Implementation of Fast Orthogonal Search

by

Guangyi (Patrick) Zhang

A Report submitted to

ELEC 841

Nonlinear Systems: Analysis and Identification

Queen's University

Kingston, Ontario, Canada

April 2020

Contents

Chapter 1

Introduction

1.1 Background

Fast Orthogonal Search (FOS) proposed by Korenberg [1] has been successfully employed to extract precise representation of time-series data and explore the most important signal component from the noise [2, 3, 4, 5].

1.2 Outline

This paper implements FOS algorithm to identify time-invariant nonlinear systems given the generated input and output nonlinear data. The remainder of this document is formed as follows. In Chapter 2, we provide an overview of input and output data generation. Chapter 3 describes the procedure of both noise-free and noisy experiments. Chapter 4 discuss the impact of different parameters and analyze the result. Chapter 5 presents the conclusions of this manuscript and code summary. Appendix includes the code and its explanation.

Chapter 2

Input/Output Data

In this chapter, we describe the generation of input and output nonlinear data.

2.1 Generation of Input

Input $x[n] \sim N(\mu, \sigma), n \in [1, 3000]$. In this context, we set mean value (μ) of zero and standard deviation (σ) of one for Gaussian distribution.

2.2 Generation of Output

We employ three different nonlinear systems to generate output according to the following structure.

$$y[n] = F[y[n-1], ..., y[n-K], x[n], ..., x[n-L]] \tag{2.1}$$

, where F is a multidimensional polynomial.

Table 2.1: Nonlinear Test Systems

System No.	$a0$	$a1$	$a2$	$a3$	$a4$	$a5$	$a6$
1	0.05	0.4	0.1	-0.2	-0.1	0.33	0.0
2	0.01	0.2	0.3	-0.1	0.05	0.2	0.0
3	0.1	0.1	0.5	-0.3	0.22	-0.4	0.1

2.2.1 Noise-Free Case

In this paper, we use the following nonlinear equation for noise free output generation:

$$y[n] = a0+a1y[n-1]+a2x[n-1]+a3x[n]x[n-2]+a4y[n-1]y[n-2]+a5x[n-2]y[n-2]+a6x[n-1]x[n-2]y[n-2] \tag{2.2}$$

We set coefficients $(a1, ..., a6)$ with different magnitude and sign in the aforementioned equation to generate different order test systems.

For example, according to Table 2.1 , in system No.1,

$$y[n] = 0.05+0.4\times y[n-1]+0.1\times x[n-1]-0.2\times x[n]x[n-2]-0.1\times y[n-1]y[n-2]+0.33\times x[n-2]y[n-2]$$

$y[n], n \in [1, 3000]$ have been generated with carefully selected different coefficients to make sure $y[n]$ not "blow up". As shown in the Table 2.1, System No.1 and No.2 have second order cross-products and system No.3 has third order cross product.

2.2.2 Noisy Case

Noise output $v[n]$ is

$$v[n] = y[n] + u[n] \tag{2.3}$$

, where $var(u[n]) = \frac{P}{100}var(y[n])$.

We select different P values of $30, 50, 70$ and 100 to evaluate the performance of FOS algorithm under different noise cases.

Chapter 3

Experiment

In this chapter, we describe the experiment procedure. The data are divided into three sections, notably training, validation and testing phases.

3.1 Training Phase

Training phase is used to train several models on the training data, where $x\_train, y\_train = x[n], y[n], n \in [N0, 1000]$.

3.1.1 Candidates Generation

In this section, we generate the candidate pool with different K and L representing maximum delay in y-terms and x-terms shown in Equation 2.1. However, in the real world scenario, it is difficult to know the best value of K, L and order of cross products. Therefore, we employ different combination of K and L while assuming the order of cross-products is 2. Table 3.1 presents the number of candidates with different maximum delay [K, L] values.

Table 3.1: Candidate Pool with different K and L

K	10	10	7	7	7	5	5	5	3	3
L	10	7	10	7	5	7	5	3	5	3
Candidates No.	252	189	189	135	104	104	77	54	54	35

3.1.2 Model Generation

In this section, we implement FOS algorithm to search and select best model quickly in order to construct an accurate model. As described in [1], the Mean Squared Error (MSE) can be calculated as

$$MSE = \overline{y[n]^2} - \sum_{m=N0}^{M} g_m^2 D[m,m] \tag{3.1}$$

, where $N0 = max(K, L)$

According to [1], the output of FOS can be presented as:

$$z[n] = \sum_{m=N0}^{M} a_m p_m = \sum_{m=N0}^{M} g_m v_m \tag{3.2}$$

3.2 Validation Phase

Validation phase is used to selected the best model with the chosen K and L values (while minimum $MSE\%$) though validation data.

3.2.1 Noise-free Case

$x\_val, y\_val, y\_pred = x[n], y[n], z[n], n \in [1001 + N0, 2000]$. The best model is selected based on the smallest MSE shown as following:

$$\%MSE = \frac{\overline{(y[n] - z[n])^2}}{\overline{(y[n] - \bar{y})^2}} \times 100\% \tag{3.3}$$

3.2.2 Noisy case

$x\_val, y\_val, y\_pred = x[n], v[n], z[n], n \in [1001 + N0, 2000]$.

Different from noise-free case, a stop criterion is proposed by [6] to terminate the model developing process once the Equation 3.4 is not satisfied:

$$g_{M+1}^2 D[M+1, M+1] > \frac{4}{N - N0 + 1}(\overline{v^2[n]} - \sum_{m=N0}^{M} g_m^2 D[m, m]) \tag{3.4}$$

MSE is presented as:

$$\%MSE = \frac{\overline{(v[n] - z[n])^2}}{var(v[n])} \times 100\% \tag{3.5}$$

3.3 Testing Phase

Testing phase is used to examine the selected model performance on the testing data.

$x\_test, y\_test = x[n], y[n], n \in [2001 + N0, 3000]$.

The MSE calculation is as Equation 3.3 for both noise-free case and noisy case. The goal in both cases is to approximate the noise-free output. Figure 3.1 is the overview of this implementation.

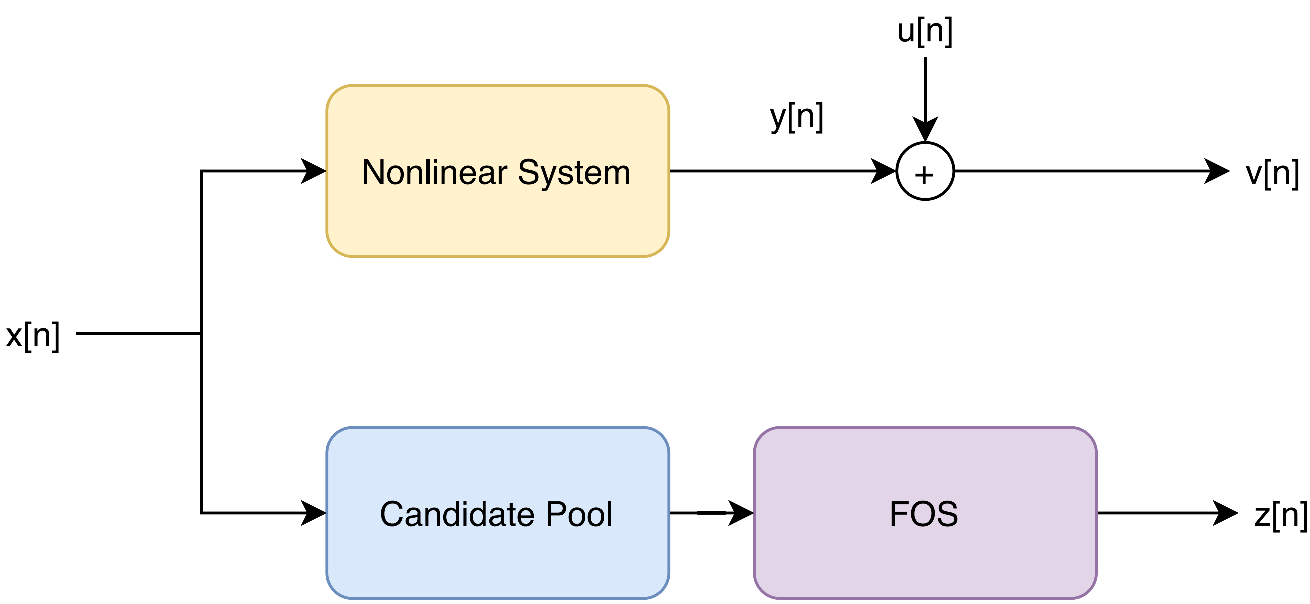

Figure 3.1: Implementation Overview: $y[n]$ is the nonlinear noise-free output, $u[n]$ is the noise added to $y[n]$. Consequently, $v[n]$ is the noisy output. FOS selected the best model through candidate pool and generate model output $z[n]$

Table 4.1: MSE % of Noise-free Case for Validation

K, L	System No.1	System No.2	System No.3
[10,10]	**0.170**	**0.022**	**0.124**
[10,7]	0.171	0.036	0.124
[7,10]	0.171	0.036	0.126
[7,7]	0.171	0.041	0.125
[7,5]	0.172	0.052	0.126
[5,7]	0.171	0.053	0.128
[5,5]	0.172	0.056	0.128
[5,3]	0.183	0.069	0.130
[3,5]	0.182	0.069	0.130
[3,3]	0.183	0.073	0.131

Chapter 4

Discussion

4.1 Noise-free Case

In this section, we show the performance of best selected FOS model to approximate the noise-free output on the testing data. We also study the effect of K and L chosen on MSE result for each of three different nonlinear systems.

Table 4.1 shows the model performance with different K and L values in there different nonlinear systems under noise-free condition.

Table 4.2: MSE % of Noise-free Case for Testing

	System No.1	System No.2	System No.3
Chosen K,L	$[10, 10]$	$[10, 10]$	$[10, 10]$
MSE %	0.207	0.052	0.117

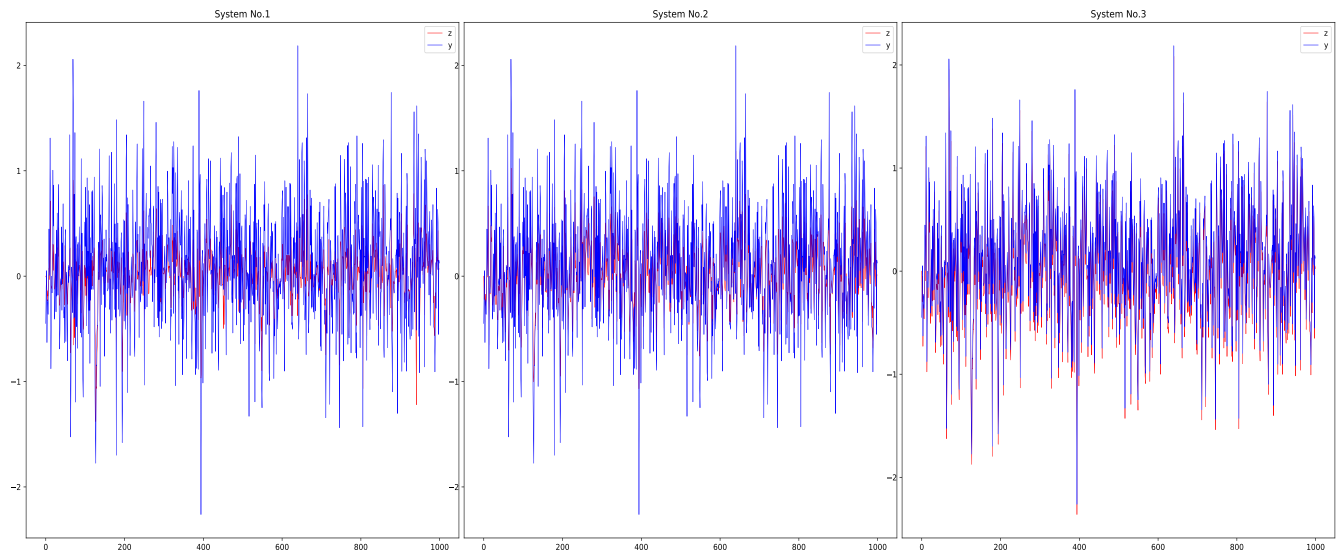

Figure 4.1: Noise-free: System Output Vs. Model Output

In the noise-free scenario, model with the highest K and L values $[10, 10]$ achieve the best performance with the minimum $MSE\%$ obtained in all three nonlinear systems on validation data. System No.2 has the best result 0.052%where the smaller coefficients has been selected for test system generation compared with other systems on testing data, as shown in Table 4.2. Figure 4.2 shows comparison of system output ($y[n]$) and model output ($z[n]$) over the testing data.

4.2 Noisy Case

In this section, we demonstrate the performance through approximation of noisy output ($v[n]$) on validation data. Then we choose best selected FOS model to estimate the noise-free output ($y[n]$)

Table 4.3: MSE % of Noisy Case for Validation, System No.1

K, L	$P = 30$	$P = 50$	$P = 70$	$P = 100$
[10,10]	**4.71**	6.61	13.11	14.71
[10,7]	4.72	6.55	12.95	14.85
[7,10]	4.71	6.49	12.75	14.41
[7,7]	4.72	6.51	12.85	14.53
[7,5]	4.71	**6.34**	12.60	14.39
[5,7]	4.72	6.51	**10.96**	14.31
[5,5]	4.71	6.34	11.22	14.21
[5,3]	4.71	6.34	11.04	14.21
[3,5]	4.71	6.34	11.20	14.24
[3,3]	4.71	6.34	11.03	**13.94**

Table 4.4: MSE % of Noise-free Estimation for Testing, System No.1

	$P = 30$	$P = 50$	$P = 70$	$P = 100$
Chosen K,L	$[10, 10]$	$[7, 5]$	$[5, 7]$	$[3, 3]$
MSE %	4.52	4.62	7.78	7.95

Table 4.5: MSE % of Noisy Case for Validation, System No.2

K, L	$P = 30$	$P = 50$	$P = 70$	$P = 100$
[10,10]	1.15	2.86	5.95	11.43
[10,7]	1.16	2.86	5.82	11.28
[7,10]	**1.13**	2.79	5.78	11.34
[7,7]	1.14	2.77	5.78	11.20
[7,5]	1.15	**2.74**	5.74	11.14
[5,7]	1.15	2.74	5.73	11.17
[5,5]	1.15	2.74	**5.70**	11.11
[5,3]	1.17	2.74	5.70	11.11
[3,5]	1.17	2.76	5.70	**11.04**
[3,3]	1.17	2.76	5.70	11.04

Table 4.6: MSE % of Noise-free Output Estimation for Testing, System No.2

	$P = 30$	$P = 50$	$P = 70$	$P = 100$
Chosen K,L	$[7, 10]$	$[7, 5]$	$[5, 5]$	$[3, 5]$
MSE %	0.14	0.25	0.25	0.24

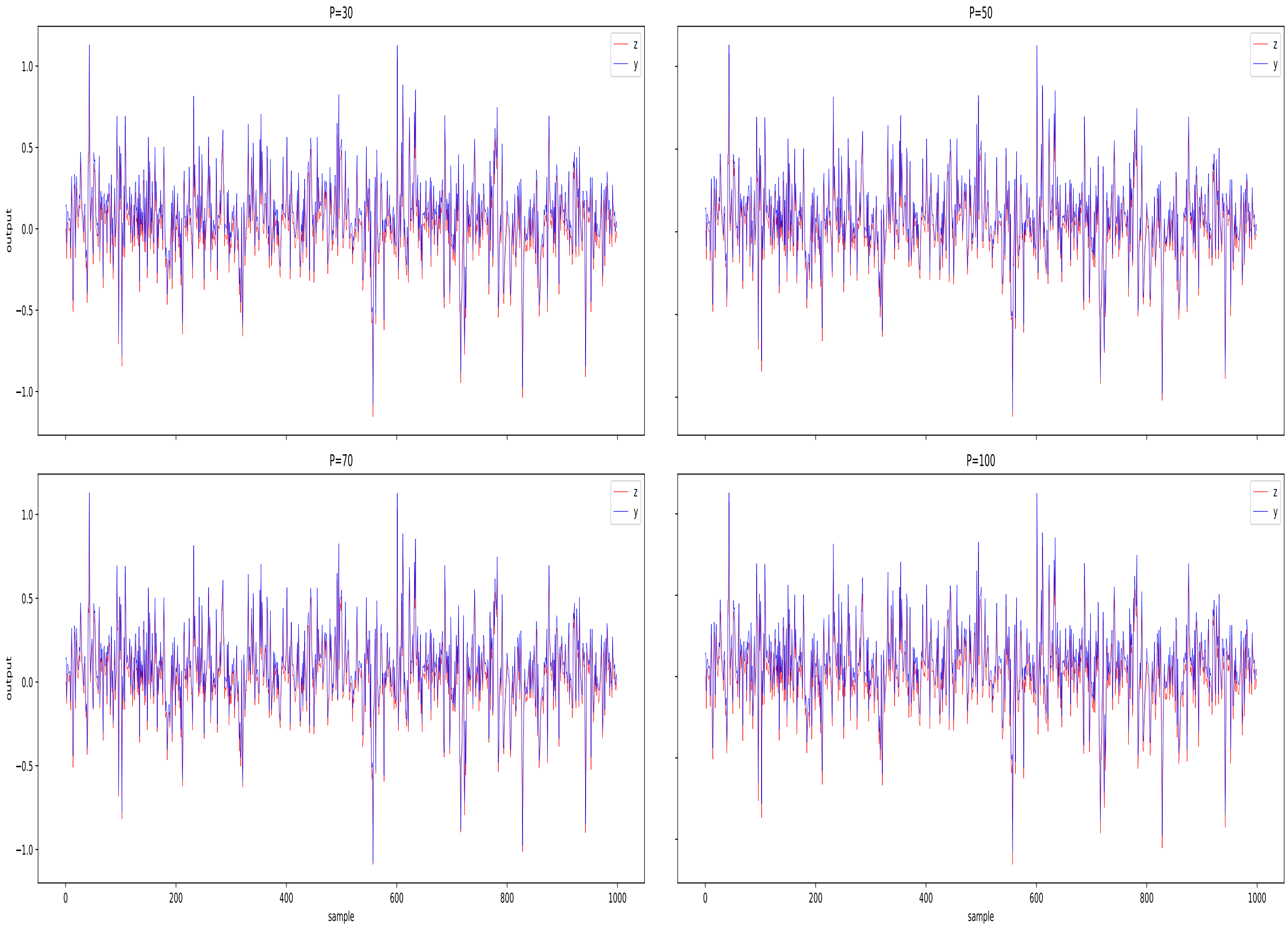

Figure 4.2: System Noise-free Output Vs. Model Output, System No.1

on the testing data. For each of three different nonlinear systems, we investigate the impact of maximum delay K,L chosen for model generation and variance strength P chosen for additional noise ($u[n]$) on the MSE obtained from validation data.

Table 4.3, 4.5, and 4.7, shows the model performance on validation data for three different test systems under different noisy cases. Table 4.4, 4.6, and 4.8 demonstrates model performance on testing data for estimating noise-free output. Figure 4.2, 4.3 and 4.4 shows the comparison of system noise-free output ($y[n]$) and model output ($z[n]$) for the three nonlinear systems with $u[n]$

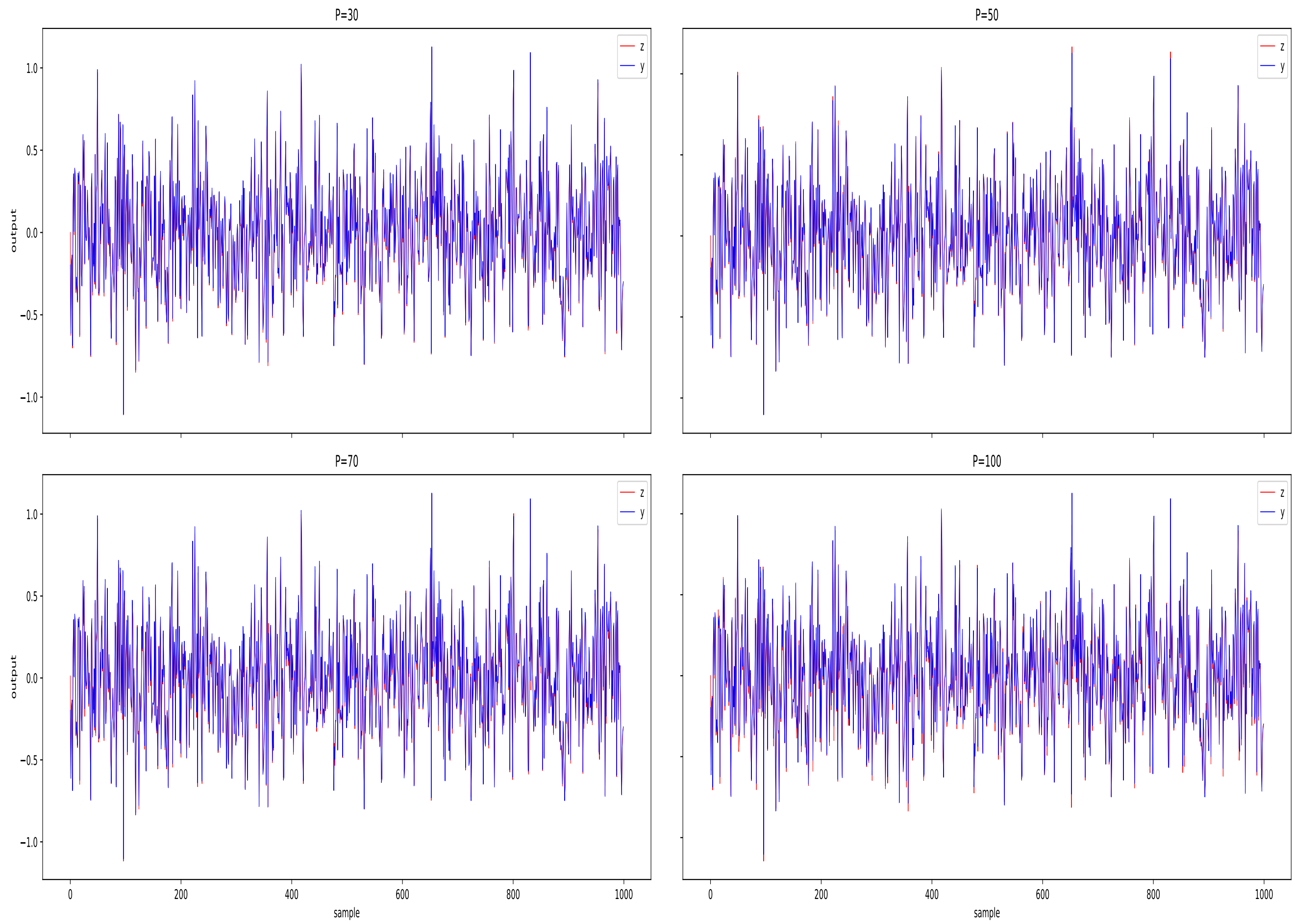

Figure 4.3: System Noise-free Output Vs. Model Output, System No.2

added. Apparently, compared with noise-free scenarios, higher $MSEs\%$ were achieved for noisy cases. $MSEs\%$ are consistently higher while P values climb from 30 to 100 in all three nonlinear systems, as observed through the result on validation data.

Different from noise-free scenario, the maximum K and L (e.g., [10,10]) are not always the best chosen case in noisy scenario. In System No.1 and No.2 where maximum cross-product order is 2, the best chosen [K, L] is smaller while P value is getting larger, as observed in Table 4.3 and 4.5. In System No.3 where maximum cross-product order is 3, best performance were achieved with smaller

Table 4.7: MSE % of Noisy Case for Validation, System No.3

K, L	$P = 30$	$P = 50$	$P = 70$	$P = 100$
[10,10]	6.48	13.42	21.96	35.47
[10,7]	6.48	**12.60**	22.34	35.01
[7,10]	6.66	13.15	21.70	34.15
[7,7]	6.49	13.04	22.30	33.37
[7,5]	6.44	13.04	22.01	33.26
[5,7]	6.49	13.21	22.28	33.12
[5,5]	6.52	13.16	22.03	32.58
[5,3]	6.50	13.16	22.44	32.19
[3,5]	**6.10**	13.32	**21.20**	31.47
[3,3]	6.11	13.29	21.68	**30.95**

Table 4.8: MSE % of Noise-free Output Estimation for Testing, System No.3

	$P = 30$	$P = 50$	$P = 70$	$P = 100$
Chosen K,L	$[3, 5]$	$[10, 7]$	$[3, 5]$	$[3, 3]$
MSE %	3.52	3.81	4.30	3.51

[K, L] values. As shown in Table 4.7, in three out of four different P cases, best performances were achieved under [K,L] value of $[3, 5]$ and $[3, 3]$. Therefore, we conclude that when test system is more noisy (higher p value), candidate pool with smaller maximum delay [K, L] is more likely to achieve the better performance (lower $MSE\%$). To better approximate the noise-free output on the testing data, smaller P value of additional noise $u[n]$is required. It is more challenging to approximate the noise-free output in the noisy case than in the noise-free case.

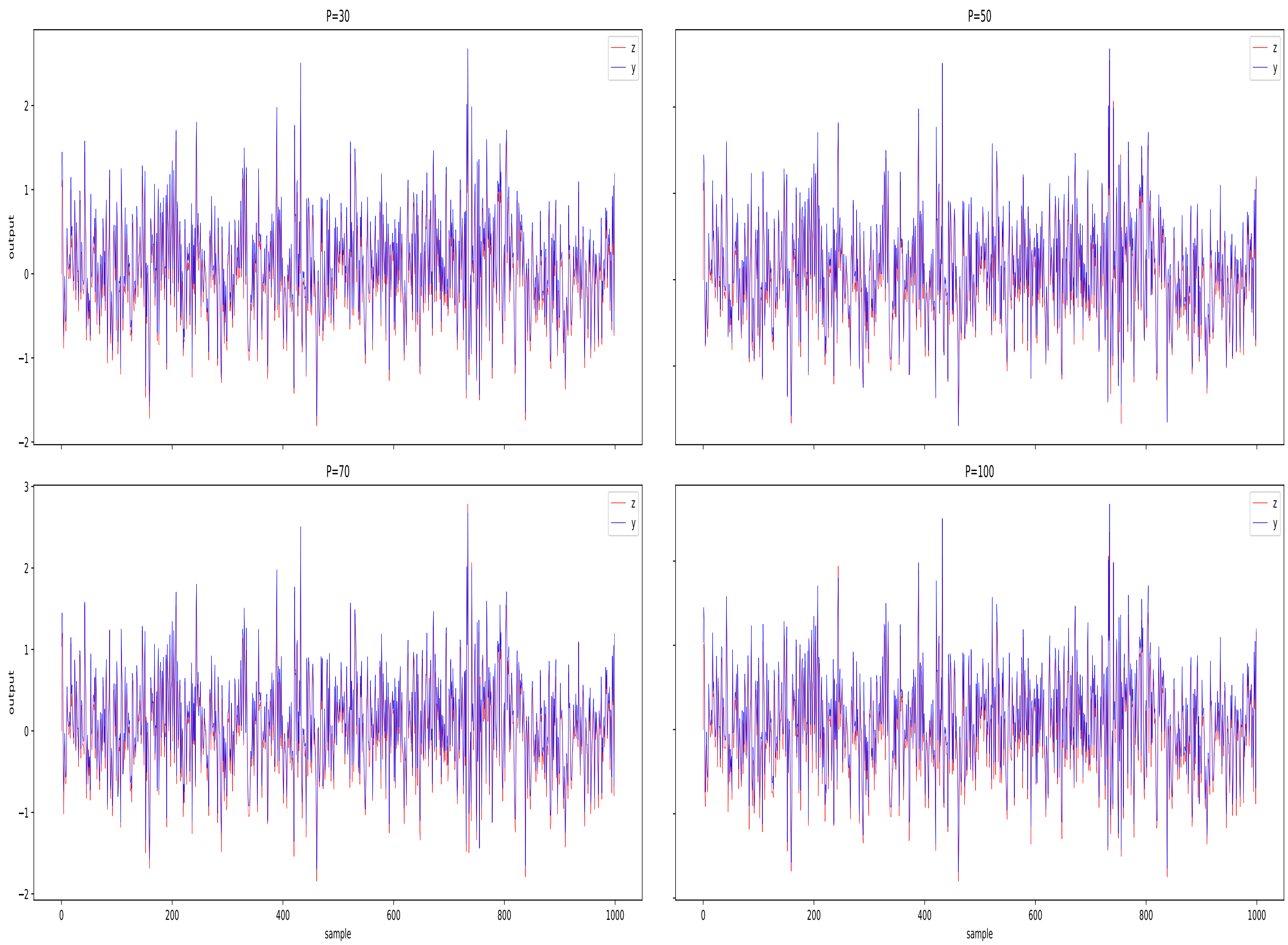

Figure 4.4: System Noise-free Output Vs. Model Output, System No.3

Chapter 5

Conclusion and Code Summary

5.1 Conclusion

This paper implements FOS algorithm for three different nonlinear system under both noise-free and noisy cases. To evaluate the performance of FOS model, we conduct several experiments through training, validation and testing phases. We show that FOS algorithm perform very well in nonlinear system modeling. In the noise-free case, model achieves the better result with larger maximum delay [K, L]. However, in the noisy cases, model achieves the better result with smaller maximum delay [K, L]. Better $MSEs\%$ are achieved with the decrease of noise (u[n] with lower P value). When the cross-product is higher, the $MSEs\%$ is higher especially during the higher level of noise ($u[n]$ with higher P value). Different level of noise has small impact on approximation of noise-free output. However, in the most cases, model performance is always higher with the smaller P chosen on the testing data. Overall, we still achieve considerable result on noise-free output even we trained our FOS model on noisy data.

5.2 Code Summary

The code consists of nonlinear data generation, candidates pool generation, FOS algorithm and experiment procedure.

5.2.1 Nonlinear Data Generation (*nonlinear_data_generation.py*)

It implements the generation of input data x[n], as well as three different nonlinear systems under both noise-free $y[n]$ and noisy $v[n]$ cases.

5.2.2 Candidates Pool Generation (*candidates_generation.py*)

It implements generation of different candidate sets with different sets of maximum delay on x-terms and y-terms. This candidate sets will be further used as the input of FOS model.

5.2.3 FOS algorithm (*FOS.py*)

It implements the FOS algorithm and calculate the coefficient list a, selected index list Idx, selected candidate pool P and MSE value MSE

5.2.4 Experiment Procedure (*main.py*)

It shows the experiment flow from training phase, validation phase to testing phase, under both noise-free and different noisy scenarios.

Bibliography

[1] M. J. Korenberg and L. D. Paarmann, "Applications of fast orthogonal search: Time-series analysis and resolution of signals in noise," *Annals of biomedical engineering*, vol. 17, no. 3, pp. 219–231, 1989.

[2] F. Mobasser, J. M. Eklund, and K. Hashtrudi-Zaad, "Estimation of elbow-induced wrist force with emg signals using fast orthogonal search," *IEEE transactions on biomedical engineering*, vol. 54, no. 4, pp. 683–693, 2007.

[3] K. H. Chon, "Accurate identification of periodic oscillations buried in white or colored noise using fast orthogonal search," *IEEE transactions on biomedical engineering*, vol. 48, no. 6, pp. 622–629, 2001.

[4] K. H. Chon, M. J. Korenberg, and N. H. Holstein-Rathlou, "Application of fast orthogonal search to linear and nonlinear stochastic systems," *Annals of biomedical engineering*, vol. 25, no. 5, pp. 793–801, 1997.

[5] K. M. Adeney and M. J. Korenberg, "Iterative fast orthogonal search algorithm for mdl-based training of generalized single-layer networks," *Neural Networks*, vol. 13, no. 7, pp. 787–799, 2000.

[6] M. J. Korenberg and L. D. Paarmann, "Orthogonal approaches to time-series analysis and system identification," *IEEE Signal Processing Magazine*, vol. 8, no. 3, pp. 29–43, 1991.

```python
# -*- coding: utf-8 -*-
"""
Created on Tue Apr 9 15:15:50 2020
@author: Patrick
"""

"""Implementation of Fast Orthogonal Search"""

#==============================================
#nonlinear_data_generation.py
#==============================================

import numpy as np
from matplotlib import pyplot as plt
from FOS import FOS


def Nonlinear_Generation(mu, sigma, x, y, P, case_index, noise):
    # case 1:
    if case_index ==1:   # 2nd order
        [a0, a1, a2, a3, a4, a5, a6] = [0.05, 0.4, 0.1, -0.2, -0.1, 0.33, 0.0]

    # case 2:
    elif case_index ==2: # 2nd order
        [a0, a1, a2, a3, a4, a5, a6] = [0.01, 0.2, 0.3, -0.1, 0.05, 0.2, 0.0]

    # case 3:
    else:                # 3rd order
        [a0, a1, a2, a3, a4, a5, a6] = [0.1, 0.1, 0.5, -0.3, 0.22, -0.4, 0.1]

    for n in range(2, len(y)):
        y[n] =  a0 + a1*y[n-1]+ a2*x[n-1]+ a3*x[n]*x[n-2]+ a4*y[n-1]*y[n-2]
        +a5*x[n-2]*y[n-2] + a6*x[n-1]*x[n-2]*y[n-2]
    yn = y + P*np.var(y)* np.expand_dims(np.random.normal(mu, sigma,len(x)), axis=1)

    if noise==True:
        y = yn
    else:
        y = y

    return y

#
```

```python
# -*- coding: utf-8 -*-
"""
Created on Tue Apr 9 15:15:50 2020
@author: Patrick
"""

"""Implementation of Fast Orthogonal Search"""

#=============================================
#candidates_generation.py
#=============================================

def CandidatePool_Generation(x_train, y_train, K, L):
    Candidates = []
    # x[n-l], l = 0,...,10 (11 Candidates)
    for l in range(0, L+1):
        zero_list = l * [0]
        data_list = x_train[:len(x_train)-l]
        xn_l = [*zero_list, *data_list]
        Candidates.append(xn_l)

    # y[n-k], l = 1,...,10 (10 Candidates)
    for k in range(1, K+1):
        zero_list = k * [0]
        data_list = y_train[:len(y_train)-k]
        yn_k = [*zero_list, *data_list]
        Candidates.append(yn_k)

    # x[n-l1]x[n-l2], l1 = 0,...,10
     # l2 = l1,...,10 (66 Candidates)
    for l1 in range(0, L+1):
        for l2 in range(l1, L+1):
            zero_list_l1 = l1 * [0]
            zero_list_l2 = l2 * [0]
            data_list_l1 = x_train[:len(x_train)-l1]
            data_list_l2 = x_train[:len(x_train)-l2]
            xn_l1 = [*zero_list_l1, *data_list_l1]
            xn_l2 = [*zero_list_l2, *data_list_l2]

            Candidates.append([i*j for i,j in zip(xn_l1,xn_l2)])

    # y[n-l1]y[n-l2], k1 = 1,...,10
    #  k2 = k1,...,10 (55 Candidates)
    for k1 in range(1, K+1):
        for k2 in range(k1, K+1):
            zero_list_k1 = k1 * [0]
            zero_list_k2 = k2 * [0]
            data_list_k1 = y_train[:len(y_train)-k1]
            data_list_k2 = y_train[:len(y_train)-k2]
            yn_k1 = [*zero_list_k1, *data_list_k1]
            yn_k2 = [*zero_list_k2, *data_list_k2]

            Candidates.append([i*j for i,j in zip(yn_k1,yn_k2)])
```

```
    # x[n-l]y[n-k], l = 1,...,10
    #  k = 1,...,10 (110 Candidates)
    for l in range(0, L+1):
        for k in range(1, K+1):
            zero_list_l = l * [0]
            zero_list_k = k * [0]
            data_list_l = x_train[:len(x_train)-l]
            data_list_k = y_train[:len(y_train)-k]
            xn_l = [*zero_list_l, *data_list_l]
            yn_k = [*zero_list_k, *data_list_k]

            Candidates.append([i*j for i,j in zip(xn_l,yn_k)])

    return Candidates

#

```

```python
# -*- coding: utf-8 -*-
"""
Created on Tue Apr 9 15:15:50 2020
@author: Patrick
"""

"""Implementation of Fast Orthogonal Search"""

#==============================================
#FOS.py
#==============================================

import numpy as np
from tqdm import tqdm


def FOS(CandidatePool, y, N0, Noise):

    N, M = CandidatePool.shape[0],CandidatePool.shape[1]
    P = np.ones((N, 1))  # Selected Candidates Pool
    D = np.zeros((M+1, M+1))
    C = np.zeros((M+1,))
    alpha = np.zeros((M+1, M))

    # Parameters Initialization
    D[0,0] = 1
    C[0] = np.mean(y)
    g = C[0]/D[0,0]
    Q = C[0]*C[0]/D[0,0]


    Idx = np.empty((0,1)) # Selected Index List with max Q
    for m in tqdm(range(1, M+1)):

        Qm  = np.empty((0,1))
        for i in range(1, M+1):
            pm = CandidatePool[:,i-1]
            pm = np.expand_dims(pm, axis=1)
            D[m,0] = np.mean(pm)

            if m == 1:
                alpha[1,0] = D[1,0]/D[0,0]
                SigmaD = alpha[1,0]* D[1,0]
                D[1,1] = np.mean(pm*pm)-SigmaD
                SigmaC = alpha[1,0]*C[0]
            else:
                SigmaC = 0
                for r in range(0, m):
                    alpha[m,r] = D[m,r]/D[r,r]

                    SigmaD = 0
                    for j in range(0, r+1):
                        SigmaD = SigmaD + alpha[r+1,j]* D[m,j]
```

```
                    if r < m-1:
                        P_update = np.expand_dims(P[:,r+1],axis=1)
                        D[m,r+1] = np.mean(pm*P_update)-SigmaD
                    else:
                        D[m,m] = np.mean(pm*pm)-SigmaD
                    SigmaC = SigmaC + alpha[m,r]*C[r]
            C[m] = np.mean(y*pm) - SigmaC
            if D[m,m] < np.exp(-30):
                Qt = 0
            else:
                Qt = C[m]*C[m]/D[m,m]
            Qm = np.append(Qm, Qt)

        [Qmax,Idxm] = [np.max(Qm), np.argmax(Qm)]

        # Stopping criterion value
        criterion_value = (4/(N-N0+1))* (np.mean(y*y)-np.sum(Q))
        if (Noise==True) and (Qmax < criterion_value): # Only while noisy data!
            print('break')
            break
        else: # Continue after model term pm[n] is selected
            pm  = CandidatePool[:,Idxm]
            pm  = np.expand_dims(pm, axis=1)
            Idx = np.append(Idx, Idxm)
            P   = np.append(P, pm, axis=1)

            D[m,0] = np.mean(pm)
            if m == 1:
                alpha[1,0] = D[1,0]/D[0,0]
                SigmaD = alpha[1, 0]* D[1,0]
                D[1,1] = np.mean(pm*pm)-SigmaD
                SigmaC = alpha[1,0]*C[0]
            else:
                SigmaC = 0
                for r in range(0, m): # r = 0,...m-1
                    alpha[m,r] = D[m,r]/D[r,r]
                    SigmaD = 0
                    for j in range(0, r+1): # for j = 0:r
                        SigmaD = SigmaD + alpha[r+1,j]* D[m,j] #
                        P_update = np.expand_dims(P[:,r+1],axis=1)
                        D[m,r+1] = np.mean(pm*P_update)-SigmaD
                    else:
                        D[m,m] = np.mean(pm*pm)-SigmaD

                    SigmaC = SigmaC + alpha[m,r]*C[r]
            C[m] = np.mean(y*pm) - SigmaC
            # Ensure D[m,m] exceeds a specified positive threshold level.
            if D[m,m] > np.exp(-30):
                Q = np.append(Q, C[m]*C[m]/D[m,m])
            else:
                continue
            g = np.append(g, C[m]/D[m,m])
```

```
   # Mean Squared Error Calculation
    MSE = np.mean(y*y)-np.sum(Q)

    # Coefficient a calculation
    a = np.empty((0,1))
    m = g.shape[0]-1

    for i in range(0,m):
        v = np.zeros((m+1,1))
        v[i] = 1
        for m in range(i+1, m+1):
            Vi = 0
            for r in range(i, m):
                Vi = Vi + alpha[m,r]*v[r]
            v[m] = -Vi
        Am = 0
        for j in range(i,m+1):
            Am = Am + g[j]*v[j]
        a = np.append(a, Am)
    a = np.append(a, g[m])

    return a, MSE, Idx, P

#

```

```python
# -*- coding: utf-8 -*-
"""
Created on Tue Apr 9 15:15:50 2020
@author: Patrick
"""

"""Implementation of Fast Orthogonal Search"""

#==============================================
#main.py
#==============================================

import numpy as np
import copy
from matplotlib import pyplot as plt
from nonlinear_data_generation import Nonlinear_Generation
from candidates_generation import CandidatePool_Generation
from FOS import FOS
from multiprocessing import Pool

#_________________________________Input Data__________________________________#
mu, sigma = 0, 1 # zero mean, 1.0 standard deviation
data_length = 3000
[train_length, val_length, test_length] = [1000, 1000, 1000]

x = np.random.normal(mu, sigma, data_length) # Random Gaussian Mean generation

y = np.zeros((data_length, 1))

p_array = np.asarray([0.3, 0.5, 0.7, 1.0]) # Different P values for noisy data
pred_list=[]
test_list=[]
free_list=[]

# for num in range(1,4):
# Three Difference Equations of Structure
for p_num in range(0, 4):
    # Nonlinear Data Generation
    # if noise-free
    # nonlinear_data = Nonlinear_Generation(mu, sigma, x, y, 0, num, False)
    # if noisy
    nonlinear_data = Nonlinear_Generation(mu, sigma, x, y, p_array[p_num], 1,True)
    noise_free_data =Nonlinear_Generation(mu, sigma, x, y, 0, 1, False)
    y = nonlinear_data
    # Obtain Training, Validation and Testing Data
    x_train = x[ :train_length]
    y_train = y[ :train_length]
    x_val   = x[train_length: train_length + val_length]
    y_val   = y[train_length: train_length + val_length]
    x_test  = x[train_length + val_length: ]
    y_test  = y[train_length + val_length: ]
    y_free  = noise_free_data[train_length + val_length: ]
```

```
#_______________________________Training Phase____________________________#
# 10 Different Combinations of K and L

KL_Comb = [[10,10], [10, 7], [7,10], [7,7], [7,5], [5,7], [5,5], [5,3],
[3,5], [3,3]]
KL_Comb = np.asarray(KL_Comb)

# Initialization
a_list   = []
MSE_list = []
Idx_list = []
P_list   = []


for i in range(KL_Comb.shape[0]):

    K, L = KL_Comb[i]
    N0 = np.max([K,L])

    CandidatePool = CandidatePool_Generation(x_train, y_train, K, L)
    CandidatePool = np.asarray(CandidatePool)
    CandidatePool = CandidatePool.T

    a, MSE,Idx,P = FOS(CandidatePool, y_train, N0, True)
    a_list.append(a)
    MSE_list.append(MSE)
    Idx_list.append(Idx)
    P_list.append(P)



#____________________________Validation Phase______________________________#
# 10 Different Combinations of K and L

MSE_val_list = []
Idx_list = np.asarray(Idx_list)

for i in range(KL_Comb.shape[0]):
    K, L = KL_Comb[i]
    N0 = np.max([K,L])

    CandidatePool = CandidatePool_Generation(x_val, y_val, K, L)
    CandidatePool = np.asarray(CandidatePool)
    CandidatePool = CandidatePool.T
    # Best Selected Candidates
    P_selected  = CandidatePool[:, Idx_list[i].astype(int)]
    Ones_Column = np.zeros((P_selected.shape[0], P_selected.shape[1]+1))
    Ones_Column[:,1:] = P_selected
    P_selected  = Ones_Column
    a_list[i] = np.asarray(a_list[i])
    y_pred = np.expand_dims(np.sum(P_selected* a_list[i], axis=1), axis=1)

    MSE_val = np.mean((y_val - y_pred)**2) / np.mean((y_val-
```

```
        np.mean(y_val))**2)*100
        MSE_val_list.append(MSE_val)

    MSE_val_list = np.asarray(MSE_val_list)
    Min_mse_index = np.argmin(MSE_val_list)

    #______________________________Testing Phase____________________________#

    K_test, L_test = KL_Comb[Min_mse_index] # Best Selected Model
    CandidatePool = CandidatePool_Generation(x_test, y_test, K_test, L_test)

    CandidatePool = np.asarray(CandidatePool)
    CandidatePool = CandidatePool.T

    # Best Selected Candidates
    P_selected  = CandidatePool[:, Idx_list[Min_mse_index].astype(int)]
    Ones_Column = np.zeros((P_selected.shape[0], P_selected.shape[1]+1))
    Ones_Column[:,1:] = P_selected
    P_selected  = Ones_Column

    a_list[Min_mse_index] = np.asarray(a_list[Min_mse_index])
    y_pred = np.expand_dims(np.sum(P_selected* a_list[Min_mse_index], axis=1), axis=1)
    # print(y_pred.shape)
    MSE_test = np.mean((y_free - y_pred)**2)/np.mean((y_free-np.mean(y_test))**2)*100

    pred_list.append(y_pred)
    test_list.append(y_test)
    free_list.append(y_free)


    print(MSE_val_list, "MSE_val_list",Min_mse_index, "Min_mse_index")
    print(MSE_test)


pred_list = np.asarray(pred_list)
test_list = np.asarray(test_list)

# fig, (ax1, ax2, ax3) = plt.subplots(1, 3)
# ax1.plot(pred_list[0], 'r', label='z')
# ax1.plot(test_list[0], 'b', label='y')
# ax1.legend('zy')
# ax1.set_title('System No.1')
# ax2.plot(pred_list[1], 'r', label='z')
# ax2.plot(test_list[1], 'b', label='y')
# ax2.legend('zy')
# ax2.set_title('System No.2')
# ax3.plot(pred_list[2], 'r', label='z')
# ax3.plot(test_list[2], 'b', label='y')
# ax3.legend('zy')
# ax3.set_title('System No.3')
# plt.show()
```

```
# Draw for noisy output

fig, axs = plt.subplots(2, 2)

axs[0, 0].plot(pred_list[0], 'r', label="z[n]")
axs[0, 0].plot(free_list[0], 'b', label="y[n]")
axs[0, 0].legend("zy")
axs[0, 0].set_title('P=30')
axs[0, 1].plot(pred_list[1], 'r', label="z[n]")
axs[0, 1].plot(free_list[1], 'b', label="y[n]")
axs[0, 1].legend("zy")
axs[0, 1].set_title('P=50')
axs[1, 0].plot(pred_list[2], 'r', label="z[n]")
axs[1, 0].plot(free_list[2], 'b', label="y[n]")
axs[1, 0].legend("zy")
axs[1, 0].set_title('P=70')
axs[1, 1].plot(pred_list[3], 'r', label="z[n]")
axs[1, 1].plot(free_list[3], 'b', label="y[n]")
axs[1, 1].legend("zy")
axs[1, 1].set_title('P=100')

for ax in axs.flat:
    ax.set(xlabel='sample', ylabel='output')

# Hide x labels and tick labels for top plots and y ticks for right plots.
for ax in axs.flat:
    ax.label_outer()

plt.show()



#

```