]$, z dokładnością 0.01. Dla każdego skoku odczekano 89 próbek czasu i doczytano końcową wartość wyjścia obiektu. Ze zgromadzonych w ten sposób pomiarów powstał wykres 1.2. Wynika z niego jednoznacznie, że obiekt jest mocno nieliniowy - krzywa charakterystyki składa się w dużym przybliżeniu z 5-6 prostych.



Rys. 1.2. Charakterystyka statyczna obiektu

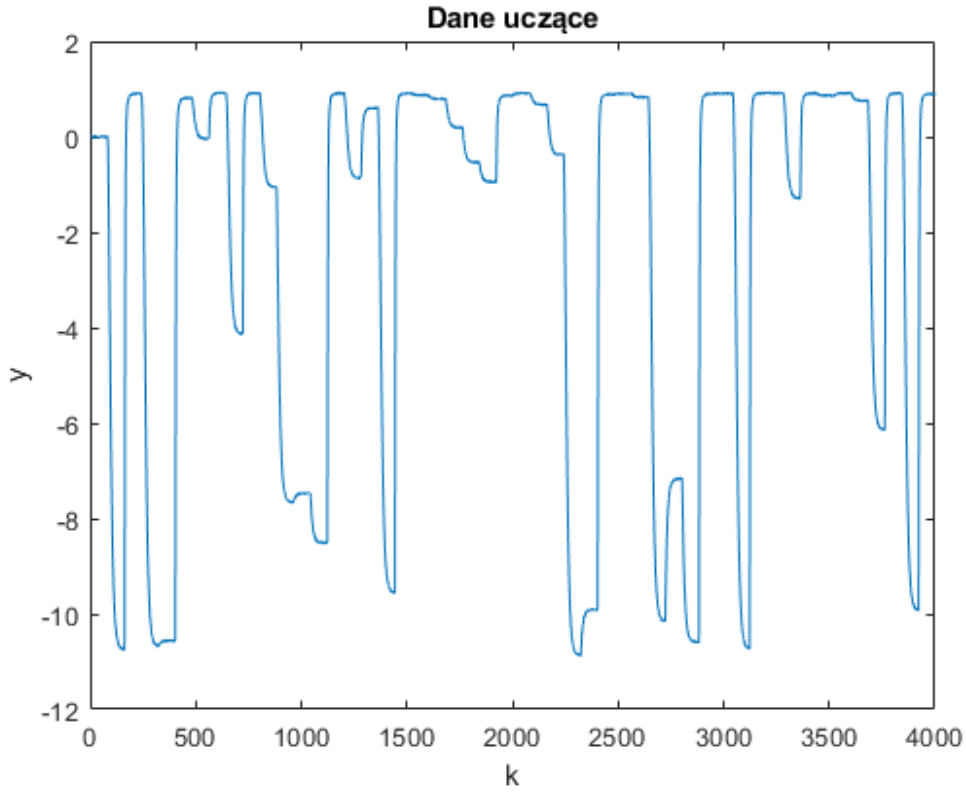
1.3. Wyznaczanie zbioru uczącego i weryfikującego

Do wyznaczenia zbioru danych weryfikujących i uczących posłużono się poniższym kodem:

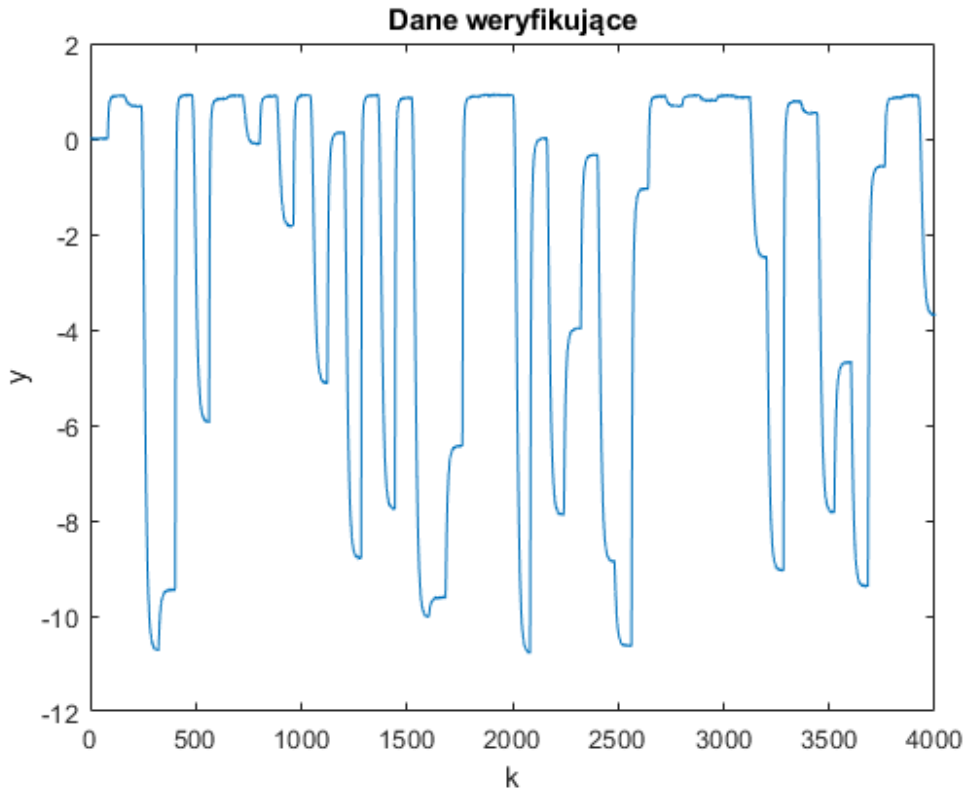
```
if zadanie == 3

    steps = 4000;
    x1 = zeros(1,steps);
    x2 = zeros(1,steps);
    y = zeros(1,steps);

    %% generowanie danych uczących i weryfikujących

    dane_wer = false; % wybór typu danych do generowania
    szum_pomiarowy = true;

    if dane_wer
        rand('seed', 23);
    else
        rand('seed', 43);
    end

    u = 2 * rand(1, 100) - 1;
    U(1:80) = 0;
    i = 1;

    % Symulacja obiektu dla losowych skoków sterowań
    for k=10:steps
        if mod(k, 80) == 0 && k ~= 4000
            U(k+1:k+80) = u(i);
            i = i+1;
        end

        x1(k) = -alfa1 * x1(k-1) + x2(k-1) + beta1 * g1(U(k-3));
        x2(k) = -alfa2 * x1(k-1) + beta2 * g1(U(k-3));
        if szum_pomiarowy
            y(k) = g2(x1(k)) + (0.03 * (2 * rand(1,1) - 0.5));
        else
            y(k) = g2(x1(k));
        end
    end
end
```

Zbiory uczący i weryfikujący mają wielkość 4000 próbek, skoki wartości sterowania następują co 80 próbek. Przyjęto stałą wartośc *seed* w funkcji rand, dla każdego zbioru, by zawsze pracować na tych samych danych i przedstawić miarodajne wyniki eksperymentów w kolejnym zadaniu. W obydwu zbiorach dodano szum pomiarowy.

Dane uczące przedstawiono na wykresie 1.3, dane weryfikujące na wykresie 1.4.



Rys. 1.3. Dane uczące

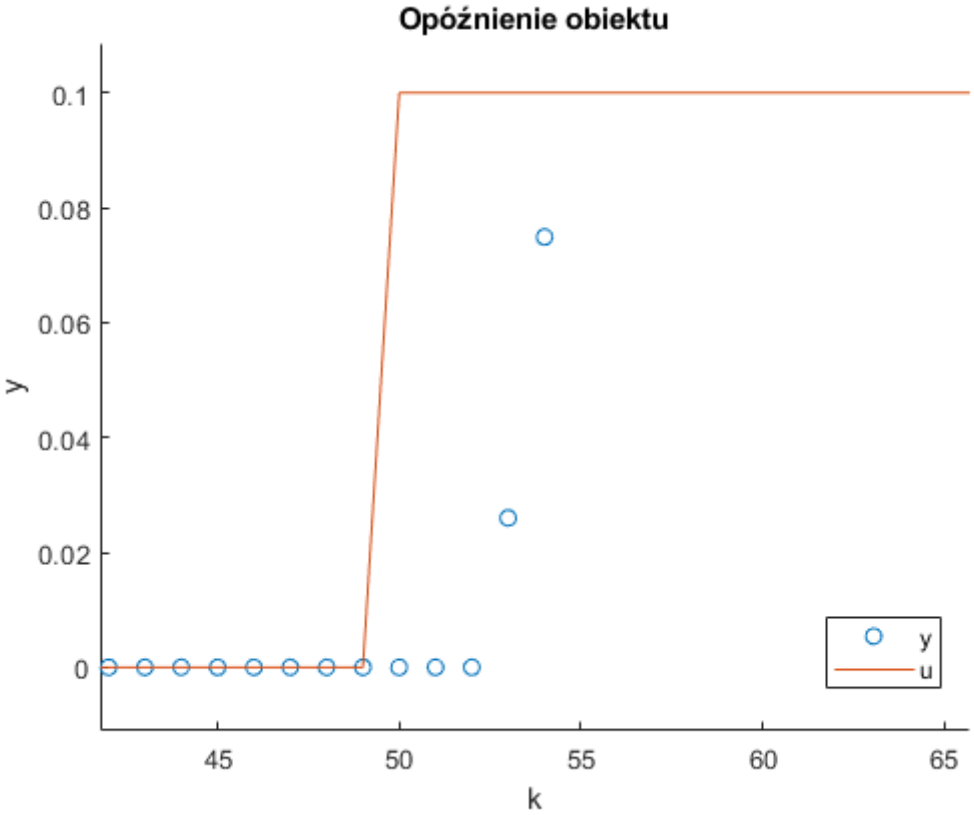


Rys. 1.4. Dane weryfikujące

2. Modelowanie procesu

2.1. Określenie opóźnienia obiektu

W celu określenia opóźnienia τ wykonano skok sterowania u w chwili $k = 50$ i odczytano je z wykresu (2.1). Zmiana wyjścia nastąpiła po 3 chwilach dyskretnych k, zatem opóźnienie jest równe $\tau = 3$.



Rys. 2.1. Określenie opóźnienia obiektu

2.2. Uczenie modeli neuronowych przy pomocy programu sieci.exe - wybór najlepszej sieci

Dla wszystkich modeli przyjęto następujące parametry programu:

— Maksymalna liczba iteracji uczących = 600
— Błąd graniczny = $1e-4$
— $n_b = 4$
— $n_a = 2$
— $\tau = 3$

2.2.1. Predyktor OE, algorytm BFGS

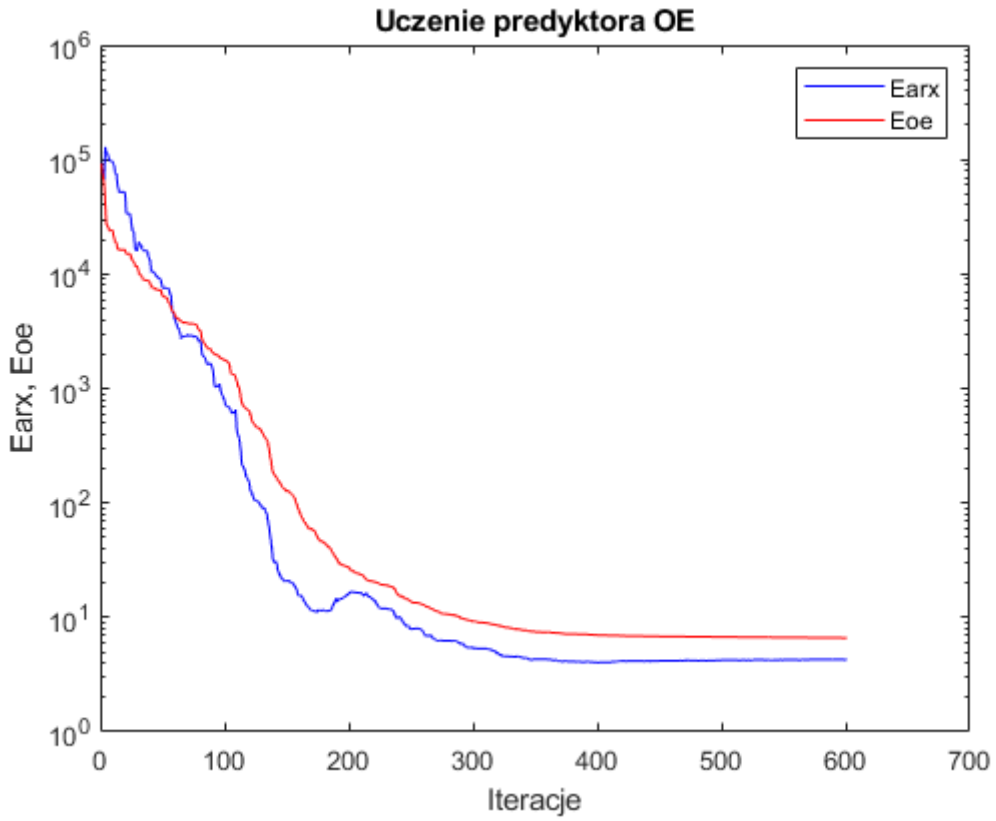
Zgodnie z poleceniem przeprowadzono ucznenie serii modeli neuronowych, z liczbą neuronów ukrytych od 1 do 10. W każdym przypadku uczenie powtórzono 5 lub więcej razy (do momentu uzyskania akceptowalnych wyników). Rezultaty przedstawiono w tabeli 2.1.

Tab. 2.1. Wpływ liczby neuronów warstwy ukrytej na błędy predyktora OE

Liczba neuronów	Błąd zbioru uczącego	Błąd zbioru weryfikującego
1	2595.5421	3047.2737
2	54.0877	80.9999
3	39.1487	64.4271
4	11.6548	58.9210
5	10.1948	89.3522
6	7.0524	19.8317
7	8.2511	126.2365
8	4.9914	60.1339
9	4.0843	113.1128
10	11.9094	516.4714

Wyżej przedstawione wyniki wymagają krótkiego skomentowania. Model z jednym neuronem po prostu nie działa - błędy rzędu $e+3$ są nieakceptowalne. Od modelu z dwoma neuronami obserwujemy poprawę jakości aproksymacji. Błąd stopniowo spada, do modelu z pięcioma neuronami, który mimo niskiego błędu dla zbioru uczącego oferuje niższą jakość niż sieci z mniejszą liczbą neuronów. Najlepszym uzyskanym modelem jest zdecydowanie ten sześcio neuronowy - zapewnia on najmniejszy błąd dla obydwu zbiorów i z tego powodu został wybrany do dalszej części projektu. Modele "6+ neuronowe" padły ofiarą zajwiska przeuczenia - mimo malejącego błędu zbioru uczącego , błąd weryfikacyjny rośnie. Ostatecznie wybrany model musi być ogólny tj. dobrze aproksymować wyjście obiektu dla różnych wejść, nie tylko tych ze zbioru uczącego, dlatego te modele zostały odrzucone. Warto na końcu zaznaczyć, że żaden z wyznaczonych modeli nie "przystosowywuje" się do dodanego szumu pomiarowego.

Na wykresie 2.2 przedstawiono błędy predyktora OE i ARX modelu z sześcioma neuronami w kolejnych iteracjach uczących

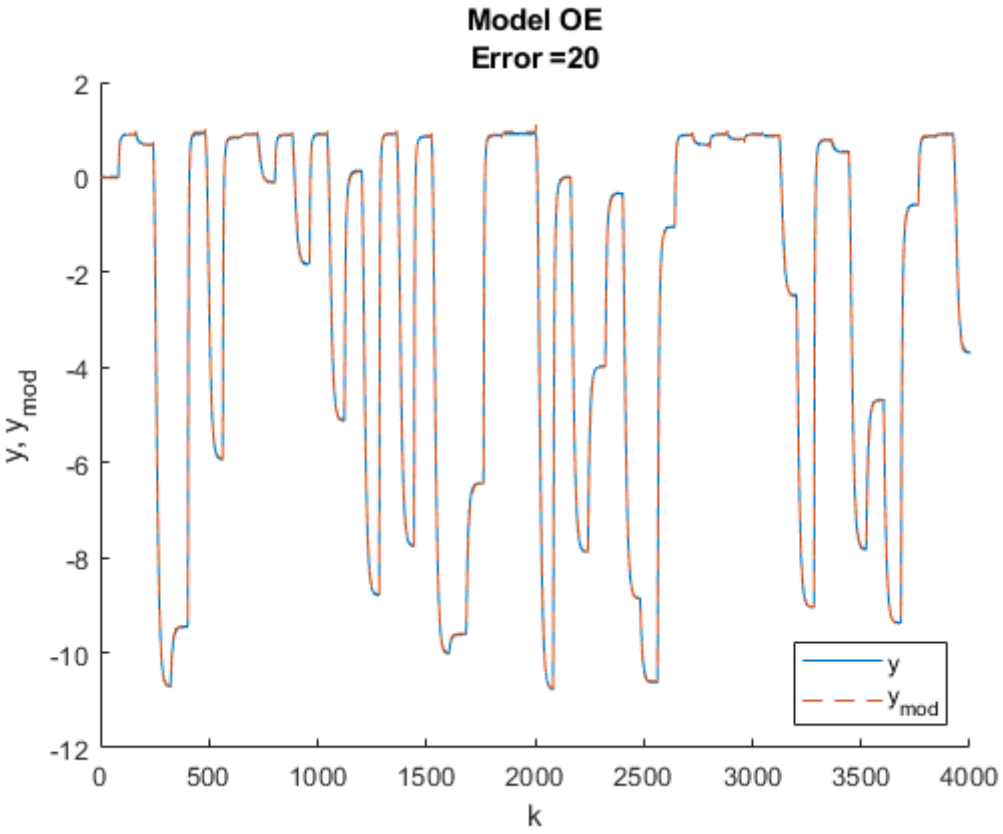


Rys. 2.2. Błędy predyktora OE i ARX modelu z sześcioma neuronami ukrytymi w kolejnych iteracjach uczących

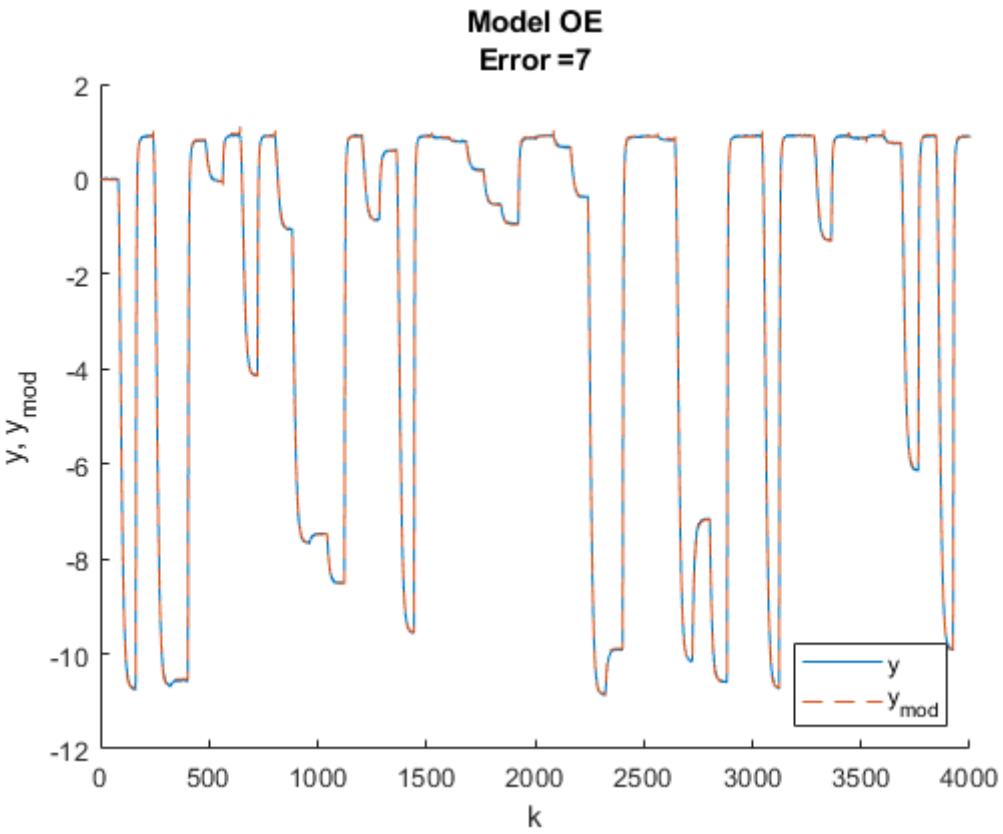
2.2.2. Symulacja wybranego modelu neuronowego dla zbioru uczącego i weryfikującego

Zgodnie z tym, co zostało napisane w punkcie 2.2.1, w dalszej części projektu wykorzystany będzie model sześcio neuronowy. Na wykresach 2.4 i 2.3 przedstawiono wyniki symulacji, a poniżej znajduje się kod wykorzystany do symulacji modelu w tym i kolejnych podpunktach.

```
clear all
load('./Najlepsze_wagi_OE_BFGS/N6.mat')

dane_wer = readmatrix('dane_wer.txt');
u_wer = dane_wer(:, 1);
y_wer = dane_wer(:, 2);

dane_ucz = readmatrix('dane_ucz.txt');
u_ucz = dane_ucz(:, 1);
y_ucz = dane_ucz(:, 2);

dane_ucz = false; % wybór danych
wykresy = true; % włączanie/wyłączanie wykresów
ARX = false;     % wybór typu modelu

if dane_ucz
    u = u_ucz;
    y = y_ucz;
else
    u = u_wer;
    y = y_wer;
end

steps = length(dane_wer);
y_mod = zeros(1, steps);
e = zeros(1, steps);

if ~ARX
    for k=10:steps
        q = [u(k - 3); u(k -4); y_mod(k-1); y_mod(k-2)];
        y_mod(k) = w20 + w2 * tanh(w10 + w1 * q);
        e(k) = y_mod(k) - y(k);
    end

else
    for k=10:steps
        q = [u(k - 3); u(k -4); y(k-1); y(k-2)];
        y_mod(k) = w20 + w2 * tanh(w10 + w1 * q);
        e(k) = y_mod(k) - y(k);
    end

end

Error = sum(e.^2);
disp(Error);
```



Rys. 2.3. Zbiór weryfikujący

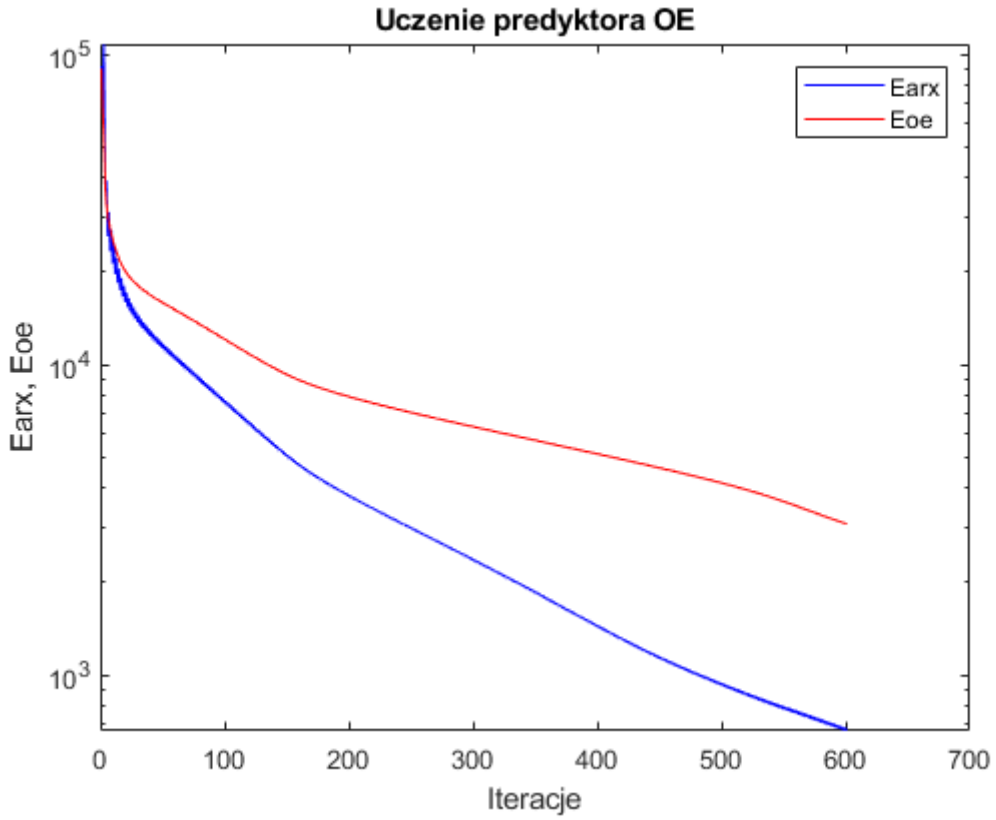


Rys. 2.4. Zbiór uczący

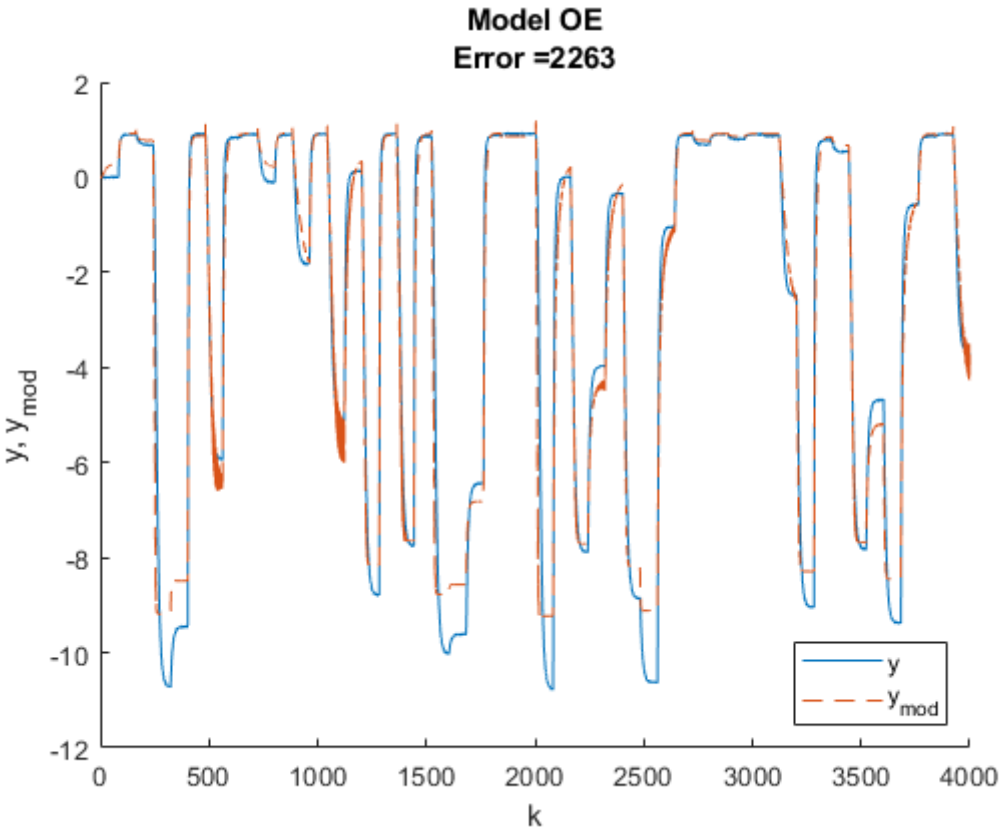
2.2.3. Predyktor OE, algorytm najszybszego spadku

Przeprowadzono 10 prób uczenia sieci algorytmem naszybszego spadku, niestety rezultaty są niezadawalające - najmniejsze błędy jakie udało się uzyskać to 3068 dla zbioru uczącego i 2263 dla zbioru weryfikującego. Wyniki symulacji dla najlepszych uzyskanych wag przedstawiono na wykresach 2.7 i 2.6. Widać na nich, że model jest zły - pożądany kształt sygnału jest słabo odwzorowany i zdarza się, że model wpada w oscylacje dla konkretnych wartości wejściowych.

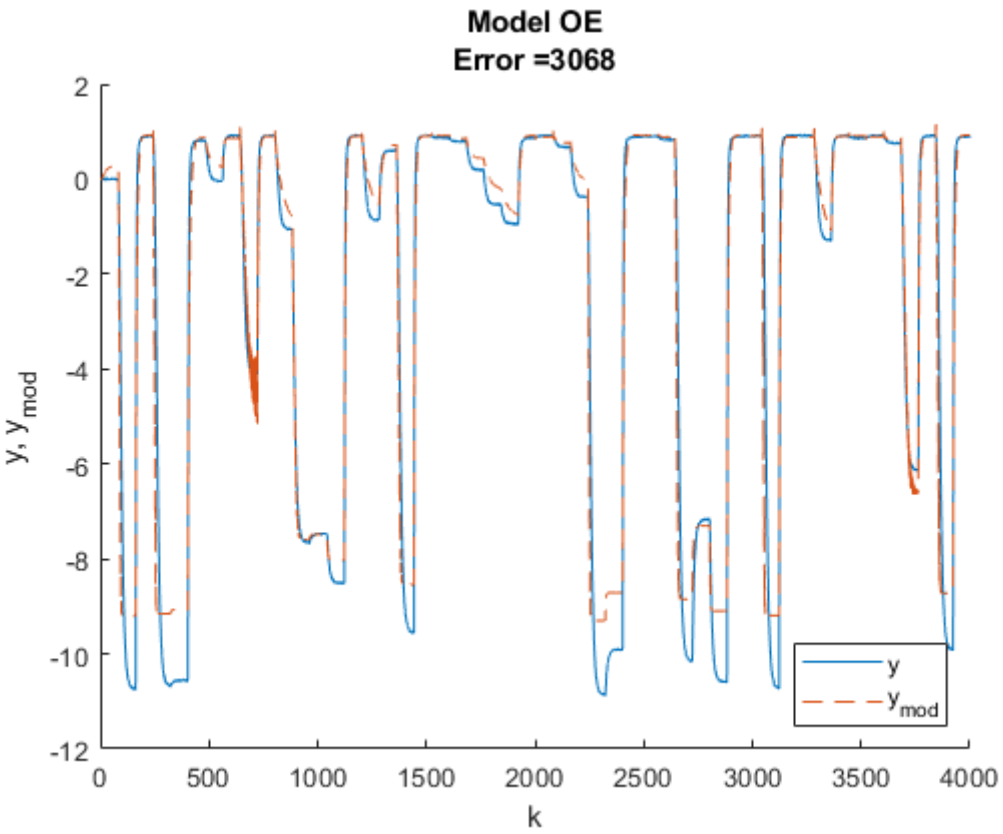
Tak słaba skuteczność algorytmu najszybszego spadku jest spowodowana jego 'naturą' - w podstawowej wersj, ma przypadłość utykania w najszybciej znalezionym minimum i nie ma mechanizmów, które pozwyoliłyby mu wyrwać się z tego minimum loklanego. Autorzy sprawozdania nie mają wiedzy jaka dokładnie wersja algorytmu najszybszego spadku została zaimplementowana w programie sieci.exe, ale przedstawiona wyżej teza wydaje się być dość prawdopodobna.



Rys. 2.5. Błędy predyktora OE i ARX modelu z sześcioma neuronami ukrytymi w kolejnych iteracjach uczących



Rys. 2.6. Zbiór weryfikujący

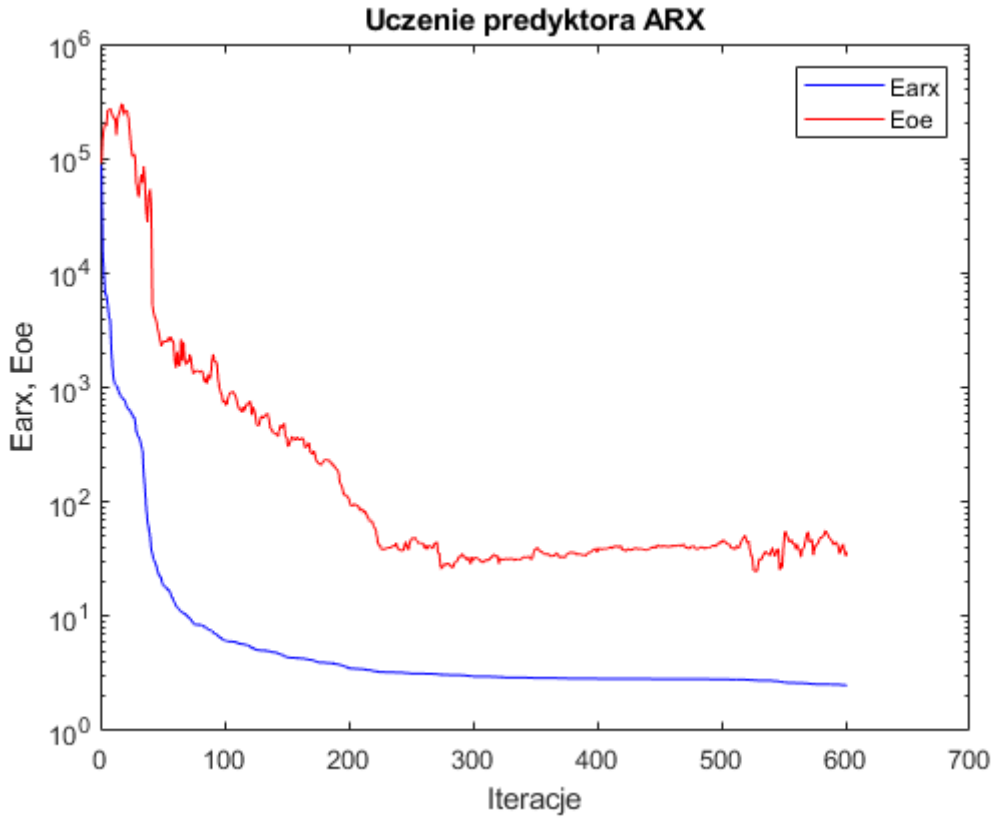


Rys. 2.7. Zbiór uczący

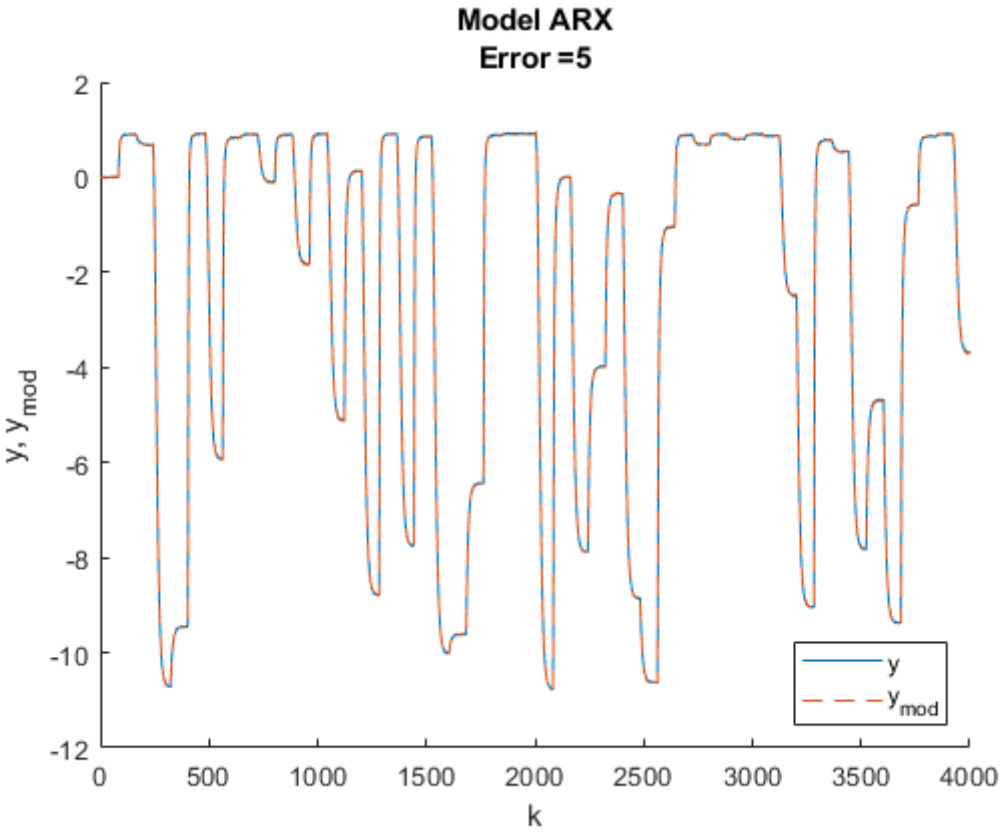
2.2.4. Predyktor ARX, alogorytm BGFS

Przeprowadzono 5 prób uczenia sieci, w trybie ARX, algorytmem BFGS. Wyniki są bardzo dobre - uzyskano niskie błędy zarówno dla zbioru weryfikującego, jak i zbioru uczącego (dopowiednio 4.5709 i 2.5010). Wyniki symulacji dla najlepszych uzyskanych wag przedstawiono na wykresach 2.9 i 2.10, na wykresie 2.8 znajdują się błędy z uczenia predyktora ARX.

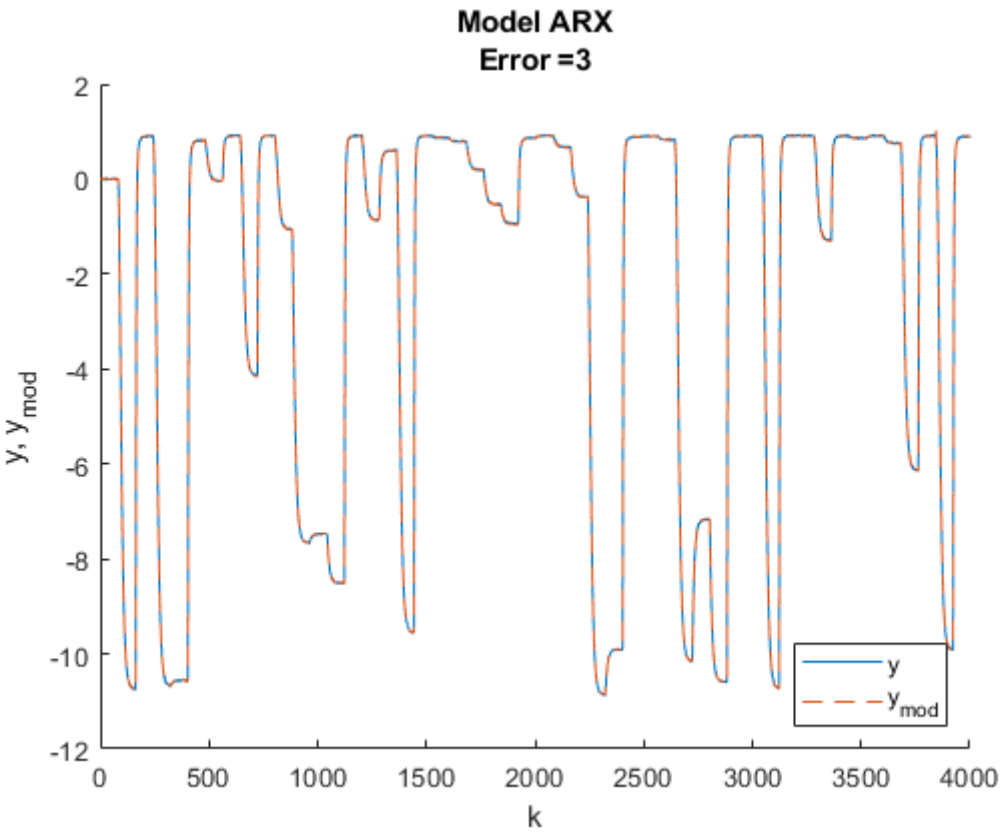
Uzyskany model jest najlepszym uzyskanym do tej pory pod względem wielkości błędu. Mogło by się wydawać, że z tego powodu jest również najlepszym modelem ogólnie, ale nie jest to niestety prawdą. Tak niskie błędy wzbudziły wątpliwości - pojawiło się podejrzenie, że sieć może dopasowywać się do szumu pomiarowego. Po sprawdzeniu okazało się, że faktycznie tak jest (Wykres 2.11). Dobry model absolutnie nie powinien się tak zachowywyać. Pokazuje to, że niższy błąd, nawet na obydwu zbiorach danych, nie zawsze oznacza lepszy model, niż ten z wyższymi błędami.



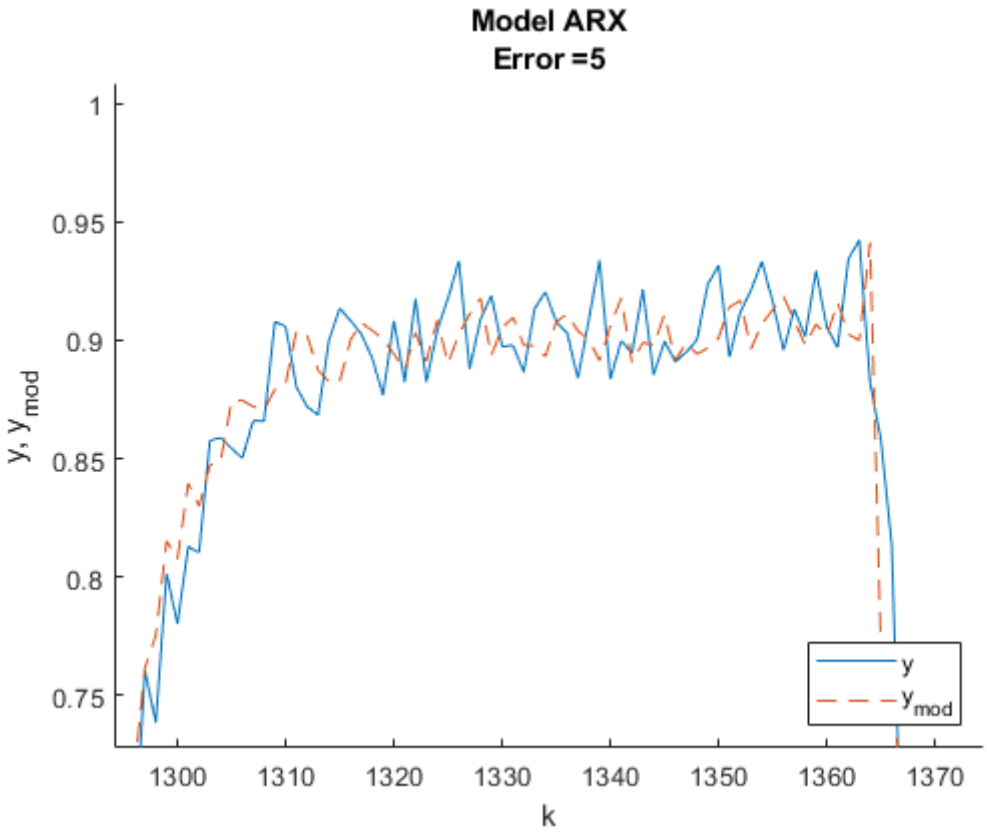
Rys. 2.8. Błędy predyktora OE i ARX modelu z sześcioma neuronami ukrytymi w kolejnych iteracjach uczących



Rys. 2.9. Zbiór weryfikujący



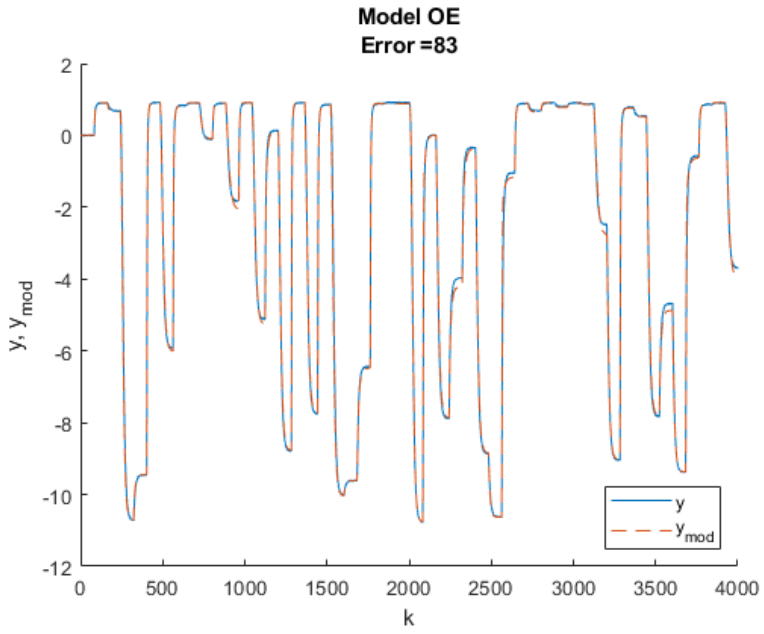
Rys. 2.10. Zbiór uczący



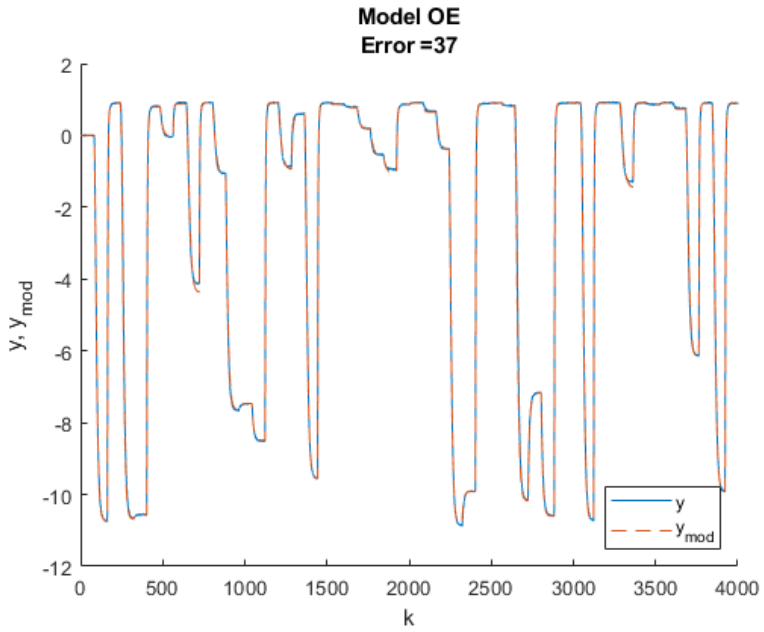
Rys. 2.11. Dopsowywanie się wyjścia modelu do szumu pomiarowego; zbiór weryfikujący

2.2.5. Symulacja modelu uczonego w trybie ARX jako predyktor OE

Na wykresach 2.12 i 2.13 przedstawiono wyniki przeprowadzonych eksperymentów. Oprócz całkiem dobrego predyktora ARX (nie licząc dopasowywania się do szumu pomiarowego), udało się również otrzymać przyzwoity predyktor OE - błędy są co prawda dość duże, ale biorąc pod uwagę fakt, że model był uczony w trybie ARX wyniki są zadawalające.



Rys. 2.12. Zbiór weryfikujący

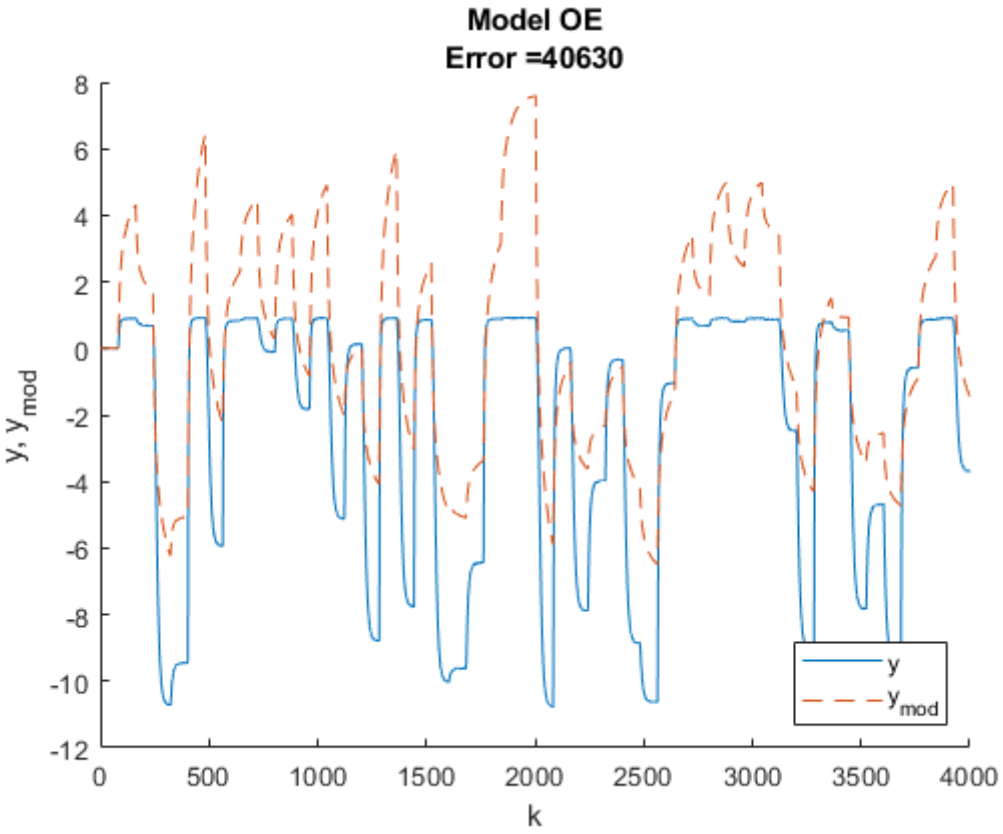


Rys. 2.13. Zbiór uczący

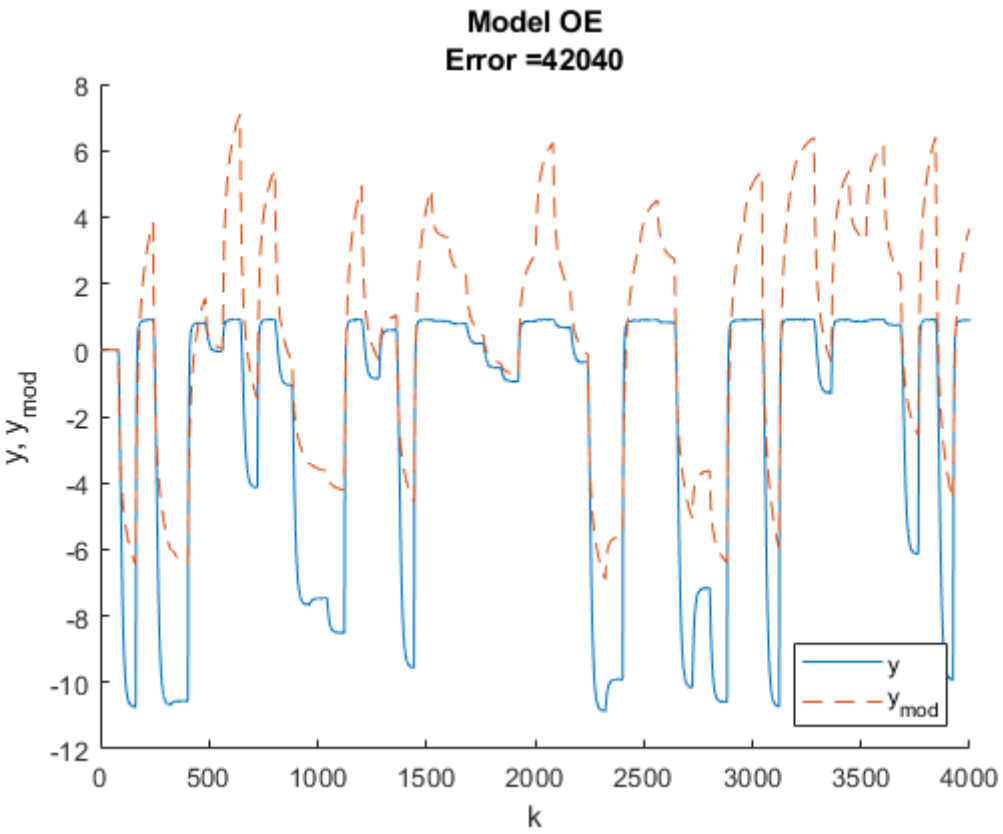
2.2.6. Metoda najmniejszych kwadratów

Metodą najmniejszych kwadratów wyznaczono model liniowy o dynamice drugiego rzędu. Następnie zasymulowano go w trybie rekurencyjnym dla zbioru danych uczących i weryfikujących. Na wykresach 2.15 i 2.14 przedstawiono wyniki eksperymentów, pomiżej znajduje się kod użyty do wygenerowania wag modelu i jego symulacji.

```
clear all
dane_wer = readmatrix('dane_wer.txt');
u_wer = dane_wer(:, 1);
y_wer = dane_wer(:, 2);
dane_ucz = readmatrix('dane_ucz.txt');
u_ucz = dane_ucz(:, 1);
y_ucz = dane_ucz(:, 2);

dane_ucz = false; % wybór danych
wykresy = true; % włączanie/wyłączanie wykresów
ARX = true;     % wybór typu modelu

if dane_ucz
    u = u_ucz;
    y = y_ucz;
else
    u = u_wer;
    y = y_wer;
end

%% MNK
steps = length(dane_wer);
Y = y_ucz;
M = zeros(steps, 4);
for i=10:steps
    M(i, :) = [u_ucz(i-3) u_ucz(i-4) y_ucz(i-1) y_ucz(i-2)];
end
w = M\Y;

y_mod = zeros(1, steps);
e = zeros(1, steps);

if ~ARX
    for k=10:steps
        y_mod(k) = w(1) * u(k-3) + w(2) * u(k-4) +
        w(3) * y_mod(k-1) + w(4) * y_mod(k-2);
        e(k) = y_mod(k) - y(k);
    end
else
    for k=10:steps
        y_mod(k) = w(1) * u(k-3) + w(2) * u(k-4)
        + w(3) * y(k-1) + w(4) * y(k-2);
        e(k) = y_mod(k) - y(k);
    end
end
```

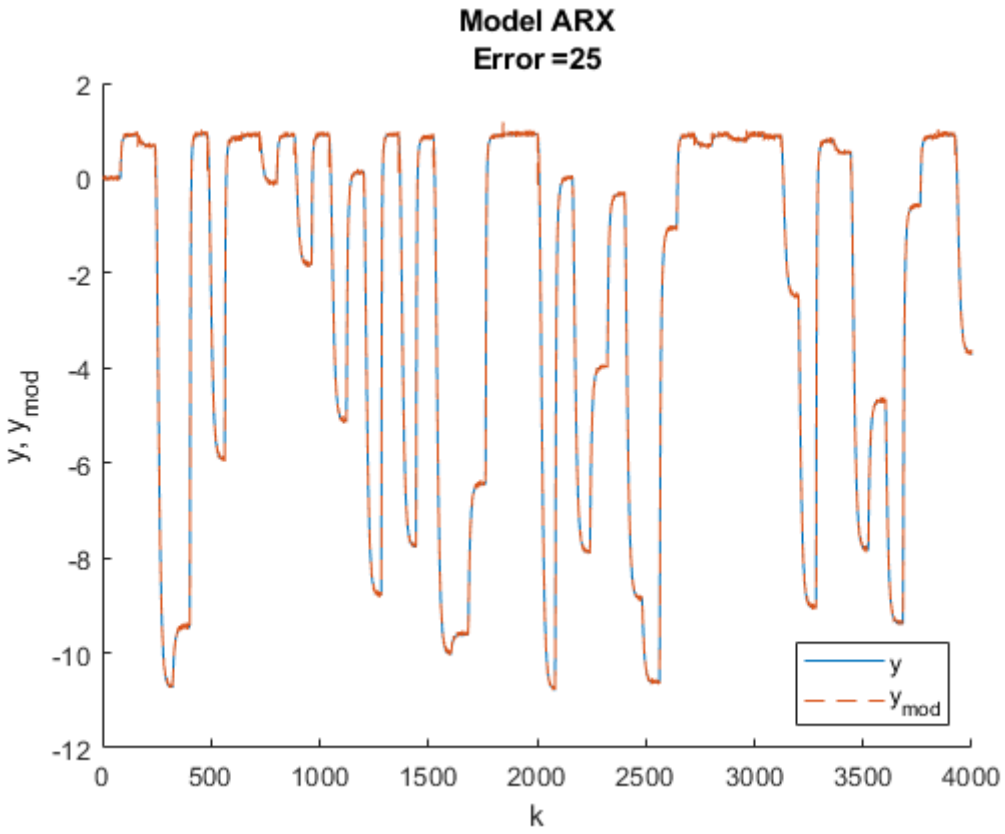


Rys. 2.14. Zbiór weryfikujący



Rys. 2.15. Zbiór uczący

Z przebiegów i wyznaczonych błędów jednoznacznie wymika, że model OE nie działa. Dzieje się tak z prostego powodu - MNK wyznacza wagi dla modelu ARX. Na przebiegu 2.16 przedstawiono dodatkowo porównanie wyjścia modelu w trybie ARX i danych weryfikujących. Uzyskany błąd jest dużo niższy niż w przypadku modelu OE, ale ponownie wystąpiło zjawisko dopasowywania się do szumu pomiarowego.



Rys. 2.16. Zbiór weryfikujący

2.3. Podsumowanie

Poniżej przedstawionio kilka najważniejszych wniosków z przeprowadzonych badań w zadaniu drugim.

— Więcej nie znaczy lepiej - zbyt duża liczba neuronów powoduje zbyt duże przystosowanie się modelu do danych uczących.
— Istotnym elementem uczenia sieci jest wybór dobrego algorytmu uczącego - przy skomplikowanych zadaniach proste metody nie sprawdzają się dobrze.
— Najmniejszy błąd nie zawsze wskazuje najlepszy model - czasami następuje adaptacja do zakłóceń, co nie jest pożądanym zjawiskiem. W takim przypadku lepiej wybrać model, który będzie bardziej "ogólny".
— Ciekawym zjawiskiem jest dobre działanie predyktora OE uczonego w trybie ARX - intuicja sugerowałaby, że taki model powinien działać źle, ale w rzeczywistości jest inaczej.
— Metoda najmniejszych kwadratów nie nadaje się do wyznaczania parametrów modeli OE - ta metoda działa tylko z modelami ARX.
— **Najlepszym wyznaczonym modelem pod względem wielkości błędu na obydwu zbiorach, jak i ogólności, jest sieć OE z sześcioma neuronami w warstwie ukrytej, uczona algortymem BGFS w trybie OE**

3. Modelowanie procesu za pomocą przyborników programu MATLAB

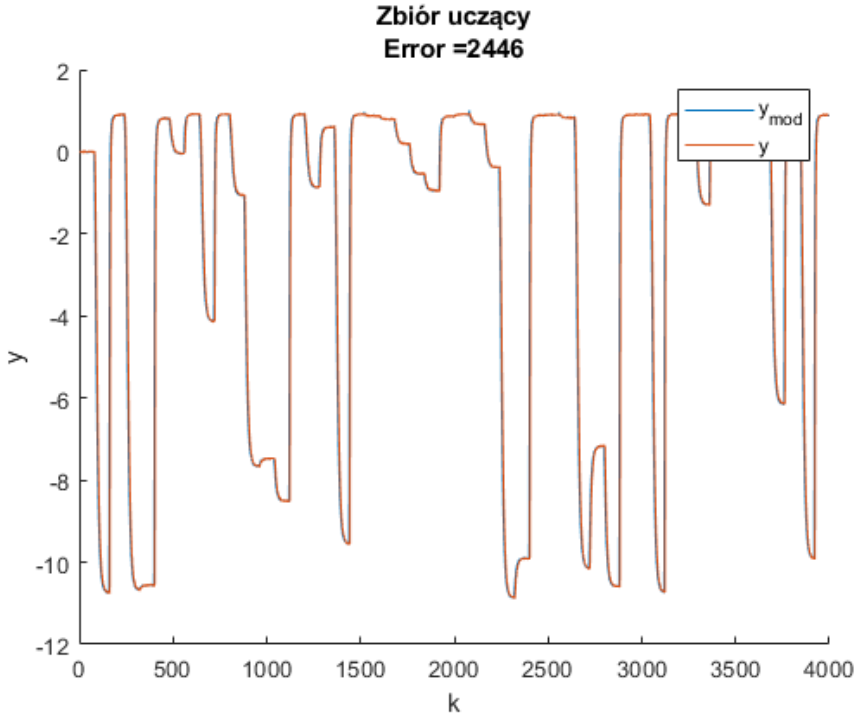
3.1. Krótki opis przybornika Deep Learning Toolbox

Deep Learning Toolbox to narzędzie, które umożliwia projektowanie i implementację sieci neuronowych z algorytmami, modelami wstępnie nauczonymi i aplikacjami. Można używać sieci neuronowych konwolucyjnych (ConvNets, CNNs) i sieci długotrwałej pamięci krótkoterminowej (LSTM) do klasyfikacji i regresji obrazów, danych szeregów czasowych i tekstów. Można budować architektury sieci, takie jak generatywne sieci przeciwników (GANs) i sieci Siamese, używając automatycznej różniczkowalności, niestandardowych pętli treningowych i udostępnionych wag.

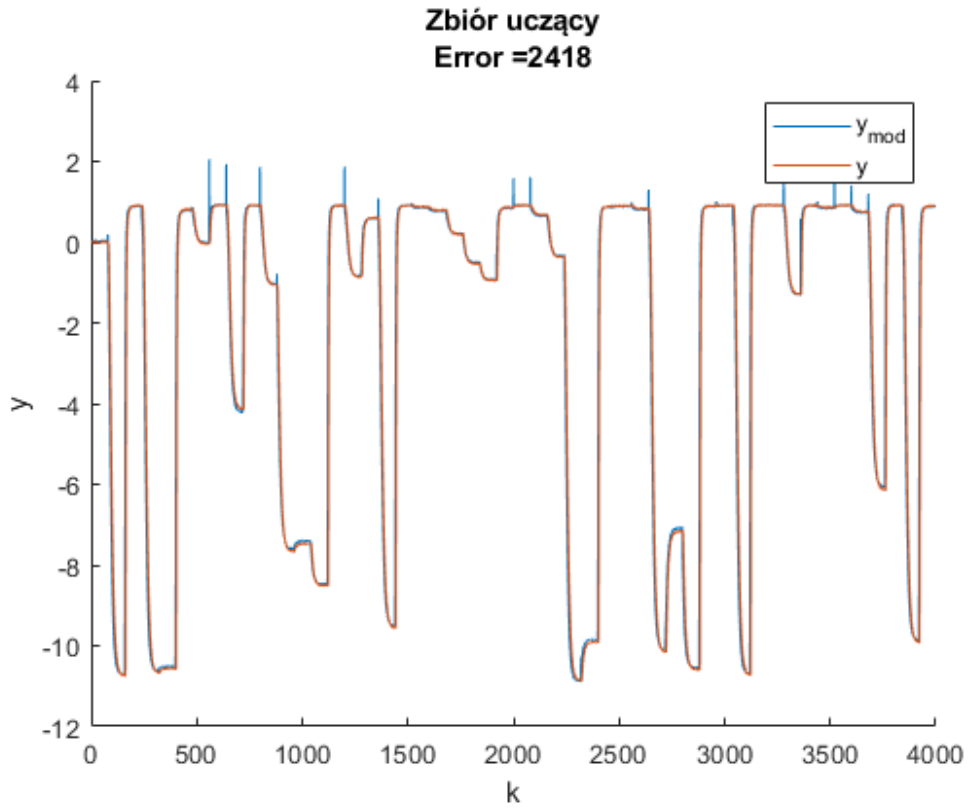
W projekcie wykorzystano sieć narxnet. Jest to nieliniowa sieć autoregresyjna z wejściem zewnętrznym. Sieci NARX mogą uczyć się przewidywać jedną serię czasową na podstawie poprzednich wartości tej samej serii czasowej, wejścia zwrotnego oraz innej serii czasowej. Idealnie nada się do modelowania obiektu dynamicznego.

3.2. Uczenie modeli neuronowych przy pomocy przybornika matalba

Uczono modele o strukturze identycznej do sieci z poprzedniego rozdziału. Zedcydowano się na sieć 6-cio neuronową, z powodów opisanych w punkcie 2.2.1.



Rys. 3.1. Uczenie sieci przy pomocy algorytmu Levenberga-Marquardta



Rys. 3.2. Uczenie sieci przy pomocy algorytmu Resilient Backpropagation

Pomimo faktu, że wykresy wyglądają w porządku, uzyskane błędy są bardzo wysokie. Dzieje się tak, ponieważ z niewiadomego powodu ustawione opóźnienie wejścia sterowania sieci (3 i 4) nie działa. Po samym wyglądzie przebiegów można jednak stwierdzić, że uczenie przebiegło poprawnie - nie wiadomo jednak z czego wynika brak opóźnień na wejściu.

Nie udało się przeprowadzić symulacji dla danych weryfikujących - MATLAB wyrzuca błąd o źle dobranych danych do wejść sieci. Jest to o tyle dziwne, że dane w pliku *dane_wer.txt* mają taką samą strukturę jak dane w pliku *dane_ucz.txt* i są obrabiane w ten sam sposób.

Nie można przeprowadzić porównania przybornika Deep Learning Toolbox z programem sieci.exe, ponieważ nie udało się zrealizowac zadania w całości. Jedynym zjawiskiem, które zaobserwowano, jest dużo szybsze uczenie się sieci przy pomocy przybornika MATLABA. Jakości uczenia nie da się porównać z powodów opisanych wcześniej.

4. Regulacja procesu

Zgodznie z poleceniem założono identyczną trajektorię zadana dla każdego z regulatorów. Ponad to regulatory NPL i GPC mają takie same horyzonty predykcji i sterowania. Błąd w kazdym regulatorze jest liczony jako sumaryczny blad kwadratowy. Zbiórcze wnioski i obserwacje znajdują się na końcu tego rozdziału.

4.1. Algorytm NPL w wersji analitycznej

4.1.1. Opis algorytmu

Algorytm NPL jest skrótem od Nieliniowej Predykcji z Linearyzacją. W tym regulatorze, w każdej iteracji, linearyzuje się sieć neuronową w aktualnym punkcie pracy i korzysta się z niej do wyznaczenia odpowiedzi skokowej. Dodatkowo sieć neuronowa jest wykorzystywana do obliczanania trajektorii swobodnej (bez linearyzacji SN). Algorytm NPL ma przewagę nad alg. NO (który będzie omówiony później), ponieważ dzięki sukcesywnej linearyzacji zadanie optymalizacji można zapisać jako zadanie optymalizacji kwadratowej - następstwem tego faktu, jest to, że regulator NPL potrzebuje mniej zasobów sprzętowych do obliczania sterowań, co ma duże znaczenie np. w dziedzinie systemów wbudowanych.

4.1.2. Opis działania

1. Pomiar wyjścia obiektu $y(k)$.
2. Obliczenie wyjścia modelu i obliczenie błędu modelu $d(k)$.
3. Obliczenie trajektorii swobodnej $Y^0(k)$ (niezlinearyzowana sieć neuronowa) z uwzględnieniem błędu $d(k)$.
4. Linearyzacja współczynników sieci neuronowej przy użyciu pochodnej cząstkowej.
5. Obliczenie, przy użyciu zlin. SN, współczynników odpowiedzi skokowej, ze wzoru:

$$s_j = \sum_{i=1}^{min(j,n_b)} b_i - \sum_{i=1}^{min(j-1,n_a)} a_i * s_{j-i} \tag{4.1}$$

$$j \in 1, 2, ..., N$$

6. Obliczyć macierze $M(k)$ i $K(k)$:

$$\boldsymbol{M(k)} = \begin{bmatrix} s_1(k) & 0 & \dots & 0 \\ s_2(k) & s_1(k) & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ s_N(k) & s_{N-1}(k) & \dots & s_{N-N_u+1}(k) \end{bmatrix}_{\mathrm{NxN_u}} \tag{4.2}$$

$$K(k) = (M(k)^T M(k) + \lambda * I)^{-1} M(k)^T \tag{4.3}$$

7. Obliczyć macierz $\triangle u$ przyszłych sterowań:

$$\triangle u(k) = K * (Y_{ZAD}(k) - Y^0(k)) \tag{4.4}$$

Gdzie:

$$\boldsymbol{Y_{zad}} = \begin{bmatrix} y_{zad}(k) \\ y_{zad}(k) \\ \vdots \\ y_{zad}(k) \end{bmatrix}_{\text{Nx1}} \tag{4.5}$$

$$\boldsymbol{Y^0} = \begin{bmatrix} y^0(k+1|k) \\ y^0(k+2|k) \\ \vdots \\ y^0(k+N|k) \end{bmatrix}_{\text{Nx1}} \tag{4.6}$$

8. Obliczyć sterowanie $U(k) = u(k-1) + \triangle u(k|k)$
9. (opcjonalne) Ograniczyć sterowanie $u(k)$:

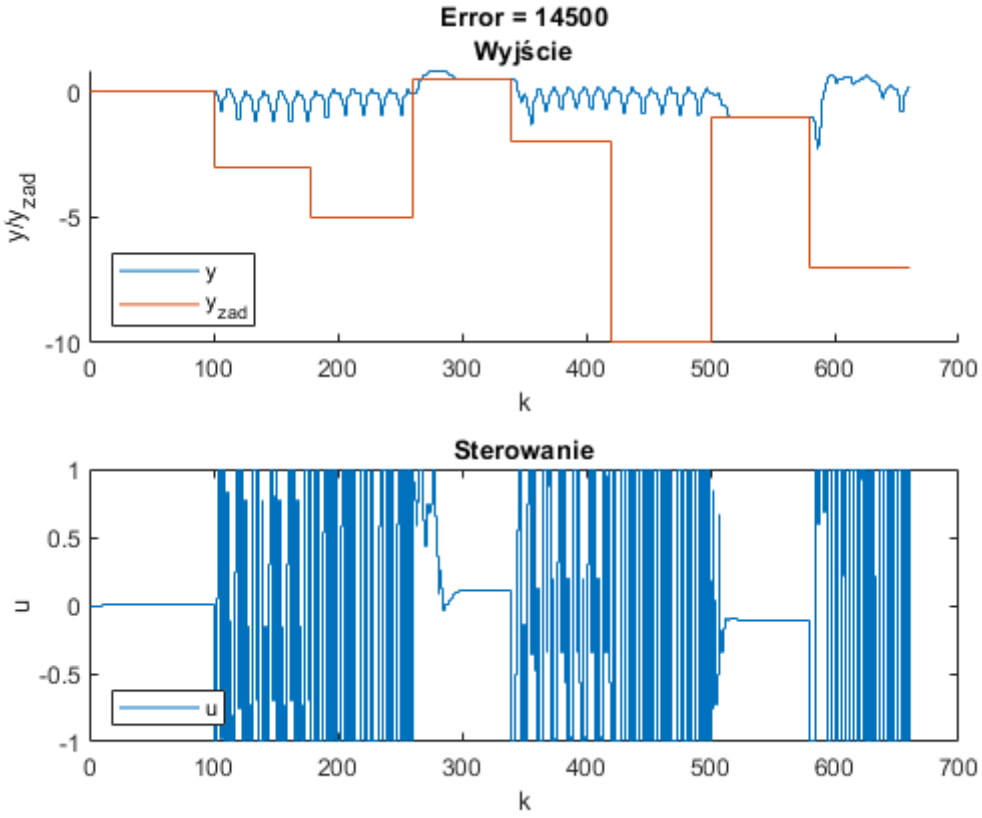
$$u(k) = min(U_{max}, u(k)) \tag{4.7}$$

$$u(k) = max(U_{min}, u(k)) \tag{4.8}$$

10. Przejdź do punktu 1

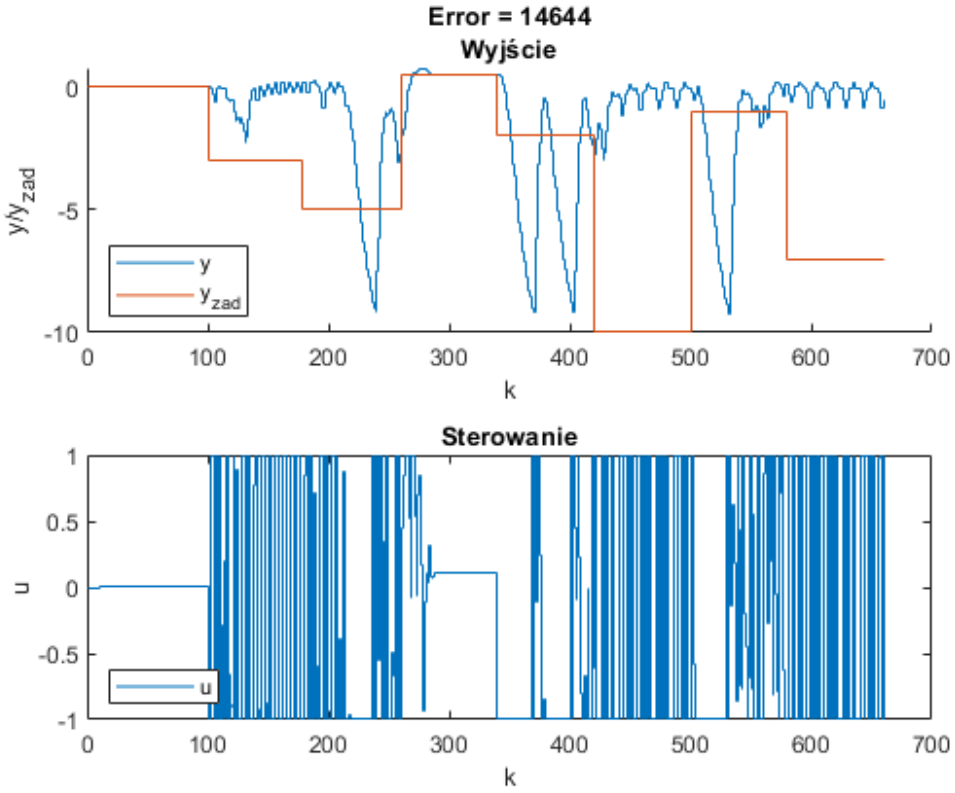
4.1.3. Strojenie i badanie działania regulatora NPL

Strojenie rozpoczęto od parametrów zarekomendowanych przez Prowadzącego: $N = 10$, $N_u = 3$ i $Lambda = 1$.

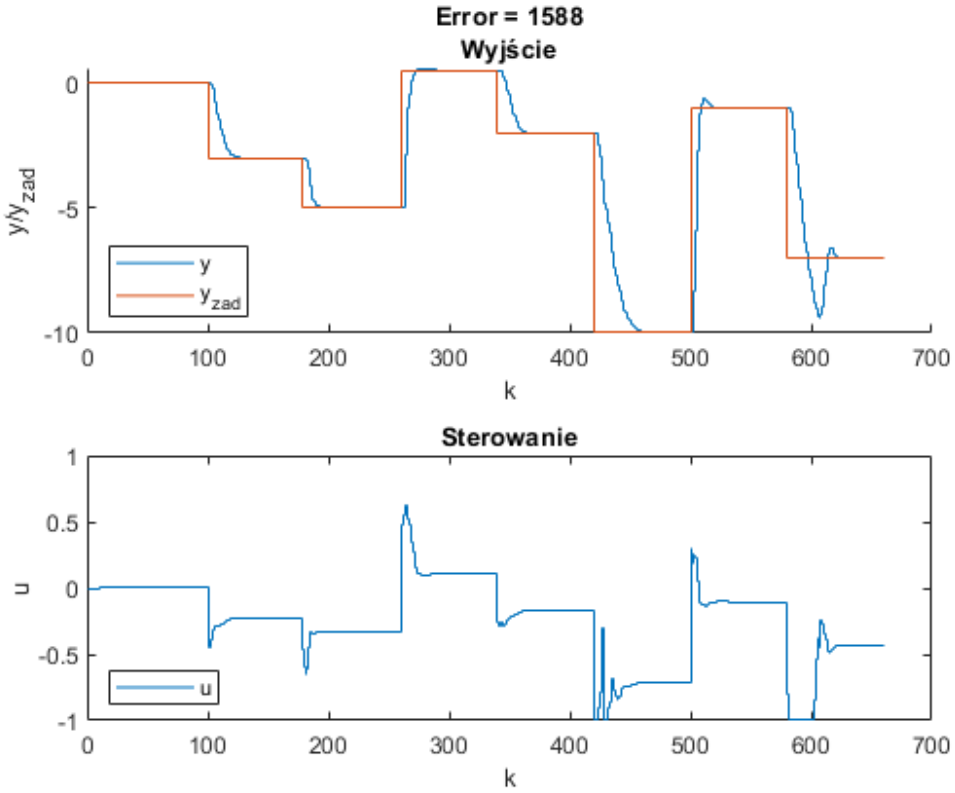


Rys. 4.1. $N = 10$; $N_u = 3$; $\lambda = 1$

Regulator nie działa z takimi nastawami, w oczy rzuca się duża zmienność sterowania. Należy zmiejszyć λ, możliwe, że wydłużenie horyzontu predykcji również poprawi jakość regulacji. Na wykresach 4.2 i 4.3 przedstawiono dalszy przebieg strojenia regulatora.

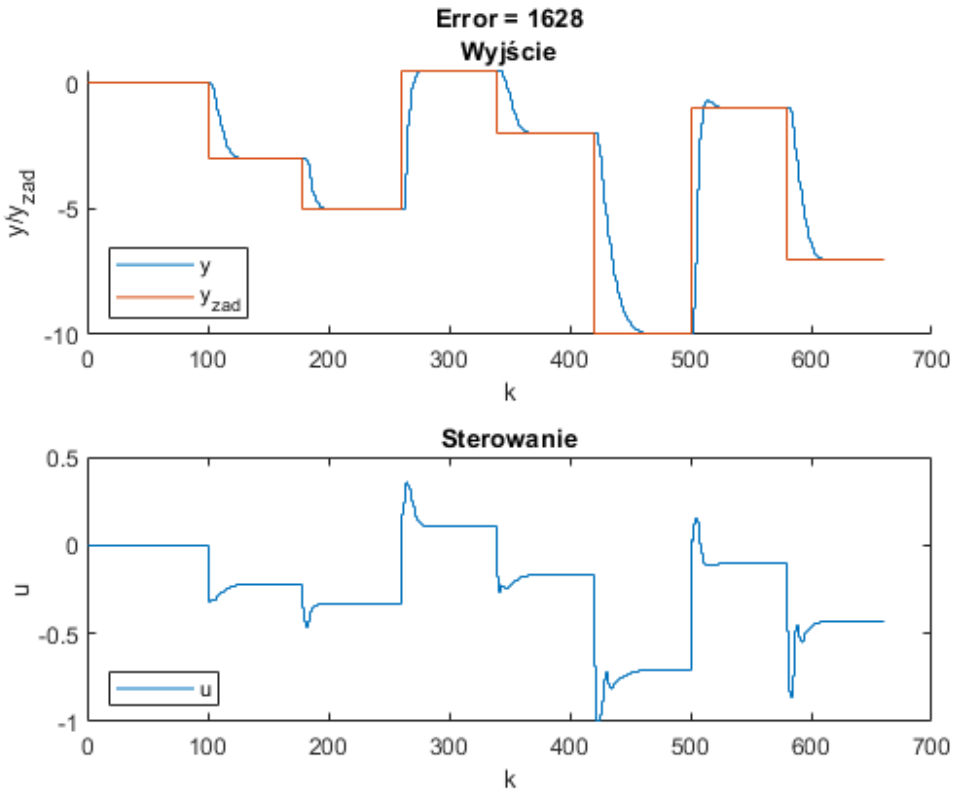


Rys. 4.2. $N = 30$; $N_u = 3$; $\lambda = 1$



Rys. 4.3. $N = 30$; $N_u = 3$; $\lambda = 50$

Samo zwiększenie parametru N nie wiele dało - to utwierdziło autorów sprawozdania, że głównym problemem są zbyt gwałtowne zmiany sterowania. Z tego powodu zwiększono drastycznie wartość parametru λ (50 razy). To działanie przyniosło oczekiwane efekty - obiekt jest całkiem dobrze regulowany, jedynym mankamentem jest przesterowanie w ostatnim skoku y_{zad}. Postanowiono zniwelować ten problem ponownie zwiększając parametr λ (wykres 4.4).



Rys. 4.4. $N = 30$; $N_u = 3$; $\lambda = 150$

Przesterowanie zniknęło, ale minimalnie zwiększył się błąd sumaryczny. Mimo to, zdecydowano, że są to najlepsze nastawy dla tego regulatora. Zapewniaja one dobrą jakość regulacji i nie powodują nadmiernego obciążenia obliczeniowego (krótkie horyzony predykcji i sterowania).

4.2. Algorytm GPC w wersji analitycznej

4.2.1. Opis algorytmu

Można powiedziec, że algorytm GPC jest ogólniejszą wersją regulatora DMC. Zamiast modelu w postacii odpowiedzi skokowej (DMC), korzystamy z modelu w postaci równań różnicowych (GPC). Pozwala to zastosować ten regulator do szerszego wachlarza obiektów (nie zawsze pozyskiwanie odpowiedzi skokowej jest możliwe/wskazane).

Poszczególne kroki algorytmu:

1. Uzyskać model liniowy obiektu (offline)
2. Obliczyć współczynniki odpowiedzi skokowej ze wzoru (offline):

$$s_j = \sum_{i=1}^{min(j,n_b)} b_i - \sum_{i=1}^{min(j-1,n_a)} a_i * s_{j-i} \tag{4.9}$$
$$j \in 1, 2, ..., N$$

3. Obliczyć macierze M i K (offline)

$$\boldsymbol{M} = \begin{bmatrix} s_1 & 0 & \dots & 0 \\ s_2 & s_1 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ s_N & s_{N-1} & \dots & s_{N-N_u+1} \end{bmatrix}_{\mathrm{NxN_u}} \tag{4.10}$$

$$K = (M^T M + \lambda * I)^{-1} M^T \tag{4.11}$$

4. Odczytać wyjście obiektu
5. Obliczyć błąd modelu d(k) ze wzoru:

$$d(k) = y(k) - y_{mod}(k) \tag{4.12}$$

6. Obliczyć trajektorię swobodną przy pomocy modelu liniowego $Y^0(k)$
7. Obliczyć macierz $\triangle u$ przyszłych sterowań:

$$\triangle u(k) = K * (Y_{ZAD}(k) - Y^0(k)) \tag{4.13}$$

Gdzie:

$$\boldsymbol{Y_{zad}} = \begin{bmatrix} y_{zad}(k) \\ y_{zad}(k) \\ \vdots \\ y_{zad}(k) \end{bmatrix}_{\mathrm{Nx1}} \tag{4.14}$$

$$\boldsymbol{Y^0} = \begin{bmatrix} y^0(k+1|k) \\ y^0(k+2|k) \\ \vdots \\ y^0(k+N|k) \end{bmatrix}_{\mathrm{Nx1}} \tag{4.15}$$

8. Obliczyć sterowanie:

$$u(k) = u(k-1) + \triangle u(k|k); \tag{4.16}$$

9. (opcjonalne) Ograniczyć sterowanie $u(k)$:

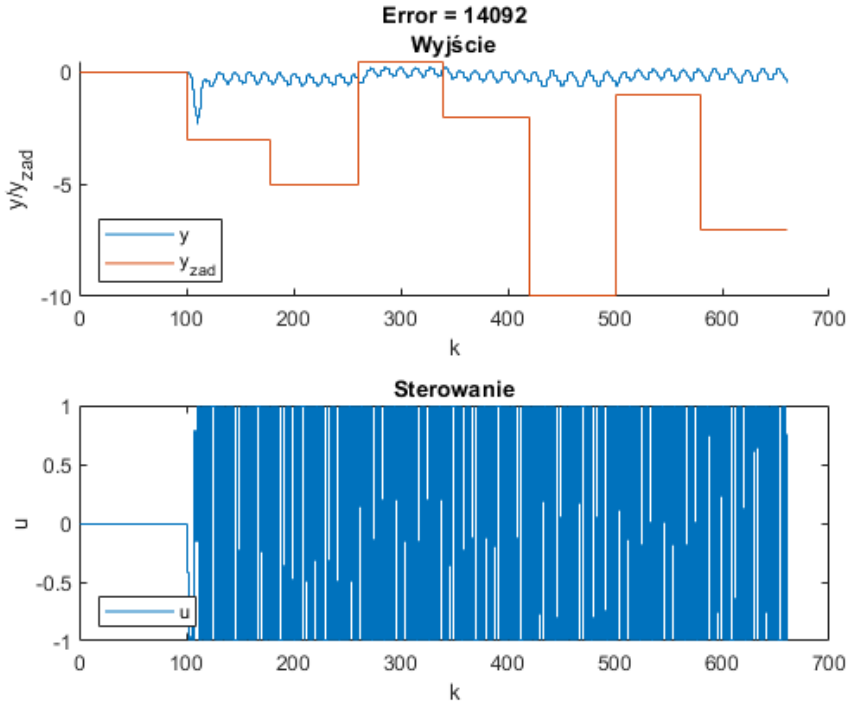
$$u(k) = min(U_{max}, u(k)) \tag{4.17}$$

$$u(k) = max(U_{min}, u(k)) \tag{4.18}$$

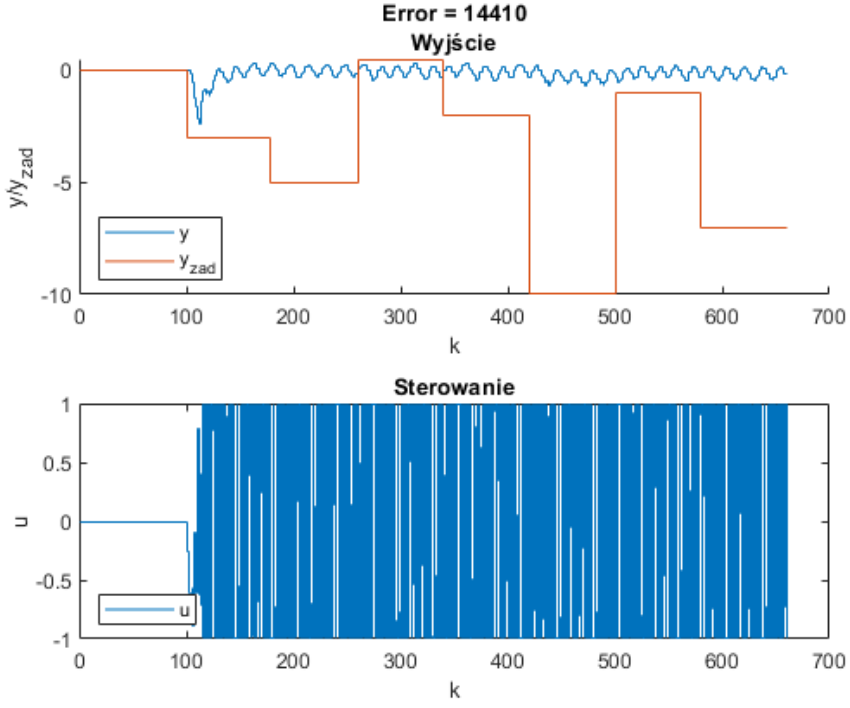
10. Wróć do pkt 4.

4.2.2. Strojenie i badanie działania algorytmu GPC

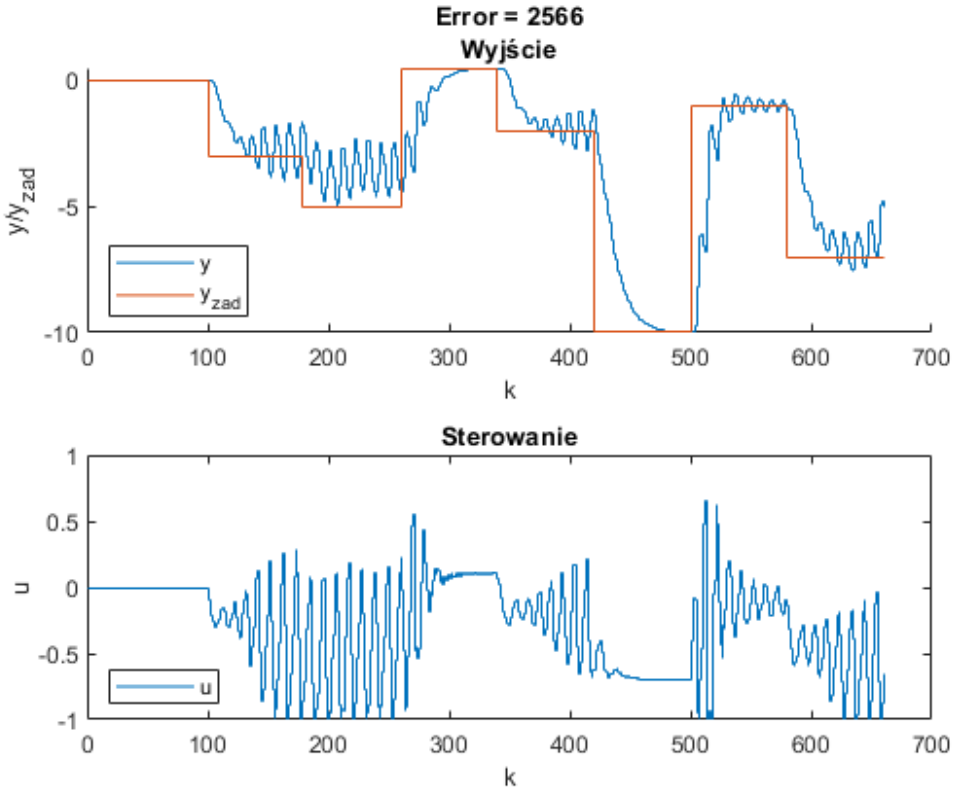
Strojenie regulatora GPC rozpoczęto od sprawdzenia, jak poradzą sobie najlepsze nastawy regulatora NPL (wykres 4.5). Następnie stopniowo zwiększano wartość parametru λ, aż do uzyskania jakkolwiek satysfakcjonujących wyników.



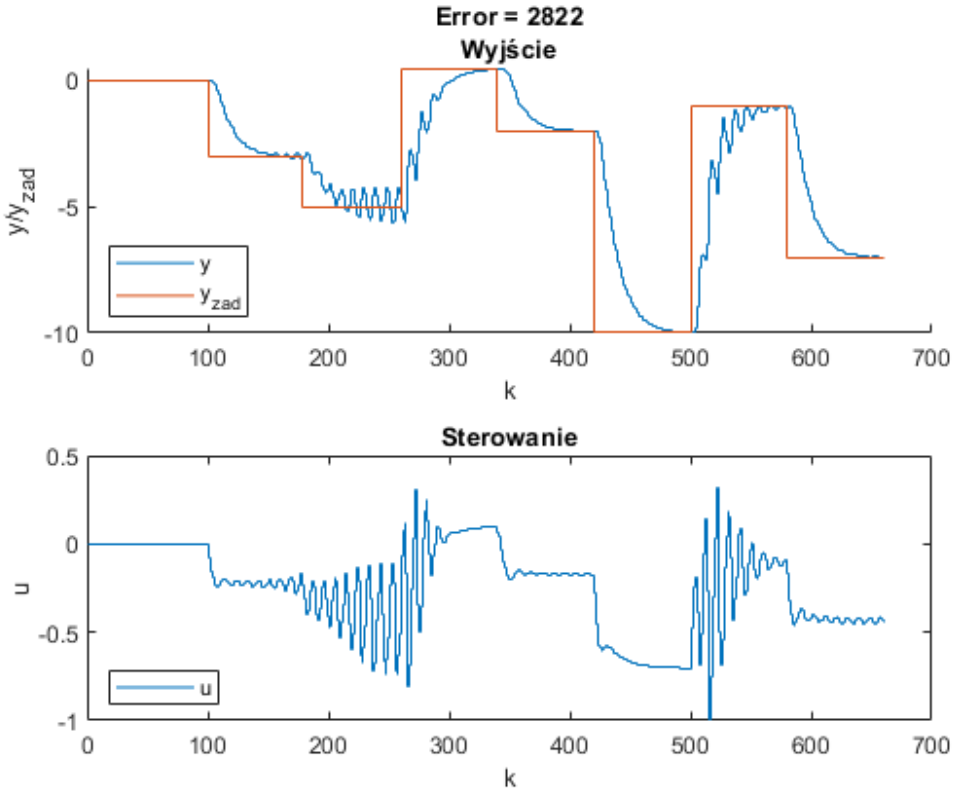
Rys. 4.5. $N = 30$; $N_u = 3$; $\lambda = 150$



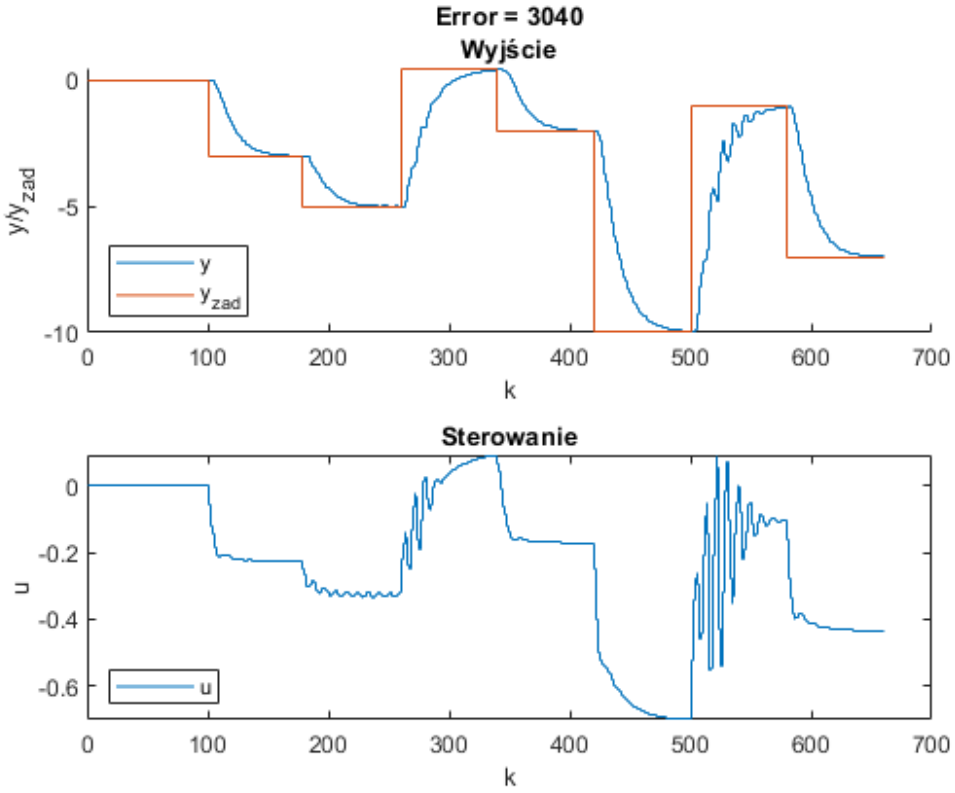
Rys. 4.6. $N = 30$; $N_u = 3$; $\lambda = 1000$



Rys. 4.7. $N = 30$; $N_u = 3$; $\lambda = 5000$



Rys. 4.8. $N = 30$; $N_u = 3$; $\lambda = 7500$



Rys. 4.9. $N = 30$; $N_u = 3$; $\lambda = 10000$

Z załączonych wykresów jasno wynika, że regulator GPC nie radzi sobie z obiektem nieliniowym. Nie jest to żadne zaskoczenie - GPC korzysta z modelu wyznaczonego przy pomocy Metody Najmniejszych Kwadratów. Dodatkowo sam model pracuje w trybie OE - przy ich omawianiu omówiono już skuteczność MNK i wyjaśniono dlaczego działa tak słabo. Przy wykorzystaniu tego modelu w regulatorze GPC tamte obserwacje się potwierdziły.

Z orginalnymi nastawami regulatora NPL, algorytm GPC nie działa wcale.

4.3. Algorytm PID

4.3.1. Krótki opis algorytmu

Zaimplementowano regulator PID, który jest opisany następującymi równaniami:

$$\begin{cases} r_0 = K * (1 + \frac{T_p}{2T_i} + \frac{T_d}{T_p}) \\ r_1 = K * (\frac{T_p}{2T_i} - \frac{2T_d}{T_p} - 1) \\ r_2 = \frac{K * T_d}{T_p} \\ u(k) = r_2 * e(k-2) + r_1 * e(k-1) + r_0 * e(k) + u(k-1) \end{cases} \tag{4.19}$$

Gdzie K to wzmocnienie, T_i stała czasowa całkowania, T_d stała czasowa różniczkowania i T_p czas próbkowania.

4.3.2. Strojenie i badanie działania algorytmu PID

Strojenie regulatora PID przeprowadzono metodą Zieglera-Nicholsa. Na wykresach 4.10 do 4.13 przedstawiono kolejno podejmowanie kroki.

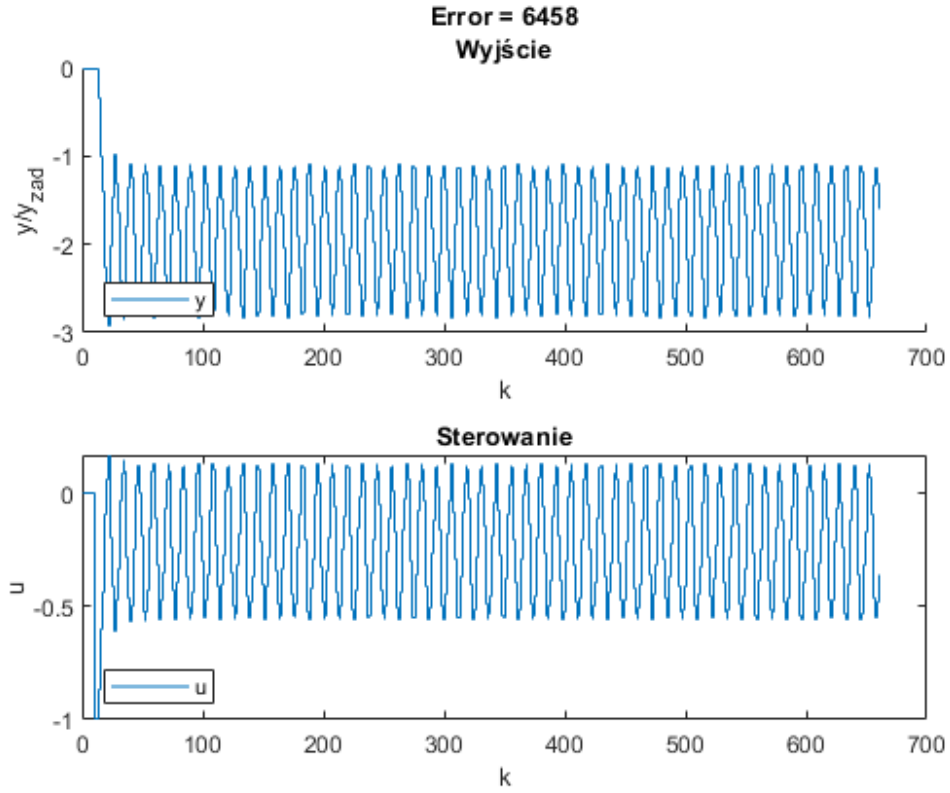
Krótki opis metody Zieglera-Nicholsa:

1. Doprowadzić układ do drań niegasnących i nierosnących.
2. Odczytać wzmocnienie krytyczne K_{kryt} i okres drgań T_{kryt}
3. Obliczyć nastawy regulatora korzystając z tableki
4. (ocjonalne) Poprawić nastawy ręcznie do mommentu uzyskania satysfakcjonujących wyników

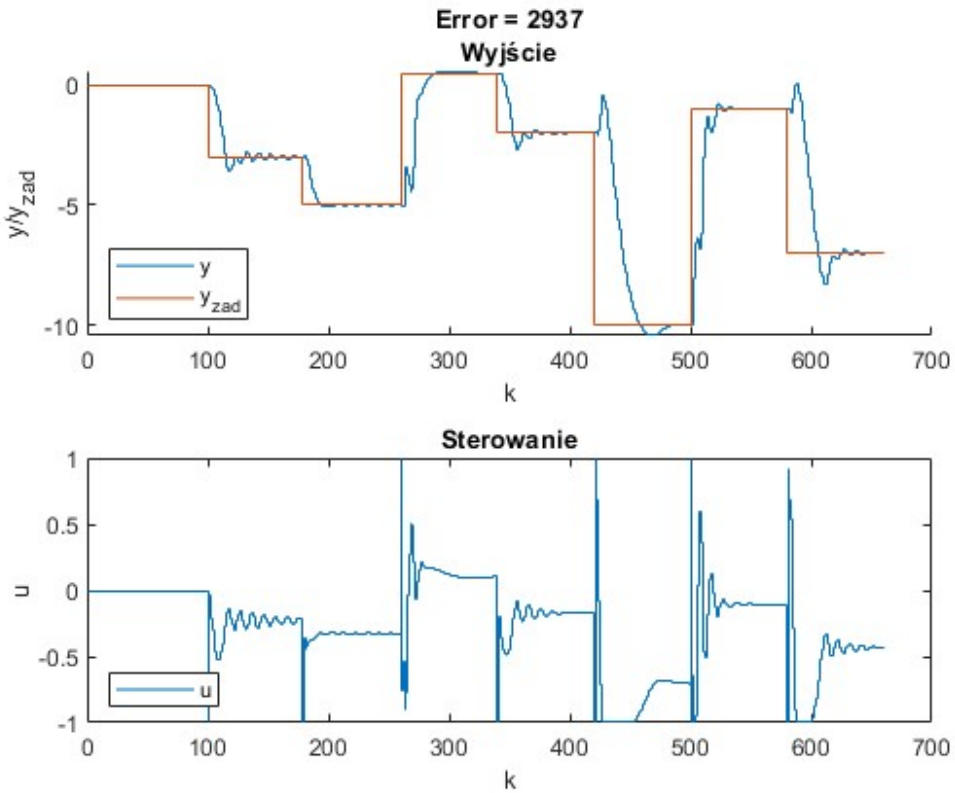
Ostatecznie obliczone nastawy mają wartośc:

$$\begin{aligned} K &= 0.24 \\ T_i &= 6 \\ T_d &= 1.5 \end{aligned} \tag{4.20}$$

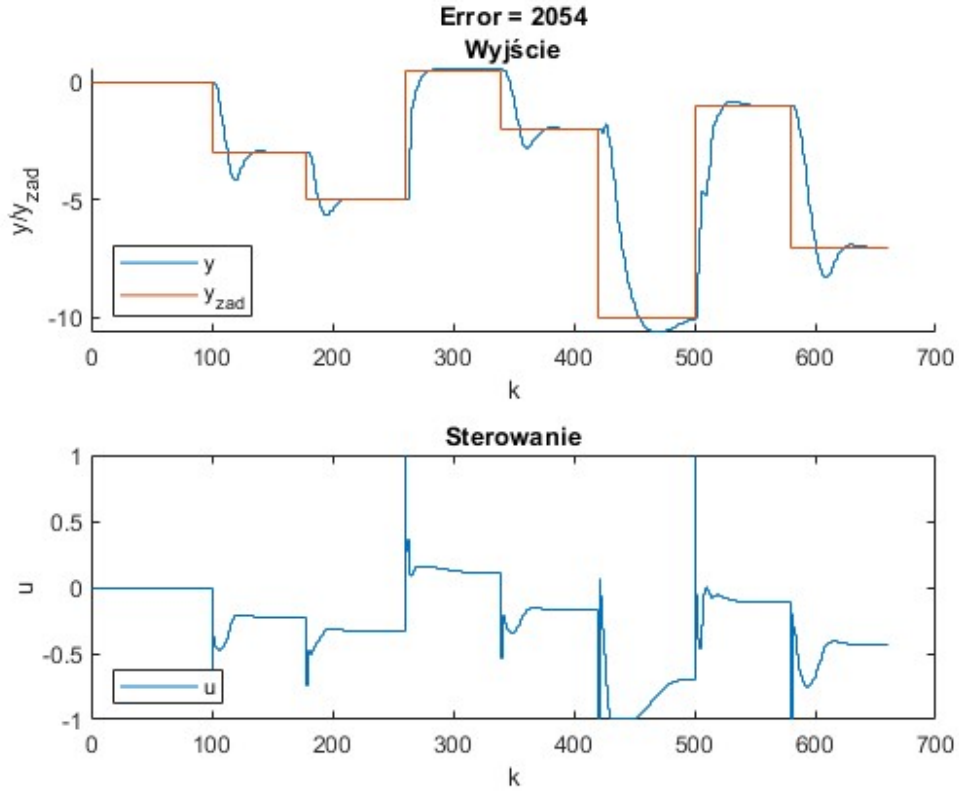
Następnie oryginlane nastawy Ziglera-Nicholsa zmodyfikowano, w celu poprawienia jakości regulacji. Zmian dokonywano arbitralnie, bazując na własnych doświadczeniach z regulatorami PID.



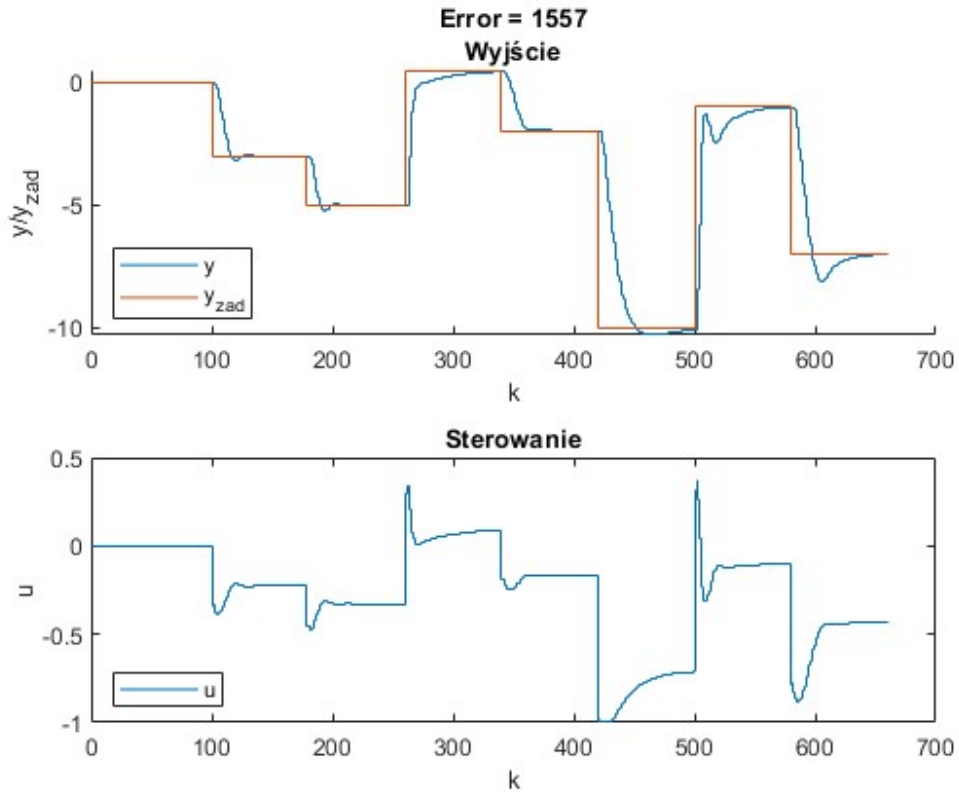
Rys. 4.10. Drgania niegasnące; $K_{kryt} = 0.4$, $T_{kryt} = 12$



Rys. 4.11. Regulacja PID - orginalne nastawy metody Ziglera-Nicholsa; $K = 0.24$, $T_i = 6, T_d = 1.5$



Rys. 4.12. Regulacja PID; $K = 0.1$, $T_i = 6, T_d = 1.5$



Rys. 4.13. Regulacja PID; $K = 0.1$, $T_i = 12, T_d = 0.1$

4.4. Regulator NO

Regulator z Nieliniową Optymalizacją działa na podobnej zasadzie co algorytm NPL, z tą różnicą, że nie występuje w nim linearyzacja modelu neuronowego - w każdej iteracji trzeba rozwiązać nieliniowe zadanie optymalizacji z wykorzystaniem wskaźnika jakości:

$$J(k) = \sum_{p=1}^{N} (y^{zad}(k) - \hat{y}(k+p|k))^2 + \lambda \sum_{p=0}^{N_u} (\Delta u(k+p|k))^2 \tag{4.21}$$

Wyznacza się takie sterowania, dla których ten wskaźnik jest najmniejszy.

Predykcja wyjścia jest liczona jako:

$$\hat{y}(k+p|k) = w20 + w2 * tanh(w10 + w1 * q) + d(k) \tag{4.22}$$

Mając wyznaczone wszystkie wartości można obliczyć zadanie optymalizacji.

4.4.1. Strojenie i badznie działania algorytmu NO

Nie udało się zaimplementować tego regulatora.

4.5. Podsumowanie przeprowadzonych eksperymentów

1. **Regulator NPL**, zdaniem autorów sprawozdania, radzi sobie najlepiej z regulacją danego w zadaniu procesu. Przy odpowiednim doborze parametrów nie występują przeregulowania, a sama regulacja jest szybka. Dodatkowo warto zwrócić uwagę na "wygodę" związaną z wykorzystaniem tego regulatora - jeśli potrzebna jest szybsza regulacja, z dozwolonymi przeregulowaniami, wystarczy zmiejszyć parametr λ; analogiczne dla odwrotnej sytuacji - jeśli potrzebna jest regulacja bez przesterowań, nawet kosztem szybkości algorytmu, wystarczy zwiększyć λ.

2. **Regulator GPC**, zgodnie z wnioskami i obserwacjami z punktu 4.2.2, z orginalnymi nastawami regulaotra NPL nie działa wcale. Dopiero po drastycznym zwiększneiu parametru ($\lambda \geqslant 7500$) GPC zaczyna działać jakkolwiek poprawinie. Nadal jednak widoczne są oscylacje wyjścia wokół y_{zad}. Dodatkowo sama regualcja jest bardzo wolna. Biorąc to wszystko pod uwagę można jednoznacznie stwierdzić, że regulator GPC nie nadaje się do regulacji procesów nieliniowych.

3. **Regulator PID** okazał się być sporym zaskoczeniem - po odpowiednim wystrojeniu uzyskiwane na nim błędy są na poziomie błędów z algorytmu NPL. Jeśli doda się do tego zdecydowanie mniejszą złożoność obliczeniową od pozostałych algorytmów, to PID staje się nienajgorszym wyborem do regulacji tego procesu. Dwa aspekty przemawiają jednak na niekorzyść tego algorytmu:

   a) Trudność strojenia - dużo łatwiej jest dobrze wystroić NPL niż PID-a (przy procesach nieliniowych).
   b) NPL jest "bardziej godny zaufania" - NPL będzie działał zawsze co najmniej dobrze dla obiektów nieliniowych, natomiast w przypadku PID-a sytuacja jest "lokalna" tzn. dla jednego obiektu może działać dobrze, a dla drugiego już nie.

   Z tego powodu PID kończy porównanie jako drugi najlepszy przebadany, w tym projekcie, regulator.