Wydział Elektroniki i Technik Informacyjnych
Politechnika Warszawska

Modelowanie i identyfikacja

Sprawozdanie projekt 2 zadanie 13

Michał Pióro
324881

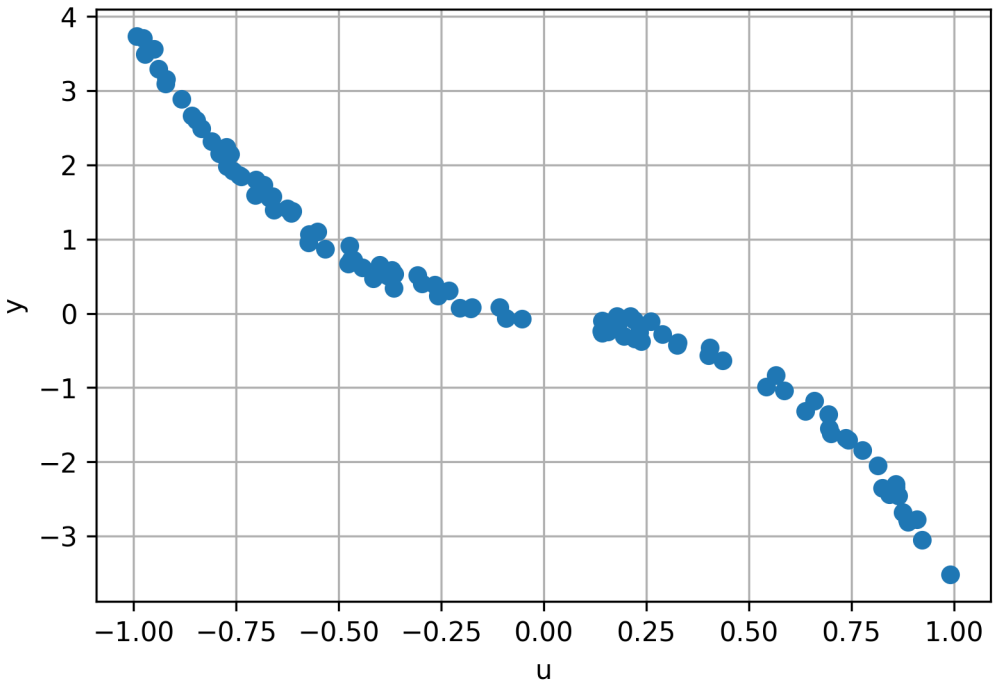
Warszawa, kwiecień 2024

Spis treści

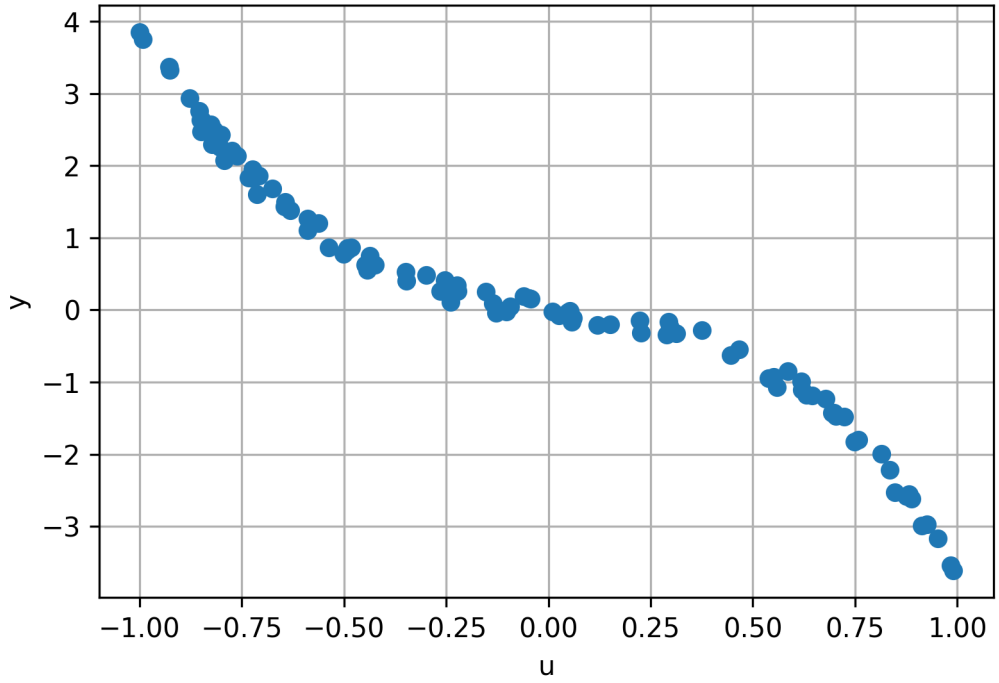
1. Identyfikacja modeli statycznych

1.1. Podział danych

W celu podzielenia danych na zbór uczący i weryfikujący, posortowano punkty (u, y) rosnąco względem u. Założono podział danych w stosunku 50 : 50. W celu jego osiągnięcia co drugi punkt licząc od pierwszego trafił do zbioru uczącego, a pozostałe punkty znalazły się w zbiorze weryfikującym. Posiadając zbiór 200 próbek utrzymano ilości $p_{ucz} = 100$ oraz $p_{wer} = 100$.



Rys. 1.1. Dane uczące modelu statycznego



Rys. 1.2. Dane weryfikujące modelu statycznego

1.2. Model liniowy

Postać matematyczna:

$$y(u) = a_0 + a_1 u \tag{1.1}$$

Implementacja w języku Python:

```
def lin_mod_stat_func(u, a):
    return a[0] + a[1] * u
```

1.2.1. Wyznaczenie parametrów

W celu wyznaczenia parametrów a_0 oraz a_1 wykorzystano metodę najmniejszych kwadratów.

$$\begin{bmatrix} a_0 \\ a_1 \end{bmatrix} = \begin{bmatrix} 1 & u_{ucz}(0) \\ 1 & u_{ucz}(1) \\ \vdots & \vdots \\ 1 & u_{ucz}(p_{ucz}) \end{bmatrix} \setminus \begin{bmatrix} y_{ucz}(0) \\ y_{ucz}(1) \\ \vdots \\ y_{ucz}(p_{ucz}) \end{bmatrix} \tag{1.2}$$

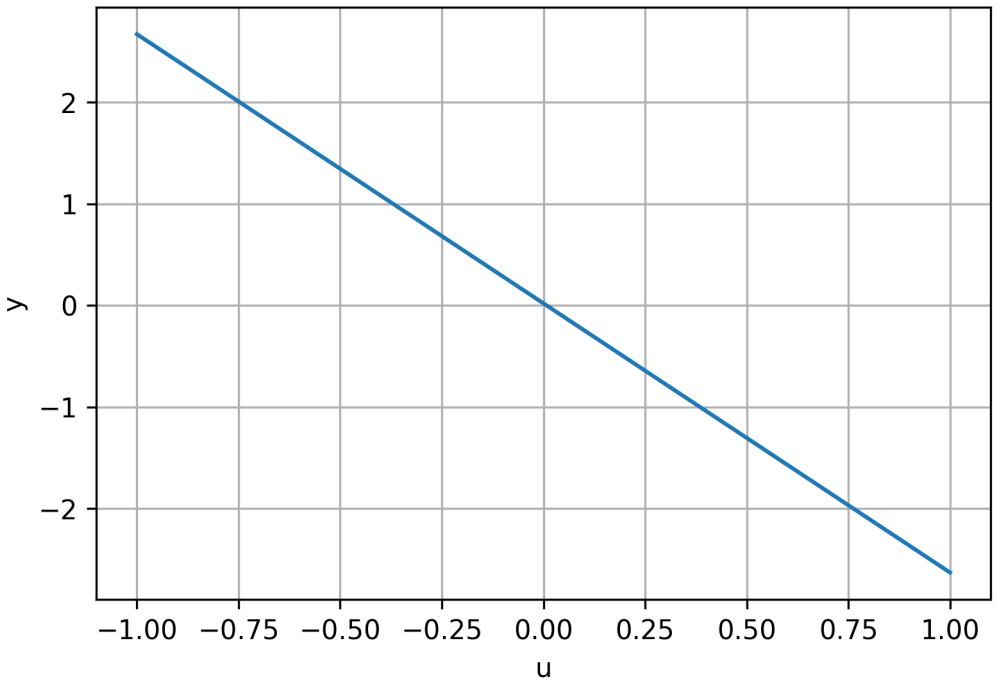
Ta metoda została zaimplementowana w autorskiej funkcji w języku Python z wykorzystaniem biblioteki NumPy.

Przyjmuje argumenty:

— u - wektor wartości wejściowych
— y - wektor wartości wyjściowych

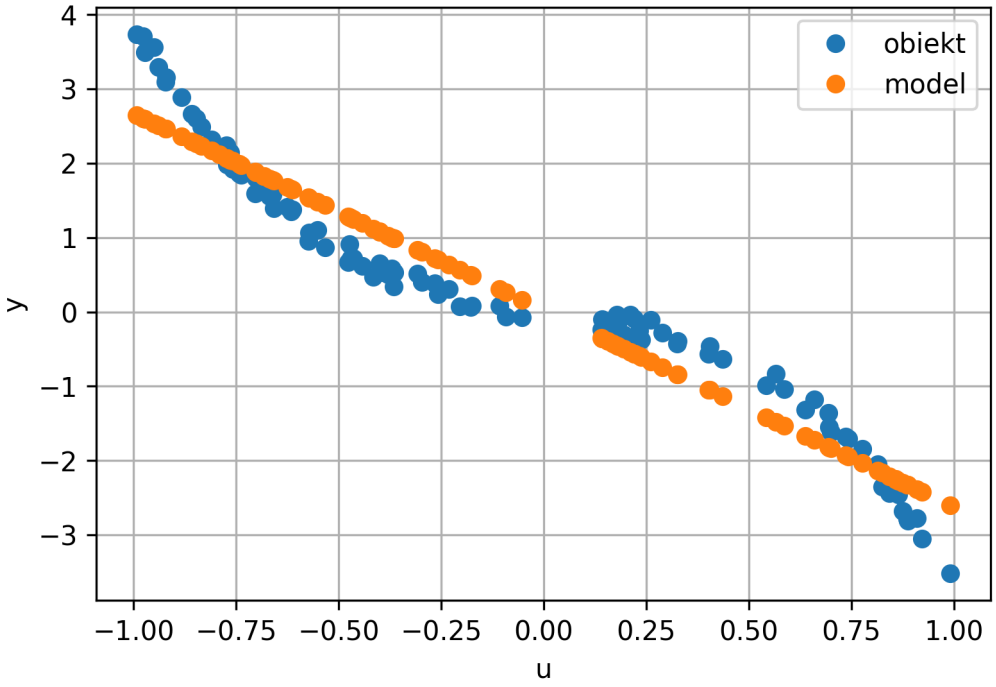
```
def nmk_lin_mod_stat(u, y):
    M = np.column_stack((np.ones(u.shape[0]), u[:, 0]))
    a = np.dot(np.dot(np.linalg.inv(np.dot(M.T, M)), M.T), y)
    return a
```

1.2.2. Charakterystyka statyczna modelu

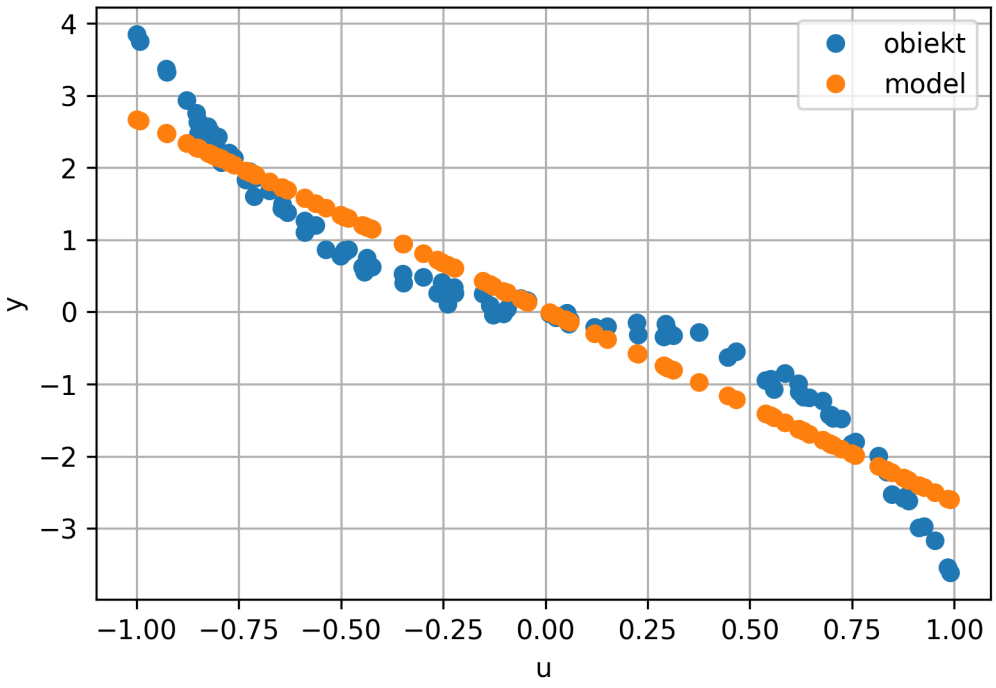


Rys. 1.3. Liniowa charakterystyka statyczna

1.2.3. Wykresy wyjść modelu na tle wyjść obiektu



Rys. 1.4. Wyjście modelu dla danych uczących na tle wyjścia obiektu



Rys. 1.5. Wyjście modelu dla danych weryfikujących na tle wyjścia obiektu

1.2.4. Błąd modelu

W celu dokładniejszego dokładniejszej analizy modelu obliczono jego błąd średniokwadratowy.

```
def model_error(y, y_mod):
    return np.sum((y[:, np.newaxis] - y_mod[:, np.newaxis]) ** 2) / y_mod.shape[0]

def model_stat_error(u, y, mod_func, a):
    y_mod = np.array([mod_func(x, a) for x in u]).T
    return model_error(y, y_mod)
```

Przedstawiony powyżej sposób obliczania błędu został wykorzystany również dla pozostałych modeli statycznych.

Dane uczące	Dane weryfikujące
0.2005	0.2011

Tab. 1.1. Błędy liniowego modelu statycznego

1.2.5. Omówienie wyników

Tak jak się spodziewano po postaci modelu, liniowa charakterystyka statyczna to po prostu prosta najbliżej zbliżona do danych. W badanym przypadku nie jest ona najlepszym wyborem, ponieważ charakterystyka obiektu jest zdecydowanie nieliniowa. Jednakże, gdyby zachodziła potrzeba przybliżenia poszczególnych odcinków charakterystyki statycznej(np. do modelu rozmytego), to w tym celu można by było zastosować dobraną, właśnie dla tych odcinków charakterystykę liniową.

1.3. Modele nieliniowe

Postać matematyczna:

$$y(u) = a_0 + \sum_{i=1}^{N} u^i \tag{1.3}$$

Implementacja w języku Python:

```
def nlin_mod_stat_func(u, a):
    val = a[0]
    N=a.shape[0]-1
    for i in range(1, N+1):
        val += a[i] * pow(u, i)
    return val
```

1.3.1. Wyznaczenie parametrów

W celu wyznaczenia wektora parametrów a wykorzystano metodę najmniejszych kwadratów.

$$\begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_N \end{bmatrix} = \begin{bmatrix} 1 & u_{ucz}(0) & \cdots & u_{ucz}^N(0) \\ 1 & u_{ucz}(1) & \cdots & u_{ucz}^N(1) \\ \vdots & \vdots & \ddots & \\ 1 & u_{ucz}(p_{ucz}) & \cdots & u_{ucz}^N(p_{ucz}) \end{bmatrix} \setminus \begin{bmatrix} y_{ucz}(0) \\ y_{ucz}(1) \\ \vdots \\ y_{ucz}(p_{ucz}) \end{bmatrix} \tag{1.4}$$

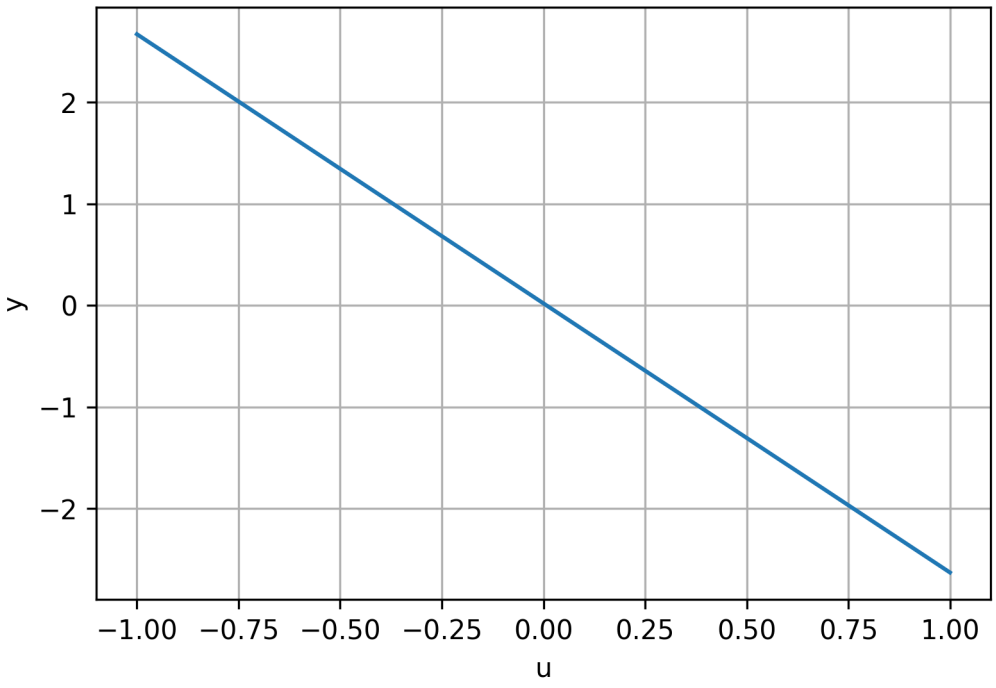
Ta metoda została zaimplementowana w autorskiej funkcji w języku Python z wykorzystaniem biblioteki NumPy.

Przyjmowane argumenty:

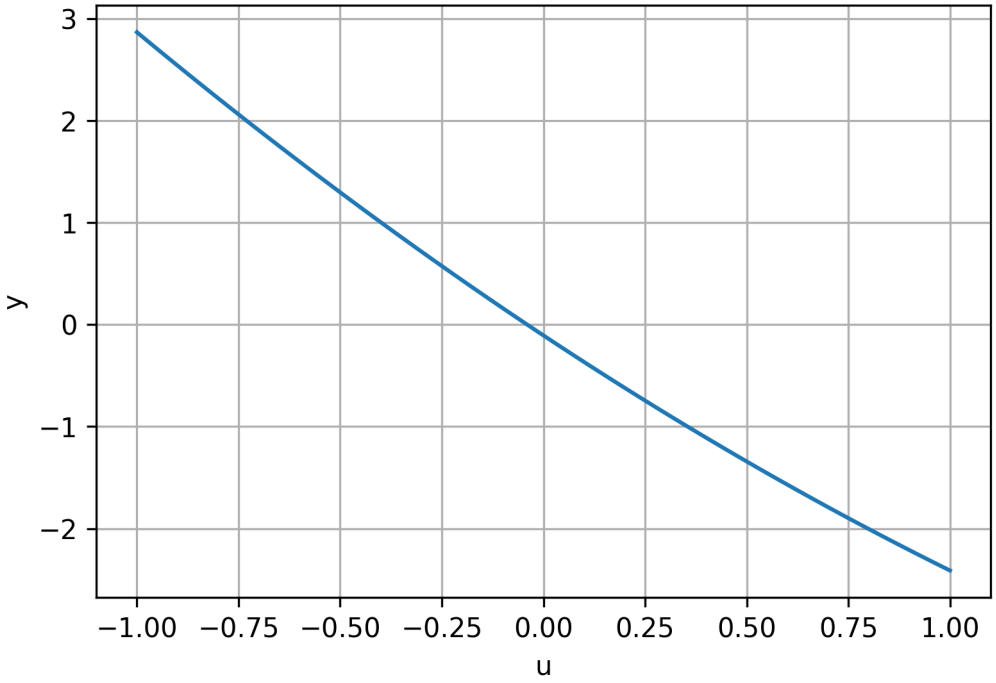
— u - wektor wartości wejściowych
— y - wektor wartości wyjściowych
— n - stopień wielomianu

```
def nmk_nlin_mod_stat(u, y, n):
    M = np.ones(u.shape[0])
    for i in range(1, n+1):
        M = np.column_stack((M, np.power(u[:, 0], i)))
        print(M)
    a = np.dot(np.dot(np.linalg.inv(np.dot(M.T, M)), M.T), y)
    return a
```

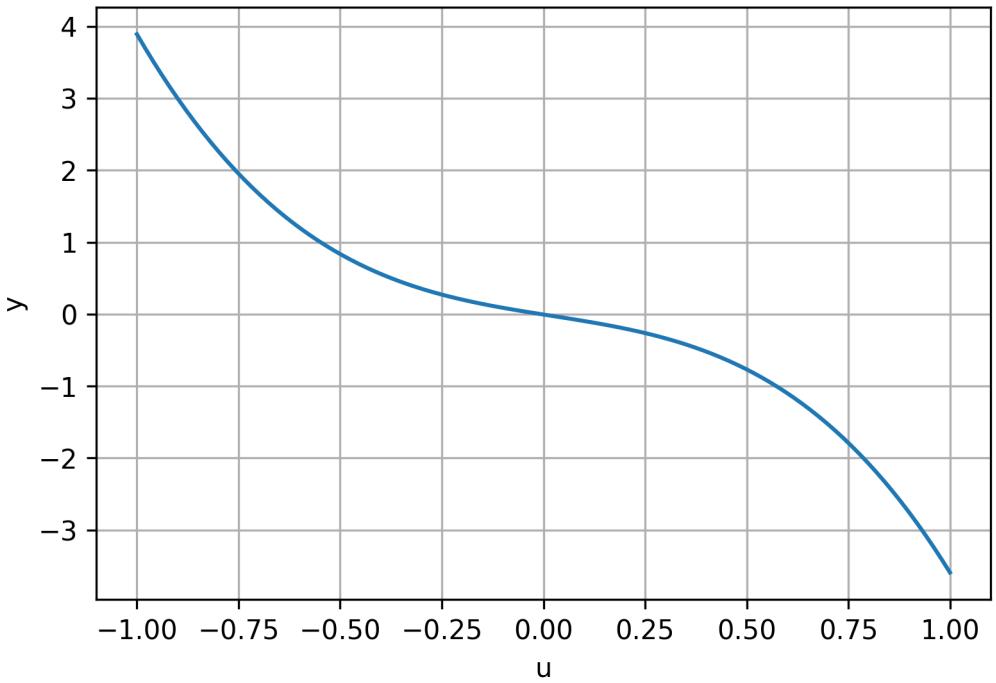
1.3.2. Charakterystyki statyczne modeli



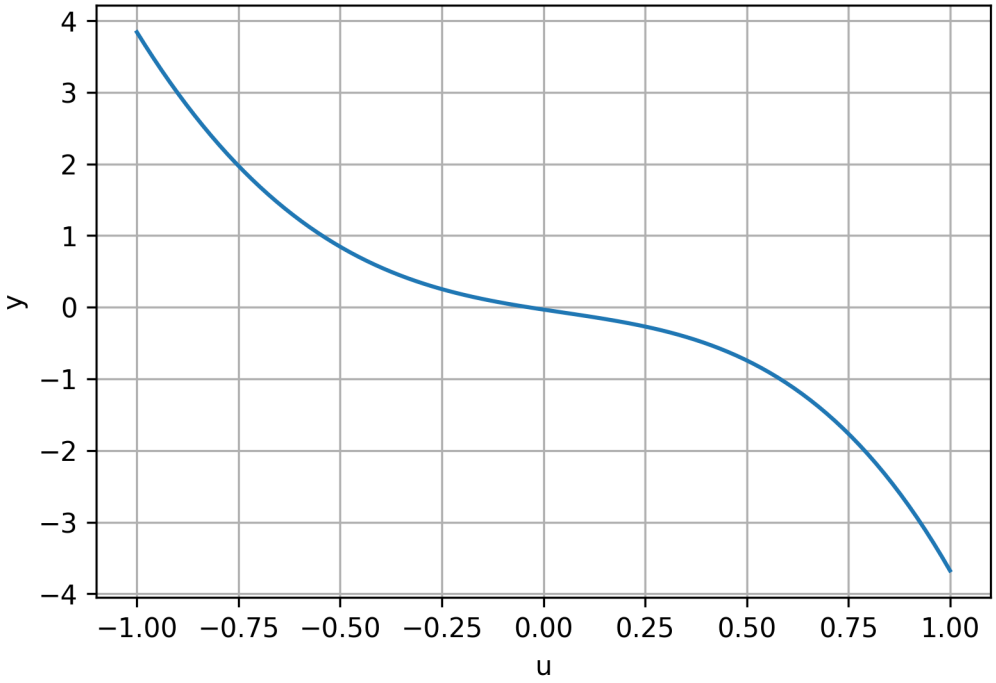
Rys. 1.6. Charakterystyka statyczna dla wielomianu o stopniu $N = 1$



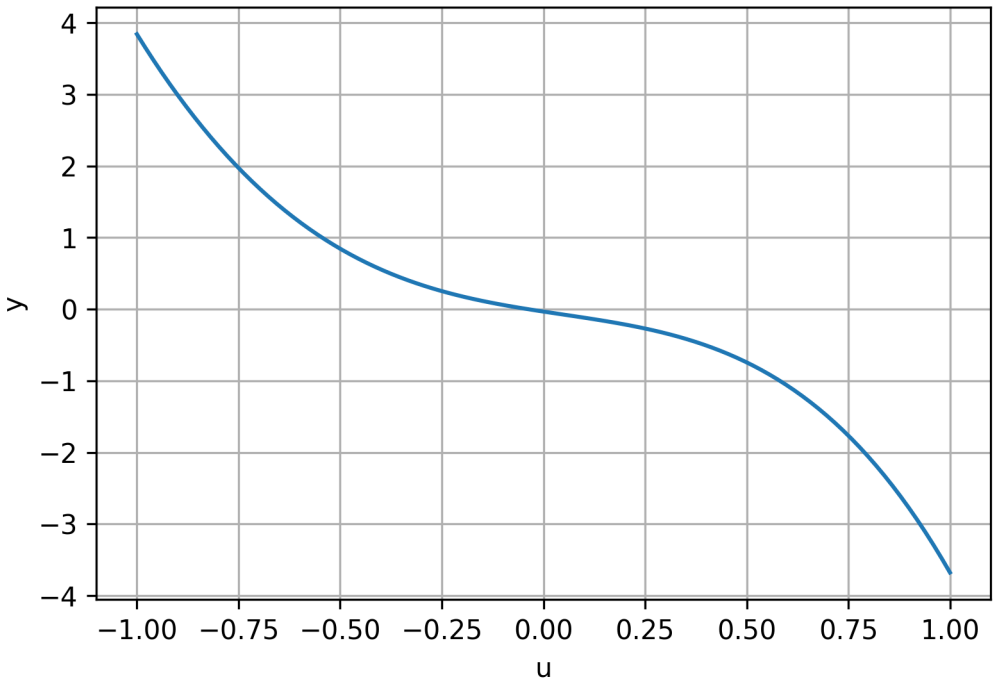
Rys. 1.7. Charakterystyka statyczna dla wielomianu o stopniu $N = 2$



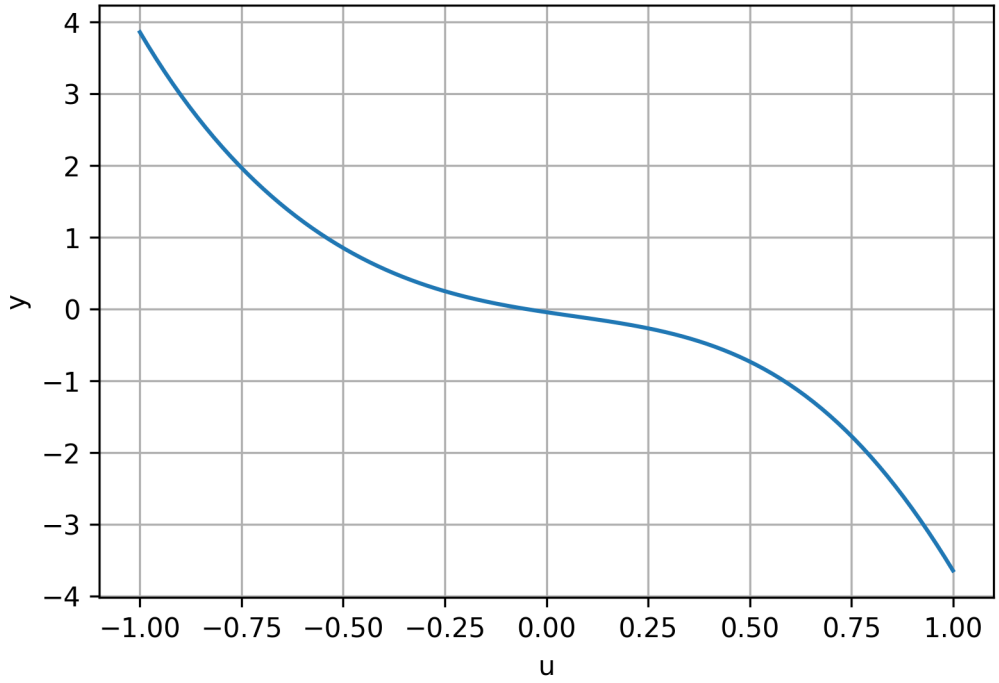
Rys. 1.8. Charakterystyka statyczna dla wielomianu o stopniu $N = 3$



Rys. 1.9. Charakterystyka statyczna dla wielomianu o stopniu $N = 4$

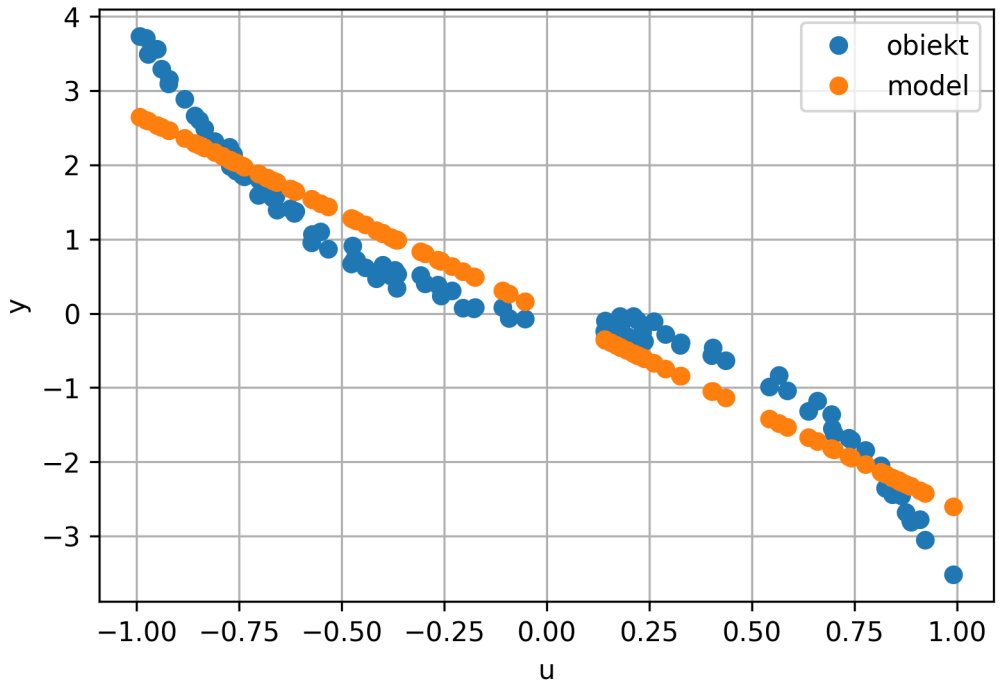


Rys. 1.10. Charakterystyka statyczna dla wielomianu o stopniu $N = 5$

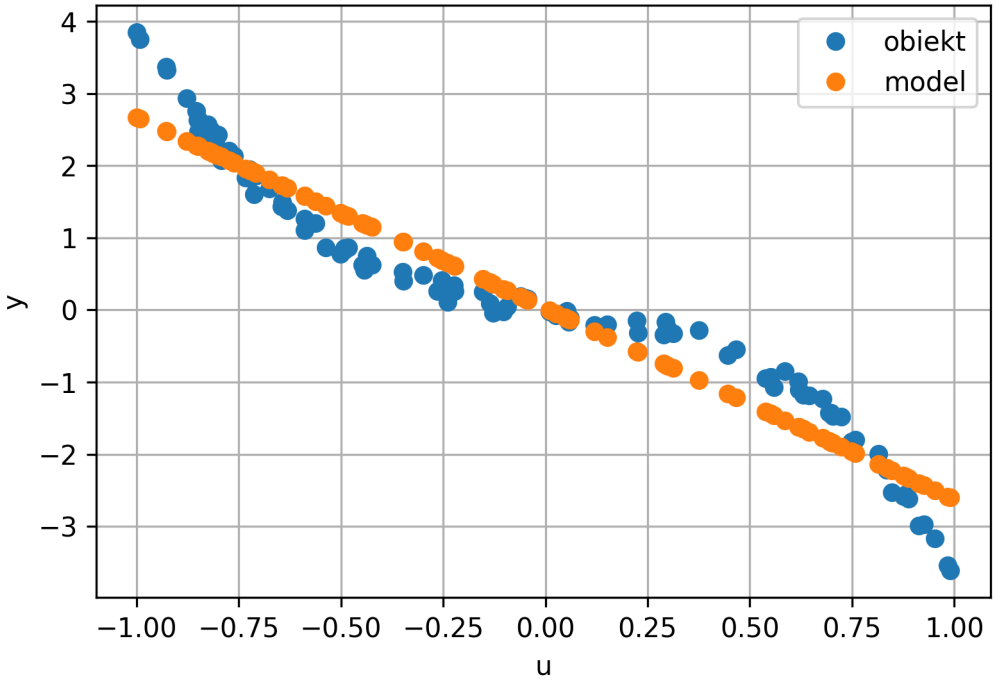


Rys. 1.11. Charakterystyka statyczna dla wielomianu o stopniu $N = 6$

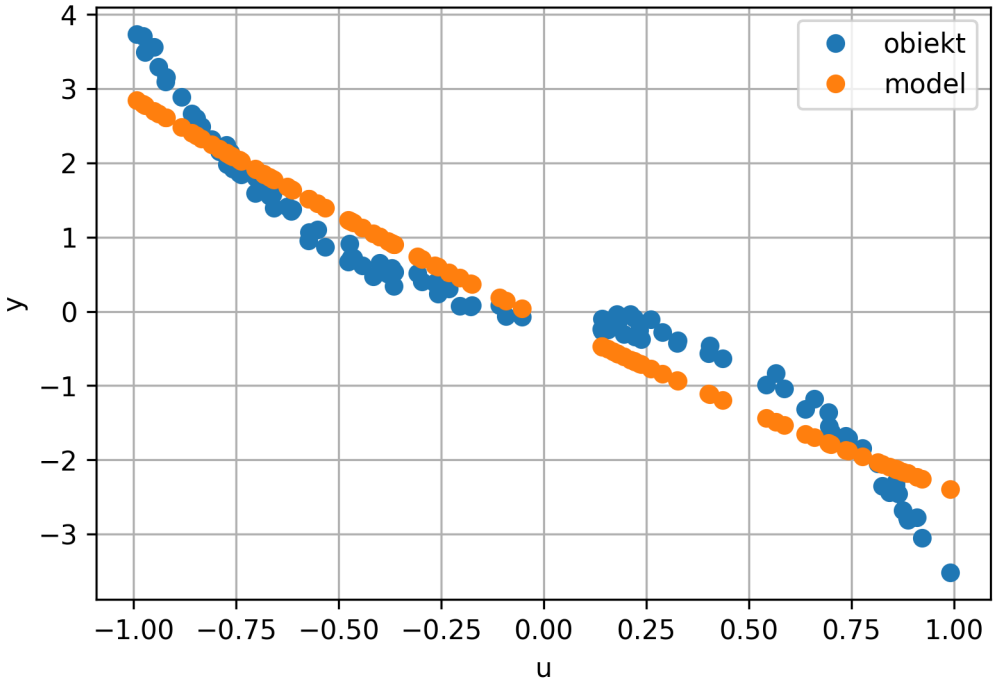
1.3.3. Wykresy wyjść modeli na tle wyjść obiektu



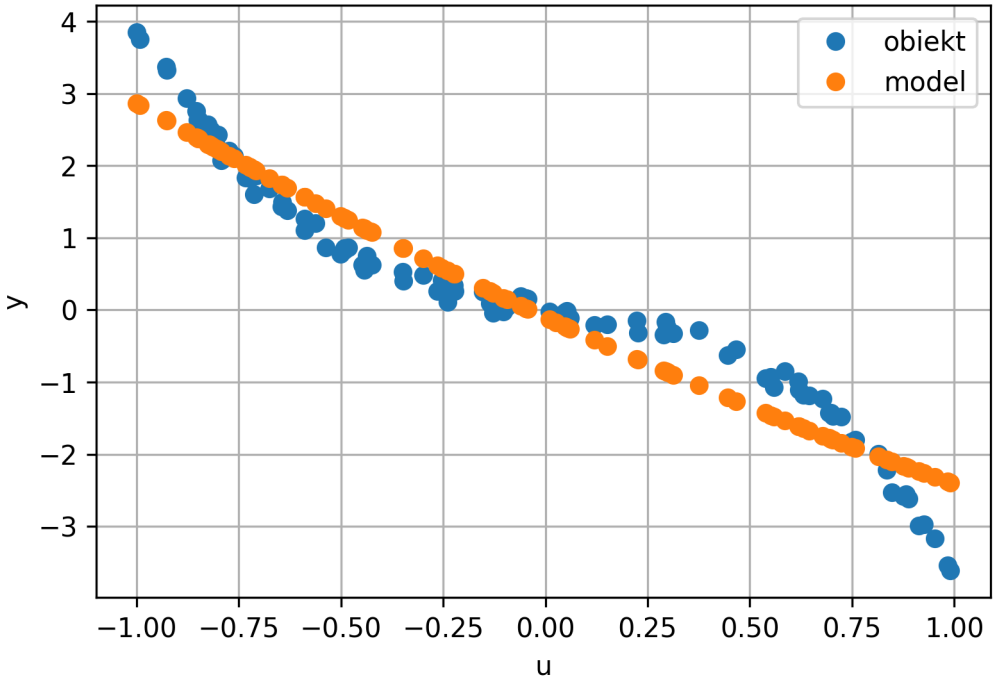
Rys. 1.12. Wyjście modelu z $N = 1$ dla danych uczących na tle wyjścia obiektu



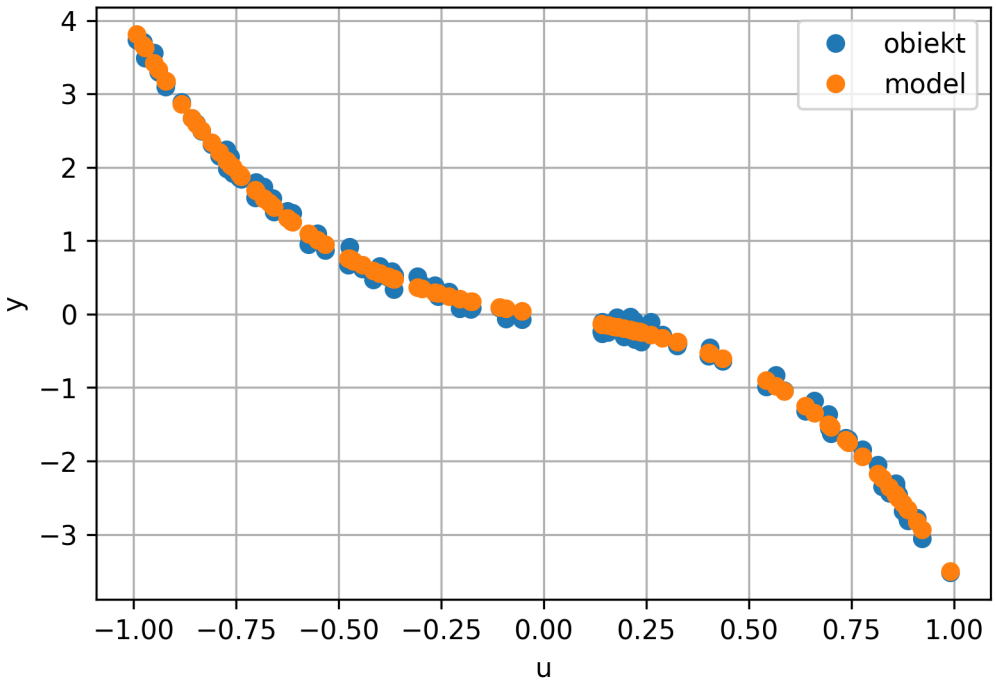
Rys. 1.13. Wyjście modelu z $N = 1$ dla danych uczących na tle wyjścia obiektu



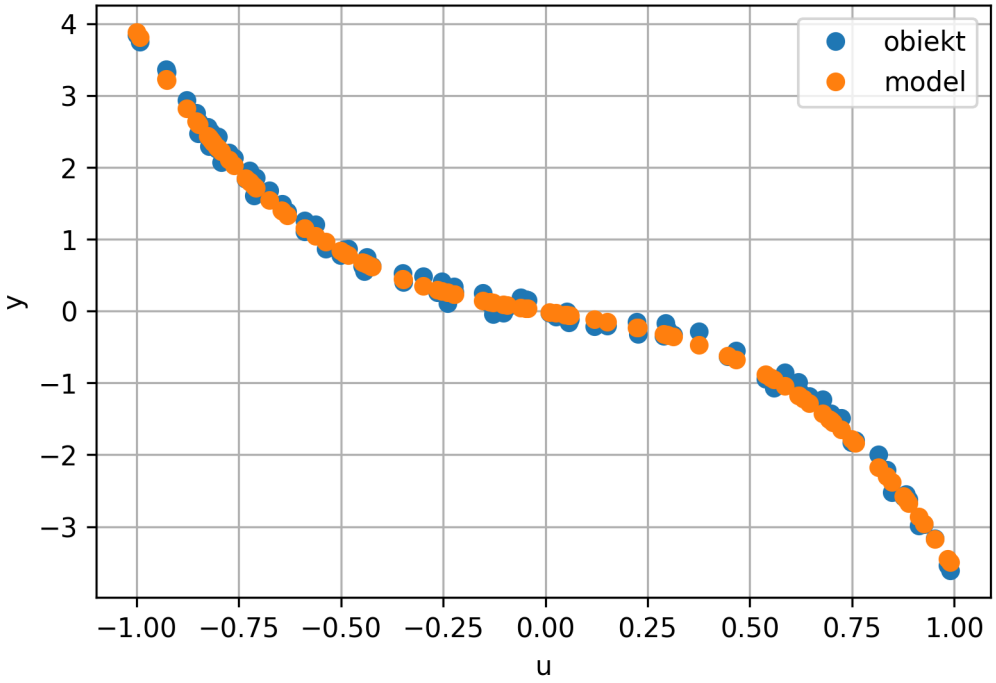
Rys. 1.14. Wyjście modelu z $N = 2$ dla danych uczących na tle wyjścia obiektu



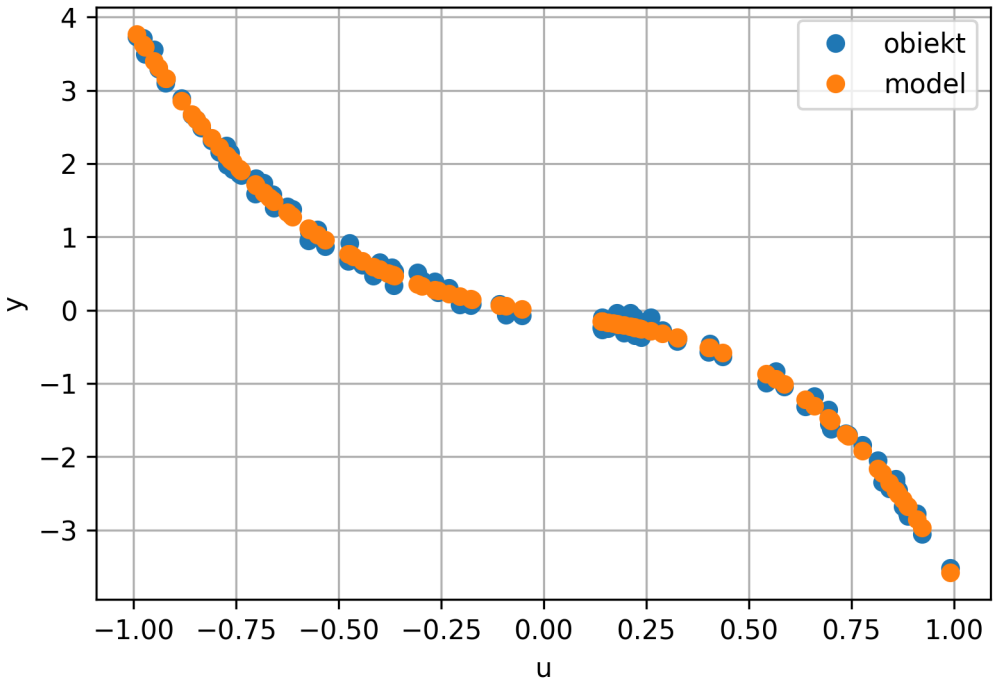
Rys. 1.15. Wyjście modelu z $N = 2$ dla danych uczących na tle wyjścia obiektu



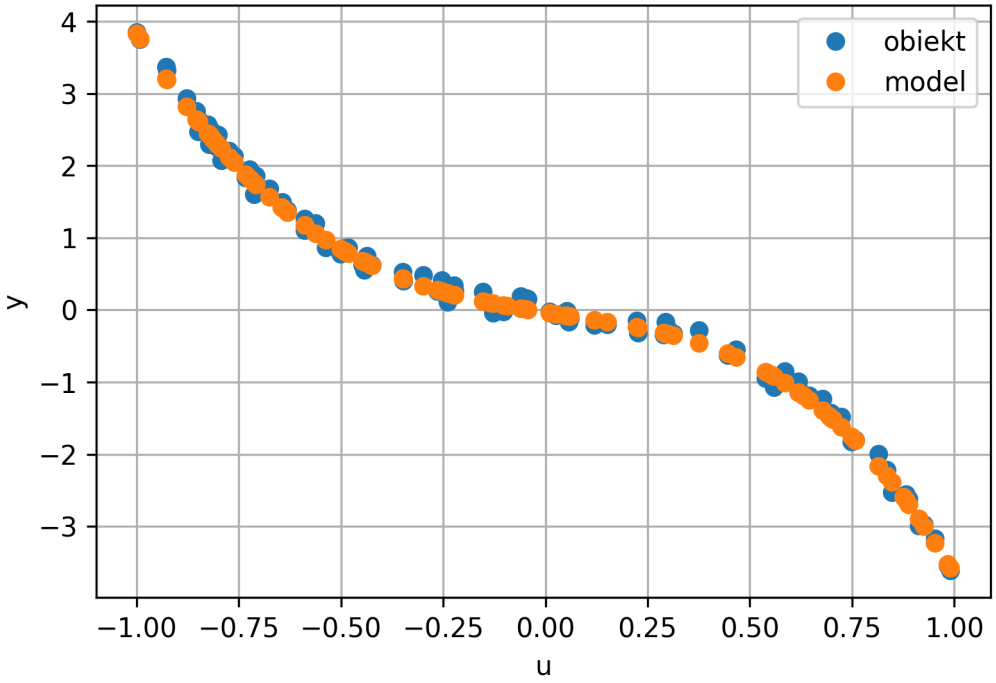
Rys. 1.16. Wyjście modelu z $N = 3$ dla danych uczących na tle wyjścia obiektu



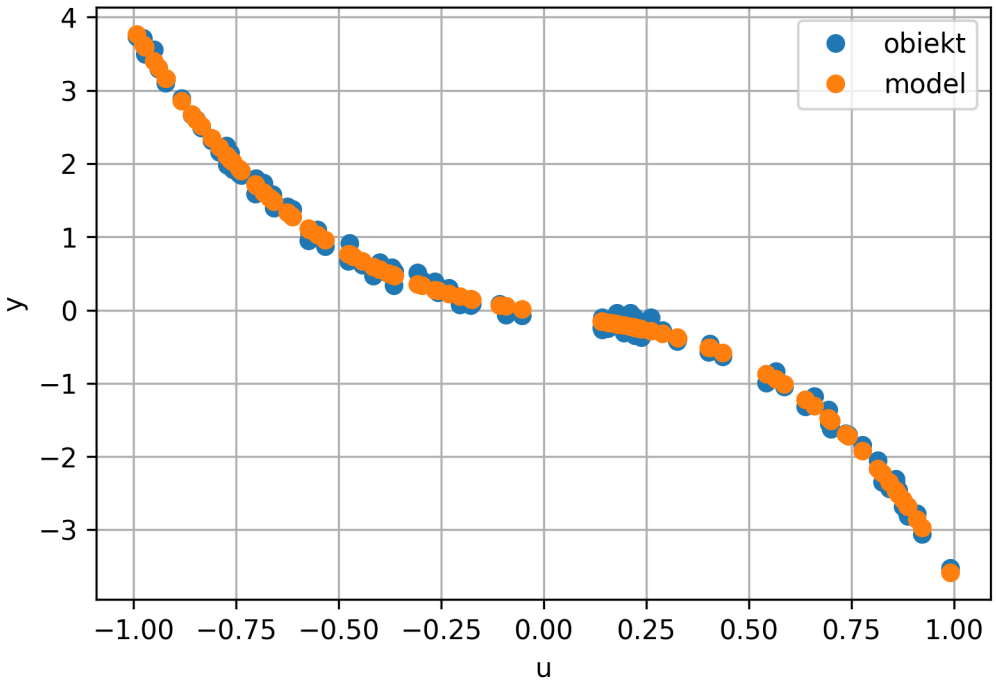
Rys. 1.17. Wyjście modelu z $N = 3$ dla danych uczących na tle wyjścia obiektu



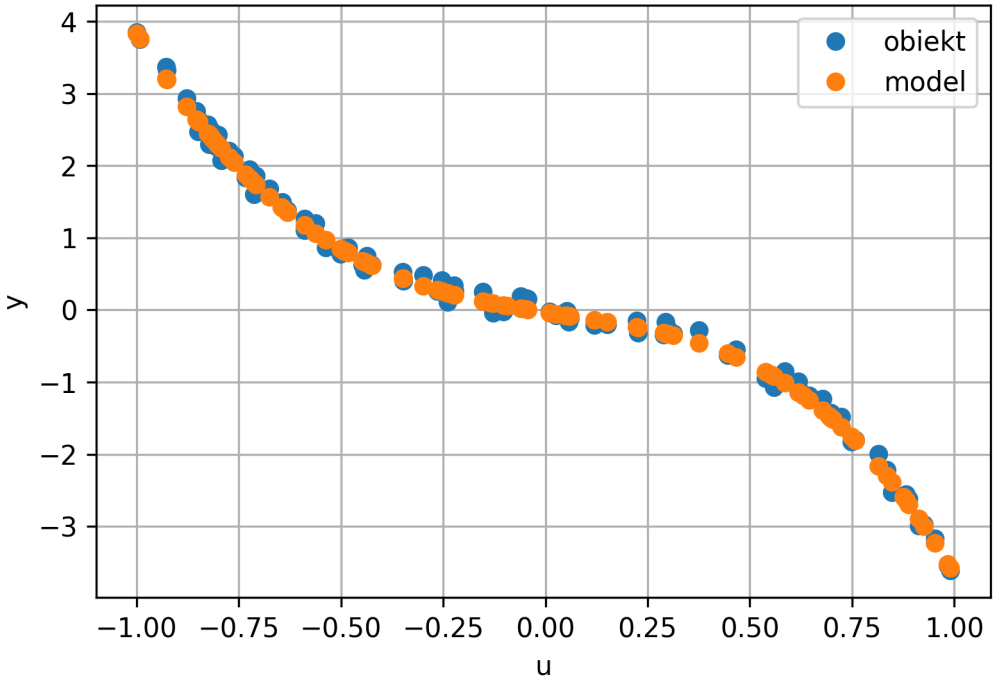
Rys. 1.18. Wyjście modelu z $N = 4$ dla danych uczących na tle wyjścia obiektu



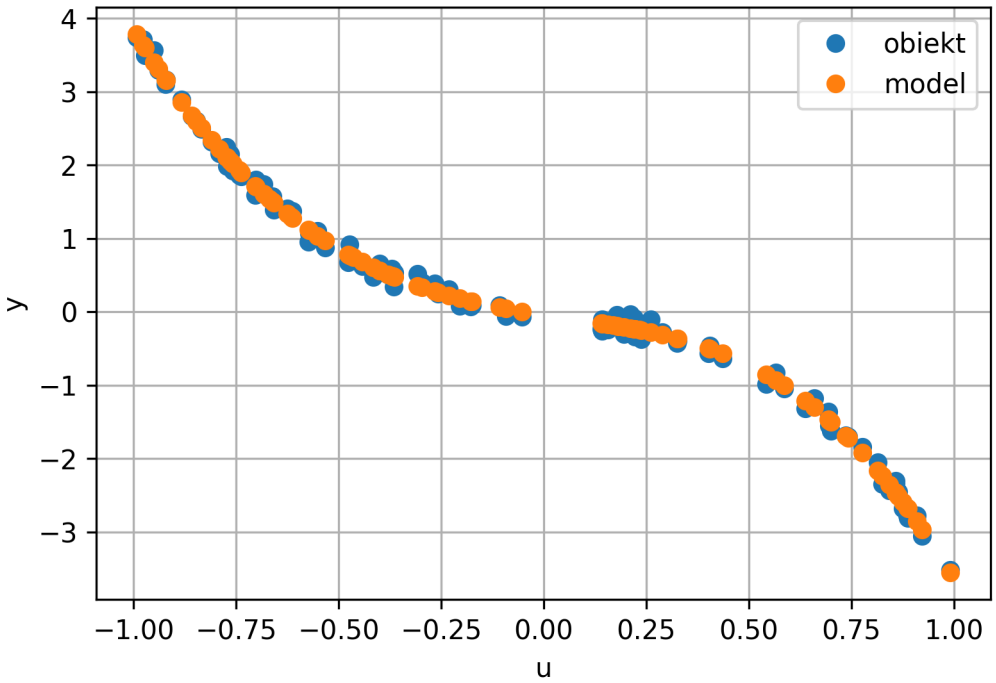
Rys. 1.19. Wyjście modelu z $N = 4$ dla danych uczących na tle wyjścia obiektu



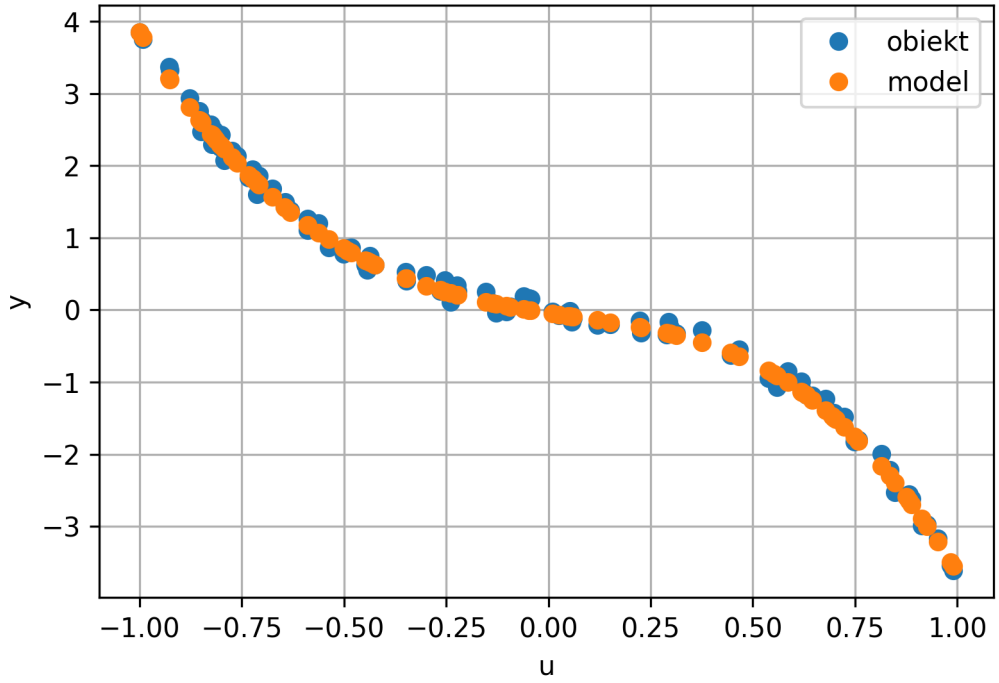
Rys. 1.20. Wyjście modelu z $N = 5$ dla danych uczących na tle wyjścia obiektu



Rys. 1.21. Wyjście modelu z $N = 5$ dla danych uczących na tle wyjścia obiektu



Rys. 1.22. Wyjście modelu z $N = 6$ dla danych uczących na tle wyjścia obiektu



Rys. 1.23. Wyjście modelu z $N = 6$ dla danych uczących na tle wyjścia obiektu

1.3.4. Błędy modeli

Przedstawione błędy w tabeli 1.2 zostały wyliczone przy wykorzystaniu funkcji nieliniowego modelu statycznego oraz funkcji przedstawionych w rozdziale 1.2.4.

Stopień wielomianu	Dane uczące	Dane weryfikujące
1	0.20054	0.20108
2	0.19073	0.19837
3	0.00885	0.01001
4	0.00842	0.00913
5	0.00842	0.00912
6	0.00838	0.00929

Tab. 1.2. Błędy nieliniowego modelu statycznego dla n-stopnia wielomianu

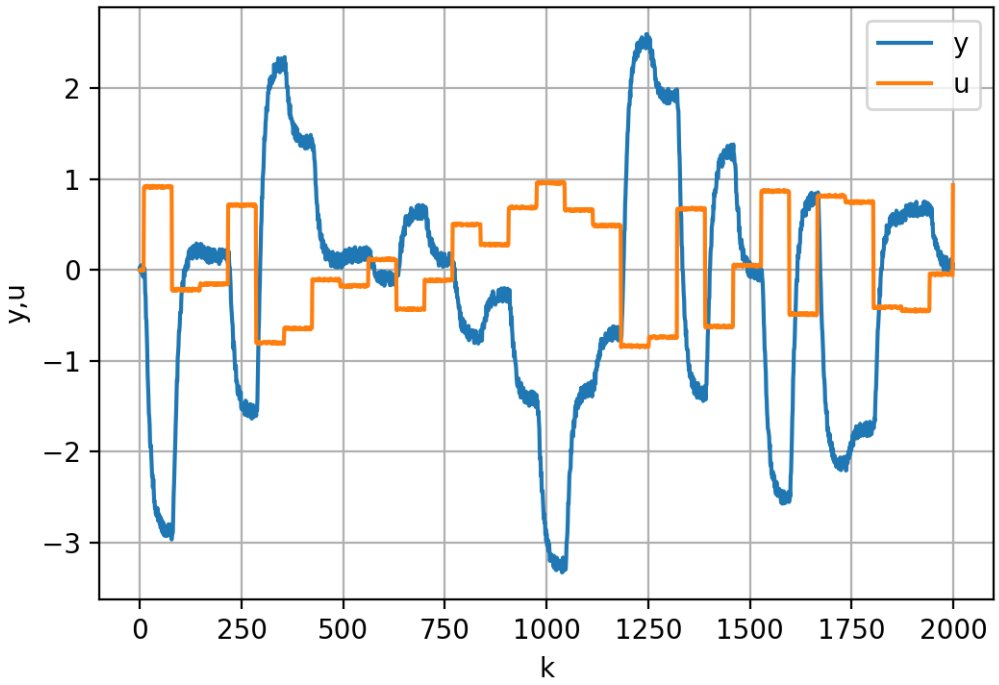
1.3.5. Omówienie wyników

Jak można zauważyć na podstawie otrzymanych rezultatów symulacji, wraz ze zwiększaniem stopnia wielomianu, otrzymuje się coraz lepszej dopasowany model do danych uczących. Jest tak też dla danych weryfikujących, jednakże do momentu w którym model za bardzo dopasowuje się już do samych danych uczących, a nie do modelu. W przypadku analizowanych danych najlepszym modelem okazuje się model z wielomianem stopnia 5, ponieważ dla tego modelu błąd danych weryfikujących jest najmniejszy.

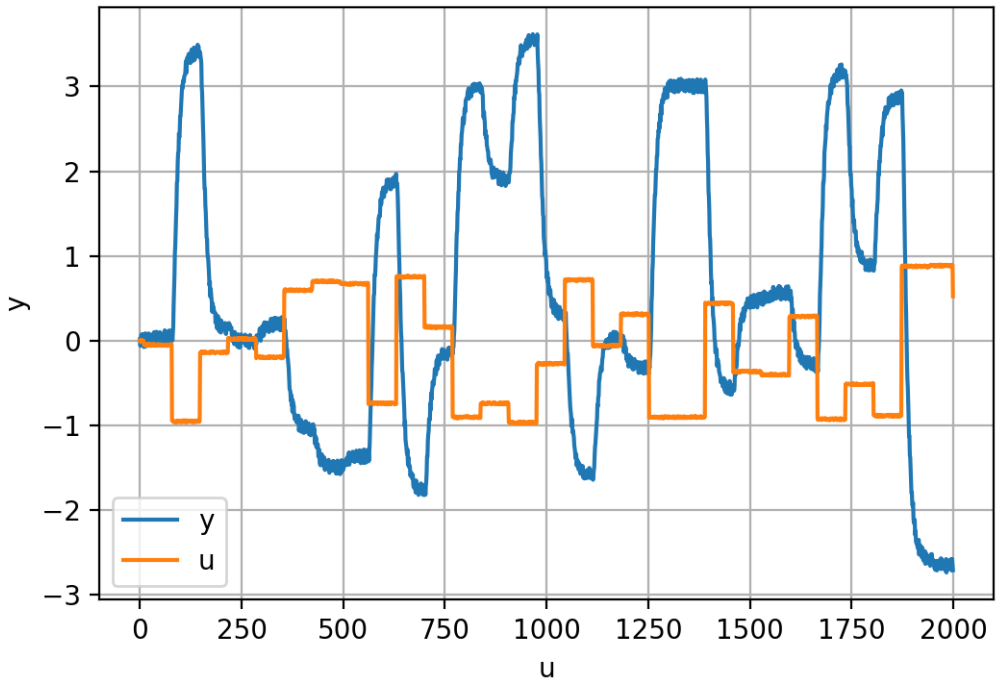
2. Identyfikacja modeli dynamicznych

2.1. Podział danych

Otrzymane dane zostały podzielone przed dostarczeniem, więc nie było konieczności ich podziału.



Rys. 2.1. Dane uczące modelu dynamicznego



Rys. 2.2. Dane weryfikujące modelu dynamicznego

2.2. Modele liniowe

Postać matematyczna:

$$y(k) = \sum_{i=1}^{n_B} b_i u(k-i) + \sum_{i=1}^{n_A} a_i u(k-i) \tag{2.1}$$

Implementacja w języku Python:

```
def nlin_mod_dyn_func(u, y, a, b, k):
    val = 0
    n = a.shape[0]
    for j in range(1, n + 1):
        val += b[j - 1] * u[k - j] + a[j - 1] * y[k - j]
    return val
```

2.2.1. Wyznaczenie parametrów

W celu wyznaczenia wektorów parametrów a oraz b wykorzystano metodę najmniejszych kwadratów.

$$\begin{bmatrix} b_0 \\ \vdots \\ b_{n_B} \\ a_0 \\ \vdots \\ a_{n_A} \end{bmatrix} = \begin{bmatrix} u_{ucz}(i-1) & \cdots & u_{ucz}(0) & y_{ucz}(i-1) & \cdots & y_{ucz}(0) \\ u_{ucz}(i) & \cdots & u_{ucz}(1) & y_{ucz}(i) & \cdots & y_{ucz}(1) \\ \vdots & & \vdots & \vdots & & \vdots \\ u_{ucz}(k_m-1) & \cdots & u_{ucz}(k_m-i) & y_{ucz}(k_m-1) & \cdots & y_{ucz}(k_m-i) \end{bmatrix} \setminus \begin{bmatrix} y_{ucz}(i) \\ y_{ucz}(i+1) \\ \vdots \\ y_{ucz}(k_m) \end{bmatrix} \tag{2.2}$$

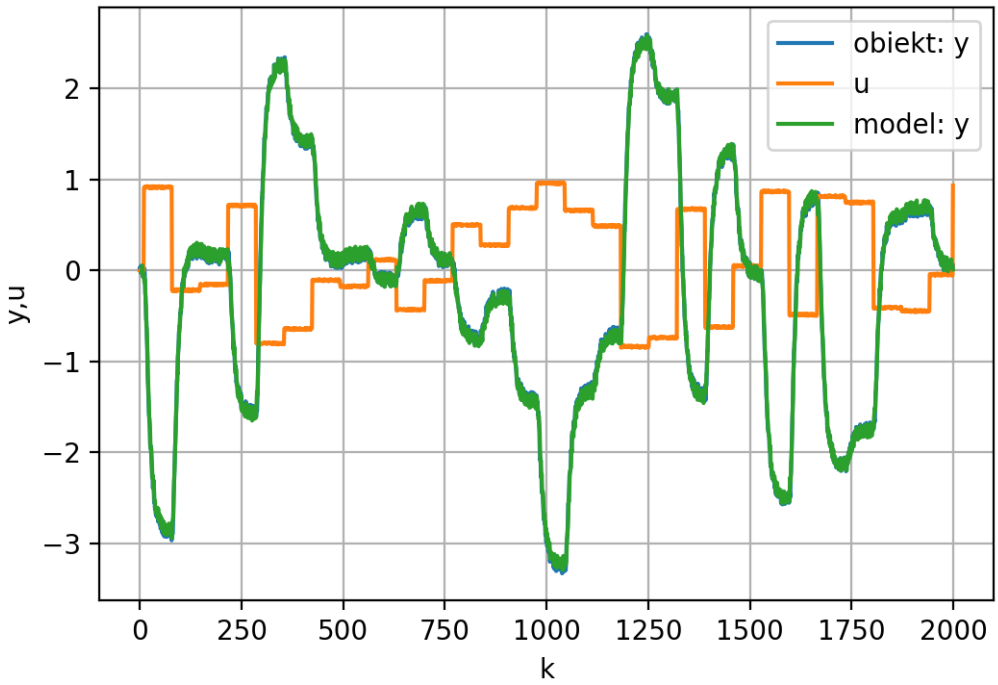
Ta metoda została zaimplementowana w autorskiej funkcji w języku Python z wykorzystaniem biblioteki NumPy.

Przyjmowane argumenty:

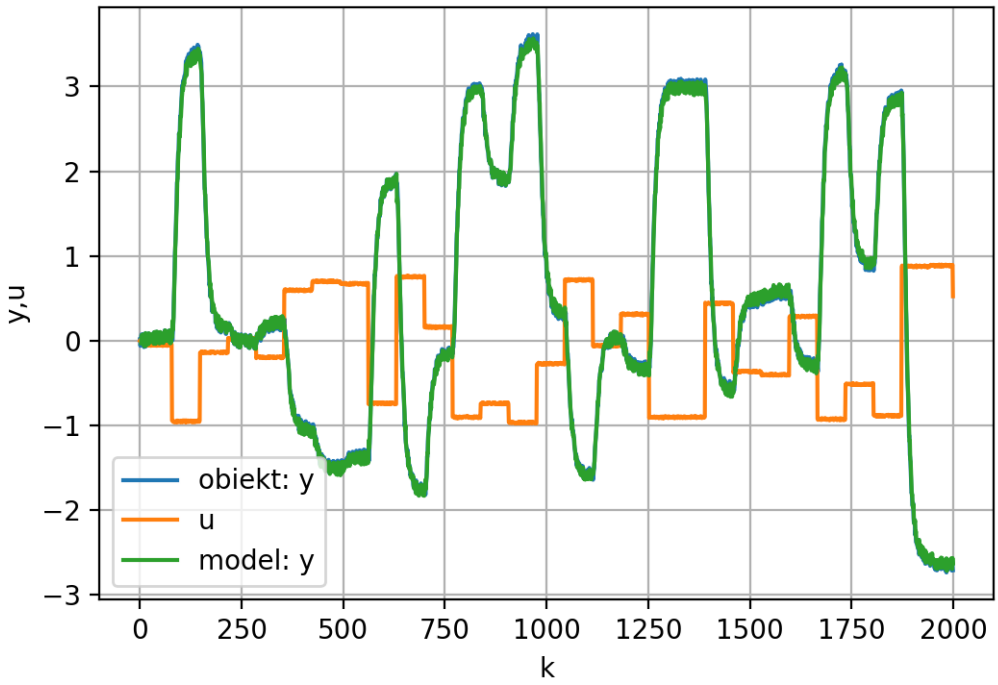
— u_k - wektor wartości wejściowych dla danych chwil czasu k
— y_k - wektor wartości wyjściowych dla danych chwil czasu k
— n - stopień wielomianu

```
def nmk_nlin_mod_dyn(u_k, y_k, n):
    M_b = np.zeros((u_k.shape[0] - n, n))
    M_a = np.zeros((u_k.shape[0] - n, n))
    for i in range(1, n + 1):
        M_b[:, i - 1] = u_k[n - i:-i]
        M_a[:, i - 1] = y_k[n - i:-i]
    M = np.column_stack((M_b, M_a))
    w_temp = np.dot(np.dot(np.linalg.inv(np.dot(M.T, M)), M.T), y_k[n:])
    b = w_temp[:n]
    a = w_temp[n:]
    return a, b
```

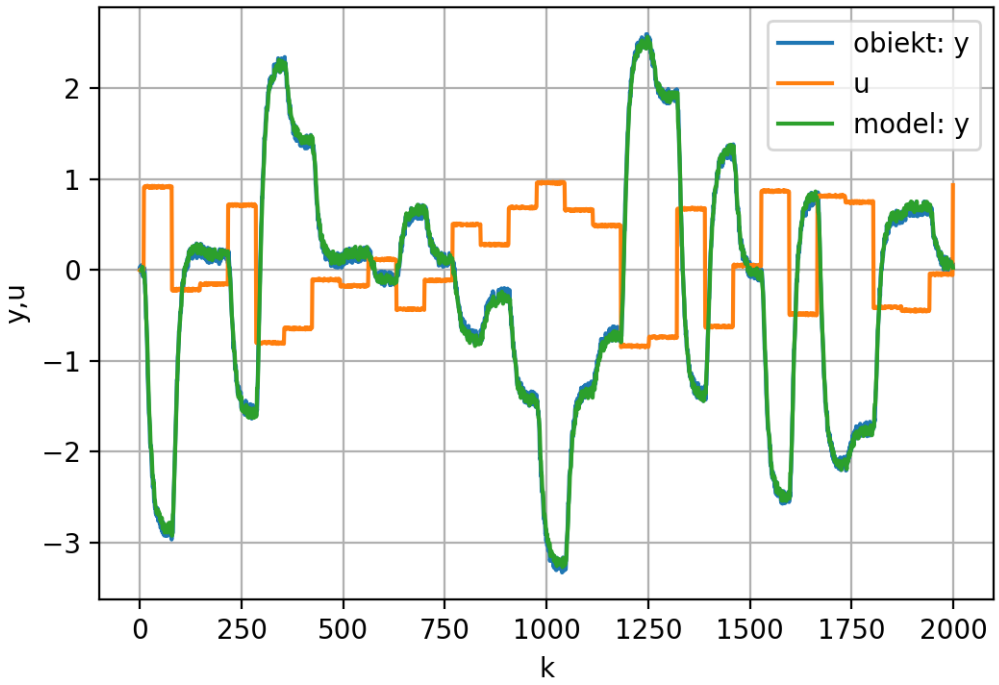
2.2.2. Wykresy wyjść modeli bez rekurencji na tle wyjść obiektu



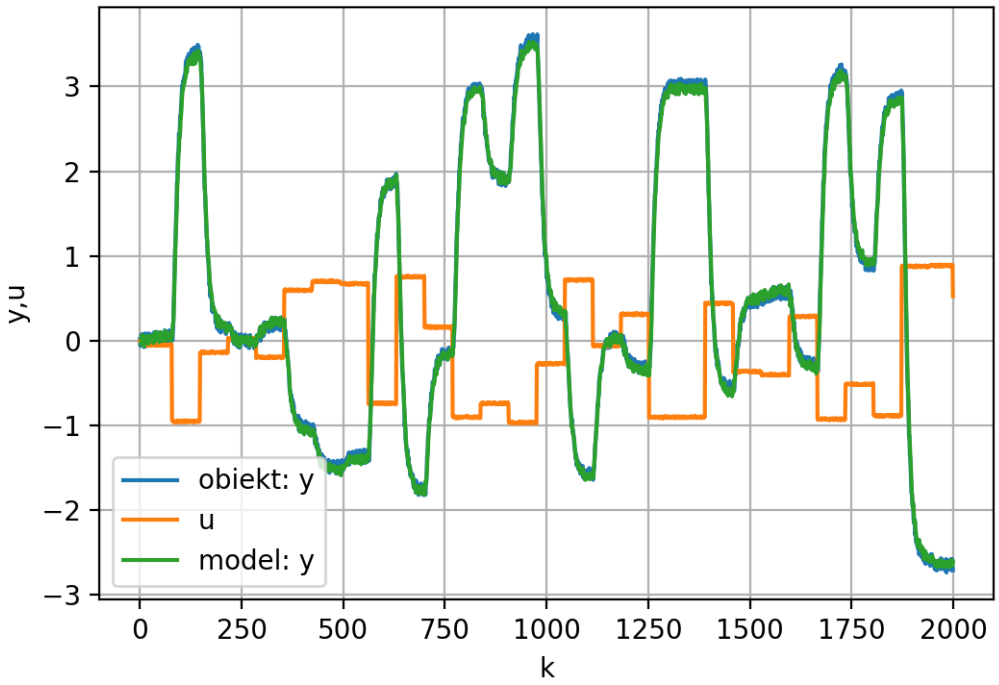
Rys. 2.3. Wyjście modelu pierwszego rzędu dla danych uczących na tle wyjścia obiektu



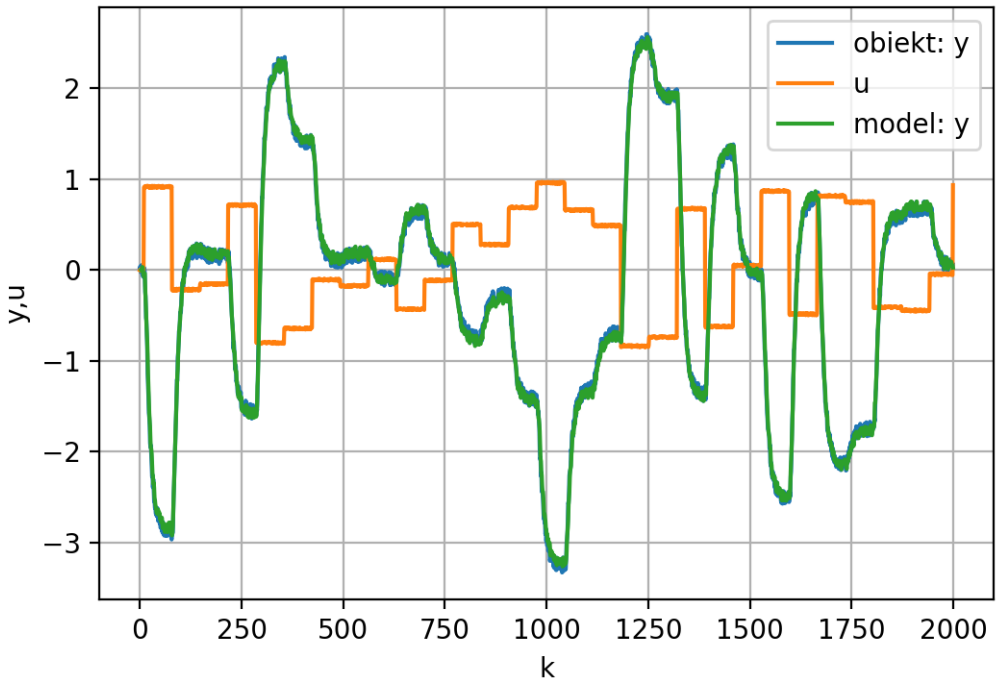
Rys. 2.4. Wyjście modelu pierwszego rzędu dla danych weryfikujących na tle wyjścia obiektu



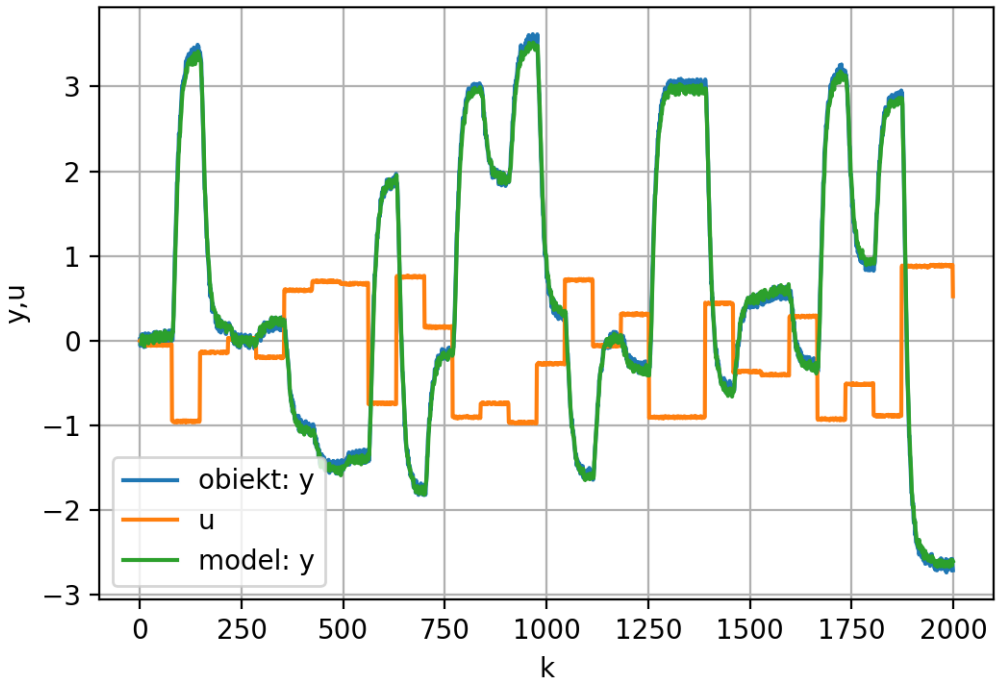
Rys. 2.5. Wyjście modelu drugiego rzędu dla danych uczących na tle wyjścia obiektu



Rys. 2.6. Wyjście modelu drugiego rzędu dla danych weryfikujących na tle wyjścia obiektu

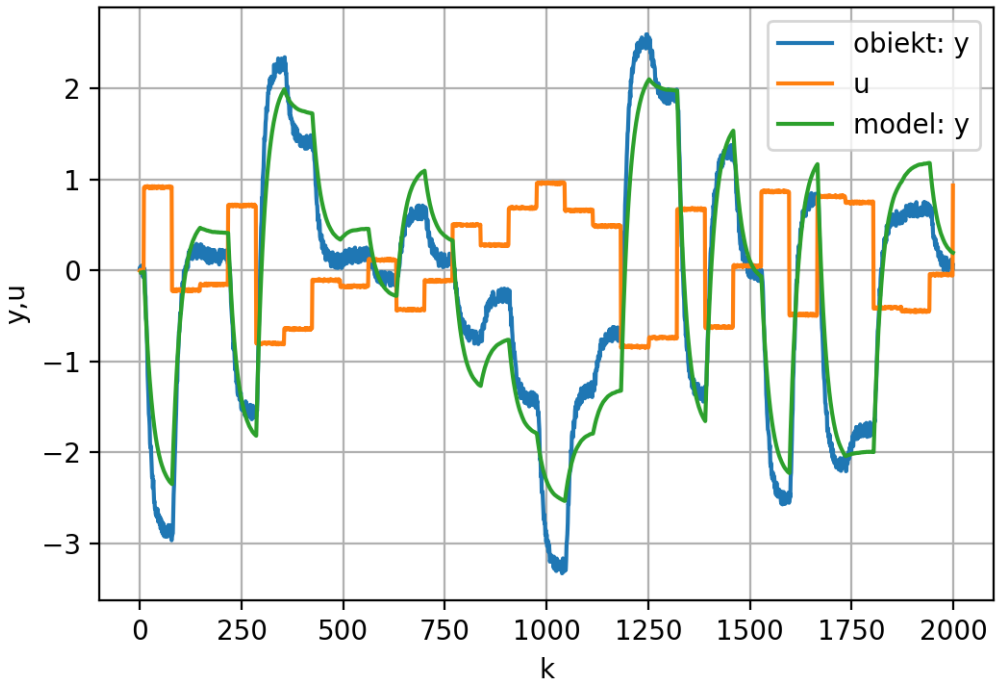


Rys. 2.7. Wyjście modelu trzeciego rzędu dla danych uczących na tle wyjścia obiektu

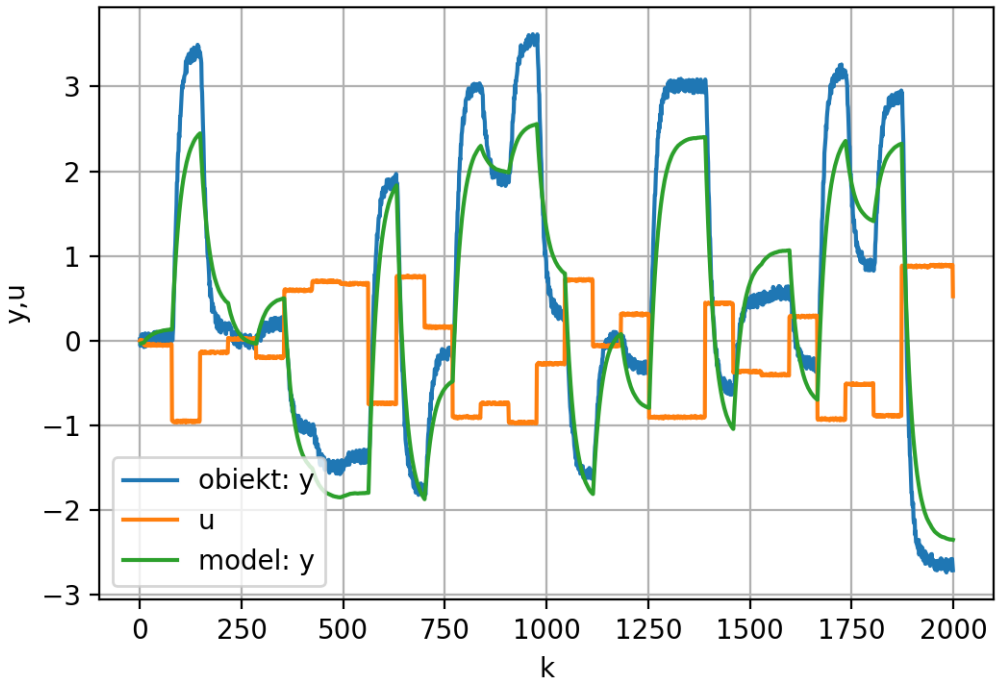


Rys. 2.8. Wyjście modelu trzeciego rzędu dla danych weryfikujących na tle wyjścia obiektu

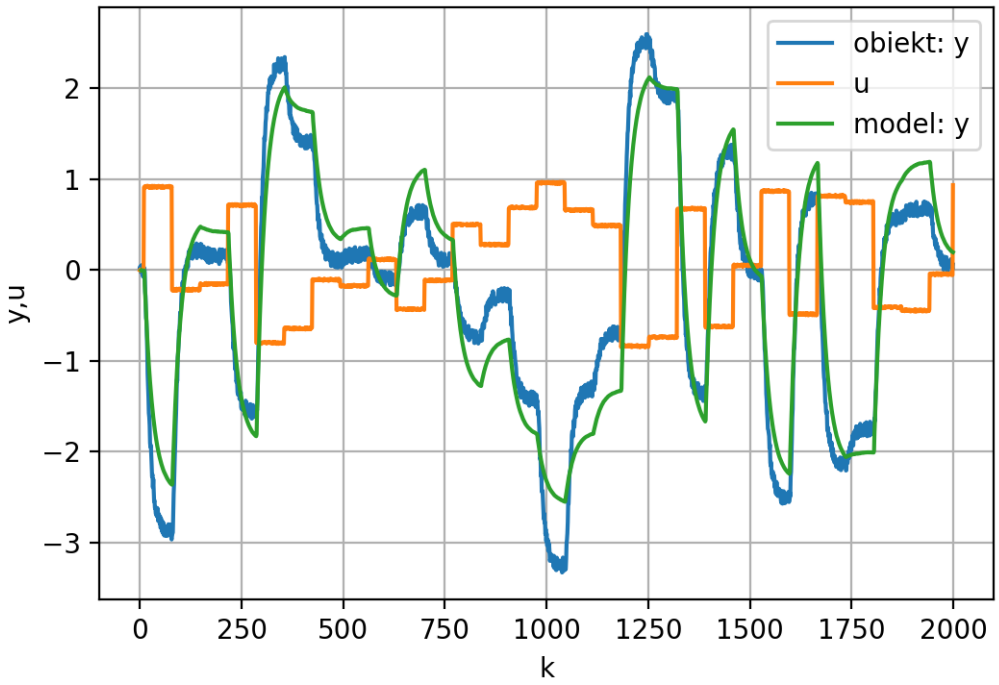
2.2.3. Wykresy wyjść modeli z rekurencją na tle wyjść obiektu



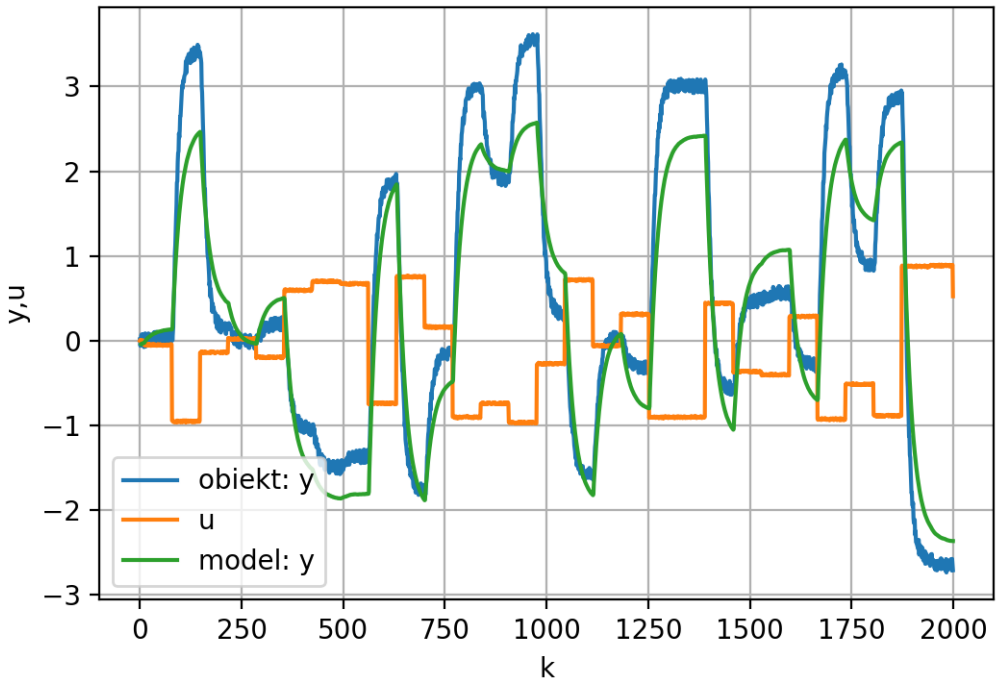
Rys. 2.9. Wyjście modelu pierwszego rzędu z rekurencją dla danych uczących na tle wyjścia obiektu



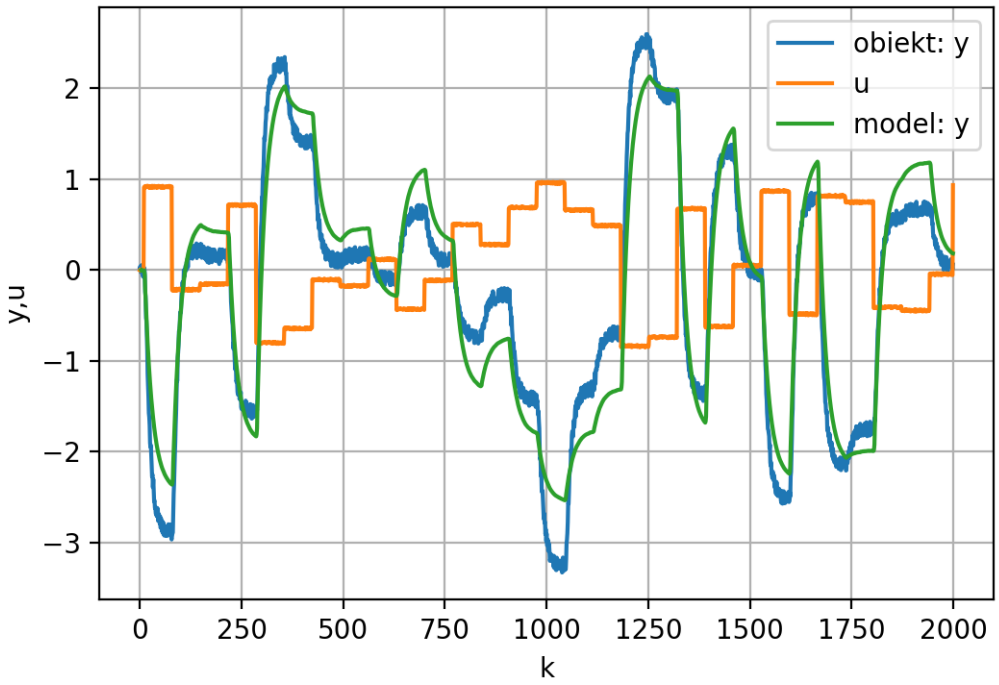
Rys. 2.10. Wyjście modelu pierwszego rzędu z rekurencją dla danych weryfikujących na tle wyjścia obiektu



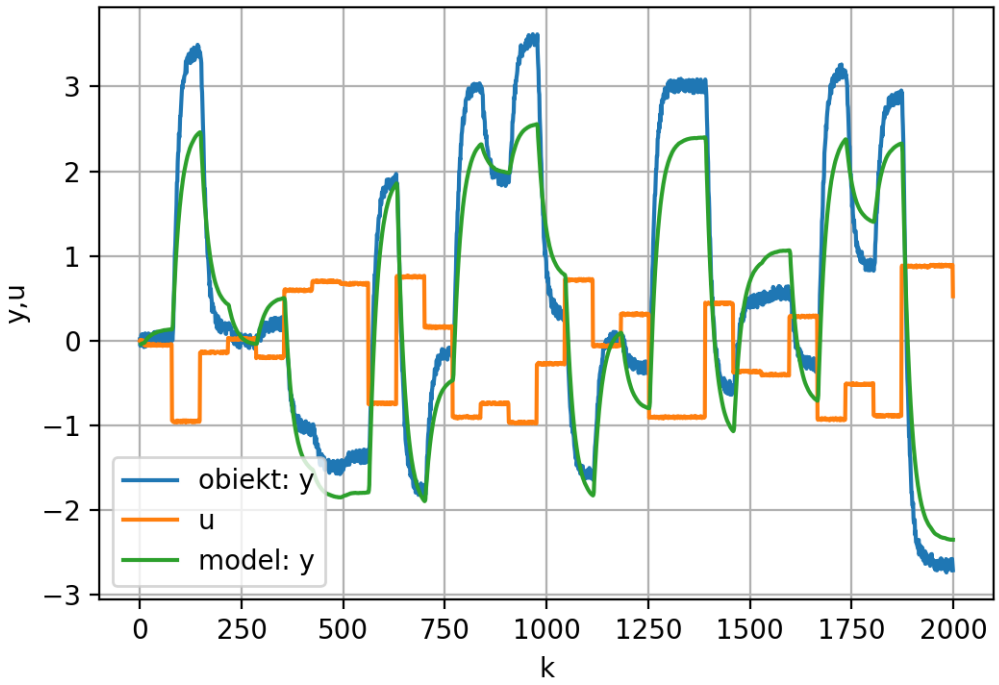
Rys. 2.11. Wyjście modelu drugiego rzędu z rekurencją dla danych uczących na tle wyjścia obiektu



Rys. 2.12. Wyjście modelu drugiego rzędu z rekurencją dla danych weryfikujących na tle wyjścia obiektu



Rys. 2.13. Wyjście modelu trzeciego rzędu z rekurencją dla danych uczących na tle wyjścia obiektu



Rys. 2.14. Wyjście modelu trzeciego rzędu z rekurencją dla danych weryfikujących na tle wyjścia obiektu

2.2.4. Błędy modeli

W celu dokładniejszego dokładniejszej analizy modelu obliczono jego błąd kwadratowy z wykorzystaniem autorskich funkcji w języku Python.

```
def model_error(y, y_mod):
    return np.sum((y[:, np.newaxis] - y_mod[:, np.newaxis]) ** 2) / y_mod.shape[0]

def model_dyn_error(u, y, a, b, k_start, func):
    k_vals = np.array(range(k_start, u.shape[0]))
    y_mod = func(u, y, a, b, k_vals, k_start)
    return model_error(y[k_start:], y_mod)

def model_dyn_error_rek(u, y, a, b, k_start):
    k_vals = np.array(range(k_start, u.shape[0]))
    y_mod = lin_mod_dyn_rek_y(u, y, a, b, k_vals, k_start)
    return model_error(y[k_start:][:, np.newaxis], y_mod[k_start:])
```

Przedstawiony powyżej sposób obliczania błędu został wykorzystany również dla pozostałych modeli dynamicznych.

Rząd dynamiki	Dane uczące		Dane weryfikujące	
	Bez rekurencji	Z rekurencją	Bez rekurencji	Z rekurencją
1	0.00567	0.14692	0.00603	0.26571
2	0.00495	0.15052	0.00565	0.26518
3	0.00481	0.14415	0.00560	**0.25584**

2.2.5. Omówienie wyników

Na podstawie danych można zaobserwować, że im wyższy rząd dynamiki tym mniejszy błąd jest w przypadku modelu dla danych uczących. W przypadku danych weryfikujących błąd zmniejsza się do pewnego momentu. W przypadku wykorzystania rekurencji, otrzymany błąd jest znacznie większy i otrzymany przebieg wyjścia modelu znacznie odbiega od rzeczywistego z obiektu. Najlepszym z modeli okazał się model rzędu 3.

2.3. Modele nieliniowe

Jako nieliniowe modele wykorzystano szeregi wielomianowe

$$y(k) = \sum_{j=1}^{r_{dyn}} \sum_{i=1}^{p} w_{i+p(j-1)} x(k-j)^i \tag{2.3}$$

Implementacja w języku Python:

```
def nlin_mod_dyn_func(u, y, w, k, r_dyn, p):
    val = 0
    # n = a.shape[0]
    for i in range(1, r_dyn + 1):
        d = (i - 1) * p
        for n in range(1, p + 1):
            u_ind = d + n - 1
            y_ind = u_ind + r_dyn * p
            val += w[u_ind] * (u[k - i] ** n) + w[y_ind] * (y[k - i] ** n)
    return val
```

2.3.1. Wyznaczenie parametrów

W celu wyznaczenia parametrów wykorzystano metodę najmniejszych kwadratów, zaimplementowaną w autorskiej funkcji w języku Python z wykorzystaniem biblioteki NumPy.

Przyjmowane argumenty:

— u_k - wektor wartości wejściowych dla danych chwil czasu k
— y_k - wektor wartości wyjściowych dla danych chwil czasu k
— r_{dyn} - rząd dynamiki
— p - stopień wielomianu

```
def nmk_nlin_mod_dyn(u_k, y_k, r_dyn, p):
    M = np.zeros((u_k.shape[0] - r_dyn, r_dyn * p * 2))
    for i in range(r_dyn * p * 2):
        if i < r_dyn * p:
            r = int(i / p + 1)
            M[:, i] = u_k[r_dyn - r:-r] ** (i % p + 1)
        else:
            r = int((i - r_dyn * p) / p + 1)
            M[:, i] = y_k[r_dyn - r:-r] ** (i % p + 1)

    return np.dot(np.dot(np.linalg.inv(np.dot(M.T, M)), M.T), y_k[r_dyn:])
```

2.3.2. Wykresy wyjść modeli bez rekurencji na tle wyjść obiektu

Z powodu ograniczonej wielkości pliku przesyłanego na serwerze studia 3, te wykresy po wygenerowania znajdą się w folderze *wykresy_dyn_nlin*

2.3.3. Wykresy wyjść modeli z rekurencją na tle wyjść obiektu

Z powodu ograniczonej wielkości pliku przesyłanego na serwerze studia 3, te wykresy po wygenerowania znajdą się w folderze *wykresy_dyn_nlin*.

2.3.4. Błędy modeli

W celu dokładniejszego dokładniejszej analizy modelu obliczono jego błąd kwadratowy z wykorzystaniem autorskich funkcji w języku Python.

```
def model_error(y, y_mod):
    return np.sum((y[: np.newaxis]-y_mod[:,np.newaxis])**2)/y_mod.shape[0]
```

Przedstawiony powyżej sposób obliczania błędu został wykorzystany również dla pozostałych modeli dynamicznych.

Rząd	Dane uczące						Dane weryfikujące					
	Bez rekurencji			Z rekurencją			Bez rekurencji			Z rekurencją		
Stop.	2	3	4	2	3	4	2	3	4	2	3	4
1	0.0056	0.0053	0.0053	0.1412	0.0128	0.0128	0.0064	0.0054	0.0054	0.2883	0.0255	0.0227
2	0.0049	0.0042	0.0041	0.1438	0.0109	0.0108	0.0065	0.0044	0.0044	0.2875	0.0175	0.0185
3	0.0047	0.0037	0.0036	0.1373	0.0077	0.0077	0.0068	0.0041	0.0041	0.2848	0.0126	**0.0141**

2.3.5. Omówienie wyników

Ponownie okazało się, że zwiększanie rzędu dynamiki oraz stopnia wielomianu sprawia, że model staje się dokładniejszy zwłaszcza dla danych uczących, jednakże w tym wypadku nie napotkano na wartości tych parametrów, które są gorsze mimo iż są większe. Jak można zauważyć z przetestowanych modeli najlepszy okazał się model z 3 rzędem dynamiki oraz wielomianem stopnia 4. Prawdopodobnie gdyby zwiększać obydwie te wartości istniałaby możliwość znalezienia jeszcze lepiej działającego modelu, jednakże tylko takie modele przyjęto do przetestowania.