

SPRAWOZDANIE Z DRUGIEGO PROJEKTU  
Z PRZEDMIOTU  
„SZTUCZNA INTELIGENCJA W  
AUTOMATYCE”

Numer zadania: 10

Wykonawcy:

Daniel Giełdowski

Piort Chachuła

# Spis treści

<b>1. Symulacja procesu</b>	2
1.1. Charakterystyka statyczna	2
1.2. Zbiory danych	3
<b>2. Modelowanie procesu</b>	5
2.1. Opóźnienie	5
2.2. Dobór liczby neuronów	6
2.3. Model z algorytmu BFGS	7
2.4. Symulacja modelu z algorytmu BFGS	8
2.5. Model z algorytmu najszybszego spadku	9
2.6. Model z algorytmu BFGS z uczeniem bez rekurencji	10
2.7. Symulacja modelu z algorytmu BFGS z uczeniem bez rekurencji	11
2.8. Model metodą najmniejszych kwadratów	12
<b>3. Regulacja procesu</b>	13
3.1. Implementacja NPL	13
3.2. Strojenie NPL	14
3.3. GPC	17
<b>4. Zadania dodatkowe</b>	19
4.1. PID	19
4.2. NO	21
<b>5. Używane skrypty</b>	22

# 1. Symulacja procesu

## 1.1. Charakterystyka statyczna

Zadany układ opisany jest równaniami:

$$\begin{cases} x_1(k) = -\alpha_1 x_1(k-1) + x_2(k-1) + \beta_1 g_1(u(k-3)) \\ x_2(k) = -\alpha_2 x_1(k-1) + \beta_2 g_1(u(k-3)) \\ y(k) = g_2(x_1(k)) \end{cases} \quad (1.1)$$

gdzie  $u$ -sygnał wejściowy,  $y$ -sygnał wyjściowy,  $x_1, x_2$  - zmienne stanu,  $\alpha_1 = -1,422574$ ,  $\alpha_2 = 0,466776$ ,  $\beta_1 = 0,017421$ ,  $\beta_2 = 0,013521$  oraz

$$g_1(u(k-3)) = \frac{\exp(5u(k-3)) - 1}{\exp(5u(k-3)) + 1}, \quad g_2(x_1(k)) = 1 - \exp(-1.5x_1(k)) \quad (1.2)$$

Podany punkt pracy układu to  $u = y = x_1 = x_2 = 0$ , więc w wersji statycznej:

$$\begin{cases} x_1 = -\alpha_1 x_1 + x_2 + \beta_1 g_1(u) \\ x_2 = -\alpha_2 x_1 + \beta_2 g_1(u) \\ y = g_2(x_1) \end{cases} \quad (1.3)$$

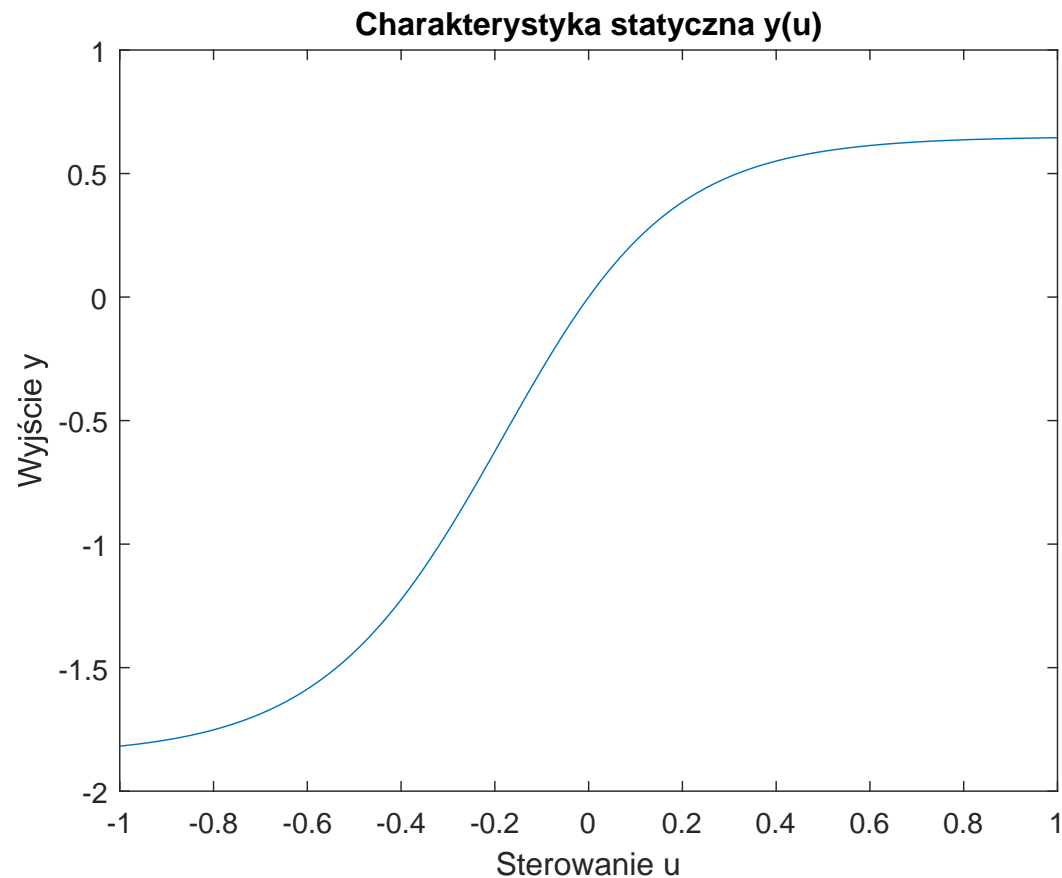
Po przekształceniach:

$$x_1 = \frac{(\beta_1 + \beta_2)g_1(u)}{1 + \alpha_1 + \alpha_2} \quad (1.4)$$

Po podstawieniu równania (1.4) do  $y$  otrzymujemy

$$y(u) = g_2\left(\frac{(\beta_1 + \beta_2)g_1(u)}{1 + \alpha_1 + \alpha_2}\right) \quad (1.5)$$

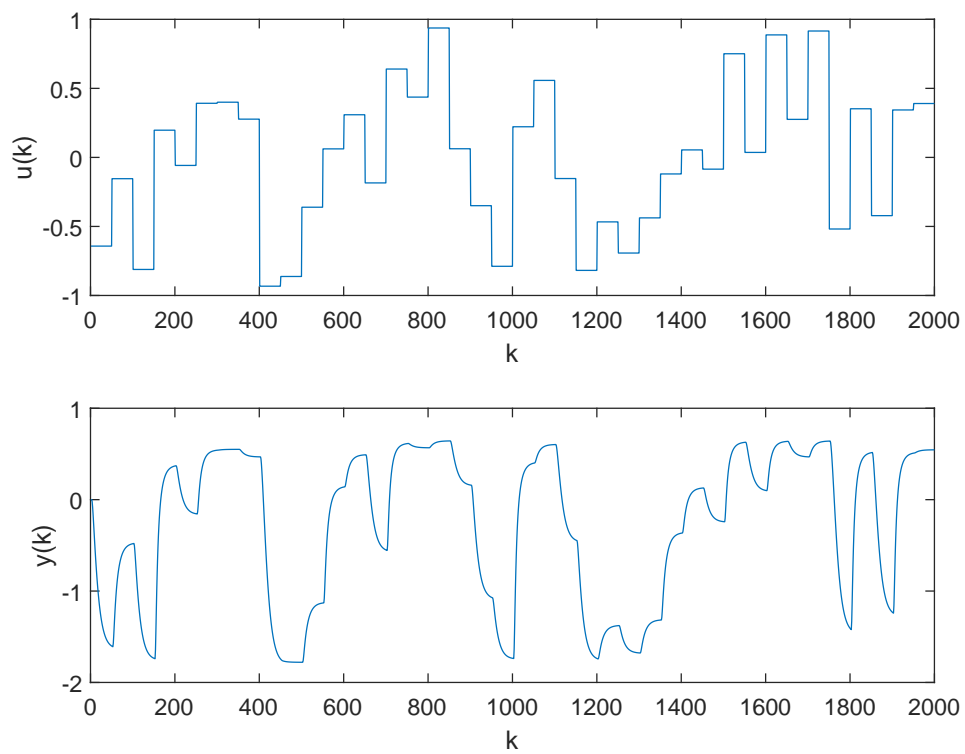
Wykres wyznaczonej charakterystyki statycznej dla zadanego zakresu wartości sterowania ( $u^{\min} = -1, u^{\max} = 1$ ) przedstawiony został na wykresie 1.1. Wykres został wygenerowany za pomocą skryptu *charakterystyka\_statyczna.m*.



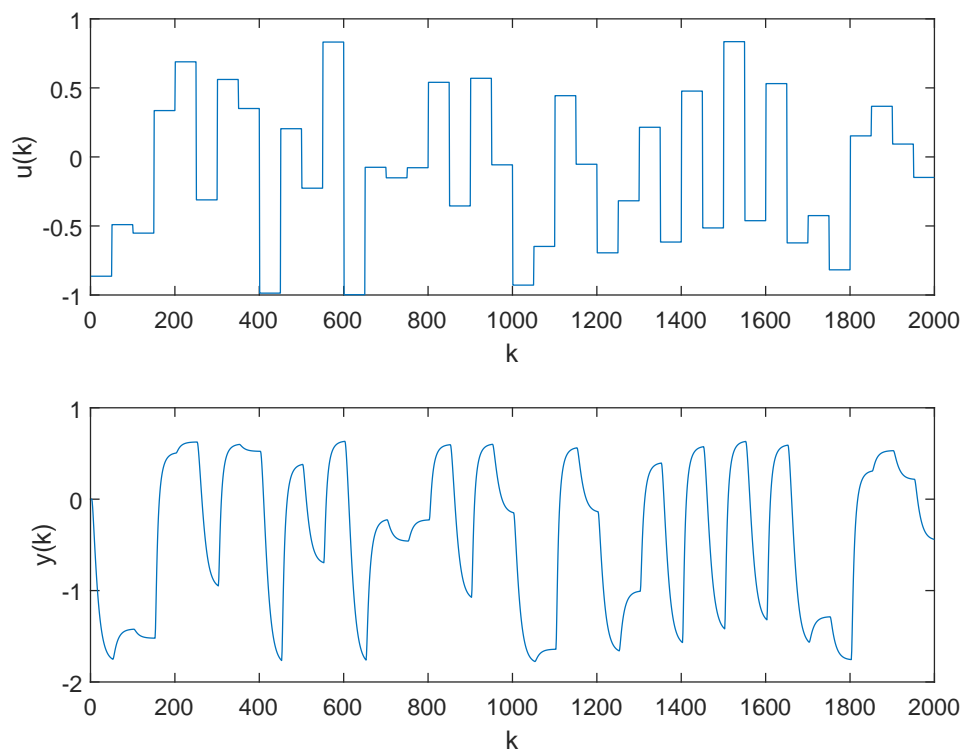
Rys. 1.1. Charakterystyka statyczna procesu

## 1.2. Zbiory danych

W celu przygotowania do uczenia sieci neuronowych wygenerowaliśmy dwa zbiory danych. Dane zostały wygenerowane poprzez zasymulowanie zadanego procesu dla sygnału sterowania złożonego o wartości zmieniającej się skokowo co 50 próbek. Obydwa zbiory danych mają po 2000 próbek. Zostały one przedstawione na wykresach 1.2 i 1.3. Użyte zostały skrypty: *generowanie\_danych.m* (do wygenerowania danych) oraz *wykres\_danych.m* (do narysowania wykresów).



Rys. 1.2. Dane uczące

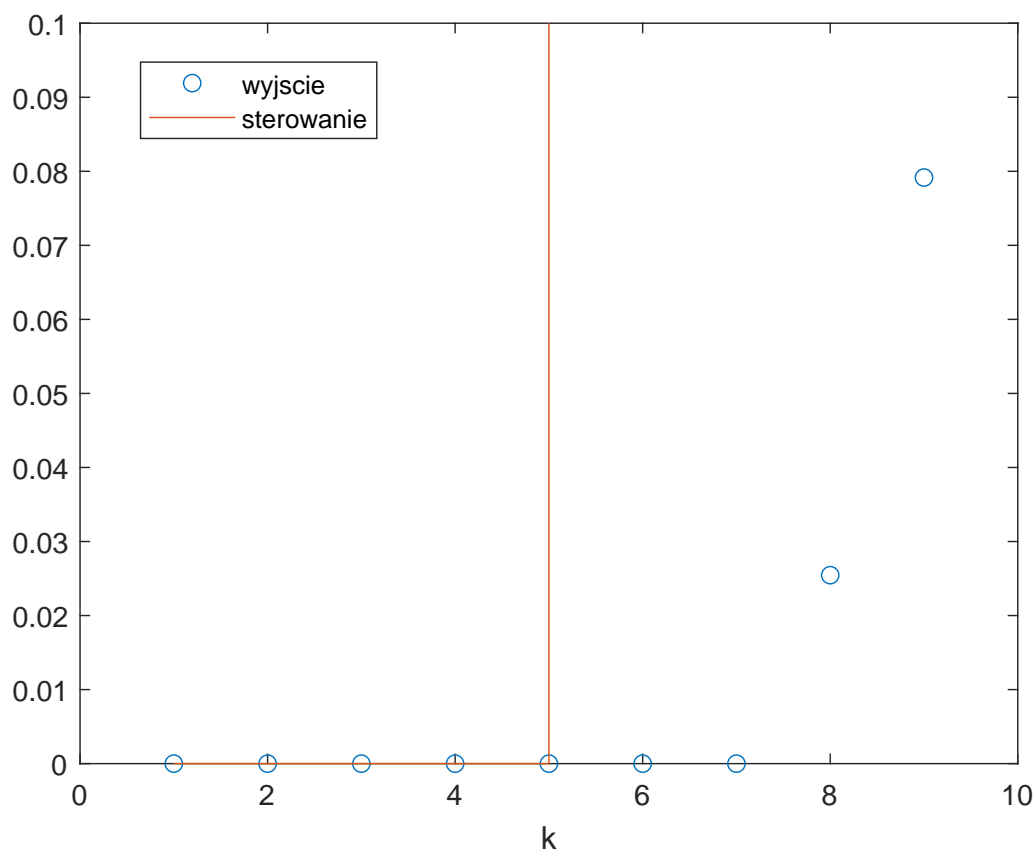


Rys. 1.3. Dane weryfikujące

## 2. Modelowanie procesu

### 2.1. Opóźnienie

W celu zdefiniowania opóźnienia  $\tau$  procesu zasymulowaliśmy go dla pojedynczego skoku sterowania. Wyniki symulacji przedstawione są na wykresie 2.1. Skok sterowania nastąpił w 5 kroku działania programu, natomiast wyjście procesu zmieniło się dopiero w kroku 8. Oznacza to, że poszukiwane przez nas opóźnienie wynosi  $\tau = 3$ . Użyty przez nas skrypt to *tauwiz.m*.



Rys. 2.1. Wizualizacja opóźnienia procesu

## 2.2. Dobór liczby neuronów

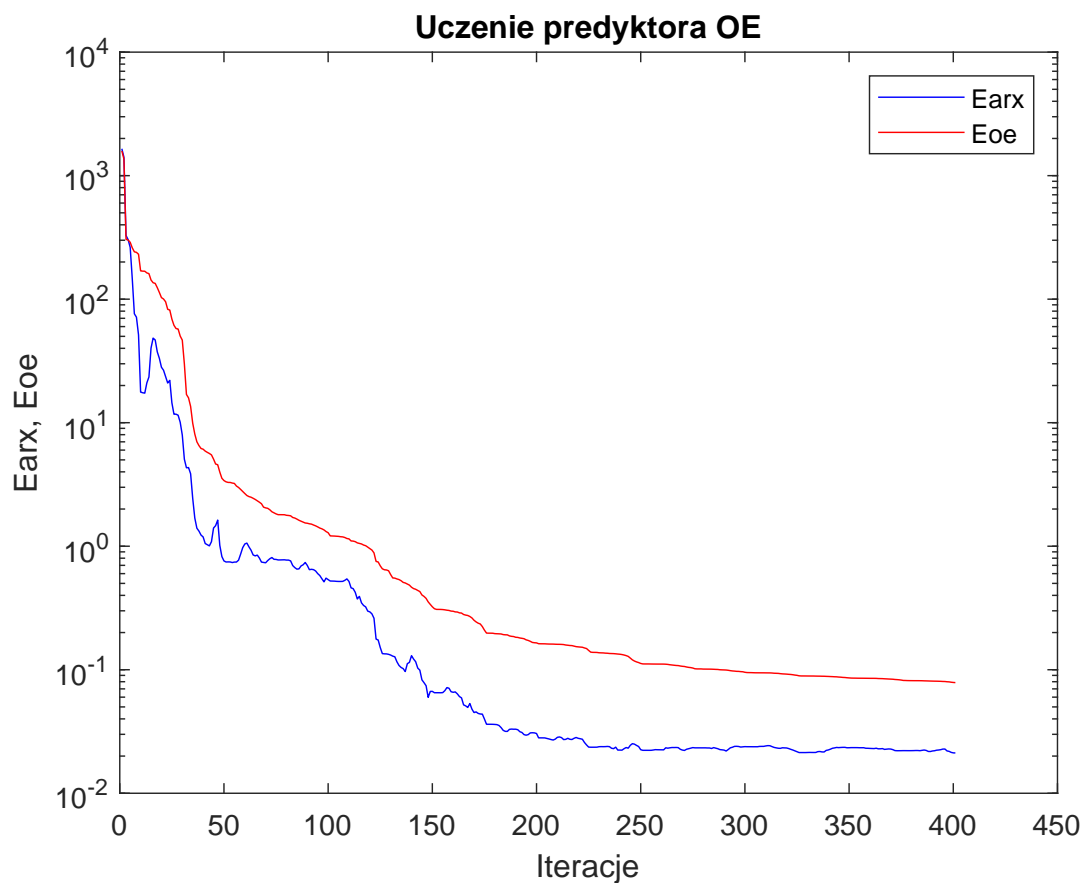
W celu dobrania odpowiedniej liczby neuronów dla sieci zastosowaliśmy wielokrotne uczenie z użyciem programu *sieci.exe*. Dla każdej ilości neuronów ukrytych od 1 do 10 dokonaliśmy 5 procesów uczenia za pomocą algorytmu BFGS z wykorzystaniem rekurencji. W tym celu wyko-rzystaliśmy skrypt *modelowanie.m*. Najmniejszy uzyskany błąd uczenia wraz ze skojarzonym z nim błędem weryfikacji przedstawiony został w tabeli poniżej. Najmniejszy błąd dla obydwu zbiorów występuje dla 9 neuronów. Ostatecznie jednak zdecydowaliśmy się na używanie sieci z pięcioma neuronami ukrytymi. Powodem tego jest mała poprawa w stosunku do większych ilości neuronów oraz chęć zmniejszenia nakładu obliczeń. Dodatkowo sieci o zbyt dużej ilości neuronów ukrytych mają tendencję do przetrenowywania się, w wyniku którego sieć przystosowuje się nie tyle do procesu co do samych danych uczących.

Liczba neuronów	Błąd uczenia	Błąd weryfikacji
1	3.070626e+01	5.548815e+01
2	4.977413e-01	1.060318e+00
3	3.206039e-01	5.111444e-01
4	1.479096e-01	2.625729e-01
5	8.734595e-02	1.534512e-01
6	7.765994e-02	2.087909e-01
7	2.614618e-02	1.727668e-01
8	1.509561e-02	1.095385e-01
9	1.355132e-02	6.725641e-02
10	2.105601e-02	1.136051e-01

Tab. 2.1. Błędy modelu dla różnej ilości neuronów

### 2.3. Model z algorytmu BFGS

Na wykresie 2.2 przedstawione zostały błędy predyktorów ARX i OE dla kolejnych iteracji uczenia modelu. Zgodnie z ustaleniami z poprzednich punktów zastosowane zostały następujące parametry:  $\tau = 3$ ,  $\text{neurony ukryte} = 5$ . Końcowe błędy dla obydwu predyktorów wynosiły odpowiednio:  $E_{oe} = 0.0787$  i  $E_{arx} = 0.0212$ . Jak widać błędy te są dosyć małe jak na 2000 próbek co oznacza, że sieć dobrze nauczyła się modelu.



Rys. 2.2. Zmiany błędów predyktora ARX i OE dla kolejnych iteracji uczenia modelu algorytmem BFGS z użyciem rekurencji



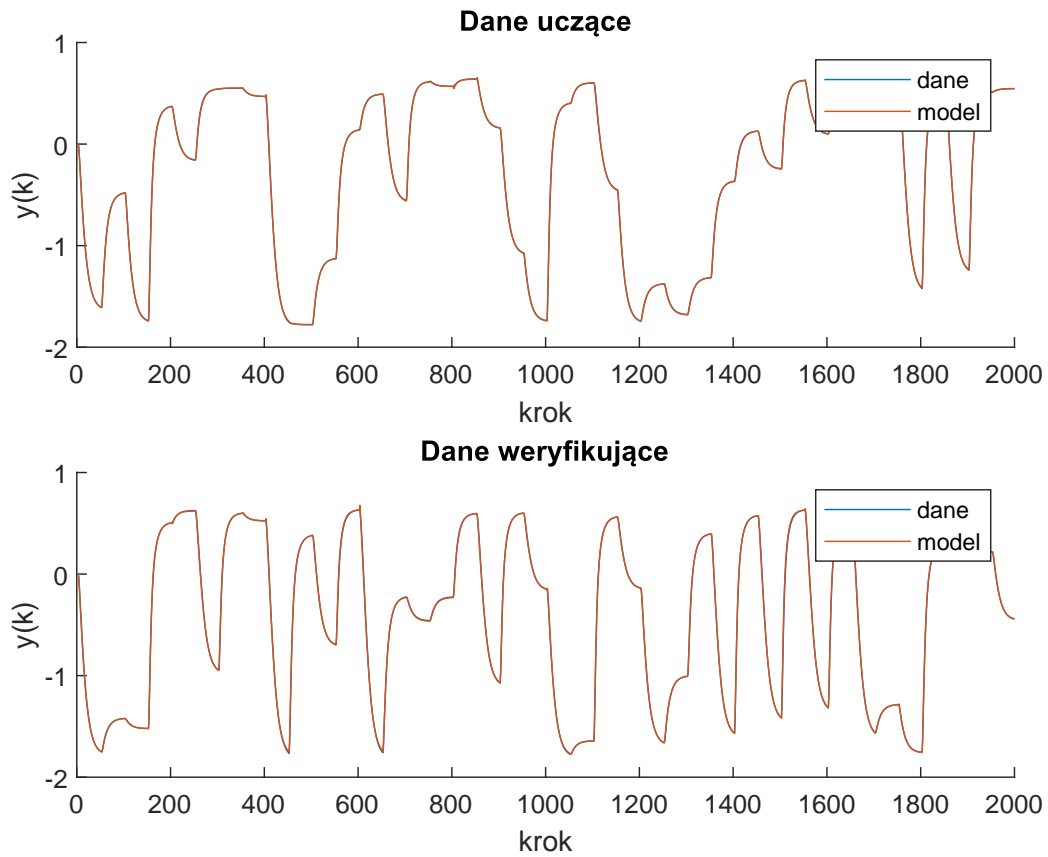
## 2.4. Symulacja modelu z algorytmu BFGS

Model z poprzedniego punktu został zasymulowany w trybie rekurencyjnym dla uczącego oraz weryfikującego zbioru danych. Błędy dla obydwu zbiorów danych wyniosły odpowiednio:  $E_{ucz} = 0.0787$  oraz  $E_{wer} = 0.2172$ . Błędy otrzymywane były ze wzoru:

$$E = (y(S : end) - y^M(S : end))' * (y(S : end) - y^M(S : end)) \quad (2.1)$$

gdzie  $S = \max(n_A, n_B) + 1$ , oraz  $n_A$  i  $n_B$  są współczynniki modelu opisanego wzorem

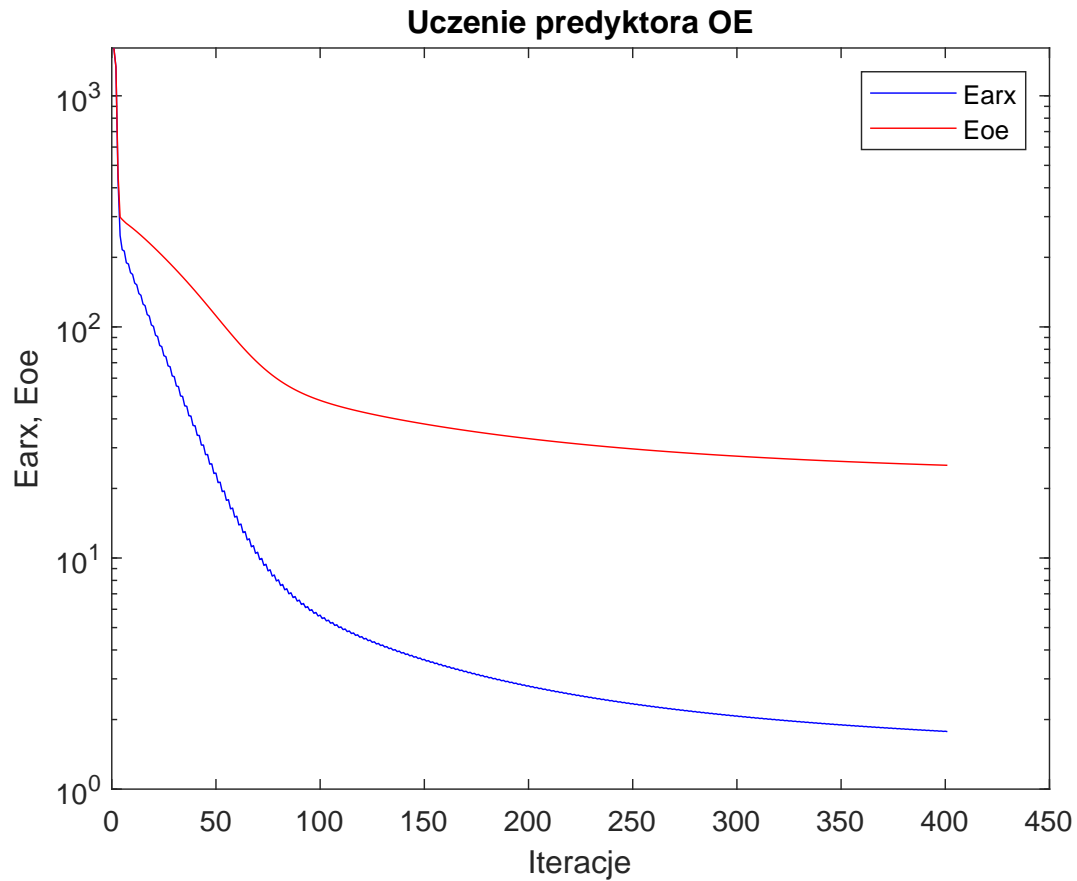
$$\hat{y}(k) = f(u(k - \tau), \dots, u(k - n_B), y(k - 1), \dots, y(k - n_A)) \quad (2.2)$$



Rys. 2.3. Symulacja modelu uczonego algorytmem BFGS z rekurencją na danych uczących i weryfikujących

## 2.5. Model z algorytmu najszybszego spadku

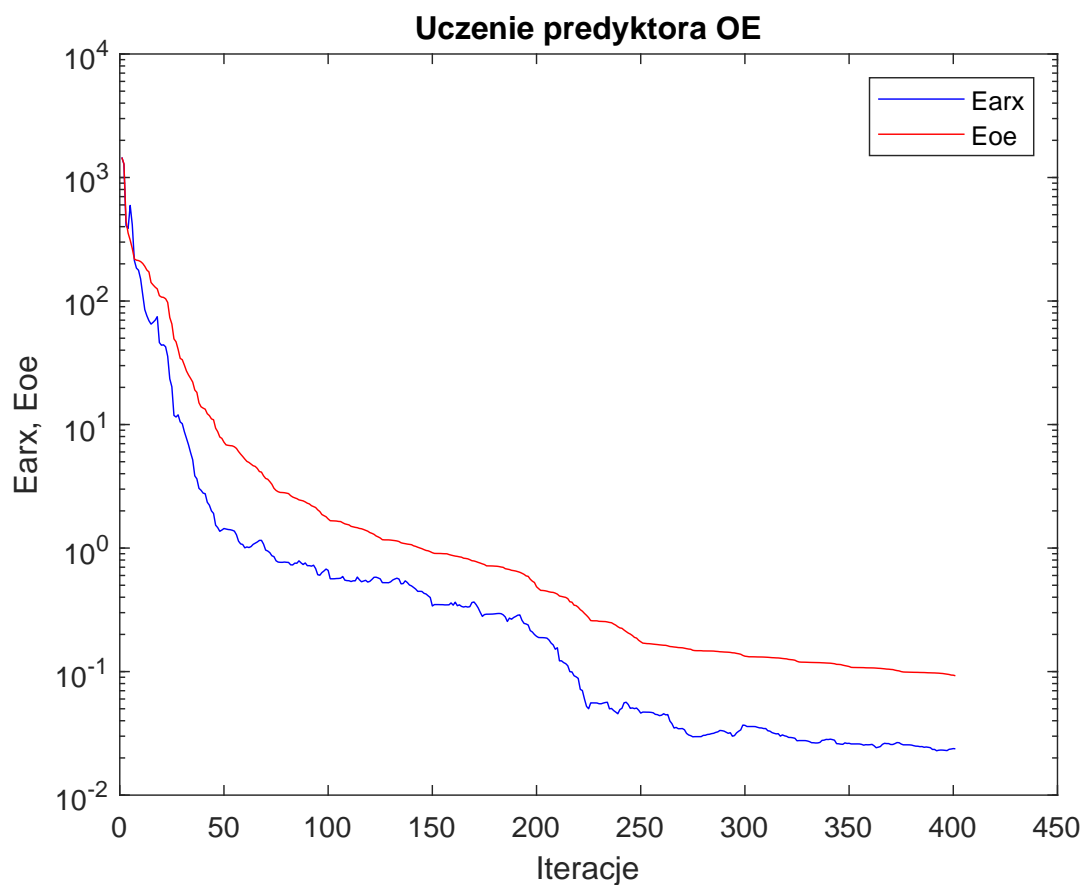
Jak widać po poniższych wartościach błędów uczenia oraz z wykresu 2.4 algorytm najszybszego spadku nie radzi sobie najlepiej z uczeniem sieci co skutkuje dużą niedokładnością modelu. Końcowe błędy dla obydwu predyktorów wyniosły  $Eoe = 25.1864$  oraz  $Earx = 1.7730$ , co w obydwu przypadkach wynosi więcej niż przy uczeniu sieci metodą BFGS.



Rys. 2.4. Zmiany błędów predyktora ARX i OE dla kolejnych iteracji uczenia modelu algorytmem najszybszego spadku z użyciem rekurencji

## 2.6. Model z algorytmu BFGS z uczeniem bez rekurencji

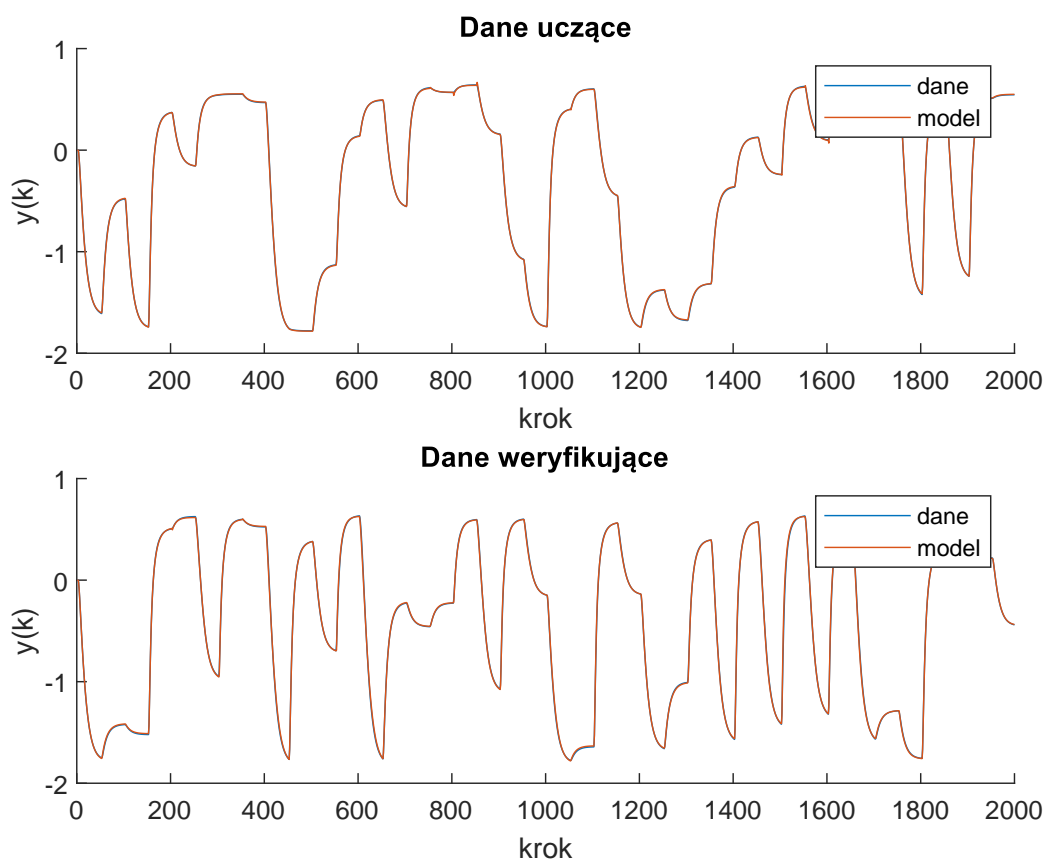
Na wykresie 2.5 można zobaczyć, że algorytm BFGS w wersji bez rekurencji radzi sobie niewiele gorzej niż algorytm BFGS w wersji z rekurencją. Po kilku próbach uczenia sieci udało się osiągnąć błędy uczenia o wartościach  $E_{oe} = 0.0925$  i  $E_{arx} = 0.0237$ , co nadal jest bardzo dobrym wynikiem.



Rys. 2.5. Zmiany błędów predyktora ARX i OE dla kolejnych iteracji uczenia modelu algorytmem BFGS bez użycia rekurencji

## 2.7. Symulacja modelu z algorytmu BFGS z uczeniem bez rekurencji

Podobnie jak w przypadku modelu nauczonego algorytmem z rekurencją, model z poprzedniego punktu został zasymulowany w trybie rekurencyjnym dla uczącego oraz weryfikującego zbioru danych. Błędy dla obydwu zbiorów danych wyniosły odpowiednio:  $E_{uc} = 0.0925$  oraz  $E_{wv} = 0.3542$ . Mimo niewielkiej różnicy model ten sprawuje się niestety gorzej niż w wersji z rekurencją, dlatego to na tym będziemy wykonywać następne zadania. Jest to logiczne, gdyż model symulowany jest w trybie rekurencyjnym, a uczony był bez niej, więc powinien być gorszy niż model uczony z rekurencją.



Rys. 2.6. Symulacja modelu uczonego algorytmem BFGS bez rekurencji na danych uczących i weryfikujących

## 2.8. Model metodą najmniejszych kwadratów

Uczenie modelu metodą najmniejszych kwadratów realizowane jest za pomocą obecnej w Matlabie operacji lewego dzielenia:

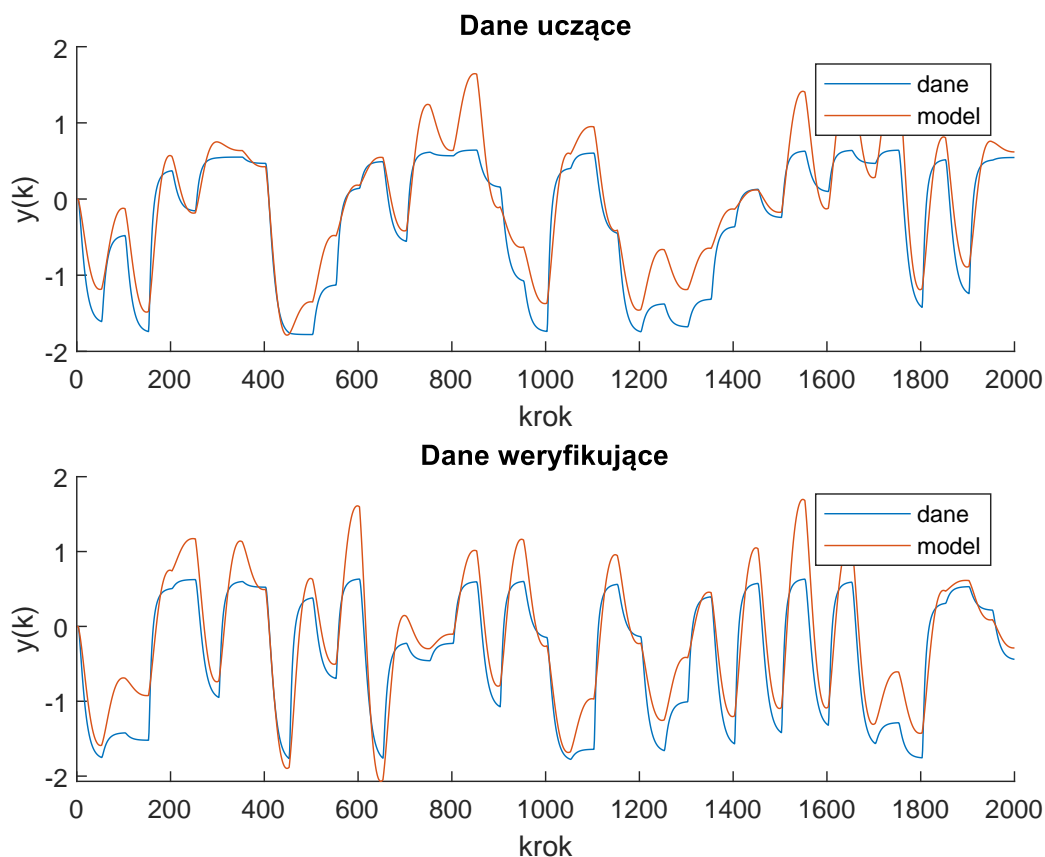
$$w = \begin{bmatrix} b_3 \\ b_4 \\ a_1 \\ a_2 \end{bmatrix} = M \backslash y_{ucz}(5 : end) \quad (2.3)$$

gdzie

$$M = \begin{bmatrix} x_{ucz}(2) & x_{ucz}(1) & y_{ucz}(4) & y_{ucz}(3) \\ \vdots & \vdots & \vdots & \vdots \\ x_{ucz}(n - \tau) & x_{ucz}(n - \tau - 1) & y_{ucz}(n - 1) & y_{ucz}(n - 2) \end{bmatrix} \quad (2.4)$$

oraz  $n$  = liczba próbek.

Otrzymane błędy były równe:  $E_{wer} = 294.6949$  oraz  $E_{ucz} = 299.7408$ . Jak widać jakość tego modelu jest bardzo zła i błędy są większe, niż w przypadku modelu uczonego algorytmem BFGS lub najszybszego spadku. Przebiegi przedstawiające działanie modelu dla danych uczących i weryfikujących pokazane są poniżej. Do wyznaczenia modelu oraz wyrysowania wykresu użyty został skrypt *mnk.m*.



Rys. 2.7. Symulacja modelu wykonanego za pomocą metody najmniejszych kwadratów

## 3. Regulacja procesu

### 3.1. Implementacja NPL

NPL jest algorytmem regulacji predykcyjnym z nieliniową predykcją i z linearyzacją oznacza to że do wyznaczania trajektorii swobodnej (która zależy tylko od przeszłych sterowań) używamy nieliniowego modelu neuronowego:

$$y^0(k+p|k) = w_{20} + w_2 * \tanh(w_{10} + w_1 * x(k+p)) + dk \quad (3.1)$$

gdzie

$$dk = y(k) - y^M(k) \quad (3.2)$$

$$x(k) = \begin{bmatrix} u(\min(k - \tau + p, k - 1)) \\ u(\min(k - \tau - 1 + p, k - 1)) \\ y(k - 1 + p) \\ y(k - 2 + p) \end{bmatrix} \quad (3.3)$$

$p$  = ilość chwil w przyszłość. Warto dodać, że we wzorze 3.3 dla chwil czasu dalszych od  $k$  zakłada się że  $y(k+p) = y^0(k+p|k)$ .

Aby móc rozwiązać algorytm analitycznie dokonuje się linearyzacji wyjścia modelu. Współczynniki  $b_3, b_4, a_1, a_2$  we wzorze

$$y(k) = b_3 u(k - \tau) + b_4 (k - \tau - 1) - a_1 y(k - 1) - a_2 y(k - 2) \quad (3.4)$$

otrzymuje się poprzez obliczenie pochodnej cząstkowej po odpowiednim wejściu modelu neuronowego. Mając obliczone współczynniki można użyć ich do wyznaczenia odpowiedzi skokowej ze wzoru

$$s_j(k) = \sum_{i=1}^{\min(j, n_B)} b_i(k) - \sum_{i=1}^{\min(j-1, n_A)} s_{j-i}(k) \quad (3.5)$$

których można użyć do wypełnienia macierzy dynamicznej  $M$ .

$$M = \begin{bmatrix} s_1 & 0 & \dots & 0 \\ s_2 & s_1 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ s_N & s_{N-1} & \dots & s_{N-N_u+1} \end{bmatrix}_{N \times N_u} \quad (3.6)$$

Na koniec można obliczyć optymalne przyszłe sterowania (przy zadanych horyzontach predykcji  $N$  i sterowania  $N_u$  oraz współczynnika kary  $\lambda$ ).

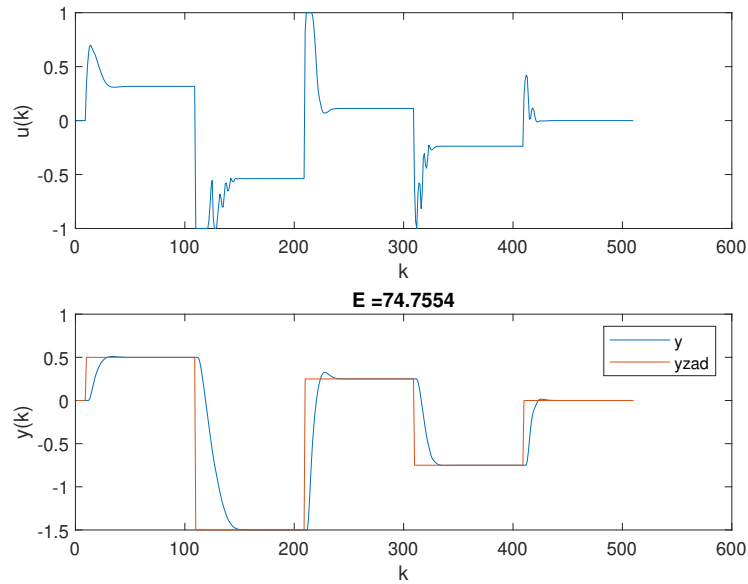
$$dU = K * (Y_{zad}(k) - Y^0) \quad (3.7)$$

gdzie  $Y_{zad}(k)$  to wektor długości  $N$  zawierający aktualną wartość zadaną,  $Y^0$  to wektor  $N$  przyszłych, predykowanych wartości wyjścia oraz

$$K = (M' * M + \lambda I)^{-1} * M' \quad (3.8)$$

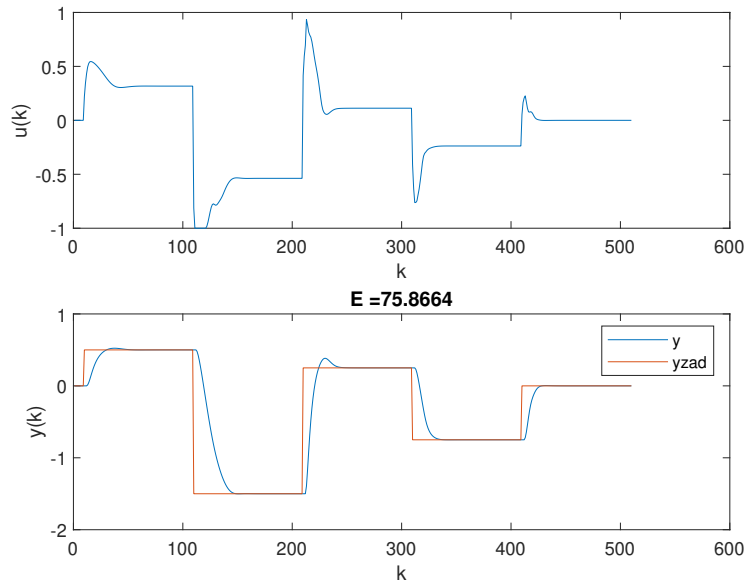
### 3.2. Strojenie NPL

Regulator NPL został nastrojony z użyciem sieci neuronowej wytrenowanej algorytmem BFGS z użyciem rekurencji opisanej w sekcji 2.3. Strojenie regulatora NPL rozpoczęliśmy od parametrów  $N = 15$ ,  $N_u = 2$  oraz  $\lambda = 1$ . Przebieg dla tych wartości zaprezentowany jest poniżej (rys. 3.1). Jak widać już od samego początku przebieg regulacji nie jest zły, aczkolwiek skoki sterowania są zdecydowanie zbyt ostre. Błąd o wartości 74.554 na 600 próbkach nie jest idealny, ale akceptowalny.



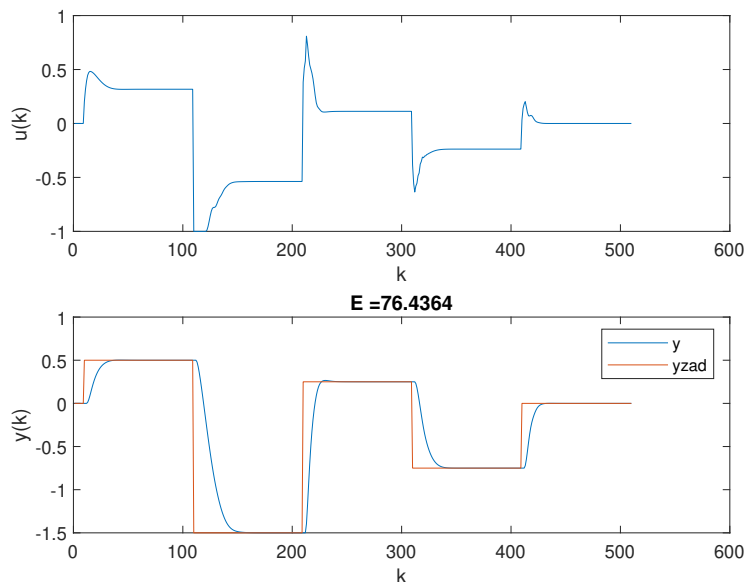
Rys. 3.1. Działanie regulatora NPL z nastawami  $N=15$ ,  $N_u=2$ ,  $\lambda=1$

W celu zmniejszenia gwałtowności sterowania postanowiliśmy zwiększać wartość  $\lambda$  do czasu od sterowanie zładownieje. Należy oczywiście pamiętać, że im większy jest parametr  $\lambda$ , tym regulator będzie wolniejszy. Kompromis pomiędzy szybkością (błędem), a kształtem sygnału sterującego osiągnęliśmy dla  $\lambda = 4$ , błąd wynosił 75.8664. Przebieg zaprezentowany jest na rys. 3.2



Rys. 3.2. Działanie regulatora NPL z nastawami  $N=15$ ,  $N_u=2$ ,  $\lambda=4$

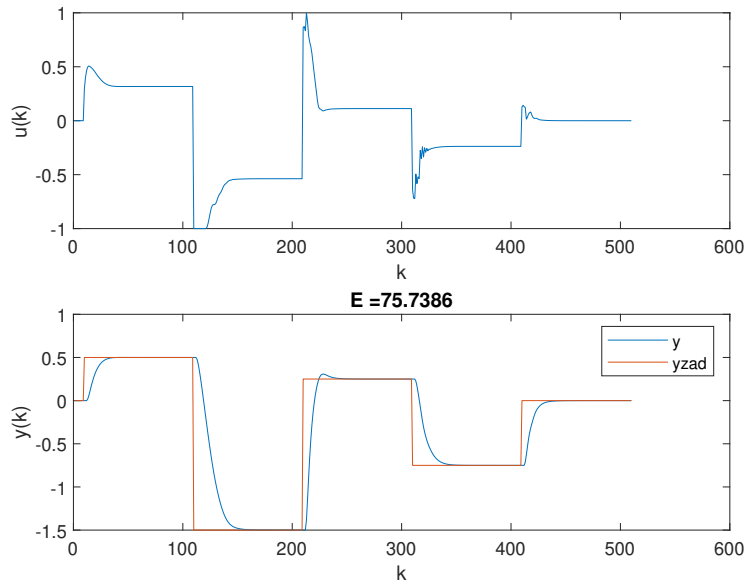
Teraz gdy sygnał sterowania jest już łagodniejszy postanowiliśmy zbadać wpływ horyzntu predykcji na jakość regulacji. Zauważyliśmy, że zarówno przy zmniejszaniu, jak i przy zwiększaniu wartości  $N$ , błąd rośnie, lecz dla dalszych horyzntów maleje przeregulowanie. Raz jeszcze postanowiliśmy znaleźć kompromis pomiędzy błędem, a przeregulowaniem. Sytuacją taką udało się osiągnąć dla  $N = 19$ . Błąd wynosił 76.4364, natomiast przeregulowania było prawie niewidoczne. Przebieg ten można zobaczyć na rys. 3.3



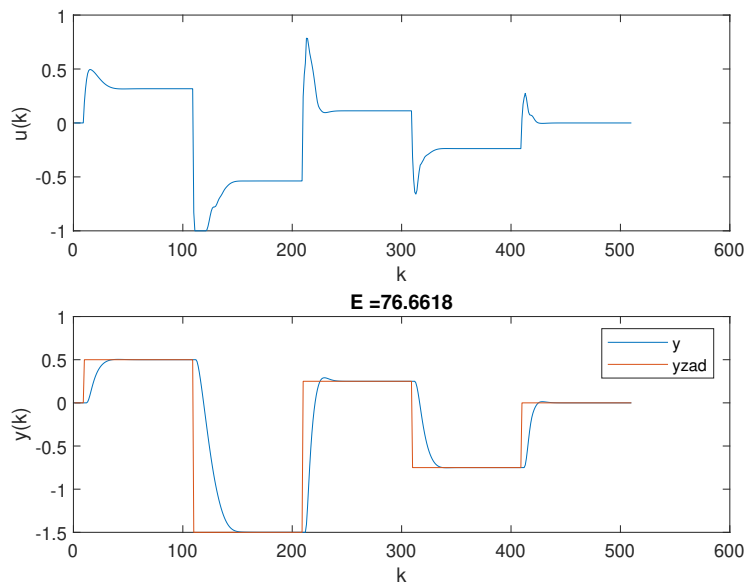
Rys. 3.3. Działanie regulatora NPL z nastawami  $N=19$ ,  $N_u=2$ ,  $\lambda=4$



Następnie postanowiliśmy dobrać horyzont sterowania. Niestety zarówno przy zwiększaniu jak i zmniejszaniu horyzontu jakość regulacji pogarszała się, co można zaobserwować na wykresach 3.4 i 3.5. Dla  $N_u$  równego 1 błąd regulacji co prawda spadł, ale następują niekontrolowane, ostre skoki sterowania oraz znów pojawiły się przeregulowania. Dla  $N_u$  równego 3 wejście prezentuje się podobnie, natomiast ucierpiało wyjście.



Rys. 3.4. Działanie regulatora NPL z nastawami  $N=19$ ,  $N_u=1$ ,  $\lambda=4$



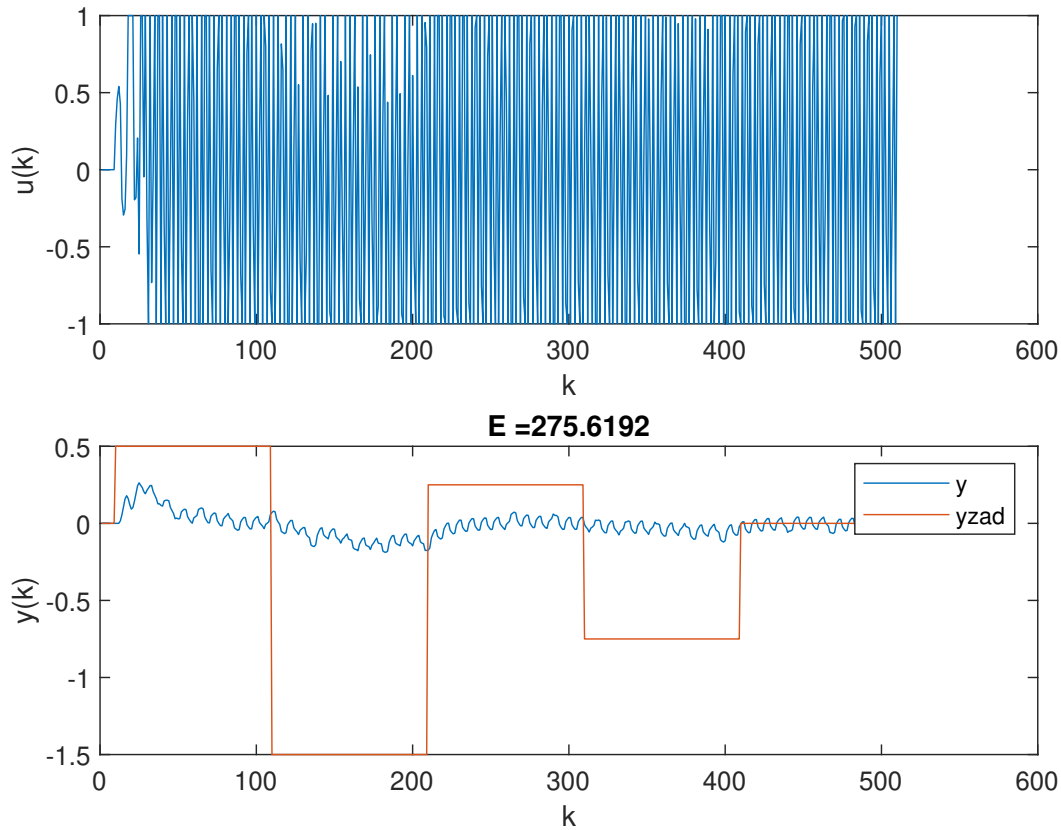
Rys. 3.5. Działanie regulatora NPL z nastawami  $N=19$ ,  $N_u=3$ ,  $\lambda=4$

### 3.3. GPC

Algorytm regulacji GPC, różni się tym od NPL, że na całym horyzoncie predykcji korzysta się z liniowego modelu wyznaczonego metodą najmniejszych kwadratów. Jak można było zauważyć z rys. 2.7 taki model nie gwarantuje najlepszego odwzorowania obiektu, przez co jak można się domyślać jakość regulacji również może być gorsza. Do wyznaczania sterowania w wersji analitycznej wyznacza predykcje wyjścia modelu  $N$  chwil do przodu ze wzoru

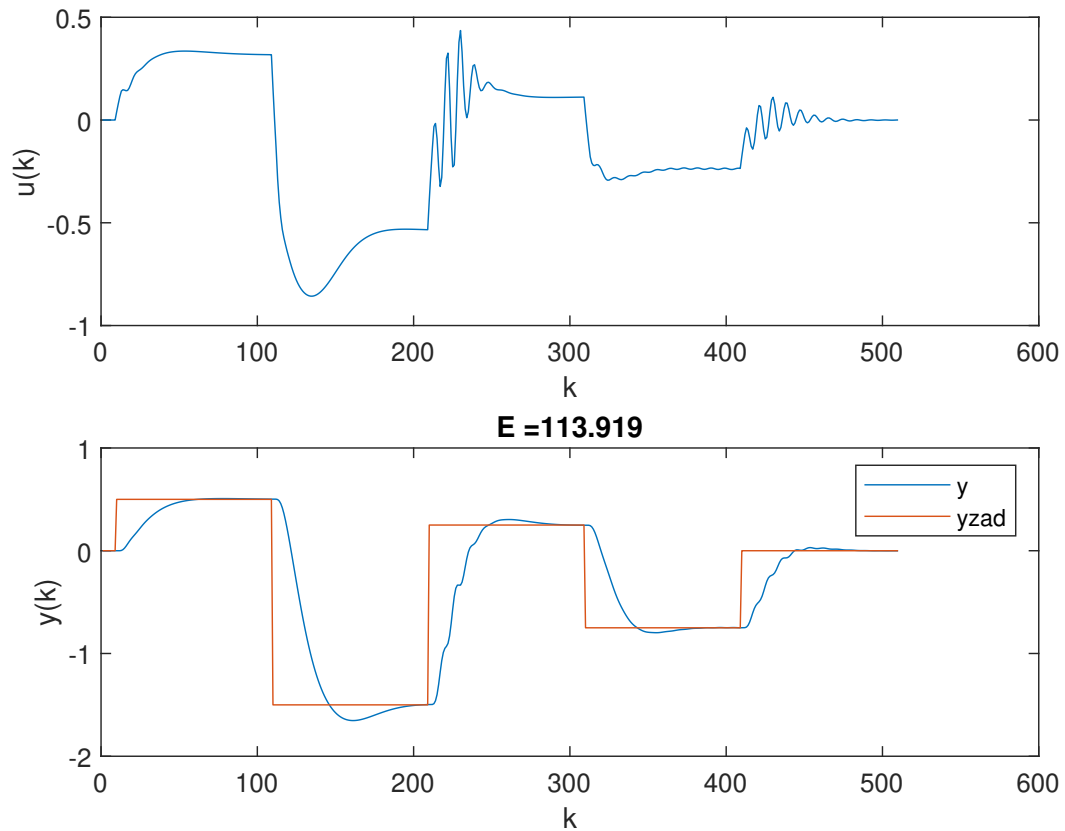
$$y^0(k+p|k) = b_3 u(\min(k-3+p, k-1)) + b_4 u(\min(k-4+p, k-1)) - a_1 y(k-1+p) - a_2 y(k-2+p) \quad (3.9)$$

oraz analogicznie do wzoru 3.1  $y(k+p) = y^0(k+p|k)$ . Parametry  $a_i$  oraz  $b_i$  otrzymywane są z metody najmniejszych kwadratów. Macierz dynamiczna jest stała i wyznaczana przy użyciu odpowiedzi skokowej ze wzoru 3.5. Na wykresie poniżej można zauważyć, że jakość regulacji w istocie pozostawia wiele do życzenia (rys 3.6).



Rys. 3.6. Działanie regulatora GPC z nastawami  $N=19$ ,  $N_u=2$ ,  $\lambda=4$

Należy wziąć pod uwagę, że przez silną nieliniowość obiektu, liniowy algorytm GPC może generować duże sterowania, które po nałożeniu ograniczeń wprowadzą obiekt w stałe oscylacje. Można temu zapobiec poprzez zwiększenie współczynnika  $\lambda$  o parę rzędów wielkości. Na rys. 3.7 można zobaczyć, że jakość regulacji polepszyła się, lecz mimo to sterowanie wciąż jest zbyt ostre, a czas regulacji wolniejszy niż w przypadku NPL. W dodatku zarówno dla sterowania jak i wyjścia występują widoczne oscylacje.

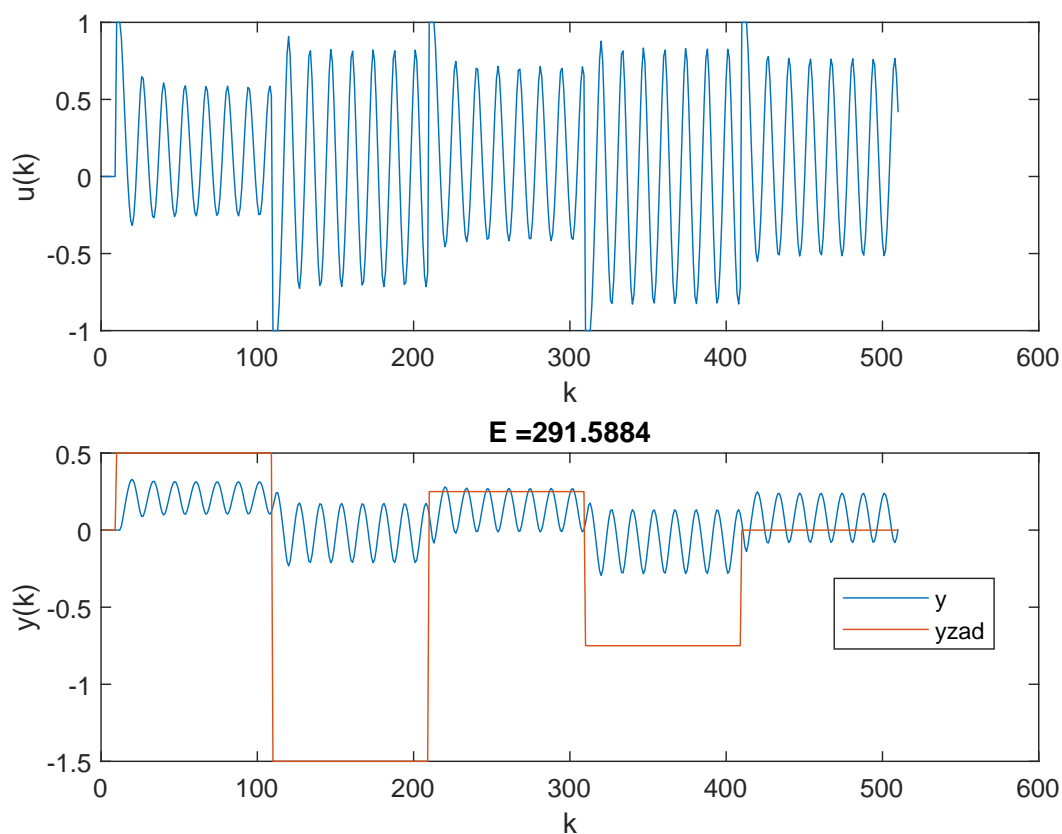


Rys. 3.7. Działanie regulatora GPC z nastawami  $N=19$ ,  $N_u=2$ ,  $\lambda=100$

## 4. Zadania dodatkowe

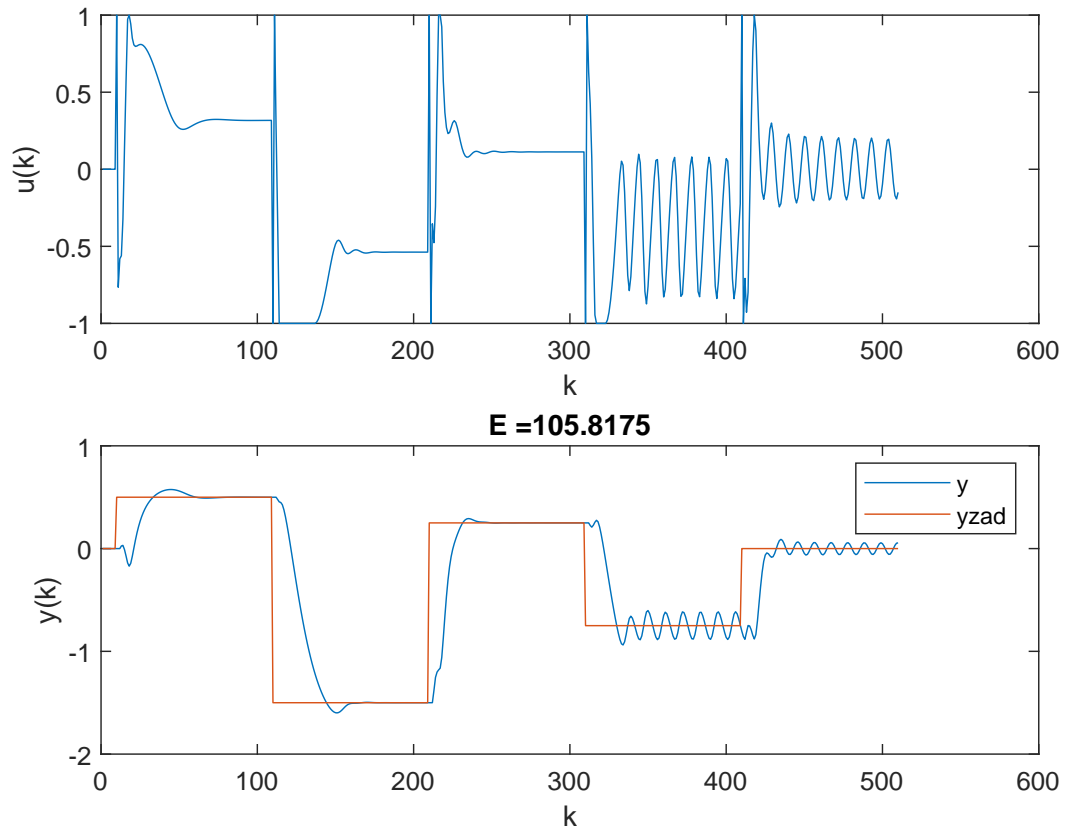
### 4.1. PID

Algorytm PID oblicze przyszłe sterowanie na podstawie wartości, pochodnej i całki uchybu w odpowiednich proporcjach. Popularną sposobem strojenia tego regulatora jest metoda Zieglera-Nicholsa, która polega na doprowadzenie obiektu na granice stabilności przy wyłączonych członach I oraz D, zmierzenia okresu drgań a następnie podstawieniu odpowiednich wartości do wzoru. W przypadku obiektu z zadania obiekt był na granicy stabilności (w we wszystkich zadanych przez nas punktach pracy) przy  $K_p = 4$ , co można zaobserwować na rys. 4.1.



Rys. 4.1. Działanie regulatora PID z nastawami  $K_p = 4$ ,  $T_i = \text{Inf}$ ,  $T_d = 0$

Następnie, po podstawieniu zmierzonych wartości (okres drgań  $T_u = 13$ ) otrzymaliśmy przebieg przedstawiony na rys. 4.2. Widać, że regulator próbuje naśladować przebieg wartości zadanej, i robi to nie najgorzej (mimo bardzo ostrego sterowania), lecz dla niektórych punktów pracy pojawiają się niegasnące oscylacje. Nastrojony w ten sposób regulator wydaje się dobrze radzić sobie w bliskości granicy przedziału sterowania, a gorzej będąc w jego centrum.



Rys. 4.2. Działanie regulatora PID z nastawami  $K_p = 2.4$ ,  $T_i = 6.5$ ,  $T_d = 1.625$

## 4.2. NO

Algorytm NO, tym różni się od algorytmu NPL, że do wyznaczania predykcji wyjścia stosuje się model nieliniowy. Oznacza to, że nie można wyznaczyć przyszłych sterowań analitycznie. Posługując się wskaźnikiem jakości

$$J(k) = \sum_{p=1}^N (y^{zad}(k) - \hat{y}(k+p|k))^2 + \lambda \sum_{p=0}^{N_u} (\Delta u(k+p|k))^2 \quad (4.1)$$

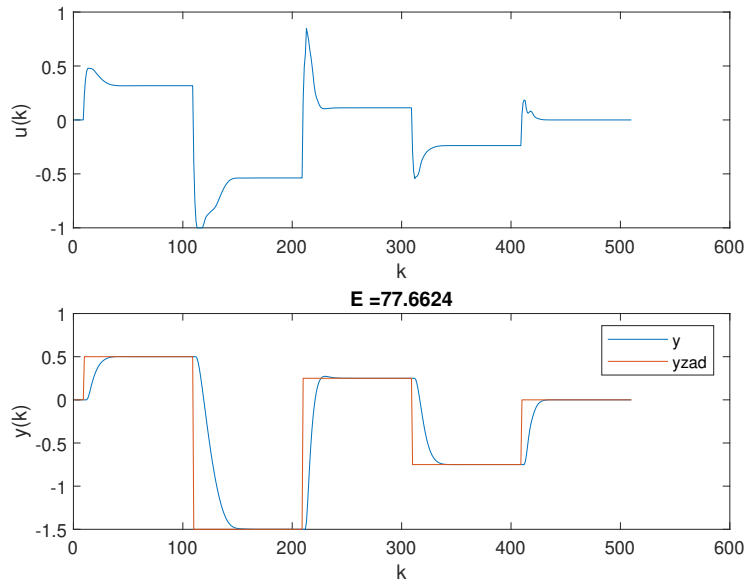
wyznacza się takie sterowania dla których jest on najmniejszy. Wyliczyjąc predykcje wyjścia jako

$$\hat{y}(k+p|k) = w_{20} + w_2 * \tanh(w_{10} + w_1 * x(k+p|k)) + dk \quad (4.2)$$

gdzie

$$x(k+p|k) = \begin{bmatrix} u(k-3+p) \\ u(k-4+p) \\ y(k-1+p) \\ y(k-2+p) \end{bmatrix} \quad (4.3)$$

We wzorze tym, podobnie jak w równaniu 3.1 i 3.9 dla  $y(k+p|k) = \hat{y}(k+p|k)$ . Dodatkowo, ponieważ przewidujemy jedynie przez cały horyzont sterowania, zakłada się, że sterowanie dla  $p > N_u - 1$  przyjmuje wartość  $u(k+p) = u(k+N_u-1)$ . Mając wyznaczone wszystkie wartości można obliczyć zadanie optymalizacji. W tym celu wykorzystaliśmy obecny w Matlabie algorytm *fmincon*, a optymalizowaną przez nas zmienną było  $N_u$  przyszłych sterowań licząc z aktualnym. Regulator NO był testowany z użyciem sieci neuronowej wytrenowanej algorytmem BFGS z użyciem rekurencji opisanej w sekcji 2.3. Wyniki działania algorytmu NO przedstawione są na rysunku 4.3. Jak widać zarówno wyjście obiektu jak i sterowania wyglądają świetnie. Nie występują oscylacje ani przesterowania, a sterowanie jest dosyć łagodne. Niestety dużą wadą algorytmu NO jest, fakt że w każdym kroku algorytmu należy rozwiązać zadanie nieliniowej optymalizacji, co w przypadku obiektów o dłuższych horyzontach predykcji potrafi prowadzić do bardzo długiego czasu wyznaczania sterowań.



Rys. 4.3. Działanie regulatora NO z nastawami  $N=19$ ,  $N_u=2$ ,  $\lambda=4$

## 5. Używane skrypty

Do wykonania powyższych zadań wykorzystane zostały następujące skrypty:

- Charakterystyka statyczna - *charakterystyka\_statyczna.m*
- Dane uczące i weryfikujące - *generowanie\_danych.m* oraz *wykres\_danych.m*
- Dobór liczby neuronów - *modelowanie.m* wraz z generowanym plikiem *osiagi.txt*
- Trenowanie różnych modeli neuronowych - *naucz\_model.m*
- Metoda najmniejszych kwadratów - *mnk.m*
- Testowanie algorytmów regulacji - *regulacja.m*, przy czym używane przez niego funkcje poszczególnych algorytmów to: *funregnpl.m*, *funreggpc.m* oraz *funregno.m*.

**Inne załączone pliki:**

- *daneucz.mat* i *danewer.mat* - wykorzystywane dane uczące i weryfikujące
- *g\_1.m* i *g\_2.m* - funkcje procesu
- *modelBFGS\_OE*, *modelBFGS\_ARX*, *modelNS\_OE*, *uczenieBFGS\_OE*, *uczenieBFGS\_ARX*, *uczenieNS\_OE* - nauczone sieci neuronowe prezentowane w sekcji drugiej
- *siec.m* - rekurencyjne liczenie wyjścia sieci neuronowej używane w ewaluowaniu modeli
- *snout.m* - liczenie wyjścia sieci neuronowej dla zadanego wejścia, używane w funkcjach regulacji