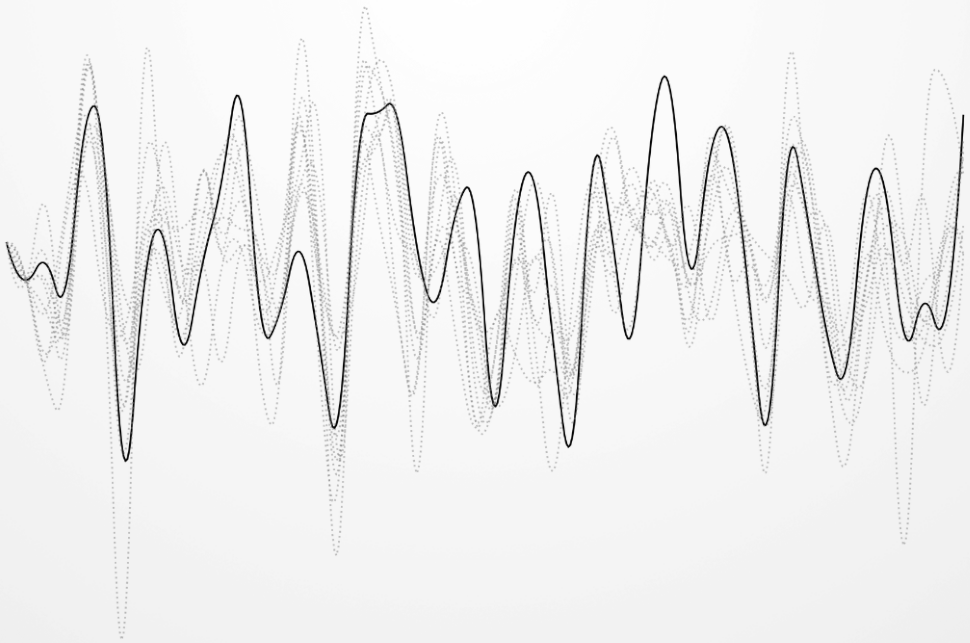
NONLINEAR SYSTEM IDENTIFICATION

THEORY AND PRACTICE WITH **SYSIDENTPY**



BY WILSON ROCHA

All the world is a nonlinear system

He linearised to the right

He linearised to the left

Till nothing was right

And nothing was left

> [Stephen A. Billings](Nonlinear System Identification: NARMAX Methods in the Time, Frequency, and Spatio-Temporal Domains | Wiley)

Nonlinear System Identification and Forecasting: Theory and Practice With SysIdentPy

Welcome to our companion book on System Identification! This book is a comprehensive approach to learning about dynamic models and forecasting. The main aim of this book is to describe a comprehensive set of algorithms for the identification, forecasting and analysis of nonlinear systems.

Our book is specifically designed for those who are interested in learning system identification and forecasting. We will guide you through the process step-by-step using Python and the SysIdentPy package. With SysIdentPy, you will be able to apply a range of techniques for modeling dynamic systems, making predictions, and exploring different design schemes for dynamic models, from polynomial to neural networks. This book is for graduates, postgraduates, researchers, and for all people from different research areas who have data and want to find models to understand their systems better.

The research literature is filled with books and papers covering various aspects of nonlinear system identification, including NARMAX methods. In this book, our objective isn't to replicate all the numerous algorithm variations available. Instead, we want to show you how to model your data using those algorithms with SysIdentPy. We'll mention all the specific details and different versions of the algorithms in the book, so if you're more interested in the theoretical aspects, you can explore those ideas further. We aim to focus on the fundamental techniques, explaining them in straightforward language and showing how to use them in real-world situations. While there will be some math and technical details involved, the aim is to keep it as easy to understand as possible. In essence, this book aims to be a resource that readers from various fields can use to learn how to model dynamic nonlinear systems.

The best part about our book is that it is open source material, meaning that it is freely available for anyone to use and contribute to. We hope this brings together people who share interest for system identification and forecasting techniques, from linear to nonlinear models.

So, whether you're a student, researcher, data scientist or practitioner, we invite you to share your knowledge and contribute with us. Let’s explore system identification and forecasting with **SysIdentPy**!

To follow along with the Python examples in the book, you’ll need to have some packages installed. We’ll cover the main ones here and let you know if any additional packages are required as we proceed.

```
import sysidentpy
import pandas as pd
import numpy as np
import torch
import matplotlib
import scipy
```

About the Author

Wilson Rocha is the Head of Data Science at RD Saúde and the creator of the SysIdentPy library. He holds a degree in Electrical Engineering and a Master's in Systems Modeling and Control, both from Federal University of São João del-Rei (UFSJ), Brazil. Wilson began his journey in Machine Learning by developing soccer-playing robots and continues to advance his research in the fields of Multi-objective Nonlinear System Identification and Time Series Forecasting.

Connect with Wilson Rocha through the following social networks:

- LinkedIn
- ResearchGate
- Discord

Referencing This Book

If you find this book useful, please cite it as follows:

```
Lacerda Junior, W.R. (2024). *Nonlinear System Identification and Forecasting:
Theory and Practice With SysIdentPy*. Web version. https://sysidentpy.org
```

If you use SysIdentPy on your project, please drop me a line.

If you use SysIdentPy on your scientific publication, we would appreciate citations to the following paper:

- Lacerda et al., (2020). SysIdentPy: A Python package for System Identification using NARMAX models. Journal of Open Source Software, 5(54), 2384, https://doi.org/10.21105/joss.02384

```
@article{Lacerda2020,
  doi = {10.21105/joss.02384},
  url = {https://doi.org/10.21105/joss.02384},
  year = {2020},
  publisher = {The Open Journal},
  volume = {5},
  number = {54},
  pages = {2384},
  author = {Wilson Rocha Lacerda Junior and Luan Pascoal Costa da Andrade and Samuel Carlos Pessoa Oliveira and Samir Angelo Milani Martins},
  title = {SysIdentPy: A Python package for System Identification using NARMAX models},
  journal = {Journal of Open Source Software}
}
```

PDF, Epub and Mobi version

Download the pdf version of the book: pdf version{:download="Nonlinear System Identification and Forecasting: Theory and Practice With SysIdentPy"}

Download the epub version of the book: epub version{:download="Nonlinear System Identification and Forecasting: Theory and Practice With SysIdentPy"}

Download the mobi version of the book: mobi version{:download="Nonlinear System Identification and Forecasting: Theory and Practice With SysIdentPy"}

Acknowledgments

The System Identification class taught by Samir Martins (in Portuguese) has been a great source of inspiration for this series. In this book, we will explore Dynamic Systems and learn how to master NARMAX models using Python and the SysIdentPy package. The Stephen A. Billings book, Nonlinear System Identification: NARMAX Methods in the Time, Frequency, and Spatio - Temporal Domains, have been instrumental in showing us how powerful System Identification can be.

In addition to these resources, we will also reference Luis Antônio Aguirre Introdução à Identificação de Sistemas. Técnicas Lineares e não Lineares Aplicadas a Sistemas. Teoria e Aplicação (in

Portuguese), which has proven to be an invaluable tool in introducing complex dynamic modeling concepts in a straightforward way. As an open source material on System Identification and Forecasting, this book aims to provide a accessible yet rigorous approach to learning dynamic models and forecasting.

Support the Project

The **Nonlinear System Identification and Forecasting: Theory and Practice With SysIdentPy** is an extensive open-source resource dedicated to the science of System Identification. Our goal is to make this knowledge accessible to everyone, both financially and intellectually.

If this book has been valuable to you, and you'd like to support our efforts, we welcome financial contributions through our Sponsor page.

If you're not in a position to contribute financially, you can still support by helping us improve the book. We encourage you to report any typos, suggest edits, or provide feedback on sections that you found challenging. You can do this by visiting the book's repository and opening an issue. Additionally, if you enjoyed the content, please consider sharing it with others who might benefit from it, and give us a star on GitHub.

Your support, in any form, helps us continue to enhance this project and maintain a high-quality resource for the community. Thank you for your contribution!

0.1 - Contents

Contents

1. Least Squares
2. Total Least Squares
3. Recursive Least Squares
4. Least Mean Squares
5. Extended Least Squares Algorithms

4. Model Structure Selection
    1. Introduction
    2. The Forward Regression Orthogonal Least Squares
        1. Case Study
    3. Information Criteria
        1. Overview of the Information Criteria Methods
            1. AIC
            2. AICc
            3. BIC
            4. LILC
            5. FPE
    4. Meta Model Structure Selection (MetaMSS)
        1. Meta-heuristics
        2. Standard Particle Swarm Optimization (PSO)
        3. Standard Gravitational Search Algorithm (GSA)
        4. The Binary Hybrid Optimization Algorithm
        5. Meta-Model Structure Selection (MetaMSS): Building NARX for Regression
        6. Case Studies: Simulation Results
        7. MetaMSS vs FROLS
        8. Meta-MSS vs RJMCMC
        9. MetaMSS algorithm using SysIdentPy
    5. Accelerated Orthogonal Least Squares
    6. Entropic Regression
5. Multiobjective Parameter Estimation
    1. Introduction
    2. Multi-objective optimization problem
    3. Pareto Optimal Definition and Pareto Dominance
    4. Affine Information Least Squares Algorithm
    5. Case Study - Buck converter
6. Multiobjective Model Structure Selection
    1. Introduction
    2. Multiobjective Error Reduction Ratio
    3. Multiobjective Meta Model Structure Selection
    4. Case Studies

18. Monthly Production of Milk (coming soon)
19. Half-hourly Electricity Demand in England and Wales (coming soon)
20. Daily Temperature in Melbourne (coming soon)
21. Weekly U.S. Product Supplied of Finished Motor Gasoline (coming soon)
22. Australian Total Wine Sales (coming soon)
23. Quarterly Production of Woollen Yarn in Australia (coming soon)
24. Hourly Nuclear Energy Generation (coming soon)

1 - Introduction

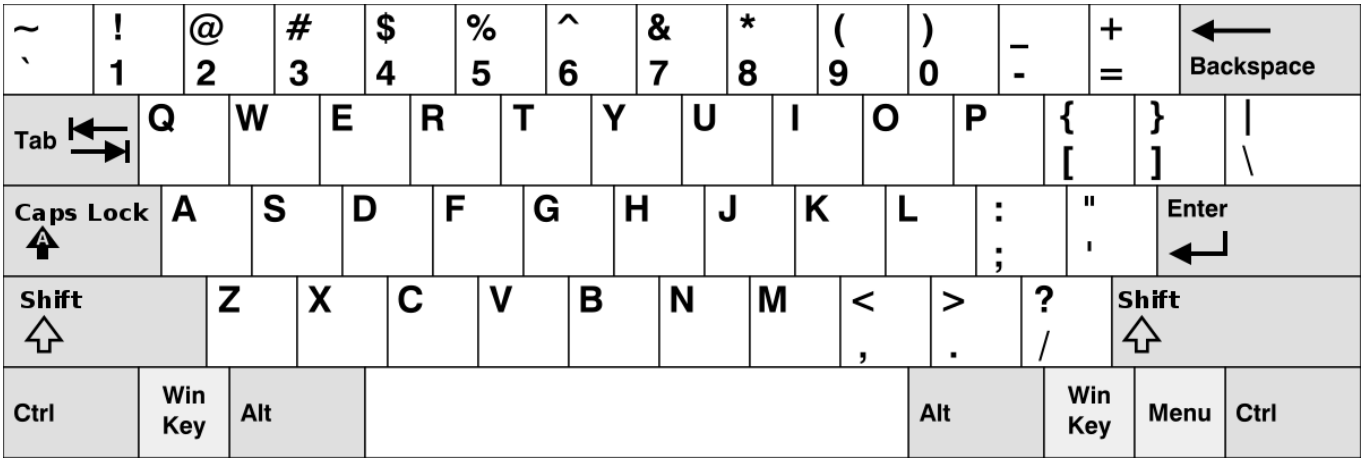
Introduction

The concept of a mathematical model is fundamental in many fields of science. From engineering to sociology, models plays a central role to the study of complex systems as they allow to simulate what will happen in different scenarios and conditions, predict its output for a given input, analyse its properties and explore different design schemes. To accomplish these goals, however, it is crucial that the model is a proper representation of the system under study. The modeling of dynamic and steady-state behaviors is, therefore, fundamental to this type of analysis and depends on System Identification (SI) procedures.

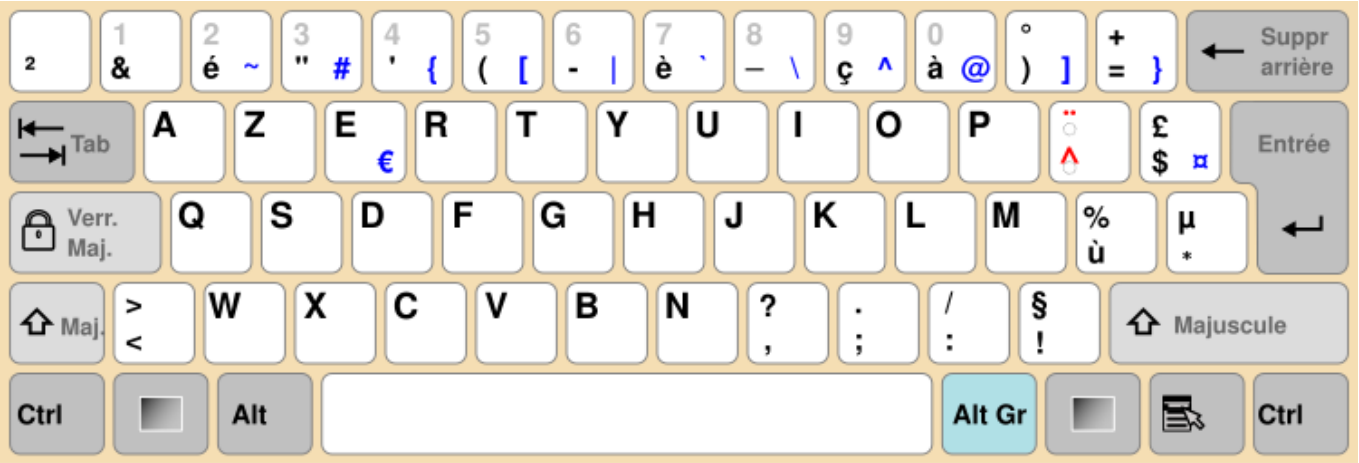
Models

Mathematical modeling is a great way to understand and analyze different parts of our world. It gives us a clear framework to make sense of complex systems and how they behave. Whether it's for everyday tasks or big-picture issues like disease control, models are a key part of how we deal with various challenges.

Typing efficiently on a conventional QWERTY keyboard layout is the result of a well-learned model of the QWERTY keyboard embedded in the individual cognitive processes. However, if you are faced with a different keyboard layout, such as the Dvorak or AZERTY, you will probably struggle to adapt to the new model. The system changed, so you will have to update you *model*.



> QWERTY - Wikipedia - ANSI QWERTY keyboard layout (US)



> AZERTY layout used on a keyboard

Mathematical modeling touches on many parts of our lives. Whether we're looking at economic trends, tracking how diseases spread, or figuring out consumer behavior, models are essential tools for gaining knowledge, making informed decisions, and take control over complex systems.

In essence, mathematical models help us make sense of the world. They let us understand human behavior and the systems we deal with every day. By using these models, we can learn, adapt, and adjust our strategies to keep up with the changes around us.

System Identification

System identification is a data-driven framework to model dynamical systems. Initially, scientists focused on linear system identification, but this has been changing over the past decades with more emphasis in nonlinear systems. Nonlinear system identification is widely considered to be one of the most important topics concerning the modeling of many different dynamical systems, from time-series to severally nonlinear dynamic behaviors.

Extensive resources, including excellent textbooks covering linear system identification and time series forecasting are readily available. In this book, we revisit some known topics, but we also try to approach such subjects in a different and complementary way. We will explore the modeling of nonlinear dynamic systems using NARMAX(Nonlinear AutoRegressive Moving Average model with eXogenous inputs) methods, which were introduced by [Stephen A. Billings] in 1981.

Linear or Nonlinear System Identification

Linear Models

While most real world systems are nonlinear, you probably should give linear models a try first. Linear models usually serves as a strong baseline and can be good enough for your case, giving satisfactory performance. Astron and Murray and Glad and Ljung showed that many nonlinear systems can be

well described by locally linear models. Besides, linear models are easy to fit, easy to interpret, and requires less computational resources than nonlinear models, allowing you to experiment fast and gather insights before thinking about gray box models or complex nonlinear models.

Linear models can be very useful, even in the presence of strong nonlinearities, because it is much easier to deal with it. Moreover, the development of linear identification algorithms is still a very active and healthy research field, with many papers being released every year Sai Li, Linjun Zhang, T. Tony Cai & Hongzhe Li, Maria Jaenada, Leandro Pardo, Xing Liu; Lin Qiu, Youtong Fang; Kui Wang; Yongdong Li, Jose Rodríguez, Alessandro D'Innocenzo and Francesco Smarra. Linear models work well most of the time and should be the first choice for many applications. However, when dealing with complex systems where linear assumptions don't hold, nonlinear models become essential.

Nonlinear Models

When linear models do not perform well enough, you should consider nonlinear models. It's important to notice, however, that changing from a linear to a nonlinear model is not always a simple task. For inexperienced users, it's common to build nonlinear models that performs worse than the linear ones. To work with nonlinear models, you must consider that characteristics such structural errors, noise, operation point, excitation signals and many others aspects of your system under study impact your modelling approach and strategy.

> As suggested by Johan Schoukens and Lennart Ljung in "Nonlinear System Identification - A User-Oriented Roadmap", only start working with nonlinear models if there is enough evidence that linear models will not solve the problem.

Nonlinear models are more flexible than linear models and can be built using many different mathematical representations, such as polynomial, generalized additive, neural networks, wavelet and many more (Billings, S. A.). Such flexibility, however, makes nonlinear system identification much more complex the linear ones, from the experiment design to the model selection. The user should consider that, besides the modeling complexity, moving to nonlinear models will require a revision in the road-map and computational resources defined when dealing with the linear models. In this respect, always ask yourself whether the potential benefits of nonlinear models are worth the effort.

NARMAX Methods

NARMAX model is one of the most frequently employed nonlinear model representation and is widely used to represent a broad class of nonlinear systems. NARMAX methods were successfully applied in many scenarios, which include industrial processes, control systems, structural systems, economic and financial systems, biology, medicine, social systems, and much more. The NARMAX model representation and the class of systems that can be represented by it will be discussed later in the book.

The main steps involved to build NARMAX models are (Billings, 2013):

1. Model Representation: define the model mathematical representation.
2. Model Structure Selection: define which terms are in the final model.
3. Parameter Estimation: estimate the coefficients of each model term selected in step 1.
4. Model validation: to make sure the model is unbiased and accurate;
5. Model Prediction/Simulation: predict future outputs or simulate the behavior of the system given different inputs.
6. Analysis: understanding the dynamical properties of the system under study
7. Control: develop control design schemes based on the obtained model.

Model Structure Selection (MSS) is the most important aspect of NARMAX methods and the most complex. Selecting the model terms is fundamental if the goal of the identification is to obtain models that can reproduce the dynamics of the original system and impacts every other aspect of the identification process. Problems related to overparameterization and numerical ill-conditioning are typical because of the limitations the identification algorithms in selecting the appropriate terms that should compose the final model (L. A. Aguirre e S. A. Billings, L. Piroddi e W. Spinelli).

> In SysIdentPy, you are allowed to interact directly with every item described in the 7 steps, except for the control one. SysIdentPy focuses on modeling, not on control design. You'll have to use some of the code bellow in every modeling task using SysIdentPy. You'll learn the details along the book, so don't worry if you are not familiar with those methods yet.

```
from sysidentpy.basis_function import Polynomial
from sysidentpy.neural_network import NARXNN
from sysidentpy.general_estimators import NARX
from sysidentpy.model_structure_selection import FROLS
from sysidentpy.parameter_estimation import RecursiveLeastSquares
from sysidentpy.metrics import root_relative_squared_error
from sysidentpy.simulation import SimulateNARMAX
```

What is the Purpose of System Identification?

Because of the Model Structure Selection problem, Billings, S. A. states that the goal of System Identification using NARMAX methods is twofold: performance and parsimony.

The first goal is often about approximation. Here, the main focus is to build a model that make predictions with with the lowest error possible. This approach is common in applications like weather forecasting, demand forecasting, predicting stock prices, speech recognition, target tracking, and pattern classification. In these cases, the specific form of the model isn't as critical. In other words,

how the terms interact (in parametric models), the mathematical representation, the static behavior and so on are not that important; what matters most is finding a way to minimize prediction errors.

But system identification isn't just about minimizing prediction errors. One of the main goals of System Identification is to build models that help the user to understand and interpret the system being modeled. Beyond just making accurate predictions, the goal is to develop models that truly capture the dynamic behavior of the system being studied, ideally in the simplest form possible. Science and engineering are all about understanding systems by breaking down complex behaviors into simpler ones that we can understand and control. For example, if the system's behavior can be described by a simple first-order dynamic model with a cubic nonlinear term in the input, system identification should help uncover that.

Is System Identification Machine Learning?

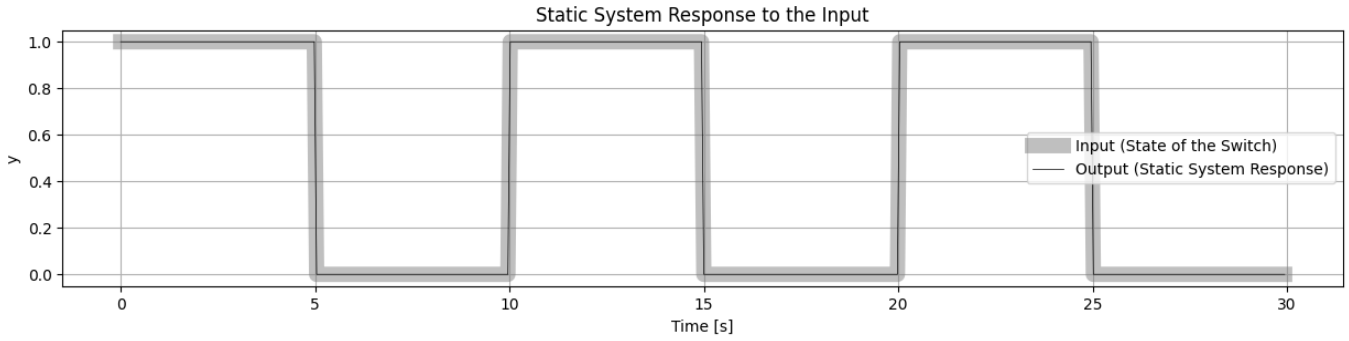
First, let's take an overview of static and dynamic systems. Imagine you have an electric guitar connected to an effect processor that can apply various audio effects, such as reverb or distortion. The effect is controlled by a switch that toggles between "on" and "off". Let’s consider this from the perspective of signals. The input signal represents the state of the effect switch: switch off (low level), switch on (high level). If we represent the guitar signal, we have a binary condition: effect off (original guitar sound), effect on (modified guitar sound). This is an example of a static system: the output (guitar sound) directly follows the input (state of the effect switch).

When the effect switch is off, the output is just the clean, unaltered guitar signal. When the effect switch is on, the output is the guitar signal with the effect applied, such as amplification or distortion. In this system, the effect being on or off directly influences the guitar signal without any delay or additional processing.

This example illustrates how a static system operates with binary control inputs, where the output directly reflects the input state, providing a straightforward mapping between the control signal and the system’s response.

```
import numpy as np
import matplotlib.pyplot as plt
from scipy import signal

t = np.linspace(0, 30, 500, endpoint=False)
u = signal.square(0.2*np.pi * t)
u[u < 0] = 0
# In a static system, the output y directly follows the input u
y = u

# Plot the input and output
plt.figure(figsize=(15, 3))
plt.plot(t, u, label='Input (State of the Switch)', color="grey", linewidth=10,
alpha=0.5)
```

```
plt.plot(t, y, label='Output (Static System Response)', color='k', linewidth=0.5)
plt.title('Static System Response to the Input')
plt.xlabel('Time [s]')
plt.ylabel('y')
plt.legend()
plt.grid(True)
plt.show()
```

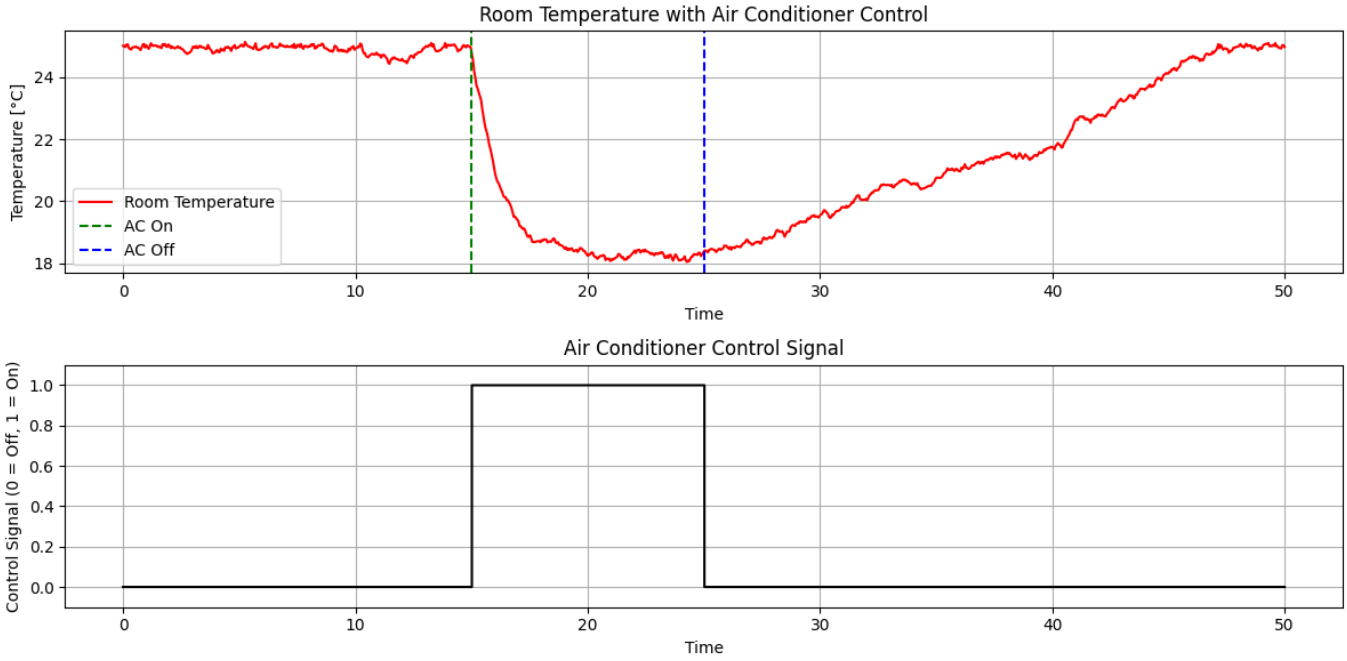


> Static response representation. The input signal representing the state of the switch (switch off (low level), switch on (high level)), and the static response: original sound (low level), processed sound (high level).

Now, let's consider a dynamic system: using an air conditioner to lower the room temperature. This example effectively illustrates the concepts of dynamic systems and how their output responds over time.

Let's imagine this from the perspective of signals. The input signal represents the state of the air conditioner's control: turning the air conditioner on (high level) or turning it off (low level). When the air conditioner is turned on, it begins to cool the room. However, the room temperature does not drop instantaneously to the desired cooler level. It takes time for the air conditioner to affect the temperature, and the rate at which the temperature decreases can vary based on factors like the room size and insulation.

Conversely, when the air conditioner is turned off, the room temperature does not immediately return to its original ambient temperature. Instead, it gradually warms up as the cooling effect diminishes.



> Using an air conditioner to lower the room temperature as dynamic system representation.

In this dynamic system, the output (room temperature) does not instantly follow the input (state of the air conditioner) because there is a time lag involved in both cooling and warming processes. The system has memory, meaning the current room temperature depends not only on the current state of the air conditioner but also on how long it has been running or off, and how much it has already cooled or allowed the room to warm up.

This example highlights the nature of dynamic systems: the response to an input is gradual and affected by the system's internal dynamics. The air conditioner's effect on the room temperature exemplifies how dynamic systems have a time-dependent response, where the output changes over time and does not immediately match the input signal.

For static systems, the output is a direct function of the input, represented by an algebraic equation:

$$y(t) = G \cdot u(t)$$

For dynamic systems, the output depends on the input and the rate of change of the input, represented by a differential equation. For example, the output $y(t)$ can be modeled as:

$$y(t) = G \cdot u(t) - \tau \cdot \frac{dy(t)}{dt}$$

Here, G is the gain, and τ is a constant that incorporates the system's memory. For discrete-time systems, we consider signals at specific and spaced intervals. The differential equation is discretized, and the derivative is approximated by a finite difference:

$$y[k] = \alpha y[k-1] + \beta u[k]$$

where α and β are constants that determine the system's response. The z-transform can be used to obtain the transfer function in the z-domain.

In summary, static systems are modeled by algebraic equations, while dynamic systems are modeled by differential equations.

> As Luis Antonio Aguirre states in one of his classes on Youtube (in Portuguese), **all physical systems are dynamic, but depending on the time scale, they can be modeled as static for simplification**. For example, the transition between the effects on the guitar sound, if taken in seconds (as we did in the example), could be treated as static depending on your analysis. However, the pedal board have components like capacitors, which are dynamic electrical components, making it a dynamic system. The response, however, is so fast that we dealt with it like a static system. Therefore, representing a system as static is a **modeling decision**.

Table 1 shows how this field can be categorized with respect to linear/nonlinear and static/dynamic systems.

System Characteristics	Linear Model	Nonlinear Model
Static	Linear Regression	Machine Learning
Dynamic	Linear System Identification	Nonlinear System Identification

> Table 1: Naming conventions in the System Identification field. Adapted from Oliver Nelles

Nonlinear System Identification and Forecasting Applications: Case Studies

There's a lot of research out there on nonlinear system identification, including NARMAX methods. However, there are a relatively small number of books and papers showing how to apply these methods to real-life systems Instead in a way that's easy to understand. Our goal with this book is to change that. We want to make these methods practical and accessible. While we'll cover the necessary math and algorithms, we'll keep things as clear and simple as possible, making it easier for readers from all backgrounds to learn how to model dynamic nonlinear systems using **SysIdentPy**.

Therefore, this book aims to fill a gap in the existing literature. In Chapter 10, we present real-world case studies to show how NARMAX methods can be applied to a variety of complex systems. Whether it's modeling a highly nonlinear system like the Bouc-Wen model, modeling a dynamic behavior in a full-scale F-16 aircraft, or working with the M4 dataset for benchmarking, we'll guide you through building NARMAX models using **SysIdentPy**.

The case studies we've selected come from a wide range of fields, not just the typical timeseries or industrial examples you might expect from traditional system identification or timeseries books. Our aim is to showcase the versatility of NARMAX algorithms and **SysIdentPy** and illustrate the kind of in-depth analysis you can achieve with these tools.

Abbreviations

Abbreviation	Full Name
AIC	Akaike Information Criterion
AICC	Corrected Akaike Information Criterion
AOLS	Accelerated Orthogonal Least Squares
ANN	Artificial Neural Network
AR	AutoRegressive
ARMAX	AutoRegressive Moving Average with eXogenous Input
ARARX	AutoRegressive AutoRegressive with eXogenous Input
ARX	AutoRegressive with eXogenous Input
BIC	Bayesian Information Criterion
ELS	Extended Least Squares
ER	Entropic Regression
ERR	Error Reduction Ratio
FIR	Finite Impulse Response
FPE	Final Prediction Error
FROLS	Forward Regression Orthogonal Least Squares
GLS	Generalized Least Squares
LMS	Least Mean Square
LS	Least Squares
LSTM	Long Short-Term Memory
MA	Moving Average
MetaMSS	Meta Model Structure Selection
MIMO	Multiple Input Multiple Output
MISO	Multiple Input Single Output
MLP	Multilayer Perceptron
MSE	Mean Squared Error
MSS	Model Structure Selection
NARMAX	Nonlinear AutoRegressive Moving Average with eXogenous Input
NARX	Nonlinear AutoRegressive with eXogenous Input
NFIR	Nonlinear Finite Impulse Response
NIIR	Nonlinear Infinite Impulse Response
NLS	Nonlinear Least Squares
NN	Neural Network
OBF	Orthonormal Basis Function

Abbreviation	Full Name
OE	Output Error
OLS	Orthogonal Least Squares
RBF	Radial Basis Function
RELS	Recursive Extended Least Squares
RLS	Recursive Least Squares
RMSE	Root Mean Squared Error
SI	System Identification
SISO	Single Input Single Output
SVD	Singular Value Decomposition
WLS	Weighted Least Squares

Variables

Variable Name	Description
$f(\cdot)$	function to be approximated
k	discrete time
m	dynamic order
x	system inputs
y	system output
$\hat{y}$	model predicted output
λ	regularization strength
σ	standard deviation
θ	parameter vector
N	number of data points
$\Psi(\cdot)$	Information Matrix
n_{m^r}	Number of potential regressors for MIMO models
$\mathcal{F}$	Arbitrary mathematical representation
$\Omega_{y^p x^m}$	Term cluster of polynomial NARX
ℓ	nonlinearity degree of the model
$\hat{\Theta}$	Estimated Parameter Vector
$\hat{y}_k$	model predicted output at discrete time k
$\mathbf{X}_k$	Column vector of multiple system inputs at discrete-time k
$\mathbf{Y}_k$	Column vector of multiple system outputs at discrete-time k

Variable Name	Description
$\mathcal{H}_t(\omega)$	Hysteresis loop of the system in continuous-time
$\mathcal{H}$	Bounding structure that delimits the system hysteresis loop
ρ	Tolerance value
$\sum_{y^p x^m}$	Cluster coefficients of polynomial NARX
e_k	error vector at discrete-time k
n_r	Number of potential regressors for SISO models
n_x	maximum lag of the input regressor
n_y	maximum lag of the output regressor
n	number of observations in a sample
x_k	system input at discrete-time k
y_k	system output at discrete-time k

Book Organization

This book focuses on making concepts easy to understand, emphasizing clear explanations and practical connections between different methods. We avoid excessive formalism and complex equations, opting instead to illustrate core ideas with plenty of hands-on examples. Written with a System Identification perspective, the book offers practical implementation details throughout the chapters.

The goals of this book are to help you:

- Understand the advantages, drawbacks, and areas of application of different NARMAX models and algorithms.
- Choose the right approach for your specific problem.
- Adjust all hyperparameters properly.
- Interpret and comprehend the obtained results.
- valuate the reliability and limitations of your models.

Many chapters includes real-world examples and data, guiding you on how to apply these methods using SysIdentPy in practice.

2 - NARMAX Model Representation

There are several NARMAX model representations, including polynomial, Fourier, generalized additive, neural networks, and wavelet (Billings, S. A, Aguirra, L. A). This book focuses on the model representations available in SysIdentPy and we'll keep things updated as new methods are added to the package. If a particular representation is mentioned but is not available in SysIdentPy, it will be explicitly mentioned.

To reproduce the codes presented in this section, make sure you have these packages installed:

```
sysidentpy, scikit-learn, scipy, pytorch, matplotlib
```

Basis Function

In System Identification, understanding the concept of basis functions is crucial for effectively modeling complex systems. Basis functions are predefined mathematical functions used to transform the input data into a new space, where the relationships within the data can be more easily modeled. By expressing the original data in terms of these basis functions, we can build nonlinear models in respect to it's structure while keeping it linear in the parameters, allowing the usage of straightforward parameter estimation methods.

Basis functions commonly used in system identification:

1. **Polynomial Basis Functions**: These functions are powers of the input variables. They are useful for capturing simple nonlinear relationships.
2. **Fourier Basis Functions**: These sinusoidal functions (sine and cosine) are ideal for representing periodic patterns within the data.
3. **Wavelet Basis Functions**: These functions are localized in both time and frequency, making them suitable for analyzing data with varying frequency components. Not available in SysIdentPy yet.

In SysIdentPy you can define the basis function you want to use in your model by just import them:

```
from sysidentpy.basis_function import Polynomial, Fourier, Bernstein
```

To keep things simple for now, we will show simple examples of how basis function can be used in a modeling task. We will show a simple polynomial basis functions, a triangular basis function, a radial basis function and a rectangular basis function.

> SysIdentPy does not currently include Vandermonde or any of the other basis functions defined below. These functions are provided solely as examples to illustrate the significance of the basis functions. The examples are based on Fredrik Bagge Carlson's PhD thesis, which I highly recommended for anyone interested in Nonlinear System Identification.

> Although Vandermonde and Radial Basis Functions (RBF) are planned for inclusion as native basis functions in SysIdentPy version 1.0, users can already create and use their own custom basis functions with SysIdentPy. An example of how to do this is available on the SysIdentPy documentation page.

Example: Vandermonde Matrix

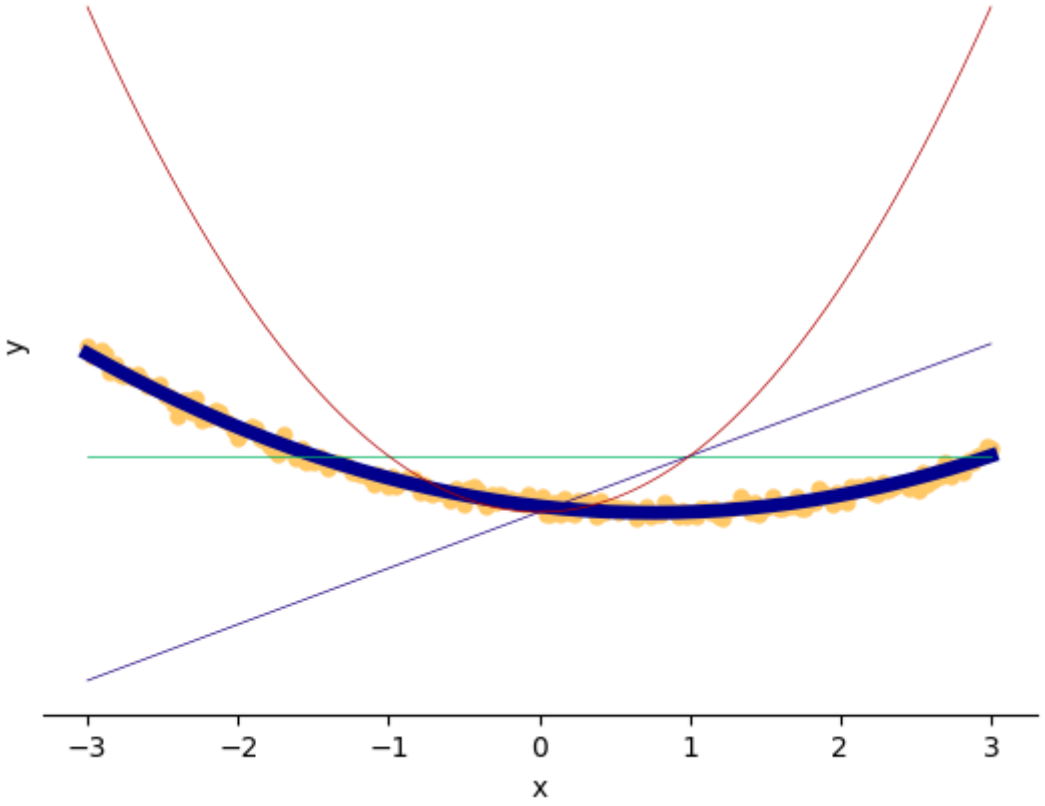
The polynomial basis functions used in this example is defined as:

$$\phi_i(x) = x^i \tag{2.1}$$

where i is the degree of the polynomial and x is the input variable.

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.linear_model import LinearRegression

# Generate simulated quadratic polynomial data
np.random.seed(0)
x = np.linspace(-3, 3, 200)
y = 0.2 * x**2 - 0.3 * x + 0.1 + np.random.normal(0, 0.1, size=x.shape)

# Polynomial basis function
def poly_basis(x, degree):
    return np.vander(x, degree + 1, increasing=True)

# Create polynomial features
degree = 2
X_poly = poly_basis(x, degree)
# Fit a linear regression model
model = LinearRegression()
model.fit(X_poly, y)
y_pred = model.predict(X_poly)
# Plot the original data (quadratic polynomial)
plt.scatter(x, y, color='#ffc865', s=25)
# Plot the polynomial approximation
plt.plot(x, y_pred, color='#00008c', linewidth=5)
# Plot the polynomial basis functions
basis_colors = ["#00b262", "#20007e", "#b20000"]
for i in range(degree + 1):
    plt.plot(x, poly_basis(x, degree)[:, i], linewidth=0.5, color=basis_colors[i %
len(basis_colors)])
```

```
plt.gca().spines['top'].set_visible(False)
plt.gca().spines['right'].set_visible(False)
plt.gca().spines['left'].set_visible(False)
plt.gca().spines['bottom'].set_visible(True)
plt.gca().xaxis.set_ticks_position('bottom')
plt.gca().yaxis.set_ticks([])
plt.show()
```



> Figure 1. Approximation using Vandermode Matrix. The yellow dots show the system data, the bold blue line represents the predicted values, and the other lines depict the basis functions.

Example: Rectangular Basis Functions

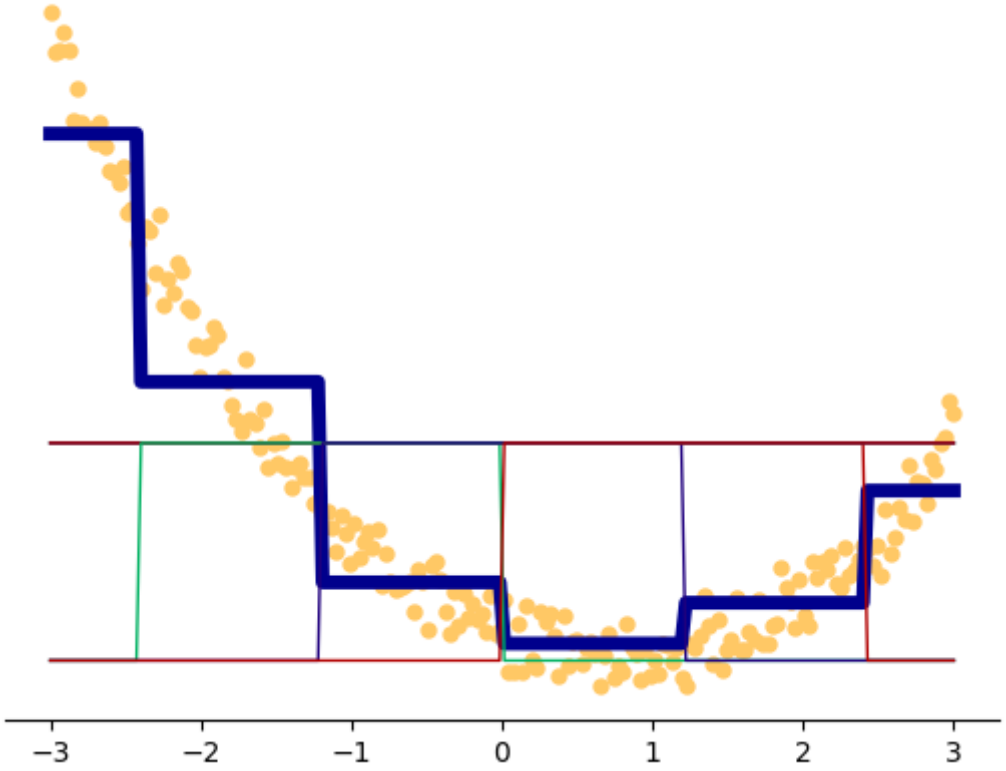
The rectangular basis functions are defined as:

$$\phi_i(x) = \begin{cases} 1 & \text{if } c_i - \frac{w}{2} \leq x < c_i + \frac{w}{2} \\ 0 & \text{otherwise} \end{cases} \tag{2.2}$$

where c_i represents the center of the basis function, w is the width, and x is the input variable.

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.linear_model import LinearRegression

# Generate simulated quadratic polynomial data
```

```
np.random.seed(0)
x = np.linspace(-3, 3, 200)
y = 0.2 * x**2 - 0.3 * x + 0.1 + np.random.normal(0, 0.1, size=x.shape)
# Rectangular basis function
def rectangular_basis(x, centers, width):
    return np.column_stack([(np.abs(x - c) < width).astype(float) for c in centers])

# Create rectangular features
centers = np.linspace(-3, 3, 6)
width = 3
X_rect = rectangular_basis(x, centers, width)
# Fit a linear regression model
model = LinearRegression()
model.fit(X_rect, y)
y_pred = model.predict(X_rect)
# Plot the original data (quadratic polynomial)
plt.scatter(x, y, color='#ffc865', s=25)
# Plot the rectangular approximation
plt.plot(x, y_pred, color='#00008c', linewidth=5)
# Plot the rectangular basis functions
basis_colors = ["#00b262", "#20007e", "#b20000"]
for i in range(len(centers)):
    plt.plot(x, rectangular_basis(x, centers, width)[:, i], linewidth=1,
color=basis_colors[i % len(basis_colors)])

plt.gca().spines['top'].set_visible(False)
plt.gca().spines['right'].set_visible(False)
plt.gca().spines['left'].set_visible(False)
plt.gca().spines['bottom'].set_visible(True)
plt.gca().xaxis.set_ticks_position('bottom')
plt.gca().yaxis.set_ticks([])
plt.show()
```



> Figure 2. Approximation using Rectangular Basis Function. The yellow dots show the system data, the bold blue line represents the predicted values, and the other lines depict the basis functions.

Example: Triangular Basis Functions

The triangular basis functions are defined as:

$$\phi_i(x) = \max\left(0, 1 - \frac{|x - c_i|}{w}\right) \tag{2.3}$$

where c_i is the center of the basis function, w is the width, and x is the input variable.

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.linear_model import LinearRegression

# Generate simulated quadratic polynomial data
np.random.seed(0)
x = np.linspace(-3, 3, 200)
y = 0.2 * x**2 - 0.3 * x + 0.1 + np.random.normal(0, 0.1, size=x.shape)
# Triangular basis function
def triangular_basis(x, centers, width):
    return np.column_stack([np.maximum(0, 1 - np.abs((x - c) / width)) for c in
centers])

# Create triangular features
centers = np.linspace(-3, 3, 6)
```

```
width = 1.5
X_tri = triangular_basis(x, centers, width)
# Fit a linear regression model
model = LinearRegression()
model.fit(X_tri, y)
y_pred = model.predict(X_tri)
# Plot the original data (quadratic polynomial)
plt.scatter(x, y, color='#ffc865', s=25)
# Plot the triangular approximation
plt.plot(x, y_pred, color='#00008c', linewidth=5)
# Plot the triangular basis functions
basis_colors = ["#00b262", "#20007e", "#b20000"]
for i in range(len(centers)):
    plt.plot(x, triangular_basis(x, centers, width)[:, i], linewidth=1,
color=basis_colors[i % len(basis_colors)])

plt.gca().spines['top'].set_visible(False)
plt.gca().spines['right'].set_visible(False)
plt.gca().spines['left'].set_visible(False)
plt.gca().spines['bottom'].set_visible(True)
plt.gca().xaxis.set_ticks_position('bottom')
plt.gca().yaxis.set_ticks([])
plt.show()
```

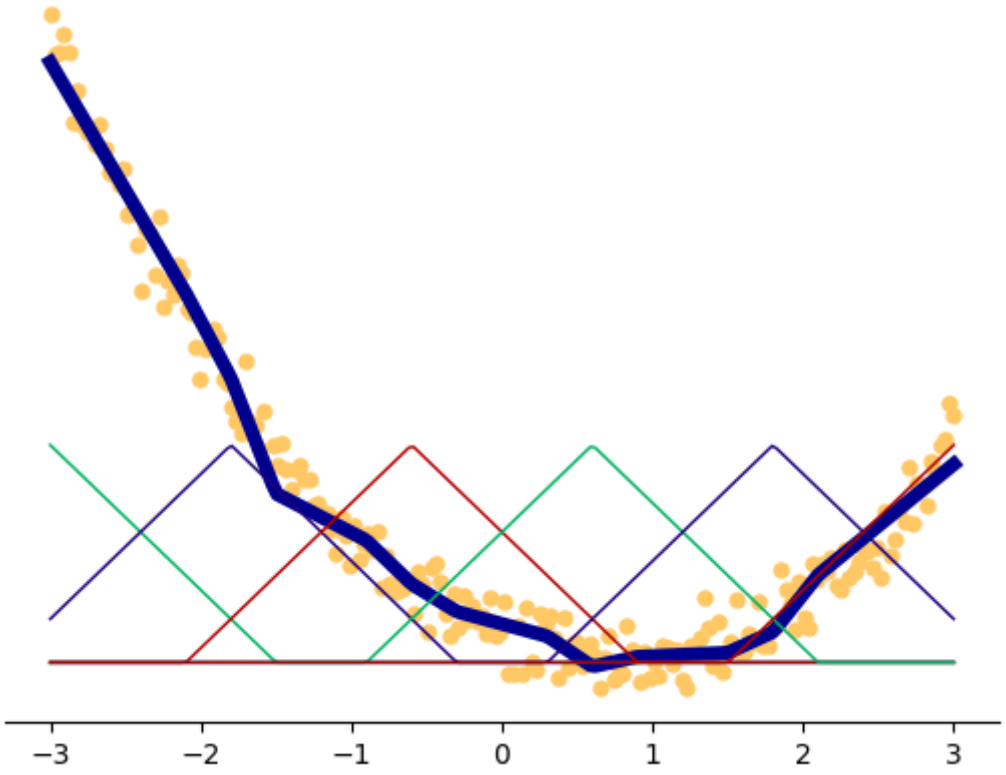


Figure 3. Approximation using a Triangular Basis Function. The yellow dots show the system data, the bold blue line represents the predicted values, and the other lines depict the basis functions.

Example: Radial Basis Function (RBF) - Gaussian

The Gaussian Radial Basis Function is defined as:

$$\phi(x; c, \sigma) = \exp\left(-\frac{(x-c)^2}{2\sigma^2}\right) \tag{2.4}$$

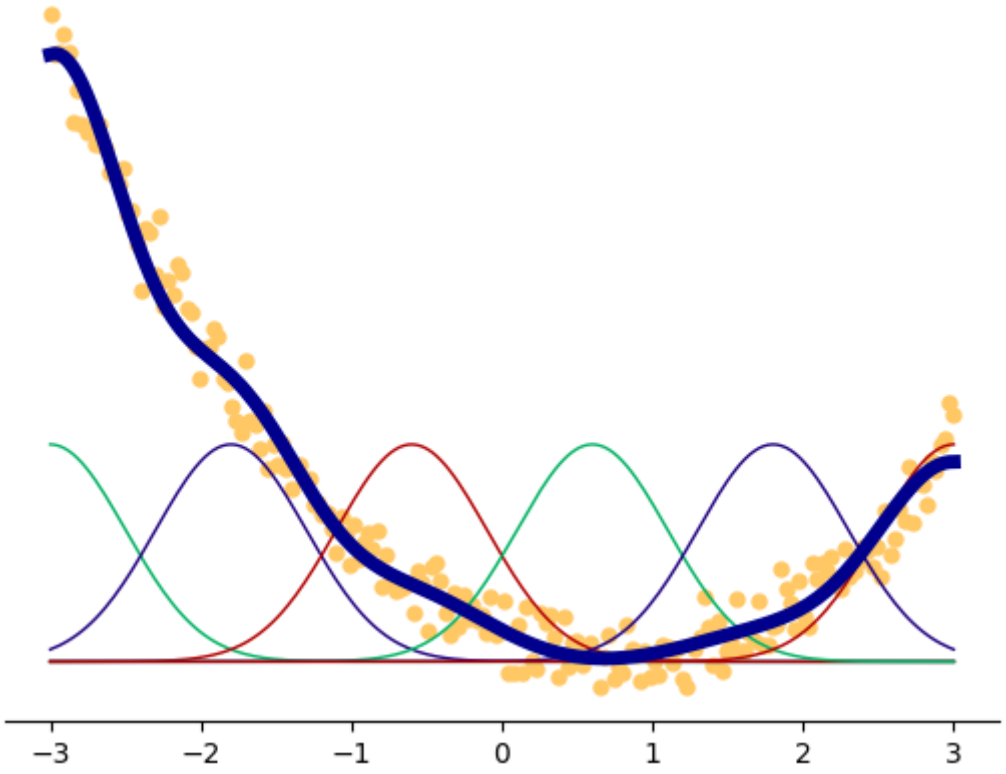
where:

- x is the input variable.
- c is the center of the RBF.
- σ is the spread (or scale) of the RBF.

This function measures the distance between x and the center c, and it decays exponentially based on the width σ. The smaller the σ, the more localized the basis function is around the center c.

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.linear_model import LinearRegression

# Generate simulated quadratic polynomial data
np.random.seed(0)
x = np.linspace(-3, 3, 200)  # More points for a smoother curve
y = 0.2 * x**2 - 0.3 * x + 0.1 + np.random.normal(0, 0.1, size=x.shape)  # Quadratic
polynomial with noise
# RBF centers and sigma
centers = np.linspace(-3, 3, 6)  # More centers for better coverage
sigma = 0.5  # Spread of the RBF
# RBF basis function
def rbf_basis(x, c, sigma):
    return np.exp(- (x - c) ** 2 / (2 * sigma ** 2))

# Create RBF features
X_rbf = np.column_stack([rbf_basis(x, c, sigma) for c in centers])
# Fit a linear regression model
model = LinearRegression()
model.fit(X_rbf, y)
y_pred = model.predict(X_rbf)
# Plot the original data (quadratic polynomial)
plt.scatter(x, y, color='#ffc865', s=25)
# Basis function colors
basis_colors = ["#00b262", "#20007e", "#b20000"]
n_colors = len(basis_colors)
# Plot the basis functions
for i, c in enumerate(centers):
    color = basis_colors[i % n_colors]
    plt.plot(x, rbf_basis(x, c, sigma), linewidth=1, color=color, label=f'RBF Center
{c:.2f}')
```

```
# Plot the approximation
plt.plot(x, y_pred, color='#00008c', linewidth=5)
plt.gca().spines['top'].set_visible(False)
plt.gca().spines['right'].set_visible(False)
plt.gca().spines['left'].set_visible(False)
plt.gca().spines['bottom'].set_visible(True)
plt.gca().xaxis.set_ticks_position('bottom')
plt.gca().yaxis.set_ticks([])
plt.show()
```



> Figure 4. Approximation using the Radial Basis Function. The yellow dots show the system data, the bold blue line represents the predicted values, and the other lines depict the basis functions.

Linear Models

ARMAX

You may have noticed the similarity between the acronym NARMAX with the well-known models ARX, ARMAX, etc., which are widely used for forecasting time series. And this resemblance is not by chance. The AutoRegressive models with Moving Average and Exogenous Input (ARMAX) and their variations AR, ARX, ARMA (to name just a few) are one of the most used mathematical representations for identifying linear systems. The ARMAX can be expressed as:

$$y_k = \phi[y_{k-1}, \ldots, y_{k-n_y}, x_{k-d}, x_{k-d-1}, \ldots, x_{k-d-n_x}, e_{k-1}, \ldots, e_{k-n_e}] + e_k \tag{2.5}$$

where $n_y \in \mathbb{N}$, $n_x \in \mathbb{N}$, $n_e \in \mathbb{N}$, are the maximum lags for the system output, input and noise regressors (representing the moving average part), respectively; $x_k \in \mathbb{R}^{n_x}$ is the system input and $y_k \in \mathbb{R}^{n_y}$ is the system output at discrete time $k \in \mathbb{N}^n$; $e_k \in \mathbb{R}^{n_e}$ stands for uncertainties and possible noise at discrete time k. In this case, ϕ is some linear function of the input and output regressors and d is a time delay typically set to $d = 1$.

If $\mathcal{F}$ is a polynomial, we have a polynomial ARMAX model

$$y_k = \sum_{0} + \sum_{i=1}^{p} \Theta_y^i y_{k-i} + \sum_{j=1}^{q} \Theta_e^j e_{k-j} + \sum_{m=1}^{r} \Theta_x^m x_{k-m} + e_k \tag{2.6}$$

where $\sum_0$, Θ_y^i, Θ_e^j, and Θ_x^m are constant parameters.

The following example is a polynomial ARMAX model:

$$y_k = 0.7213 y_{k-1} - 0.5692 y_{k-2} + 0.1139 x_{k-1} - 0.1691 x_{k-1} + 0.2245 e_{k-1} \tag{2.7}$$

You can easily build a polynomial ARMAX model using **SysIdentPy**:

```
from sysidentpy.model_structure_selection import FROLS
from sysidentpy.basis_function import Polynomial
from sysidentpy.parameter_estimation import LeastSquares

basis_function = Polynomial(degree=1)
model = FROLS(
        basis_function=basis_function,
        estimator=LeastSquares(unbiased=True)
)
```

In the example above, we define the linear polynomial basis function by importing the Polynomial basis and setting the degree equal to 1 (this ensure that we do not have a nonlinear combination of the regressors). Don't worry about the `FROLS` and `LeastSquares` yet. We'll talk about them in chapters 3 and 4, respectively.

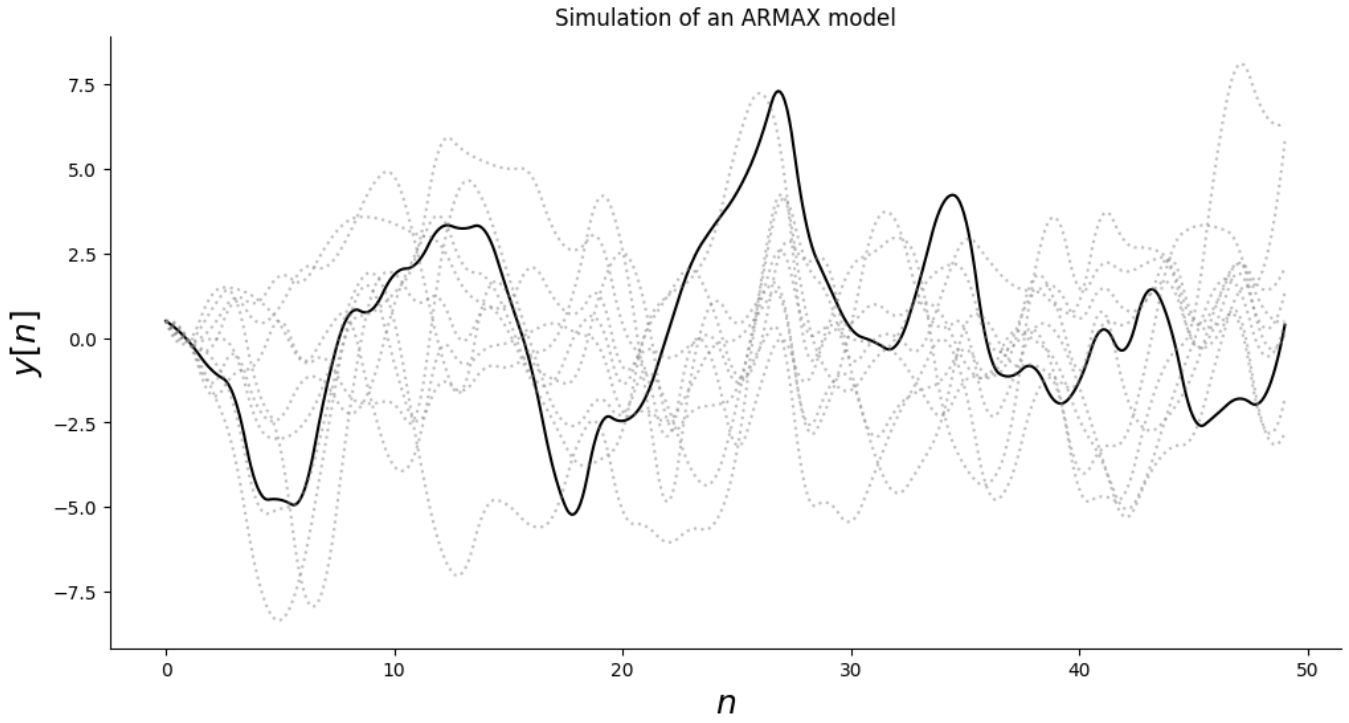
For Figure 4, we conducted 10 separate simulations to analyse the effects of different noise process generation on the ARMAX system's behavior. Each simulation uses a unique sample of noise to observe how variations in this random component influence the overall system output. To illustrate this, we highlight one specific simulation while the others are displayed with less emphasis.

It's important to notice that all simulations, whether highlighted or not, are governed by the same underlying model. The deterministic part of the model equation explains the behavior of all the signals shown. The noticeable differences among the signals arise solely from the distinct noise samples used in each simulation. Despite these variations, the core dynamics of the signal remain consistent and are described by the model's deterministic component.

> Most of the code presented in this chapter is intended to illustrate fundamental concepts rather than demonstrating how to use SysIdentPy specifically. Many examples are implemented using

> pure Python to help you better understand the underlying concepts, replicate the examples, and adapt them as needed. SysIdentPy itself will be introduced and be used in the examples in the following chapter.

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.interpolate import interp1d

random_samples = 50
n = np.arange(random_samples)
def system_equation(y, u, nu):
    yk = 0.9*y[0] - 0.24*y[1] + 0.92*u[0] + 0.92*nu[0] + nu[1]
    return yk

# Create a single figure and axis for all plots
fig, ax = plt.subplots(figsize=(12, 6))
u = np.random.normal(size=(random_samples,), scale=1)
for k in range(10):
    nu = np.random.normal(size=(random_samples,), scale=0.9)
    y = np.empty_like(nu)
    # Initial Conditions
    y0 = [0.5, -0.1]
    y[0:2] = y0
    for i in range(2, len(y)):
        y[i] = system_equation([y[i - 1], y[i - 2]], [u[i - 1]], [nu[i - 1], nu[i]])

    # Interpolate the data just to make the plot "nicer"
    interpolation_function = interp1d(n, y, kind='quadratic')
    n_fine = np.linspace(n.min(), n.max(), 10*len(n))  # More points for a smoother
curve
    y_interpolated = interpolation_function(n_fine)
    # Plotting the interpolated data
    if k == 0:
        ax.plot(n_fine, y_interpolated, color='k', alpha=1, linewidth=1.5)
    else:
        ax.plot(n_fine, y_interpolated, color='grey', linestyle=":", alpha=0.5,
linewidth=1.5)

ax.set_xlabel("$n$", fontsize=18)
ax.set_ylabel("$y[n]$", fontsize=18)
ax.set_title("Simulation of an ARMAX model")
plt.show()
```



> Figure 4. Simulations to show the effects of different noise process generation on the ARMAX model's behavior.

ARX

If we do not include noise terms e_{k-n_e} in equation (1), we have ARX models.

$$y_k = \sum_{0} + \sum_{i=1}^{p} \Theta_y^i y_{k-i} + \sum_{m=1}^{r} \Theta_x^m x_{k-m} + e_k \tag{2.8}$$

The following example is a polynomial ARX model:

$$y_k = 0.7213 y_{k-1} - 0.5692 y_{k-2} + 0.1139 x_{k-1} - 0.1691 x_{k-1} \tag{2.9}$$

The only difference in SysIdentPy is setting the `unbiased=False`

```
from sysidentpy.model_structure_selection import FROLS
from sysidentpy.basis_function import Polynomial
from sysidentpy.parameter_estimation import LeastSquares

basis_function = Polynomial(degree=1)
model = FROLS(
        basis_function=basis_function,
        estimator=LeastSquares(unbiased=False)
)
```

The following example shows 10 separate simulations to analyse the effects of different noise process generation on the ARX system's behavior.

```python
import numpy as np
import matplotlib.pyplot as plt
from scipy.interpolate import interp1d

random_samples = 50
n = np.arange(random_samples)
def system_equation(y, u, nu):
    yk = 0.9*y[0] - 0.24*y[1] + 0.92*u[0] + nu[0]
    return yk

# Create a single figure and axis for all plots
fig, ax = plt.subplots(figsize=(12, 6))
u = np.random.normal(size=(random_samples,), scale=1)
for k in range(10):
    nu = np.random.normal(size=(random_samples,), scale=0.9)
    y = np.empty_like(nu)
    # Initial Conditions
    y0 = [0.5, -0.1]
    y[0:2] = y0
    for i in range(2, len(y)):
        y[i] = system_equation([y[i - 1], y[i - 2]], [u[i - 1]], [nu[i]])

    # Interpolate the data just to make the plot easier to understand
    interpolation_function = interp1d(n, y, kind='quadratic')
    n_fine = np.linspace(n.min(), n.max(), 10*len(n))  # More points for a smoother
curve
    y_interpolated = interpolation_function(n_fine)
    # Plotting the interpolated data
    if k == 0:
        ax.plot(n_fine, y_interpolated, color='k', alpha=1, linewidth=1.5)
    else:
        ax.plot(n_fine, y_interpolated, color='grey', linestyle=":", alpha=0.5,
linewidth=1.5)

ax.set_xlabel("$n$", fontsize=18)
ax.set_ylabel("$y[n]$", fontsize=18)
ax.set_title("Simulation of an ARX model")
plt.show()
```

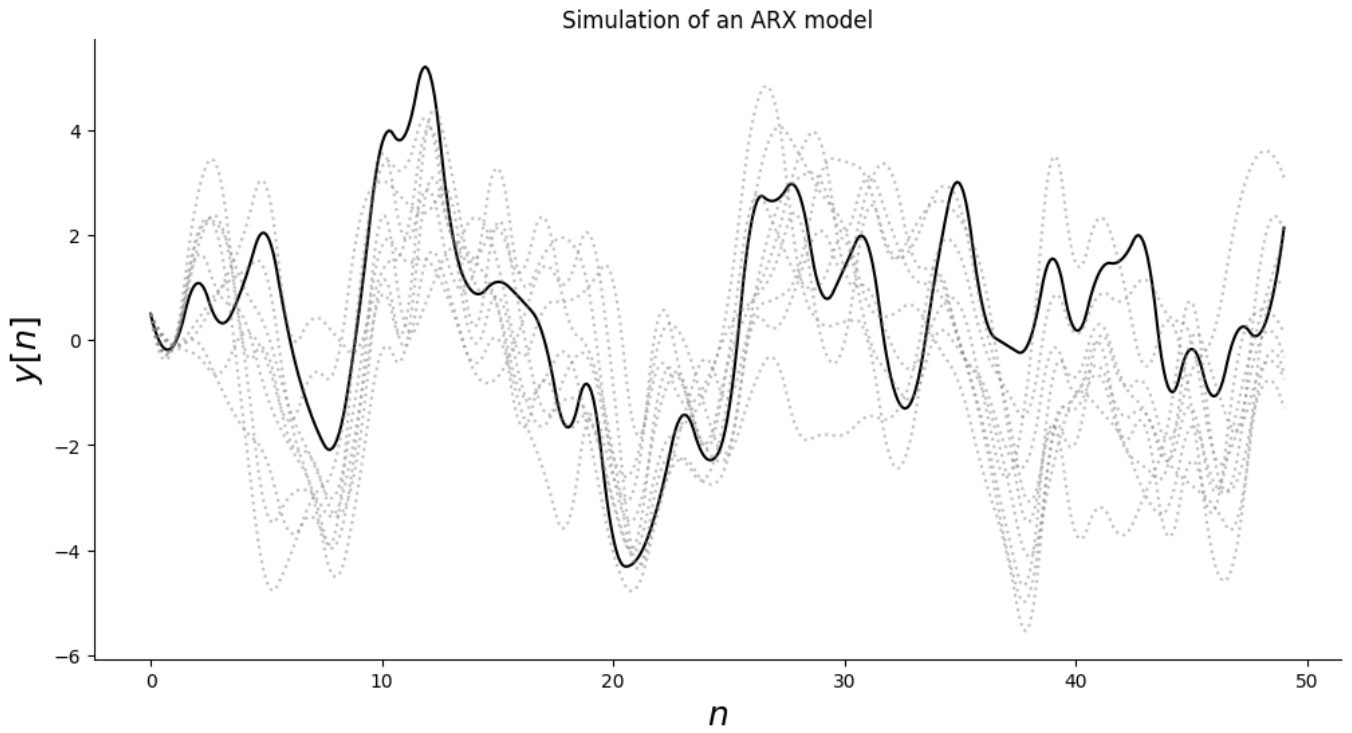


Figure 5. Simulations to show the effects of different noise process generation on the ARX model's behavior.

ARMA

if we do not include input terms in equation (1), it turns to ARMA model

$$y_k = \sum_{0} + \sum_{i=1}^{p} \Theta_y^i y_{k-i} + \sum_{j=1}^{q} \Theta_e^j e_{k-j} + e_k \tag{2.10}$$

The following example is a polynomial ARMA model:

$$y_k = 0.7213 y_{k-1} - 0.5692 y_{k-2} + 0.1139 y_{k-3} - 0.1691 y_{k-4} + 0.2245 e_{k-1} \tag{2.11}$$

Since the model representation do not have inputs, we have to set the model type to `NAR` and set `unbiased=True` again in `LeastSquares`:

```
from sysidentpy.model_structure_selection import FROLS
from sysidentpy.basis_function import Polynomial
from sysidentpy.parameter_estimation import LeastSquares

basis_function = Polynomial(degree=1)
model = FROLS(
        basis_function=basis_function,
        estimator=LeastSquares(unbiased=True),
        model_type="NAR"
)
```

The figure bellow shows 10 separate simulations to analyse the effects of different noise process generation on the ARX system's behavior.

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.interpolate import interp1d

random_samples = 50
n = np.arange(random_samples)
def system_equation(y, nu):
    yk = 0.5*y[0] - 0.4*y[1] + 0.8*nu[0] + nu[1]
    return yk

# Create a single figure and axis for all plots
fig, ax = plt.subplots(figsize=(12, 6))
for k in range(10):
    nu = np.random.normal(size=(random_samples,), scale=0.9)
    y = np.empty_like(nu)
    # Initial Conditions
    y0 = [0.5, -0.1]
    y[0:2] = y0
    for i in range(2, len(y)):
        y[i] = system_equation([y[i - 1], y[i - 2]], [nu[i - 1], nu[i]])

    # Interpolate the data just to make the plot easier to understand
    interpolation_function = interp1d(n, y, kind='quadratic')
    n_fine = np.linspace(n.min(), n.max(), 10*len(n))  # More points for a smoother
curve
    y_interpolated = interpolation_function(n_fine)
    # Plotting the interpolated data
    if k == 0:
        ax.plot(n_fine, y_interpolated, color='k', alpha=1, linewidth=1.5)
    else:
        ax.plot(n_fine, y_interpolated, color='grey', linestyle=":", alpha=0.5,
linewidth=1.5)

ax.set_xlabel("$n$", fontsize=18)
ax.set_ylabel("$y[n]$", fontsize=18)
ax.set_title("Simulation of an ARMA model")
plt.show()
```

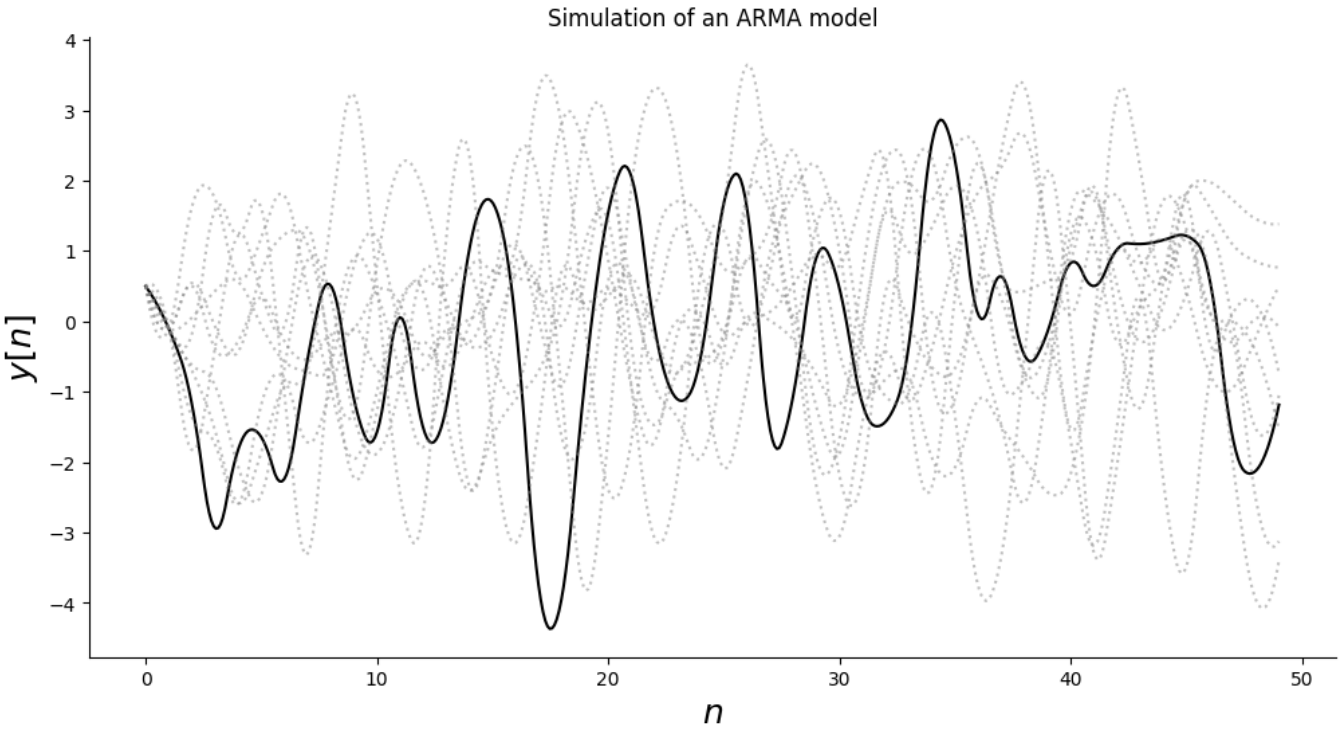


Figure 6. Simulations to show the effects of different noise process generation on the ARMA model's behavior.

AR

if we do not include input terms and noise terms in equation (1), it turns to AR model

$$y_k = \sum_{0} + \sum_{i=1}^{p} \Theta_y^i y_{k-i} + e_k \tag{2.12}$$

The following example is a polynomial AR model:

$$y_k = 0.7213y_{k-1} - 0.5692y_{k-2} + 0.1139y_{k-3} - 0.1691y_{k-4} \tag{2.13}$$

In this case, we have to set the model type to `NAR` and set `unbiased=False` in `LeastSquares`:

```
from sysidentpy.model_structure_selection import FROLS
from sysidentpy.basis_function import Polynomial
from sysidentpy.parameter_estimation import LeastSquares

basis_function = Polynomial(degree=1)
model = FROLS(
        basis_function=basis_function,
        estimator=LeastSquares(unbiased=False),
        model_type="NAR"
)
```

The figure bellow shows 10 separate simulations to analyse the effects of different noise process generation on the AR system's behavior.

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.interpolate import interp1d

random_samples = 50
n = np.arange(random_samples)
def system_equation(y, nu):
    yk = 0.5*y[0] - 0.3*y[1] + nu[0]
    return yk

# Create a single figure and axis for all plots
fig, ax = plt.subplots(figsize=(12, 6))
for k in range(10):
    nu = np.random.normal(size=(random_samples,), scale=0.9)
    y = np.empty_like(nu)
    # Initial Conditions
    y0 = [0.5, -0.1]
    y[0:2] = y0
    for i in range(2, len(y)):
        y[i] = system_equation([y[i - 1], y[i - 2]], [nu[i]])

    # Interpolate the data just to make the plot easier to understand
    interpolation_function = interp1d(n, y, kind='quadratic')
    n_fine = np.linspace(n.min(), n.max(), 10*len(n))  # More points for a smoother
curve
    y_interpolated = interpolation_function(n_fine)
    # Plotting the interpolated data
    if k == 0:
        ax.plot(n_fine, y_interpolated, color='k', alpha=1, linewidth=1.5)
    else:
        ax.plot(n_fine, y_interpolated, color='grey', linestyle=":", alpha=0.5,
linewidth=1.5)

ax.set_xlabel("$n$", fontsize=18)
ax.set_ylabel("$y[n]$", fontsize=18)
ax.set_title("Simulation of an AR model")
plt.show()
```

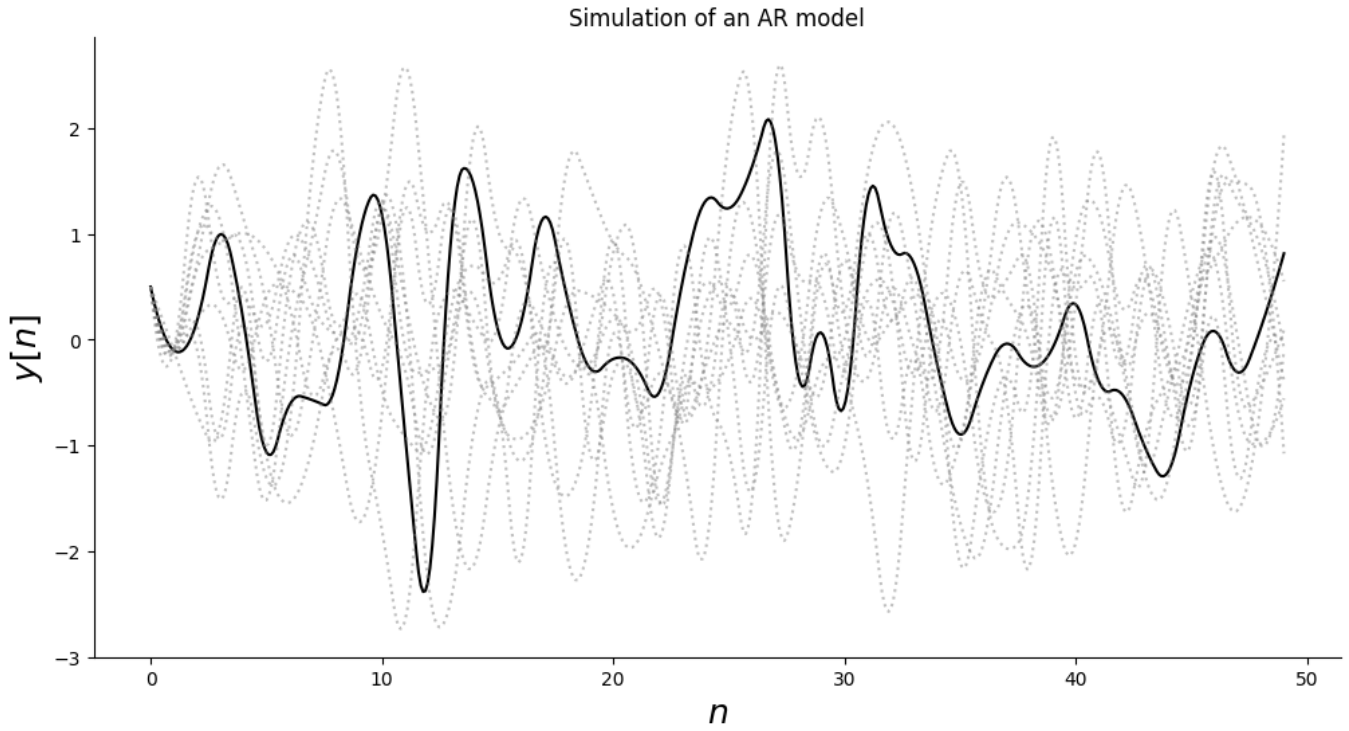


Figure 7. Simulations to show the effects of different noise process generation on the AR model's behavior.

FIR

if we only keep input terms in equation (1), it turns to NFIR model

$$y_k = \sum_{m=1}^{r} \Theta_x^m x_{k-m} + e_k \tag{2.14}$$

The following example is a polynomial FIR model:

$$y_k = 0.7213x_{k-1} - 0.5692x_{k-2} + 0.1139x_{k-3} - 0.1691x_{k-4} \tag{2.15}$$

In this case, we have to set the model type to `NFIR` and set `unbiased=False` in `LeastSquares`:

```
from sysidentpy.model_structure_selection import FROLS
from sysidentpy.basis_function import Polynomial
from sysidentpy.parameter_estimation import LeastSquares

basis_function = Polynomial(degree=1)
model = FROLS(
        basis_function=basis_function,
        estimator=LeastSquares(unbiased=False),
        model_type="NFIR"
)
```

The figure bellow shows 10 separate simulations to analyse the effects of different noise process generation on the FIR system's behavior.

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.interpolate import interp1d

random_samples = 50
n = np.arange(random_samples)
def system_equation(u, nu):
    yk = 0.28*u[0] - 0.34*u[1] + nu[0]
    return yk

u = np.random.normal(size=(random_samples,), scale=1)
# Create a single figure and axis for all plots
fig, ax = plt.subplots(figsize=(12, 6))
for k in range(10):
    nu = np.random.normal(size=(random_samples,), scale=0.9)
    y = np.empty_like(nu)
    # Initial Conditions
    y0 = [0.5, -0.1]
    y[0:2] = y0
    for i in range(2, len(y)):
        y[i] = system_equation([0.1*u[i - 1], u[i - 2]], [nu[i]])

    # Interpolate the data just to make the plot easier to understand
    interpolation_function = interp1d(n, y, kind='quadratic')
    n_fine = np.linspace(n.min(), n.max(), 10*len(n))  # More points for a smoother
curve
    y_interpolated = interpolation_function(n_fine)
    # Plotting the interpolated data
    if k == 0:
        ax.plot(n_fine, y_interpolated, color='k', alpha=1, linewidth=1.5)
    else:
        ax.plot(n_fine, y_interpolated, color='grey', linestyle=":", alpha=0.5,
linewidth=1.5)

ax.set_xlabel("$n$", fontsize=18)
ax.set_ylabel("$y[n]$", fontsize=18)
ax.set_title("Simulation of an FIR model")
plt.show()
```

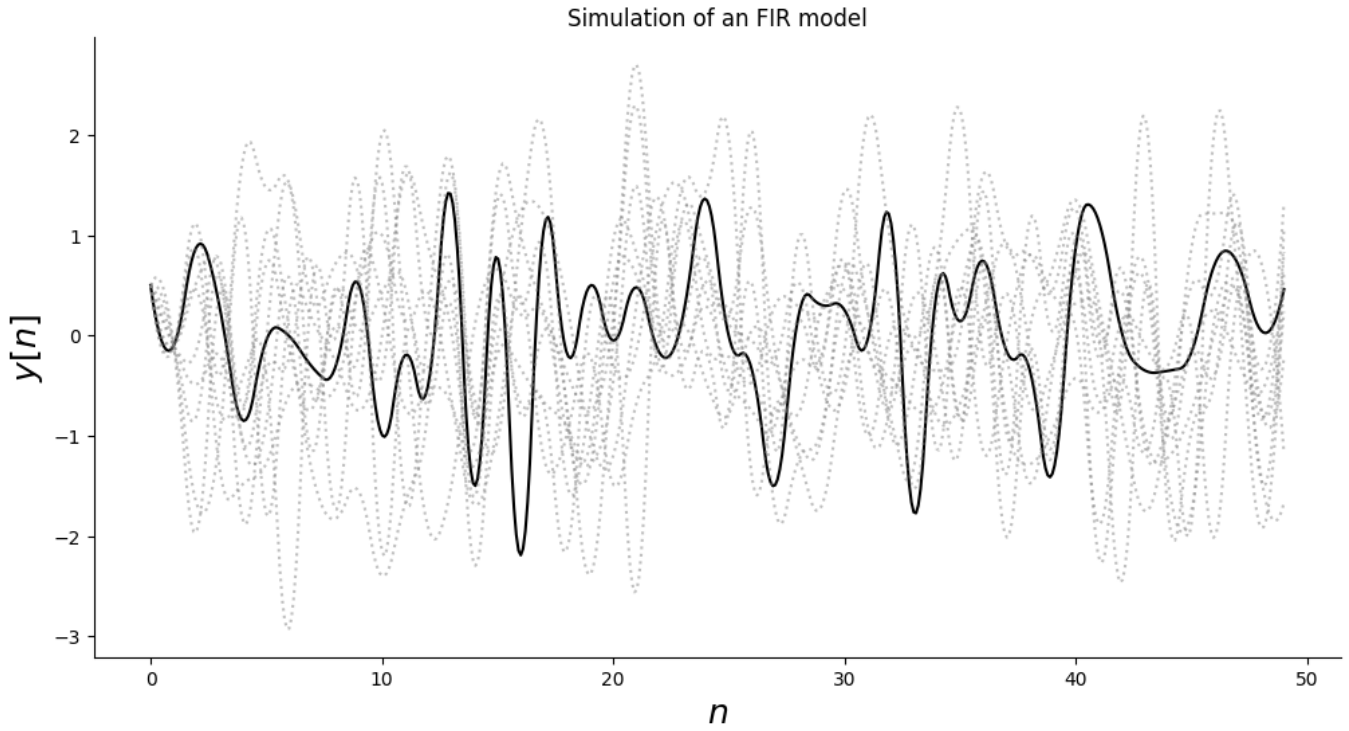


Figure 8. Simulations to show the effects of different noise process generation on the **FIR** model's behavior.

We didn't set the model_type for ARMAX and ARX because the default is `NARMAX`. SysIdentPy allows three different model types: `NARMAX`, `NAR`, and `NFIR`. Because ARMAX, ARX and others linear variants are subsets of NARMAX models, there is no need for specific `ARMAX` model type. The idea is to have model types for model with input and output regressors; models with only output regressors; and models with only input regressors.

Other Variants

For the sake of simplicity, we defined Equation 2.5 and only approach the polynomial representations. However, you can extend the representations to other basis functions, like the Fourier. If you set $\mathcal{F}$ as the Fourier extension

$$\mathcal{F}(x) = [\cos(\pi x), \sin(\pi x), \cos(2\pi x), \sin(2\pi x), \ldots, \cos(N\pi x), \sin(N\pi x)] \tag{2.16}$$

In this case, the Fourier ARX representation will be:

$$\begin{aligned} y_k = &[\cos(\pi y_{k-1}), \sin(\pi y_{k-1}), \cos(2\pi y_{k-1}), \sin(2\pi y_{k-1}), \ldots, \cos(N\pi y_{k-1}), \sin(N\pi y_{k-1}), \\ &\cos(\pi y_{k-n_y}), \sin(\pi y_{k-n_y}), \cos(2\pi y_{k-n_y}), \sin(2\pi y_{k-n_y}), \ldots, \cos(N\pi y_{k-n_y}), \sin(N\pi y_{k-n_y}), \\ &\cos(\pi x_{k-1}), \sin(\pi x_{k-1}), \cos(2\pi x_{k-1}), \sin(2\pi x_{k-1}), \ldots, \cos(N\pi x_{k-1}), \sin(N\pi x_{k-1}), \\ &\cos(\pi y_{k-n_y}), \sin(\pi y_{k-n_y}), \cos(2\pi y_{k-n_y}), \sin(2\pi y_{k-n_y}), \ldots, \cos(N\pi y_{k-n_y}), \sin(N\pi y_{k-n_y})] \\ &+ e_k \end{aligned} \tag{2.17}$$

To do that in SysIdentPy, just import the Fourier basis instead of the Polynomial

```
from sysidentpy.model_structure_selection import FROLS
from sysidentpy.basis_function import Fourier
from sysidentpy.parameter_estimation import LeastSquares

basis_function = Fourier(degree=1)
model = FROLS(
        basis_function=basis_function,
        estimator=LeastSquares(unbiased=False),
        model_type="NARMAX"
)
```

Nonlinear Models

NARMAX

The NARMAX model was proposed by Stephen A. Billings and I.J. Leontaritis in 1981 (Billins, S. A.), and can be described as

$$y_k = \mathcal{F}[y_{k-1}, \ldots, y_{k-n_y}, x_{k-d}, x_{k-d-1}, \ldots, x_{k-d-n_x}, e_{k-1}, \ldots, e_{k-n_e}] + e_k, \tag{2.18}$$

where $n_y \in \mathbb{N}^*$, $n_x \in \mathbb{N}$, $n_e \in \mathbb{N}$, are the maximum lags for the system output and input respectively; $x_k \in \mathbb{R}^{n_x}$ is the system input and $y_k \in \mathbb{R}^{n_y}$ is the system output at discrete time $k \in \mathbb{N}^n$; $e_k \in \mathbb{R}^{n_e}$ represents uncertainties and possible noise at discrete time k. In this case, $\mathcal{F}$ is some nonlinear function of the input and output regressors and d is a time delay typically set to $d = 1$.

You can notice that the difference between Equation 2.5 and Equation 2.18 if the function representing the system. For NARMAX models, $\mathcal{F}$ can be any nonlinear function, while for Equation 2.5 only linear functions are allowed. Although there are many possible approximations of $\mathcal{F}(\cdot)$ (e.g., Neural Networks, Fuzzy, Wavelet, Radial Basis Function), the power-form Polynomial NARMAX model is the most commonly used (Billings, S. A.; Khandelwal, D. and Schoukens, M. and Toth, R.):

$$y_k = \sum_{i=1}^{p} \Theta_i \times \prod_{j=0}^{n_x} x_{k-j}^{b_i,j} \prod_{l=1}^{n_e} e_{k-l}^{d_i,l} \prod_{m=1}^{n_y} y_{k-m}^{a_i,m} \tag{2.19}$$

where p is the number of regressors, Θ_i are the model parameters, and a_i, m, b_i, j and $d_i, l \in \mathbb{N}$ are the exponents of the output, input and noise terms, respectively.

The Equation 2.20 describes a polynomial NARMAX model where the nonlinearity degree is equal to 2, identified from experimental data of a DC motor/generator with no prior knowledge of the model form, taken from [Lacerda Junior, W. R, Almeida, V. M., Martins, S. A. M.](LAM2017.pdf (ufsj.edu.br)):

$$\begin{aligned} y_k =& 1.7813y_{k-1} - 0.7962y_{k-2} + 0.0339x_{k-1} - 0.1597x_{k-1}y_{k-1} + 0.0338x_{k-2} + \\ &+ 0.1297x_{k-1}y_{k-2} - 0.1396x_{k-2}y_{k-1} + 0.1086x_{k-2}y_{k-2} + 0.0085y_{k-2}^2 + 0.0247e_{k-1}e_{k-2} \end{aligned} \tag{2.20}$$

The Θ values are the coefficients of each term of the polynomial equation.

Polynomial basis functions are one of the most used representations of NARMAX models due to several interesting attributes, such as (Billings, S. A.):

- All polynomial functions are smooth in $\mathbb{R}$.
- The Weierstrass approximation theorem states that any continuous real-valued function defined on a closed and bounded space $[a, b]$ can be uniformly approximated using a polynomial on that interval.
- They can describe several nonlinear dynamical systems, including industrial processes, control systems, structural systems, economic and financial systems, biology, medicine, and social systems (some examples are detailed in Lacerda Junior, W. R. and Martins, S. A. M. and Nepomuceno, E. G. and Lacerda, Marcio J. ; Fung, E. H. K. and Wong, Y. K. and Ho, H. F. and Mignolet, M. P.; Kukreja, S. L. and Galiana, H. L. and Kearney, R. E.; Billings, S. A; Aguirre, L. A.; and many others).
- Several algorithms have been developed for structure selection and parameter estimation of polynomial NARMAX models, and it remains an active area of research.
- Polynomial NARMAX models are versatile and can be used both for prediction and inference. The structure of polynomial NARMAX models are easy to interpret and can be related to the underlying system, which is much harder to achieve with neural networks or wavelet functions, for instance.

You can easily build a polynomial NARMAX model using SysIdentPy. Note that the difference for ARMAX, in this case, is the degree of the polynomial function.

```
from sysidentpy.model_structure_selection import FROLS
from sysidentpy.basis_function import Polynomial
from sysidentpy.parameter_estimation import LeastSquares

basis_function = Polynomial(degree=2)
model = FROLS(
        basis_function=basis_function,
        estimator=LeastSquares(unbiased=True)
)
```

One could think that is a simple change, but in nonlinear scenarios the course of dimensionality becomes a real problem. The number of candidate regressors, n_r, of polynomial NARX can be defined as ([Korenberg, M. L. and Billings, S. A. and Liu, Y. P. and McIlroy, P. J.](Orthogonal parameter estimation algorithm for non-linear stochastic systems)):

$$n_r = M + 1, \tag{2.21}$$

where

$$
\begin{aligned}
M &= \sum_{i=1}^{\ell} n_i \\
n_i &= \frac{n_{i-1}(n_y + n_x + i - 1)}{i}, n_0 = 1.
\end{aligned}
\tag{2.22}
$$

As we mentioned in the Introduction of the book, NARMAX methods aims to build the simplest models possible. The idea is to be reproduce a wide range of behaviors using a small subset of terms from the vast search space formed by candidate regressors.

Lets use SysIdentPy to see how the search space grows in the linear versus the nonlinear scenario. The `regressor_code` method available in `narmax_tools` can be used the check how many regressors exists in the search space given the number of inputs, the delays of `y` and `x` regressors and the basis function. We will use `xlag=ylag=10` and the polynomial basis function. The user can simulate different scenarios by setting different parameters.

```
from sysidentpy.utils.narmax_tools import regressor_code
from sysidentpy.basis_function._basis_function import Polynomial]
import numpy as np
```

For the linear case with 1 input we have 21 regressors:

```
x_train = np.random.rand(10, 1)  # simulating a case with 1 input
basis_function = Polynomial(degree=1)
regressors = regressor_code(
    X=x_train,
    xlag=10,
    ylag=10,
    model_type="NARMAX",
    model_representation="Polynomial",
    basis_function=basis_function,
)
n_regressors = regressors.shape[0]  # the number of features of the NARX net
n_regressors
>>> 21
```

For the linear case with 2 inputs, the number of regressors jumps to 111:

```
x_train = np.random.rand(10, 2)  # simulating a case with 2 inputs
basis_function = Polynomial(degree=1)
xlag = [list(range(1, 11))] * x_train.shape[1]
regressors = regressor_code(
    X=x_train,
    xlag=xlag,
    ylag=10,
    model_type="NARMAX",
    model_representation="Polynomial",
```

```
    basis_function=basis_function,
)
n_regressors = regressors.shape[0]  # the number of features of the NARX net
n_regressors
>>> 111
```

If we consider a nonlinear case with 1 input by just changing the degree to 2, we have 231 regressors.

```
x_train = np.random.rand(10, 1)  # simulating a case with 1 input
basis_function = Polynomial(degree=2)
regressors = regressor_code(
    X=x_train,
    xlag=10,
    ylag=10,
    model_type="NARMAX",
    model_representation="Polynomial",
    basis_function=basis_function,
)
n_regressors = regressors.shape[0]  # the number of features of the NARX net
n_regressors
>>> 231
```

If we set the degree to 3, the number of terms increases significantly to 1771 regressors.

```
x_train = np.random.rand(10, 1)  # simulating a case with 1 input
basis_function = Polynomial(degree=2)
regressors = regressor_code(
    X=x_train,
    xlag=10,
    ylag=10,
    model_type="NARMAX",
    model_representation="Polynomial",
    basis_function=basis_function,
)
n_regressors = regressors.shape[0]  # the number of features of the NARX net
n_regressors
>>> 1771
```

If you have 2 inputs in the nonlinear scenario with `degree=2`, the number of regressors is 496:

```
x_train = np.random.rand(10, 2)  # simulating a case with 2 input
basis_function = Polynomial(degree=2)
xlag = [list(range(1, 11))] * x_train.shape[1]
regressors = regressor_code(
    X=x_train,
    xlag=xlag,
```

```
    ylag=10,
    model_type="NARMAX",
    model_representation="Polynomial",
    basis_function=basis_function,
)
n_regressors = regressors.shape[0]  # the number of features of the NARX net
n_regressors
>>> 496
```

If you have 2 inputs in the nonlinear scenario with `degree=3`, the number jumps to 5456 regressors:

```
x_train = np.random.rand(10, 2)  # simulating a case with 2 input
basis_function = Polynomial(degree=3)
xlag = [list(range(1, 11))] * x_train.shape[1]
regressors = regressor_code(
    X=x_train,
    xlag=xlag,
    ylag=10,
    model_type="NARMAX",
    model_representation="Polynomial",
    basis_function=basis_function,
)
n_regressors = regressors.shape[0]  # the number of features of the NARX net
n_regressors
>>> 5456
```

As you can notice, the number of regressors increases significantly as the degree of the polynomial and the number of inputs increases. That makes the model structure selection much more complex! In the linear case with 10 inputs we have `2^31=2.15e+09` possible model combinations. When `degree=2` with 2 inputs we have `2^496=2.05e+149` possible combinations! Try to get the number of possible model combinations when `degree=3` with 2 inputs. Moreover, try that with more inputs and higher nonlinear degree and see how the course of dimensionality is a big problem.

As you can see, getting a simple model in such a large search space is complex model structure selection task. To select the most significant terms from a huge dictionary of possible terms is not an easy task. And it is hard not only because the complex combinatoric problem and the uncertainty concerning the model order. Identifying the most significant terms in a nonlinear scenario is very difficult because depends on the type of the nonlinearity (sparse singularity or near-singular behavior, memory or dumping effects and many others), dynamical response (spatial-temporal systems, time-dependent), the steady-state response, frequency of the data, the noise and many more.

Because of the model structure selection algorithms developed for NARMAX models, even linear models like ARMAX can have different performance when obtained using SysIdentPy when compared to other libraries, like Statsmodels. We have a case study showing exactly that in Chapter 10.

NARX

If we do not include noise terms e_{k-n_e} in Equation (2.19), we have NARX models.

$$y_k = \sum_{i=1}^{p} \Theta_i \times \prod_{j=0}^{n_x} x_{k-j}^{b_i,j} \prod_{m=1}^{n_y} y_{k-m}^{a_i,m} \tag{2.23}$$

The Equation 2.24 describes a simple polynomial NARX model:

$$y_k = 0.7213 y_{k-1} - 0.5692 y_{k-2}^2 + 0.1139 y_{k-1} x_{k-1} \tag{2.24}$$

The only difference in SysIdentPy is setting the `unbiased=False`

```
from sysidentpy.model_structure_selection import FROLS
from sysidentpy.basis_function import Polynomial
from sysidentpy.parameter_estimation import LeastSquares

basis_function = Polynomial(degree=2)
model = FROLS(
        basis_function=basis_function,
        estimator=LeastSquares(unbiased=False)
)
```

> The user can use the codes provided for linear models to analyse the nonlinear models with different noise realizations.

NARMA

if we do not include input terms in Equation 2.19, it turns to NARMA model

$$y_k = \sum_{i=1}^{p} \Theta_i \times \prod_{l=1}^{n_e} e_{k-l}^{d_i,l} \prod_{m=1}^{n_y} y_{k-m}^{a_i,m} \tag{2.25}$$

The following example is a polynomial NARMA model:

$$y_k = 0.7213 y_{k-1} - 0.5692 y_{k-2}^3 + 0.1139 y_{k-3} y_{k-4} + 0.2245 e_{k-1} \tag{2.26}$$

Since the model representation do not have inputs, we have to set the model type to `NAR` and set `unbiased=True` again in `LeastSquares`:

```
from sysidentpy.model_structure_selection import FROLS
from sysidentpy.basis_function import Polynomial
from sysidentpy.parameter_estimation import LeastSquares

basis_function = Polynomial(degree=2)
model = FROLS(
        basis_function=basis_function,
        estimator=LeastSquares(unbiased=True),
```

```
        model_type="NAR"
)
```

NAR

if we do not include input terms and noise terms in Equation 2.19, it turns to AR model

$$y_k = \sum_{i=1}^{p} \Theta_i \times \prod_{m=1}^{n_y} y_{k-m}^{a_i,m} \tag{2.27}$$

The following example is a polynomial NAR model:

$$y_k = 0.7213y_{k-1} - 0.5692y_{k-2}^2 + 0.1139y_{k-3}^3 - 0.1691y_{k-4}y_{k-5} \tag{2.28}$$

In this case, we have to set the model type to `NAR` and set `unbiased=False` in `LeastSquares`:

```
from sysidentpy.model_structure_selection import FROLS
from sysidentpy.basis_function import Polynomial
from sysidentpy.parameter_estimation import LeastSquares

basis_function = Polynomial(degree=2)
model = FROLS(
        basis_function=basis_function,
        estimator=LeastSquares(unbiased=False),
        model_type="NAR"
)
```

NFIR

If we only keep input terms in Equation 2.19, it becomes a NFIR model

$$y_k = \sum_{i=1}^{p} \Theta_i \times \prod_{j=0}^{n_x} x_{k-j}^{b_i,j} \tag{2.29}$$

The following example is a polynomial NFIR model:

$$y_k = 0.7213x_{k-1} - 0.5692x_{k-2}^2 + 0.1139x_{k-3}x_{k-4} - 0.1691x_{k-4}^3 \tag{2.30}$$

In this case, we have to set the model type to `NFIR` and set `unbiased=False` in `LeastSquares`:

```
from sysidentpy.model_structure_selection import FROLS
from sysidentpy.basis_function import Polynomial
from sysidentpy.parameter_estimation import LeastSquares

basis_function = Polynomial(degree=2)
```

```
model = FROLS(
        basis_function=basis_function,
        estimator=LeastSquares(unbiased=False),
        model_type="NFIR"
)
```

Mixed NARMAX Models

In some applications, using a single basis functions cannot provide a satisfactory description for the relationship between the input (or independent) variables and the output (or response) variable. In order to improve the performance of the model, it has been proposed to use a linear combination of a set of nonlinear functions to replace the linear counterparts.

You can achieve that in SysIdentPy using ensembles in basis functions. You can build a Fourier model where terms have interactions. You can also build a model with mixed basis functions, using terms expanded by polynomial basis and Fourier basis or any other basis function available is the package.

> You can only mix a basis function with the polynomial basis for now in SysIdentPy. You can mix Fourier with Polynomial, but you can't mix Fourier with Bernstein.

To mix Fourier or Bernstein basis with Polynomial, the user just have to set `ensamble=True` in the basis function definition

```
from sysidentpy.basis_function import Fourier

basis_function = Fourier(degree=2, ensamble=True)
```

Neural NARX Network

Neural networks are models composed of interconnected layers of nodes (neurons) designed for tasks like classification and regression. Each neuron is a basic unit within these networks. Mathematically, a neuron is represented by a function f that takes an input vector $\mathbf{x} = [x_1, x_2, \ldots, x_n]$ and generates an output y. This function usually involves a weighted sum of the inputs, an optional bias term b, and an activation function ϕ:

$$y = \phi\left(\sum_{i=1}^{n} w_i x_i + b\right) \tag{2.31}$$

where $\mathbf{w} = [w_1, w_2, \ldots, w_n]$ are the weights associated with the inputs. The activation function ϕ introduces nonlinearity into the model, allowing the network to learn complex patterns. Common activation functions include:

- **Sigmoid**: $\phi(z) = \frac{1}{1+e^{-z}}$
  Produces outputs between 0 and 1, making it useful for binary classification.
- **Hyperbolic Tangent (tanh)**: $\phi(z) = \tanh(z) = \frac{e^{z}-e^{-z}}{e^{z}+e^{-z}}$
  Outputs values between -1 and 1, often used to center data around zero.
- **Rectified Linear Unit (ReLU)**: $\phi(z) = \max(0, z)$
  Outputs zero for negative values and the input value itself for positive values, helping to mitigate the vanishing gradient problem.
- **Leaky ReLU**: $\phi(z) = \max(0.01z, z)$
  A variant of ReLU that allows a small, non-zero gradient when the input is negative, addressing the problem of dying neurons.
- **Softmax**: $\phi(z_i) = \frac{e^{z_i}}{\sum_j e^{z_j}}$
  Converts logits into probabilities for multi-class classification, ensuring that the outputs sum to 1.

Each activation function has its own advantages and is chosen based on the specific needs of the neural network and the task at hand.

As mentioned, neural network is composed of multiple layers, each consisting of several neurons. In this respect, the layers can be categorized into:

- **Input Layer**: The layer that receives the input data.
- **Hidden Layers**: Intermediate layers that process the inputs through weighted connections and activation functions.
- **Output Layer**: The final layer that produces the output of the network.

> The network itself therefore has a very simple architecture. The terminology used in neural networks is also slightly different from the standard notation that is universal in system identification and statistics. So, instead of talking about model parameters, the term *network weights* is used, and instead of estimation, the term *learning* is used. This terminology was no doubt introduced to make it appear that something completely new was being discussed, whereas some of the problems addressed are quite traditional - Stephen A. Billings

Notice that the network itself is simply a collection of nonlinear activation units $\phi(\cdot)$ that are simple static functions. There are no dynamics within the network. This is fine for applications such as pattern recognition, but to use the network in system identification lagged inputs and outputs are necessary and these have to be supplied as inputs either explicitly or through a recurrent procedure. In this respect, if we set $\mathcal{F}$ as a neural function, we can adapt it to create a neural NARX model by transforming the neural architecture into a NARX architecture. The neural NARX, however, is not linear in the parameters like the NARMAX models based on basis functions. So, algorithms like Orthogonal Least Squares are not adequate to estimate the weights of the model.

SysIdentPy support a Series-Parallel (open-loop) Feedforward Network training process, which make the training process easier. We convert the NARX network from Series-Parallel to the Parallel (closed-loop) configuration for prediction.

Series-Parallel allows us to use `pytorch` directly for training, so **SysIdentPy** uses `pytorch` in the backend for neural NARX along with auxiliary methods available only in **SysIdentPy**.

A simple neural NARX model can be represented as a Multi-Layer Perceptron neural network with autoregressive component along with delayed inputs.

> Figure 9. Parallel and series-parallel neural network architectures for modeling the dynamic system $\mathbf{y}[k] = \mathbf{F}(\mathbf{y}[k-1], \mathbf{y}[k-2], \mathbf{u}[k-1], \mathbf{u}[k-2])$. The delay operator q^{-1} is such that $\mathbf{y}[k-1] = q^{-1}\mathbf{y}[k]$. Reference: Antonio H. Ribeiro and Luis A. Aguirre

> Neural NARX is not the same model as Recurrent Neural Networks (RNN). The user is referred to the following paper for more details A Note on the Equivalence of NARX and RNN

To build a Neural NARX network in SysIdentPy, the user must use `pytorch`. We use `pytorch` to make the definition of the network architecture flexible. However, this require that the user have a better understanding of how a neural networks. See the script bellow of how to build a simple Neural NARX model in **SysIdentPy**

```
from torch import nn
import torch

from sysidentpy.neural_network import NARXNN
from sysidentpy.basis_function._basis_function import Polynomial
from sysidentpy.utils.generate_data import get_siso_data
from sysidentpy.utils.narmax_tools import regressor_code

# simulated data
x_train, x_valid, y_train, y_valid = get_siso_data(
    n=1000, colored_noise=False, sigma=0.01, train_percentage=80
)
```

The user can use `cuda` following the same approach when build a neural network in **pytorch**

```
torch.cuda.is_available()

device = "cuda" if torch.cuda.is_available() else "cpu"
print(f"Using {device} device")
```

The user can create a NARXNN object and choose the maximum lag of both input and output for building the regressor matrix to serve as input of the network. In addition, you can choose the loss function, the optimizer, the optional parameters of the optimizer, the number of epochs.

Because we built this feature on top of Pytorch, you can choose any of the loss function of the torch.nn.functional. Click here for a list of the loss functions you can use. You just need to pass the

name of the loss function you want.

Similarly, you can choose any of the optimizer of the torch.optim. Click here for a list of optimizer available.

```
basis_function = Polynomial(degree=1)
narx_net = NARXNN(
    ylag=2,
    xlag=2,
    basis_function=basis_function,
    model_type="NARMAX",
    loss_func="mse_loss",
    optimizer="Adam",
    epochs=2000,
    verbose=False,
    device=device,
    optim_params={
        "betas": (0.9, 0.999),
        "eps": 1e-05,
    },  # optional parameters of the optimizer
)
```

Because the NARXNN model were defined using $ylag = 2$, $xlag = 2$ and a polynomial basis function with $degree = 1$, we have a regressor matrix with 4 features. We need the size of the regressor matrix to build the layers of our network. Our input data(`x_train`) have only one feature, but since we are creating an NARX network, a regressor matrix is built behind the scenes with new features based on the `xlag` and `ylag` .

If you need help finding how many regressors are created behind the scenes you can use the `narmax_tools` function `regressor_code` and take the size of the regressor code generated:

```
basis_function = Polynomial(degree=1)
regressors = regressor_code(
    X=x_train, # t
    xlag=2,
    ylag=2,
    model_type="NARMAX",
    model_representation="neural_network",
    basis_function=basis_function,
)

n_features = regressors.shape[0]  # the number of features of the NARX net
n_features
>>> 4
```

The configuration of your network follows exactly the same pattern of a network defined in Pytorch. The following representing our NARX neural network.

```
class NARX(nn.Module):
    def __init__(self):
        super().__init__()
        self.lin = nn.Linear(n_features, 30)
        self.lin2 = nn.Linear(30, 30)
        self.lin3 = nn.Linear(30, 1)
        self.tanh = nn.Tanh()


    def forward(self, xb):
        z = self.lin(xb)
        z = self.tanh(z)
        z = self.lin2(z)
        z = self.tanh(z)
        z = self.lin3(z)
        return z
```

The user have to pass the defined network to our NARXNN estimator and set `cuda` if available (or needed):

```
narx_net.net = NARX()

if device == "cuda":
    narx_net.net.to(torch.device("cuda"))
```

Because we have a fit (for training) and predict function for Polynomial NARMAX, we create the same pattern for the NARX net. So, you only have to fit and predict using the following:

```
narx_net.fit(X=x_train, y=y_train, X_test=x_valid, y_test=y_valid)
yhat = narx_net.predict(X=x_valid, y=y_valid)
```

If the net configuration is built before calling the NARXNN, just pass the model to the NARXNN as follows:

```
class NARX(nn.Module):
    def __init__(self):
        super().__init__()
        self.lin = nn.Linear(n_features, 30)
        self.lin2 = nn.Linear(30, 30)
        self.lin3 = nn.Linear(30, 1)
        self.tanh = nn.Tanh()
```

```
    def forward(self, xb):
        z = self.lin(xb)
        z = self.tanh(z)
        z = self.lin2(z)
        z = self.tanh(z)
        z = self.lin3(z)
        return z


narx_net2 = NARXNN(
    net=NARX(),
    ylag=2,
    xlag=2,
    basis_function=basis_function,
    model_type="NARMAX",
    loss_func="mse_loss",
    optimizer="Adam",
    epochs=2000,
    verbose=False,
    optim_params={
        "betas": (0.9, 0.999),
        "eps": 1e-05,
    },  # optional parameters of the optimizer
)

narx_net2.fit(X=x_train, y=y_train)
yhat = narx_net2.predict(X=x_valid, y=y_valid)
```

General Model Set Representation

Based on the ideia of transforming a static neural network in a neural NARX model, we can extend the method for basically any model class. SysIdentPy do not aim to implement every model class that exists in literature. However, we created a functionality that allows the usage of any other machine learning package that follows a `fit` and `predict` API inside SysIdentPy to convert such models to NARX versions of them.

Lets take XGboost (eXtreme Gradient Boosting) Algorithm as an example. XGBoost is a well kown model class used for regression tasks. XGboost, however, are not a common choice when you are dealing with a dynamical system identification task because they are originally made for modeling static systems. You can easily transform XGboost into a NARX model using SysIdentPy.

Scikit-learn, for example, is another great example. You can transform any Scikit-learn model into NARX models using SysIdentPy. We will see such applications in detail at Chapter 11, but you can see how easy it in the script bellow

```
from sysidentpy.general_estimators import NARX
from sysidentpy.basis_function._basis_function import Polynomial
from sklearn.linear_model import BayesianRidge
import xgboost as xgb

basis_function = Fourier(degree=1)
# define the scikit estimator
scikit_estimator = BayesianRidge()
# transform scikit_estimator into NARX model
gb_narx = NARX(
    base_estimator=scikit_estimator,
    xlag=2,
    ylag=2,
    basis_function=basis_function,
    model_type="NARMAX",
)

gb_narx.fit(X=x_train, y=y_train)
yhat = gb_narx.predict(X=x_valid, y=y_valid)

# XGboost examples
xgb_estimator = xgb.XGBRegressor()
xgb_narx = NARX(
    base_estimator=xgb_estimator,
    xlag=2,
    ylag=2,
    basis_function=basis_function,
    model_type="NARMAX",
)

xgb_narx.fit(X=x_train, y=y_train)
yhat = xgb_narx.predict(X=x_valid, y=y_valid)
```

You can use any other model by just changing the model class and passing it to the `base_estimator` in `NARX` functionality.

MIMO Models

To keep things simple, only SISO models were represented in previous sections. However, the NARMAX models can effortlessly be extended to MIMO case (Billings, S. A. and Chen, S. and Korenberg, M. J.):

$$
\begin{aligned}
y_{i}k = & F_{i}^{\ell}\big[y_{1}k-1, \ldots, y_{1}k-n_{y_1}^{i}, \ldots, y_{s}k-1, \ldots, y_{s}k-n_{y_s}^{i}, x_{1}k-d, \\
& x_{1}k-d-1, \ldots, x_{1}k-d-n_{x_1}^{i}, \ldots, x_{r}k-d, x_{r}k-d-1, \ldots, x_{r}k-d-n_{x_r}^{i}\big] + \xi_{i}k,
\end{aligned}
\tag{2.32}
$$

where for $i = 1, \ldots, s$, each linear in the parameter sub-model can change regarding different maximum lags. More generally, considering

$$Y_k = \begin{bmatrix} y_{1}k \\ y_{2}k \\ \vdots \\ y_{s}k \end{bmatrix}, X_k = \begin{bmatrix} x_{1}k \\ x_{2}k \\ \vdots \\ x_{r}k \end{bmatrix}, \Xi_k = \begin{bmatrix} \xi_{1}k \\ \xi_{2}k \\ \vdots \\ \xi_{r}k \end{bmatrix}, \tag{2.33}$$

the MIMO model can be denoted as

$$Y_k = F^{\ell}[Y_{k-1}, \ldots, Y_{k-n_y}, X_{k-d}, X_{k-d-1}, \ldots, X_{k-d-n_x}] + \Xi_k, \tag{2.34}$$

where $Xk = \{x_{1}k, x_{2}k, \ldots, x_{r}k\} \in \mathbb{R}^{n_{xr}^{i}}$ and $Yk = \{y_{1}k, y_{2}k, \ldots, y_{s}k\} \in \mathbb{R}^{n_{ys}^{i}}$. The number of possibles terms of MIMO NARX model given the i-th polynomial degree, ℓ_i, is:

$$n_{mr} = \sum_{j=0}^{\ell_i} n_{ij}, \tag{2.35}$$

where

$$n_{ij} = \frac{n_{ij-1}\left[\sum_{k=1}^{s} n_{y_k}^{i} + \sum_{k=1}^{r} n_{x_k}^{i} + j - 1\right]}{j}, \qquad n_{i0} = 1, j = 1, \ldots, \ell_i. \tag{2.36}$$

If $s = 1$, we have a MISO model that can be represented by a single polynomial function. Additionally, a MIMO model can be decomposed into MISO models, as presented in the following figure:

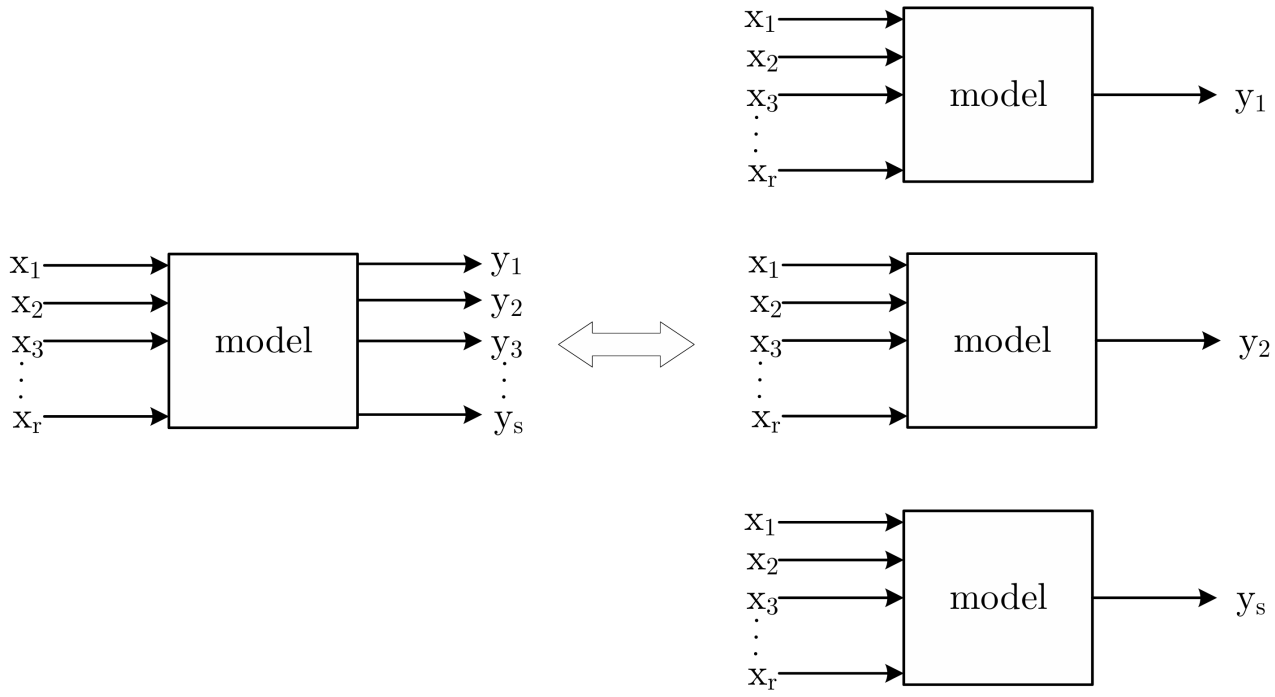


Figure 10. A MIMO model split into individual MISO models.

> SysIdentPy do not support MIMO models yet, only MISO models. You can, however, decompose a MIMO system as presented in Figure 9 and use SysIdentPy to create models for each subsystem.

3 - Parameter Estimation

Least Squares

Consider the NARX model described in a generic form as

$$y_k = \psi_{k-1}^\top \hat{\Theta} + \xi_k, \tag{3.1}$$

where $\psi_{k-1}^\top \in \mathbb{R}^{n_r \times n}$ is the information matrix, also known as the regressors matrix. The information matrix is the input and output transformation based in a basis function and $\hat{\Theta} \in \mathbb{R}^{n_\Theta}$ the vector of estimated parameters. The model above can also be represented in a matrix form as:

$$y = \Psi\hat{\Theta} + \Xi, \tag{3.2}$$

where

$$Y = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix}, \Psi = \begin{bmatrix} \psi_1 \\ \psi_2 \\ \vdots \\ \psi_{n_\Theta} \end{bmatrix}^\top = \begin{bmatrix} \psi_1 1 & \psi_2 1 & \dots & \psi_{n_\Theta} 1 \\ \psi_1 2 & \psi_2 2 & \dots & \psi_{n_\Theta} 2 \\ \vdots & \vdots & & \vdots \\ \psi_1 n & \psi_2 n & \dots & \psi_{n_\Theta} n \end{bmatrix}, \hat{\Theta} = \begin{bmatrix} \hat{\Theta}_1 \\ \hat{\Theta}_2 \\ \vdots \\ \hat{\Theta}_{n_\Theta} \end{bmatrix}, \Xi = \begin{bmatrix} \xi_1 \\ \xi_2 \\ \vdots \\ \xi_n \end{bmatrix}. \tag{3.3}$$

> We will consider the polynomial basis function to keep the examples straightforward, but the methods here will work for any other basis function.

The parametric NARX model is linear in the parameters Θ, so we can use well known algorithms, like the linear Least Squares algorithm developed by Gauss in 1795, to estimate the model parameters. The idea is to find the parameter vector that minimizes the $l2$-norm, also known as the residual sum of squares, described as

$$J_{\hat{\Theta}} = \Xi^\top \Xi = (y - \Psi\hat{\Theta})^\top (y - \Psi\hat{\Theta}) = \|y - \Psi\hat{\Theta}\|^2. \tag{3.4}$$

In Equation 3.4 , $\Psi\hat{\Theta}$ is the one-step ahead prediction of y_k, expressed as

$$\hat{y}_{1_k} = g(y_{k-1}, u_{k-1} | \, \Theta), \tag{3.5}$$

where g is some unknown polynomial function. If the gradient of J_Θ with respect to Θ is equal to zero, then we have the normal equation and the Least Squares estimate is expressed as

$$\hat{\Theta} = (\Psi^\top \Psi)^{-1} \Psi^\top y, \tag{3.6}$$

where $(\Psi^\top \Psi)^{-1} \Psi^\top$ is called the pseudo-inverse of the matrix Ψ, denoted $\Psi^+ \in \mathbb{R}^{n \times n_r}$.

In order to have a bias-free estimator, the following are the basic assumptions needed for the least-squares method:

- A1 - There is no correlation between the error vector, Ξ, and the matrix of regressors, Ψ. Mathematically:
- $\mathrm{E}\{[(\Psi^\top\Psi)^{-1}\Psi^\top]\Xi\} = \mathrm{E}[(\Psi^\top\Psi)^{-1}\Psi^\top]\mathrm{E}[\Xi]$;
- A2 - The error vector Ξ is a zero mean white noise sequence:
- $\mathrm{E}[\Xi] = 0$;
- A3 - The covariance matrix of the error vector is
- $\mathrm{Cov}[\hat{\Theta}] = \mathrm{E}[(\Theta - \hat{\Theta})(\Theta - \hat{\Theta})^\top] = \sigma^2(\Psi^\top\Psi)$;
- A4 - The matrix of regressors, Ψ, is full rank.

The aforementioned assumptions are needed to guarantee that the Least Squares algorithm produce a unbiased final model.

Example

Lets see a practical example. Consider the model

$$y_k = 0.2y_{k-1} + 0.1y_{k-1}x_{k-1} + 0.9x_{k-2} + e_k \tag{3.10}$$

We can generate the input `X` and output `y` using SysIdentPy. Before getting in the details, let run a simple model using SysIdentPy. Because we know a priori that the system we are trying to have is not linear (the simulated system have an interaction term $0.1y_{k-1}x_{k-1}$) and the order is 2 (the maximum lag of the input and output), we will set the hyperparameters accordingly. Note that this a simulated scenario and you'll not have such information a priori in a real identification task. But don't worry, the idea, for now, is just show how things works and we will develop some real models along the book.

```
from sysidentpy.utils.generate_data import get_siso_data
from sysidentpy.model_structure_selection import FROLS
from sysidentpy.basis_function._basis_function import Polynomial
from sysidentpy.parameter_estimation import LeastSquares
from sysidentpy.utils.display_results import results

x_train, x_test, y_train, y_test = get_siso_data(
    n=1000, colored_noise=False, sigma=0.001, train_percentage=90
)

basis_function = Polynomial(degree=2)
estimator = LeastSquares()
model = FROLS(
    n_info_values=3,
    ylag=1,
    xlag=2,
    estimator=estimator,
    basis_function=basis_function,
)
model.fit(X=x_train, y=y_train)
```

```
# print the identified model
r = pd.DataFrame(
    results(
        model.final_model,
        model.theta,
        model.err,
        model.n_terms
    ),
    columns=["Regressors", "Parameters", "ERR"],
)
print(r)

Regressors   Parameters             ERR
0        x1(k-2)  9.0001E-01  9.56885108E-01
1         y(k-1)  2.0000E-01  3.96313039E-02
2  x1(k-1)y(k-1)  1.0001E-01  3.48355000E-03
```

As you can see, the final model have the same 3 regressors of the simulated system and the parameters are very close the ones used to simulate the system. This show us that the Least Squares performed well for this data.

In this example, however, we are applying a Model Structure Selection algorithm (FROLS), which we will see in chapter 6. That's why the final model have only 3 regressors. The parameter estimation algorithm do not choose which terms to include in the model, so if we have a expanded basis function with 6 regressors, it will estimate the parameter for each one of the regressors.

To check how this work, we can use SysIdentPy without Model Structure Selection by generating the information matrix and applying the parameter estimation algorithm directly.

```
from sysidentpy.utils.generate_data import get_siso_data
from sysidentpy.model_structure_selection import FROLS
from sysidentpy.basis_function._basis_function import Polynomial
from sysidentpy.parameter_estimation import LeastSquares
from sysidentpy.narmax_base import InformationMatrix


x_train, x_test, y_train, y_test = get_siso_data(
        n=1000, colored_noise=False, sigma=0.001, train_percentage=90
)
regressor_matrix = InformationMatrix(xlag=2, ylag=2).build_input_output_matrix(
        X=x_train, y=y_train
)
max_lag = 2
# apply the basis function
psi = Polynomial(degree=2).fit(regressor_matrix, max_lag=max_lag)
theta = LeastSquares().optimize(psi, y_train[max_lag:, :])
theta
```

```
array([
   [-1.08377785e-05],
   [ 2.00002486e-01],
   [ 1.73422294e-05],
   [-3.50957931e-06],
   [ 8.99927332e-01],
   [ 2.04427279e-05],
   [-1.47542408e-04],
   [ 1.00062340e-01],
   [ 4.53379771e-05],
   [ 8.90006341e-05],
   [ 1.15234873e-04],
   [ 1.57770755e-04],
   [ 1.58414037e-04],
   [-3.09236444e-05],
   [-1.60377753e-04]]
   )
```

In this case, we have 15 model parameters. If we take a look in the basis function expansion where the degree of the polynomial is equal to 2 and the lags for `y` and `x` are set to 2, we have

```
from sysidentpy.utils.narmax_tools import regressor_code
basis_function = Polynomial(degree=2)
regressors = regressor_code(
    X=x_train,
    xlag=2,
    ylag=2,
    model_type="NARMAX",
    model_representation="Polynomial",
    basis_function=basis_function,
)
regressors

array([[ 0,  0],
   [1001,  0],
   [1002,  0],
   [2001,  0],
   [2002,  0],
   [1001, 1001],
   [1002, 1001],
   [2001, 1001],
   [2002, 1001],
   [1002, 1002],
   [2001, 1002],
   [2002, 1002],
   [2001, 2001],
   [2002, 2001],
```

```
    [2002, 2002]]
    )
```

The regressors is how SysIdentPy encode the polynomial basis function following this codification pattern:

- 0 is the constant term,\n",
- $[1001] = y_{k-1}$
- $[100n] = y_{k-n}$
- $[200n] = x1_{k-n}$
- $[300n] = x2_{k-n}$
- $[1011, 1001] = y_{k-11}timesy_{k-1}$
- $[100n, 100m] = y_{k-n} \times y_{k-m}$
- $[12001, 1003, 1001] = x11_{k-1} \times y_{k-3} \times y_{k-1}$,
- and so on

So, if you take a look at the parameters, we can see that the Least Squares algorithm estimation for the terms that belongs to the simulated system are very close to the real values.

```
[1001, 0] -> [ 2.00002486e-01]
[2002, 0] -> [ 8.99927332e-01]
[2001, 1001] -> [ 1.00062340e-01]
```

Moreover, the parameters estimated for the other regressors are considerably lower values than the ones estimated for the correct terms, indicating that the other might not be relevant to the model.

You can start thinking that we only need to define a basis function and apply some parameter estimation technique to build NARMAX models. However, as mentioned before, the main goal of the NARMAX methods is to build the best model possible while keeping it simple. And that's true for the case where we applied the FROLS algorithm. Besides, when dealing with system identification we want to recover the dynamics of the system under study, so adding more terms than necessary can lead to unexpected behaviors, poor performance and unstable models. Remember, this is only a toy example, so in real cases the model structure selection is fundamental.

You can implement Least Squares method as simple as

```
import numpy as np

def simple_least_squares(psi, y):
    return np.linalg.pinv(psi.T @ psi) @ psi.T @ y

# use the psi and y data created in previous examples or
# create them again here to run the example.
theta = simple_least_squares(psi, y_train[max_lag:, :])
```

```
theta

array(
        [
           [-1.08377785e-05],
           [ 2.00002486e-01],
           [ 1.73422294e-05],
           [-3.50957931e-06],
           [ 8.99927332e-01],
           [ 2.04427279e-05],
           [-1.47542408e-04],
           [ 1.00062340e-01],
           [ 4.53379771e-05],
           [ 8.90006341e-05],
           [ 1.15234873e-04],
           [ 1.57770755e-04],
           [ 1.58414037e-04],
           [-3.09236444e-05],
           [-1.60377753e-04]
        ]
)
```

As you can see, the estimated parameters are very close. However, be careful when using such approach in under-, well-, or over-determined systems. We recommend to use the numpy or scipy `lstsq` methods.

Total Least Squares

This section is based on the [Overview of total least squares methods paper](Markovsky+VanHuffel-SP-2007.pdf (duke.edu)) .

The Total Least Squares (TLS) algorithm, is a statistical method used to find the best-fitting linear relationship between variables when both the input and output signals present white noise pertubation. Unlike ordinary least squares (OLS), which assumes that only the dependent variable is subject to error, TLS considers errors in all measured variables, providing a more robust solution in many practical applications. The algorithm was proposed by Golub and Van Loan.

In TLS, we assume errors in both $\mathbf{X}$ and $\mathbf{Y}$, denoted as $\Delta\mathbf{X}$ and $\Delta\mathbf{Y}$, respectively. The true model becomes:

$$\mathbf{Y} + \Delta\mathbf{Y} = (\mathbf{X} + \Delta\mathbf{X})\mathbf{B} \tag{3.11}$$

Rearranging, we get:

$$\Delta\mathbf{Y} = \Delta\mathbf{X}\mathbf{B} \tag{3.12}$$

Objective Function

The TLS solution minimizes the Frobenius norm of the total perturbations in $\mathbf{X}$ and $\mathbf{Y}$:

$$\min_{\Delta\mathbf{X},\Delta\mathbf{Y}} \|[\Delta\mathbf{X}, \Delta\mathbf{Y}]\|_F \tag{3.13}$$

subject to:

$$(\mathbf{X} + \Delta\mathbf{X})\mathbf{B} = \mathbf{Y} + \Delta\mathbf{Y} \tag{3.14}$$

where $\|\cdot\|_F$ denotes the Frobenius norm.

Classical Solution

The classical approach to solve the TLS problem is by using Singular Value Decomposition (SVD). The augmented matrix $[\mathbf{X}, \mathbf{Y}]$ is decomposed as:

$$[\mathbf{X}, \mathbf{Y}] = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^T \tag{3.15}$$

where $\mathbf{U}$ is an $n \times n$ orthogonal matrix, $\Sigma = \operatorname{diag}(\sigma_1, \ldots, \sigma_{n+d})$ is a diagonal matrix of singular values; and $\mathbf{V}$ is an orthogonal matrix defined as

$$V := \begin{bmatrix} V_{11} & V_{12} \\ V_{21} & V_{22} \end{bmatrix} \qquad \text{and} \qquad \Sigma := \begin{bmatrix} \Sigma_1 & 0 \\ 0 & \Sigma_2 \end{bmatrix}. \tag{3.16}$$

A total least squares solution exists if and only if V_{22} is non-singular. In addition, it is unique if and only if $\sigma_n \neq \sigma_{n+1}$. In the case when the total least squares solution exists and is unique, it is given by

$$\widehat{X}_{\text{tls}} = -V_{12}V_{22}^{-1} \tag{3.17}$$

and the corresponding total least squares correction matrix is

$$\Delta C_{\text{tls}} := [\Delta A_{\text{tls}} \quad \Delta B_{\text{tls}}] = -U \operatorname{diag}(0, \Sigma_2) V^{\top}. \tag{3.18}$$

This is implemented in SysIdentPy as follows:

```
def optimize(self, psi: np.ndarray, y: np.ndarray) -> np.ndarray:
        """Estimate the model parameters using Total Least Squares method.

        Parameters
        ----------
        psi : ndarray of floats
                The information matrix of the model.
        y : array-like of shape = y_training
                The data used to training the model.

        Returns
        -------
```

```
        theta : array-like of shape = number_of_model_elements
                The estimated parameters of the model.

        """
        self._check_linear_dependence_rows(psi)
        full = np.hstack((psi, y))
        n = psi.shape[1]
        _, _, v = np.linalg.svd(full, full_matrices=True)
        theta = -v.T[:n, n:] / v.T[n:, n:]
        return theta.reshape(-1, 1)
```

To use it in the modeling task, just import it like we did in the Least Squares example.

From now on the examples will not include the Model Structure Selection step. The goal here is to focus on the parameter estimation methods. However, we already provided an example including MSS in the Least Squares section, so you will not have any problem to test that with other parameter estimation algorithms.

```
from sysidentpy.utils.generate_data import get_siso_data
from sysidentpy.model_structure_selection import FROLS
from sysidentpy.basis_function._basis_function import Polynomial
from sysidentpy.parameter_estimation import TotalLeastSquares
from sysidentpy.narmax_base import InformationMatrix


x_train, x_test, y_train, y_test = get_siso_data(
        n=1000, colored_noise=False, sigma=0.001, train_percentage=90
)
regressor_matrix = InformationMatrix(xlag=2, ylag=2).build_input_output_matrix(
        X=x_train, y=y_train
)
max_lag = 2
# apply the basis function
psi = Polynomial(degree=2).fit(regressor_matrix, max_lag=max_lag)
theta = TotalLeastSquares().optimize(psi, y_train[max_lag:, :])
theta

array([
   [-4.29547026e-05],
   [ 2.00085880e-01],
   [-5.20210280e-05],
   [ 4.03019271e-05],
   [ 8.99982943e-01],
   [-1.49492072e-05],
   [ 9.72399351e-05],
   [ 1.00072619e-01],
   [ 7.25253323e-05],
   [-1.37338363e-04],
```

```
    [ 1.37115512e-04],
    [ 2.42269843e-04],
    [ 1.20466767e-04],
    [ 6.32937185e-05],
    [ 1.36101591e-04]])
```

Recursive Least Squares

Consider the regression model

$$y_k = \mathbf{\Psi}_k^T \theta_k + \epsilon_k \tag{3.19}$$

where:

- y_k is the observed output at time $ k $.
- $\mathbf{\Psi}_k$ is the information matrix at time k.
- θ_k is the parameter vector to be estimated at time k.
- ϵ_k is the noise at time k.

The Recursive Least Squares (RLS) algorithm updates the parameter estimate θ_k recursively as new data points $(\mathbf{x}_k, y_k)$ become available, minimizing a weighted linear least squares cost function relating to the information matrix in a sequencial manner. RLS is particularly useful in real-time applications where the data arrives sequentially and the model needs continuous updating or for modeling time varying systems (if the forgetting factor is included).

Because its a recursive estimation, it is useful to relate $\hat{\Theta}_k$ to $\hat{\Theta}_{k-1}$. In other words, the new $\hat{\Theta}_k$ depends on the last estimated value (k). Moreover, to estimate $\hat{\Theta}_k$, we need to incorporate the current information present in y_k.

Aguirre BOOK defines the Recursive Least Squares estimator with forgetting factor λ as

$$\begin{cases} K_k = Q_k \psi_k = \frac{P_{k-1}\psi_k}{\psi_k^{\mathrm{T}} P_{k-1}\psi_k + \lambda} \\ \hat{\theta}_k = \hat{\theta}_{k-1} + K_k \left[y(k) - \psi_k^{\mathrm{T}} \hat{\theta}_{k-1} \right] \\ P_k = \frac{1}{\lambda}\left(P_{k-1} - \frac{P_{k-1}\psi_k \psi_k^{\mathrm{T}} P_{k-1}}{\psi_k^{\mathrm{T}} P_{k-1}\psi_k + \lambda} \right) \end{cases} \tag{3.20}$$

where K_k is the gain vector calculation (also known as Kalman gain), P_k is the covariance matrix update, and $y_k - \mathbf{\Psi}_k^T \theta_{k-1}$ is the a priori estimation error. The forgetting factor λ $(0 < \lambda \leq 1)$ is usually defined between 0.94 and 0.99. If you set $\lambda = 1$ you will be using the traditional recursive algorithm. The equation above consider that the regressor vector $\psi(k-1)$ has been rewritten as ψ_k, since this vector is updated at iteration k and contains information up to time instant $k-1$. We can Initialize the parameter estimate θ_0 as

$$\theta_0 = \mathbf{0} \tag{3.21}$$

and Initialize the inverse of the covariance matrix $\mathbf{P}_0$ with a large value:

$$\mathbf{P}_0 = \frac{\mathbf{I}}{\delta} \tag{3.22}$$

where δ is a small positive constant, and $\mathbf{I}$ is the identity matrix.

The forgetting factor λ controls how quickly the algorithm forgets past data:

- $\lambda = 1$ means no forgetting, and all past data are equally weighted.
- $\lambda < 1$ means that when new data is available, all weights are multiplied by λ, which can be interpreted as the ratio between consecutive weights for the same data.

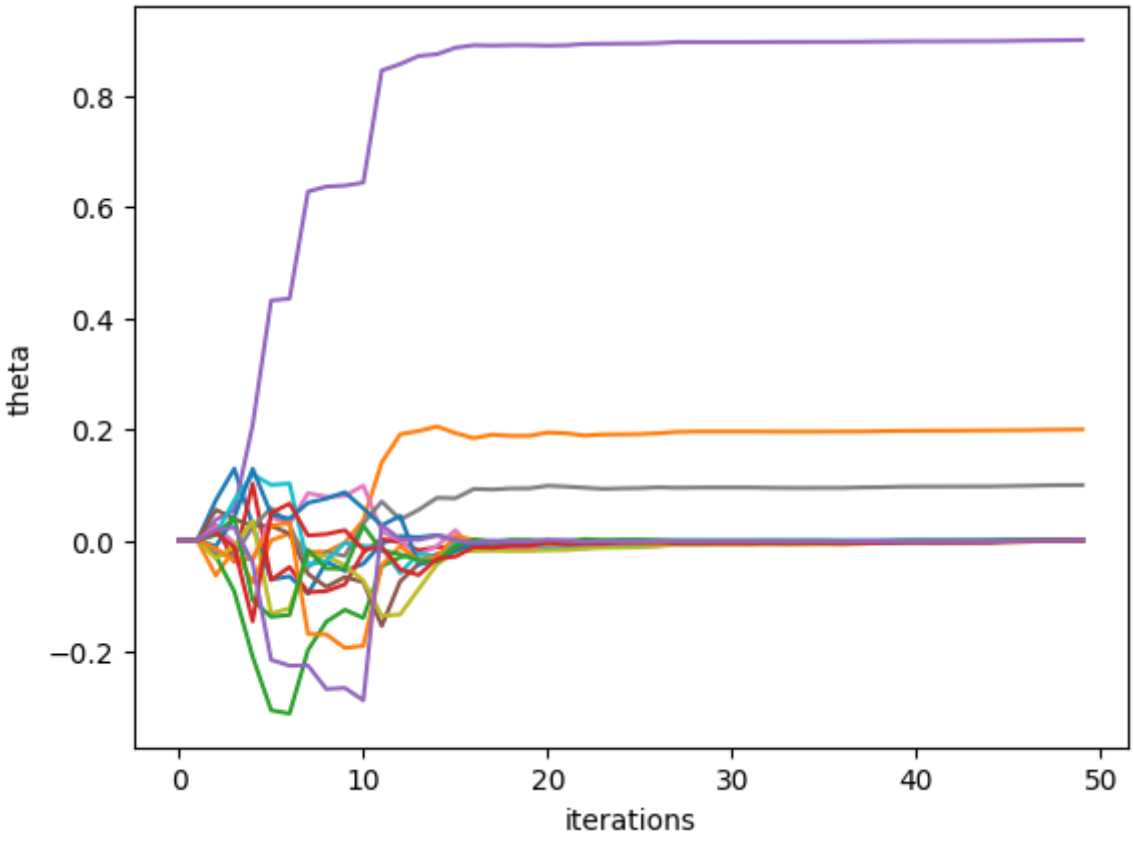
You can access the source code to check how SysIdentPy implements the RLS algorithm. The following example present how you can use it in SysIdentPy.

```
from sysidentpy.utils.generate_data import get_siso_data
from sysidentpy.model_structure_selection import FROLS
from sysidentpy.basis_function._basis_function import Polynomial
from sysidentpy.parameter_estimation import RecursiveLeastSquares
from sysidentpy.narmax_base import InformationMatrix
import matplotlib.pyplot as plt

x_train, x_test, y_train, y_test = get_siso_data(
    n=1000, colored_noise=False, sigma=0.001, train_percentage=90
)

regressor_matrix = InformationMatrix(xlag=2, ylag=2).build_input_output_matrix(
    X=x_train, y=y_train
)
max_lag = 2
psi = Polynomial(degree=2).fit(regressor_matrix, max_lag=max_lag)
estimator = RecursiveLeastSquares(lam=0.99)
theta = estimator.optimize(psi, y_train[max_lag:, :])
theta

array(
  [
  [-1.42674741e-04],
  [ 2.00108231e-01],
  [-7.15658788e-05],
  [-2.63118243e-05],
  [ 9.00023979e-01],
  [ 3.38729072e-04],
  [ 8.22835678e-07],
  [ 9.95974120e-02],
  [ 8.52966331e-05],
  [-1.47007370e-04],
  [-4.65885373e-04],
```

```
    [-2.57612698e-04],
    [-2.61078457e-04],
    [ 4.78599408e-04],
    [ 3.50698606e-04]
    ]
    )
```

You can plot the evolution of the estimated parameters over time by accessing the `theta_evolution` values

```
# plotting only the first 50 values
plt.plot(estimator.theta_evolution.T[:50, :])
plt.xlabel("iterations")
plt.ylabel("theta")
```



> Figure 1. Evolution of the estimated parameters over time using the RLS algorithm.

Least Mean Squares

The Least Mean Squares (LMS) adaptive filter is a popular stochastic gradient algorithm developed by Widrow and Hoff in 1960. The LMS adaptive filter aims to adaptively change its filter coefficients to achieve the best possible filtering of a signal. This is done by minimizing the error between the desired signal $d(n)$ and the filter output $y(n)$. We can derive the LMS algorithm from the RLS formulation.

In RLS, the λ is related to the minimization of the sum of weighted squares of the innovation

$$J_k = \sum_{j=1}^{k} \lambda^{k-j} e_j^2. \tag{3.23}$$

The Q_k in Equation 3.20, defined as

$$Q_k = \frac{P_{k-1}}{\psi_k^{\mathrm{T}} P_{k-1} \psi_k + \lambda} \tag{3.24}$$

is derived from the general form the Kalman Filter (KF) algorithm.

$$Q_k = \frac{P_{k-1}}{\psi_k^{\mathrm{T}} P_{k-1} \psi_k + v_0} \tag{3.25}$$

where v_0 is the variance of the noise in the definition of the KF, in which the cost function is defined as the sum of squares of the innovation (noise). You can check the details in the [Billings, S. A. book] (Nonlinear System Identification: NARMAX Methods in the Time, Frequency, and Spatio-Temporal Domains)

If we change Q_k in Equation 3.25 to scaled identity matrix

$$Q_k = \frac{\mu}{\|\psi_k\|^2} I \tag{3.26}$$

where $\mu \in \mathbb{R}^+$, the Q_k and $\hat{\theta}_k$ in Equation 3.20 becomes

$$\hat{\theta}_k = \hat{\theta}_{k-1} + \frac{\mu \left[y(k) - \psi_k^{\mathrm{T}} \hat{\theta}_{k-1} \right]}{\|\psi_k\|^2} \psi_k \tag{3.27}$$

where $\psi_k^{\mathrm{T}} \hat{\theta}_{k-1} = \hat{y}_k$, which is known as the LMS algorithm.

Convergence and Step-Size

The step-size parameter μ plays a crucial role in the performance of the LMS algorithm. If μ is too large, the algorithm may become unstable and fail to converge. If μ is too small, the algorithm will converge slowly. The choice of μ is typically:

$$0 < \mu < \frac{2}{\lambda_{\max}} \tag{3.28}$$

where $\lambda_{\max}$ is the largest eigenvalue of the input signal's autocorrelation matrix.

In SysIdentPy, you can use several variants of the LMS algorithm:

1. **LeastMeanSquareMixedNorm**
2. **LeastMeanSquares**
3. **LeastMeanSquaresFourth**

4. **LeastMeanSquaresLeaky**
5. **LeastMeanSquaresNormalizedLeaky**
6. **LeastMeanSquaresNormalizedSignRegressor**
7. **LeastMeanSquaresNormalizedSignSign**
8. **LeastMeanSquaresSignError**
9. **LeastMeanSquaresSignSign**
10. **AffineLeastMeanSquares**
11. **NormalizedLeastMeanSquares**
12. **NormalizedLeastMeanSquaresSignError**
13. **LeastMeanSquaresSignRegressor**

To use any one on the methods above, you just need to import it and set the `estimator` using the option you want:

```
from sysidentpy.utils.generate_data import get_siso_data
from sysidentpy.model_structure_selection import FROLS
from sysidentpy.basis_function._basis_function import Polynomial
from sysidentpy.parameter_estimation import LeastMeanSquares
from sysidentpy.narmax_base import InformationMatrix
import matplotlib.pyplot as plt

x_train, x_test, y_train, y_test = get_siso_data(
    n=1000, colored_noise=False, sigma=0.001, train_percentage=90
)

regressor_matrix = InformationMatrix(xlag=2, ylag=2).build_input_output_matrix(
    X=x_train, y=y_train
)
max_lag = 2
psi = Polynomial(degree=2).fit(regressor_matrix, max_lag=max_lag)
estimator = LeastMeanSquares(mu=0.1)
theta = estimator.optimize(psi, y_train[max_lag:, :])
theta

array([[ 7.53481862e-05],
       [ 2.00336327e-01],
       [ 4.03075867e-04],
       [ 5.16401623e-05],
       [ 8.99539417e-01],
       [-6.38551780e-04],
       [-5.68962465e-05],
       [ 1.00643282e-01],
       [ 5.48097106e-04],
       [ 5.15983109e-04],
       [ 5.46693676e-04],
       [-3.18759644e-04],
```

```
       [-4.90420481e-04],
       [ 4.09662242e-04],
       [ 5.39826714e-04]])
```

Extended Least Squares Algorithm

Let's show an example of the effect of a biased parameter estimation. To make things simple,The data is generated by simulating the following model:

$$y_k = 0.2y_{k-1} + 0.1y_{k-1}x_{k-1} + 0.9x_{k-2} + e_k$$

In this case, we know the values of the true parameters, so it will be easier to understand how they are affected by a biased estimation. The data is generated using a method from SysIdentPy. If *colored_noise* is set to True in the method, a colored noise is added to the data:

$$e_k = 0.8\nu_{k-1} + \nu_k$$

where x is a uniformly distributed random variable and ν is a gaussian distributed variable with $\mu = 0$ and σ is defined by the user.

We will generate a data with 1000 samples with white noise and selecting 90% of the data to train the model.

```
x_train, x_valid, y_train, y_valid = get_siso_data(
    n=1000, colored_noise=True, sigma=0.2, train_percentage=90
)
```

First we will train a model without the Extended Least Squares Algorithm for comparison purpose.

```
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import pandas as pd
from sysidentpy.model_structure_selection import FROLS
from sysidentpy.basis_function._basis_function import Polynomial
from sysidentpy.parameter_estimation import LeastSquares
from sysidentpy.utils.generate_data import get_siso_data
from sysidentpy.utils.display_results import results

basis_function = Polynomial(degree=2)
estimator = LeastSquares(unbiased=False)
model = FROLS(
    order_selection=False,
    n_terms=3,
    ylag=2,
    xlag=2,
```

```
    info_criteria="aic",
    estimator=estimator,
    basis_function=basis_function,
    err_tol=None,
)

model.fit(X=x_train, y=y_train)
yhat = model.predict(X=x_valid, y=y_valid)
rrse = root_relative_squared_error(y_valid, yhat)

r = pd.DataFrame(
    results(
        model.final_model,
        model.theta,
        model.err,
        model.n_terms,
        err_precision=8,
        dtype="sci",
    ),
    columns=["Regressors", "Parameters", "ERR"],
)

print(r)
```

Regressors	Parameters	ERR
x1(k-2)	9.0442E-01	7.55518391E-01
y(k-1)	2.7405E-01	7.57565084E-02
x1(k-1)y(k-1)	9.8757E-02	3.12896171E-03

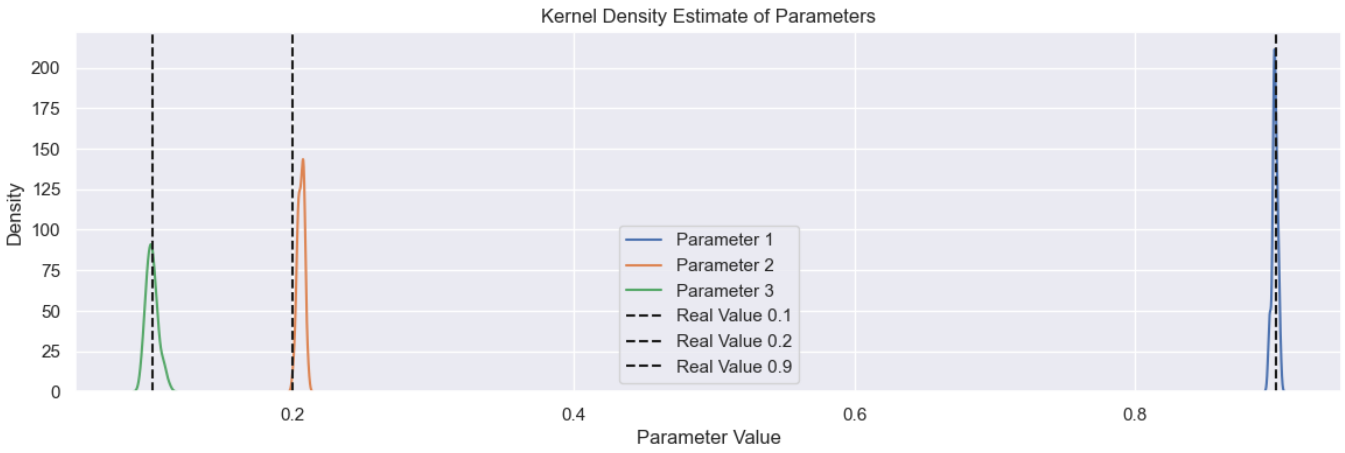
Clearly we have something wrong with the obtained model. The estimated parameters differs from the true one defined in the equation that generated the data. As we can observe above, the model structure is exact the same the one that generate the data. You can se that the ERR ordered the terms in the correct way. And this is an important note regarding the ERR algorithm: **it is very robust to colored noise!!**

That is a great feature! However, although the structure is correct, the model *parameters* are not correct! Here we have a biased estimation! For instance, the real parameter for y_{k-1} is 0.2, not 0.274.

In this case, we are actually modeling using a NARX model, not a NARMAX. The MA part exists to allow a unbiased estimation of the parameters. To achieve a unbiased estimation of the parameters we have the Extend Least Squares algorithm.

Before applying the Extended Least Squares Algorithm we will run several NARX models to check how different the estimated parameters are from the real ones.

```
parameters = np.zeros([3, 50])
for i in range(50):
    x_train, x_valid, y_train, y_valid = get_siso_data(
        n=3000, colored_noise=True, train_percentage=90
    )
    model.fit(X=x_train, y=y_train)
    parameters[:, i] = model.theta.flatten()

# Set the theme for seaborn (optional)
sns.set_theme()
plt.figure(figsize=(14, 4))
# Plot KDE for each parameter
sns.kdeplot(parameters.T[:, 0], label='Parameter 1')
sns.kdeplot(parameters.T[:, 1], label='Parameter 2')
sns.kdeplot(parameters.T[:, 2], label='Parameter 3')
# Plot vertical lines where the real values must lie
plt.axvline(x=0.1, color='k', linestyle='--', label='Real Value 0.1')
plt.axvline(x=0.2, color='k', linestyle='--', label='Real Value 0.2')
plt.axvline(x=0.9, color='k', linestyle='--', label='Real Value 0.9')
plt.xlabel('Parameter Value')
plt.ylabel('Density')
plt.title('Kernel Density Estimate of Parameters')
plt.legend()
plt.show()
```

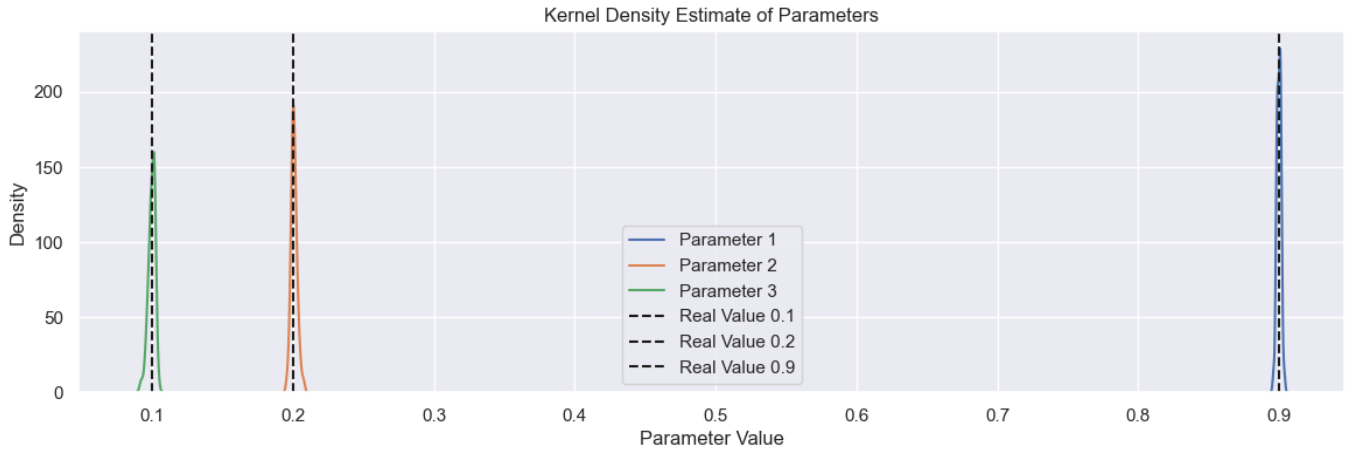


> Figure 2.: Kernel Density Estimates (KDEs) of the estimated parameters obtained from 50 NARX models realizations, each fitted to data with colored noise. The vertical dashed lines indicate the true parameter values used to generate the data. While the model structure is correctly identified, the estimated parameters are biased due to the omission of the Moving Average (MA) component, highlighting the need for the Extended Least Squares algorithm to achieve unbiased parameter estimation

As shown in figure above, we have a problem to estimate the parameter for y_{k-1}. Now we will use the Extended Least Squares Algorithm. In SysIdentPy, just set `unbiased=True` in the parameter

estimation definition and the ELS algorithm will be applied.

```
basis_function = Polynomial(degree=2)
estimator = LeastSquares(unbiased=True)
parameters = np.zeros([3, 50])
for i in range(50):
    x_train, x_valid, y_train, y_valid = get_siso_data(
        n=3000, colored_noise=True, train_percentage=90
    )
    model = FROLS(
        order_selection=False,
        n_terms=3,
        ylag=2,
        xlag=2,
        elag=2,
        info_criteria="aic",
        estimator=estimator,
        basis_function=basis_function,
    )

    model.fit(X=x_train, y=y_train)
    parameters[:, i] = model.theta.flatten()

plt.figure(figsize=(14, 4))
# Plot KDE for each parameter
sns.kdeplot(parameters.T[:, 0], label='Parameter 1')
sns.kdeplot(parameters.T[:, 1], label='Parameter 2')
sns.kdeplot(parameters.T[:, 2], label='Parameter 3')
# Plot vertical lines where the real values must lie
plt.axvline(x=0.1, color='k', linestyle='--', label='Real Value 0.1')
plt.axvline(x=0.2, color='k', linestyle='--', label='Real Value 0.2')
plt.axvline(x=0.9, color='k', linestyle='--', label='Real Value 0.9')
plt.xlabel('Parameter Value')
plt.ylabel('Density')
plt.title('Kernel Density Estimate of Parameters')
plt.legend()
plt.show()
```



> Figure 3. Kernel Density Estimates (KDEs) of the estimated parameters obtained from 50 NARX models using the Extended Least Squares (ELS) algorithm with unbiased estimation. The vertical dashed lines indicate the true parameter values used to generate the data.

Unlike the previous biased estimation, these KDEs in Figure 3 show that the estimated parameters are now closely aligned with the true values, demonstrating the effectiveness of the ELS algorithm in achieving unbiased parameter estimation, even in the presence of colored noise.

> The Extended Least Squares algorithm is iterative by nature. In SysIdentPy, the default number of iterations is set to 30 (`uiter=30`). However, the literature suggests that the algorithm typically converges quickly, often within 10 to 20 iterations. Therefore, you may want to test different numbers of iterations to find the optimal balance between convergence speed and computational efficiency.

4 - Model Structure Selection (MSS)

Introduction

> This section is taken mainly from my master thesis, which was based on Billings, S. A.

Selecting the model structure is crucial to develop models that can correctly reproduce the system behavior. If some prior information about the system are known, e.g., the dynamic order and degree of nonlinearity, determining the terms and then estimate the parameters is trivial. In real life scenarios, however, in most of the times there is no information about what terms should be included in the model and the correct regressors has to be selected in the identification framework. If the MSS is not performed with the necessary concerns, the scientific law that describes the system may will not be revealed and resulting in misleading interpretations about the system. To illustrate this scenario, consider the following example.

Let $\mathcal{D}$ denote an arbitrary dataset

$$\mathcal{D} = \{(x_k, y_k), k = 1, 2, \ldots, n\}, \tag{1}$$

where $x_k \in \mathbb{R}^{n_x}$ and $y_k \in \mathbb{R}^{n_y}$ are the input and output of an unknown system and n is the number of samples in the dataset. The following are two polynomial NARX models built to describe that system:

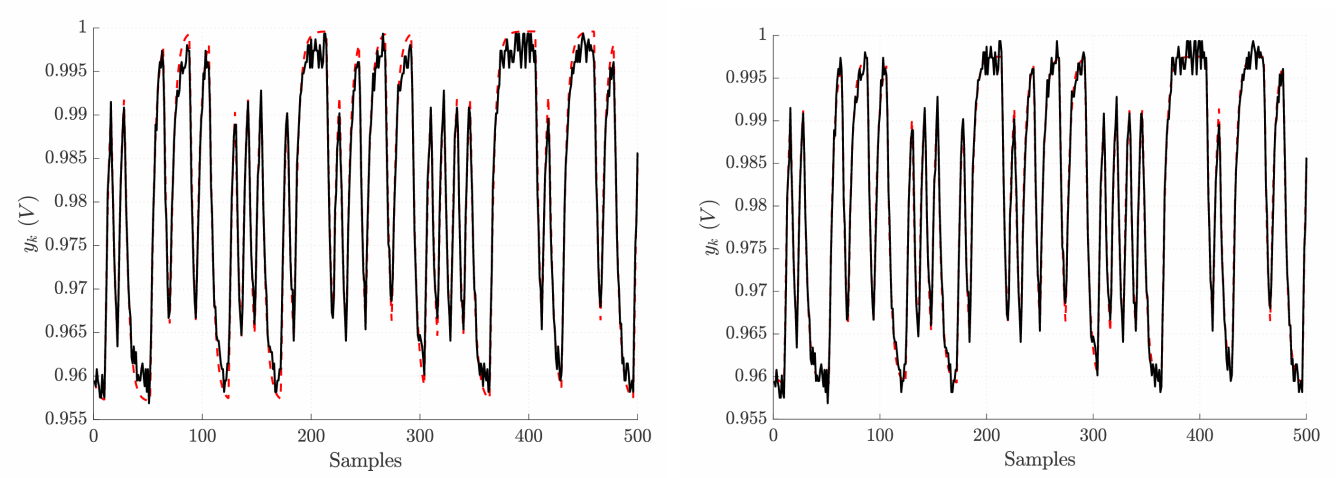
$$y_{ak} = 0.7077 y_{ak-1} + 0.1642 u_{k-1} + 0.1280 u_{k-2} \tag{2}$$

$$\begin{aligned} y_{bk} = {} & 0.7103 y_{bk-1} + 0.1458 u_{k-1} + 0.1631 u_{k-2} \\ & -1467 y^3_{bk-1} + 0.0710 y^3_{bk-2} + 0.0554 y^2_{bk-3} u_{k-3}. \end{aligned} \tag{3}$$

Figure 1 shows the predicted values of each model and the real data. As can be observed, the nonlinear model 2 seems to fit the data better than the linear model 1. The original system under consideration is an RLC circuit, consisting of a resistor (R), inductor (L), and capacitor (C) connected in series with a voltage source. It is well known that the behavior of such an RLC series circuit can be accurately described by a linear second-order differential equation that relates the current $I(t)$ and the applied voltage $V(t)$:

$$L\frac{d^2I(t)}{dt^2} + R\frac{dI(t)}{dt} + \frac{1}{C}I(t) = \frac{dV(t)}{dt} \tag{4}$$

Given this linear relationship, an adequate model for the RLC circuit should reflect this second-order linearity. While Model 2, which includes nonlinear terms, may provide a closer fit to the data, it is clearly over-parameterized. Such over-parameterization can introduce spurious nonlinear effects, often referred to as "ghost" nonlinearities, which do not correspond to the actual dynamics of the system. Therefore, these models need to be interpreted with caution, as the use of an overly complex model could obscure the true linear nature of the system and lead to incorrect conclusions about its behavior.



> Figure 1.Results for two polynomial NARX models fitted to data from an unknown system. Model 1 (left) is a linear model, while Model 2 (right) includes nonlinear terms. The figure illustrates that Model 2 provides a closer fit to the data compared to Model 1. However, since the original system is a linear RLC circuit known to have a second-order linear behavior, the improved fit of Model 2 may be misleading due to over-parameterization. This highlights the importance of considering the physical characteristics of the system when interpreting model results to avoid misinterpretation of artificial nonlinearities. Reference: Meta Model Structure Selection: An Algorithm For Building Polynomial NARX Models For Regression And Classification

Correctly identifying the structure of a model is crucial for accurately analyzing the system's dynamics. A well-chosen model structure ensures that the model reflects the true behavior of the system, allowing for consistent and meaningful analysis. In this respect, several algorithms have been developed to select the appropriate terms for constructing a polynomial NARX model. The primary goal of model structure selection (MSS) algorithms is to reveal the system's characteristics by producing the simplest model that adequately describes the data. While some systems may indeed require more complex models, it is essential to strike a balance between simplicity and accuracy. As Einstein aptly put it:

> A model should be as simple as possible, but not simpler.

This principle emphasizes the importance of avoiding unnecessary complexity while ensuring that the model still captures the essential dynamics of the system.

We see at chapter 2 that regressors selection, however, is not a simple task. If the nonlinear degree, the order of the model and the number inputs increases, the number of candidate models becomes too large for brute force approach. Considering the MIMO case, this problem is far worse than the SISO one if many inputs and outputs are required. The number of all different models can be calculated as

$$n_m = \begin{cases} 2^{n_r} & \text{for SISO models,} \\ 2^{n_{mr}} & \text{for MIMO models,} \end{cases} \tag{5}$$

where n_r and and n_{mr} are the values computed using the equations presented in Chapter 2.

A classical solution to regressors selection problem is the Forward Regression Orthogonal Least Squares (FROLS) algorithm associated with Error Reduction Ratio (ERR) algorithm. This technique is based on the Prediction Error Minimization framework and, one at time, select the most relevant regressor by using a step-wise regression. The FROLS method adapt the set of regressors in the search space into a set of orthogonal vectors, which ERR evaluates the individual contribution to the desired output variance.

The Forward Regression Orthogonal Least Squares Algorithm

Consider the general NARMAX model defined in Equation 2.23 described in a generic form as

$$y_k = \psi_{k-1}^{\top}\hat{\Theta} + \xi_k, \tag{6}$$

where $\psi_{k-1}^{\top} \in \mathbb{R}^{n_r \times n}$ is a vector of some combinations of the regressors and $\hat{\Theta} \in \mathbb{R}^{n_\Theta}$ the vector of estimated parameters. In a more compact form, the NARMAX model can be represented in a matrix form as:

$$y = \Psi\hat{\Theta} + \Xi, \tag{7}$$

where

$$Y = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix}, \Psi = \begin{bmatrix} \psi_{_1} \\ \psi_{_2} \\ \vdots \\ \psi_{_{n_\Theta}} \end{bmatrix}^{\top} = \begin{bmatrix} \psi_{_1}1 & \psi_{_2}1 & \dots & \psi_{_{n_\Theta}}1 \\ \psi_{_1}2 & \psi_{_2}2 & \dots & \psi_{_{n_\Theta}}2 \\ \vdots & \vdots & & \vdots \\ \psi_{_1}n & \psi_{_2}n & \dots & \psi_{_{n_\Theta}}n \end{bmatrix}, \hat{\Theta} = \begin{bmatrix} \hat{\Theta}_1 \\ \hat{\Theta}_2 \\ \vdots \\ \hat{\Theta}_{n_\Theta} \end{bmatrix}, \Xi = \begin{bmatrix} \xi_1 \\ \xi_2 \\ \vdots \\ \xi_n \end{bmatrix}. \tag{8}$$

The parameters in equation above could be estimated as a result of a Least Squares-based algorithm, but this would require to optimize all parameters at the same time on account of the fact of interaction between regressors due to non-orthogonality characteristic. Consequently, the computational demand becomes impractical for high number of regressors. In this respect, the FROLS transforms the non-orthogonal model presented in the equation above into a orthogonal one.

The regressor matrix Ψ can be orthogonally decomposed as

$$\Psi = QA, \tag{9}$$

where $A \in \mathbb{R}^{n_\Theta \times n_\Theta}$ is an unit upper triangular matrix according to

$$A = \begin{bmatrix} 1 & a_{12} & a_{13} & \dots & a_{1n_\Theta} \\ 0 & 1 & a_{23} & \dots & a_{2n_\Theta} \\ 0 & 0 & 1 & \dots & \vdots \\ \vdots & \vdots & \vdots & \ddots & a_{n_\Theta - 1 n_\Theta} \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}, \tag{10}$$

and $Q \in \mathbb{R}^{n \times n_\Theta}$ is a matrix with orthogonal columns q_i, described as

$$Q = \begin{bmatrix} q_1 & q_2 & q_3 & \dots & q_{n_\Theta} \end{bmatrix}, \tag{11}$$

such that $Q^\top Q = \Lambda$ and Λ is diagonal with entry d_i and can be expressed as:

$$d_i = q_i^\top q_i = \sum_{n}^{k=1} q_{ik} q_{ik}, \qquad 1 \leq i \leq n_\Theta.$$

Because the space spanned by the orthogonal basis Q (Equation 11) is the same as that spanned by the basis set Ψ (Equation 8) (i.e, contains every linear combination of elements of such subspace), we can define the Equation 7 as

$$Y = \underbrace{(\Psi A^{-1})}_{Q} \underbrace{(A\Theta)}_{g} + \Xi = Qg + \Xi, \tag{12}$$

where $g \in \mathbb{R}^{n_\Theta}$ is an auxiliary parameter vector. The solution of the model described in Equation 12 is given by

$$g = \left(Q^\top Q\right)^{-1} Q^\top Y = \Lambda^{-1} Q^\top Y \tag{13}$$

or

$$g_i = \frac{q_i^\top Y}{q_i^\top q_i}. \tag{14}$$

Since the parameter Θ and g satisfies the triangular system $A\Theta = g$, any orthogonalization method like Householder, Gram-Schmidt, modified Gram-Schmidt or Givens transformations can be used to solve the equation and estimate the original parameters. Assuming that $E[\Psi^\top \Xi] = 0$, the output variance can be derived by multiplying Equation 12 with itself and dividing by n, resulting in

$$\frac{1}{n} Y^\top Y = \underbrace{\frac{1}{n} \sum_{n_\Theta}^{i=1} g_i^2 q_i^\top q_i}_{\text{output explained by the regressors}} + \underbrace{\frac{1}{n} \Xi^\top \Xi}_{\text{unexplained variance}}. \tag{15}$$

Thus, the ERR due to the inclusion of the regressor q_i is expressed as:

$$[\mathrm{ERR}]_i = \frac{g_i^2 \cdot q_i^\top q_i}{Y^\top Y}, \qquad \text{for } i = 1, 2, \dots, n_\Theta.$$

There are many ways to terminate the algorithm. An approach often used is stop the algorithm if the model output variance drops below some predetermined limit ε:

$$1 - \sum_{i=1}^{n_\Theta} \mathrm{ERR}_i \leq \varepsilon, \tag{17}$$

Keep it simple

For the sake of simplicity, let's present the FROLS along with simple examples to make the intuition clear. First, let define the ERR calculation and then explain the idea of the FRLOS in simple terms.

Orthogonal case

Consider the case where we have a set of inputs defined as $x_1, x_2, \ldots, x_n$ and an output called y. These inputs are orthogonal vectors.

Lets suppose that we want to create a model to approximate y using $x_1, x_2, \ldots, x_n$, as follows:

$$y = \hat{\theta}_1 x_1 + \hat{\theta}_2 x_2 + \ldots + \hat{\theta}_n x_n + e \tag{18}$$

where $\hat{\theta}_1, \hat{\theta}_2, \ldots, \hat{\theta}_n$ are parameters and e is white noise and independent of x and y (remember the $E[\Psi^\top \Xi] = 0$, in previous section). In this case, we can rewrite the equation above as

$$y = \hat{\theta} x \tag{19}$$

so

$$\langle x, y \rangle = \left\langle \hat{\theta} x, x \right\rangle = \hat{\theta} \langle x, x \rangle \tag{20}$$

Which implies that

$$\hat{\theta} = \frac{\langle x, y \rangle}{\langle x, x \rangle} \tag{21}$$

Therefore we can show that

$$\begin{aligned}
\langle x_1, y \rangle = \hat{\theta}_1 \langle x_1, x_1 \rangle \Rightarrow \hat{\theta}_1 = \frac{\langle x_1, y \rangle}{\langle x_1, x_1 \rangle} = \frac{x_1^T y}{x_1^T x_1} \\
\langle x_2, y \rangle = \hat{\theta}_2 \langle x_2, x_2 \rangle \Rightarrow \hat{\theta}_2 = \frac{\langle x_2, y \rangle}{\langle x_2, x_2 \rangle} = \frac{x_2^T y}{x_2^T x_2}, \ldots \\
\langle x_n, y \rangle = \hat{\theta}_n \langle x_n, x_n \rangle \Rightarrow \hat{\theta}_n = \frac{\langle x_n, y \rangle}{\langle x_n, x_n \rangle} = \frac{x_n^T y}{x_n^T x_n},
\end{aligned} \tag{22}$$

Following the same idea, we can also show that

$$\langle y, y \rangle = \hat{\theta}_1^2 \langle x_1, x_1 \rangle + \hat{\theta}_2^2 \langle x_2, x_2 \rangle + \ldots + \hat{\theta}_n^2 \langle x_n, x_n \rangle + \langle e, e \rangle \tag{23}$$

which can be described as

$$y^T y = \hat{\theta}_1^2 x_1^T x_1 + \hat{\theta}_2^2 x_2^T x_2 + \ldots + \hat{\theta}_n^2 x_n^T x_n + e^T e \tag{24}$$

or

$$\|y\|^2 = \hat{\theta}_1^2 \|x_1\|^2 + \hat{\theta}_2^2 \|x_2\|^2 + \ldots + \hat{\theta}_n^2 \|x_n\|^2 + \|e\|^2 \tag{25}$$

So, dividing both sides of the equation by y and rearranging the equation, we have

$$\frac{\|e\|^2}{\|y\|^2} = 1 - \hat{\theta}_1^2 \frac{\|x_1\|^2}{\|y\|^2} - \hat{\theta}_2^2 \frac{\|x_2\|^2}{\|y\|^2} - \ldots - \hat{\theta}_n^2 \frac{\|x_n\|^2}{\|y\|^2} \tag{26}$$

Because $\hat{\theta}_k = \frac{x_k^T y}{x_k^T x_k} = \frac{x_k^T y}{\|x_k\|^2}, k = 1, 2, .., n$, we have

$$\begin{aligned} \frac{\|e\|^2}{\|y\|^2} &= 1 - \left(\frac{x_1^T y}{\|x_1\|^2} \right)^2 \frac{\|x_1\|^2}{\|y\|^2} - \left(\frac{x_2^T y}{\|x_2\|^2} \right)^2 \frac{\|x_2\|^2}{\|y\|^2} - \ldots - \left(\frac{x_n^T y}{\|x_n\|^2} \right)^2 \frac{\|x_n\|^2}{\|y\|^2} \\ &= 1 - \frac{\left(x_1^T y\right)^2}{\|x_1\|\,\|^2 \|y\|^2} - \frac{\left(x_2^T y\right)^2}{\|x_2\|^2 \|y\|^2} - \ldots - \frac{\left(x_n^T y\right)^2}{\|x_n\|^2 \|y\|^2} \\ &= 1 - ERR_1 \quad - ERR_2 - \cdots - ERR_n \end{aligned} \tag{27}$$

where $\mathrm{ERR}_k (k = 1, 2 \ldots, n)$ is the Error Reduction Ratio defined in previous section.

Check the example bellow using the fundamental basis

```
import numpy as np

y = np.array([3, 7, 8])
# Orthogonal Basis
x1 = np.array([1, 0, 0])
x2 = np.array([0, 1, 0])
x3 = np.array([0, 0, 1])

theta1 = (x1.T@y)/(x1.T@x1)
theta2 = (x2.T@y)/(x2.T@x2)
theta3 = (x3.T@y)/(x3.T@x3)

squared_y = y.T @ y
err1 = (x1.T@y)**2/((x1.T@x1) * squared_y)
err2 = (x2.T@y)**2/((x2.T@x2) * squared_y)
err3 = (x3.T@y)**2/((x3.T@x3) * squared_y)

print(f"x1 represents {round(err1*100, 2)}% of the variation in y, \n x2 represents
{round(err2*100, 2)}% of the variation in y, \n x3 represents {round(err3*100, 2)}%
of the variation in y")
```

```
x1 represents 7.38% of the variation in y,
x2 represents 40.16% of the variation in y,
x3 represents 52.46% of the variation in y
```

Lets see what happens in a non-orthogonal scenario.

```
y = np.array([3, 7, 8])
x1 = np.array([1, 2, 2])
x2 = np.array([-1, 0, 2])
x3 = np.array([0, 0, 1])
```

```
theta1 = (x1.T@y)/(x1.T@x1)
theta2 = (x2.T@y)/(x2.T@x2)
theta3 = (x3.T@y)/(x3.T@x3)

squared_y = y.T @ y
err1 = (x1.T@y)**2/((x1.T@x1) * squared_y)
err2 = (x2.T@y)/((x2.T@x2) * squared_y)
err3 = (x3.T@y)**2/((x3.T@x3) * squared_y)

print(f"x1 represents {round(err1*100, 2)}% of the variation in y, \n x2 represents
{round(err2*100, 2)}% of the variation in y, \n x3 represents {round(err3*100, 2)}%
of the variation in y")

>>> x1 represents 99.18% of the variation in y,
>>> x2 represents 2.13% of the variation in y,
>>> x3 represents 52.46% of the variation in y
```

In this case, $x1$ have the highest err value, so we have choose it to be the first orthogonal vector.

```
q1 = x1.copy()

v1 = x2 - (q1.T@x2)/(q1.T@q1)*q1
errv1 = (v1.T@y)**2/((v1.T@v1) * squared_y)

v2 = x3 - (q1.T@x3)/(q1.T@q1)*q1
errv2 = (v2.T@y)**2/((v2.T@v2) * squared_y)

print(f"v1 represents {round(errv1*100, 2)}% of the variation in y, \n v2 represents
{round(errv2*100, 2)}% of the variation in y")

>>> v1 represents 0.82% of the variation in y,
>>> v2 represents 0.66% of the variation in y
```

So, in this case, when we sum the err values of the first two orthogonal vectors, $x1$ and $v1$, we get $err_3 + errv1 = 100\%$. Then there is no need to keep the iterations looking for more terms. The model with this two terms already explain all the variance in the data.

> That's the idea of the FROLS algorithm. We calculate the ERR, choose the vector with the highest ERR to be the first orthogonal vector, orthogonalize every vector but the one we choose in the first step, calculate the ERR for each one of them, choose the vector with the highest ERR value and keep doing that until we reach some criteria.

In SysIdentPy, we have 2 hyperparameters called `n_terms` and `err_tol`. Both of them can be used to stop the iterations. The first one will iterate until `n_terms` are chosen. The second one iterate until the $\sum ERR_i > err_{tol}$. If you set both, the algorithm stop when any of the conditions is true.

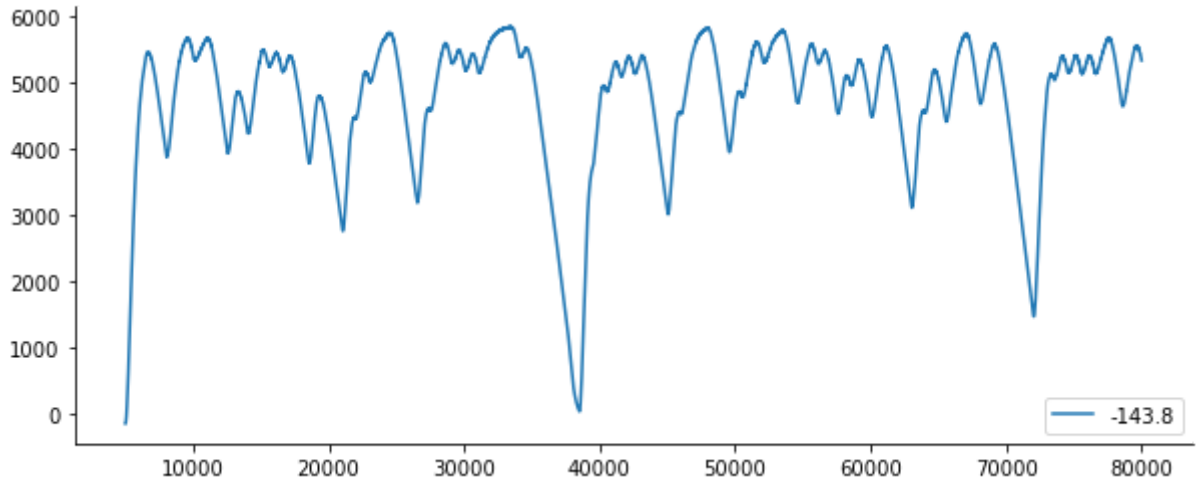
```
model = FROLS(
        n_terms=50,
        ylag=7,
        xlag=7,
        basis_function=basis_function,
        err_tol=0.98
    )
```

SysIdentPy apply the Golub -Householder method for the orthogonal decomposition. A more detailed discussion about Householder and orthogonalization procedures in general can be found in Chen, S. and Billings, S. A. and Luo, W.

Case Study

An example using real data will be described using SysIdentPy. In this example, we will build models linear and nonlinear models to describe the behavior of a DC motor operating as generator. Details of the experiment used to generate this data can be found in the paper (in Portuguese) IDENTIFICAÇÃO DE UM MOTOR/GERADOR CC POR MEIO DE MODELOS POLINOMIAIS AUTORREGRESSIVOS E REDES NEURAIS ARTIFICIAIS

```
import numpy as np
import pandas as pd
from sysidentpy.model_structure_selection import FROLS
from sysidentpy.basis_function._basis_function import Polynomial
from sysidentpy.parameter_estimation import LeastSquares
from sysidentpy.utils.display_results import results
from sysidentpy.utils.plotting import plot_results

df1 = pd.read_csv("examples/datasets/x_cc.csv")
df2 = pd.read_csv("examples/datasets/y_cc.csv")

# checking the ouput
df2[5000:80000].plot(figsize=(10, 4))
```



> Figure 2. Output of the electromechanical system.

In this example, we will decimate the data using $d = 500$. The rationale behind decimation here is that the data is oversampled due to the experimental setup. A future section will provide a detailed explanation of how to handle oversampled data in the context of system identification. For now, consider this approach as the most appropriate solution.

```
x_train, x_valid = np.split(df1.iloc[::500].values, 2)
y_train, y_valid = np.split(df2.iloc[::500].values, 2)
```

In this case, we will build a NARX model. In SysIdentPy, this means setting `unbiased=False` in the `LeastSquares` definition. We'll use a `Polynomial` basis function and set the maximum lag for both input and output to 2. This configuration results in 15 terms in the information matrix, so we'll set `n_terms=15`. This specification is necessary because, in this example, `order_selection` is set to `False`. We will discuss `order_selection` in more detail in the Information Criteria section later on.

> `order_selection` is `True` by default in SysIdentPy. When `order_selection=False` the user must pass a values to `n_terms` because it is an optional argument and its default value is `None`. If we set `n_terms=5`, for exemple, the FROLS will stop after choosing the first 5 regressors. We do not want that in this case because we want the FROLS stop only when `e_tol` is reached.

```
basis_function = Polynomial(degree=2)

model = FROLS(
        order_selection=False,
    ylag=2,
    xlag=2,
    estimator=LeastSquares(unbiased=False),
    basis_function=basis_function,
    e_tol=0.9999
    n_terms=15
)
```

SysIdentPy aims to simplify the use of algorithms like `FROLS` for the user. Building, training, or fitting a model is made straightforward through a simple interface called `fit`. By using this method, the entire process is handled internally, requiring no further interaction from the user.

```
model.fit(X=x_train, y=y_train)
```

SysIdentPy also offers a method to retrieve detailed information about the fitted model. Users can check the terms included in the model, the estimated parameters, the Error Reduction Ratio (ERR) values, and more.

> We're using `pandas` here only to make the output more readable, but it's optional.

```
r = pd.DataFrame(
    results(
        model.final_model,
        model.theta,
        model.err,
        model.n_terms,
        err_precision=8,
        dtype="sci",
    ),
    columns=["Regressors", "Parameters", "ERR"],
)

print(r)
```

Regressors	Parameters	ERR
y(k-1)	1.0998E+00	9.86000384E-01
x1(k-1)^2	1.0165E+02	7.94805130E-03
y(k-2)^2	-1.9786E-05	2.50905908E-03
x1(k-1)y(k-1)	-1.2138E-01	1.43301039E-03
y(k-2)	-3.2621E-01	1.02781443E-03
x1(k-1)y(k-2)	5.3596E-02	5.35200312E-04
x1(k-2)	3.4655E+02	2.79648078E-04
x1(k-2)y(k-1)	-5.1647E-02	1.12211942E-04
x1(k-2)x1(k-1)	-8.2162E+00	4.54743448E-05
y(k-2)y(k-1)	4.0961E-05	3.25346101E-05

> Table 1

The table above shows that 10 regressors (out of the 15 available) were needed to reach the defined `e_tol`, with the sum of the ERR for the selected regressors being 0.99992.

Next, let's evaluate the model's performance using the test data. Similar to the `fit` method, SysIdentPy provides a `predict` method. To obtain the predicted values and plot the results, simply follow these steps:

```
yhat = model.predict(X=x_valid, y=y_valid)
# plot only the first 100 samples (n=100)
plot_results(y=y_valid, yhat=yhat, n=100)
```

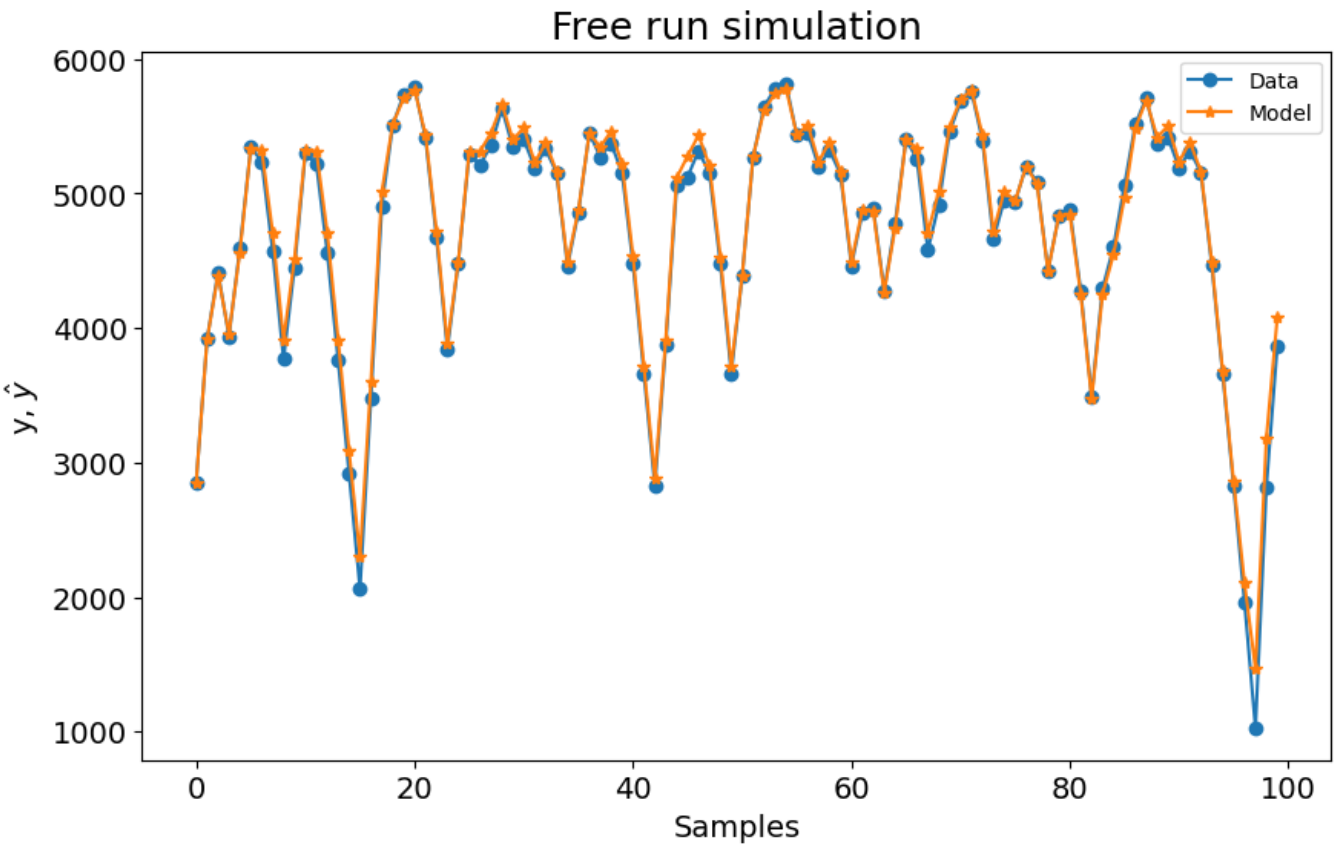


Figure 3. Free run simulation (or infinity-steps ahead prediction) of the fitted model.

Information Criteria

We said that there are many ways to terminate the algorithm and select the model terms, but only ERR criteria was defined in previous section. Different ways to terminate the algorithm is by using some information criteria, e.g, Akaike Information Criteria (AIC). For Least Squares based regression analysis, the AIC indicates the number of regressors by minimizing the objective function ([Akaike, H.] (A new look at the statistical model identification | IEEE Journals & Magazine | IEEE Xplore)):

$$J_{\text{AIC}} = \underbrace{n \log\left(Var[\xi_k]\right)}_{\text{first component}} + \underbrace{2n_{\Theta}}_{\text{second component}} \quad . \tag{28}$$

It is important to note that the equation above illustrates a trade-off between model fit and model complexity. Specifically, this trade-off involves balancing the model's ability to accurately fit the data (the first component) against its complexity, which is related to the number of parameters included

(the second component). As additional terms are included in the model, the Akaike Information Criterion (AIC) value initially decreases, reaching a minimum that represents an optimal balance between model complexity and predictive accuracy. However, if the number of parameters becomes excessive, the penalty for complexity outweighs the benefit of a better fit, causing the AIC value to increase. The AIC and many others variants have been extensively used for linear and nonlinear system identification. Check [Wei, H. and Zhu, D. and Billings, S. A. and Balikhin, M. A.](Forecasting the geomagnetic activity of the Dst index using multiscale radial basis function networks), [Martins, S. A. M. and Nepomuceno, E. G. and Barroso, M. F. S.](Improved Structure Detection For Polynomial NARX Models Using a Multiobjective Error Reduction Ratio), [Hafiz, F. and Swain, A. and Mendes, E. M. A. M. and Patel, N.](Structure Selection of Polynomial NARX Models Using Two Dimensional (2D) Particle Swarms), [Gu, Y. and Wei, H. and Balikhin, M. M.](Nonlinear predictive model selection and model averaging using information criteria) and references therein.

Despite their effectiveness in many linear model selection scenarios, information criteria such as AIC can struggle to select an appropriate number of parameters when dealing with systems exhibiting significant nonlinear behavior. Additionally, these criteria may lead to suboptimal models if the search space does not encompass all the necessary terms required to accurately represent the *true* model. Consequently, in highly nonlinear systems or when critical model components are missing, information criteria might not provide reliable guidance, resulting in models that exhibit poor performance.

Besides AIC, SysIdentPy provides other four different information criteria: Bayesian Information Criteria (BIC), Final Prediction Error (FPE), Low of Iterated Logarithm Criteria (LILC), and Corrected Akaike Information Criteria (AICc), which can be described respectively as

$$
\begin{aligned}
\mathrm{FPE}\left(n_\theta\right) &= N \ln\left[\sigma^2_{\text{erro}}\left(n_\theta\right)\right] + N \ln\left[\frac{N + n_\theta}{N - n_\theta}\right] \\
BIC\left(n_\theta\right) &= N \ln\left[\sigma^2_{\text{erro}}\left(n_\theta\right)\right] + n_\theta \ln N \\
AICc &= AIC + 2n_p * \frac{n_p + 1}{N - n_p - 1} \\
LILC &= 2n_\theta \ln(\ln(N)) + N \ln([\sigma^2_{\text{erro}}\left(n_\theta\right)])
\end{aligned}
\tag{29}
$$

To use any information criteria in SysIdentPy, set `order_selection=True` (as said before, the default value is already `True`). Besides `order_selection`, you can define how many regressors you want to evaluate before stopping the algorithm by using the `n_info_values` hyperparameter. The default value is 15, but the user should increase it based on how many regressors exists given the `ylag`, `xlag` and the degree of the basis function.

> Using information Criteria can take a long time depending on how many regressors you are evaluating and the number of samples. To calculate the criteria, the ERR algorithm is executed `n` times where `n` is the number defined in `n_info_values`. Make sure to understand how it works to define whether you have to use it or not.

Running the same example, but now using the BIC information criteria to select the order of the model, we have

```
model = FROLS(
    order_selection=True,
    n_info_values=15,
    ylag=2,
    xlag=2,
    info_criteria="bic",
    estimator=LeastSquares(unbiased=False),
    basis_function=basis_function
)
model.fit(X=x_train, y=y_train)
r = pd.DataFrame(
    results(
        model.final_model,
        model.theta,
        model.err,
        model.n_terms,
        err_precision=8,
        dtype="sci",
    ),
    columns=["Regressors", "Parameters", "ERR"],
)

print(r)
```

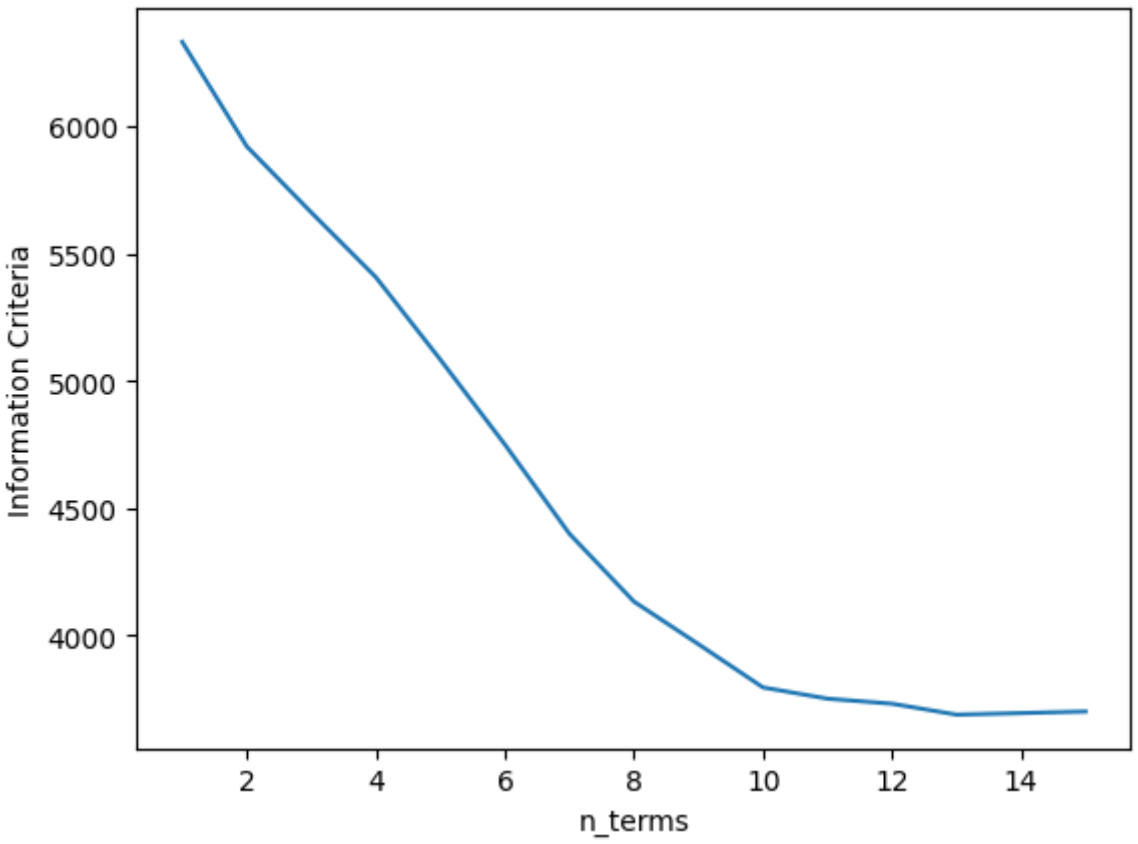
Regressors	Parameters	ERR
y(k-1)	1.3666E+00	9.86000384E-01
x1(k-1)^2	1.0500E+02	7.94805130E-03
y(k-2)^2	-5.8577E-05	2.50905908E-03
x1(k-1)y(k-1)	-1.2427E-01	1.43301039E-03
y(k-2)	-5.1414E-01	1.02781443E-03
x1(k-1)y(k-2)	5.3001E-02	5.35200312E-04
x1(k-2)	3.1144E+02	2.79648078E-04
x1(k-2)y(k-1)	-4.8013E-02	1.12211942E-04
x1(k-2)x1(k-1)	-8.0561E+00	4.54743448E-05
x1(k-2)y(k-2)	4.1381E-03	3.25346101E-05
1	-5.6653E+01	7.54107553E-06
y(k-2)y(k-1)	1.5679E-04	3.52002717E-06
y(k-1)^2	-9.0164E-05	6.17373260E-06

> Table 2

In this case, instead of 8 regressors, the final model have 13 terms.

Currently, the number of regressors is determined by identifying the index of the last value where the difference between the current and previous value is less than 0. To inspect these values, you can use the following approach:

```
xaxis = np.arange(1, model.n_info_values + 1)
plt.plot(xaxis, model.info_values)
plt.xlabel("n_terms")
plt.ylabel("Information Criteria")
```



> Figure 4. The plot shows the Information Criterion values (BIC) as a function of the number of terms included in the model. The model selection process, using the BIC criterion, iteratively adds regressors until the BIC reaches a minimum, indicating the optimal balance between model complexity and fit. The point where the BIC value stops decreasing marks the optimal number of terms, resulting in a final model with 13 terms.

The model prediction in this case is shown in Figure 5

```
yhat = model.predict(X=x_valid, y=y_valid)
# plot only the first 100 samples (n=100)
plot_results(y=y_valid, yhat=yhat, n=100)
```

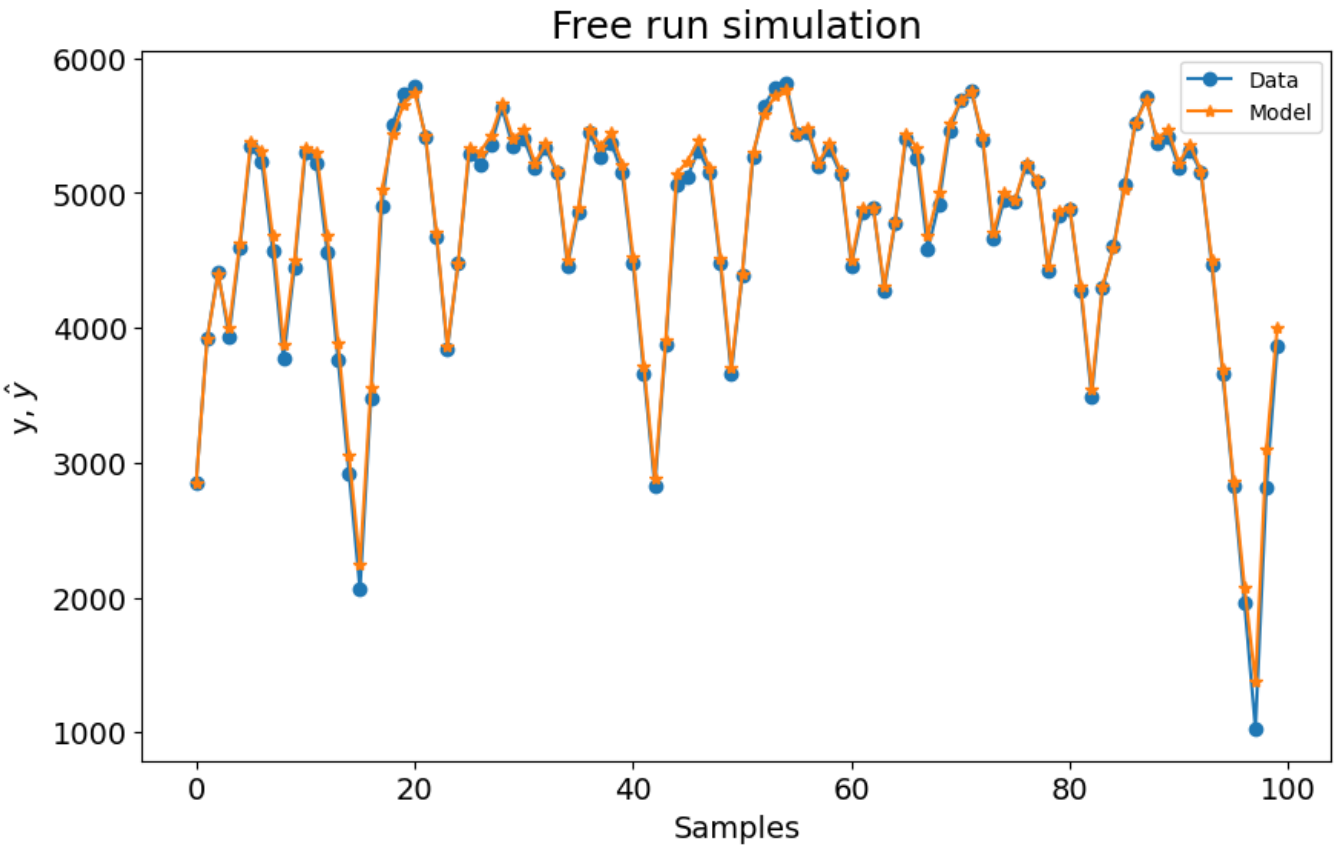


Figure 5. Free run simulation (or infinity-steps ahead prediction) of the fitted model using BIC.

Overview of the Information Criteria Methods

In this section, simulated data are used to provide users with a clearer understanding of the information criteria available in SysIdentPy.

> Here, we're working with a known model structure, which allows us to focus on how different information criteria perform. When dealing with real data, the correct number of terms in the model is unknown, making these methods invaluable for guiding model selection.

> If you review the metrics below, you'll notice excellent performance across all models. However, it's crucial to remember that System Identification is about finding the optimal model structure. Model Structure Selection is at the heart of NARMAX methods!

The data is generated by simulating the following model:

$$y_k = 0.2y_{k-1} + 0.1y_{k-1}x_{k-1} + 0.9x_{k-1} + e_k \tag{30}$$

If `colored_noise` is set to `True`, the noise term is defined as:

$$e_k = 0.8\nu_{k-1} + \nu_k \tag{31}$$

where x is a uniformly distributed random variable and ν is a Gaussian-distributed variable with $\mu = 0$ and $\sigma = 0.1$.

In the next example, we will generate data with 100 samples, using white noise, and select 70% of the data to train the model.

```
import numpy as np
import pandas as pd
from sysidentpy.model_structure_selection import FROLS
from sysidentpy.basis_function._basis_function import Polynomial
from sysidentpy.utils.generate_data import get_siso_data
from sysidentpy.utils.display_results import results


x_train, x_valid, y_train, y_valid = get_siso_data(
    n=100, colored_noise=False, sigma=0.1, train_percentage=70
)
```

The idea is to show the impact of the information criteria to select the number of terms to compose the final model. You will se why it is an auxiliary tool and let the algorithm select the number of terms based on the minimum value is not always a good idea when dealing with data highly corrupted by noise (even white noise).

AIC

```
basis_function = Polynomial(degree=2)
model = FROLS(
    order_selection=True,
    ylag=2,
    xlag=2,
    info_criteria="aic",
    basis_function=basis_function,
)

model.fit(X=x_train, y=y_train)
yhat = model.predict(X=x_valid, y=y_valid)

r = pd.DataFrame(
    results(
        model.final_model,
        model.theta,
        model.err,
        model.n_terms,
        err_precision=8,
        dtype="sci",
    ),
    columns=["Regressors", "Parameters", "ERR"],
)
```

```
print(r)

xaxis = np.arange(1, model.n_info_values + 1)
plt.plot(xaxis, model.info_values)
plt.xlabel("n_terms")
plt.ylabel("Information Criteria")
```

The regressors, the free run simulation and the AIC values are detailed bellow.

Regressors	Parameters	ERR
x1(k-2)	9.4236E-01	9.26094341E-01
y(k-1)	2.4933E-01	3.35898283E-02
x1(k-1)y(k-1)	1.3001E-01	2.35736200E-03
x1(k-1)	8.4024E-02	4.11741791E-03
x1(k-1)^2	7.0807E-02	2.54231877E-03
x1(k-2)^2	-9.1138E-02	1.39658893E-03
y(k-1)^2	1.1698E-01	1.70257419E-03
x1(k-2)y(k-2)	8.3745E-02	1.11056684E-03
y(k-2)^2	-4.1946E-02	1.01686239E-03
x1(k-2)x1(k-1)	5.9034E-02	7.47435512E-04

> Table 3

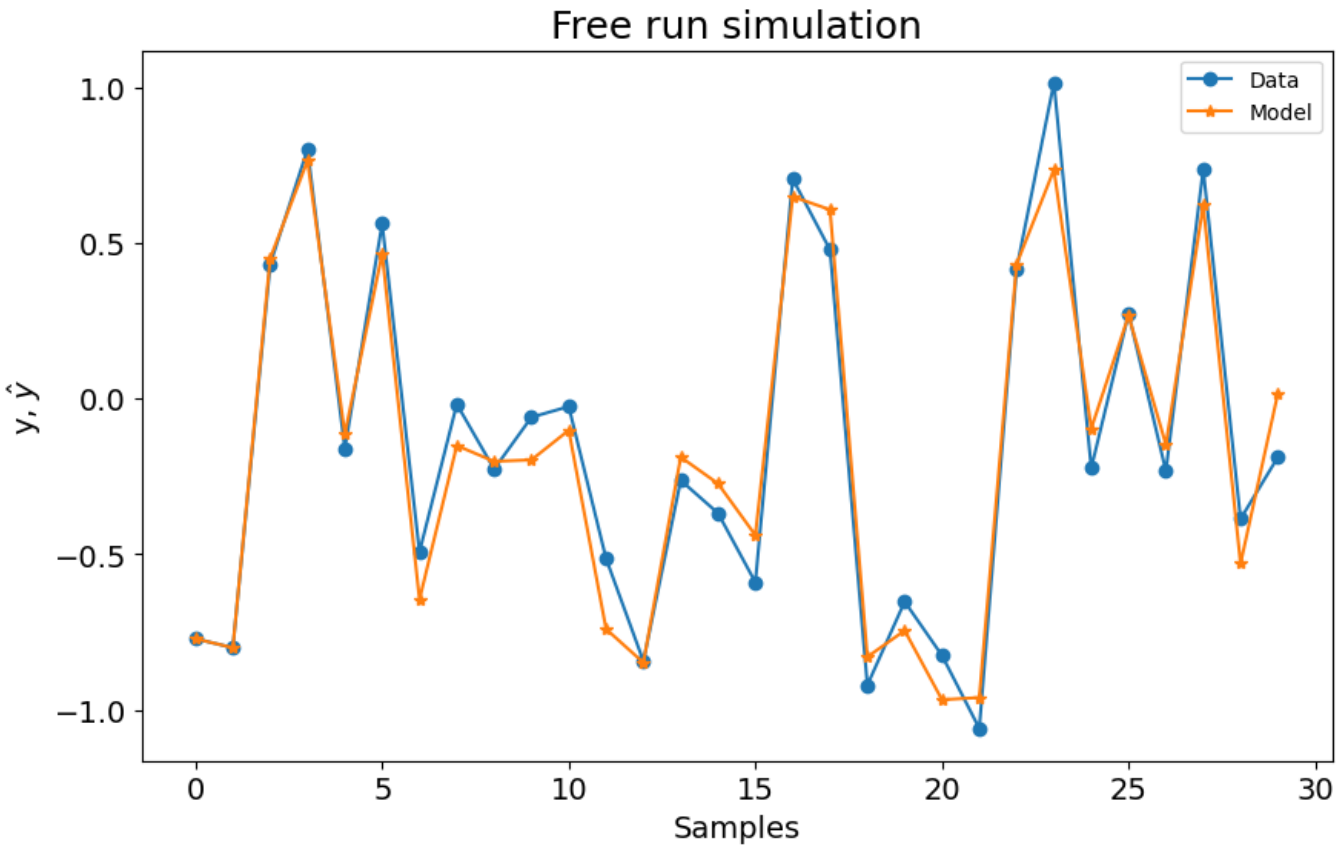
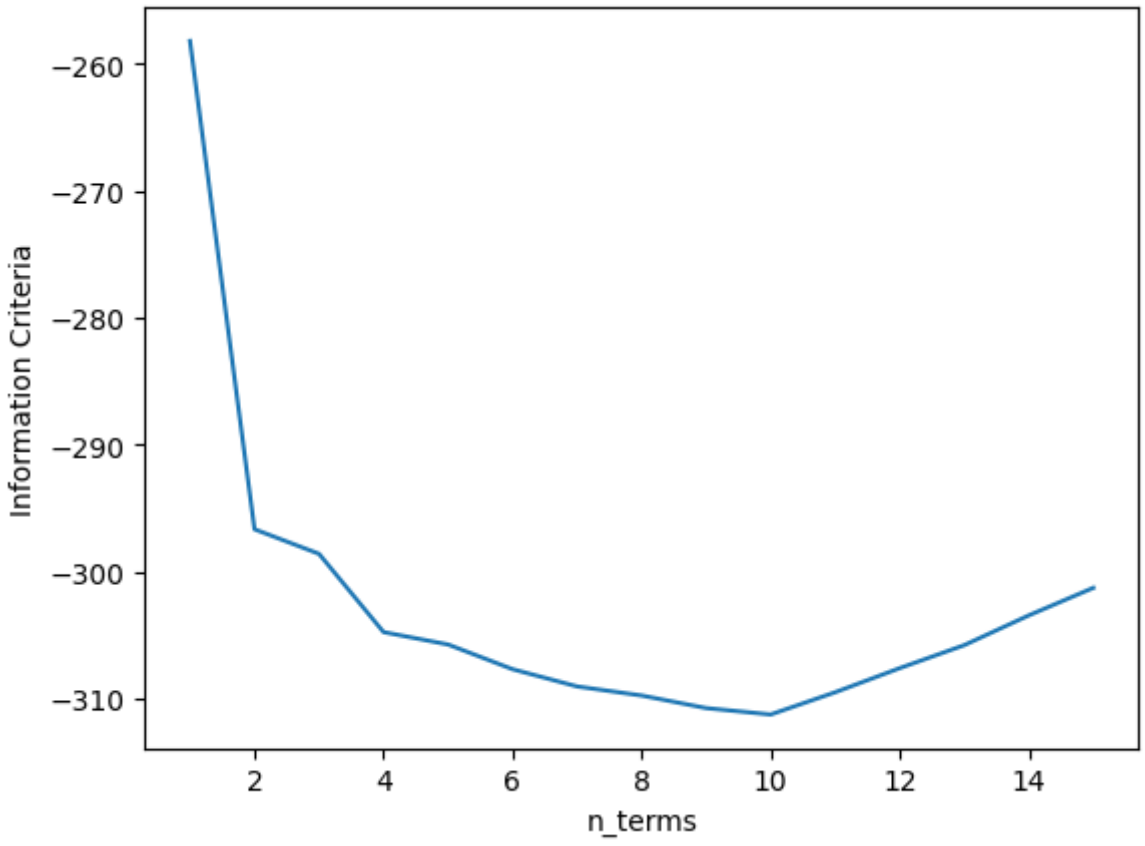


Figure 5. Free run simulation (or infinity-steps ahead prediction) of the fitted model using AIC.



> Figure 6. The plot shows the Information Criterion values (AIC) as a function of the number of terms included in the model. The model selection process, using the AIC criterion, iteratively adds regressors until the AIC reaches a minimum, indicating the optimal balance between model complexity and fit. The point where the AICc value stops decreasing marks the optimal number of terms, resulting in a final model with 10 terms.

For this case, we have a model with 10 terms. We know that the correct number is 3 because of the simulated system we are using as example.

AICc

The only change we have to do to use AICc instead of AIC is changing the information criteria hyperparameter: `information_criteria="aicc"`

```
basis_function = Polynomial(degree=2)
model = FROLS(
    order_selection=True,
    n_info_values=15,
    ylag=2,
    xlag=2,
    info_criteria="aicc",
    basis_function=basis_function,
)
model.fit(X=x_train, y=y_train)
yhat = model.predict(X=x_valid, y=y_valid)
r = pd.DataFrame(
    results(
        model.final_model,
        model.theta,
        model.err,
        model.n_terms,
        err_precision=8,
        dtype="sci",
    ),
    columns=["Regressors", "Parameters", "ERR"],
)
print(r)
plot_results(y=y_valid, yhat=yhat, n=1000)

xaxis = np.arange(1, model.n_info_values + 1)
plt.plot(xaxis, model.info_values)
plt.xlabel("n_terms")
plt.ylabel("Information Criteria")
```

Regressors	Parameters	ERR
x1(k-2)	9.2282E-01	9.26094341E-01

Regressors	Parameters	ERR
y(k-1)	2.4294E-01	3.35898283E-02
x1(k-1)y(k-1)	1.2753E-01	2.35736200E-03
x1(k-1)	6.9597E-02	4.11741791E-03
x1(k-1)^2	7.0578E-02	2.54231877E-03
x1(k-2)^2	-1.0523E-01	1.39658893E-03
y(k-1)^2	1.0949E-01	1.70257419E-03
x1(k-2)y(k-2)	7.1821E-02	1.11056684E-03
y(k-2)^2	-3.9756E-02	1.01686239E-03

Table 4

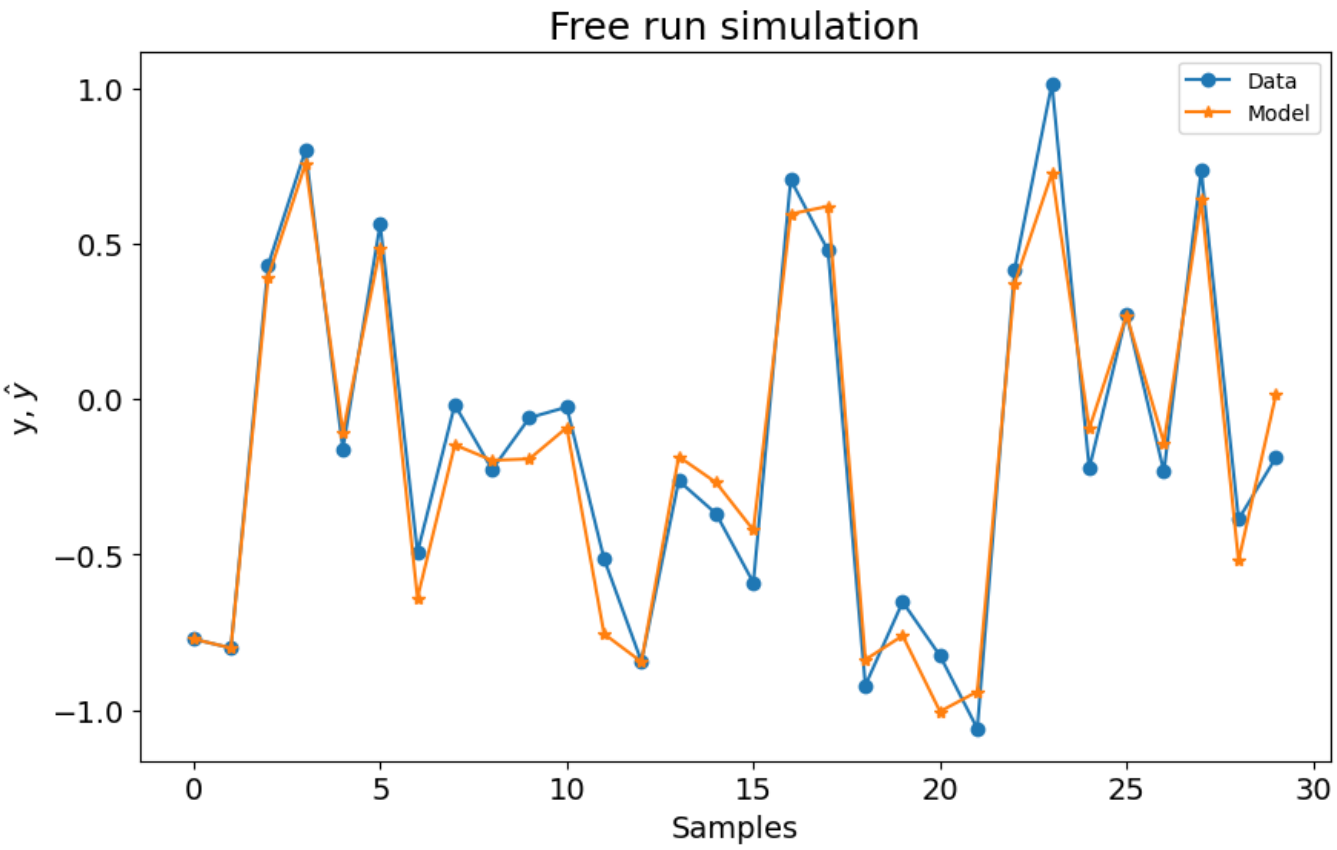
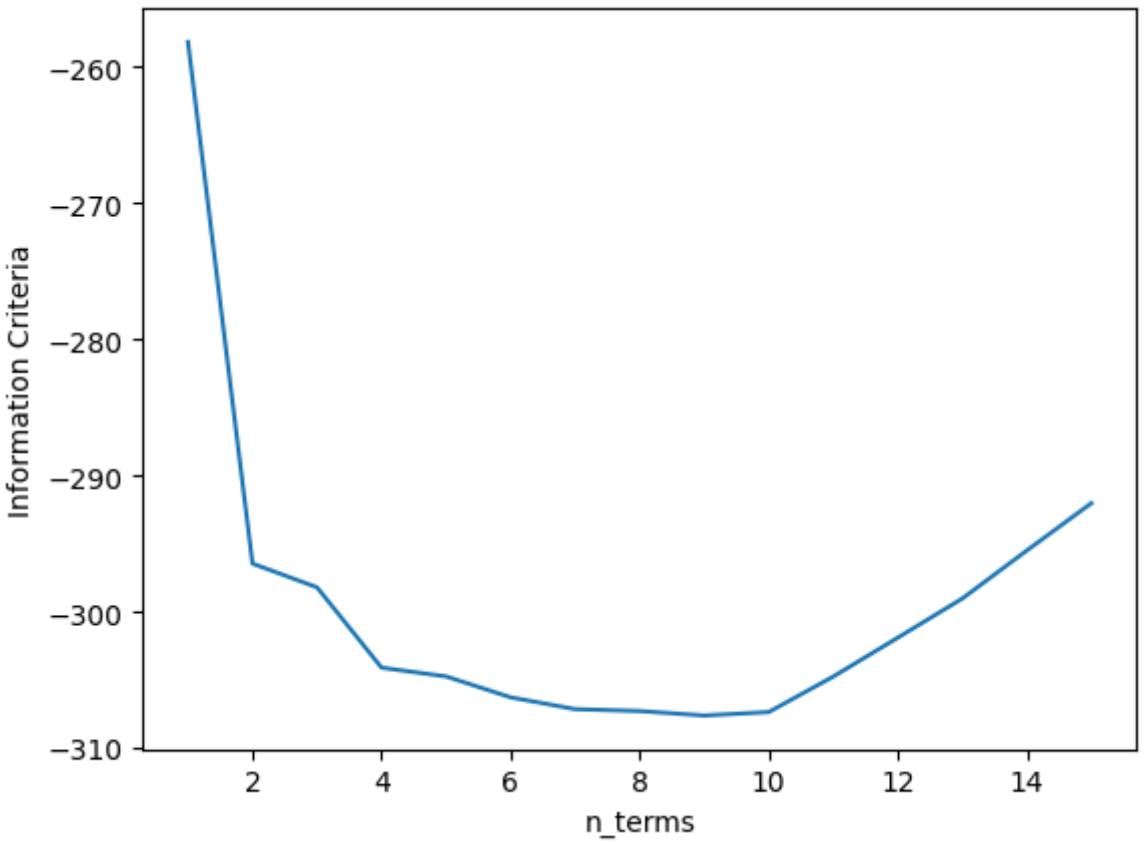


Figure 7. Free run simulation (or infinity-steps ahead prediction) of the fitted model using AICc.



> Figure 8. The plot shows the Information Criterion values (AICc) as a function of the number of terms included in the model. The model selection process, using the AIC criterion, iteratively adds regressors until the AICc reaches a minimum, indicating the optimal balance between model complexity and fit. The point where the AICc value stops decreasing marks the optimal number of terms, resulting in a final model with 9 terms.

This time we have a model with 9 regressors.

BIC

```
basis_function = Polynomial(degree=2)
model = FROLS(
    order_selection=True,
    n_info_values=15,
    ylag=2,
    xlag=2,
    info_criteria="bic",
    basis_function=basis_function,
)
model.fit(X=x_train, y=y_train)
yhat = model.predict(X=x_valid, y=y_valid)
r = pd.DataFrame(
    results(
        model.final_model,
```

```
        model.theta,
        model.err,
        model.n_terms,
        err_precision=8,
        dtype="sci",
    ),
    columns=["Regressors", "Parameters", "ERR"],
)
print(r)
plot_results(y=y_valid, yhat=yhat, n=1000)

xaxis = np.arange(1, model.n_info_values + 1)
plt.plot(xaxis, model.info_values)
plt.xlabel("n_terms")
plt.ylabel("Information Criteria")
```

Regressors	Parameters	ERR
x1(k-2)	9.1726E-01	9.26094341E-01
y(k-1)	1.8670E-01	3.35898283E-02

Table 5

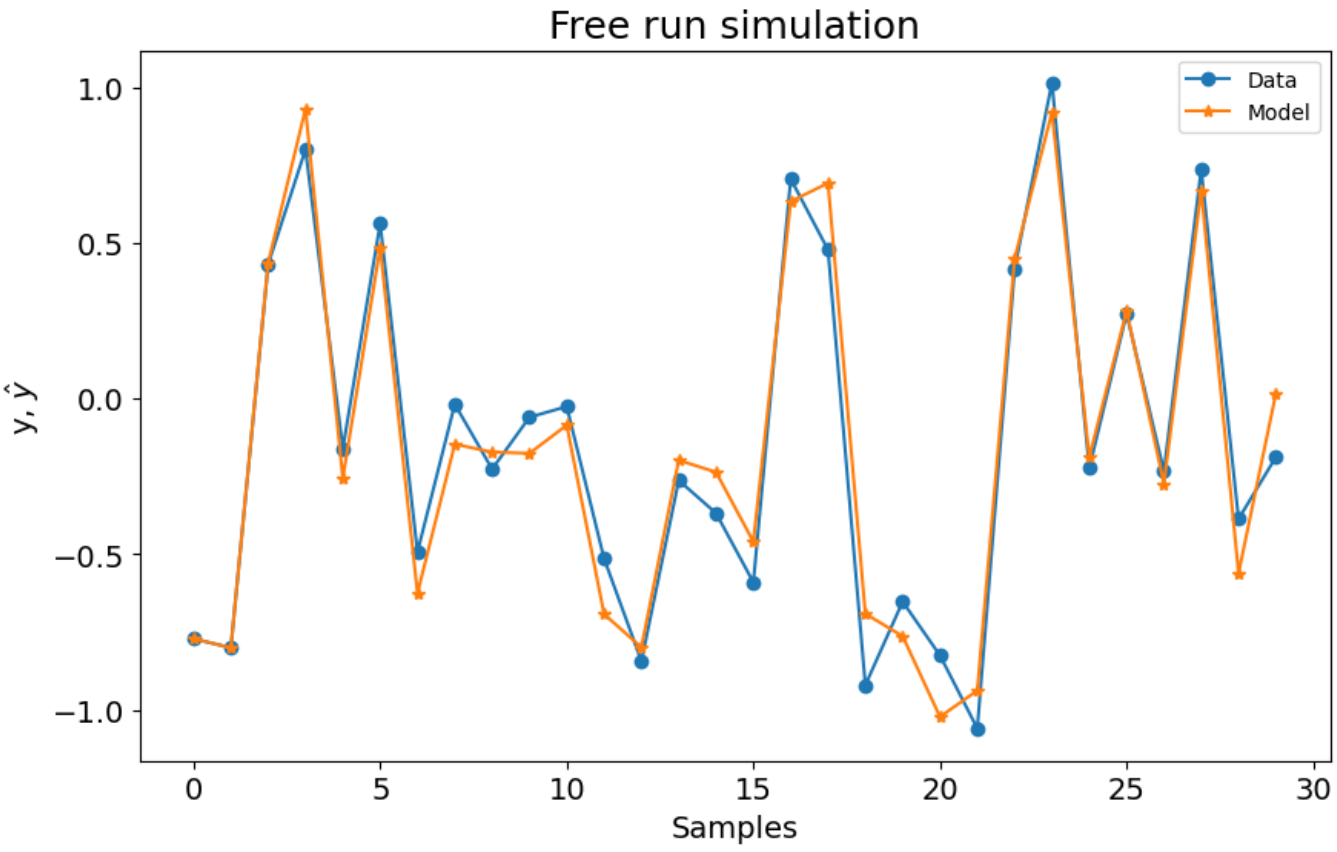
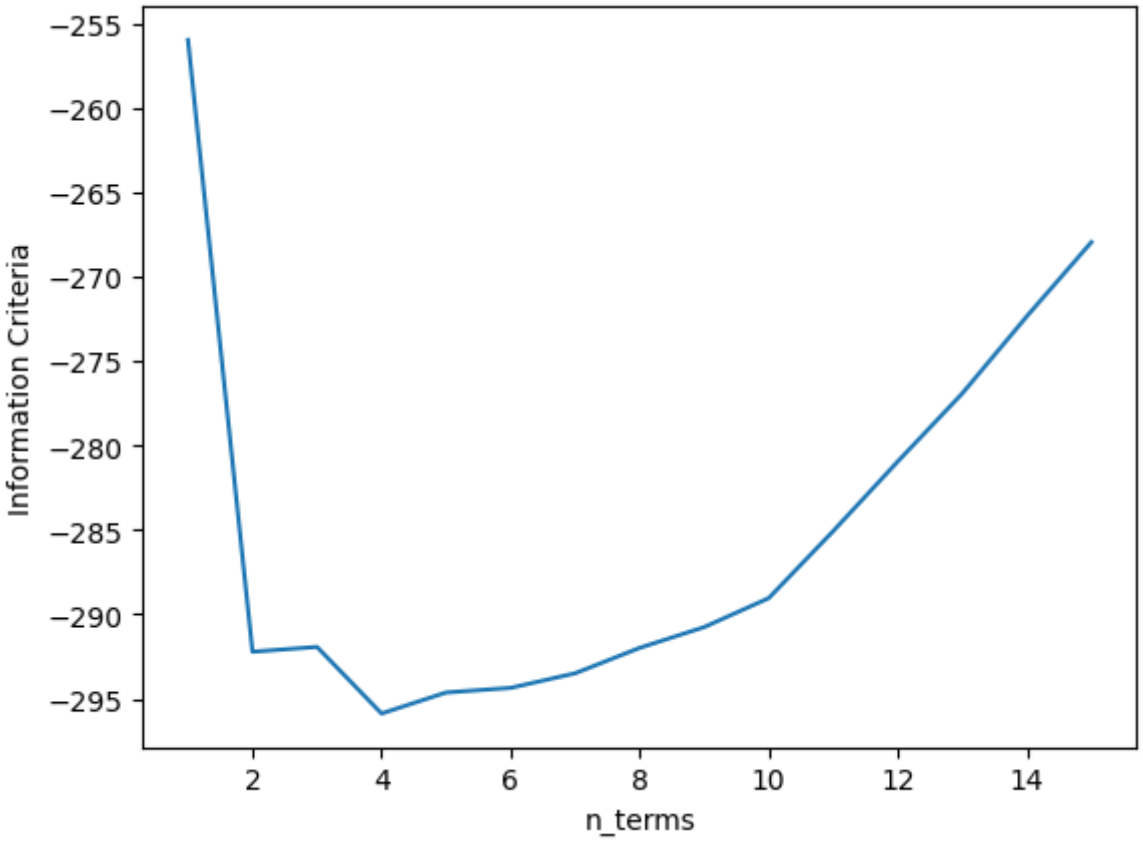


Figure 9. Free run simulation (or infinity-steps ahead prediction) of the fitted model using BIC.



> Figure 10. The plot shows the Information Criterion values (BIC) as a function of the number of terms included in the model. The model selection process, using the BIC criterion, iteratively adds regressors until the BIC reaches a minimum, indicating the optimal balance between model complexity and fit. The point where the BIC value stops decreasing marks the optimal number of terms, resulting in a final model with 2 terms.

BIC returned a model with only 2 regressors!

LILC

```
basis_function = Polynomial(degree=2)
model = FROLS(
    order_selection=True,
    n_info_values=15,
    ylag=2,
    xlag=2,
    info_criteria="lilc",
    basis_function=basis_function,
)
model.fit(X=x_train, y=y_train)
yhat = model.predict(X=x_valid, y=y_valid)
r = pd.DataFrame(
    results(
        model.final_model,
```

```
        model.theta,
        model.err,
        model.n_terms,
        err_precision=8,
        dtype="sci",
    ),
    columns=["Regressors", "Parameters", "ERR"],
)
print(r)
plot_results(y=y_valid, yhat=yhat, n=1000)

xaxis = np.arange(1, model.n_info_values + 1)
plt.plot(xaxis, model.info_values)
plt.xlabel("n_terms")
plt.ylabel("Information Criteria")
```

Regressors	Parameters	ERR
x1(k-2)	9.1160E-01	9.26094341E-01
y(k-1)	2.3178E-01	3.35898283E-02
x1(k-1)y(k-1)	1.2080E-01	2.35736200E-03
x1(k-1)	6.3113E-02	4.11741791E-03
x1(k-1)^2	5.4088E-02	2.54231877E-03
x1(k-2)^2	-9.0683E-02	1.39658893E-03
y(k-1)^2	8.2157E-02	1.70257419E-03

> Table 6

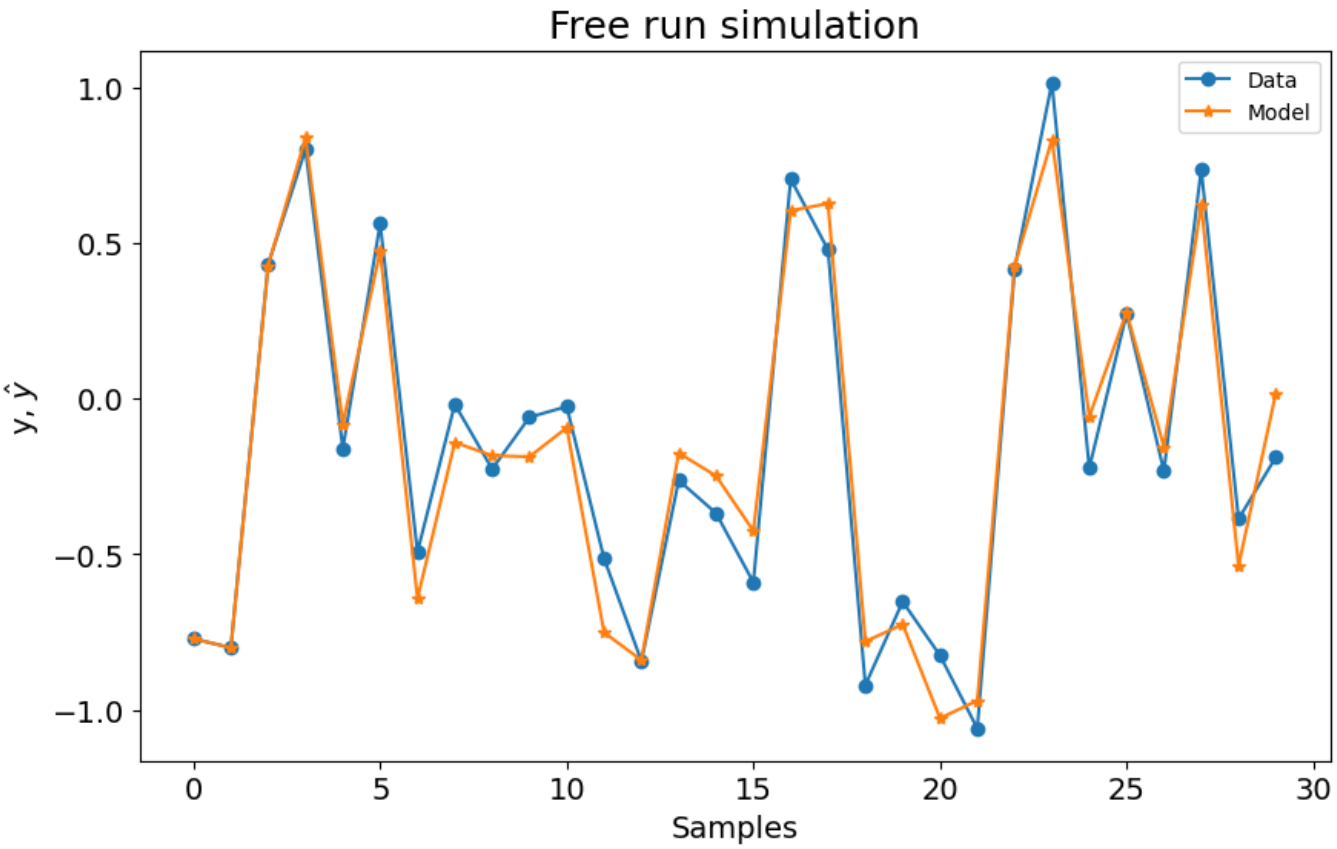
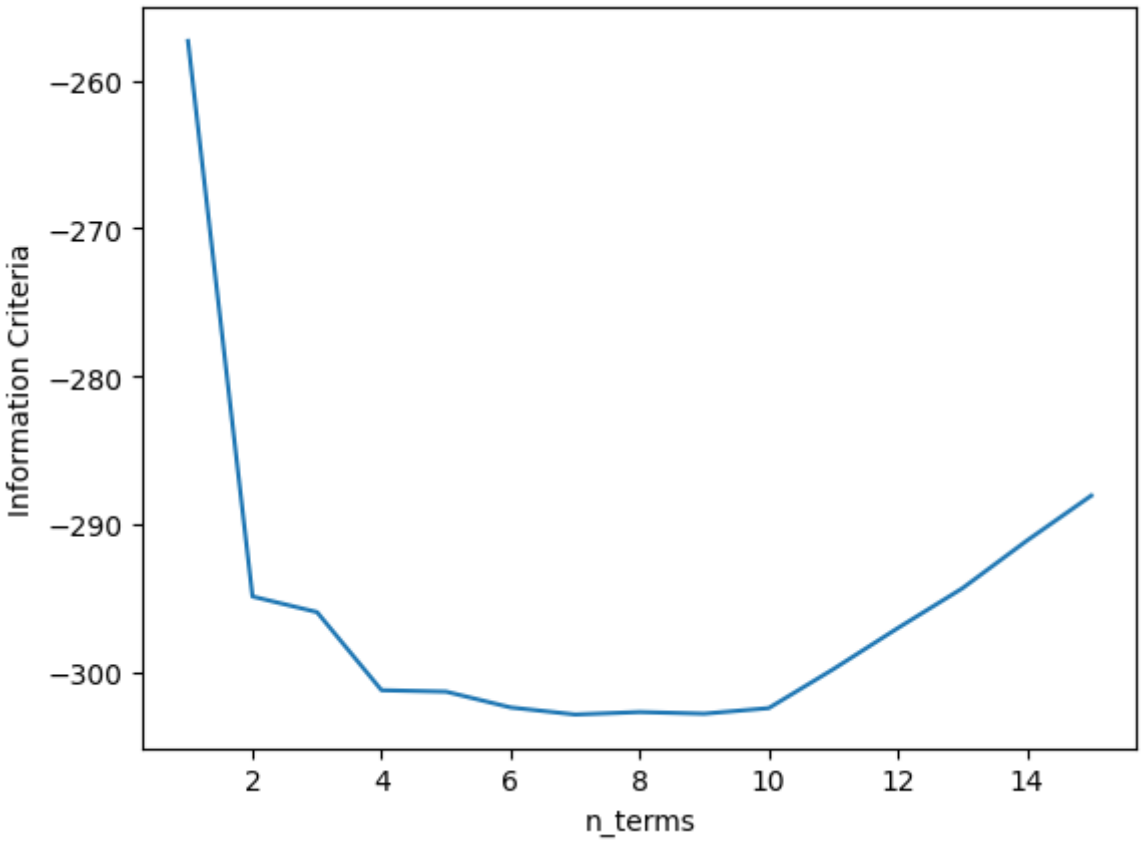


Figure 11. Free run simulation (or infinity-steps ahead prediction) of the fitted model using LILC.



> Figure 12. The plot shows the Information Criterion values (LILC) as a function of the number of terms included in the model. The model selection process, using the LILC criterion, iteratively adds regressors until the LILC reaches a minimum, indicating the optimal balance between model complexity and fit. The point where the LILC value stops decreasing marks the optimal number of terms, resulting in a final model with 7 terms.

LILC returned a model with 7 regressors.

FPE

```
basis_function = Polynomial(degree=2)
model = FROLS(
    order_selection=True,
    n_info_values=15,
    ylag=2,
    xlag=2,
    info_criteria="fpe",
    basis_function=basis_function,
)
model.fit(X=x_train, y=y_train)
yhat = model.predict(X=x_valid, y=y_valid)
r = pd.DataFrame(
    results(
        model.final_model,
        model.theta,
        model.err,
        model.n_terms,
        err_precision=8,
        dtype="sci",
    ),
    columns=["Regressors", "Parameters", "ERR"],
)
print(r)
plot_results(y=y_valid, yhat=yhat, n=1000)

xaxis = np.arange(1, model.n_info_values + 1)
plt.plot(xaxis, model.info_values)
plt.xlabel("n_terms")
plt.ylabel("Information Criteria")
```

Regressors	Parameters	ERR
x1(k-2)	9.4236E-01	9.26094341E-01
y(k-1)	2.4933E-01	3.35898283E-02
x1(k-1)y(k-1)	1.3001E-01	2.35736200E-03

Regressors	Parameters	ERR
x1(k-1)	8.4024E-02	4.11741791E-03
x1(k-1)^2	7.0807E-02	2.54231877E-03
x1(k-2)^2	-9.1138E-02	1.39658893E-03
y(k-1)^2	1.1698E-01	1.70257419E-03
x1(k-2)y(k-2)	8.3745E-02	1.11056684E-03
y(k-2)^2	-4.1946E-02	1.01686239E-03
x1(k-2)x1(k-1)	5.9034E-02	7.47435512E-04

Table 7

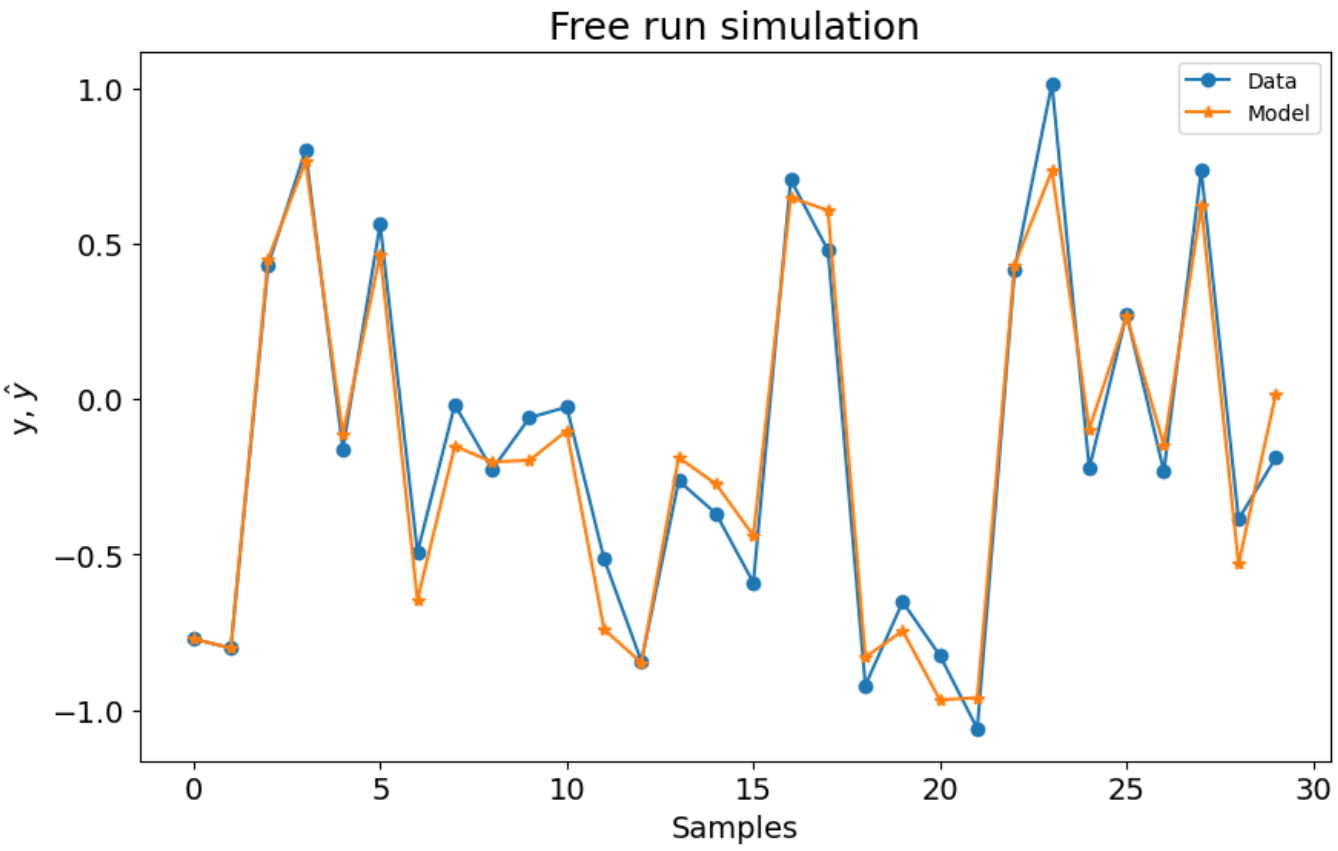
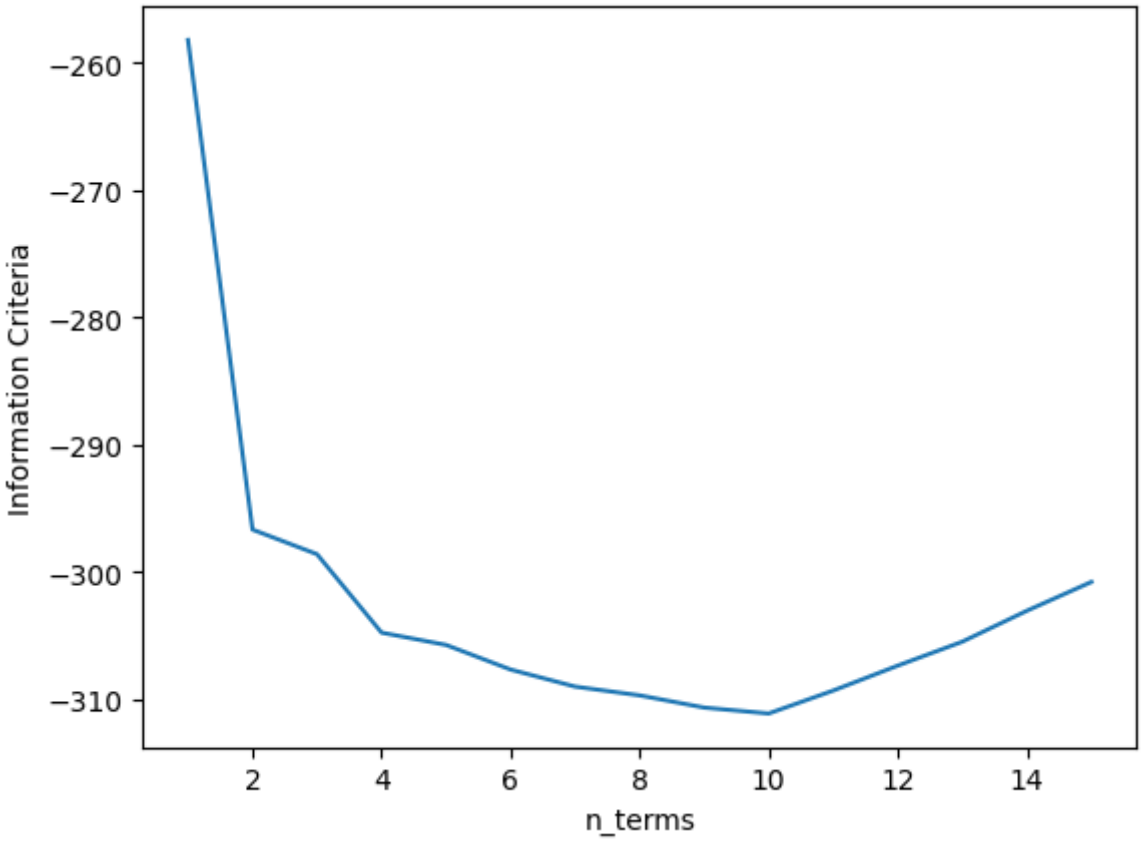


Figure 13. Free run simulation (or infinity-steps ahead prediction) of the fitted model using FPE.



> Figure 14. The plot shows the Information Criterion values (FPE) as a function of the number of terms included in the model. The model selection process, using the FPE criterion, iteratively adds regressors until the FPE reaches a minimum, indicating the optimal balance between model complexity and fit. The point where the FPE value stops decreasing marks the optimal number of terms, resulting in a final model with 10 terms.

FPE returned a model with 10 regressors.

Meta Model Structure Selection (MetaMSS)

> This section largely reflects content from a paper I published on ArXiv titled *"Meta-Model Structure Selection: Building Polynomial NARX Models for Regression and Classification."* This paper was initially written for journal publication based on the results of my master's thesis. However, as I transitioned into a Data Scientist role and considering the lengthy journal submission process and academic delays, I decided not to pursue journal publication at this time. Thus, the paper remains available only on ArXiv.

> The work extends a previous paper I presented at a Brazilian conference (in Portuguese), where part of the results were initially shared.

This section introduces a meta-heuristic approach for selecting the structure of polynomial NARX models in regression tasks. The proposed method considers both the complexity of the model and the contribution of each term to construct parsimonious models through a novel cost function formulation.

The robustness of this new algorithm is evaluated using various simulated and experimental systems with different nonlinear characteristics. The results demonstrate that the algorithm effectively identifies the correct model when the true structure is known and produces parsimonious models for experimental data, even in cases where traditional and contemporary methods often fail. The new approach is compared against classical methods such as FROLS and recent randomized techniques.

We mentioned that selecting the appropriate model terms is crucial for accurately capturing the dynamics of the original system. Challenges such as overparameterization and numerical ill-conditioning often arise due to the limitations of existing identification algorithms in selecting the right terms for the final model. Check [Aguirre, L. A. and Billings, S. A.](Dynamical effects of overparametrization in nonlinear models), [Piroddi, L. and Spinelli, W.](An identification algorithm for polynomial NARX models based on simulation error minimization). We also mentioned that one of the most traditionally algorithms for structure selection of polynomial NARMAX is the ERR algorithm. Numerous variants of FROLS algorithm has been developed to improve the model selection performance such as [Billings, S. A. and Chen, S. and Korenberg, M. J.](Identification of MIMO non-linear systems using a forward-regression orthogonal estimator), [Farina, M. and Piroddi, L.] (Simulation Error Minimization–Based Identification of Polynomial Input–Output Recursive Models), [Guo, Y. and Guo, L. Z. and Billings, S. A. and Wei, H.](A New Iterative Orthogonal Forward Regression Algorithm), [Mao, K. Z. and Billings, S. A.](VARIABLE SELECTION IN NON-LINEAR SYSTEMS MODELLING). The drawbacks of the FROLS have been extensively reviewed in the literature, e.g., in Billings, S. A. and Aguirre, L. A., Palumbo, P. and Piroddi, L., Falsone, A. and Piroddi, L. and Prandini, M.. Most of these weak points are related to (i) the Prediction Error Minimization (PEM) framework; (ii) the inadequacy of the ERR index in measuring the absolute importance of regressors; (iii) the use of information criteria such as AIC, FPE and the BIC, to select the model order. Regarding the information criteria, although these techniques work well for linear models, in a nonlinear context no simple relation between model size and accuracy can be established [Falsone, A. and Piroddi, L. and Prandini, M.](A randomized algorithm for nonlinear model structure selection) , [Chen, S. and Hong, X. and Harris, C. J.](Sparse kernel regression modeling using combined locally regularized orthogonal least squares and D-optimality experimental design).

Due to the limitations of Ordinary Least Squares (OLS)-based algorithms, recent research has presented solutions that diverged from the classical FROLS approach. New methods have reformulated the Model Structure Selection (MSS) process within a probabilistic framework and employed random sampling techniques [Falsone, A. and Piroddi, L. and Prandini, M.](A randomized algorithm for nonlinear model structure selection), [Tempo, R. and Calafiore, G. and Dabbene, F.] (Randomized Algorithms for Analysis and Control of Uncertain Systems: With Applications), [Baldacchino, T. and Anderson, S. R. and Kadirkamanathan, V.](Computational system identification for Bayesian NARMAX modelling), [Rodriguez-Vazquez, K. and Fonseca, C. M. and Fleming, P. J.] (Identifying the structure of nonlinear dynamic systems using multiobjective genetic programming), Severino, A. G. V. and Araujo, F. M. U. de. Despite their advancements, these meta-heuristic and probabilistic approaches exhibit certain shortcomings. In particular, these methods often rely on information criteria such as AIC, FPE, and BIC to define the cost function for optimization, which frequently leads to over-parameterized models.

Consider $\mathcal{F}$ as a class of bounded functions $\phi : \mathbf{R} \mapsto \mathbf{R}$. If the properties of $\phi(x)$ satisfy

$$\begin{aligned} &\lim_{x \to \infty} \phi(x) = \alpha \\ &\lim_{x \to -\infty} \phi(x) = \beta \quad \text{with } \alpha > \beta, \end{aligned} \tag{32}$$

the function is called sigmoidal.

In this particular case and following definition Equation 32 with $alpha = 0$ and $\beta = 1$, we write a "S" shaped curve as

$$\varsigma(x) = \frac{1}{1 + e^{-a(x-c)}}. \tag{33}$$

In that case, we can specify a, the rate of change. If a is close to zero, the sigmoid function will be gradual. If a is large, the sigmoid function will have an abrupt or sharp transition. If a is negative, the sigmoid will go from 1 to zero. The parameter c corresponds to the x value where $y = 0.5$.

The Sigmoid Linear Unit Function (SiLU) is defined by the sigmoid function multiplied by its input

$$\text{silu}(x) = x\varsigma(x), \tag{34}$$

which can be viewed as an steeper sigmoid function with overshoot.

Meta-heuristics

Over the past two decades, nature-inspired optimization algorithms have gained prominence due to their flexibility, simplicity, versatility, and ability to avoid local optima in real-world applications.

Meta-heuristic algorithms are characterized by two fundamental features: exploitation and exploration [Blum, C. and Roli, A.](Metaheuristics in combinatorial optimization: Overview and conceptual comparison). **Exploitation** focuses on utilizing local information to refine the search around the current best solution, improving the quality of nearby solutions. Conversely, **exploration** aims to search a broader area of the solution space to discover potentially superior solutions and prevent the algorithm from getting trapped in local optima.

Despite the lack of a universal consensus on the definitions of exploration and exploitation in evolutionary computing, as highlighted by Eiben, Agoston E and Schippers, Cornelis A, it is generally agreed that these concepts function as opposing forces that are challenging to balance. To address this challenge, hybrid metaheuristics combine multiple algorithms to leverage both exploitation and exploration, resulting in more robust optimization methods.

The Binary hybrid Particle Swarm Optimization and Gravitational Search Algorithm (BPSOGSA) algorithm

Achieving a balance between exploration and exploitation is a significant challenge in most meta-heuristic algorithms. For this method, we enhance performance and flexibility in the search process by employing a hybrid approach that combines Binary Particle Swarm Optimization (BPSO) with Gravitational Search Algorithm (GSA), as proposed by Mirjalili, S. and Hashim, S. Z. M.. This hybrid method incorporates a low-level co-evolutionary heterogeneous technique originally introduced by Talbi, E. G..

The BPSOGSA approach leverages the strengths of both algorithms: the Particle Swarm Optimization (PSO) component is known to be good in exploring the entire search space to identify the global optimum, while the Gravitational Search Algorithm (GSA) component effectively refines the search by focusing on local solutions within a binary space. This combination aims to provide a more comprehensive and effective optimization strategy, ensuring a better balance between exploration and exploitation.

Standard Particle Swarm Optimization (PSO)

In Particle Swarm Optimization (PSO) Kennedy, J. and Eberhart, R. C., Kennedy, J., each particle represents a candidate solution and is characterized by two components: its position in the search space, denoted as $\vec{x}_{\,np,d} \in \mathbb{R}^{np \times d}$, and its velocity, $\vec{v}_{\,np,d} \in \mathbb{R}^{np \times d}$. Here, $np = 1, 2, \ldots, n_a$ where n_a is the size of the swarm, and d is the dimensionality of the problem. The initial population is represented as follows:

$$\vec{x}_{\,np,d} = \begin{bmatrix} x_{1,1} & x_{1,2} & \cdots & x_{1,d} \\ x_{2,1} & x_{2,2} & \cdots & x_{2,d} \\ \vdots & \vdots & \ddots & \vdots \\ x_{n_a,1} & x_{n_a,2} & \cdots & x_{n_a,d} \end{bmatrix} \tag{35}$$

At each iteration t, the position and velocity of a particle are updated using the following equations:

$$v_{np,d}^{t+1} = \zeta v_{np,d}^{t} + c_1 \kappa_1 (pbest_{np}^{t} - x_{np,d}^{t}) + c_2 \kappa_2 (gbest_{np}^{t} - x_{np,d}^{t}), \tag{36}$$

where $\kappa_j \in \mathbb{R}$ for $j = [1, 2]$ are continuous random variables in the interval $[0, 1]$, $\zeta \in \mathbb{R}$ is the inertia factor that controls the influence of the previous velocity on the current one and represents a trade-off between exploration and exploitation, c_1 is the cognitive factor associated with the personal best position $pbest$, and c_2 is the social factor associated with the global best position $gbest$. The velocity $\vec{v}_{\,np,d}$ is typically constrained within the range $[v_{min}, v_{max}]$ to prevent particles from moving outside the search space. The updated position is then computed as:

$$x_{np,d}^{t+1} = x_{np,d}^{t} + v_{np,d}^{t+1}. \tag{37}$$

Standard Gravitational Search Algorithm (GSA)

In the Gravitational Search Algorithm (GSA) [Rashedi, Esmat and Nezamabadi-Pour, Hossein and Saryazdi, Saeid](GSA: A Gravitational Search Algorithm), agents are represented by masses, where the magnitude of each mass is proportional to the fitness value of the agent. These masses interact

through gravitational forces, attracting each other towards locations closer to the global optimum. Heavier masses (agents with better fitness) move more slowly, while lighter masses (agents with poorer fitness) move more rapidly. Each mass in GSA has four properties: position, inertial mass, active gravitational mass, and passive gravitational mass. The position of a mass represents a candidate solution to the problem, and its gravitational and inertial masses are derived from the fitness function.

Consider a population of agents as described by the following equations. At a specific time t, the velocity and position of each agent are updated as follows:

$$\begin{aligned} v_{i,d}^{t+1} &= \kappa_i \times v_{i,d}^{t} + a_{i,d}^{t}, \\ x_{i,d}^{t+1} &= x_{i,d}^{t} + v_{i,d}^{t+1}. \end{aligned} \tag{38}$$

Here, κ_i introduces stochastic characteristics to the search process. The acceleration $a_{i,d}^{t}$ is computed according to the law of motion Rashedi, Esmat and Nezamabadi-Pour, Hossein and Saryazdi, Saeid:

$$a_{i,d}^{t} = \frac{F_{i,d}^{t}}{M_{ii}^{t}}, \tag{39}$$

where M_{ii}^{t} is the inertial mass of agent i and $F_{i,d}^{t}$ represents the gravitational force acting on agent i in the d-dimensional space. The detailed process for calculating and updating $F_{i,d}$ and M_{ii} can be found in Rashedi, Esmat and Nezamabadi-Pour, Hossein and Saryazdi, Saeid.

The Binary Hybrid Optimization Algorithm

The combination of algorithms follows the approach described in [Mirjalili, S. and Hashim, S. Z. M.](A new hybrid PSOGSA algorithm for function optimization):

$$v_i^{t+1} = \zeta \times v_i^{t} + \mathrm{c}_1' \times \kappa \times a_i^{t} + \mathrm{c}_2' \times \kappa \times (gbest - x_i^{t}), \tag{40}$$

where $\mathrm{c}_j' \in \mathbb{R}$ are acceleration coefficients. This formulation accelerates the exploitation phase by incorporating the best mass location found so far. However, this method may negatively impact the exploration phase. To address this issue, [Mirjalili, S. and Mirjalili, S. M. and Lewis, A.](Grey Wolf Optimizer) proposed adaptive values for c_j', as described in [Mirjalili, S. and Wang, Gai-Ge and Coelho, L. dos S.](Binary optimization using hybrid particle swarm optimization and gravitational search algorithm):

$$\begin{aligned} \mathrm{c}_1' &= -2 \times \frac{t^3}{\max(t)^3} + 2, \\ \mathrm{c}_2' &= 2 \times \frac{t^3}{\max(t)^3} + 2. \end{aligned} \tag{41}$$

In each iteration, the positions of particles are updated according to the following rules, with continuous space mapped to discrete solutions using a transfer function ([Mirjalili, S. and Lewis, A.](S-shaped versus V-shaped transfer functions for binary Particle Swarm Optimization)):

$$S(v_{ik}) = \left| \frac{2}{\pi} \arctan\left(\frac{\pi}{2} v_{ik}\right) \right|. \tag{42}$$

With a uniformly distributed random number $\kappa \in (0, 1)$, the positions of the agents in the binary space are updated as follows:

$$x_{np,d}^{t+1} = \begin{cases} (x_{np,d}^{t})^{-1}, & \text{if } \kappa < S(v_{ik}^{t+1}), \\ x_{np,d}^{t}, & \text{if } \kappa \geq S(v_{ik}^{t+1}). \end{cases} \tag{43}$$

Meta-Model Structure Selection (MetaMSS): Building NARX for Regression

In this section, we explore the meta-heuristic approach for selecting the structure of NARX models using BPSOGSA proposed in my master's thesis. This method searches for the optimal model structure within a decision space defined by a predefined dictionary of regressors. The objective function for this optimization problem is based on the root mean squared error (RMSE) of the free-run simulation output, augmented by a penalty factor that accounts for the model's complexity and the contribution of each regressor to the final model.

Encoding Scheme

The process of using BPSOGSA for model structure selection involves defining the dimensions of the test function. Specifically, n_y, n_x, and ℓ are set to cover all possible regressors, and a general matrix of regressors, Ψ, is constructed. The number of columns in Ψ is denoted as noV, and the number of agents, N, is specified. A binary $noV \times N$ matrix, referred to as $\mathcal{X}$, is then randomly generated to represent the position of each agent in the search space. Each column of $\mathcal{X}$ represents a potential solution, which is essentially a candidate model structure to be evaluated in each iteration. In this matrix, a value of 1 indicates that the corresponding column of Ψ is included in the reduced matrix of regressors, while a value of 0 indicates exclusion.

For example, consider a case where all possible regressors are defined with $\ell = 1$ and $n_y = n_u = 2$. The matrix Ψ is:

$$[\text{constant} \quad y(k-1) \quad y(k-2) \quad u(k-1) \quad u(k-2)] \tag{44}$$

With 5 possible regressors, $noV = 5$. Assuming $N = 5$, $\mathcal{X}$ might be represented as:

$$\mathcal{X} = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 1 & 0 \end{bmatrix} \tag{45}$$

Each column of $\mathcal{X}$ is transposed to generate a candidate solution. For example, the first column of $\mathcal{X}$ yields:

$$\mathcal{X} = \begin{bmatrix} \text{constant} & y(k-1) & y(k-2) & u(k-1) & u(k-2) \\ 1 & 1 & 1 & 0 & 1 \end{bmatrix} \tag{46}$$

In this scenario, the first model to be evaluated is $\alpha y(k-1) + \beta u(k-2)$, where α and β are parameters estimated using some parameter estimation method. The process is repeated for each subsequent column of $\mathcal{X}$.

Formulation of the objective function

For each candidate model structure randomly defined, the linear-in-the-parameters system can be solved directly using the Least Squares algorithm or any other method available in SysIdentPy. The variance of the estimated parameters can be calculated as:

$$\hat{\sigma}^2 = \hat{\sigma}_e^2 V_{jj}, \tag{47}$$

where $\hat{\sigma}_e^2$ is the estimated noise variance, given by:

$$\hat{\sigma}_e^2 = \frac{1}{N-m} \sum_{k=1}^{N} (y_k - \psi_{k-1}^\top \hat{\Theta}), \tag{48}$$

and V_{jj} is the jth diagonal element of $(\Psi^\top \Psi)^{-1}$.

The estimated standard error of the jth regression coefficient $\hat{\Theta}_j$ is the positive square root of the diagonal elements of $\hat{\sigma}^2$:

$$\text{se}(\hat{\Theta}_j) = \sqrt{\hat{\sigma}_{jj}^2}. \tag{49}$$

To assess the statistical relevance of each regressor, a penalty test considers the standard error of the regression coefficients. In this case, the t-test is used to perform a hypothesis test on the coefficients, evaluating the significance of individual regressors in the multiple linear regression model. The hypothesis statements are:

$$\begin{aligned} H_0 &: \Theta_j = 0, \\ H_a &: \Theta_j \neq 0. \end{aligned} \tag{50}$$

The t-statistic is computed as:

$$T_0 = \frac{\hat{\Theta}_j}{\text{se}(\hat{\Theta}_j)}, \tag{51}$$

which measures how many standard deviations $\hat{\Theta}_j$ is from zero. More precisely, if:

$$-t_{\alpha/2, N-m} < T_0 < t_{\alpha/2, N-m}, \tag{52}$$

where $t_{\alpha/2, N-m}$ is the t value obtained considering α as the significance level and $N-m$ as the degrees of freedom, then if T_0 falls outside this acceptance region, the null hypothesis $H_0 : \Theta_j = 0$ is

rejected. This implies that Θ_j is significant at the α level. Otherwise, if T_0 lies within the acceptance region, Θ_j is not significantly different from zero, and the null hypothesis cannot be rejected.

Penalty value based on the Derivative of the Sigmoid Linear Unit function

We propose a penalty value based on the derivative of the sigmoid function, defined as:

$$\dot{\varsigma}(x(\varrho)) = \varsigma(x)[1 + (a(x - c))(1 - \varsigma(x))]. \tag{53}$$

In this formulation, the parameters are defined as follows: x has the dimension of noV; $c = noV/2$; and a is set as the ratio of the number of regressors in the current test model to c. This approach results in a distinct curve for each model, with the slope of the sigmoid curve becoming steeper as the number of regressors increases. The penalty value, ϱ, corresponds to the y value of the sigmoid curve for the given number of regressors in x. Since the derivative of the sigmoid function can return negative values, we normalize ς as:

$$\varrho = \varsigma - \min(\varsigma), \tag{54}$$

ensuring that $\varrho \in \mathbb{R}^+$.

However, two different models with the same number of regressors can yield significantly different results due to the varying importance of each regressor. To address this, we use the t-student test to assess the statistical relevance of each regressor and incorporate this information into the penalty function. Specifically, we calculate n_{Θ,H_0}, the number of regressors that are not significant for the model. The penalty value is then adjusted based on the model size:

$$\text{model\_size} = n_\Theta + n_{\Theta,H_0}. \tag{55}$$

The objective function, which integrates the relative root mean squared error of the model with ϱ, is defined as:

$$\mathcal{F} = \frac{\sqrt{\sum_{k=1}^{n}(y_k - \hat{y}_k)^2}}{\sqrt{\sum_{k=1}^{n}(y_k - \bar{y})^2}} \times \varrho. \tag{56}$$

This approach ensures that even if models have the same number of regressors, those with redundant regressors are penalized more heavily.

Case Studies: Simulation Results

In this section, six simulation examples are considered to illustrate the effectiveness of the MetaMSS algorithm. An analysis of the algorithm performance has been carried out considering different tuning parameters. The selected systems are generally used as a benchmark for model structures algorithms and were taken from [Wei, H. and Billings, S. A.](Model structure selection using an integrated forward

orthogonal search algorithm assisted by squared correlation and mutual information), [Falsone, A. and Piroddi, L. and Prandini, M.](A randomized algorithm for nonlinear model structure selection), [Baldacchino, T. and Anderson, S. R. and Kadirkamanathan, V.](Computational system identification for Bayesian NARMAX modelling), [Piroddi, L. and Spinelli, W.](An identification algorithm for polynomial NARX models based on simulation error minimization), [Guo, Y. and Guo, L. Z. and Billings, S. A. and Wei, H.](A New Iterative Orthogonal Forward Regression Algorithm), [Bonin, M. and Seghezza, V. and Piroddi, L.](NARX model selection based on simulation error minimisation and LASSO), [Aguirre, L. A. and Barbosa, B. H. G. and Braga, A. P.](Prediction and simulation errors in parameter estimation for nonlinear systems). Finally, a comparative analysis with respect to the [Randomized Model Structure Selection (RaMSS)](A randomized algorithm for nonlinear model structure selection), the FROLS, and the [Reversible-jump Markov chain Monte Carlo](Computational system identification for Bayesian NARMAX modelling) (RJMCMC) algorithms has been accomplished to check out the goodness of the proposed method.

The simulation models are described as:

$$
\begin{aligned}
S_1: \quad & y_k = -1.7y_{k-1} - 0.8y_{k-2} + x_{k-1} + 0.81x_{k-2} + e_k, \\
& \text{with } x_k \sim \mathcal{U}(-2, 2) \text{ and } e_k \sim \mathcal{N}(0, 0.01^2); \\
S_2: \quad & y_k = 0.8y_{k-1} + 0.4x_{k-1} + 0.4x_{k-1}^2 + 0.4x_{k-1}^3 + e_k, \\
& \text{with } x_k \sim \mathcal{N}(0, 0.3^2) \text{ and } e_k \sim \mathcal{N}(0, 0.01^2). \\
S_3: \quad & y_k = 0.2y_{k-1}^3 + 0.7y_{k-1}x_{k-1} + 0.6x_{k-2}^2 \\
& - 0.7y_{k-2}x_{k-2}^2 - 0.5y_{k-2} + e_k, \\
& \text{with } x_k \sim \mathcal{U}(-1, 1) \text{ and } e_k \sim \mathcal{N}(0, 0.01^2). \\
S_4: \quad & y_k = 0.7y_{k-1}x_{k-1} - 0.5y_{k-2} + 0.6x_{k-2}^2 \\
& - 0.7y_{k-2}x_{k-2}^2 + e_k, \\
& \text{with } x_k \sim \mathcal{U}(-1, 1) \text{ and } e_k \sim \mathcal{N}(0, 0.04^2). \\
S_5: \quad & y_k = 0.7y_{k-1}x_{k-1} - 0.5y_{k-2} + 0.6x_{k-2}^2 \\
& - 0.7y_{k-2}x_{k-2}^2 + 0.2e_{k-1} \\
& -0.3x_{k-1}e_{k-2} + e_k, \\
& \text{with } x_k \sim \mathcal{U}(-1, 1) \text{ and } e_k \sim \mathcal{N}(0, 0.02^2); \\
S_6: \quad & y_k = 0.75y_{k-2} + 0.25x_{k-2} - 0.2y_{k-2}x_{k-2} + e_k \\
& \text{with } x_k \sim \mathcal{N}(0, 0.25^2) \text{ and } e_k \sim \mathcal{N}(0, 0.02^2);
\end{aligned}
\tag{57}
$$

where $\mathcal{U}(a, b)$ are samples evenly distributed over~$[a, b]$, and $\mathcal{N}(\eta, \sigma^2)$ are samples with a Gaussian distribution with mean η and standard deviation σ. All realizations of the systems are composed of a total of 500 input-output data samples. Also, the same random seed is used to reproducibility purpose.

All tests shown in this section are based on the original implementation and are took from the results of my master thesis. At the time, the algorithm was performed in Matlab 2018a environment, on a Dell Inspiron 5448 Core i$5-5200$U CPU 2.20GHz with 12GB of RAM. However, it is not a hard task to adapt them to SysIdentPy.

Following the aforementioned studies, the maximum lags for the input and output are chosen to be, respectively, $n_u = n_y = 4$ and the nonlinear degree is $\ell = 3$. The parameters related to the BPSOGSA are detailed on Table 8.

Parameters	n_u	n_y	ℓ	**p-value**	**max_iter**	**n_agents**	α	G_0
Values	4	4	3	0.05	30	10	23	100

Table 8. Parameters used in MetaMSS

300 runs of the Meta-MSS algorithm have been executed for each model, aiming to compare some statistics about the algorithm performance. The elapsed time, the time required to obtain the final model, and correctness, the percentage of exact model selections, are analyzed.

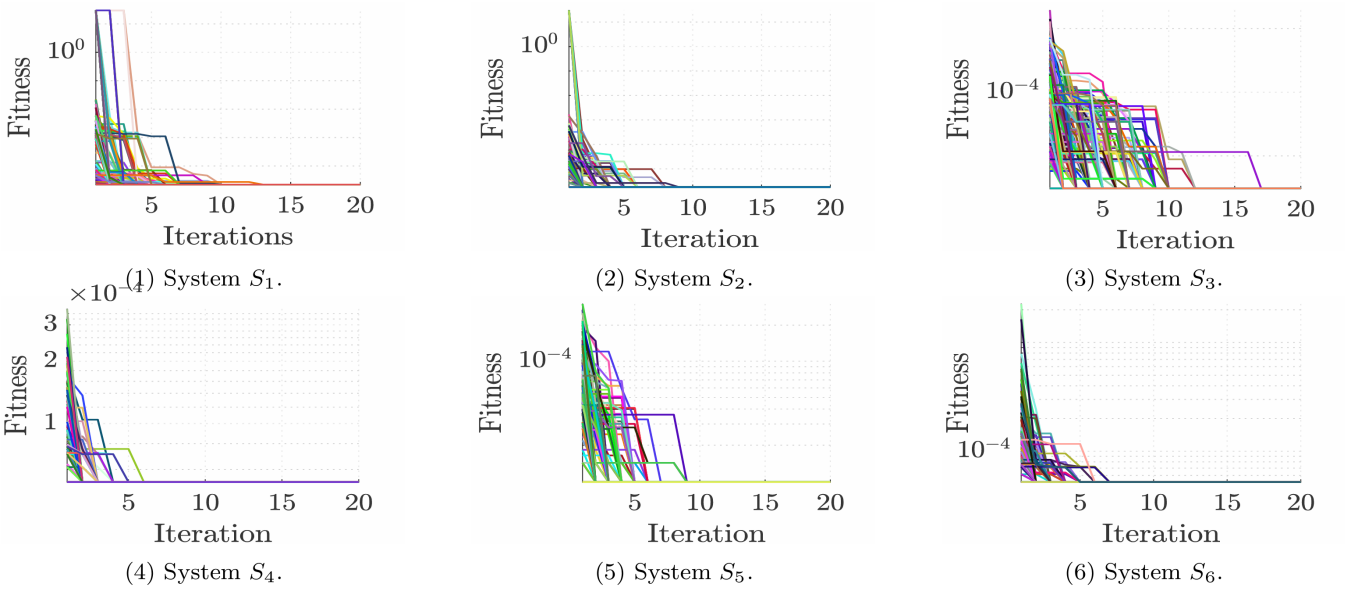
The results in Table 9 are obtained with the parameters configured accordingly to Table 8.

	S_1	S_2	S_3	S_4	S_5	S_6
Correct model	100%	100%	100%	100%	100%	100%
Elapsed time (mean)	5.16s	3.90s	3.40s	2.37s	1.40s	3.80s

Table 9. Overall performance of the MetaMSS

Table 9 shows that all the model terms are correctly selected using the Meta-MSS. It is worth to notice that even the model S_5, which have an autoregressive noise, was correctly selected using the proposed algorithm. This result resides in the evaluation of all regressors individually, and the ones considered redundant are removed from the model.

Figure 15 presents the convergence of each execution of Meta-MSS. It is noticeable that the majority of executions converges to the correct model structures with 10 or fewer iterations. The reason for this relies on the maximum number of iterations and the number of search agents. The first one is related to the acceleration coefficient, which boosts the exploration phase of the algorithm, while the latter increases the number of candidate models to be evaluated. Intuitively, one can see that both parameters influence the elapsed time and, more importantly, the model structure selected to compose the final model. Consequently, an inappropriate choice of one of them may results in sub/over-parameterized models, since the algorithm can converge to a local optimum. The next subsection presents an analysis of the max_iter and n_agents influence in the algorithm performance.



(1) System S_1. (2) System S_2. (3) System S_3.

(4) System S_4. (5) System S_5. (6) System S_6.

Figure 15. Convergence of Meta-MSS for different model structures. The figure illustrates the convergence behavior of the Meta-MSS algorithm across multiple executions. Each curve represents the convergence trajectory for a specific model structure from S_1 to S_6 over a maximum of 30 iterations.

Influence of the max_iter and n_agents parameters

The simulation models are used to show the performance of the Meta-MSS considering different tuning for `max_iter` and `n_agents` parameters. First, we set and uphold the `max_iter=30` while the `n_agents` are changed. Then, we set and uphold the `n_agents` while the `max_iter` is modified. The results detailed in this section have been obtained by setting the remaining parameters according to Table 8.

		S_1	S_2	S_3	S_4	S_5	S_6
max_iter = 30, n_agents = 1	**Correct model**	65%	55.66%	14%	14%	7.3%	20.66%
	Elapsed time (mean)	0.26s	0.19s	0.15s	0.11s	0.13s	0.13s
max_iter = 30, n_agents = 5	**Correct model**	100%	100%	99%	98%	91.66%	98.33%
	Elapsed time (mean)	2.08s	1.51s	1.41s	0.99s	0.59s	1.13s
max_iter = 30, n_agents = 20	**Correct model**	100%	100%	100%	100%	100%	100%
	Elapsed time (mean)	12.88s	9.10s	8.77s	5.70s	3.37s	9.50s
max_iter = 5, n_agents = 10	**Correct model**	96.33%	99%	86%	93.66%	93%	97.33%

		S_1	S_2	S_3	S_4	S_5	S_6
	Elapsed time (mean)	0.92s	0.73s	0.72s	0.52s	0.29s	0.64s
max_iter = 15, n_agents = 10	**Correct model**	100%	100%	99%	99%	100%	100%
	Elapsed time (mean)	2.80s	2.33s	2.25s	1.60s	0.90s	2.30s
max_iter = 50, n_agents = 10	**Correct model**	100%	100%	100%	100%	100%	100%
	Elapsed time (mean)	7.38s	5.44s	4.56s	3.01s	2.10s	4.52s

Table 10.

The aggregated results in Table 10 confirms the expected performance regarding the elapsed time and percentage of correct models. Indeed, both metrics increases significantly as the number of agents and the maximum number of iteration increases. The number of agents is very relevant because it yields a broader exploration of the search space. All system are affected by the increase in the number of agents and the maximum number of iterations.

Regarding all tested systems, it is straightforward to notice that the more extensive exploration dramatically impacts on the exactitude of the selection procedure. If only a few agents are assigned, the performance of Meta-MSS algorithm deteriorates significantly, especially for systems S_3, S_4 and S_5. The maximum number of iteration empowers agents to explore, globally and locally, the space around the candidate models tested so far. In this sense, as the number of iterations increases, more the agents can explore the search space and examine different regressors.

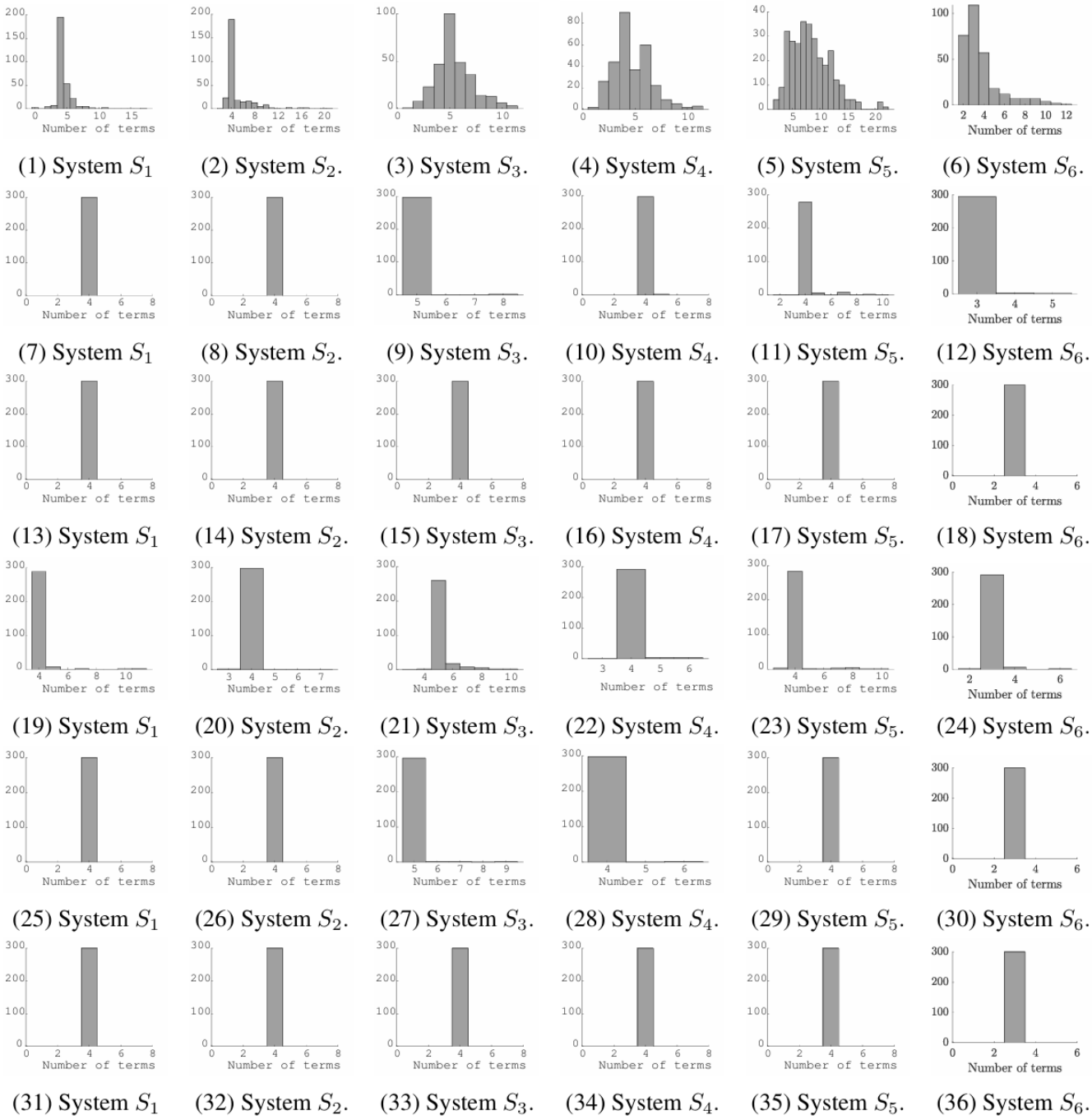
If these parameters are improperly chosen, the algorithm might fail to select the best model structure. In this respect, the results presented here concerns only the selected systems. The larger the search space, the larger the number of agents and iterations should be. Although the computational effort increases with larger values for n_agents and max_iteration, the algorithm remains very efficient regarding the elapsed time for all tuning configurations that ensured the selection of the exact model structures.

Selection of over and sub-parameterized models

Regardless of the successful selection of all models structures by the Meta-Structure Selection Algorithm, one can ask how the models differs from the true ones in the cases presented in Table 10 where the algorithm failed to ensure 100% of correctness. Figure 16 depicts the distribution of terms number selected in each case. It is evident that the number of over-parameterized models selected is higher than the sub-parameterized in overall. Regarding the cases where the number of search agents are low, due to low exploration and exploitation capacity, the algorithm converged early and resulted in models with a high number of spurious regressors. In respect to S_2 and S_5, for example,

with n_agents$= 1$, the algorithm ends up selecting models with more than 20 terms. One can say this was a extreme scenario for comparison purpose. However, a suitable choice for the parameters is intrinsically related to the dimension of the search space. Referring to cases where n_agents≥ 5, the number of spurious terms decreased significantly where the algorithm failed to select the true models.

Furthermore, it is interesting to point out the importance of tuning the parameters properly because since the exploration and exploitation phase of the algorithm are strongly dependent on them. A premature convergence of the algorithm may result in models with the factual number of terms, but with wrong ones. This happened with all cases with `n_agents=1`. For example, the algorithm generates models with correct number of terms in 33.33% of the cases analyzed regarding S_3. However, Table 10 shows that only 14% are, in fact, equivalent to the true model.



(1) System S_1 (2) System S_2. (3) System S_3. (4) System S_4. (5) System S_5. (6) System S_6.

(7) System S_1 (8) System S_2. (9) System S_3. (10) System S_4. (11) System S_5. (12) System S_6.

(13) System S_1 (14) System S_2. (15) System S_3. (16) System S_4. (17) System S_5. (18) System S_6.

(19) System S_1 (20) System S_2. (21) System S_3. (22) System S_4. (23) System S_5. (24) System S_6.

(25) System S_1 (26) System S_2. (27) System S_3. (28) System S_4. (29) System S_5. (30) System S_6.

(31) System S_1 (32) System S_2. (33) System S_3. (34) System S_4. (35) System S_5. (36) System S_6.

Figure 16. The distribution of terms number selected for each simulated models concerning the variation of the `max_iter` and `n_agents`.

Selection of over and sub-parameterized models

Regardless of the successful selection of all models structures by the MetaMSS, one can ask how the models differ from the true ones in the cases presented in Table 10 where the algorithm failed to ensure 100% of correctness. Figure 16 depicts the distribution of terms number selected in each case. It is evident that the number of over-parameterized models selected is higher than the sub-parameterized in overall. Regarding the cases where the number of search agents is low, due to low exploration and exploitation capacity, the algorithm converged early and resulted in models with a high number of spurious regressors. In respect to S_2 and S_5, for example, with `n_agents=1`, the algorithm ends up selecting models with more than 20 terms. One can say this was an extreme scenario for comparison purpose. However, a suitable choice for the parameters is intrinsically related to the dimension of the search space. By referring to cases where `n_agents` ≥ 5, the number of spurious terms decreased significantly where the algorithm failed to select the true models.

Furthermore, it is interesting to point out the importance of tuning the parameters properly because since the exploration and exploitation phase of the algorithm is strongly dependent on them. A premature convergence of the algorithm may result in models with the actual number of terms, but with wrong ones. This issue happened with all cases with `n_agents=1`. For example, the algorithm generates models with the correct number of terms in 33.33% of the cases analyzed regarding S_3. However, Table 10 shows that only 14% are, in fact, equivalent to the true model.

The systems S_1, S_2, S_3, S_4 and S_6 has been used as benchmark by Bianchi, F., Falsone, A., Prandini, M. and Piroddi, L., so we can compare directly our results with those reported by the author in his thesis. All techniques used $n_y = n_u = 4$ and $\ell = 3$. The RaMSS and the RaMSS with Conditional Linear Family (C-RaMSS) used the following configuration for the tuning parameters: $K = 1$, $\alpha = 0.997$, $NP = 200$ and $v = 0.1$. The Meta-Structure Selection Algorithm was tuned according to Table 8.

		S_1	S_2	S_3	S_4	S_6
Meta-MSS	**Correct model**	100%	100%	100%	100%	100%
	Elapsed time (mean)	5.16s	3.90s	3.40s	2.37s	3.80s
RaMSS- $NP = 100$	**Correct model**	90.33%	100%	100%	100%	66%
	Elapsed time (mean)	3.27s	1.24s	2.59s	1.67s	6.66s
RaMSS- $NP = 200$	**Correct model**	78.33%	100%	100%	100%	82%
	Elapsed time (mean)	6.25s	2.07s	4.42s	2.77s	9.16s
C-RaMSS	**Correct model**	93.33%	100%	100%	100%	100%
	Elapsed time (mean)	18s	10.50s	16.96s	10.56s	48.52s

Table 11. Comparative analysis between MetaMSS, RaMSS, and C-RaMSS

In terms of correctness, the MetaMSS outperforms (or at least equals) the RaMSS and C-RaMSS for all analyzed systems as shown in Table 11. Regarding S_6, the correctness rate increased by 18% when compared with RaMSS and the elapsed time required for C-RaMSS obtain 100% of correctness is 1276.84% higher than the MetaMSS. Furthermore, the MetaMSS is notably more computationally efficient than C-RaMSS and similar to RaMSS.

MetaMSS vs FROLS

The FROLS algorithm was applied to all tested systems, with the results summarized in Table 12. The algorithm successfully selected the correct model terms for S_2 and S_6. However, it failed to identify two out of four regressors for S_1. For S_3, FROLS included y_{k-1} instead of the correct term y_{k-1}^3. Similarly, S_4 incorrectly included y_{k-4} rather than the required term y_{k-2}. Additionally, for S_5, the algorithm produced an incorrect model structure by including the spurious term y_{k-4}.

	Meta-MSS Regressor	**Correct**	**FROLS Regressor**	**Correct**
S_1	y_{k-1}	yes	y_{k-1}	yes
	y_{k-2}	yes	y_{k-4}	no
	x_{k-1}	yes	x_{k-1}	yes
	x_{k-2}	yes	x_{k-4}	no
S_2	y_{k-1}	yes	y_{k-1}	yes
	x_{k-1}	yes	x_{k-1}	yes
	x_{k-1}^2	yes	x_{k-1}^2	yes
	x_{k-1}^3	yes	x_{k-1}^3	yes
S_3	y_{k-1}^3	yes	y_{k-1}	no
	$y_{k-1}x_{k-1}$	yes	$y_{k-1}x_{k-1}$	yes
	x_{k-2}^2	yes	x_{k-2}^2	yes
	$y_{k-2}x_{k-2}^2$	yes	$y_{k-2}x_{k-2}^2$	yes
	y_{k-2}	yes	y_{k-2}	yes
S_4	$y_{k-1}x_{k-1}$	yes	$y_{k-1}x_{k-1}$	yes
	y_{k-2}	yes	y_{k-4}	no
	x_{k-2}^2	yes	x_{k-2}^2	yes
	$y_{k-2}x_{k-2}^2$	yes	$y_{k-2}x_{k-2}^2$	yes
S_5	$y_{k-1}x_{k-1}$	yes	$y_{k-1}x_{k-1}$	yes
	y_{k-2}	yes	y_{k-4}	no
	x_{k-2}^2	yes	x_{k-2}^2	yes
	$y_{k-2}x_{k-2}^2$	yes	$y_{k-2}x_{k-2}^2$	yes

	Meta-MSS Regressor	Correct	FROLS Regressor	Correct
S_6	y_{k-2}	yes	y_{k-2}	yes
	x_{k-1}	yes	x_{k-1}	yes
	$y_{k-2}x_{k-2}$	yes	$y_{k-2}x_{k-1}$	yes

Table 12. Comparative analysis between MetaMSS and FROLS

Meta-MSS vs RJMCMC

The S_4 model is taken from Baldacchino, Anderson, and Kadirkamanathan's work (Computational System Identification for Bayesian NARMAX Modelling). In their study, the maximum lags for the input and output are $n_y = n_u = 4$, and the nonlinear degree is $\ell = 3$. The authors ran the RJMCMC algorithm 10 times on the same input-output data. The RJMCMC method successfully identified the true model structure 7 out of 10 times. In contrast, the MetaMSS algorithm consistently identified the true model structure in all runs. These results are summarized in Table 13.

Additionally, the RJMCMC method has notable drawbacks that are addressed by the MetaMSS algorithm. Specifically, RJMCMC is computationally intensive, requiring $30,000$ iterations to achieve results. Furthermore, it relies on various probability distributions to simplify the parameter estimation process, which can complicate the computations. In contrast, MetaMSS offers a more efficient and straightforward approach, avoiding these issues.

	Meta-MSS Model	Correct	RJMCMC Model 1 ($7\times$)	RJMCMC Model 2	RJMCMC Model 3	RJMCMC Model 4	Correct
S_4	$y_{k-1}x_{k-1}$	yes	$y_{k-1}x_{k-1}$	$y_{k-1}x_{k-1}$	$y_{k-1}x_{k-1}$	$y_{k-1}x_{k-1}$	yes
	y_{k-2}	yes	y_{k-2}	y_{k-2}	y_{k-2}	y_{k-2}	yes
	x_{k-2}^2	yes	x_{k-2}^2	x_{k-2}^2	x_{k-2}^2	x_{k-2}^2	yes
	$y_{k-2}x_{k-2}^2$	yes	$y_{k-2}x_{k-2}^2$	$y_{k-2}x_{k-2}^2$	$y_{k-2}x_{k-2}^2$	$y_{k-2}x_{k-2}^2$	yes
	-	-	-	$y_{k-3}x_{k-3}$	x_{k-4}^2	$x_{k-1}x_{k-3}^2$	no

Table 13. Comparative analysis between MetaMSS and RJMCMC.

MetaMSS algorithm using SysIdentPy

Consider the same data used in the Overview of the Information Criteria Methods.

```
from sysidentpy.model_structure_selection import MetaMSS


basis_function = Polynomial(degree=2)
model = MetaMSS(
    ylag=2,
```

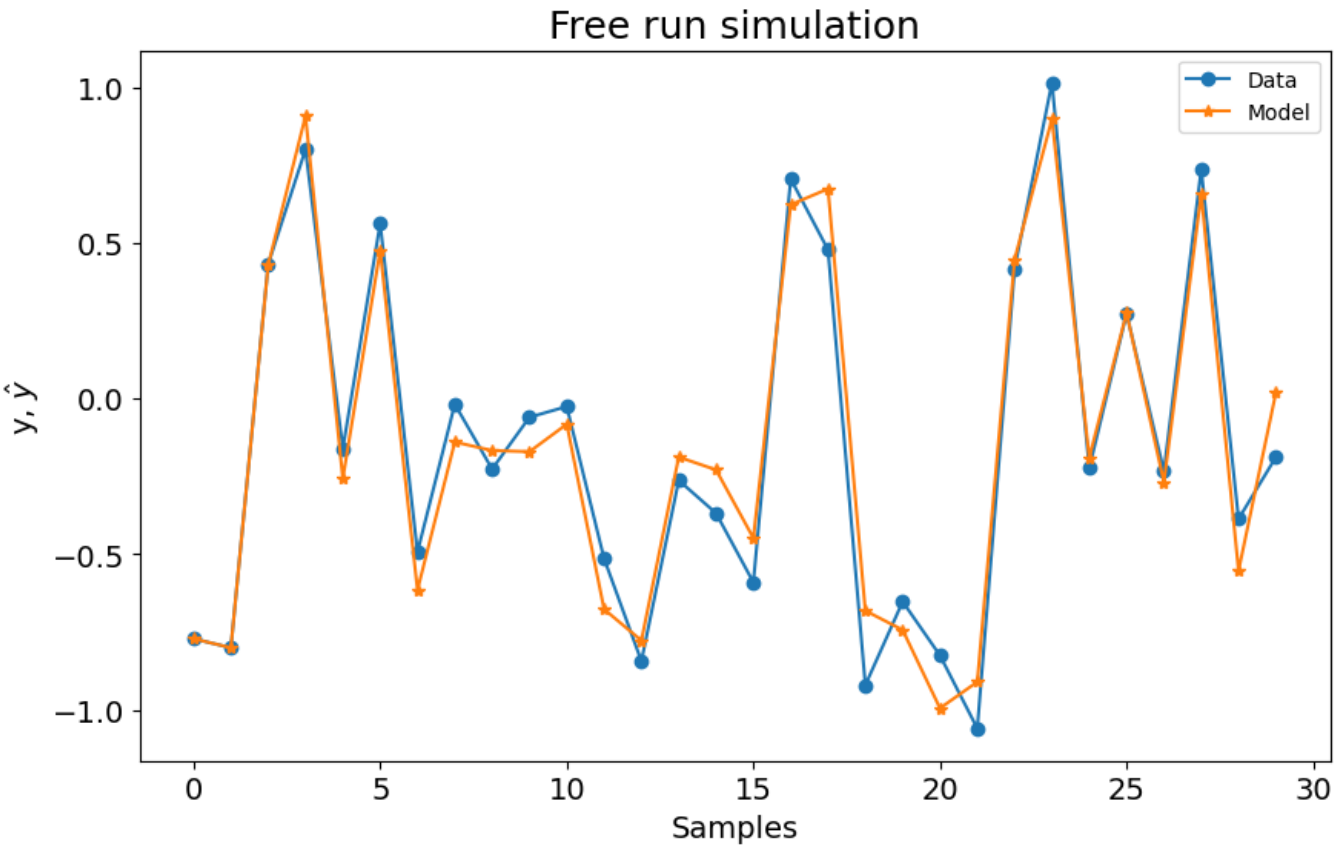
```
    xlag=2,
    random_state=42,
    basis_function=basis_function,
)

model.fit(X=x_train, y=y_train)
yhat = model.predict(X=x_valid, y=y_valid)

r = pd.DataFrame(
    results(
        model.final_model,
        model.theta,
        model.err,
        model.n_terms,
        err_precision=8,
        dtype="sci",
    ),
    columns=["Regressors", "Parameters", "ERR"],
)
print(r)
plot_results(y=y_valid, yhat=yhat, n=1000)
```

The MetaMSS algorithm does not rely on information criteria methods such as ERR for model structure selection, which is why it does not involve those hyperparameters. This is also true for the AOLS and ER algorithms. For more details on how to use these methods and their associated hyperparameters, please refer to the documentation.

When it comes to parameter estimation, SysIdentPy allows the use of any available method, regardless of the model structure selection algorithm. Users can select from a range of parameter estimation methods to apply to their chosen model structure. This flexibility enables users to explore various modeling approaches and customize their system identification process. While the examples provided use the default parameter estimation method, users are encouraged to experiment with different options to find the best fit for their needs.

The results of the MetaMSS are

Regressors	Parameters	ERR
y(k-1)	1.8004E-01	0.00000000E+00
x1(k-2)	8.9747E-01	0.00000000E+00



> Figure 17. Free Run Simulation for the model fitted using MetaMSS.

The `results` method brings ERR as 0 for every regressor because, as mentioned, ERR algorithm is not executed in this case.

Accelerated Orthogonal Least Squares (AOLS) and Entropic Regression (ER)

In addition to FROLS and MetaMSS, SysIdentPy includes two other methods for model structure selection: Accelerated Orthogonal Least Squares (AOLS) and Entropic Regression (ER). While I won't delve into the details of these methods in this section as I have with FROLS and MetaMSS, I will provide an overview and references for further reading:

- **Accelerated Orthogonal Least Squares (AOLS):** For an in-depth exploration of AOLS, refer to the original paper here.
- **Entropic Regression (ER):** Detailed information about ER can be found in the original paper here.

For now, I will demonstrate how to use these methods within SysIdentPy.

Accelerated Orthogonal Least Squares

```
from sysidentpy.model_structure_selection import AOLS

basis_function = Polynomial(degree=2)
model = AOLS(
    ylag=2,
    xlag=2,
    basis_function=basis_function,
)

model.fit(X=x_train, y=y_train)
yhat = model.predict(X=x_valid, y=y_valid)

r = pd.DataFrame(
    results(
        model.final_model,
        model.theta,
        model.err,
        model.n_terms,
        err_precision=8,
        dtype="sci",
    ),
    columns=["Regressors", "Parameters", "ERR"],
)
print(r)
plot_results(y=y_valid, yhat=yhat, n=1000)
```

Regressors	Parameters	ERR
x1(k-2)	9.1542E-01	0.00000000E+00

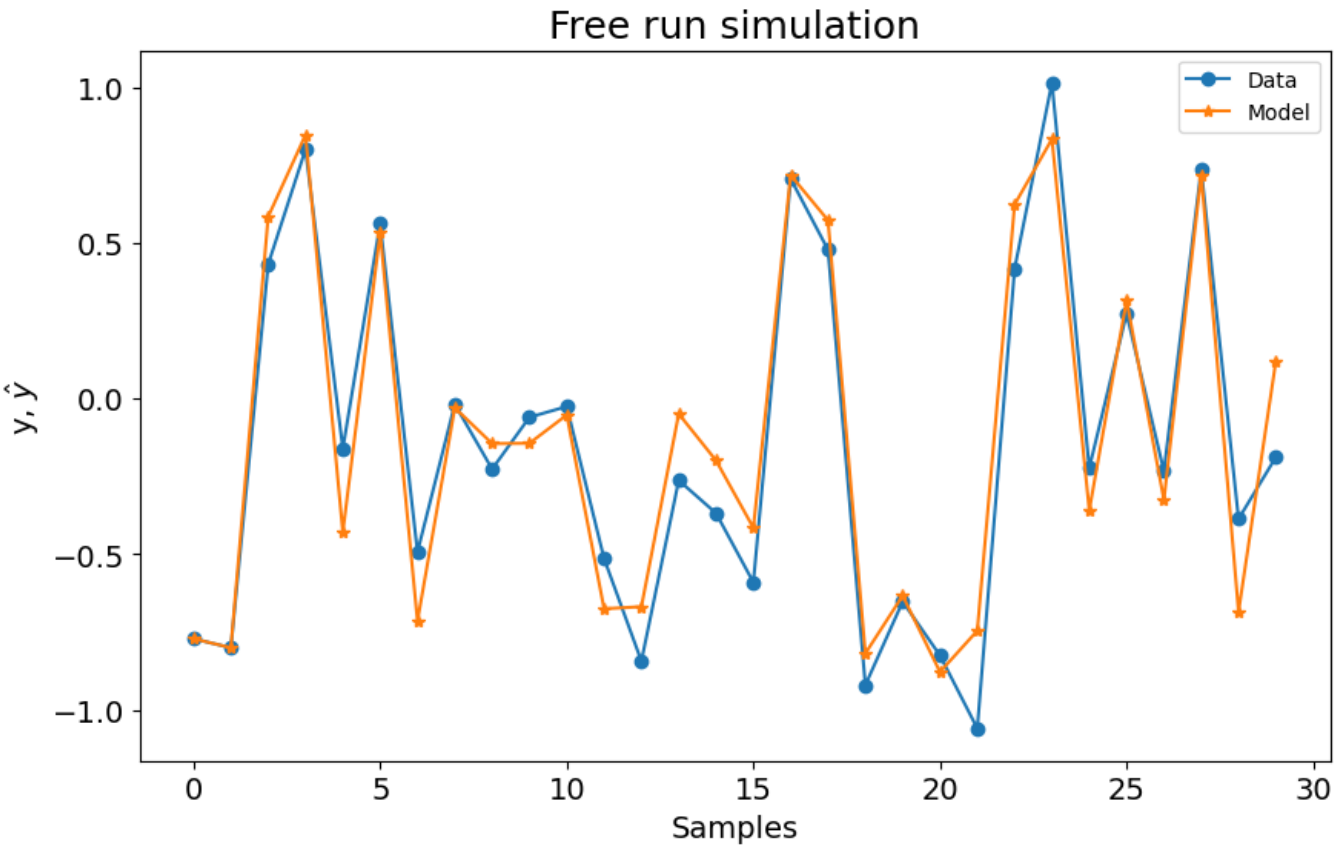


Figure 18. Free Run Simulation for the model fitted using AOLS algorithm.

Entropic Regression

```
from sysidentpy.model_structure_selection import ER

basis_function = Polynomial(degree=2)
model = ER(
    ylag=2,
    xlag=2,
    basis_function=basis_function,
)
model.fit(X=x_train, y=y_train)
yhat = model.predict(X=x_valid, y=y_valid)

r = pd.DataFrame(
    results(
        model.final_model,
        model.theta,
        model.err,
        model.n_terms,
        err_precision=8,
        dtype="sci",
    ),
    columns=["Regressors", "Parameters", "ERR"],
```

```
)

print(r)
plot_results(y=y_valid, yhat=yhat, n=1000)
```

Regressors	Parameters	ERR
1	-2.4554E-02	0.00000000E+00
x1(k-2)	9.0273E-01	0.00000000E+00

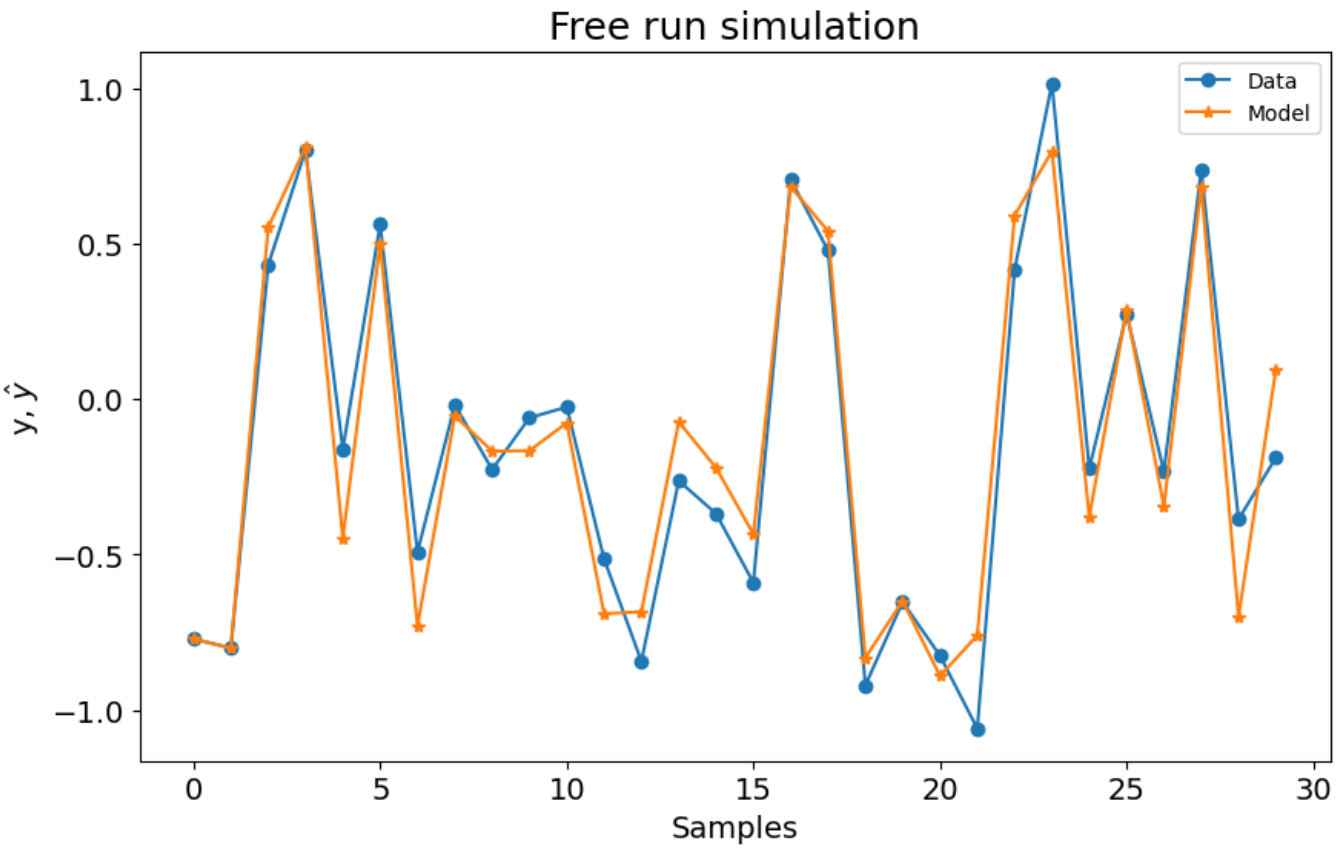


Figure 19. Free Run Simulation for the model fitted using Entropic Regression algorithm.

5 - Multiobjective Parameter Estimation

Multiobjective parameter estimation represents a fundamental paradigm shift in the way we approach the parameter tuning problem for NARMAX models. Instead of seeking a single set of parameter values that optimally fits the model to the data, multiobjective approaches aim to identify a set of parameter solutions, known as the Pareto front, that provide a trade-off between competing objectives. These objectives often encompass a spectrum of model performance criteria, such as goodness-of-fit, model complexity, and robustness.

What does that mean? It means that when we are modeling a dynamical system we are, most of the time, building models that are only good to represent the dynamical behavior of the system under study. Well, that is valid most of the time because we are building dynamical models, so if it doesn't perform well in static scenarios, it won't be a problem. However, that's not always the case and we might build a model that is good in both dynamical and static behavior. In such cases, the methods for purely dynamical systems are not adequate and multiobjective algorithms can help us in such task.

The main idea in multiobjective parameter estimation is the inclusion of the *affine information*. The affine information is an auxiliary information that can be defined *a priori* such as the static gain and the static function of the system. Formally, the affine information can be defined as follows:

Let the parameter vector $\Theta \in \mathbb{R}^{n_\Theta}$, a vector $\mathrm{v} \in \mathbb{R}^p$ and a matrix $\mathrm{G} \in \mathbb{R}^{n_\Theta \times p}$ where v and G are assumed to be accessible. Suppose $\mathrm{G}\Theta$ be an estimate of v. Hence $\mathrm{v} = \mathrm{G}\Theta + \xi$. Then, $[\mathrm{v}, \mathrm{G}]$ is a pair of affine information of the system.

Multi-objective optimization problem

Lets define what is a multiobjective problem. Given m objective functions

$$\mathrm{J}(\hat{\Theta}) = [J_1(\hat{\Theta}), J_2(\hat{\Theta}), \cdots, J_m(\hat{\Theta})]^\top, \tag{5.1}$$

where $\mathrm{J}(\cdot) : \mathbb{R}^n \mapsto \mathbb{R}^m$, a multi-objective optimization problem can be generally stated as (A. Baykasoglu, S. Owen, e N. Gindy)

$$\begin{aligned} \underset{\Theta}{\text{minimize}} \quad & \mathrm{J}(\Theta) \\ \text{subject to} \quad & \Theta \in \mathrm{S} = \{\Theta \mid \Theta \in \mathrm{A}^n, g_i(\Theta) \leq a_i, h_j(\Theta) = b_j\}, \\ & i = 1, \ldots, m, \quad j = 1, \ldots, n \end{aligned} \tag{5.2}$$

where Θ is an n-dimensional vector of the decision variables, S is the set of feasible solutions bounded by m inequality constraints (g_i) and n equality constraints (h_j), and a_i and b_j are constants. For continuous variables $A = \mathbb{R}$ while A contains the set of permissible values for discrete variables.

> Usually problems with $1 < m < 4$ are *called multiobjective optimization problems*. When there are more objectives ($m \geq 4$), it is referred as *many-objective optimization problems*, a emergence

> class of multi-objective problems for solving complex modern real-world tasks. More details can be found in [Fleming, P. J. and Purshouse, R. C. and Lygoe, R. J.](Many-Objective Optimization: An Engineering Design Perspective), [Li, B. and Li, J. and Tang, K. and Yao, X.](A survey on multi-objective evolutionary algorithms for many-objective problems).

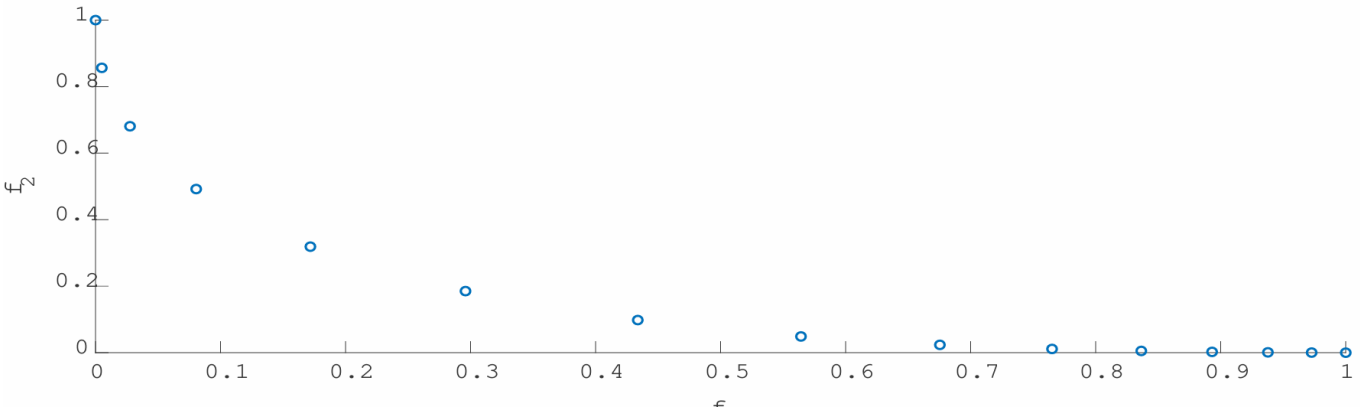
Pareto Optimal Definition and Pareto Dominance

> Consider $[y^{(1)}, y^{(2)}] \in \mathbb{R}^m$ two vectors in the objective space. If and only if $\forall \in \{1, \ldots, m\} : y_i^{(1)} \leq y_i^{(2)}$ and $\exists j \in \{1, \ldots, m\} : y_j^{(1)} < y_j^{(2)}$ one can said $y^{(1)} \prec y^{(2)}$ ([P. L. Yu](Cone convexity, cone extreme points, and nondominated solutions in decision problems with multiobjectives)).

The concept of Pareto optimality is generally used to describe the trade-off among the minimization of different objectives. Following the pareto definition: the pareto optimal is any parameter vector representing an efficient solution where no objective function can be improved without making at least one objective function worse off will be referred to as a Pareto-model.

In the system identification field, that means to find a model where you can't get a better dynamic performance without making the static performance worse.

A hypothetical Pareto set is shown in Figure 1.



> Figure 1. The figure illustrates the concept of Pareto optimality, where each point in the objective space represents a solution. The Pareto front is depicted as a curve, demonstrating the trade-off between two conflicting objectives. Points on the front cannot be improved in one objective without worsening the other, highlighting the balance in optimal solutions.

In this case the model structure is assumed to be known and therefore there is a one-to-one correspondence between each parameter vector on the Pareto optimal solution and a model ([Nepomuceno, E. G. and Takahashi, R. H. C. and Aguirre, L. A.](Multiobjective parameter estimation for non-linear systems: affine information and least-squares formulation)). One can build a Pareto set by applying the Weighted Sum Method, where a set of objectives are scalarized into a single objective by adding each objective multiplied by a user supplied weight. Consider

$$\mathrm{W} = \left\{ w | w \in \mathbb{R}^m, w_j \geq 0 \quad \text{and} \quad \sum_{j=1}^{m} w_j = 1 \right\} \tag{5.3}$$

as non-negative weights. Then, the convex optimization problem can be stated as

$$\Theta^* = \underset{\Theta}{\operatorname{argmin}} \langle w, \mathrm{J}(\Theta) \rangle \tag{5.4}$$

where w is a combination of weights to the different objectives functions. Therefore the Pareto-set is associated to the set of realizations of $w \in \mathrm{W}$. An efficient single-step computational strategy was presented by ([Nepomuceno, E. G. and Takahashi, R. H. C. and Aguirre, L. A.](Multiobjective parameter estimation for non-linear systems: affine information and least-squares formulation)) for solving Equation 5.4 by means of a Least Squares formulation, which is presented in the following section.

Affine Information Least Squares Algorithm

Consider the m affine information pairs $[\mathrm{v}_i \in \mathbb{R}^{pi}, \mathrm{G}_i \mathbb{R}^{pi \times n}]$ with $i = 1, \ldots, m$. Assume there is exist a full column rank G_i and let M be a model of the form

$$y = \Psi\Theta + \epsilon. \tag{5.5}$$

Then the m affine information pairs can be considered in the parameter estimation by solving

$$\Theta^* = \underset{\Theta}{\operatorname{argmin}} \sum_{i=1}^{m} w_i (\mathrm{v}_i - \mathrm{G}_i\Theta)^\top (\mathrm{v}_i - \mathrm{G}_i\Theta) \tag{5.6}$$

with $w = [w_i, \ldots, w_m]^\top \in \mathrm{W}$. The solution of equation above is given by

$$\Theta^* = \left[\sum_{i=1}^{m} w_i \mathrm{G}_i^\top \mathrm{G}_i \right]^{-1} \left[\sum_{i=1}^{m} w_i \mathrm{G}_i^\top \mathrm{v}_i \right]. \tag{5.7}$$

If there exists only one information, the problem reduces to the mono-objective Least Squares solution.

To make things straightforward, lets check a detailed case study.

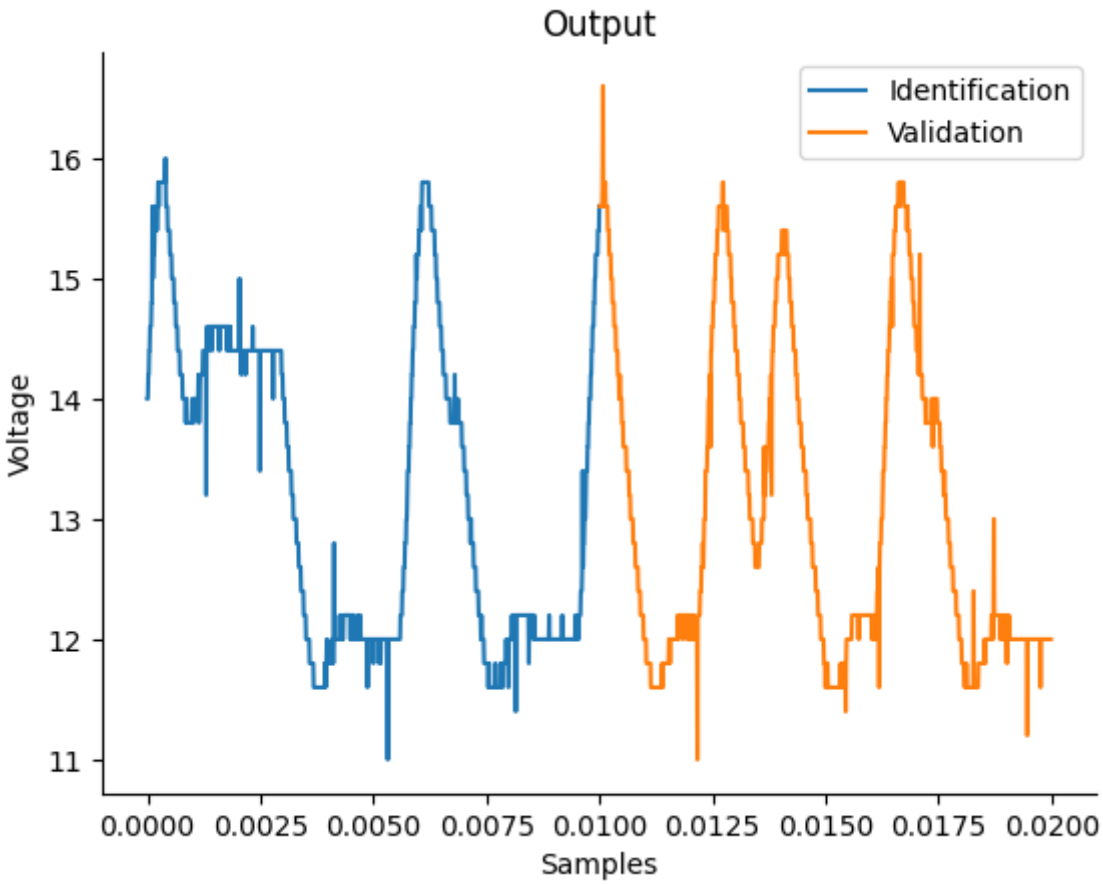
Case Study - Buck converter

A buck converter is a type of DC/DC converter that decreases the voltage (while increasing the current) from its input (power supply) to its output (load). It is similar to a boost converter (elevator) and is a type of switched-mode power supply (SMPS) that typically contains at least two semiconductors (a diode and a transistor, although modern buck converters replace the diode with a second transistor used for synchronous rectification) and at least one energy storage element, a capacitor, inductor or both combined.

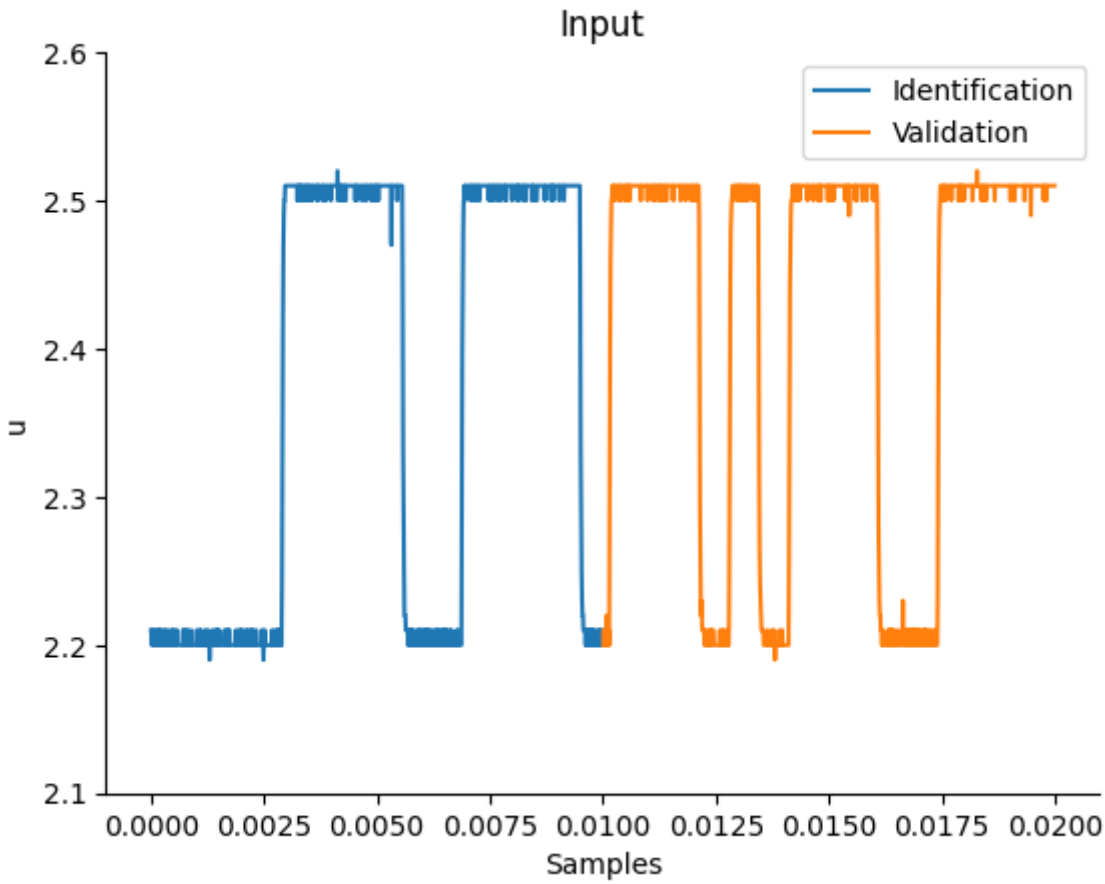
```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
from sysidentpy.model_structure_selection import FROLS
from sysidentpy.multiobjective_parameter_estimation import AILS
from sysidentpy.basis_function._basis_function import Polynomial
from sysidentpy.utils.display_results import results
from sysidentpy.utils.plotting import plot_results
from sysidentpy.metrics import root_relative_squared_error
from sysidentpy.utils.narmax_tools import set_weights
```

Dynamic Behavior

```
df_train = pd.read_csv(r"datasets/buck_id.csv")
df_valid = pd.read_csv(r"datasets/buck_valid.csv")

# Plotting the measured output (identification and validation data)
plt.figure(1)
plt.title("Output")
plt.plot(df_train.sampling_time, df_train.y, label="Identification", linewidth=1.5)
plt.plot(df_valid.sampling_time, df_valid.y, label="Validation", linewidth=1.5)
plt.xlabel("Samples")
plt.ylabel("Voltage")
plt.legend()
plt.show()
```




```
# Plotting the measured input (identification and validation data)
plt.figure(2)
plt.title("Input")
plt.plot(df_train.sampling_time, df_train.input, label="Identification",
linewidth=1.5)
plt.plot(df_valid.sampling_time, df_valid.input, label="Validation", linewidth=1.5)
plt.ylim(2.1, 2.6)
plt.ylabel("u")
plt.xlabel("Samples")
plt.legend()
plt.show()
```



Buck Converter Static Function

The duty cycle, represented by the symbol D, is defined as the ratio of the time the system is on (T_{on}) to the total operation cycle time (T). Mathematically, this can be expressed as $D = \frac{T_{on}}{T}$. The complement of the duty cycle, represented by D', is defined as the ratio of the time the system is off (T_{off}) to the total operation cycle time (T) and can be expressed as $D' = \frac{T_{off}}{T}$.

The load voltage (V_o) is related to the source voltage (V_d) by the equation $V_o = D \cdot V_d = (1 - D') \cdot V_d$. For this particular converter, it is known that $D\prime = \frac{\bar{u}-1}{3}$, which means that the static function of this system can be derived from theory to be:

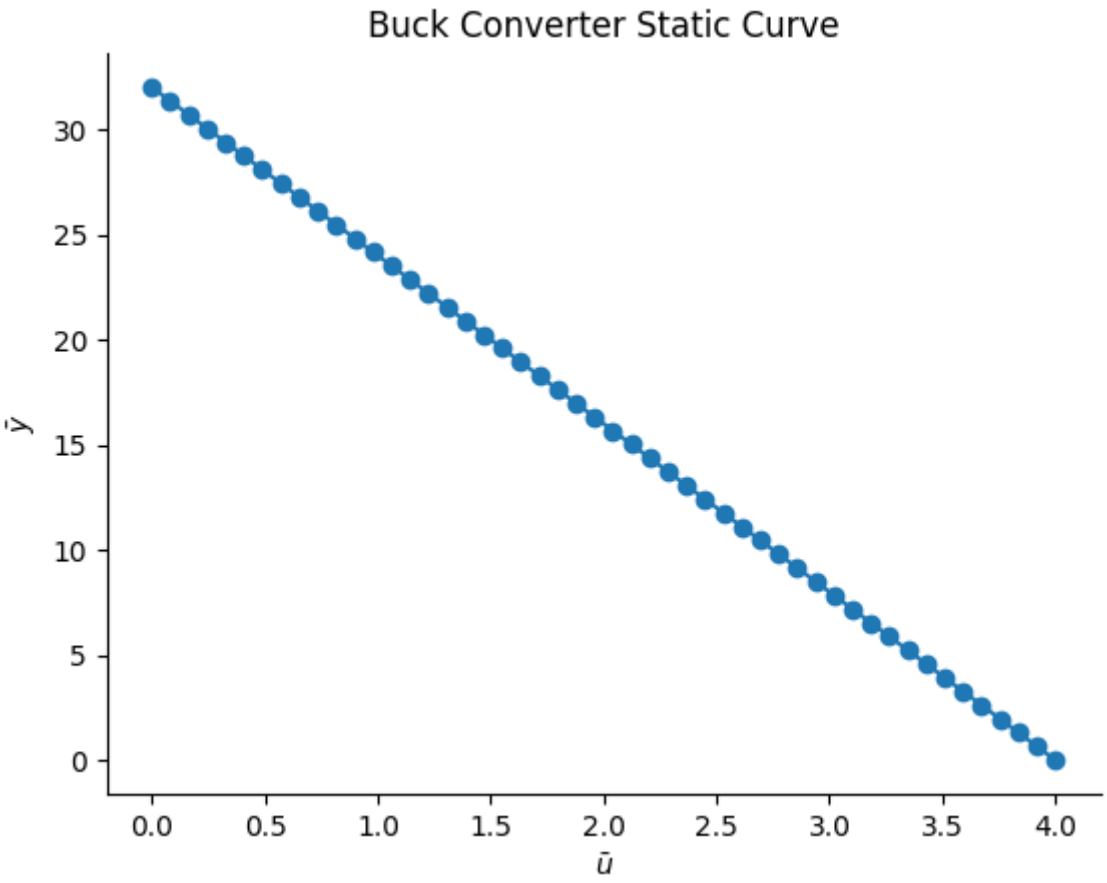
$$V_o = \frac{4V_d}{3} - \frac{V_d}{3} \cdot \bar{u}$$

If we assume that the source voltage V_d is equal to 24 V, then we can rewrite the above expression as follows:

$$V_o = (4 - \bar{u}) \cdot 8$$

```
# Static data
Vd = 24
Uo = np.linspace(0, 4, 50)
Yo = (4 - Uo) * Vd / 3
```

```
Uo = Uo.reshape(-1, 1)
Yo = Yo.reshape(-1, 1)
plt.figure(3)
plt.title("Buck Converter Static Curve")
plt.xlabel("$\\bar{u}$")
plt.ylabel("$\\bar{y}$")
plt.plot(Uo, Yo, linewidth=1.5, linestyle="-", marker="o")
plt.show()
```



Buck converter Static Gain

The gain of a Buck converter is a measure of how its output voltage changes in response to changes in its input voltage. Mathematically, the gain can be calculated as the derivative of the converter's static function, which describes the relationship between its input and output voltages.

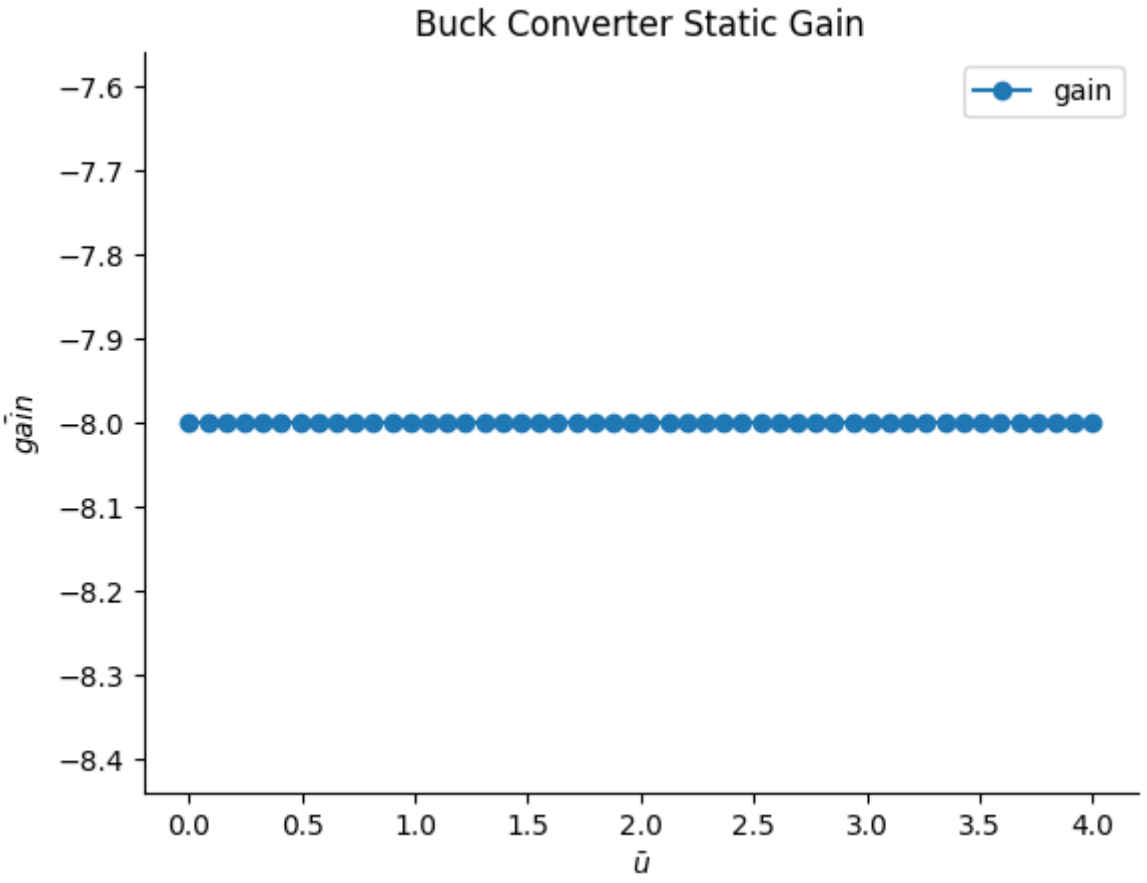
In this case, the static function of the Buck converter is given by the equation:

$$V_o = (4 - \bar{u}) \cdot 8$$

Taking the derivative of this equation with respect to $\hat{u}$, we find that the gain of the Buck converter is equal to −8. In other words, for every unit increase in the input voltage $\hat{u}$, the output voltage Vo will decrease by 8 units, so

$$gain = V_o' = -8$$

```
# Defining the gain
gain = -8 * np.ones(len(Uo)).reshape(-1, 1)
plt.figure(3)
plt.title("Buck Converter Static Gain")
plt.xlabel("$\\bar{u}$")
plt.ylabel("$\\bar{gain}$")
plt.plot(Uo, gain, linewidth=1.5, label="gain", linestyle="-", marker="o")
plt.legend()
plt.show()
```



Building a dynamic model using the mono-objective approach

```
x_train = df_train.input.values.reshape(-1, 1)
y_train = df_train.y.values.reshape(-1, 1)
x_valid = df_valid.input.values.reshape(-1, 1)
y_valid = df_valid.y.values.reshape(-1, 1)

basis_function = Polynomial(degree=2)
model = FROLS(
    order_selection=True,
```

```
    n_info_values=8,
    extended_least_squares=False,
    ylag=2,
    xlag=2,
    info_criteria="aic",
    estimator="least_squares",
    basis_function=basis_function,
)

model.fit(X=x_train, y=y_train)
```

Affine Information Least Squares Algorithm (AILS)

AILS is a multiobjective parameter estimation algorithm, based on a set of affine information pairs. The multiobjective approach proposed in the mentioned paper and implemented in SysIdentPy leads to a convex multiobjective optimization problem, which can be solved by AILS. AILS is a LeastSquares-type non-iterative scheme for finding the Pareto-set solutions for the multiobjective problem.

So, with the model structure defined (we will be using the one built using the dynamic data above), one can estimate the parameters using the multiobjective approach.

The information about static function and static gain, besides the usual dynamic input/output data, can be used to build the pair of affine information to estimate the parameters of the model. We can model the cost function as:

$$\gamma(\hat{\theta}) = w_1 \cdot J_{LS}(\hat{\theta}) + w_2 \cdot J_{SF}(\hat{\theta}) + w_3 \cdot J_{SG}(\hat{\theta})$$

Multiobjective parameter estimation considering 3 different objectives: the prediction error, the static function and the static gain

```
# you can use any set of model structure you want in your use case, but in this
notebook we will use the one obtained above the compare with other work
mo_estimator = AILS(final_model=model.final_model)
# setting the log-spaced weights of each objective function
w = set_weights(static_function=True, static_gain=True)
# you can also use something like
# w = np.array(
#     [
#         [0.98, 0.7, 0.5, 0.35, 0.25, 0.01, 0.15, 0.01],
#         [0.01, 0.1, 0.3, 0.15, 0.25, 0.98, 0.35, 0.01],
#         [0.01, 0.2, 0.2, 0.50, 0.50, 0.01, 0.50, 0.98],
#     ]
# )
```

```
# to set the weights. Each row correspond to each objective
```

AILS has an `estimate` method that returns the cost functions (J), the Euclidean norm of the cost functions (E), the estimated parameters referring to each weight (theta), the regressor matrix of the gain and static_function affine information HR and QR, respectively.

```
J, E, theta, HR, QR, position = mo_estimator.estimate(
    X=x_train, y=y_train, gain=gain, y_static=Yo, X_static=Uo, weighing_matrix=w
)
result = {
    "w1": w[0, :],
    "w2": w[2, :],
    "w3": w[1, :],
    "J_ls": J[0, :],
    "J_sg": J[1, :],
    "J_sf": J[2, :],
    "||J||:": E,
}
pd.DataFrame(result)
```

w1	w2	w3	J_ls	J_sg	J_sf	$\|\|J\|\|$
0.006842	0.003078	0.990080	0.999970	1.095020e-05	0.000013	0.245244
0.007573	0.002347	0.990080	0.999938	2.294665e-05	0.000016	0.245236
0.008382	0.001538	0.990080	0.999885	6.504913e-05	0.000018	0.245223
0.009277	0.000642	0.990080	0.999717	4.505541e-04	0.000021	0.245182
0.006842	0.098663	0.894495	1.000000	7.393246e-08	0.000015	0.245251
...	...	...	...	...	...	...
0.659632	0.333527	0.006842	0.995896	3.965699e-04	1.000000	0.244489
0.730119	0.263039	0.006842	0.995632	5.602981e-04	0.972842	0.244412
0.808139	0.185020	0.006842	0.995364	8.321071e-04	0.868299	0.244300
0.894495	0.098663	0.006842	0.995100	1.364999e-03	0.660486	0.244160
0.990080	0.003078	0.006842	0.992584	9.825987e-02	0.305492	0.261455

Now we can set theta related to any weight results

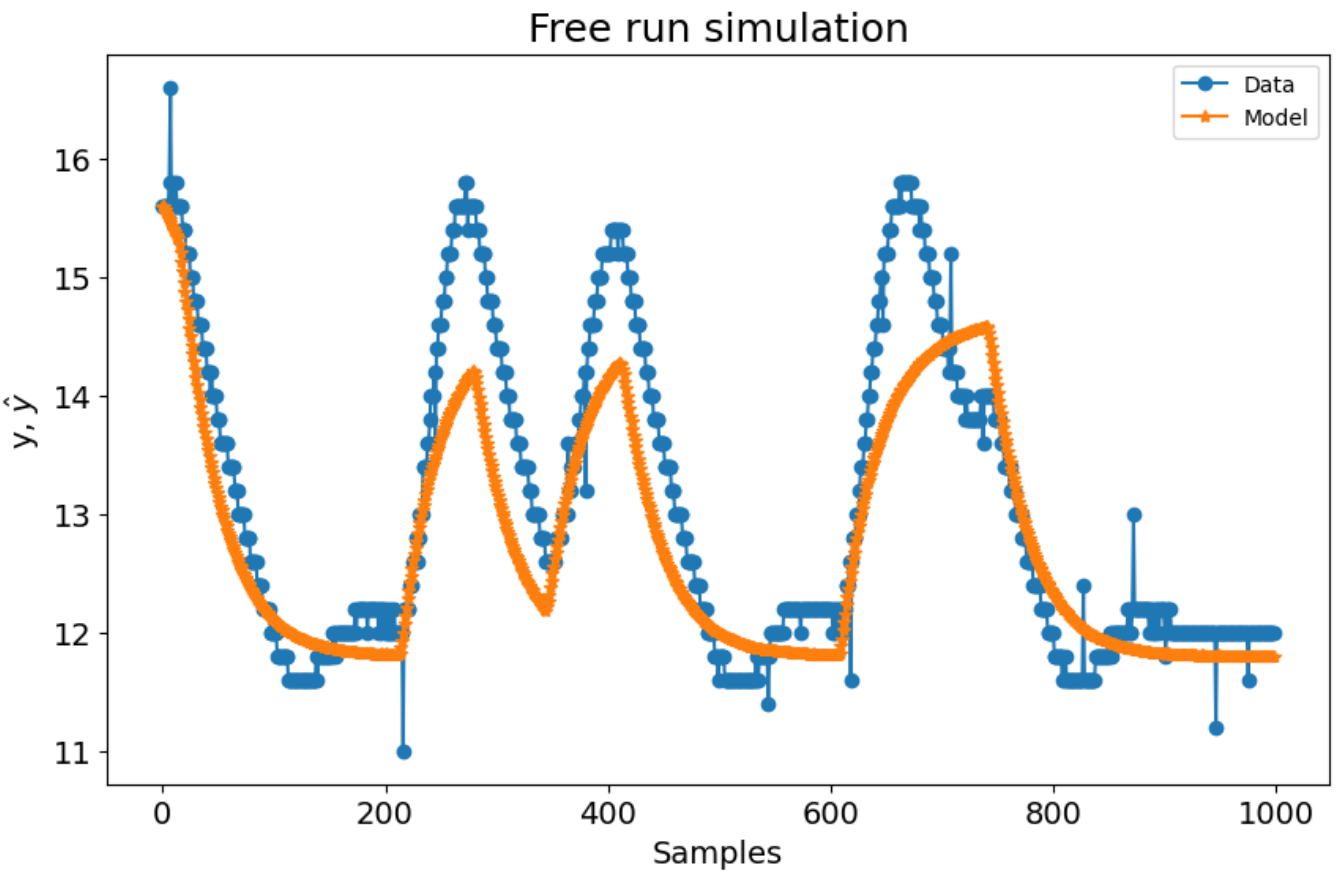
```
model.theta = theta[-1, :].reshape(
    -1, 1
)  # setting the theta estimated for the last combination of the weights

# the model structure is exactly the same, but the order of the regressors is
changed in estimate method. Thats why you have to change the model.final_model
```

```
model.final_model = mo_estimator.final_model
yhat = model.predict(X=x_valid, y=y_valid)
rrse = root_relative_squared_error(y_valid, yhat)
r = pd.DataFrame(
    results(
        model.final_model,
        model.theta,
        model.err,
        model.n_terms,
        err_precision=3,
        dtype="sci",
    ),
    columns=["Regressors", "Parameters", "ERR"],
)
r
```

Regressors	Parameters	ERR
1	2.2930E+00	9.999E-01
y(k-1)	2.3307E-01	2.042E-05
y(k-2)	6.3209E-01	1.108E-06
x1(k-1)	-5.9333E-01	4.688E-06
y(k-1)^2	2.7673E-01	3.922E-07
y(k-2)y(k-1)	-5.3228E-01	8.389E-07
x1(k-1)y(k-1)	1.6667E-02	5.690E-07
y(k-2)^2	2.5766E-01	3.827E-06

The dynamic results for that chosen theta is

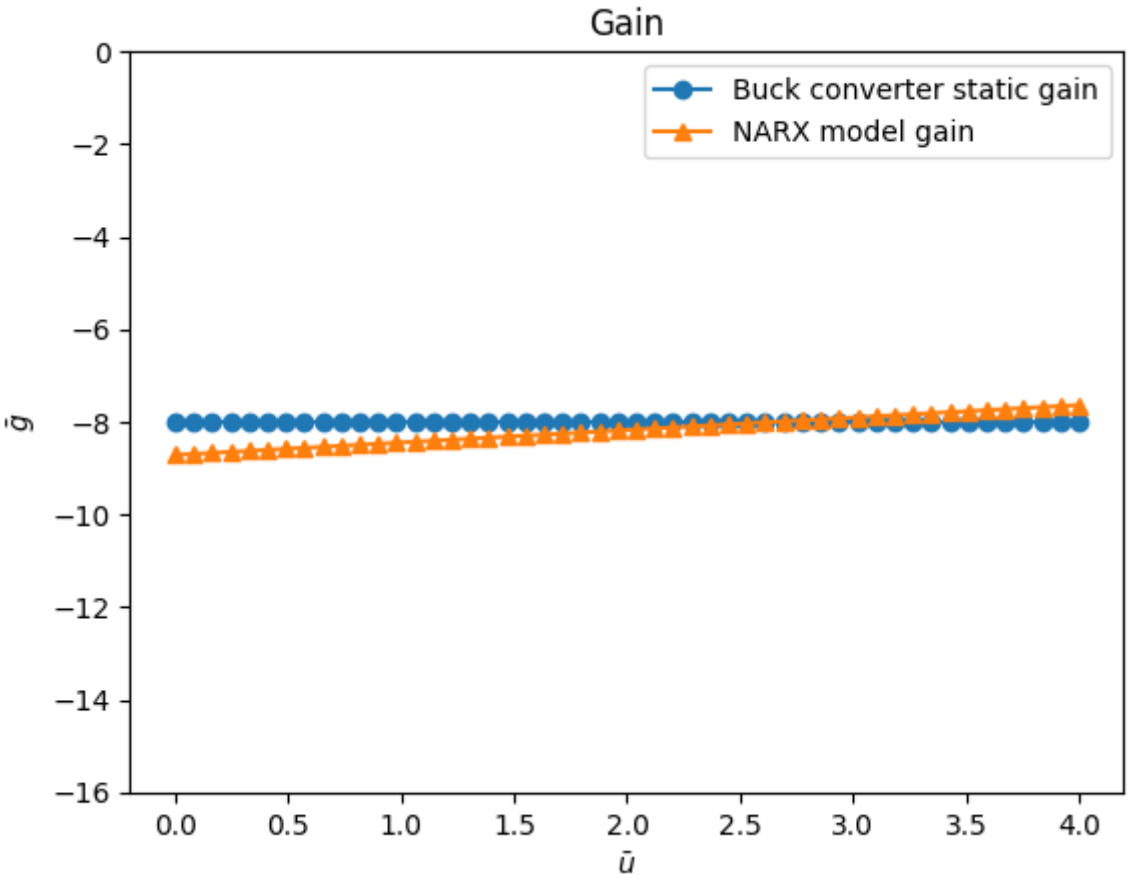
```
plot_results(y=y_valid, yhat=yhat, n=1000)
```



The static gain result is

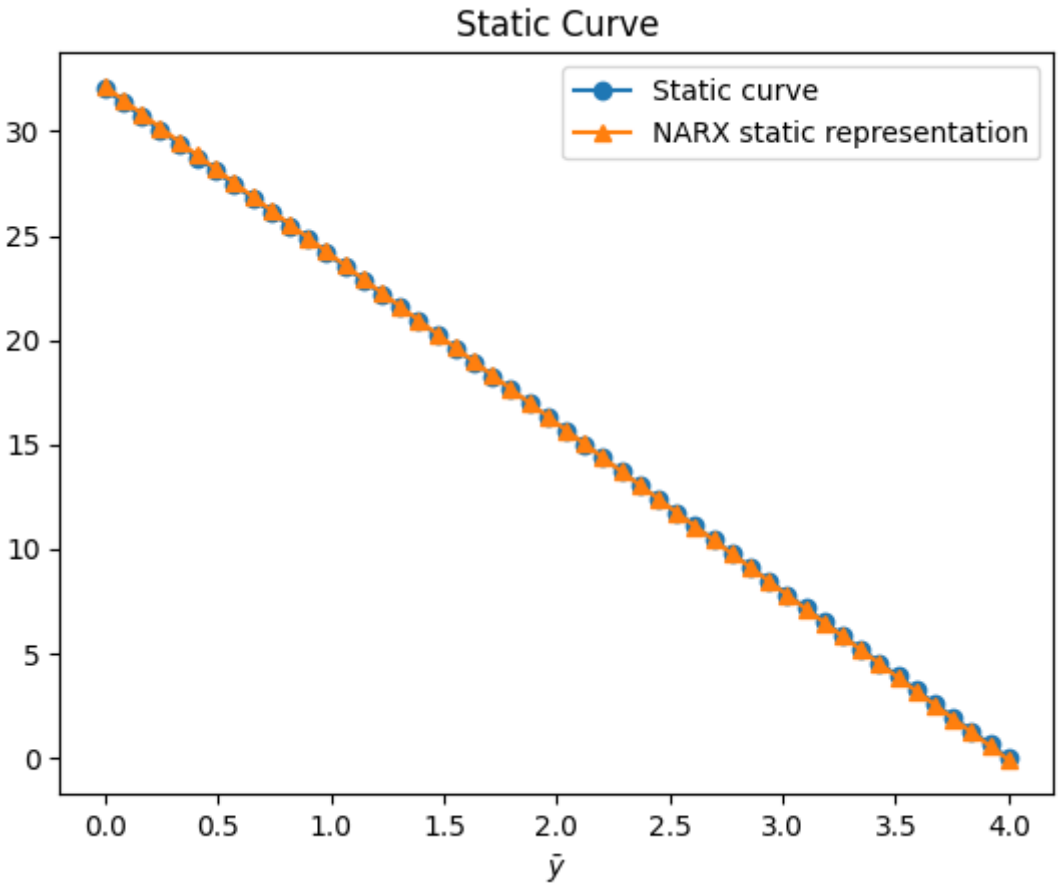
```
plt.figure(4)
plt.title("Gain")
plt.plot(
    Uo,
    gain,
    linewidth=1.5,
    linestyle="-",
    marker="o",
    label="Buck converter static gain",
)
plt.plot(
    Uo,
    HR.dot(model.theta),
    linestyle="-",
    marker="^",
    linewidth=1.5,
    label="NARX model gain",
)
plt.xlabel("$\\bar{u}$")
plt.ylabel("$\\bar{g}$")
plt.ylim(-16, 0)
```

```
plt.legend()
plt.show()
```



The static function result is

```
plt.figure(5)
plt.title("Static Curve")
plt.plot(Uo, Yo, linewidth=1.5, label="Static curve", linestyle="-", marker="o")
plt.plot(
    Uo,
    QR.dot(model.theta),
    linewidth=1.5,
    label="NARX static representation",
    linestyle="-",
    marker="^",
)
plt.xlabel("$\\bar{u}$")
plt.xlabel("$\\bar{y}$")
plt.legend()
plt.show()
```



Getting the best weight combination based on the norm of the cost function

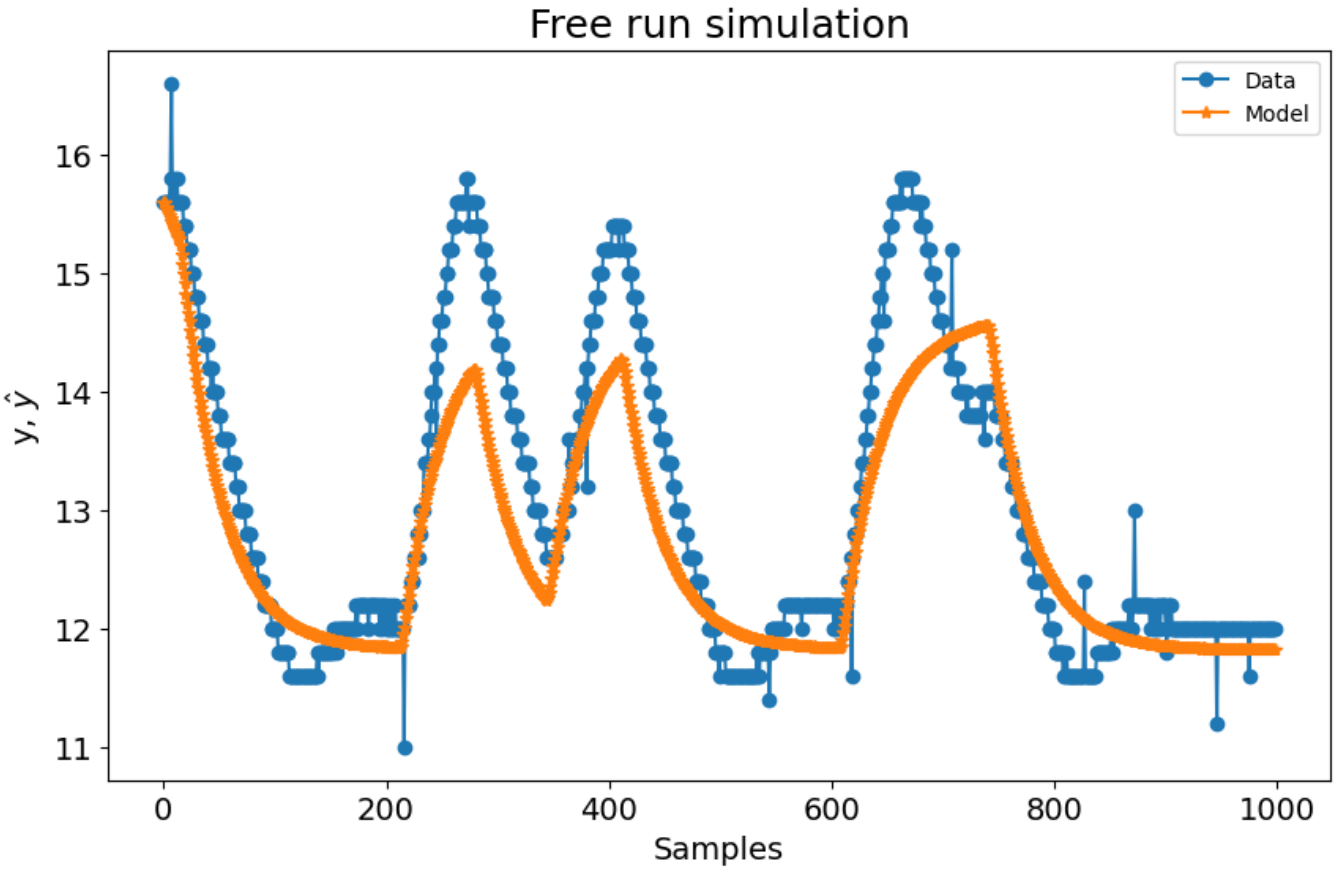
The variable `position` returned in `estimate` method give the position of the best weight combination. The model structure is exactly the same, but the order of the regressors is changed in estimate method. Thats why you have to change the model.final_model. The dynamic, static gain, and the static function results for that chosen theta is shown below.

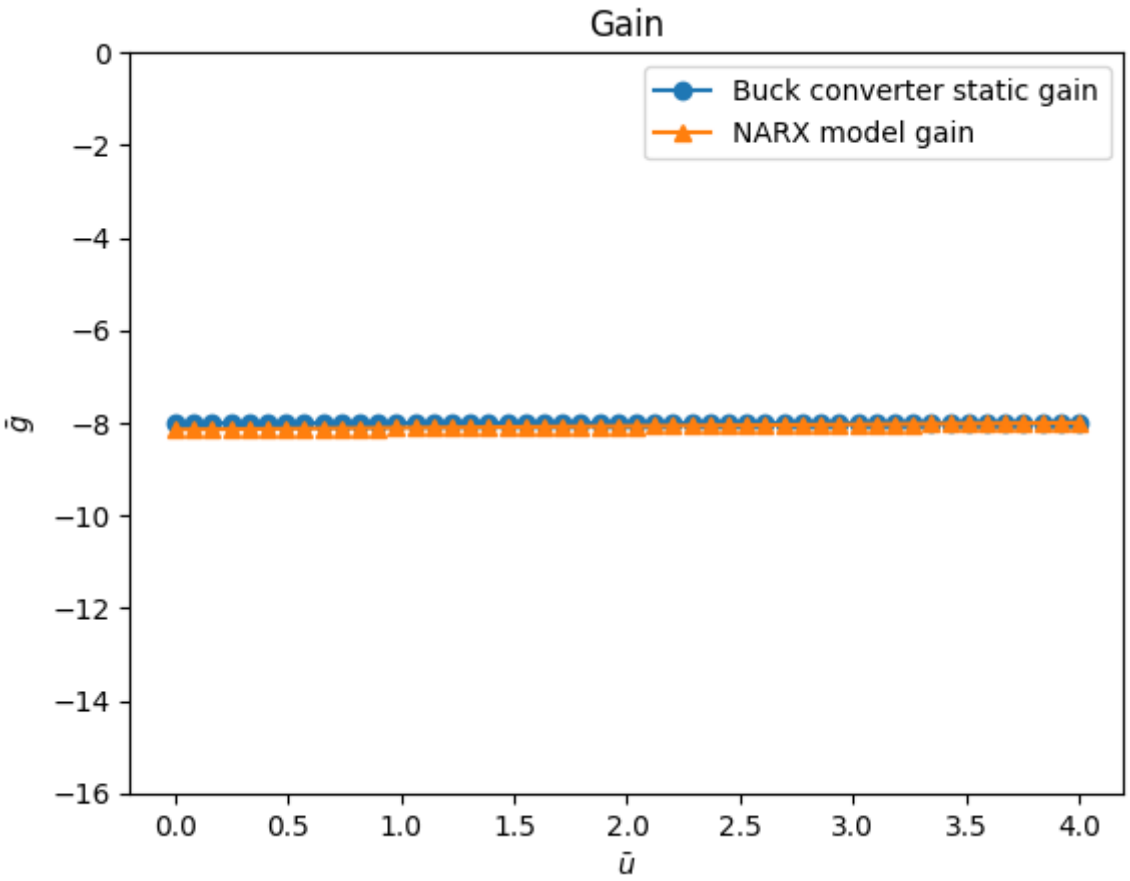
```
model.theta = theta[position, :].reshape(
    -1, 1
)  # setting the theta estimated for the best combination of the weights

# changing the model.final_model

model.final_model = mo_estimator.final_model
yhat = model.predict(X=x_valid, y=y_valid)
rrse = root_relative_squared_error(y_valid, yhat)
r = pd.DataFrame(
    results(
        model.final_model,
        model.theta,
        model.err,
```

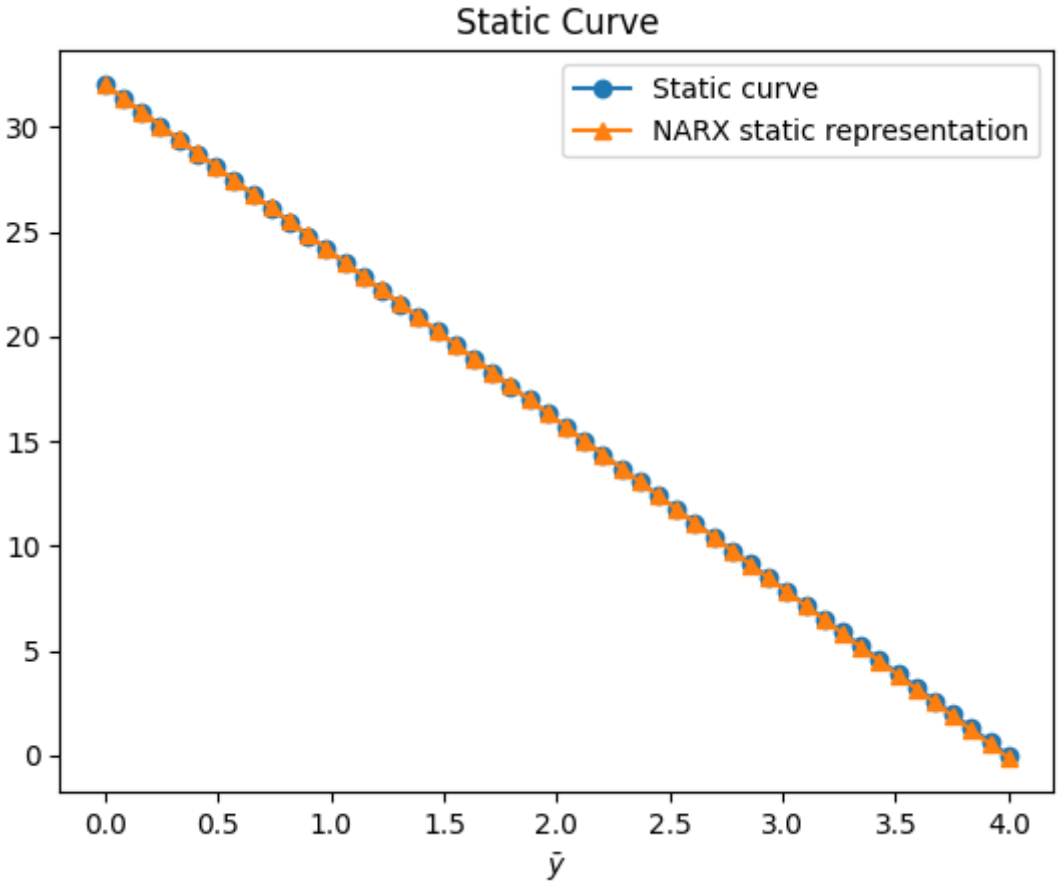
```
        model.n_terms,
        err_precision=3,
        dtype="sci",
    ),
    columns=["Regressors", "Parameters", "ERR"],
)
print(r)

# dynamic results
plot_results(y=y_valid, yhat=yhat, n=1000)

# static gain
plt.figure(4)
plt.title("Gain")
plt.plot(
    Uo,
    gain,
    linewidth=1.5,
    linestyle="-",
    marker="o",
    label="Buck converter static gain",
)
plt.plot(
    Uo,
    HR.dot(model.theta),
    linestyle="-",
    marker="^",
    linewidth=1.5,
    label="NARX model gain",
)
plt.xlabel("$\\bar{u}$")
plt.ylabel("$\\bar{g}$")
plt.ylim(-16, 0)
plt.legend()
plt.show()

# static function
plt.figure(5)
plt.title("Static Curve")
plt.plot(Uo, Yo, linewidth=1.5, label="Static curve", linestyle="-", marker="o")
plt.plot(
    Uo,
    QR.dot(model.theta),
    linewidth=1.5,
    label="NARX static representation",
    linestyle="-",
    marker="^",
)
```

```
plt.xlabel("$\\bar{u}$")
plt.xlabel("$\\bar{y}$")
plt.legend()
plt.show()
```

Regressors	Parameters	ERR
1	1.5405E+00	9.999E-01
y(k-1)	2.9687E-01	2.042E-05
y(k-2)	6.4693E-01	1.108E-06
x1(k-1)	-4.1302E-01	4.688E-06
y(k-1)^2	2.7671E-01	3.922E-07
y(k-2)y(k-1)	-5.3474E-01	8.389E-07
x1(k-1)y(k-1)	4.0624E-03	5.690E-07
y(k-2)^2	2.5832E-01	3.827E-06

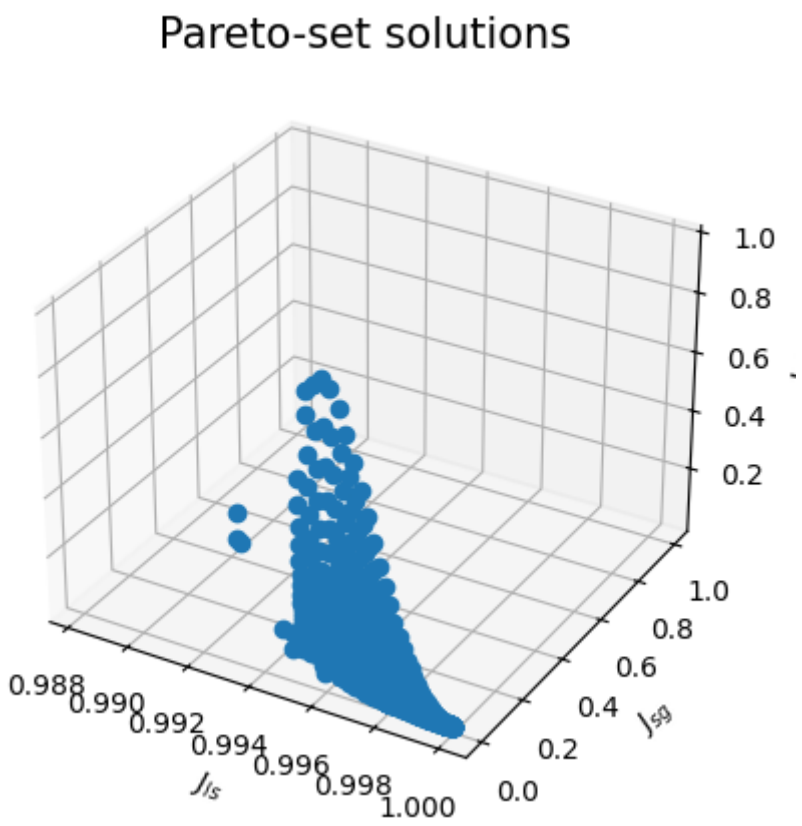



Gain
Buck converter static gain
NARX model gain
0
−2
−4
−6
−8
−10
−12
−14
−16
$\bar{g}$
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
4.0
$\bar{u}$


Static Curve
Static curve
NARX static representation
30
25
20
15
10
5
0
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
4.0
$\bar{y}$

You can also plot the pareto-set solutions

```
plt.figure(6)
ax = plt.axes(projection="3d")
ax.plot3D(J[0, :], J[1, :], J[2, :], "o", linewidth=0.1)
ax.set_title("Pareto-set solutions", fontsize=15)
ax.set_xlabel("$J_{ls}$", fontsize=10)
ax.set_ylabel("$J_{sg}$", fontsize=10)
ax.set_zlabel("$J_{sf}$", fontsize=10)
plt.show()
```



Detailing AILS

The polynomial NARX model built using the mono-objective approach has the following structure:

$$y(k) = \theta_1 y(k-1) + \theta_2 y(k-2) + \theta_3 u(k-1)y(k-1) + \theta_4 + \theta_5 y(k-1)^2 + \theta_6 u(k-1) + \theta_7 y(k-2)y(k-1) + \theta_8$$

The, the goal when using the static function and static gain information in the multiobjective scenario is to estimate the vector $\hat{\theta}$ based on:

$$\theta = [w_1 \Psi^T \Psi + w_2 (HR)^T (HR) + w_3 (QR)(QR)^T]^{-1} [w_1 \Psi^T y + w_2 (HR)^T \overline{g} + w_3 (QR)^T \overline{y}]$$

The Ψ matrix is built using the usual mono-objective dynamic modeling approach in SysIdentPy. However, it is still necessary to find the Q, H and R matrices. AILS have the methods to compute all of

those matrices. Basically, to do that, q_i^T is first estimated:

$$q_i^T = \begin{bmatrix} 1 & \overline{y_i} & \overline{u_1} & \overline{y_i}^2 & \cdots & \overline{y_i}^l & nbsp; F_{yu} & \overline{u_i}^2 & \cdots & \overline{u_i}^l \end{bmatrix}$$

where F_{yu} stands for all non-linear monomials in the model that are related to $y(k)$ and $u(k)$, l is the largest non-linearity in the model for input and output terms. For a model with a degree of nonlinearity equal to 2, we can obtain:

$$q_i^T = \begin{bmatrix} 1 & \overline{y_i} & \overline{u_i} & \overline{y_i}^2 & \overline{u_i}\,\overline{y_i} & \overline{u_i}^2 \end{bmatrix}$$

It is possible to encode the q_i^T matrix so that it follows the model encoding defined in SysIdentPy. To do this, 0 is considered as a constant, y_i equal to 1 and u_i equal to 2. The number of columns indicates the degree of nonlinearity of the system and the number of rows reflects the number of terms:

$$q_i = \begin{bmatrix} 0 & 0 \\ 1 & 0 \\ 2 & 0 \\ 1 & 1 \\ 2 & 1 \\ 2 & 2 \end{bmatrix} = \begin{bmatrix} 1 \\ \overline{y_i} \\ \overline{u_i} \\ \overline{y_i}^2 \\ \overline{u_i}\,\overline{y_i} \\ \overline{u_i}^2 \end{bmatrix}$$

Finally, the result can be easily obtained using the ‘regressor_space’ method of SysIdentPy

```
from sysidentpy.narmax_base import RegressorDictionary

object_qit = RegressorDictionary(xlag=1, ylag=1)
R_example = object_qit.regressor_space(n_inputs=1) // 1000
print(f"R = {R_example}")
```

$$R = \begin{bmatrix} 0 & 0 \\ 1 & 0 \\ 2 & 0 \\ 1 & 1 \\ 2 & 1 \\ 2 & 2 \end{bmatrix}$$

such that:

$$\overline{y_i} = q_i^T R\theta$$

and:

$$\overline{g_i} = HR\theta$$

where R is the linear mapping of the static regressors represented by q_i^T. In addition, the H matrix holds affine information regarding $\overline{g_i}$, which is equal to $\overline{g_i} = \frac{d\overline{y}}{d\overline{u}}\Big|_{(\overline{u_i}\ \overline{y_i})}$.

From now on, we will begin to apply the parameter estimation in a multiobjective manner. This will be done with the NARX polynomial model of the BUCK converter in mind. In this context, q_i^T will be generic and will assume a specific format for the problem at hand. For this task, the R_qit method will be used, whose objective is to return the q_i^T related to the model and the matrix of the linear mapping R:

```
R, qit = mo_estimator.build_linear_mapping()
print("R matrix:")
print(R)
print("qit matrix:")
print(qit)
```

$$R = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$

and

$$qit = \begin{bmatrix} 0 & 0 \\ 1 & 0 \\ 0 & 1 \\ 2 & 0 \\ 1 & 1 \end{bmatrix}$$

So

$$q_i = \begin{bmatrix} 0 & 0 \\ 1 & 0 \\ 2 & 0 \\ 1 & 1 \\ 2 & 1 \end{bmatrix} = \begin{bmatrix} 1 \\ \overline{y} \\ \overline{u} \\ \overline{y^2} \\ \overline{u} \cdot \overline{y} \end{bmatrix}$$

You can notice that the method produces outputs consistent with what is expected:

$$y(k) = \theta_1 y(k-1) + \theta_2 y(k-2) + \theta_3 u(k-1)y(k-1) + \theta_4 + \theta_5 y(k-1)^2 + \theta_6 u(k-1) + \theta_7 y(k-2)y(k-1) + \theta_8$$

and:

$$R = \begin{bmatrix} term/\theta & \theta_1 & \theta_2 & \theta_3 & \theta_4 & \theta_5 & \theta_6 & \theta_7 & \theta_8 \\ 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ \overline{y} & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ \overline{u} & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ \overline{y^2} & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 \\ \overline{y}\,\overline{u} & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Validation

The following model structure will be used to validate the approach:

$$y(k) = \theta_1 y(k-1) + \theta_2 y(k-2) + \theta_3 + \theta_4 u(k-1) + \theta_5 u(k-1)^2 + \theta_6 u(k-2)u(k-1) + \theta_7 u(k-2) + \theta_8 u(k-2$$

$$\therefore$$

$$final\_model = \begin{bmatrix} 1001 & 0 \\ 1002 & 0 \\ 0 & 0 \\ 2001 & 0 \\ 2001 & 2001 \\ 2002 & 2001 \\ 2002 & 0 \\ 2002 & 2002 \end{bmatrix}$$

defining in code:

```
final_model = np.array(
    [
        [1001, 0],
        [1002, 0],
        [0, 0],
        [2001, 0],
        [2001, 2001],
        [2002, 2001],
        [2002, 0],
        [2002, 2002],
    ]
)
final_model
```

1001	**0**
1002	0
0	0
2001	0
2001	2001
2002	2001
2002	0
2002	2002

```
mult2 = AILS(final_model=final_model)

def psi(X, Y):
    PSI = np.zeros((len(X), 8))
    for k in range(2, len(Y)):
        PSI[k, 0] = Y[k - 1]
```

```
            PSI[k, 1] = Y[k - 2]
            PSI[k, 2] = 1
            PSI[k, 3] = X[k - 1]
            PSI[k, 4] = X[k - 1] ** 2
            PSI[k, 5] = X[k - 2] * X[k - 1]
            PSI[k, 6] = X[k - 2]
            PSI[k, 7] = X[k - 2] ** 2
        return np.delete(PSI, [0, 1], axis=0)
```

The value of theta with the lowest mean squared error obtained with the same code implemented in Scilab was:

$$W_{LS} = 0.3612343$$

and:

$$W_{SG} = 0.3548699$$

and:

$$W_{SF} = 0.3548699$$

```
PSI = psi(x_train, y_train)
w = np.array([[0.3612343], [0.2838959], [0.3548699]])
J, E, theta, HR, QR, position = mult2.estimate(
    y=y_train, X=x_train, gain=gain, y_static=Yo, X_static=Uo, weighing_matrix=w
)
result = {
    "w1": w[0, :],
    "w2": w[2, :],
    "w3": w[1, :],
    "J_ls": J[0, :],
    "J_sg": J[1, :],
    "J_sf": J[2, :],
    "||J||:": E,
}

pd.DataFrame(result)
```

w1	w2	w3	J_ls	J_sg	J_sf	$\|J\|$
0.361234	0.35487	0.283896	1.0	1.0	1.0	1.0

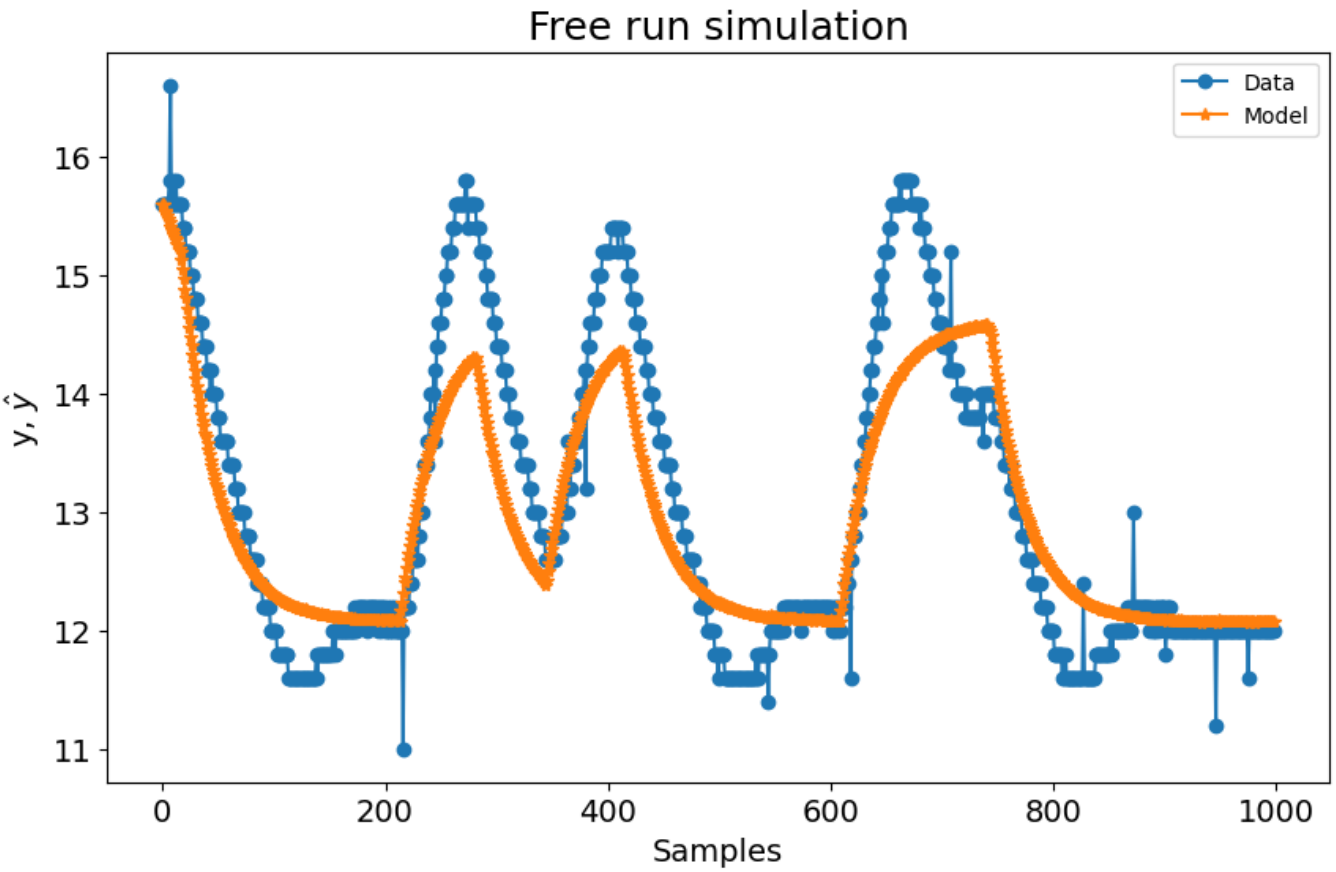
The order of the weights is different because the way we implemented in Python, but the results are very close as expected.

Dynamic results

```
model.theta = theta[position, :].reshape(-1, 1)
model.final_model = mult2.final_model
yhat = model.predict(X=x_valid, y=y_valid)

rrse = root_relative_squared_error(y_valid, yhat)
r = pd.DataFrame(
    results(
        model.final_model,
        model.theta,
        model.err,
        model.n_terms,
        err_precision=3,
        dtype="sci",
    ),
    columns=["Regressors", "Parameters", "ERR"],
)
r
```

Regressors	Parameters	ERR
1	1.4287E+00	9.999E-01
y(k-1)	5.5147E-01	2.042E-05
y(k-2)	4.0449E-01	1.108E-06
x1(k-1)	-1.2605E+01	4.688E-06
x1(k-2)	1.2257E+01	3.922E-07
x1(k-1)^2	8.3274E+00	8.389E-07
x1(k-2)x1(k-1)	-1.1416E+01	5.690E-07
x1(k-2)^2	3.0846E+00	3.827E-06

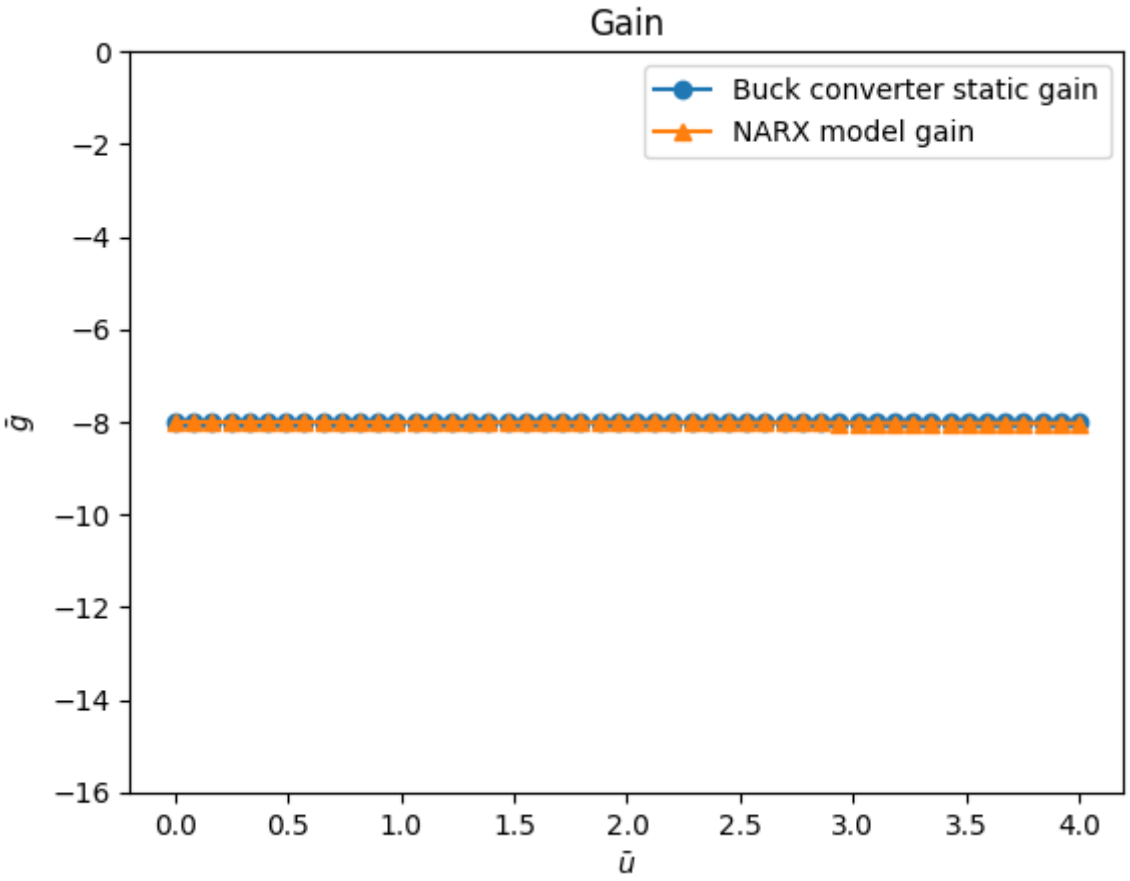
```
plot_results(y=y_valid, yhat=yhat, n=1000)
```



Static gain

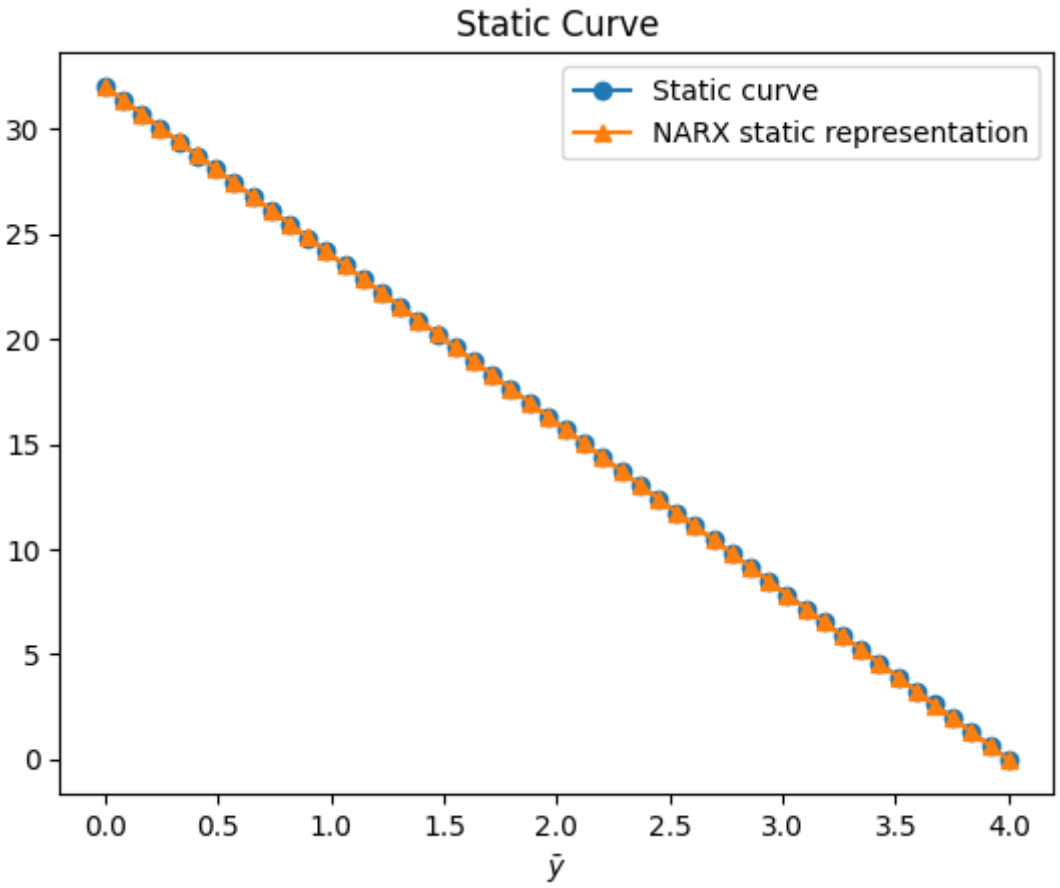
```
plt.figure(7)
plt.title("Gain")
plt.plot(
    Uo,
    gain,
    linewidth=1.5,
    linestyle="-",
    marker="o",
    label="Buck converter static gain",
)

plt.plot(
    Uo,
    HR.dot(model.theta),
    linestyle="-",
    marker="^",
    linewidth=1.5,
    label="NARX model gain",
)
plt.xlabel("$\\bar{u}$")
plt.ylabel("$\\bar{g}$")
plt.ylim(-16, 0)
```

```
plt.legend()
plt.show()
```



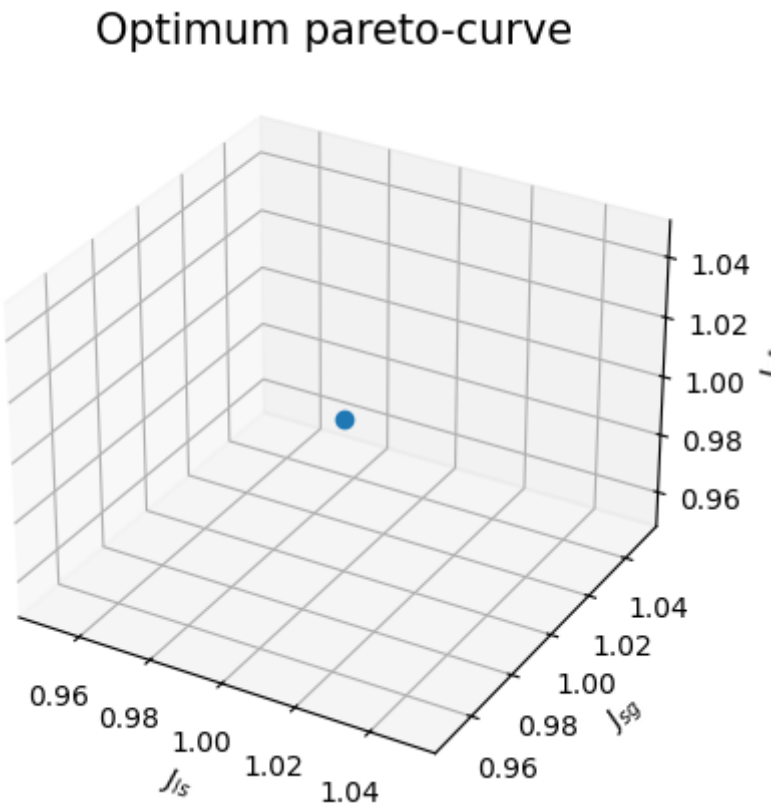
Static function

```
plt.figure(8)
plt.title("Static Curve")
plt.plot(Uo, Yo, linewidth=1.5, label="Static curve", linestyle="-", marker="o")
plt.plot(
    Uo,
    QR.dot(model.theta),
    linewidth=1.5,
    label="NARX static representation",
    linestyle="-",
    marker="^",
)

plt.xlabel("$\\bar{u}$")
plt.xlabel("$\\bar{y}$")
plt.legend()
plt.show()
```



Pareto-set solutions

```
plt.figure(9)
ax = plt.axes(projection="3d")
ax.plot3D(J[0, :], J[1, :], J[2, :], "o", linewidth=0.1)
ax.set_title("Optimum pareto-curve", fontsize=15)
ax.set_xlabel("$J_{ls}$", fontsize=10)
ax.set_ylabel("$J_{sg}$", fontsize=10)
ax.set_zlabel("$J_{sf}$", fontsize=10)
plt.show()
```



The following table show the results reported in ‘IniciacaoCientifica2007’ and the ones obtained with SysIdentPy implementation

Theta	SysIdentPy	IniciacaoCientifica2007
θ_1	0.5514725	0.549144
θ_2	0.40449005	0.408028
θ_3	1.42867821	1.45097
θ_4	-12.60548863	-12.55788
θ_5	8.32740057	8.1516315
θ_6	-11.41574116	-11.09728
θ_7	12.25729955	12.215782
θ_8	3.08461195	2.9319577

where:

E_{Scilab} =   17.426613

and:

$$E_{Python} = 17.474865$$

Note: as mentioned before, the order of the regressors in the model change, but it is the same structure. The tables shows the respective regressor parameter concerning `SysIdentPy` and

`IniciacaoCientifica2007`, but the order Θ_1, Θ_2 and so on are not the same of the ones in `model.final_model`

```
R, qit = mult2.build_linear_mapping()
print("R matrix:")
print(R)
print("qit matrix:")
print(qit)
```

$$R = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \end{bmatrix}$$

and

$$qit = \begin{bmatrix} 0 & 0 \\ 1 & 0 \\ 0 & 1 \\ 0 & 2 \end{bmatrix}$$

model's structure that will be utilized ('IniciacaoCientifica2007'):

$$y(k) = \theta_1 y(k-1) + \theta_2 y(k-2) + \theta_3 + \theta_4 u(k-1) + \theta_5 u(k-1)^2 + \theta_6 u(k-2)u(k-1) + \theta_7 u(k-2) + \theta_8 u(k-2$$

$$q_i = \begin{bmatrix} 0 & 0 \\ 1 & 0 \\ 2 & 0 \\ 2 & 2 \end{bmatrix} = \begin{bmatrix} 1 \\ \overline{y} \\ \overline{u} \\ \overline{u^2} \end{bmatrix}$$

Biobjective optimization

An use case applied to Buck converter CC-CC using as objectives the static curve information and the prediction error (dynamic)

```
bi_objective = AILS(
    static_function=True, static_gain=False, final_model=final_model, normalize=True
)
```

the value of theta with the lowest mean squared error obtained through the routine in Scilab was:

$$W_{LS} = 0.9931126$$

and:

$$W_{SF} = 0.0068874$$

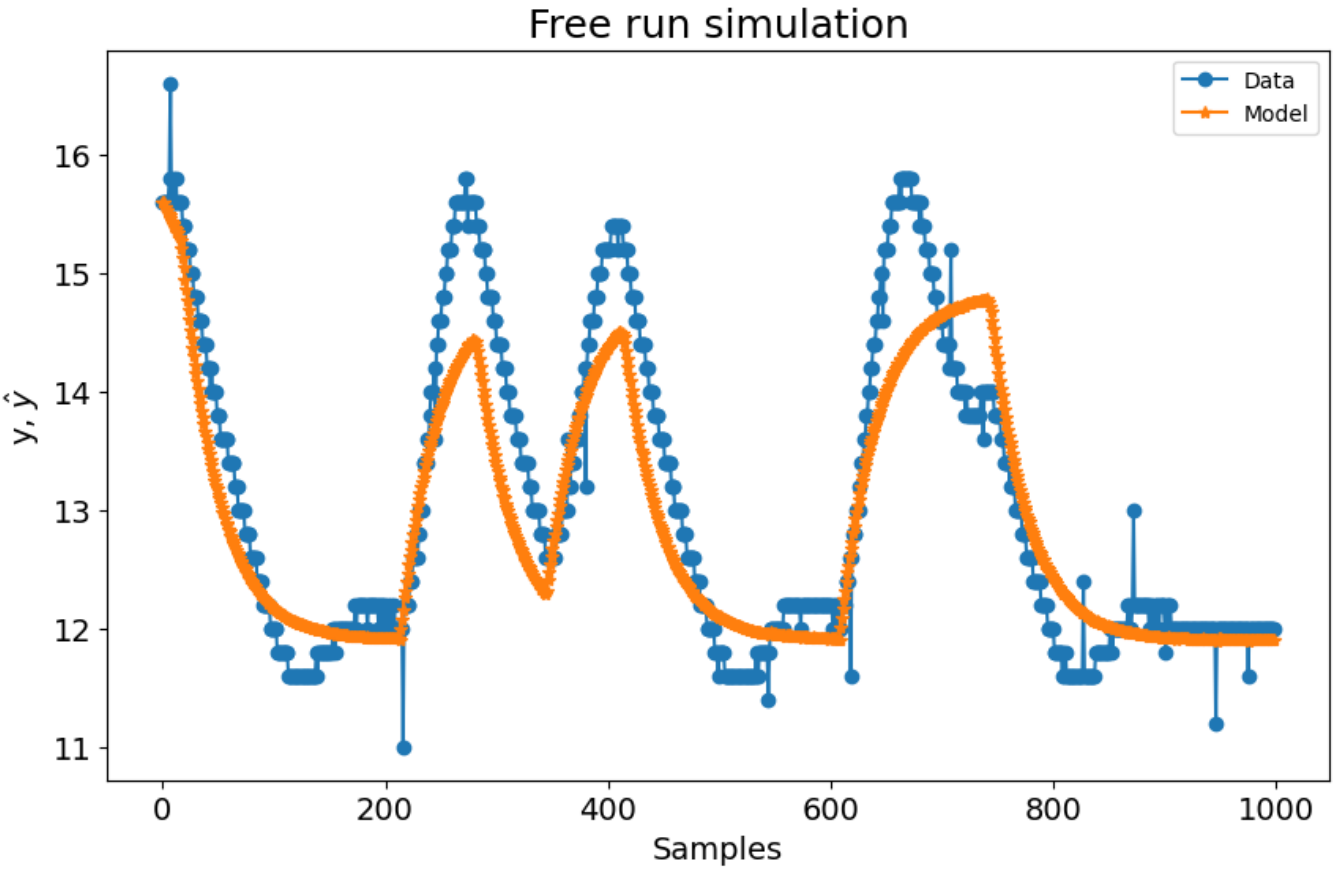
```
w = np.zeros((2, 2000))
w[0, :] = np.logspace(-0.01, -6, num=2000, base=2.71)
w[1, :] = np.ones(2000) - w[0, :]
J, E, theta, HR, QR, position = bi_objective.estimate(
    y=y_train, X=x_train, y_static=Yo, X_static=Uo, weighing_matrix=w
)

result = {"w1": w[0, :], "w2": w[1, :], "J_ls": J[0, :], "J_sg": J[1, :], "||J||:":
E}

pd.DataFrame(result)
```

w1	w2	J_ls	J_sg	$\|\|J\|\|$
0.990080	0.009920	0.990863	1.000000	0.990939
0.987127	0.012873	0.990865	0.987032	0.990939
0.984182	0.015818	0.990867	0.974307	0.990939
0.981247	0.018753	0.990870	0.961803	0.990940
0.978320	0.021680	0.990873	0.949509	0.990941
...	...	...	...	...
0.002555	0.997445	0.999993	0.000072	0.999993
0.002547	0.997453	0.999994	0.000072	0.999994
0.002540	0.997460	0.999996	0.000071	0.999996
0.002532	0.997468	0.999998	0.000071	0.999998
0.002525	0.997475	1.000000	0.000070	1.000000

```
model.theta = theta[position, :].reshape(-1, 1)
model.final_model = bi_objective.final_model
yhat = model.predict(X=x_valid, y=y_valid)
rrse = root_relative_squared_error(y_valid, yhat)
r = pd.DataFrame(
    results(
        model.final_model,
        model.theta,
        model.err,
        model.n_terms,
        err_precision=3,
        dtype="sci",
    ),
    columns=["Regressors", "Parameters", "ERR"],
)

r
```

	Regressors	Parameters	ERR
0	1	1.3873E+00	9.999E-01
1	y(k-1)	5.4941E-01	2.042E-05
2	y(k-2)	4.0804E-01	1.108E-06
3	x1(k-1)	-1.2515E+01	4.688E-06
4	x1(k-2)	1.2227E+01	3.922E-07
5	x1(k-1)^2	8.1171E+00	8.389E-07
6	x1(k-2)x1(k-1)	-1.1047E+01	5.690E-07
7	x1(k-2)^2	2.9043E+00	3.827E-06

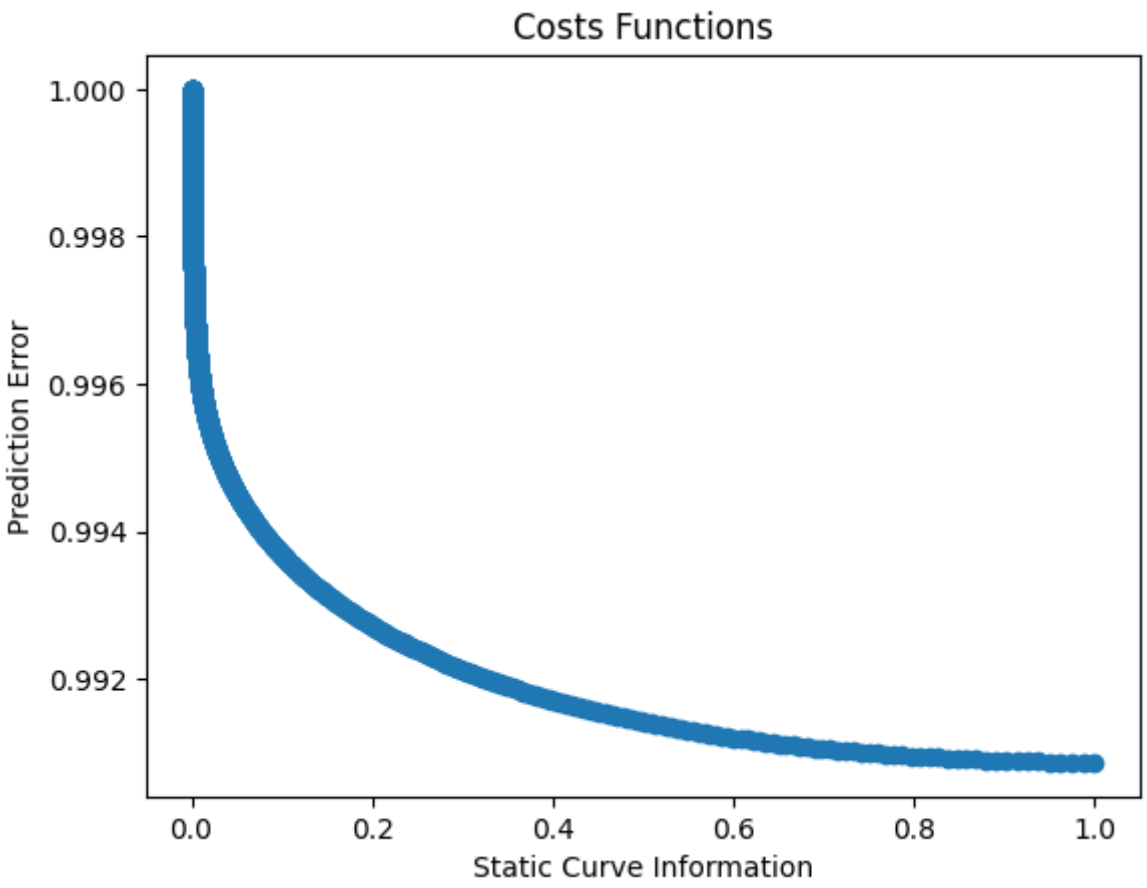
```
plot_results(y=y_valid, yhat=yhat, n=1000)
```




```
plt.figure(10)
plt.title("Static Curve")
plt.plot(Uo, Yo, linewidth=1.5, label="Static curve", linestyle="-", marker="o")
plt.plot(
    Uo,
    QR.dot(model.theta),
    linewidth=1.5,
    label="NARX static representation",
```

```
    linestyle="-",
    marker="^",
)

plt.xlabel("$\\bar{u}$")
plt.xlabel("$\\bar{y}$")
plt.legend()
plt.show()
```

```
plt.figure(11)
plt.title("Costs Functions")
plt.plot(J[1, :], J[0, :], "o")
plt.xlabel("Static Curve Information")
plt.ylabel("Prediction Error")
plt.show()
```



where the best estimated Θ is

Theta	SysIdentPy	IniciacaoCientifica2007
θ_1	0.54940883	0.5494135
θ_2	0.40803995	0.4080312
θ_3	1.38725684	3.3857601

Theta	SysIdentPy	IniciacaoCientifica2007
θ_4	-12.51466378	-12.513688
θ_5	8.11712897	8.116575
θ_6	-11.04664789	-11.04592
θ_7	12.22693907	12.227184
θ_8	2.90425844	2.9038468

where:

$$E_{Scilab} = 17.408934$$

and:

$$E_{Python} = 17.408947$$

Multiobjective parameter estimation

Use case considering 2 different objectives: the prediction error and the static gain

```
bi_objective_gain = AILS(
    static_function=False, static_gain=True, final_model=final_model,
normalize=False
)
```

the value of theta with the lowest mean squared error obtained through the routine in Scilab was:

$$W_{LS} = 0.9931126$$

and:

$$W_{SF} = 0.0068874$$

```
w = np.zeros((2, 2000))
w[0, :] = np.logspace(0, -6, num=2000, base=2.71)
w[1, :] = np.ones(2000) - w[0, :]
J, E, theta, HR, QR, position = bi_objective_gain.estimate(
    X=x_train, y=y_train, gain=gain, y_static=Yo, X_static=Uo, weighing_matrix=w
)

result = {"w1": w[0, :], "w2": w[1, :], "J_ls": J[0, :], "J_sg": J[1, :], "||J||:":
E}

pd.DataFrame(result)
```

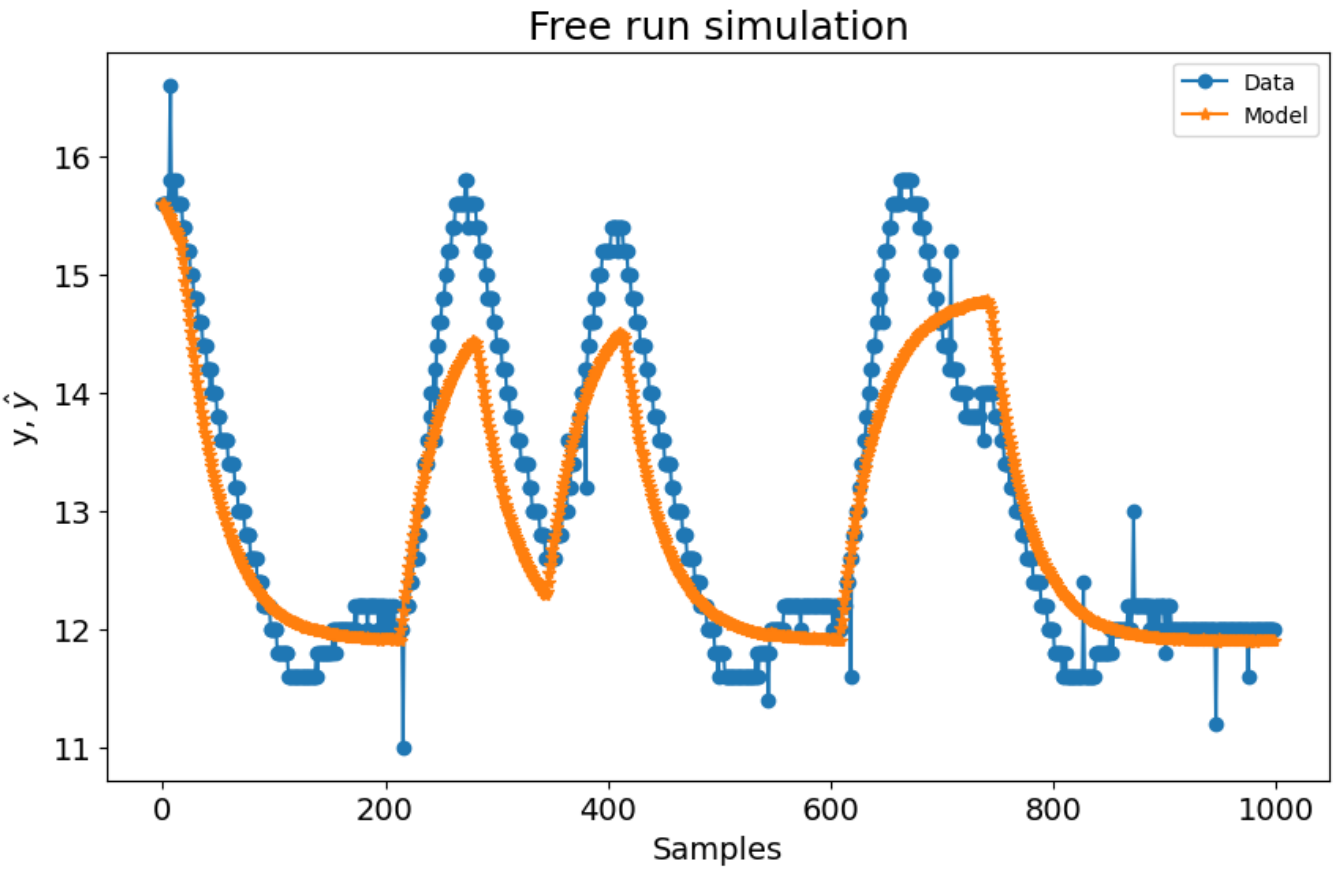
w1	w2	J_ls	J_sg	$\|\|J\|\|$
1.000000	0.000000	17.407256	3.579461e+01	39.802849
0.997012	0.002988	17.407528	2.109260e-01	17.408806
0.994033	0.005967	17.407540	2.082067e-01	17.408785
0.991063	0.008937	17.407559	2.056636e-01	17.408774
0.988102	0.011898	17.407585	2.031788e-01	17.408771
...	...	...	...	...
0.002555	0.997445	17.511596	3.340081e-07	17.511596
0.002547	0.997453	17.511596	3.320125e-07	17.511596
0.002540	0.997460	17.511597	3.300289e-07	17.511597
0.002532	0.997468	17.511598	3.280571e-07	17.511598
0.002525	0.997475	17.511599	3.260972e-07	17.511599

```
# Writing the results
model.theta = theta[position, :].reshape(-1, 1)
model.final_model = bi_objective_gain.final_model
yhat = model.predict(X=x_valid, y=y_valid)
rrse = root_relative_squared_error(y_valid, yhat)
r = pd.DataFrame(
    results(
        model.final_model,
        model.theta,
        model.err,
        model.n_terms,
        err_precision=3,
        dtype="sci",
    ),
    columns=["Regressors", "Parameters", "ERR"],
)

r
```

	Regressors	Parameters	ERR
0	1	1.4853E+00	9.999E-01
1	y(k-1)	5.4940E-01	2.042E-05
2	y(k-2)	4.0806E-01	1.108E-06
3	x1(k-1)	-1.2581E+01	4.688E-06
4	x1(k-2)	1.2210E+01	3.922E-07
5	x1(k-1)^2	8.1686E+00	8.389E-07
6	x1(k-2)x1(k-1)	-1.1122E+01	5.690E-07

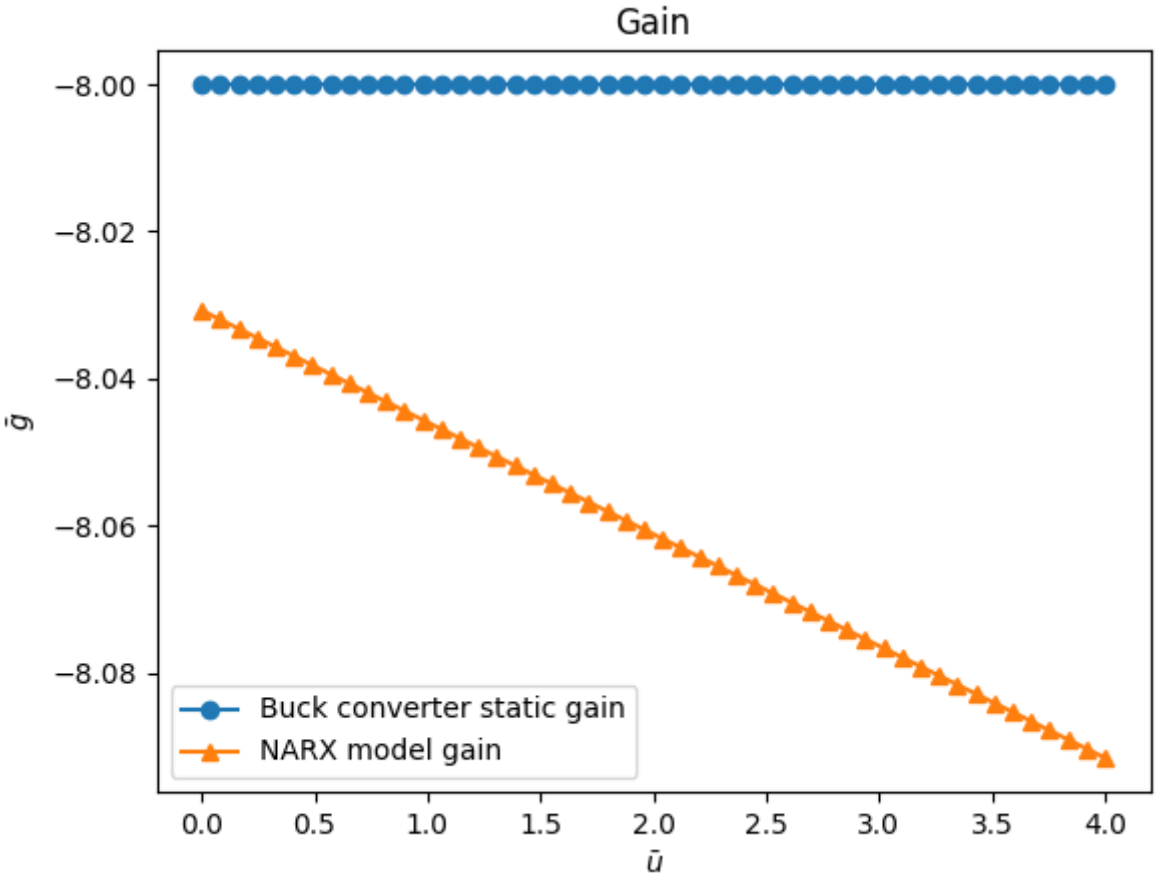
	Regressors	Parameters	ERR
7	x1(k-2)^2	2.9455E+00	3.827E-06

```
plot_results(y=y_valid, yhat=yhat, n=1000)
```

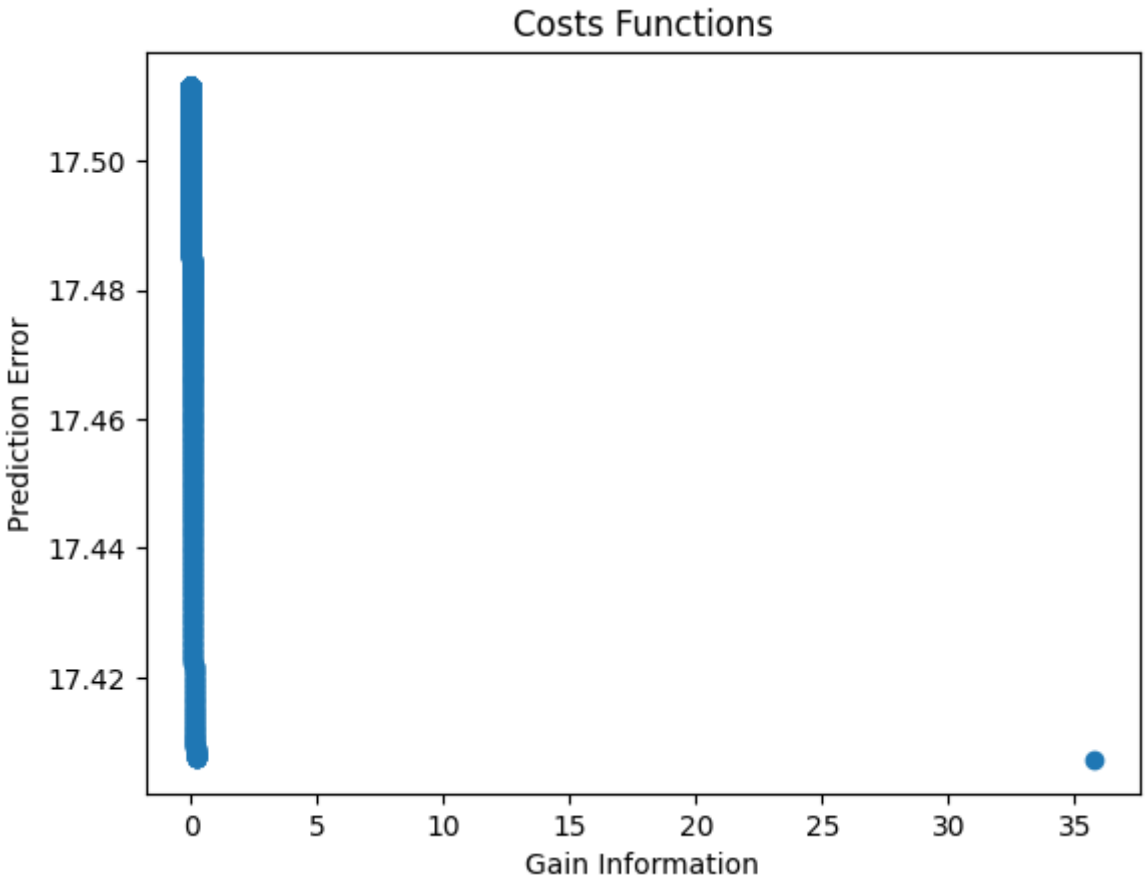



```
plt.figure(12)
plt.title("Gain")
plt.plot(
    Uo,
    gain,
    linewidth=1.5,
    linestyle="-",
    marker="o",
    label="Buck converter static gain",
)

plt.plot(
    Uo,
    HR.dot(model.theta),
    linestyle="-",
    marker="^",
    linewidth=1.5,
    label="NARX model gain",
```

```
)
plt.xlabel("$\\bar{u}$")
plt.ylabel("$\\bar{g}$")
plt.legend()
plt.show()
```




```
plt.figure(11)
plt.title("Costs Functions")
plt.plot(J[1, :], J[0, :], "o")
plt.xlabel("Gain Information")
plt.ylabel("Prediction Error")
plt.show()
```



being the selected θ:

Theta	SysIdentPy	IniciacaoCientifica2007
θ_1	0.54939785	0.54937289
θ_2	0.40805603	0.40810168
θ_3	1.48525190	1.48663719
θ_4	-12.58066084	-12.58127183
θ_5	8.16862622	8.16780294
θ_6	-11.12171897	-11.11998621
θ_7	12.20954849	12.20927355
θ_8	2.94548501	2.9446532

where:

E_{Scilab} = 17.408997

and:

$$E_{Python} = 17.408781$$

Additional Information

You can also access the matrix Q and H using the following methods

Matrix Q:

```
bi_objective_gain.build_static_function_information(Uo, Yo)[1]
```

Matrix H+R:

```
bi_objective_gain.build_static_gain_information(Uo, Yo, gain)[1]
```

6 - Multiobjective Model Structure Selection

Coming soon

7 - NARX Neural Network

Coming soon

8 - Severely Nonlinear System Identification

We have categorized systems into two different classes for now: **linear systems** and **nonlinear** systems. As mentioned, **linear systems** has been extensively studied with several different well-established methods available, while **nonlinear** systems is a very active field with several problems that are still open for research. Besides linear and nonlinear systems, there are the ones called **Severely Nonlinear Systems**. Severely Nonlinear Systems are the ones that exhibit highly complex and exotic dynamic behaviors like sub-harmonics, chaotic behavior and hysteresis. For now, we will focus on system with hysteresis.

Modeling Hysteresis With Polynomial NARX Model

Hysteresis nonlinearity is a severely nonlinear behavior commonly found in electromagnetic devices, sensors, semiconductors, intelligent materials, and many more, which have memory effects between quasi-static input and output ([Visintin, A.](Differential Models of Hysteresis), [Ahmad, I.](Two Degree-of-Freedom Robust Digital Controller Design With Bouc-Wen Hysteresis Compensator for Piezoelectric Positioning Stage)). A hysteretic system is one that exhibits a path-dependent behavior, meaning its response depends not only on its current state but also on its history. In a hysteretic system, when you apply an input, the system's response (like displacement or stress) doesn't follow the same path to the starting point when you remove the input. Instead, it forms a loop-like pattern called a hysteresis loop. This is because the system have the *ability* to preserve a deformation caused by an input, characterizing a memory effect.

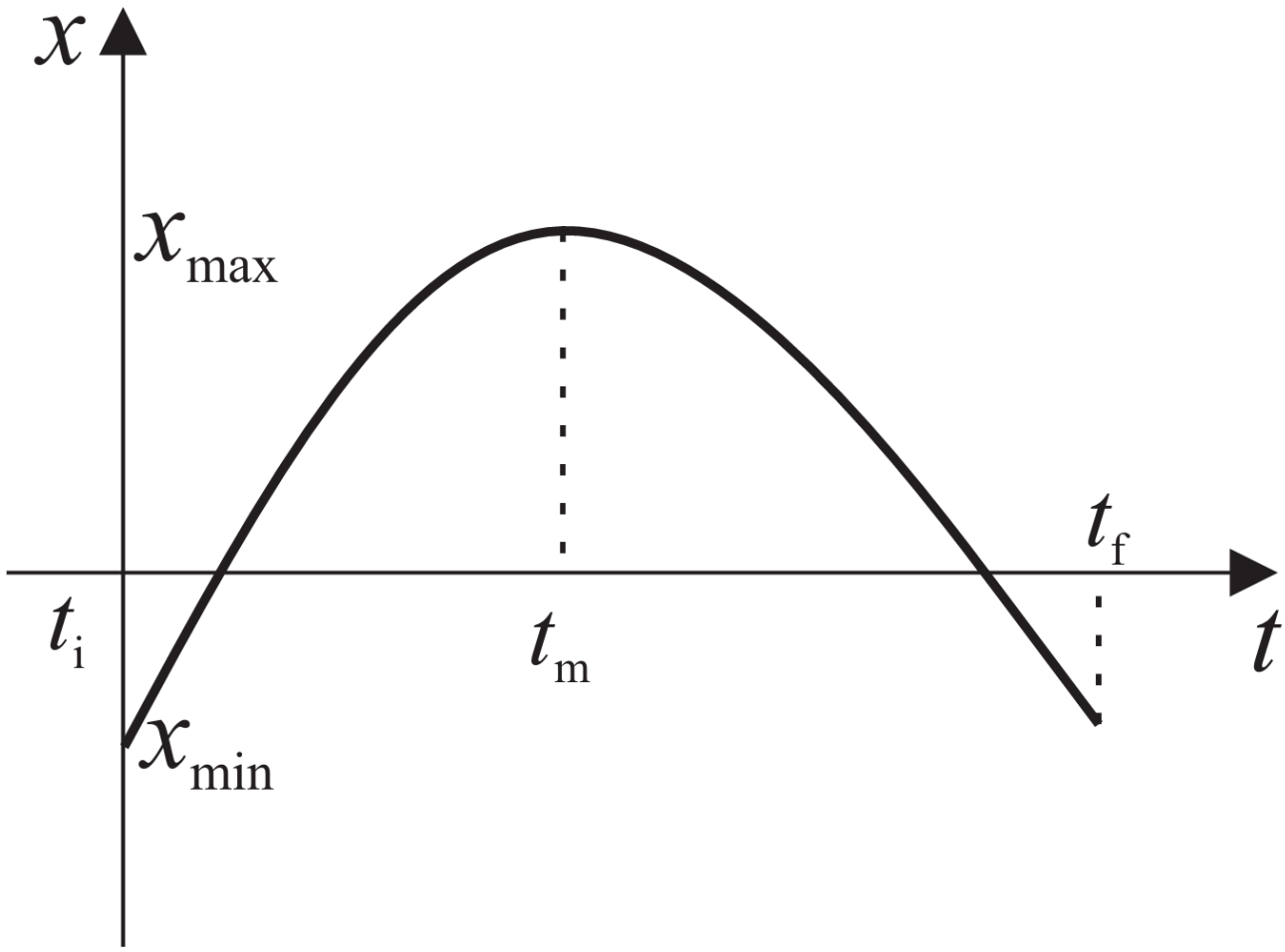
The identification of hysteretic systems using polynomial NARX models is typically an intriguing task because the traditional Model Structure Selection algorithms do not work properly ([Martins, S. A. M. and Aguirre, L. A.](Sufficient conditions for rate-independent hysteresis in autoregressive identified models), [Leva, A. and Piroddi, L.](NARX-based technique for the modelling of magneto-rheological damping devices)). Martins, S. A. M. and Aguirre, L. A. presented the sufficient conditions to describe hysteresis using polynomial models by providing the concept of bounding structure $\mathcal{H}$. Polynomial NARX models with a single equilibrium can be used in a full characterization of the hysteresis behavior adopting the bounding structure concept.

The following are some of the essential concepts and formal definitions for understanding how NARX model can be used to describe systems with hysteresis.

Continuous-time loading-unloading quasi-static signal

One important characteristic to model hysteretic systems is the input signal. A loading-unloading quasi-static signal is a periodic continuous time signal x_t with period $T = (t_f - t_i)$ and frequency $\omega = 2\pi f$ where x_t increases monotonically from x_{min} to x_{max}, considering $t_i \leq t \leq t_m$ (loading) and

decreases monotonically from x_{max} to x_{min}, considering $t_m \leq t \leq t_f$ (unloading). If the loading-unloading signal changes with $\omega \to 0$, the signal is also called a quasi-static signal. Visually, this is much more simple to understand. The following image shows a continuous-time loading-unloading quasi-static signal.



> Figure 1. Continuous-time loading-unloading quasi-static signal, demonstrating the periodic increase and decrease of the input signal.

In this respect, Martins, S. A. M. and Aguirre, L. A. also presented the idea of transforming the inputs of the system using multi-valued functions.

> Multi-valued functions - Let $\phi(\Delta x_k) : \mathbb{R} \to \mathbb{R}$. If~$\Delta x_k = x_k - x_{k-1}$, $\phi(\Delta x_k)$ is a multi-valued function if:

$$\phi(\Delta x_k) = \begin{cases} \phi_1, & if\ \Delta x_k > \epsilon; \\ \phi_2, & if\ \Delta x_k < \epsilon; \\ \phi_3, & if\ \Delta x_k = \epsilon; \end{cases} \tag{1}$$

where $\epsilon \in \mathbb{R}$, $\phi_1 \neq \phi_2 \neq \phi_3$. For some inputs $\Delta x_k \neq \epsilon$, $\forall k \in \mathbb{N}$, and the last value in equation above is not used.

A frequently used multi-valued function is the $\text{sign}(\cdot) : \mathbb{R} \to \mathbb{R}$:

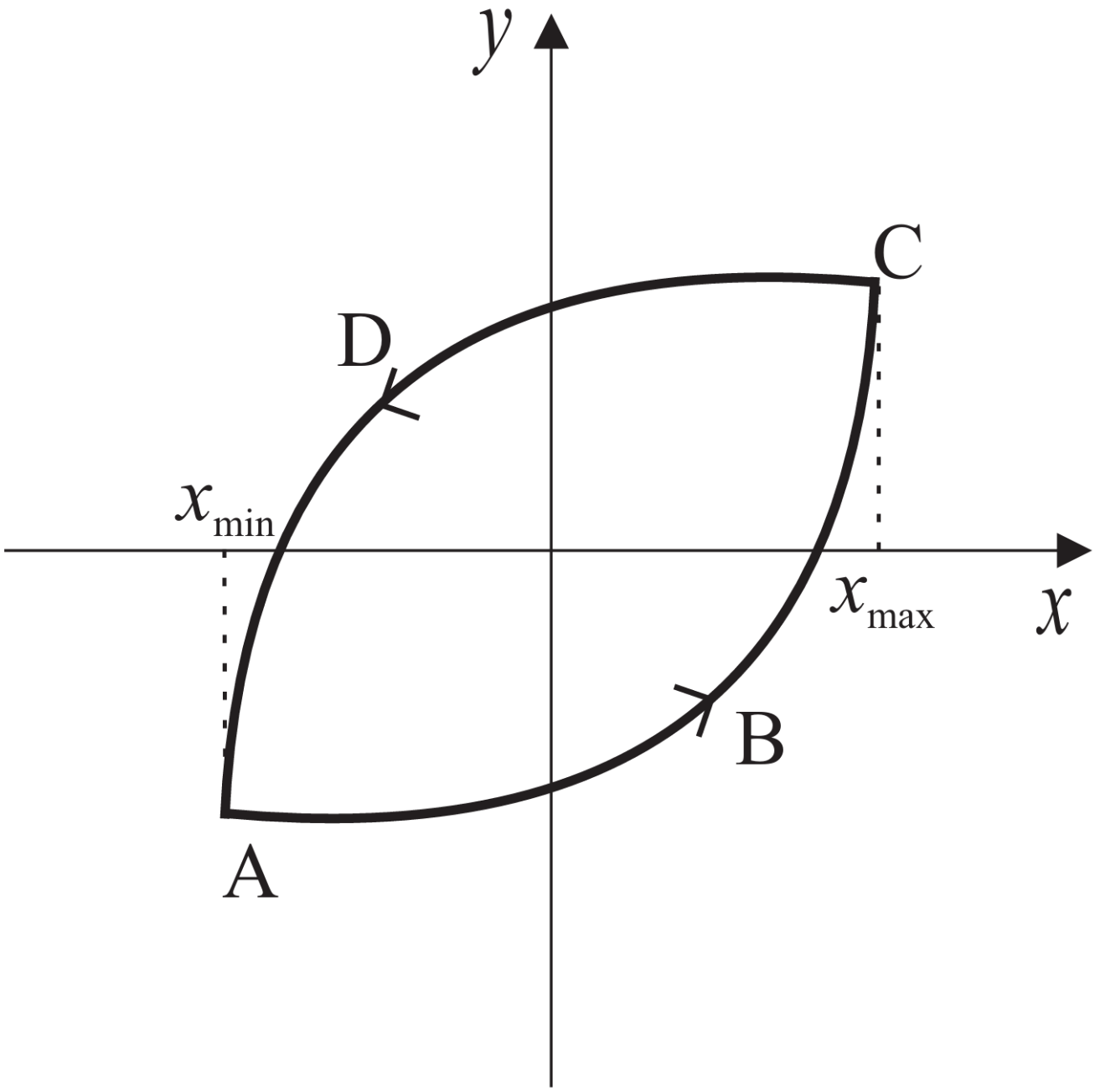
$$sign(x) = \begin{cases} 1, & if\ x > 0; \\ -1, & if\ x < 0; \\ 0, & if\ x = 0. \end{cases} \tag{2}$$

Hysteresis loops in continuous time $\mathcal{H}_t(\omega)$

Let x_t be a continuous-time loading-unloading quasi-static signal applied to a continuous-time system and y_t is the system output. $\mathcal{H}_t(\omega)$ denotes a closed loop in the $x_t - y_t$ plane, which shape depend on ω. If the system presents hysteretic nonlinearity, $\mathcal{H}_t(\omega)$ is denoted as:

$$\mathcal{H}_t(\omega) = \begin{cases} \mathcal{H}_t(\omega)^+, \ for\ t_i \ \leq \ t \ \leq \ t_m, \\ \mathcal{H}_t(\omega)^-; \ for\ t_m \ \leq \ t \ \leq \ t_f, \end{cases} \tag{3}$$

where $\mathcal{H}_t(\omega)^+ \neq \mathcal{H}_t(\omega)^-$, $\forall t \neq t_m$. $t_i \leq t \leq t_m$ and~$t_m \leq t \leq t_f$ correspond to the regime when x_t is loading and unloading, respectively. $\mathcal{H}_t(\omega)^+$ corresponds to the part of the loop formed in the $x_t - y_t$ plane, while $t_i \leq t \leq t_m$ (when x_t is loading) whereas $\mathcal{H}_t(\omega)^-$ is the part of the loop formed in the~$x_t - y_t$ plane for~$t_m \leq t \leq t_f$ (when x_t is unloading), as shown in the Figure 2:



> Figure 2. Example of a hysteresis curve.

> Rate Independent Hysteresis (RIH) ([Visintin, A.](Differential Models of Hysteresis))- The hysteresis behavior is called to be rate independent if the path $ABCD$, which depends on pair $x(t), y(t)$, is invariant with respect to any increasing diffeomorphism~$\varphi : [0,T] \to [0,T]$, i.e.:

$$F(u\ o\ \varphi, y^0) = F(u, y^0)\ o\ \varphi\ em\ [0,T]. \tag{4}$$

This means that at any instant t, $y(t)$ depends only on $u : [0,T] \to \mathbb{R}$ and on the order in which values have been attained before t. In other words, the memory effect is not affected by the frequency of the input.

Rate Independent Hysteresis in polynomial NARX model

Martins, S. A. M. and Aguirre, L. A. presented the sufficient conditions for NARX model to represent hysteresis. One of the developed concepts in the Bounding Structure $\mathcal{H}$.

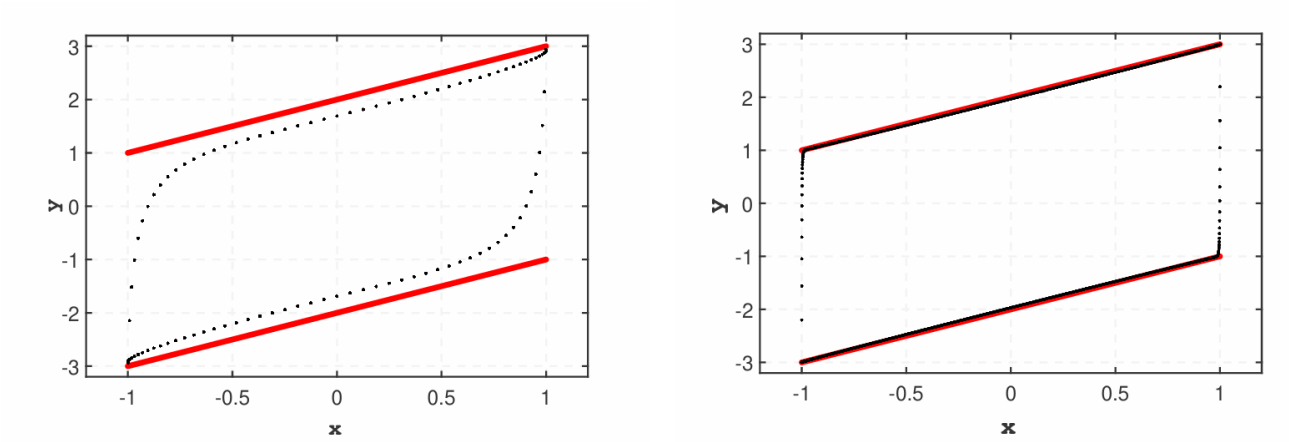
> Bounding Structure $\mathcal{H}$ (Martins, S. A. M. and Aguirre, L. A.) - Let $\mathcal{H}_t(\omega)$ be the system hysteresis. $\mathcal{H} = \lim_{\omega \to 0} \mathcal{H}_t(\omega)$ is defined as the bounding structure that delimits $\mathcal{H}_t(\omega)$.

Now, consider a polynomial NARX excited by a loading-unloading quasi-static signal. If the model has one real and stable equilibrium point whose location depends on input and loading/unloading regime, the polynomial will exhibit a Rate Independent Hysteresis loop $\mathcal{H}_t(\omega)$ in the $x - y$ plane.

Here is an example. Let $y_k = 0.8y_{k-1} + 0.4\phi_{k-1} + 0.2x_{k-1}$, where $\phi_k = \text{sign}(\Delta(\text{x}_\text{k}))$ and $x_k = sin(\omega k)$ and ω is the frequency of the input signal x. The equilibria of this model is given by:

$$\overline{y}(\overline{\phi}, \overline{x}) = \begin{cases} \frac{0.6+0.2\overline{x}}{1-0.8} = 3 + \overline{x}\,, & for\ loading; \\ \frac{-0.6+0.2\overline{x}}{1-0.8} = -3 + \overline{x}\,, & for\ unloading; \end{cases} \tag{5}$$

where $\overline{x}$ is a loading-unloading quasi-static input signal. Since the equilibrium points are asymptotically stable, the output converges to $\mathcal{H}_k(w)$ in the $x - y$ plane. Note that for a constant input value $x = 1 = \overline{x}$, the equilibrium lies in $\overline{y} = 3$ for loading regime and $\overline{y} = -1$ for unloading regime. Analogously, for $\overline{x} = -1$, the equilibrium lies in $\overline{y} = 1$ for loading regime and $\overline{y} = -3$ for unloading regime, as shown in the figure below:



(1) $\omega = 1$. (2) $\omega = 0.1$.

> Figure 3. Example of a bounding structure $\mathcal{H}$. The black dots are on $\mathcal{H}_k(\omega)$ for model $y_k = 0.8y_{k-1} + 0.4\phi_{k-1} + 0.2x_{k-1}$. The bounding structure $\mathcal{H}$, in red, confines $\mathcal{H}_k(\omega)$.}

As can be observed in the Figure 3, in we guarantee the sufficient conditions proposed by Martins, S. A. M. and Aguirre, L. A., a NARX model can reproduce a histeretic behavior. Chapter 10 presents a case study of a system with hysteresis.

The following code can be used to reproduce the behavior shown in Figure 3. Change `w` from 1 to 0.1 to see how the bounded structure $\mathcal{H}$ converge to the equilibria of the system.

```python
import numpy as np
import matplotlib.pyplot as plt

# Parameters
w = 1
t = np.arange(0, 60.1, 0.1)
y = np.zeros(len(t))
x = np.sin(w * t)

# Initialize y and fi
fi = np.zeros(len(t))
# Iterate over the time array to calculate y
for k in range(1, len(t)):
    fi[k] = np.sign(x[k] - x[k-1])
    y[k] = 0.8 * y[k-1] + 0.2 * x[k-1] + 0.4 * fi[k-1]

plt.plot(x, y)
plt.xlabel('x')
plt.ylabel('y')
plt.title('Example')
plt.show()
```

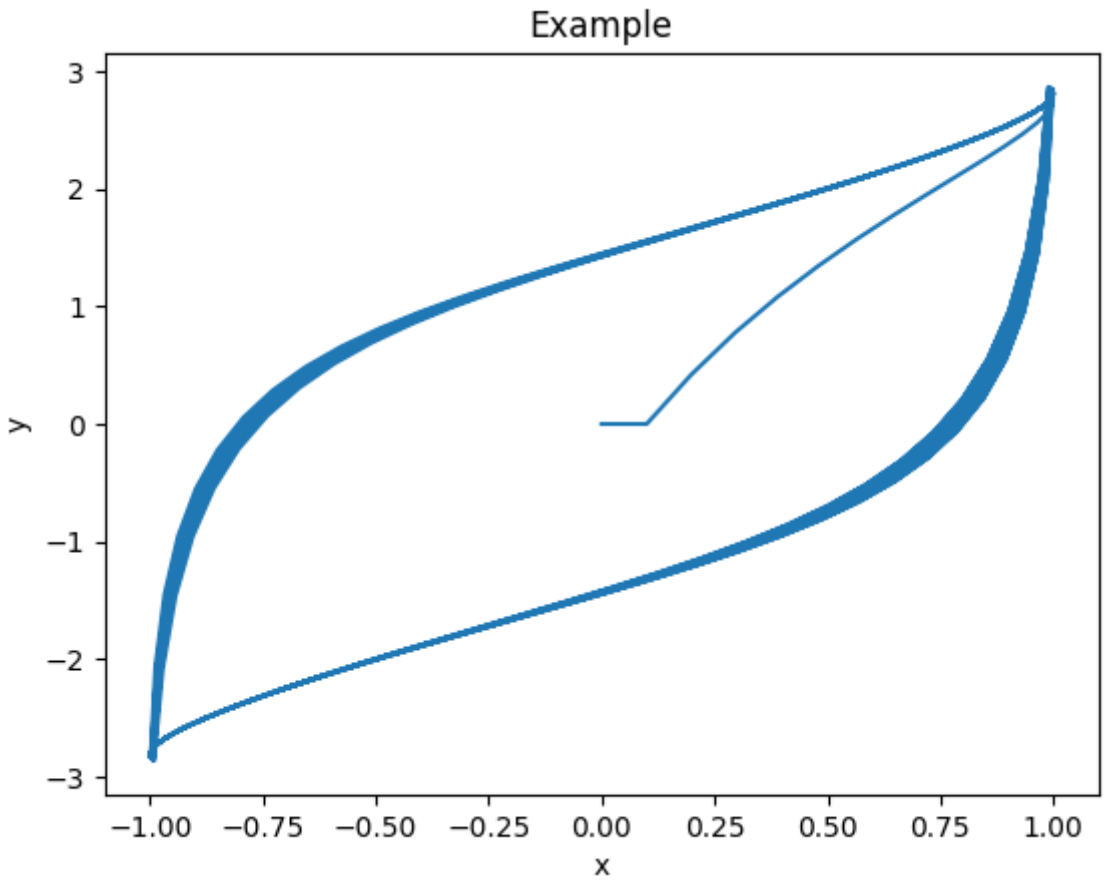


Figure 4. Reproduction of a bounding structure $\mathcal{H}$ using python.

9 - Validation

The `predict` Method in SysIdentPy

Before getting into the validation process in System Identification, it's essential to understand how the `predict` method works in SysIdentPy.

Using the `predict` Method

A typical usage of the `predict` method in SysIdentPy looks like this:

```
yhat = model.predict(X=x_test, y=y_test)
```

SysIdentPy users often have two common questions about this method:

1. Why do we need to pass the test data, `y_test`, as an argument in the `predict` method?
2. Why are the initial predicted values identical to the values in the test data?

To address these questions, let's first explain the concepts of infinity-step-ahead prediction, n-step-ahead prediction, and one-step-ahead prediction in dynamic systems.

Infinity-Step-Ahead Prediction

Infinity-step-ahead prediction, also known as *free run simulation*, refers to making predictions using previously **predicted** values, $\hat{y}_{k-n_y}$, in the prediction loop.

For example, consider the following test input and output data:

$$x_{test} = [1, 2, 3, 4, 5, 6, 7]$$

$$y_{test} = [8, 9, 10, 11, 12, 13, 14]$$

Suppose we want to validate a model m defined by:

$$m \rightarrow y_k = 1 * y_{k-1} + 2 * x_{k-1}$$

To predict the first value, we need access to both y_{k-1} and x_{k-1}. This requirement explains why you need to pass `y_test` as an argument in the `predict` method. It also answers the second question: SysIdentPy requires the user to provide the initial conditions explicitly. The `y_test` data passed in the `predict` method is not used entirely; only the initial values needed for the model's lag structure are used.

In this example, the model's maximum lag is 1, so we need only 1 initial condition. The predicted values, `yhat`, are then calculated as follows:

```
y_initial = yhat(0) = 8
yhat(1) = 1*8 + 2*1 = 10
yhat(2) = 1*10 + 2*2 = 14
yhat(3) = 1*14 + 2*3 = 20
yhat(4) = 1*20 + 2*4 = 28
```

As shown, the first value of `yhat` matches the first value of `y_test` because it serves as the initial condition. Another key point is that the prediction loop uses the previously **predicted** values, not the actual `y_test` values, which is why it's called infinity-step-ahead or free run simulation.

In system identification, we often aim for models that perform well in infinity-step-ahead predictions. Since the prediction error propagates over time, a model that shows good performance in free run simulation is considered a robust model.

In SysIdentPy, users only need to pass the initial conditions when performing an infinity-step-ahead prediction. If you pass only the initial conditions, the results will be the same! Therefore

```
yhat = model.predict(X=x_test, y=y_test)
```

is actually the same as

```
yhat = model.predict(X=x_test, y=y_test[:model.max_lag].reshape(-1, 1))
```

> `model.max_lag` can be accessed after we fit the model using the code below.

```
model = FROLS(
        order_selection=False,
    ylag=2,
    xlag=2,
    estimator=LeastSquares(unbiased=False),
    basis_function=basis_function,
    e_tol=0.9999
    n_terms=15
)
model.fit(X=x, y=y)
model.max_lag
```

> Its important to mention that, in current version of SysIdentPy, the maximum lag considered is actually the maximum lag between `xlag` and `ylag` definition. This is important because you can pass `ylag = xlag = 10` and the final model, after the model structure selection, select terms where the maximum lag is 3. You have to pass 10 initial conditions, but internally the calculations

> are done using the correct regressors. This is necessary due the way the regressors are created after that the model is fitted. Therefore is recommend to use the `model.max_lag` to be sure.

1-step ahead prediction

The difference between 1 step-ahead prediction and infinity-steps ahead prediction is that the model take the previous real `y_test` test values in the loop instead of the predicted `yhat` values. And that is a huge and important difference. Lets do prediction using 1-step ahead method:

```
y_initial = yhat(0) = 8
yhat(1) = 1*8 + 2*1 = 10
yhat(2) = 1*9 + 2*2 = 13
yhat(3) = 1*10 + 2*3 = 16
yhat(4) = 1*11 + 2*4 = 19
and so on
```

The model uses real values in the loop and only predict the next value. The prediction error, in this case, is always corrected because we are not propagating the error using the predicted values in the loop.

SysIdentPy's `predict` method allow the user to perform a 1-step ahead prediction by setting `steps_ahead=1`

```
yhat = model.predict(X=x_test, y=y_test, steps_ahead=1)
```

In this case, as you can imagine, we need to pass the all the `y_test` data because the method have to access the real values at each iteration. If you pass only the initial conditions, `yhat` will have only the initial conditions plus 1 more sample, that is the 1-step ahead prediction. To predict another point, you would need to pass the new initial conditions again and so on. SysIdentPy already to everything for you, so just pass all the data you want to validate using the 1-step ahead method.

n-steps ahead prediction

The n-steps ahead prediction is almost the same as the 1-step ahead, but here you can define the number of steps ahead you want to test your model. If you set `steps_ahead=5`, for example, it means that the first 5 values will be predicted using `yhat` in the loop, but then the process is *restarted* by feeding the real values in `y_test` in the next iteration, then performing other 5 predictions using the `yhat` and so on. Lets check the example considering `steps_ahead=2`:

```
y_initial = yhat(0) = 8
yhat(1) = 1*8 + 2*1 = 10
yhat(2) = 1*10 + 2*2 = 14
yhat(3) = 1*10 + 2*3 = 16
```

```
yhat(4) = 1*16 + 2*4 = 24
and so on
```

Model Performance

Model validation is one of the most crucial part in system identification. As we mentioned before, in system identification we are trying the model the dynamic of the process without for task like control design. In such cases, we can not only rely on regression metrics, but also ensuring that residuals are unpredictable across various combinations of past inputs and outputs ([Billings, S. A. and Voon, W. S. F.](Structure detection and model validity tests in the identification of nonlinear systems)). One often used statistical tests is the normalized RMSE, called RRSE, which can be expressed by

$$\mathrm{RRSE} = \frac{\sqrt{\sum_{k=1}^{n}(y_k - \hat{y}_k)^2}}{\sqrt{\sum_{k=1}^{n}(y_k - \bar{y})^2}}, \tag{1}$$

where $\hat{y}_k \in \mathbb{R}$ the model predicted output and $\bar{y} \in \mathbb{R}$ the mean of the measured output y_k. The RRSE gives some indication regarding the quality of the model, but concluding about the best model by evaluating only this quantity may leads to an incorrect interpretation, as shown in following example.

Consider the models

$$y_{a}k = 0.7077y_{a}k-1 + 0.1642u_{k-1} + 0.1280u_{k-2}$$

and $y_{b}k = 0.7103y_{b}k-1 + 0.1458u_{k-1} + 0.1631u_{k-2} - 1467y^{3}_{b}k-1 + 0.0710y^{3}_{b}k-2 + 0.0554y^{2}_{b}k-3u_{k-3}$ defined in the Meta Model Structure Selection: An Algorithm For Building Polynomial NARX Models For Regression And Classification. The former results in a $RRSE = 0.1202$ while the latter gives $RRSE = 0.0857$. Although the model $y_{b}k$ fits the data better, it is only a biased representation to one piece of data and not a good description of the entire system.

The RRSE (or any other metric) shows that validations test might be performed carefully. Another traditional practice is split the data set in two parts. In this respect, one can test the models obtained from the estimation part of the data using an specific data for validation. However, the one-step-ahead performance of NARX models generally results in misleading interpretations because even strongly biased models may fit the data well. Therefore, a free run simulation approach usually allows a better interpretation if the model is adequate or not (Billings, S. A.).

Statistical tests for SISO models based on the correlation functions were proposed in [Billings, S. A. and Voon, W. S. F.](A prediction-error and stepwise-regression estimation algorithm for non-linear systems), Model validity tests for non-linear signal processing applications. The tests are:

$$
\begin{aligned}
\phi_{\xi\xi\tau} &= E\{\xi_k \xi_{k-\tau}\} = \delta_\tau, \\
\phi_{\xi x\tau} &= E\{\xi_k x_{k-\tau}\} = 0 \forall \tau, \\
\phi_{\xi\xi x\tau} &= E\{\xi_k \xi_{k-\tau} x_{k-\tau}\} = 0 \forall \tau, \\
\phi_{x^2\xi\tau} &= E\{(u_k^2 - E\{x_k^2\})\xi_{k-\tau}\} = 0 \forall \tau, \\
\phi_{x^2\xi^2\tau} &= E\{(u_k^2 - E\{x_k^2\})\xi_{k-\tau}^2\} = 0 \forall \tau, \\
\phi_{(y\xi)x^2\tau} &= E\{(y_k\xi_k - E\{y_k\xi_k\})(x_{k-\tau}^2 - E\{x_k^2\})\} = 0 \forall \tau,
\end{aligned}
\tag{2}
$$

where δ is the Dirac delta function and the cross-correlation function ϕ is denoted by (Billings, S. A. and Voon, W. S. F.):

$$
\phi_{ab\tau} = \frac{\frac{1}{n}\sum_{k=1}^{n-\tau}(a_k - \hat{a})(b_{k+\tau} - \hat{b})}{\sqrt{\frac{1}{n}\sum_{k=1}^{n}(a_k - \hat{a})^2}\sqrt{\frac{1}{n}\sum_{k=1}^{n}(b_k - \hat{b})^2}} = \frac{\sum_{k=1}^{n-\tau}(a_k - \hat{a})(b_{k+\tau} - \hat{b})}{\sqrt{\sum_{k=1}^{n}(a_k - \hat{a})^2}\sqrt{\sum_{k=1}^{n}(b_k - \hat{b})^2}},
\tag{3}
$$

where a and b are two signal sequences. If the tests are true, then the model residues can be considered as white noise.

Metrics Available in SysIdentPy

SysIdentPy provides the following regression metrics out of the box:

- forecast_error
- mean_forecast_error
- mean_squared_error
- root_mean_squared_error
- normalized_root_mean_squared_error
- root_relative_squared_error
- mean_absolute_error
- mean_squared_log_error
- median_absolute_error
- explained_variance_score
- r2_score
- symmetric_mean_absolute_percentage_error

To use them, the user only need to import the desired metric using, for example

```
from sysidentpy.metrics import root_relative_squared_error
```

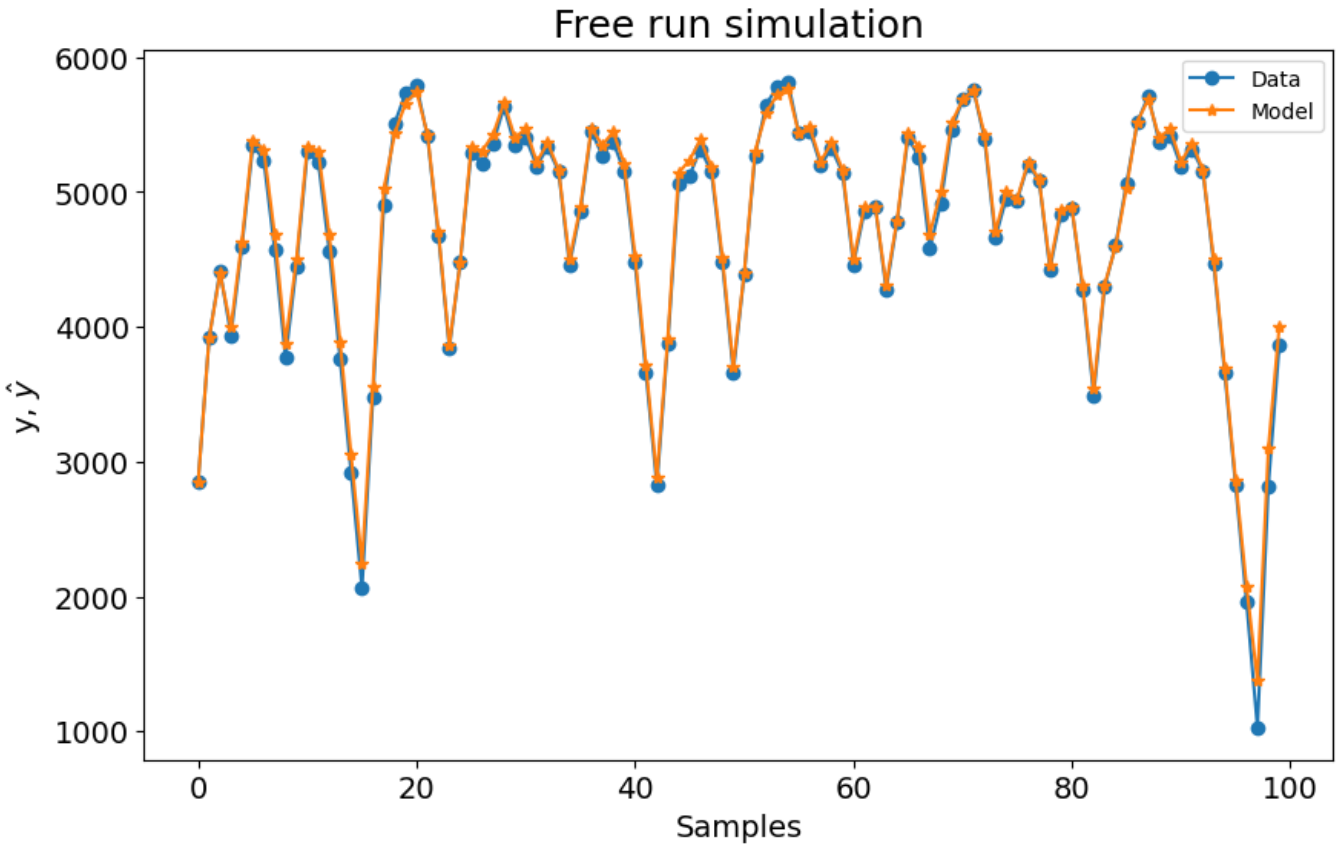
SysIdentPy also provides methods for calculate and analyse the residues correlation

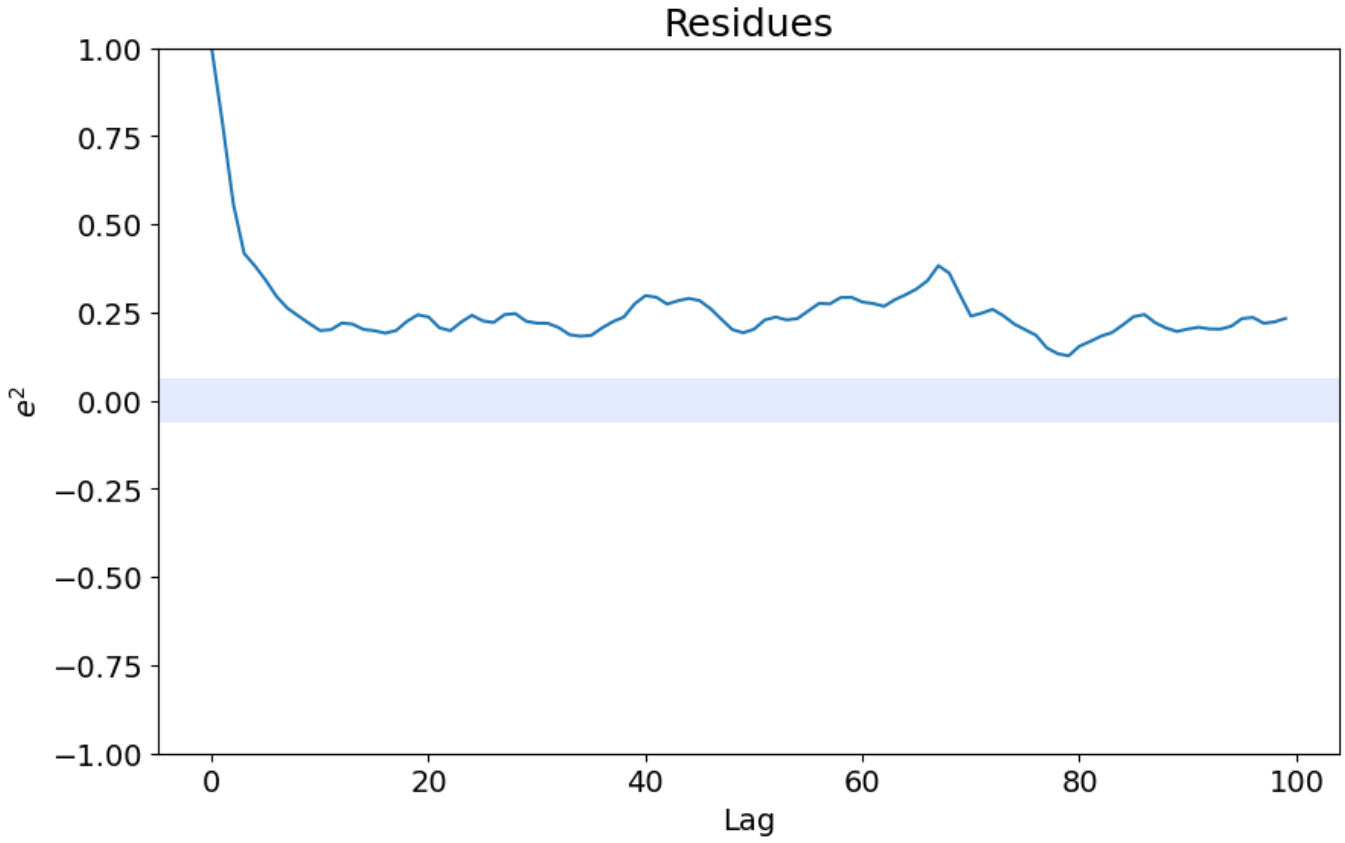
```
from sysidentpy.utils.plotting import plot_residues_correlation
from sysidentpy.residues.residues_correlation import (
    compute_residues_autocorrelation,
    compute_cross_correlation,
)
```

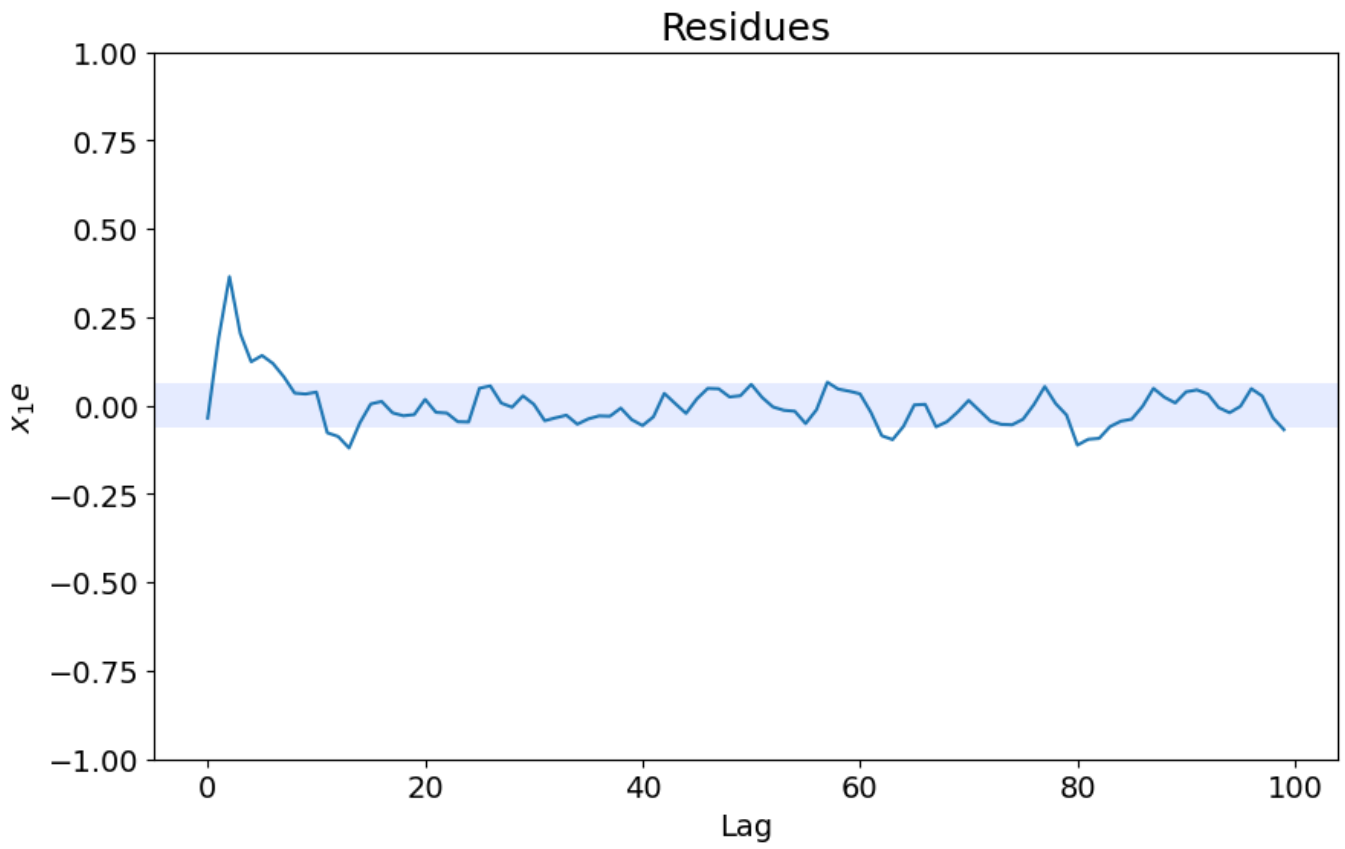
Lets check the metrics of the eletro mechanical system modeled in Chapter 4.

```
import numpy as np
import pandas as pd
from sysidentpy.model_structure_selection import FROLS
from sysidentpy.basis_function._basis_function import Polynomial
from sysidentpy.parameter_estimation import LeastSquares
from sysidentpy.utils.display_results import results
from sysidentpy.utils.plotting import plot_residues_correlation, plot_results
from sysidentpy.residues.residues_correlation import (
    compute_residues_autocorrelation,
    compute_cross_correlation,
)
from sysidentpy.metrics import root_relative_squared_error

df1 = pd.read_csv("examples/datasets/x_cc.csv")
df2 = pd.read_csv("examples/datasets/y_cc.csv")

x_train, x_valid = np.split(df1.iloc[::500].values, 2)
y_train, y_valid = np.split(df2.iloc[::500].values, 2)

basis_function = Polynomial(degree=2)
model = FROLS(
    order_selection=True,
    n_info_values=15,
    ylag=2,
    xlag=2,
    info_criteria="bic",
    estimator=LeastSquares(unbiased=False),
    basis_function=basis_function
)
model.fit(X=x_train, y=y_train)
yhat = model.predict(X=x_valid, y=y_valid)
rrse = root_relative_squared_error(y_valid, yhat)
print(rrse)
# plot only the first 100 samples (n=100)
plot_results(y=y_valid, yhat=yhat, n=100)

ee = compute_residues_autocorrelation(y_valid, yhat)
plot_residues_correlation(data=ee, title="Residues", ylabel="$e^2$")
```

```
x1e = compute_cross_correlation(y_valid, yhat, x_valid)
plot_residues_correlation(data=x1e, title="Residues", ylabel="$x_1e$")
```







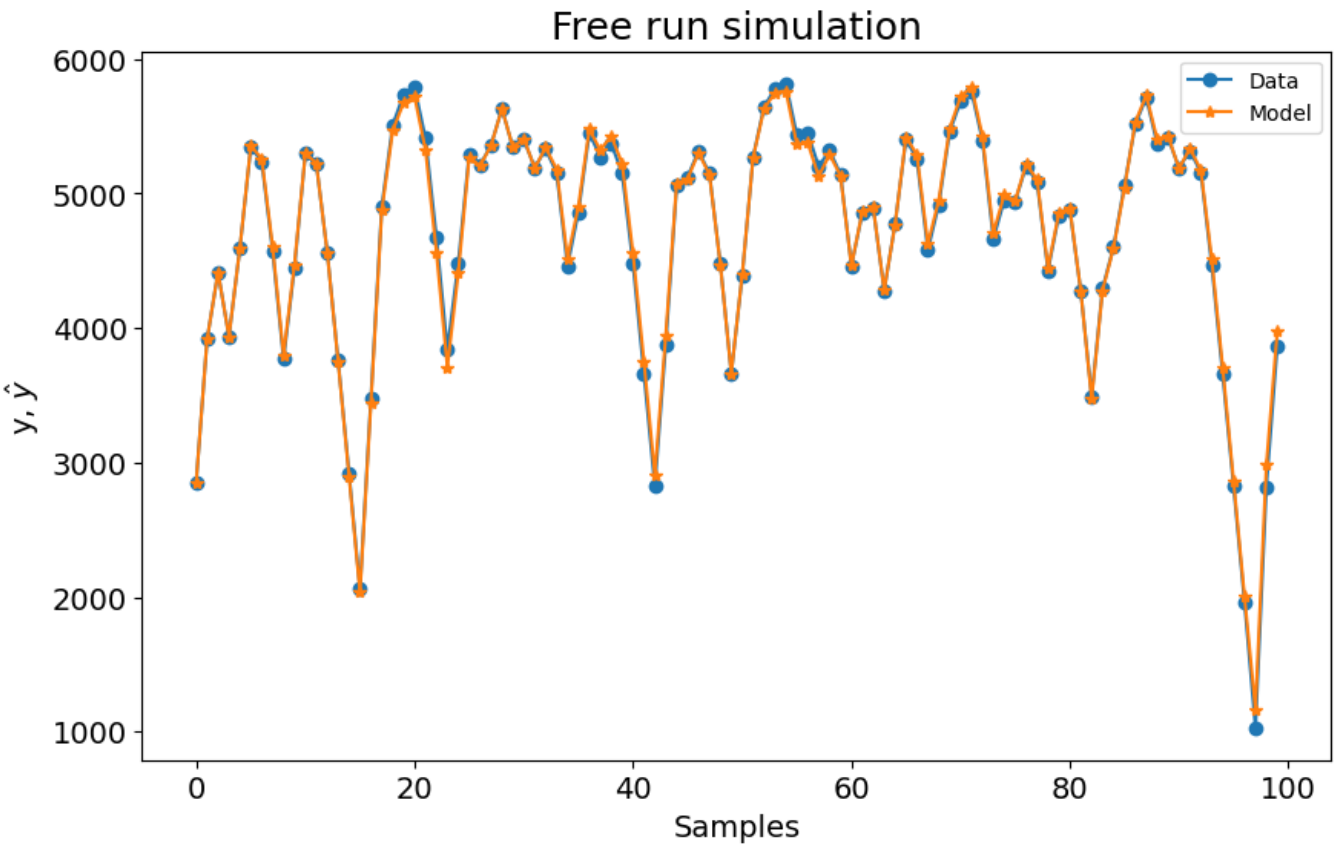
The RRSE is 0.0800, which is a very good metric. However, we can see that the residues have somo high auto-correlations anda with the input. This mean that our model maybe is not good enough as it could be.

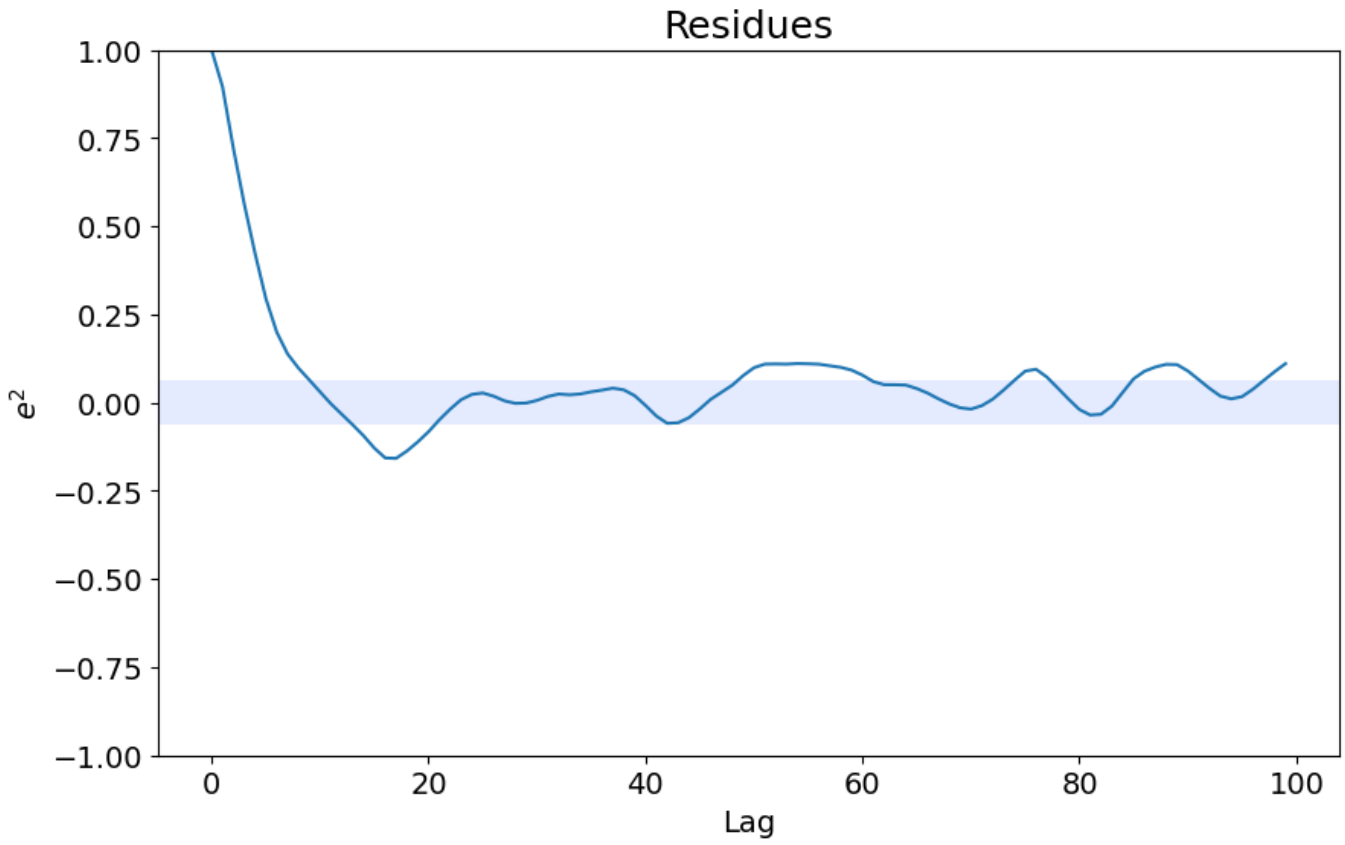
Lets check what happens if we increase `xlag`, `ylag` and change the parameter estimation algorithm from Least Squares to the Recursive Least Squares

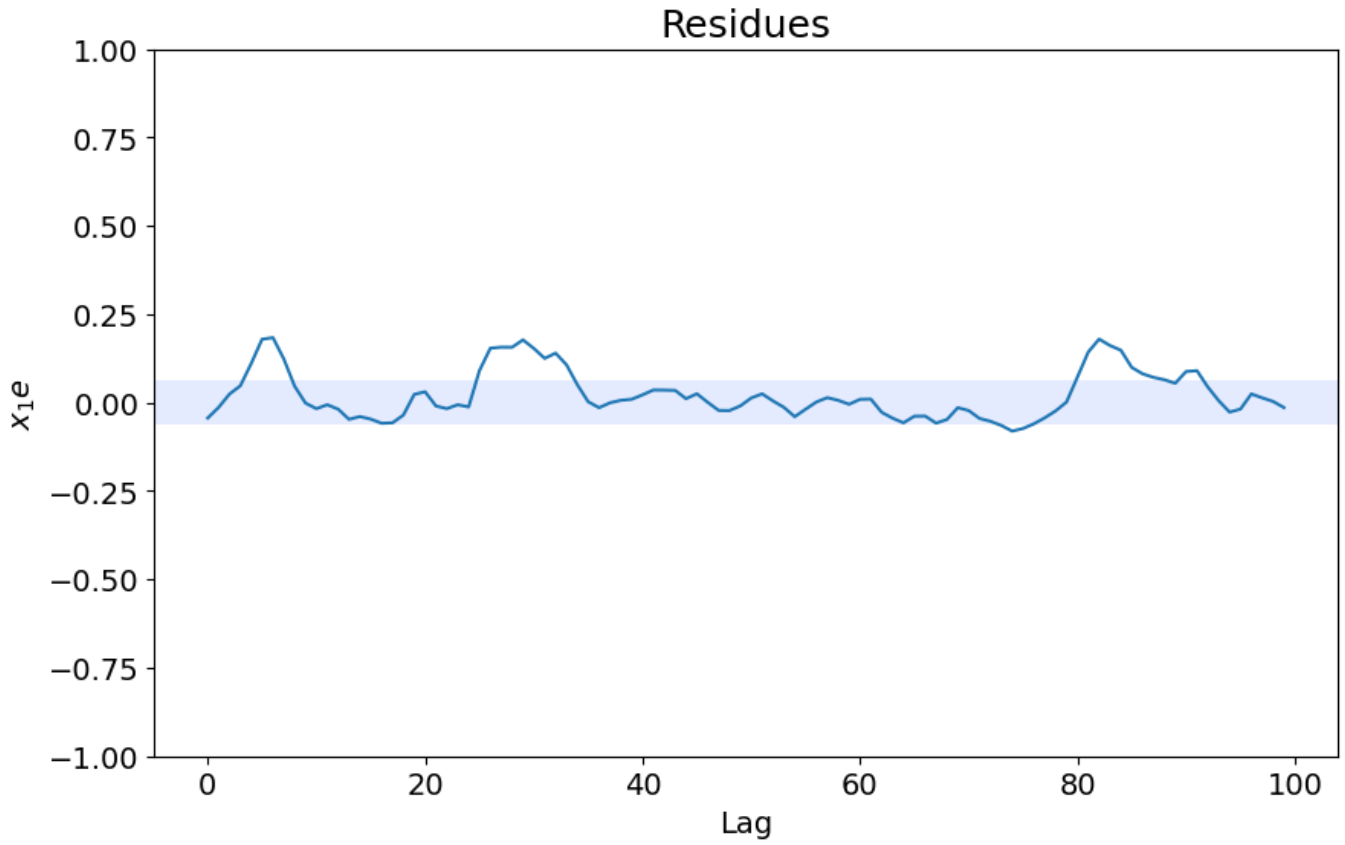
```
basis_function = Polynomial(degree=2)
model = FROLS(
    order_selection=True,
    n_info_values=50,
    ylag=5,
    xlag=5,
    info_criteria="bic",
    estimator=RecursiveLeastSquares(unbiased=False),
    basis_function=basis_function
)

model.fit(X=x_train, y=y_train)
yhat = model.predict(X=x_valid, y=y_valid)
rrse = root_relative_squared_error(y_valid, yhat)
print(rrse)
# plot only the first 100 samples (n=100)
plot_results(y=y_valid, yhat=yhat, n=100)
ee = compute_residues_autocorrelation(y_valid, yhat)
```

```
plot_residues_correlation(data=ee, title="Residues", ylabel="$e^2$")
x1e = compute_cross_correlation(y_valid, yhat, x_valid)
plot_residues_correlation(data=x1e, title="Residues", ylabel="$x_1e$")
```

Now the RRSE is 0.0568 and we have a better residual correlation!





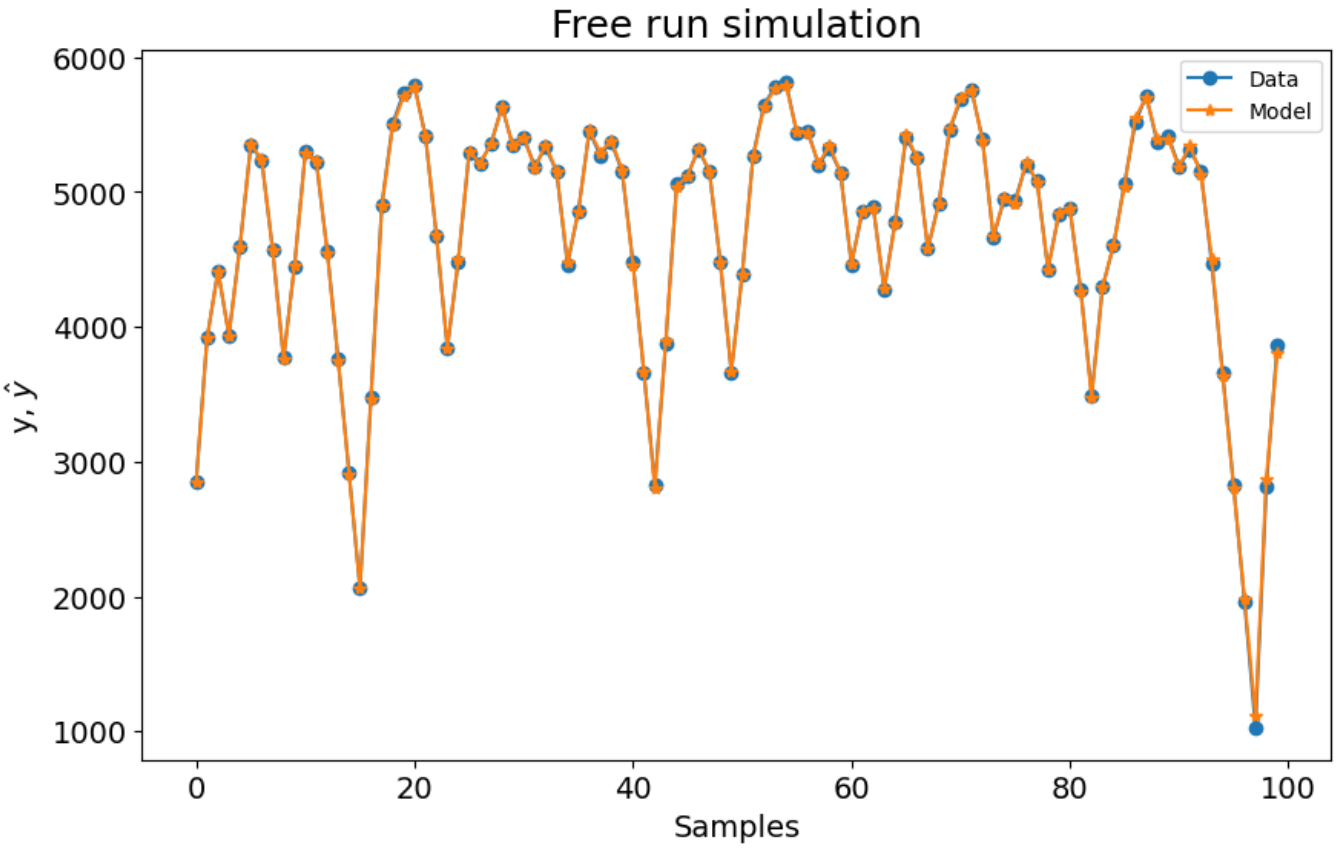


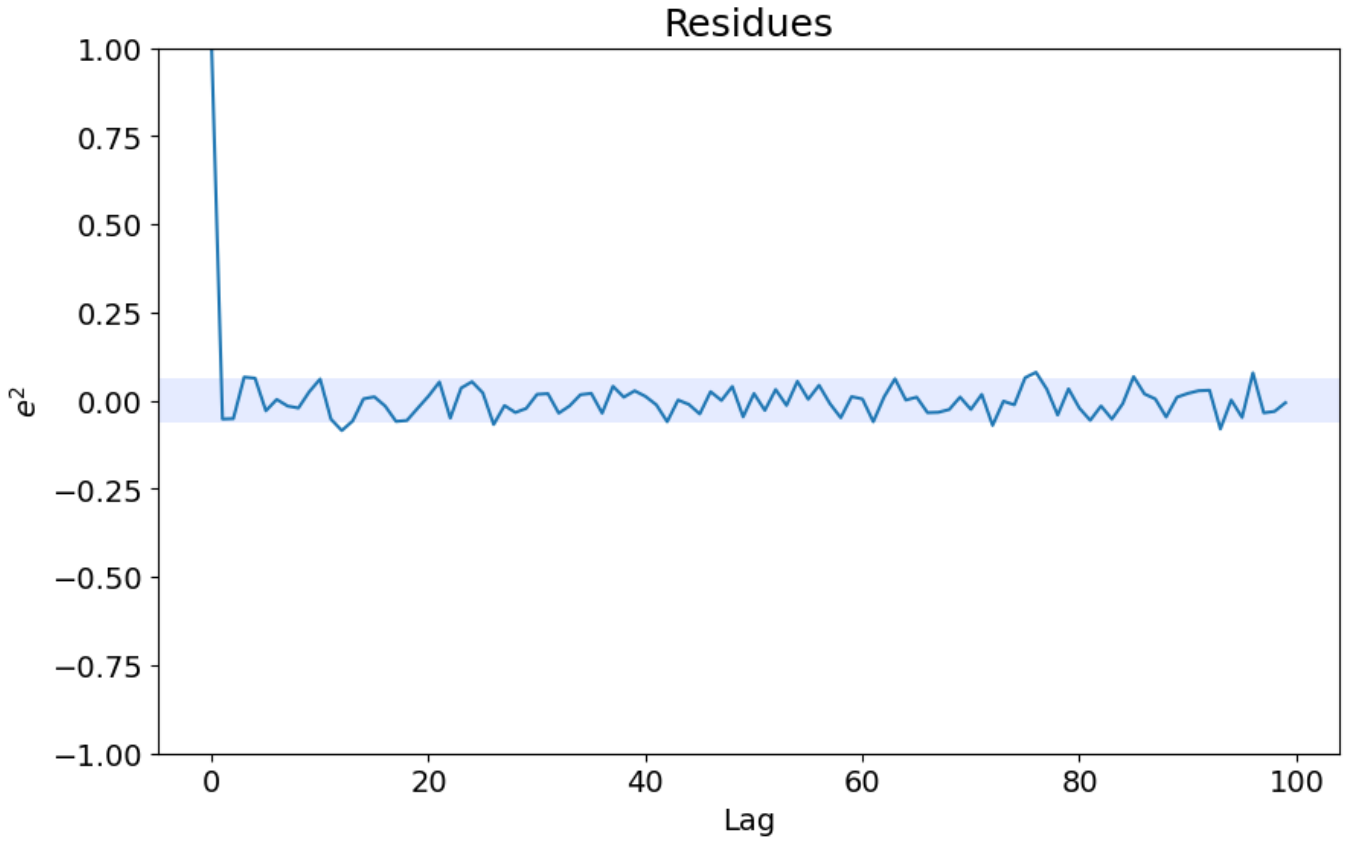
In the end of day, the best model will be the model that satisfy the user needs. However, its important to understand how to analyse the models so you can have an idea if you can get some improvements without too much work.

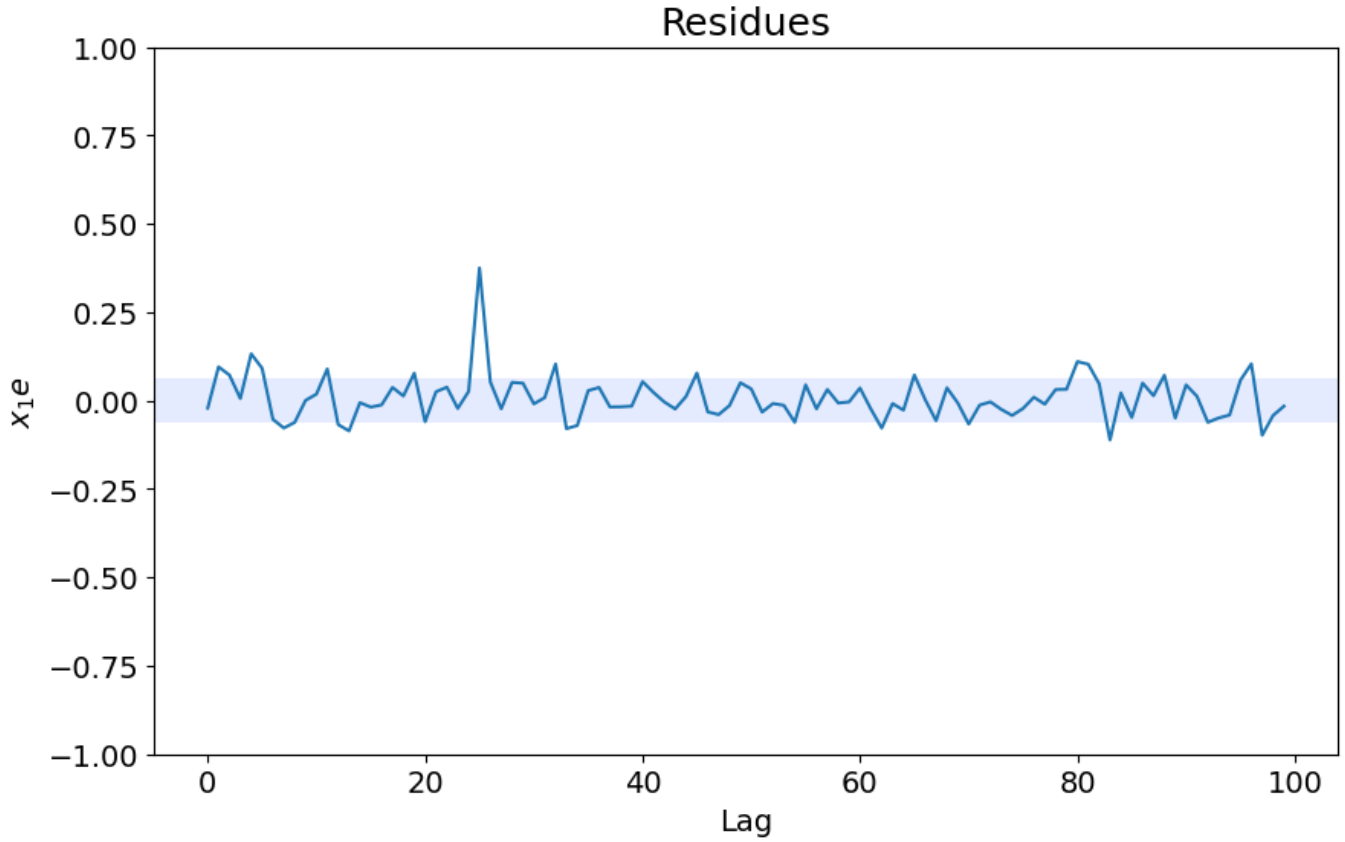
For the sake of curiosity, lets check how the model perform if we run a 1-step ahead prediction. We don't need to fit the model again, just make another prediction using the 1-step option.

```
yhat = model.predict(X=x_valid, y=y_valid, steps_ahead=1)
rrse = root_relative_squared_error(y_valid, yhat)
print(rrse)
# plot only the first 100 samples (n=100)
plot_results(y=y_valid, yhat=yhat, n=100)
ee = compute_residues_autocorrelation(y_valid, yhat)
plot_residues_correlation(data=ee, title="Residues", ylabel="$e^2$")
x1e = compute_cross_correlation(y_valid, yhat, x_valid)
plot_residues_correlation(data=x1e, title="Residues", ylabel="$x_1e$")
```

The same model, but evaluating the 1-step ahead prediction, now return a RRSE$= 0.02044$ and the residues are even better. But remember, that is expected, as explained in the previous section.




Residues
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
e^2
0
20
40
60
80
100
Lag


Residues
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
$x_1 e$
0
20
40
60
80
100
Lag

10 - Case Studies

M4 Dataset

The M4 dataset is a well known resource for time series forecasting, offering a wide range of data series used to test and improve forecasting methods. Created for the M4 competition organized by Spyros Makridakis, this dataset has driven many advancements in forecasting techniques.

The M4 dataset includes 100,000 time series from various fields such as demographics, finance, industry, macroeconomics, and microeconomics, which were selected randomly from the ForeDeCk database. The series come in different frequencies (yearly, quarterly, monthly, weekly, daily, and hourly), making it a comprehensive collection for testing forecasting methods.

In this case study, we will focus on the hourly subset of the M4 dataset. This subset consists of time series data recorded hourly, providing a detailed and high-frequency look at changes over time. Hourly data presents unique challenges due to its granularity and the potential for capturing short-term fluctuations and patterns.

The M4 dataset provides a standard benchmark to compare different forecasting methods, allowing researchers and practitioners to evaluate their models consistently. With series from various domains and frequencies, the M4 dataset represents real-world forecasting challenges, making it valuable for developing robust forecasting techniques. The competition and the dataset itself have led to the creation of new algorithms and methods, significantly improving forecasting accuracy and reliability.

We will present a end to end walkthrough using the M4 hourly dataset to demonstrate the capabilities of SysIdentPy. SysIdentPy offers a range of tools and techniques designed to effectively handle the complexities of time series data, but we will focus on fast and easy setup for this case. We will cover model selection and evaluation metrics specific to the hourly dataset.

By the end of this case study, you will have a solid understanding of how to use SysIdentPy for forecasting with the M4 hourly dataset, preparing you to tackle similar forecasting challenges in real-world scenarios.

Required Packages and Versions

To ensure that you can replicate this case study, it is essential to use specific versions of the required packages. Below is a list of the packages along with their respective versions needed for running the case studies effectively.

To install all the required packages, you can create a `requirements.txt` file with the following content:

```
sysidentpy==0.4.0
datasetsforecast==0.0.8
pandas==2.2.2
numpy==1.26.0
matplotlib==3.8.4
s3fs==2024.6.1
```

Then, install the packages using:

```
pip install -r requirements.txt
```

- Ensure that you use a virtual environment to avoid conflicts between package versions.
- Versions specified are based on compatibility with the code examples provided. If you are using different versions, some adjustments in the code might be necessary.

SysIdentPy configuration

In this section, we will demonstrate the application of SysIdentPy to the Silver box dataset. The following code will guide you through the process of loading the dataset, configuring the SysIdentPy parameters, and building a model for mentioned system.

```
import warnings
import numpy as np
import pandas as pd
from pandas.errors import SettingWithCopyWarning
import matplotlib.pyplot as plt

from sysidentpy.model_structure_selection import FROLS, AOLS
from sysidentpy.basis_function._basis_function import Polynomial
from sysidentpy.parameter_estimation import LeastSquares
from sysidentpy.metrics import root_relative_squared_error,
symmetric_mean_absolute_percentage_error
from sysidentpy.utils.plotting import plot_results

from datasetsforecast.m4 import M4, M4Evaluation

warnings.simplefilter(action='ignore', category=FutureWarning)
warnings.simplefilter(action='ignore', category=UserWarning)
warnings.simplefilter(action='ignore', category=SettingWithCopyWarning)

train = pd.read_csv('https://auto-arima-results.s3.amazonaws.com/M4-Hourly.csv')
test = pd.read_csv('https://auto-arima-results.s3.amazonaws.com/M4-Hourly-
test.csv').rename(columns={'y': 'y_test'})
```

The following plots provide a visualization of the training data for a small subset of the time series. The plot shows the raw data, giving you an insight into the patterns and behaviors inherent in each series.

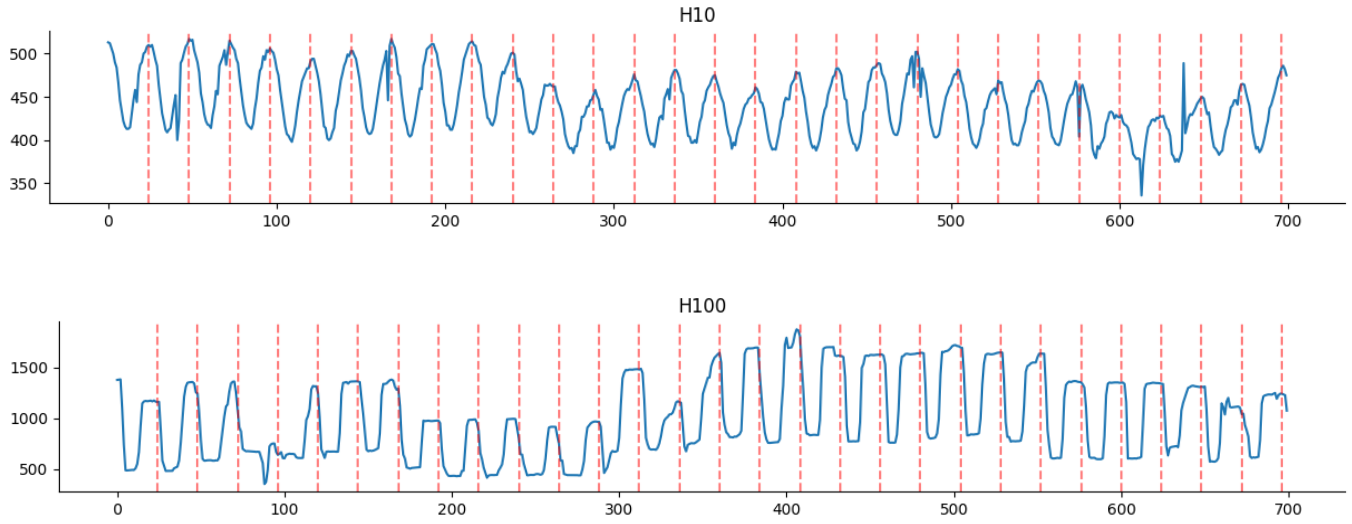
By observing the data, you can get a sense of the variety and complexity of the time series we are working with. The plots can reveal important characteristics such as trends, seasonal patterns, and potential anomalies within the time series. Understanding these elements is crucial for the development of accurate forecasting models.

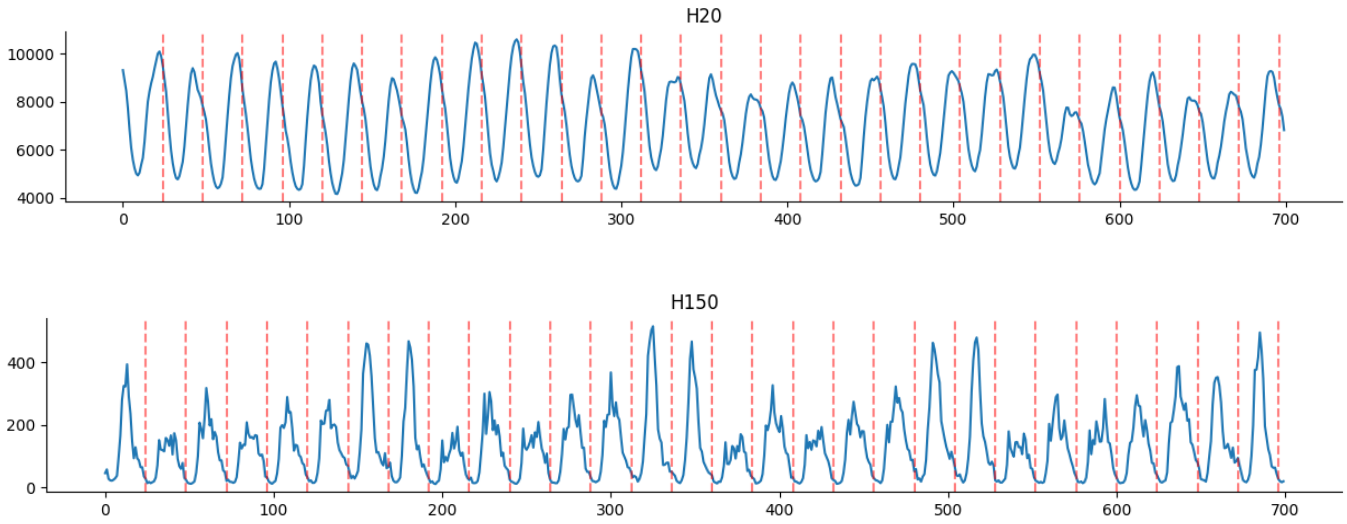
However, when dealing with a large number of different time series, it is common to start with broad assumptions rather than detailed individual analysis. In this context, we will adopt a similar approach. Instead of going into the specifics of each dataset, we will make some general assumptions and see how SysIdentPy handles them.

This approach provides a practical starting point, demonstrating how SysIdentPy can manage different types of time series data without too much work. As you become more familiar with the tool, you can refine your models with more detailed insights. For now, let's focus on using SysIdentPy to create the forecasts based on these initial assumptions.

Our first assumption is that there is a 24-hour seasonal pattern in the series. By examining the plots below, this seems reasonable. Therefore, we'll begin building our models with `ylag=24`.

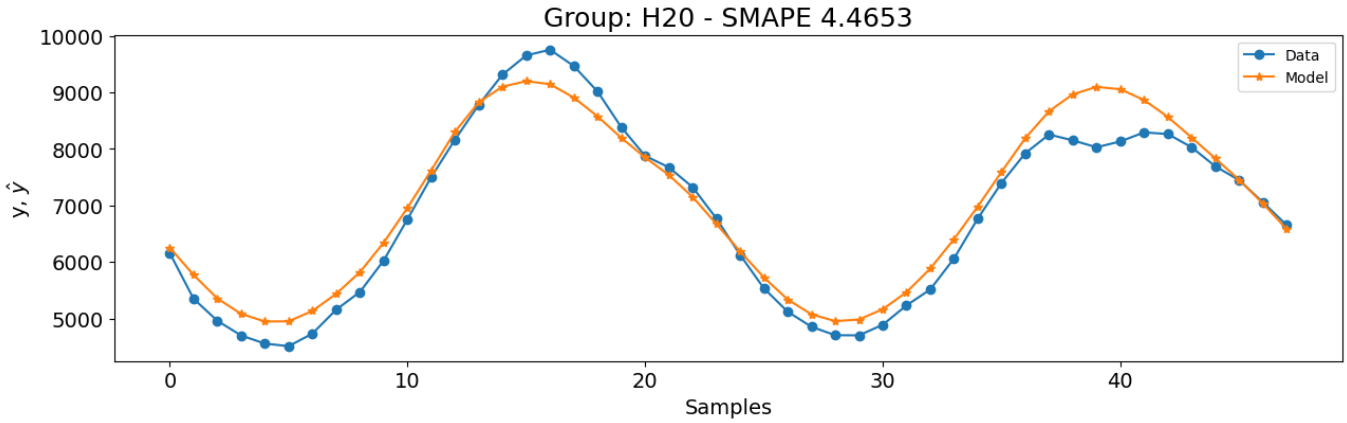
```
ax = train[train["unique_id"]=="H10"].reset_index(drop=True)["y"].plot(figsize=(15,
2), title="H10")
xcoords = [a for a in range(24, 24*30, 24)]

for xc in xcoords:
    plt.axvline(x=xc, color='red', linestyle='--', alpha=0.5)
```





Lets check build a model for the `H20` group before we extrapolate the settings for every group. Because there are no input features, we will be using a `NAR` model type in SysIdentPy. To keep things simple and fast, we will start with Polynomial basis function with degree 1.

```
unique_id = "H20"
y_id = train[train["unique_id"]==unique_id]["y"].values.reshape(-1, 1)
y_val = test[test["unique_id"]==unique_id]["y_test"].values.reshape(-1, 1)

basis_function = Polynomial(degree=1)
model = FROLS(
        order_selection=True,
        ylag=24,
        estimator=LeastSquares(),
        basis_function=basis_function,
        model_type="NAR",
)

model.fit(y=y_id)
y_val = np.concatenate([y_id[-model.max_lag :], y_val])
y_hat = model.predict(y=y_val, forecast_horizon=48)
smape = symmetric_mean_absolute_percentage_error(y_val[model.max_lag::],
y_hat[model.max_lag::])

plot_results(y=y_val[model.max_lag :], yhat=y_hat[model.max_lag :], n=30000,
figsize=(15, 4), title=f"Group: {unique_id} - SMAPE {round(smape, 4)}")
```



Probably, the result are not optimal and will not work for every group. However, let's check how this setting performs against the winner model M4 time series competition: the Exponential Smoothing with Recurrent Neural Networks (ESRNN).

```
esrnn_url = 'https://github.com/Nixtla/m4-forecasts/raw/master/forecasts/submission-118.zip'
esrnn_forecasts = M4Evaluation.load_benchmark('data', 'Hourly', esrnn_url)
esrnn_evaluation = M4Evaluation.evaluate('data', 'Hourly', esrnn_forecasts)

esrnn_evaluation
```

	SMAPE	MASE	OWA
Hourly	9.328	0.893	0.440

Table 1. ESRNN SOTA results

The following code took only 49 seconds to run on my machine (AMD Ryzen 5 5600x processor, 32GB RAM at 3600MHz). Because of its efficiency, I didn't create a parallel version. By the end of this use case, you will see how SysIdentPy can be both fast and effective, delivering good results without too much optimization.

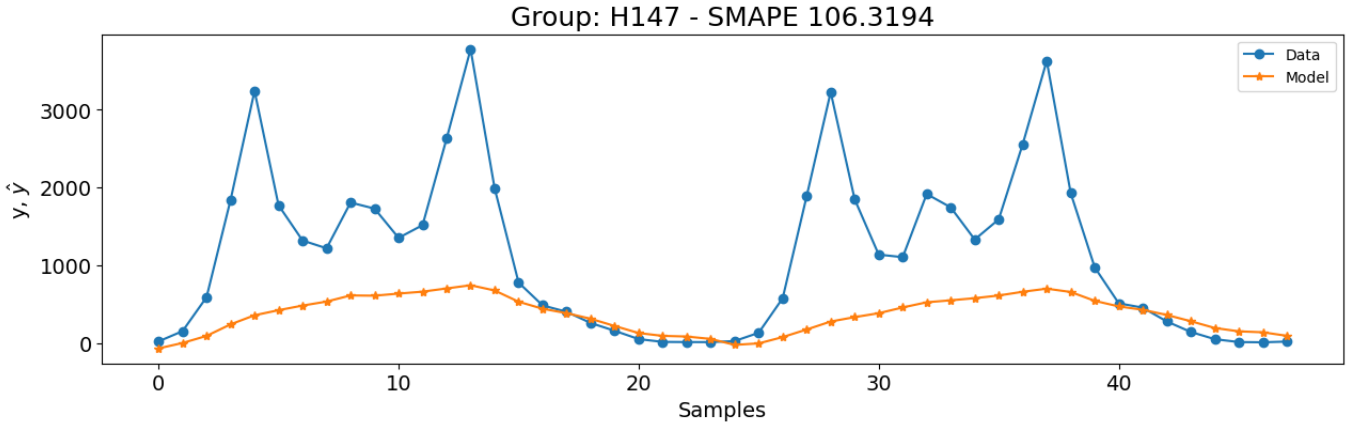
```
r = []
ds_test = list(range(701, 749))
for u_id, data in train.groupby(by=["unique_id"], observed=True):
    y_id = data["y"].values.reshape(-1, 1)
    basis_function = Polynomial(degree=1)
    model = FROLS(
                ylag=24,
                estimator=LeastSquares(),
                basis_function=basis_function,
                model_type="NAR",
                n_info_values=25,
        )
    try:
```

```
        model.fit(y=y_id)
        y_val = y_id[-model.max_lag :].reshape(-1, 1)
        y_hat = model.predict(y=y_val, forecast_horizon=48)
        r.append(
            [
                u_id*len(y_hat[model.max_lag::]),
                ds_test,
                y_hat[model.max_lag::].ravel()
            ]
        )
    except Exception:
        print(f"Problem with {u_id}")

results_1 = pd.DataFrame(r, columns=["unique_id", "ds",
"NARMAX_1"]).explode(['unique_id', 'ds', 'NARMAX_1'])
results_1["NARMAX_1"] = results_1["NARMAX_1"].astype(float)#.clip(lower=10)
pivot_df = results_1.pivot(index='unique_id', columns='ds', values='NARMAX_1')
results = pivot_df.to_numpy()

M4Evaluation.evaluate('data', 'Hourly', results)
```

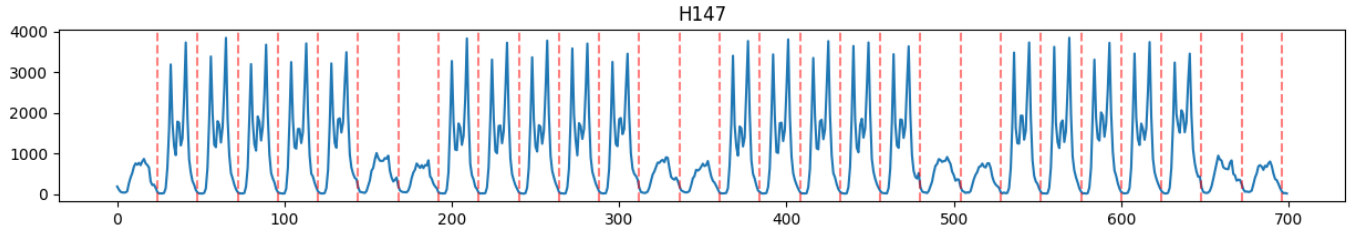
	SMAPE	MASE	OWA
Hourly	16.034196	0.958083	0.636132

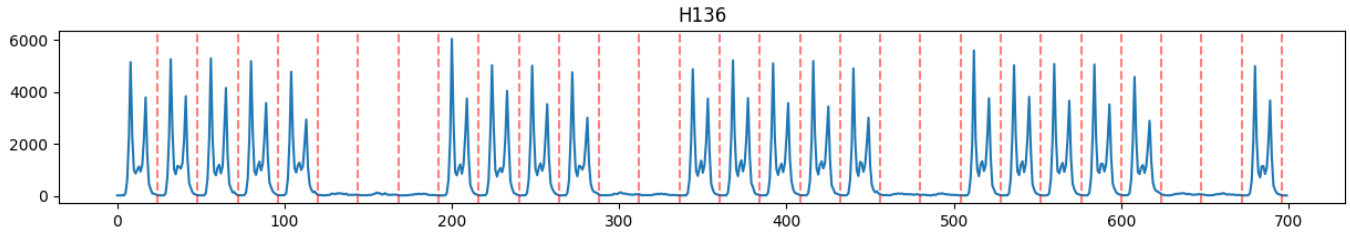
Table 2. First test with SysIdentPy

The initial results are reasonable, but they don't quite match the performance of `ESRNN`. These results are based solely on our first assumption. To better understand the performance, let's examine the groups with the worst results.

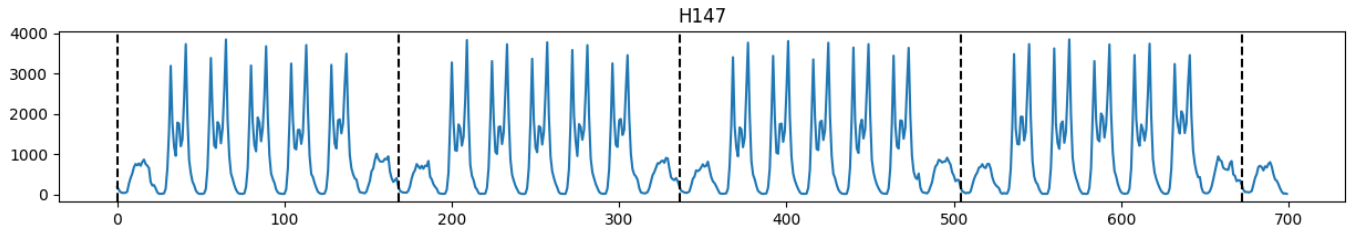


The following plot illustrates two such groups, `H147` and `H136`. Both exhibit a 24-hour seasonal pattern.

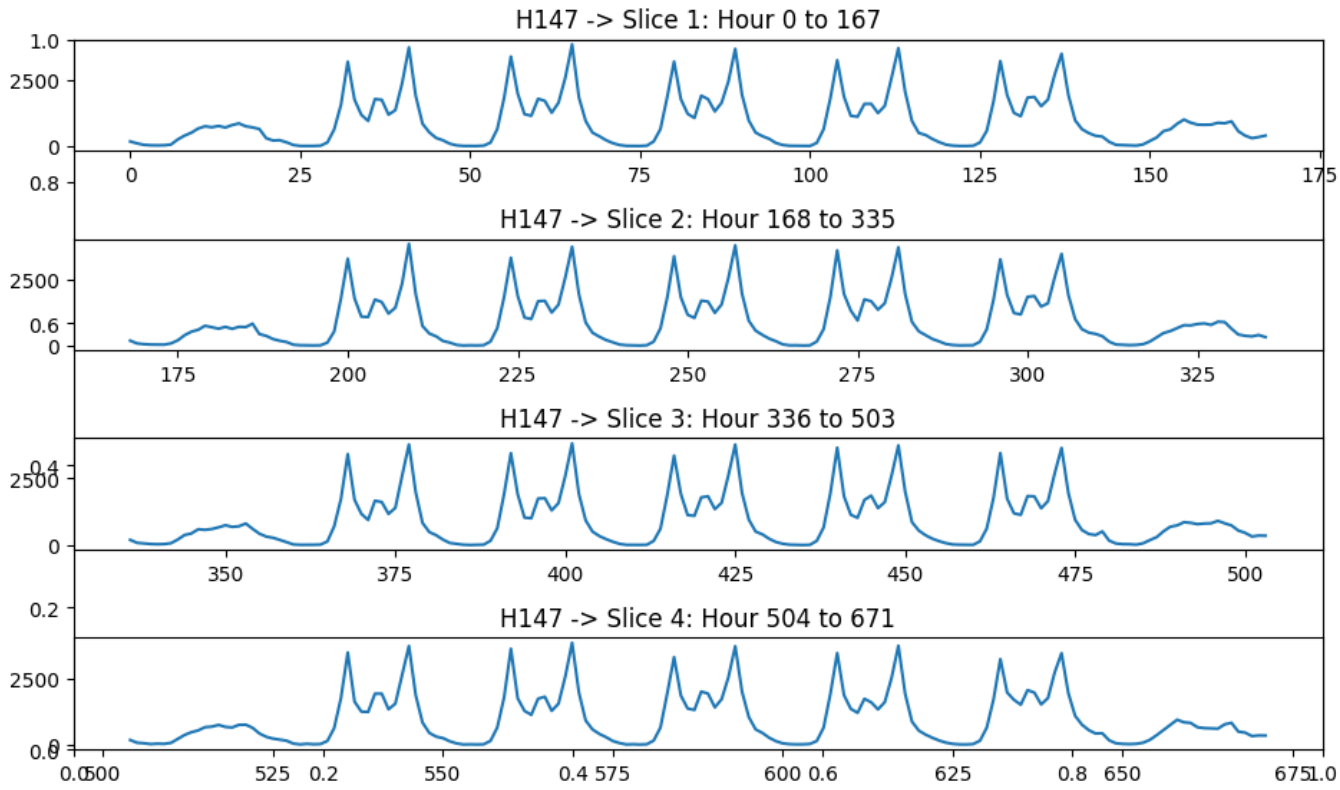




However, a closer look reveals an additional insight: in addition to the daily pattern, these series also show a weekly pattern. Observe how the data looks like when we split the series into weekly segments.




```
xcoords = list(range(0, 168*5, 168))
filtered_train = train[train["unique_id"] == "H147"].reset_index(drop=True)

fig, ax = plt.subplots(figsize=(10, 1.5 * len(xcoords[1:])))
for i, start in enumerate(xcoords[:-1]):
    end = xcoords[i + 1]
    ax = fig.add_subplot(len(xcoords[1:]), 1, i + 1)
    filtered_train["y"].iloc[start:end].plot(ax=ax)
    ax.set_title(f'H147 -> Slice {i+1}: Hour {start} to {end-1}')

plt.tight_layout()
plt.show()
```



Therefore, we will build models setting `ylag=168`.

> Note that this is a very high number for lags, so be careful if you want to try it with higher polynomial degrees because the time to run the models can increase significantly. I tried some configurations with polynomial degree equal to 2 and only took 6 minutes to run (even less, using `AOLS`), without making the code run in parallel. As you can see, SysIdentPy can be very fast and you can make it faster by applying parallelization.

```
# this took 2min to run on my computer.
r = []
ds_test = list(range(701, 749))
for u_id, data in train.groupby(by=["unique_id"], observed=True):
    y_id = data["y"].values.reshape(-1, 1)
    basis_function = Polynomial(degree=1)
    model = FROLS(
            ylag=168,
            estimator=LeastSquares(),
            basis_function=basis_function,
            model_type="NAR",
        )
    try:
        model.fit(y=y_id)
        y_val = y_id[-model.max_lag :].reshape(-1, 1)
        y_hat = model.predict(y=y_val, forecast_horizon=48)
        r.append(
            [
                u_id*len(y_hat[model.max_lag::]),
```

```
                ds_test,
                y_hat[model.max_lag::].ravel()
            ]
        )
    except Exception:
        print(f"Problema no id {u_id}")

results_1 = pd.DataFrame(r, columns=["unique_id", "ds",
"NARMAX_1"]).explode(['unique_id', 'ds', 'NARMAX_1'])
results_1["NARMAX_1"] = results_1["NARMAX_1"].astype(float)#.clip(lower=10)
pivot_df = results_1.pivot(index='unique_id', columns='ds', values='NARMAX_1')
results = pivot_df.to_numpy()
M4Evaluation.evaluate('data', 'Hourly', results)
```

	SMAPE	MASE	OWA
Hourly	10.475998	0.773749	0.446471

> Table 3. Improved results using SysIdentPy

Now, the results are much closer to those of the `ESRNN` model! While the Symmetric Mean Absolute Percentage Error (`SMAPE`) is slightly worse, the Mean Absolute Scaled Error (`MASE`) is better when comparing against `ESRNN`, leading to a very similar Overall Weighted Average (`OWA`) metric. Remarkably, these results are achieved using only simple `AR` models. Next, let's see if the `AOLS` method can provide even better results.

```
r = []
ds_test = list(range(701, 749))
for u_id, data in train.groupby(by=["unique_id"], observed=True):
    y_id = data["y"].values.reshape(-1, 1)
    basis_function = Polynomial(degree=1)
    model = AOLS(
                ylag=168,
                basis_function=basis_function,
                model_type="NAR",
                # due to high lag settings, k was increased to 6 as an initial guess
                k=6,
        )
    try:
        model.fit(y=y_id)
        y_val = y_id[-model.max_lag :].reshape(-1, 1)
        y_hat = model.predict(y=y_val, forecast_horizon=48)
        r.append(
            [
                u_id*len(y_hat[model.max_lag::]),
                ds_test,
                y_hat[model.max_lag::].ravel()
            ]
        )
```

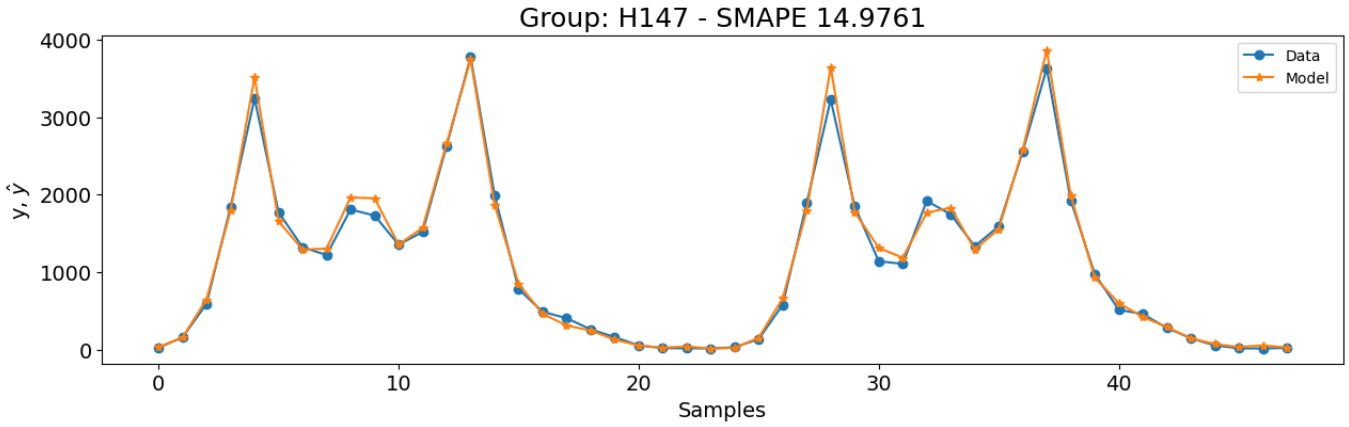
```
    except Exception:
        print(f"Problema no id {u_id}")

results_1 = pd.DataFrame(r, columns=["unique_id", "ds",
"NARMAX_1"]).explode(['unique_id', 'ds', 'NARMAX_1'])
results_1["NARMAX_1"] = results_1["NARMAX_1"].astype(float)#.clip(lower=10)
pivot_df = results_1.pivot(index='unique_id', columns='ds', values='NARMAX_1')
results = pivot_df.to_numpy()
M4Evaluation.evaluate('data', 'Hourly', results)
```

	SMAPE	MASE	OWA
Hourly	9.951141	0.809965	0.439755

> Table 4. SysIdentPy results using AOLS algorithm

The Overall Weighted Average (`OWA`) is even better than that of the `ESRNN` model! Additionally, the `AOLS` method was incredibly efficient, taking only **6 seconds to run**. This combination of high performance and rapid execution makes `AOLS` a compelling alternative for time series forecasting in cases with multiple series.

Before we finish, let's verify how the performance of the `H147` model has improved with the `ylag=168` setting.



> Based on the M4 benchmark paper, we could also clip the predictions lower than 10 to 10 and the results would be slightly better. But this is left to the user.

We could achieve even better performance with some fine-tuning of the model configuration. However, I'll leave exploring these alternative adjustments as an exercise for the user. However, keep in mind that experimenting with different settings does not always guarantee improved results. A deeper theoretical knowledge can often lead you to better configurations and, hence, better results.

Coupled Eletric Device

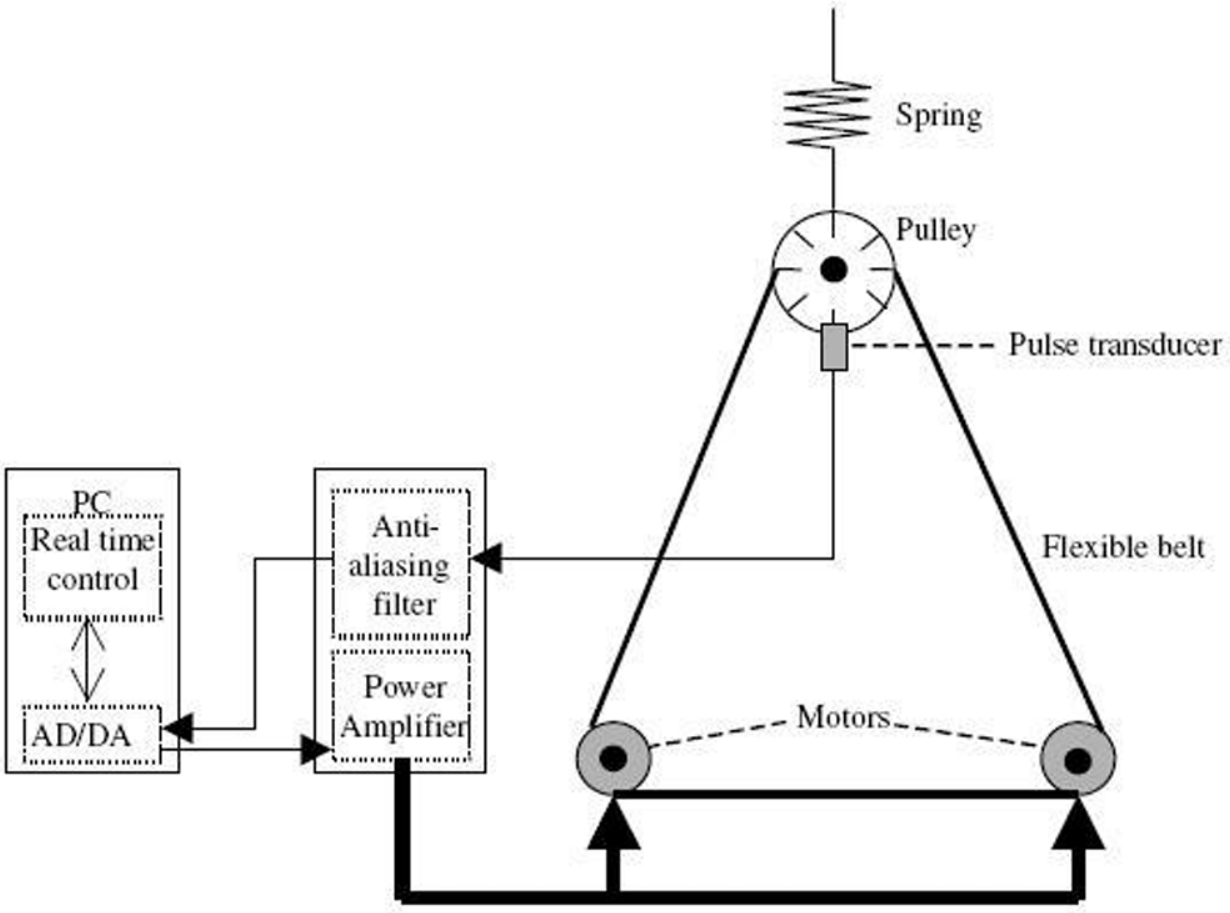
The CE8 coupled electric drives [dataset](Nonlinear Benchmark) presents a compelling use case for demonstrating the performance of SysIdentPy. This system involves two electric motors driving a pulley with a flexible belt, creating a dynamic environment ideal for testing system identification tools.

> The [nonlinear benchmark website]((Nonlinear Benchmark)) stands as a significant contribution to the system identification and machine learning community. The users are encouraged to explore all the papers referenced on the site.

System Overview

The CE8 system, illustrated in Figure 1, features:

- **Two Electric Motors**: These motors independently control the tension and speed of the belt, providing symmetrical control around zero. This enables both clockwise and counterclockwise movements.
- **Pulley Mechanism**: The pulley is supported by a spring, introducing a lightly damped dynamic mode that adds complexity to the system.
- **Speed Control Focus**: The primary focus is on the speed control system. The pulley's angular speed is measured using a pulse counter, which is insensitive to the direction of the velocity.



> Figure 1. CE8 system design.

Sensor and Filtering

The measurement process involves:

- **Pulse Counter**: This sensor measures the angular speed of the pulley without regard to the direction.
- **Analogue Low Pass Filtering**: This reduces high-frequency noise, followed by anti-aliasing filtering to prepare the signal for digital processing. The dynamic effects are mainly influenced by the electric drive time constants and the spring, with the low pass filtering having a minimal impact on the output.

SOTA Results

SysIdentPy can be used to build robust models for identifying and modeling the complex dynamics of the CE8 system. The performance will be compared against a benchmark provided by Max D. Champneys, Gerben I. Beintema, Roland Tóth, Maarten Schoukens, and Timothy J. Rogers - Baselines for Nonlinear Benchmarks, Workshop on Nonlinear System Identification Benchmarks, 2024.

Benchmark	Silverbox			W-H	EMPS	Cascaded Tanks	CED	
Unit	RMS (mV)			RMS (mV)	RMS (mm)	RMS (V)	RMS (ticks/s)	
Test Set	multisine	arrow (full)	arrow (no extrap.)	test	test	test	test 1	test 2
LTI SS	6.96	14.4	6.58	43.4	5.53	0.588	0.224	0.384
LTI ARX	6.95	14.2	6.59	43.4	12.6	0.685	0.291	0.331
pNARX	0.640	2.25	0.571	27.3	5.17	0.413	0.10	0.141
GP NARX	0.301	0.60	0.259	23.1	1690	0.622	0.102	0.090
MLP NARX	2.80	5.18	2.40	310	133	2.04	0.142	0.101
MLP FIR	18.9	20.2	15.3	24.8	121	1.43	0.122	0.095
RNN	1.64	4.88	1.73	6.87	89.2	0.543	0.096	0.235
GRU	1.49	2.80	0.947	3.97	220	0.396	0.286	0.238
LSTM	1.50	2.68	0.960	4.45	74.1	0.490	0.303	0.145
OLSTM	1.51	2.38	1.08	4.55	111	0.471	0.151	0.259
SOTA*	0.36	0.26	0.32	0.241	2.64	0.191	0.115	0.074

The benchmark evaluate the average metric between the two experiments. That's why the SOTA method do not have the better metric for `test 1`, but it is still the best overall. The goal of this case study is not only to showcase the robustness of SysIdentPy but also provides valuable insights into its practical applications in real-world dynamic systems.

Required Packages and Versions

To ensure that you can replicate this case study, it is essential to use specific versions of the required packages. Below is a list of the packages along with their respective versions needed for running the

case studies effectively.

To install all the required packages, you can create a `requirements.txt` file with the following content:

```
sysidentpy==0.4.0
pandas==2.2.2
numpy==1.26.0
matplotlib==3.8.4
nonlinear_benchmarks==0.1.2
```

Then, install the packages using:

```
pip install -r requirements.txt
```

- Ensure that you use a virtual environment to avoid conflicts between package versions.
- Versions specified are based on compatibility with the code examples provided. If you are using different versions, some adjustments in the code might be necessary.

SysIdentPy configuration

In this section, we will demonstrate the application of SysIdentPy to the CE8 coupled electric drives dataset. This example showcases the robust performance of SysIdentPy in modeling and identifying complex dynamic systems. The following code will guide you through the process of loading the dataset, configuring the SysIdentPy parameters, and building a model for CE8 system.

This practical example will help users understand how to effectively utilize SysIdentPy for their own system identification tasks, leveraging its advanced features to handle the complexities of real-world dynamic systems. Let's dive into the code and explore the capabilities of SysIdentPy.

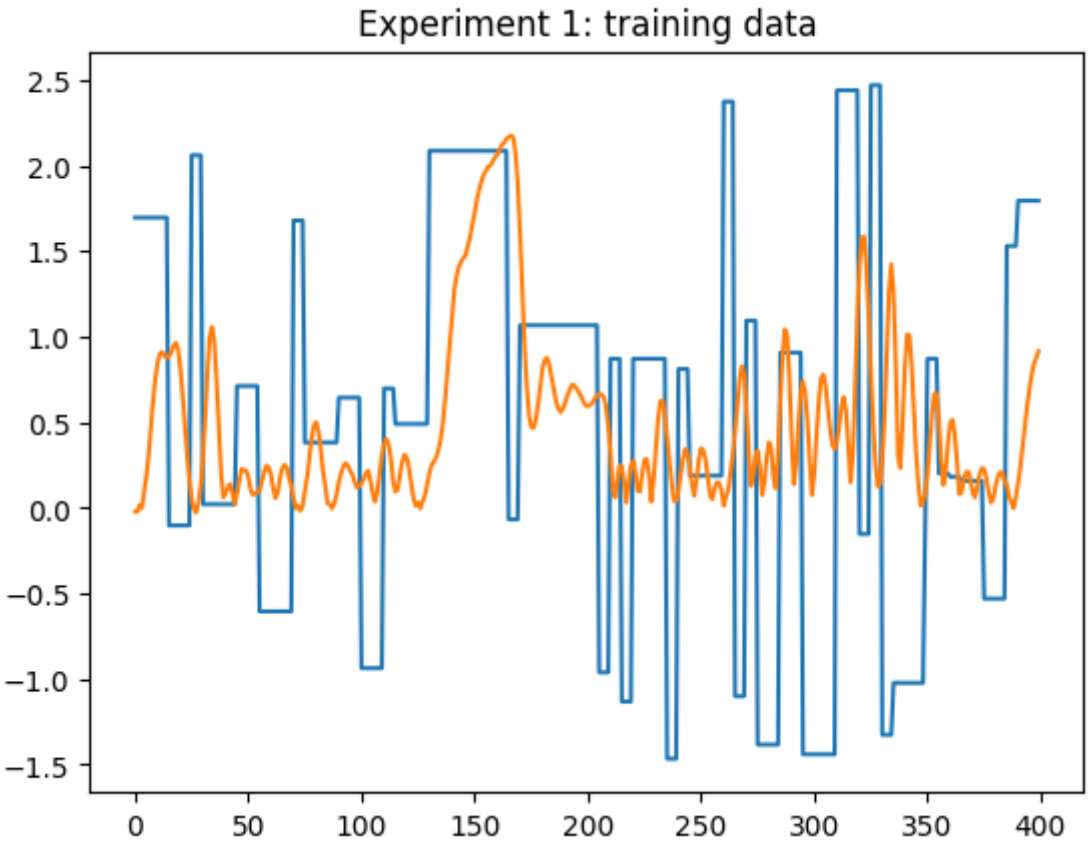
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

from sysidentpy.model_structure_selection import FROLS
from sysidentpy.basis_function import Polynomial, Fourier
from sysidentpy.utils.display_results import results
from sysidentpy.parameter_estimation import LeastSquares
from sysidentpy.metrics import root_mean_squared_error
from sysidentpy.utils.plotting import plot_results

import nonlinear_benchmarks
```

```
train_val, test = nonlinear_benchmarks.CED(atleast_2d=True)
data_train_1, data_train_2 = train_val
data_test_1, data_test_2   = test
```

We used the `nonlinear_benchmarks` package to load the data. The user is referred to the [package documentation](GerbenBeintema/nonlinear_benchmarks: The official dataload for http://www.nonlinearbenchmark.org/ (github.com)) to check the details of how to use it.

The following plot detail the training and testing data of both experiments. Here we are trying to get two models, one for each experiment, that have a better performance than the mentioned baselines.

```
plt.plot(data_train_1.u)
plt.plot(data_train_1.y)
plt.title("Experiment 1: training data")
plt.show()

plt.plot(data_test_1.u)
plt.plot(data_test_1.y)
plt.title("Experiment 1: testing data")
plt.show()

plt.plot(data_train_2.u)
plt.plot(data_train_2.y)
plt.title("Experiment 2: training data")
plt.show()

plt.plot(data_test_2.u)
plt.plot(data_test_2.y)
plt.title("Experiment 2: testing data")
plt.show()
```


Experiment 1: training data
2.5
2.0
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
0
50
100
150
200
250
300
350
400


Experiment 1: testing data
2.0
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
0
20
40
60
80
100


Experiment 2: training data
3
2
1
0
−1
0
50
100
150
200
250
300
350
400


Experiment 2: testing data
2.5
2.0
1.5
1.0
0.5
0.0
−0.5
−1.0
0
20
40
60
80
100

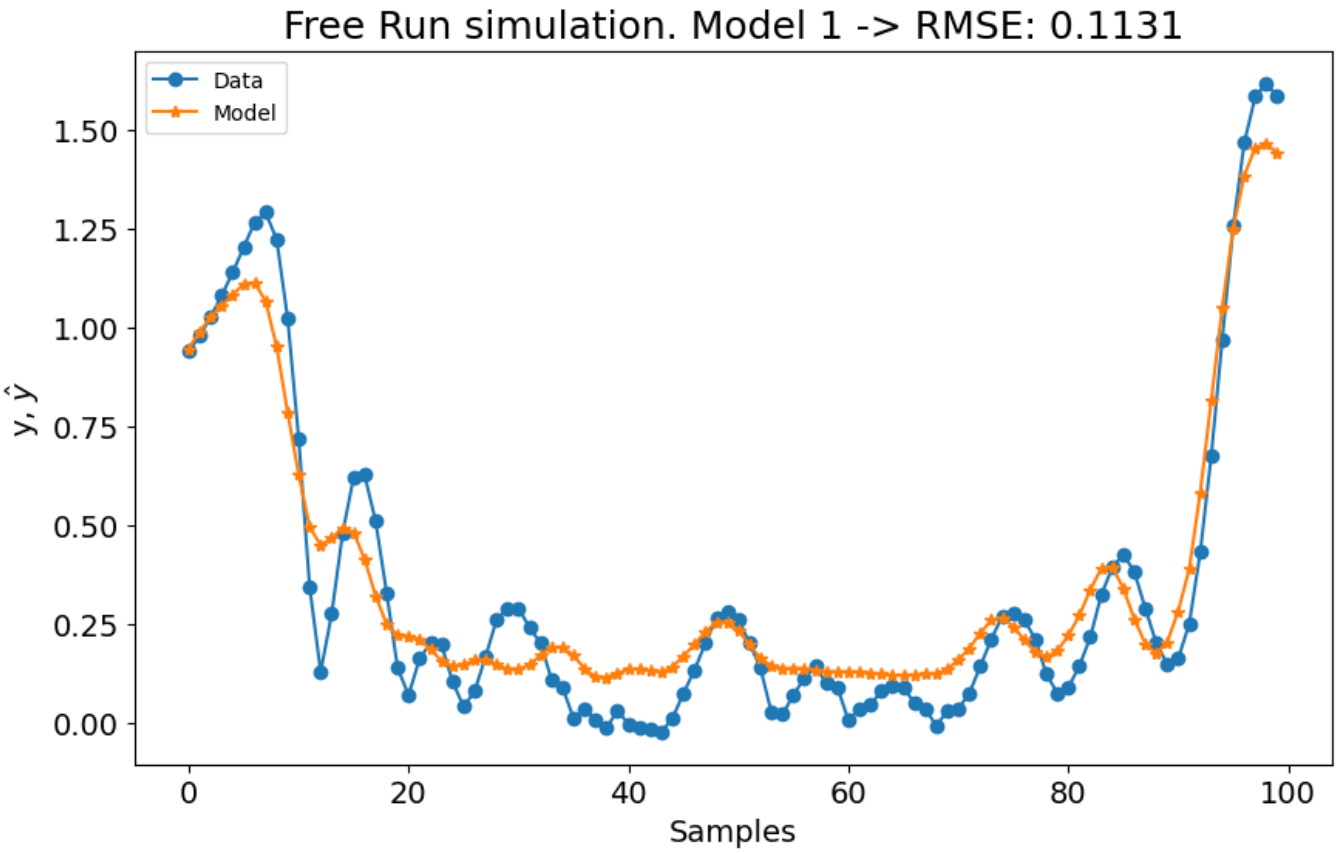
Results

First, we will set the exactly same configuration to built models for both experiments. We can have better models by optimizing the configurations individually, but we will start simple.

A basic configuration of FROLS using a polynomial basis function with degree equal 2 is defined. The information criteria will be the default one, the `aic`. The `xlag` and `ylag` are set to 7 in this first example.

Model for experiment 1:

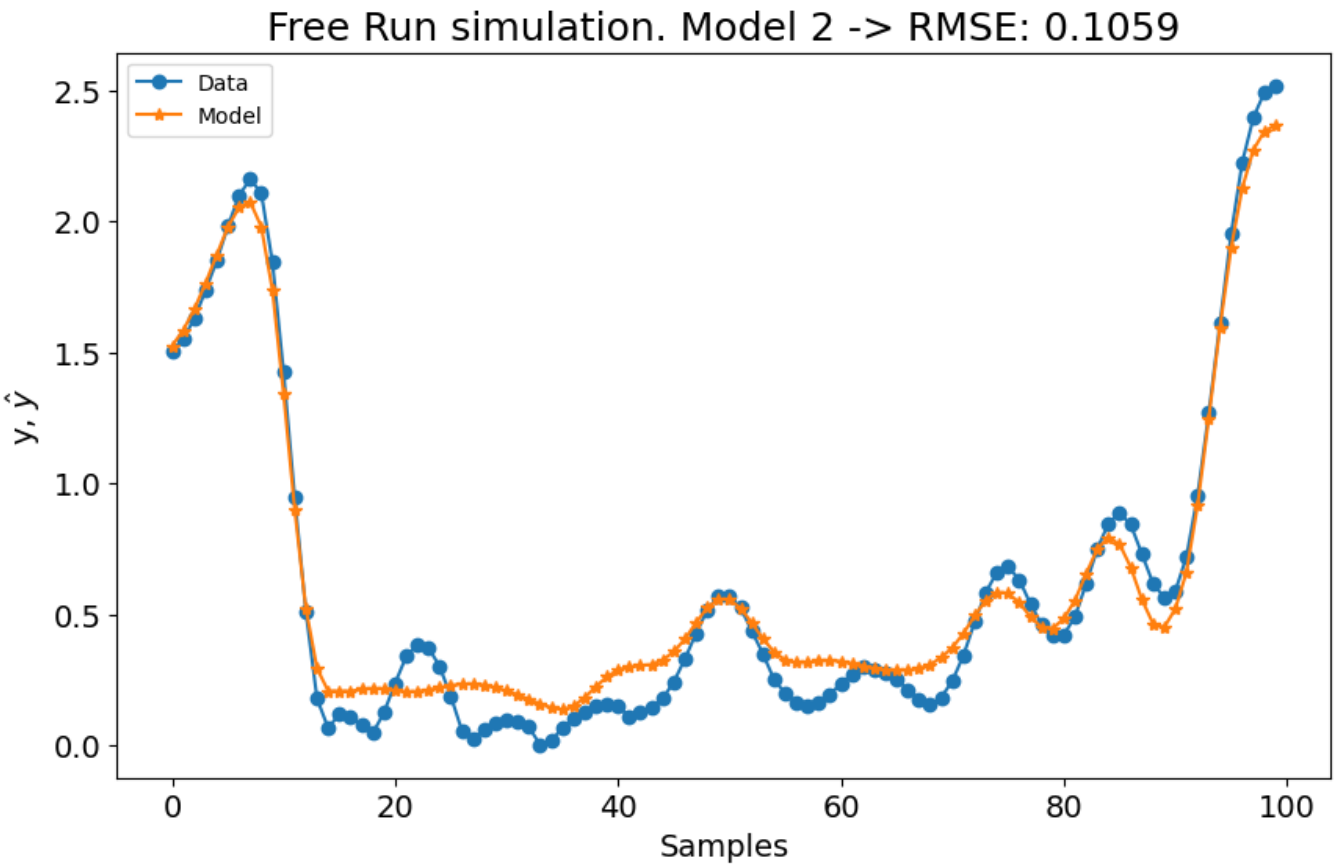
```
y_train = data_train_1.y
y_test = data_test_1.y
x_train = data_train_1.u
x_test = data_test_1.u

n = data_test_1.state_initialization_window_length

basis_function = Polynomial(degree=2)
model = FROLS(
    xlag=7,
    ylag=7,
    basis_function=basis_function,
    estimator=LeastSquares(),
    info_criteria="aic",
    n_info_values=120
)

model.fit(X=x_train, y=y_train)
y_test = np.concatenate([y_train[-model.max_lag:], y_test])
x_test = np.concatenate([x_train[-model.max_lag:], x_test])
yhat = model.predict(X=x_test, y=y_test[:model.max_lag, :])
rmse = root_mean_squared_error(y_test[model.max_lag + n :], yhat[model.max_lag +
n:])
plot_results(y=y_test[model.max_lag :], yhat=yhat[model.max_lag :], n=10000,
title=f"Free Run simulation. Model 1 -> RMSE: {round(rmse, 4)}")
```



Model for experiment 2:

```
y_train = data_train_2.y
y_test = data_test_2.y
x_train = data_train_2.u
x_test = data_test_2.u

n = data_test_2.state_initialization_window_length

basis_function = Polynomial(degree=2)
model = FROLS(
    xlag=7,
    ylag=7,
    basis_function=basis_function,
    estimator=LeastSquares(),
    info_criteria="aic",
    n_info_values=120
)

model.fit(X=x_train, y=y_train)
y_test = np.concatenate([y_train[-model.max_lag:], y_test])
x_test = np.concatenate([x_train[-model.max_lag:], x_test])
yhat = model.predict(X=x_test, y=y_test[:model.max_lag, :])
rmse = root_mean_squared_error(y_test[model.max_lag + n :], yhat[model.max_lag +
n:])
```

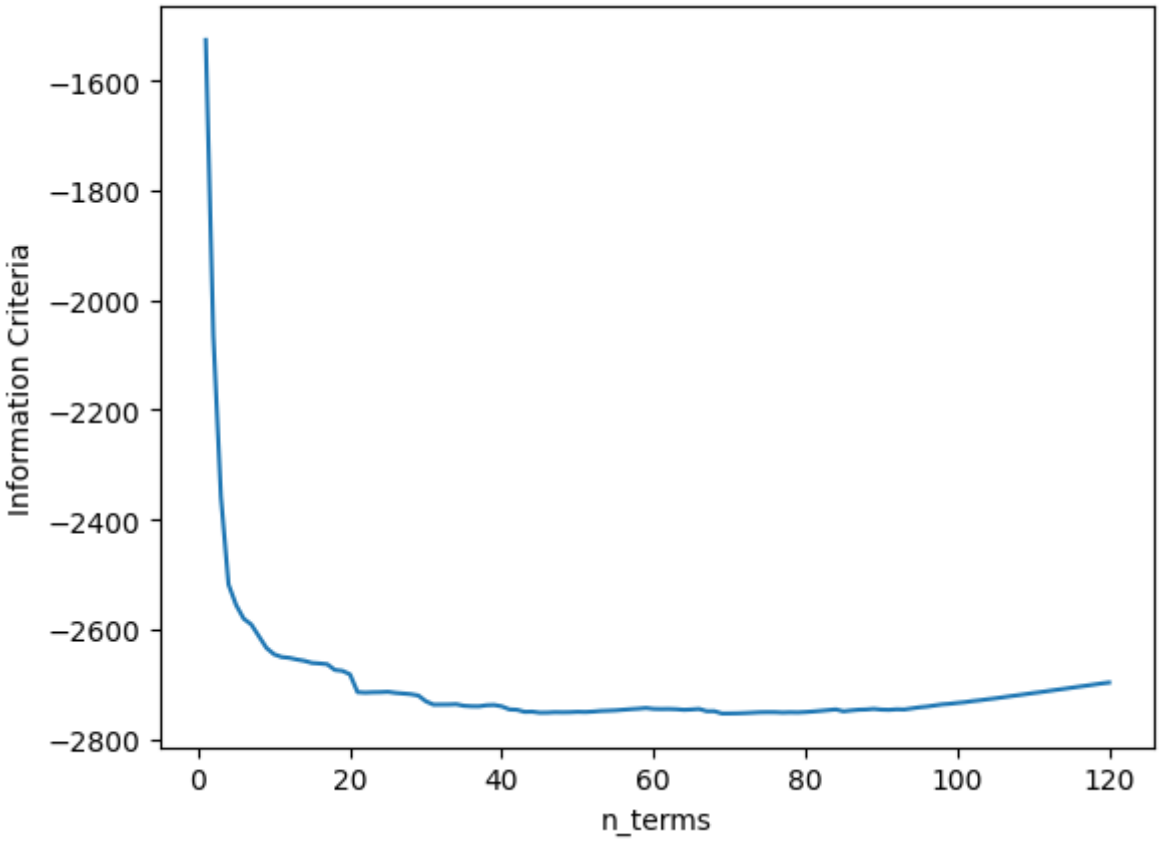
```
plot_results(y=y_test[model.max_lag :], yhat=yhat[model.max_lag :], n=10000,
title=f"Free Run simulation. Model 2 -> RMSE: {round(rmse, 4)}")
```



The first configuration for experiment 1 is already better than the **LTI ARX**, **LTI SS**, **GRU**, **LSTM**, **MLP NARX**, **MLP FIR**, **OLSTM**, and the **SOTA** models shown in the benchmark table. Better than 8 out 11 models shown in the benchmark. For experiment 2, its better than **LTI ARX**, **LTI SS**, **GRU**, **RNN**, **LSTM**, **OLSTM**, and **pNARX** (7 out 11). It's a good start, but let's check if the performance improves if we set a higher lag for both `xlag` and `ylag`.

The average metric is $(0.1131 + 0.1059)/2 = 0.1095$, which is very good, but worse than the SOTA (0.0945). We will now increase the lags for `x` and `y` to check if we get a better model. Before increasing the lags, the information criteria is shown:

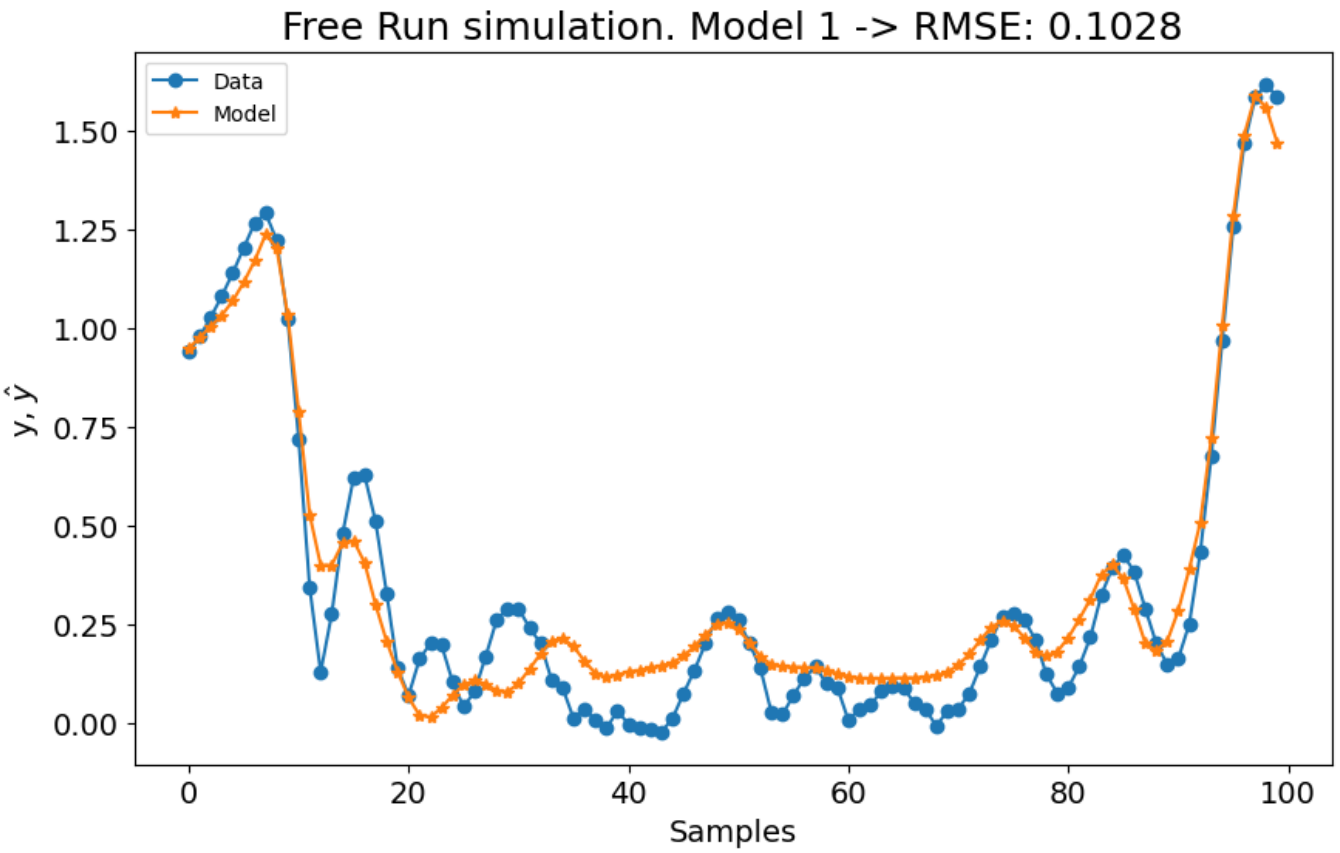
```
xaxis = np.arange(1, model.n_info_values + 1)
plt.plot(xaxis, model.info_values)
plt.xlabel("n_terms")
plt.ylabel("Information Criteria")
```



It can be observed that after 22 regressors, adding new regressors do not improve the model performance (considering the configuration defined for that model). Because we want to try models with higher lags and higher nonlinearity degree, the stopping criteria will be changed to `err_tol` instead of information criteria. This will made the algorithm runs considerably faster.

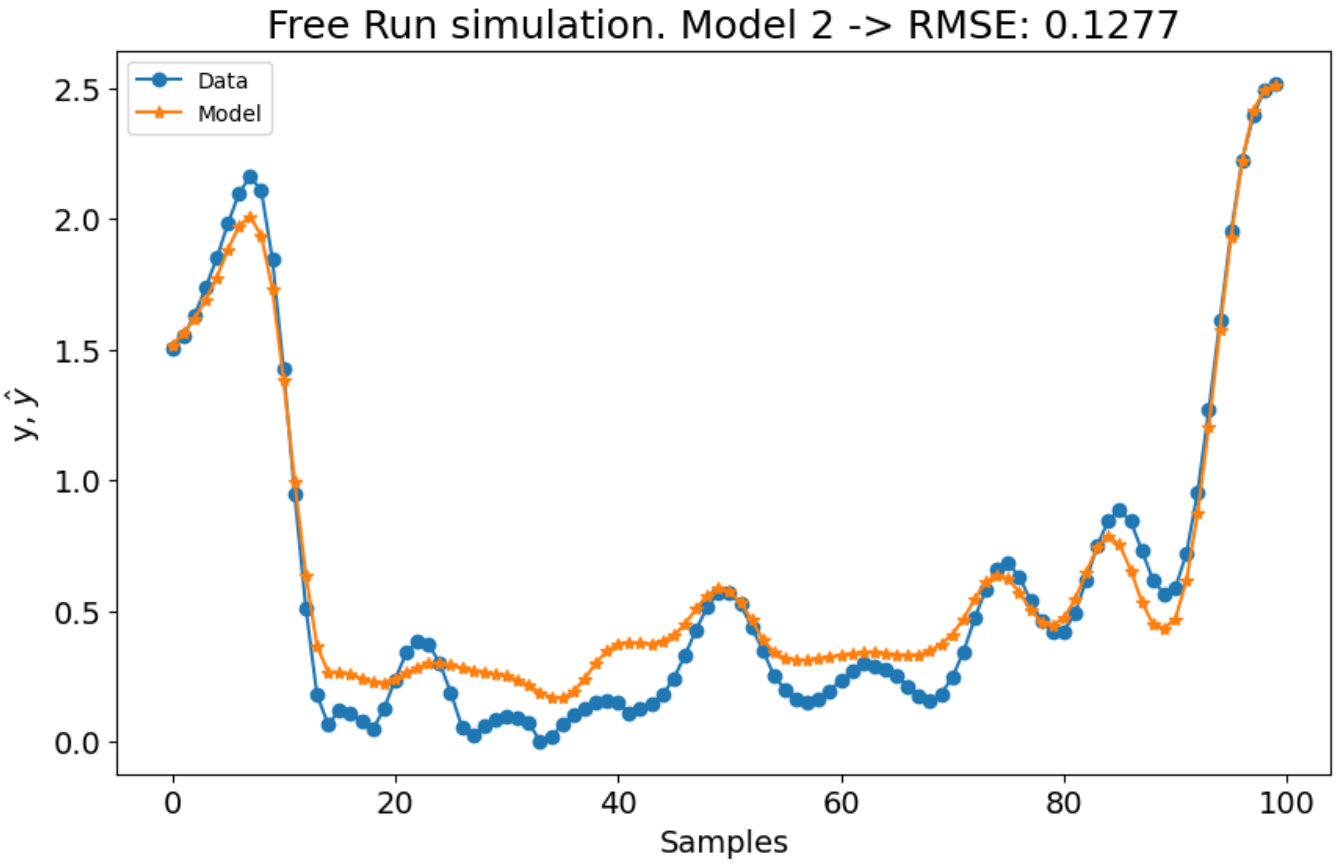
```
# experiment 1
y_train = data_train_1.y
y_test = data_test_1.y
x_train = data_train_1.u
x_test = data_test_1.u

n = data_test_1.state_initialization_window_length

basis_function = Polynomial(degree=2)
model = FROLS(
    xlag=14,
    ylag=14,
    basis_function=basis_function,
    estimator=LeastSquares(),
    err_tol=0.9996,
    n_terms=22,
    order_selection=False
)

model.fit(X=x_train, y=y_train)
```

```
print(model.final_model.shape, model.err.sum())
y_test = np.concatenate([y_train[-model.max_lag:], y_test])
x_test = np.concatenate([x_train[-model.max_lag:], x_test])
yhat = model.predict(X=x_test, y=y_test[:model.max_lag, :])

rmse = root_mean_squared_error(y_test[model.max_lag + n :], yhat[model.max_lag +
n:])

plot_results(y=y_test[model.max_lag :], yhat=yhat[model.max_lag :], n=10000,
title=f"Free Run simulation. Model 1 -> RMSE: {round(rmse, 4)}")
```




```
# experiment 2
y_train = data_train_2.y
y_test = data_test_2.y
x_train = data_train_2.u
x_test = data_test_2.u

n = data_test_2.state_initialization_window_length

basis_function = Polynomial(degree=2)
model = FROLS(
    xlag=14,
    ylag=14,
    basis_function=basis_function,
    estimator=LeastSquares(),
```

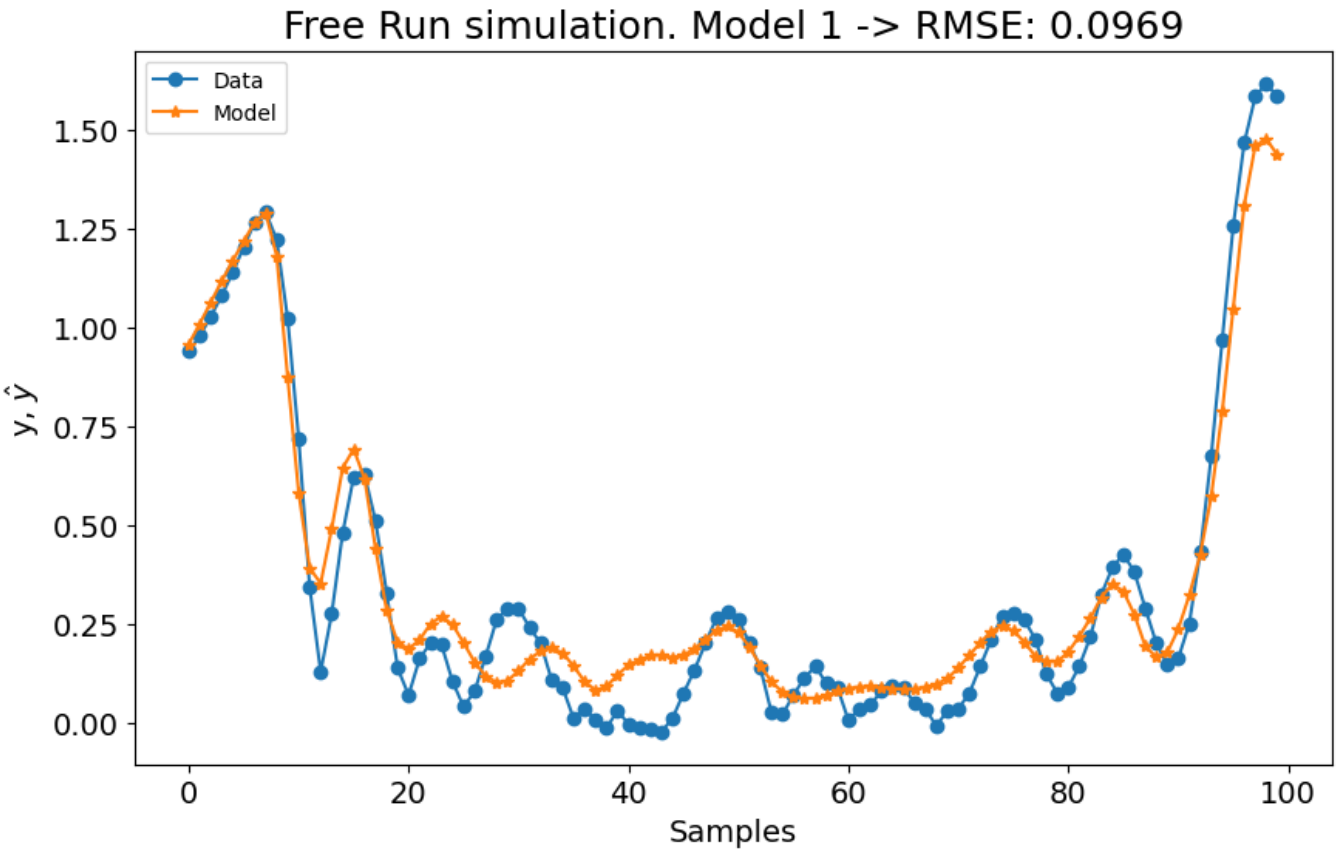
```
    info_criteria="aicc",
    err_tol=0.9996,
    n_terms=22,
    order_selection=False
)

model.fit(X=x_train, y=y_train)
y_test = np.concatenate([y_train[-model.max_lag:], y_test])
x_test = np.concatenate([x_train[-model.max_lag:], x_test])
yhat = model.predict(X=x_test, y=y_test[:model.max_lag, :])

rmse = root_mean_squared_error(y_test[model.max_lag + n :], yhat[model.max_lag +
n:])

plot_results(y=y_test[model.max_lag :], yhat=yhat[model.max_lag :], n=10000,
title=f"Free Run simulation. Model 2 -> RMSE: {round(rmse, 4)}")
```



In the first experiment, the model showed a slight improvement, while the performance of the second experiment experienced a minor decline. Increasing the lag settings with these configurations did not result in significant changes. Therefore, let's set the polynomial degree to 3 and increase the number of terms to build the model to `n_terms=40` if the `err_tol` is not reached. It's important to note that these values are chosen empirically. We could also adjust the parameter estimation technique, the `err_tol`, the model structure selection algorithm, and the basis function, among other factors. Users

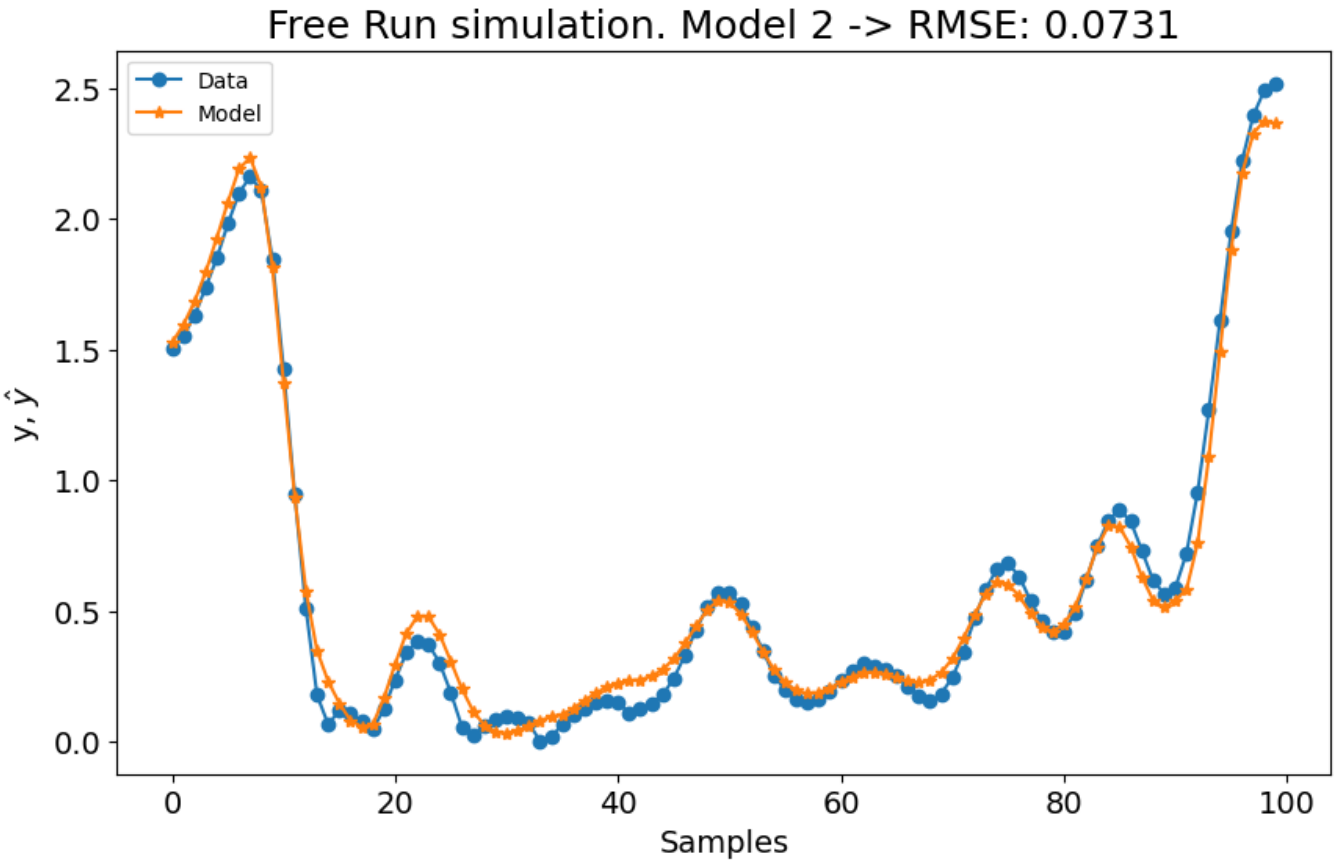
are encouraged to employ hyperparameter tuning techniques to find the optimal combinations of hyperparameters.

```
# experiment 1
y_train = data_train_1.y
y_test = data_test_1.y
x_train = data_train_1.u
x_test = data_test_1.u

n = data_test_1.state_initialization_window_length

basis_function = Polynomial(degree=3)
model = FROLS(
    xlag=14,
    ylag=14,
    basis_function=basis_function,
    estimator=LeastSquares(),
    err_tol=0.9996,
    n_terms=40,
    order_selection=False
)

model.fit(X=x_train, y=y_train)
print(model.final_model.shape, model.err.sum())
y_test = np.concatenate([y_train[-model.max_lag:], y_test])
x_test = np.concatenate([x_train[-model.max_lag:], x_test])
yhat = model.predict(X=x_test, y=y_test[:model.max_lag, :])

rmse = root_mean_squared_error(y_test[model.max_lag + n :], yhat[model.max_lag +
n:])

plot_results(y=y_test[model.max_lag :], yhat=yhat[model.max_lag :], n=10000,
title=f"Free Run simulation. Model 1 -> RMSE: {round(rmse, 4)}")
```




```
# experiment 2
y_train = data_train_2.y
y_test = data_test_2.y
x_train = data_train_2.u
x_test = data_test_2.u

n = data_test_2.state_initialization_window_length

basis_function = Polynomial(degree=3)
model = FROLS(
    xlag=14,
    ylag=14,
    basis_function=basis_function,
    estimator=LeastSquares(),
    info_criteria="aicc",
    err_tol=0.9996,
    n_terms=40,
    order_selection=False
)

model.fit(X=x_train, y=y_train)
y_test = np.concatenate([y_train[-model.max_lag:], y_test])
x_test = np.concatenate([x_train[-model.max_lag:], x_test])
yhat = model.predict(X=x_test, y=y_test[:model.max_lag, :])

rmse = root_mean_squared_error(y_test[model.max_lag + n :], yhat[model.max_lag +
```

```
n:])

plot_results(y=y_test[model.max_lag :], yhat=yhat[model.max_lag :], n=10000,
title=f"Free Run simulation. Model 2 -> RMSE: {round(rmse, 4)}")
```



As shown in the plot, we have surpassed the state-of-the-art (SOTA) results with an average metric of $(0.0969 + 0.0731)/2 = 0.0849$. Additionally, the metric for the first experiment matches the best model in the benchmark, and the metric for the second experiment slightly exceeds the benchmark's best model. Using the same configuration for both models, we achieved the best overall results!

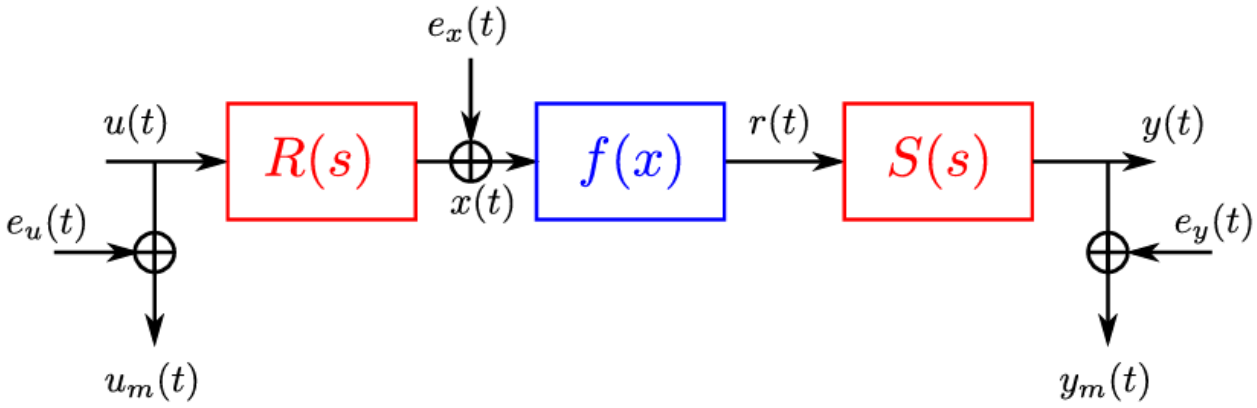
Wiener-Hammerstein

The description content primarily derives from the [benchmark website](Nonlinear Benchmark) and [associated paper](Wiener-Hammerstein benchmark with process noise (dataset) (4tu.nl)). For a detailed description, readers are referred to the linked references.

> The nonlinear benchmark website stands as a significant contribution to the system identification and machine learning community. The users are encouraged to explore all the papers referenced on the site.

This benchmark focuses on a Wiener-Hammerstein electronic circuit where process noise plays a significant role in distorting the output signal.

The Wiener-Hammerstein structure is a well-known block-oriented system which contains a static nonlinearity sandwiched between two Linear Time-Invariant (LTI) blocks (Figure 2). This arrangement presents a challenging identification problem due to the presence of these LTI blocks.



> Figure 2: the Wiener-Hammerstein system

In Figure 2, the Wiener-Hammerstein system is illustrated with process noise $e_x(t)$ entering before the static nonlinearity $f(x)$, sandwiched between LTI blocks represented by $R(s)$ and $S(s)$ at the input and output, respectively. Additionally, small, negligible noise sources $e_u(t)$ and $e_y(t)$ affect the measurement channels. The measured input and output signals are denoted as $u_m(t)$ and $y_m(t)$.

The first LTI block $R(s)$ is effectively modeled as a third-order lowpass filter. The second LTI subsystem $S(s)$ is configured as an inverse Chebyshev filter with a stop-band attenuation of $40dB$ and a cutoff frequency of $5kHz$. Notably, $S(s)$ includes a transmission zero within the operational frequency range, complicating its inversion.

The static nonlinearity $f(x)$ is implemented using a diode-resistor network, resulting in saturation nonlinearity. Process noise $e_x(t)$ is introduced as filtered white Gaussian noise, generated from a discrete-time third-order lowpass Butterworth filter followed by zero-order hold and analog low-pass reconstruction filtering with a cutoff of $20kHz$.

Measurement noise sources $e_u(t)$ and $e_y(t)$ are minimal compared to $e_x(t)$. The system's inputs and process noise are generated using an Arbitrary Waveform Generator (AWG), specifically the Agilent/HP E1445A, sampling at $78125Hz$, synchronized with an acquisition system (Agilent/HP E1430A) to ensure phase coherence and prevent leakage errors. Buffering between the acquisition cards and the system's inputs and outputs minimizes measurement equipment distortion.

The benchmark provides two standard test signals through the benchmarking website: a random phase multisine and a sine-sweep signal. Both signals have an rms value of $0.71Vrms$ and cover frequencies from DC to $15kHz$ (excluding DC). The sine-sweep spans this frequency range at a rate of $4.29MHz/min$. These test sets serve as targets for evaluating the model's performance, emphasizing accurate representation under varied conditions.

The Wiener-Hammerstein benchmark highlights three primary nonlinear system identification challenges:

1. **Process Noise:** Significant in the system, influencing output fidelity.
2. **Static Nonlinearity:** Indirectly accessible from measured data, posing identification challenges.
3. **Output Dynamics:** Complex inversion due to transmission zero presence in $S(s)$.

The goal of this benchmark is to develop and validate robust models using separate estimation data, ensuring accurate characterization of the Wiener-Hammerstein system's behavior.

Required Packages and Versions

To ensure that you can replicate this case study, it is essential to use specific versions of the required packages. Below is a list of the packages along with their respective versions needed for running the case studies effectively.

To install all the required packages, you can create a `requirements.txt` file with the following content:

```
sysidentpy==0.4.0
pandas==2.2.2
numpy==1.26.0
matplotlib==3.8.4
nonlinear_benchmarks==0.1.2
```

Then, install the packages using:

```
pip install -r requirements.txt
```

- Ensure that you use a virtual environment to avoid conflicts between package versions.
- Versions specified are based on compatibility with the code examples provided. If you are using different versions, some adjustments in the code might be necessary.

SysIdentPy configuration

In this section, we will demonstrate the application of SysIdentPy to the Wiener-Hammerstein system dataset. The following code will guide you through the process of loading the dataset, configuring the SysIdentPy parameters, and building a model for Wiener-Hammerstein system.

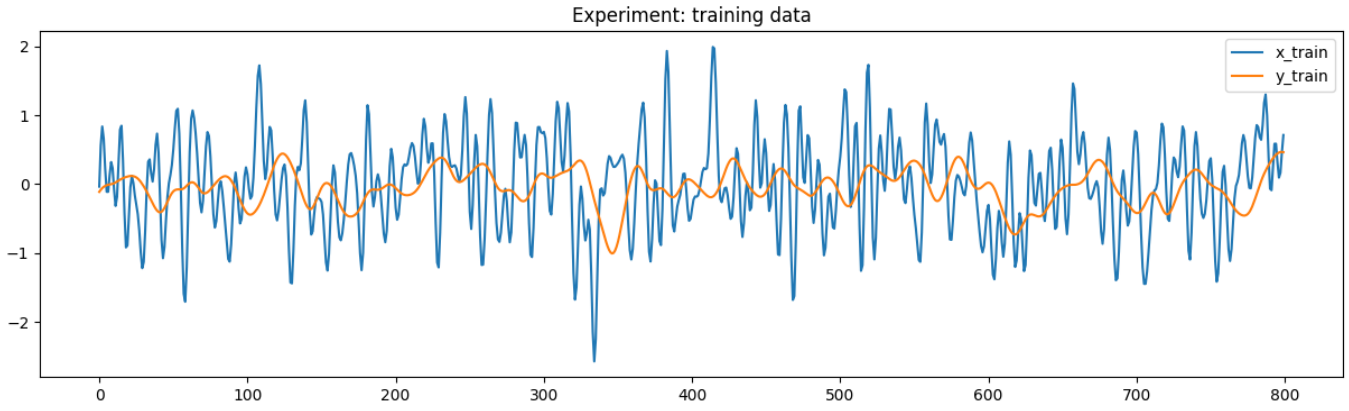
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

from sysidentpy.model_structure_selection import FROLS, AOLS, MetaMSS
```

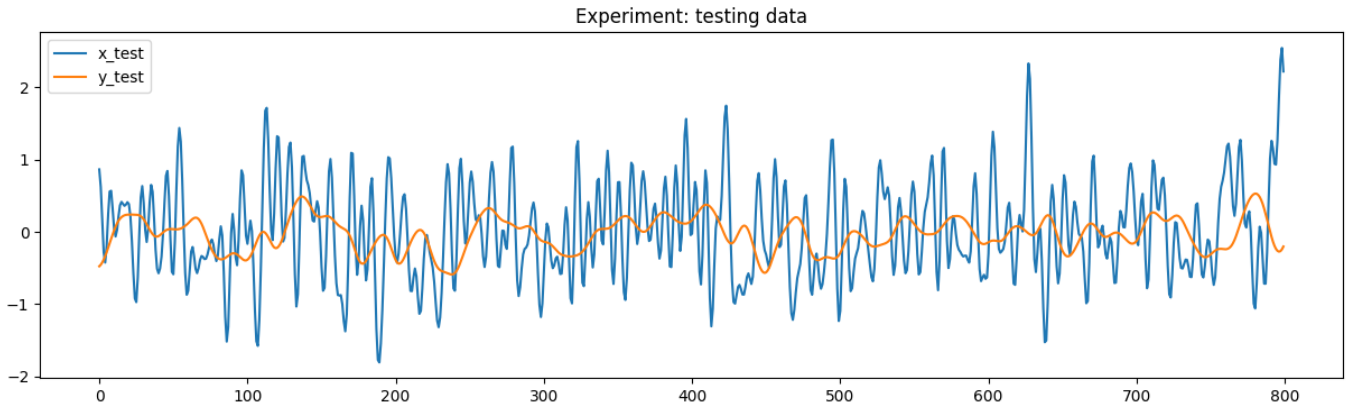
```
from sysidentpy.basis_function import Polynomial, Fourier
from sysidentpy.utils.display_results import results
from sysidentpy.parameter_estimation import LeastSquares,
BoundedVariableLeastSquares, NonNegativeLeastSquares, LeastSquaresMinimalResidual

from sysidentpy.metrics import root_mean_squared_error
from sysidentpy.utils.plotting import plot_results

import nonlinear_benchmarks

train_val, test = nonlinear_benchmarks.WienerHammerBenchMark(atleast_2d=True)
x_train, y_train = train_val
x_test, y_test = test
```

We used the `nonlinear_benchmarks` package to load the data. The user is referred to the [package documentation](GerbenBeintema/nonlinear_benchmarks: The official dataload for http://www.nonlinearbenchmark.org/ (github.com)) to check the details of how to use it.

The following plot detail the training and testing data of the experiment.

```
plot_n = 800

plt.figure(figsize=(15, 4))
plt.plot(x_train[:plot_n])
plt.plot(y_train[:plot_n])
plt.title("Experiment: training data")
plt.legend(["x_train", "y_train"])
plt.show()

plt.figure(figsize=(15, 4))
plt.plot(x_test[:plot_n])
plt.plot(y_test[:plot_n])
plt.title("Experiment: testing data")
plt.legend(["x_test", "y_test"])
plt.show()
```





The goal of this benchmark it to get a model that have a better performance than the SOTA model provided in the benchmarking paper.

Reference Figure of Merit (Unit)	Silverbox RMS (mV)			W-H RMS (mV)	EMPS RMS (mm)	Cascaded Tanks RMS (V)	CED RMS (ticks/s)	
Test Set	multisine	arrow (full)	arrow (no extrap.)	test	test	test	test 1	test 2
DT subnet (Beintema et al., 2021)	0.36	1.4	0.32	0.241	-	0.37	0.169	0.117
CT subnet (Beintema et al., 2023)	-	-	-	-	4.61	0.306	0.115	0.074
Grey-box (Worden et al., 2018)	-	-	-	-	-	0.191	-	-
PNLSS (Paduart et al., 2010) (Paduart et al., 2012; Relan et al., 2017)	-	0.26	-	0.42	-	0.45	-	-
TSEM (Forgione and Piga, 2021a)	-	-	-	-	-	0.33	-	-
SCI (Forgione and Piga, 2021a)	-	-	-	-	-	0.4	-	-
dynoNet (Forgione and Piga, 2021b)	-	-	-	0.452	2.64	-	-	-

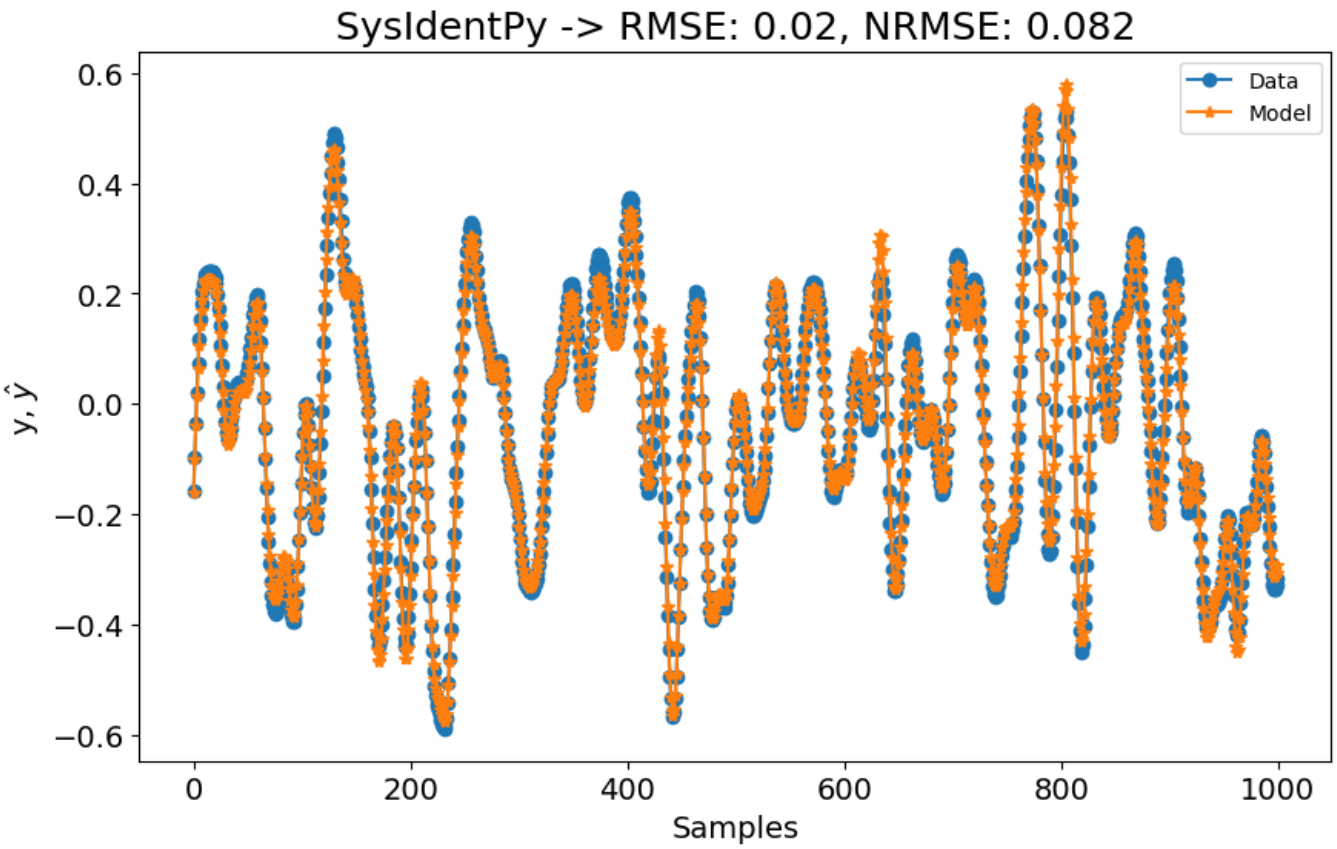
> State of the art results presented in the [benchmarking paper](2405.10779 (arxiv.org)). In this section we are only working with the Wiener-Hammerstein results, which are presented in the $W - H$ column.

Results

We will start with a basic configuration of FROLS using a polynomial basis function with degree equal 2. The `xlag` and `ylag` are set to 7 in this first example. Because the dataset is considerably large, we will start with `n_info_values=50`. This means the FROLS algorithm will not include all regressors when calculating the information criteria used to determine the model order. While this approach might result in a sub-optimal model, it is a reasonable starting point for our first attempt.

```
n = test.state_initialization_window_length

basis_function = Polynomial(degree=2)
model = FROLS(
    xlag=7,
    ylag=7,
    basis_function=basis_function,
    estimator=LeastSquares(unbiased=False),
    n_info_values=50,
)
```

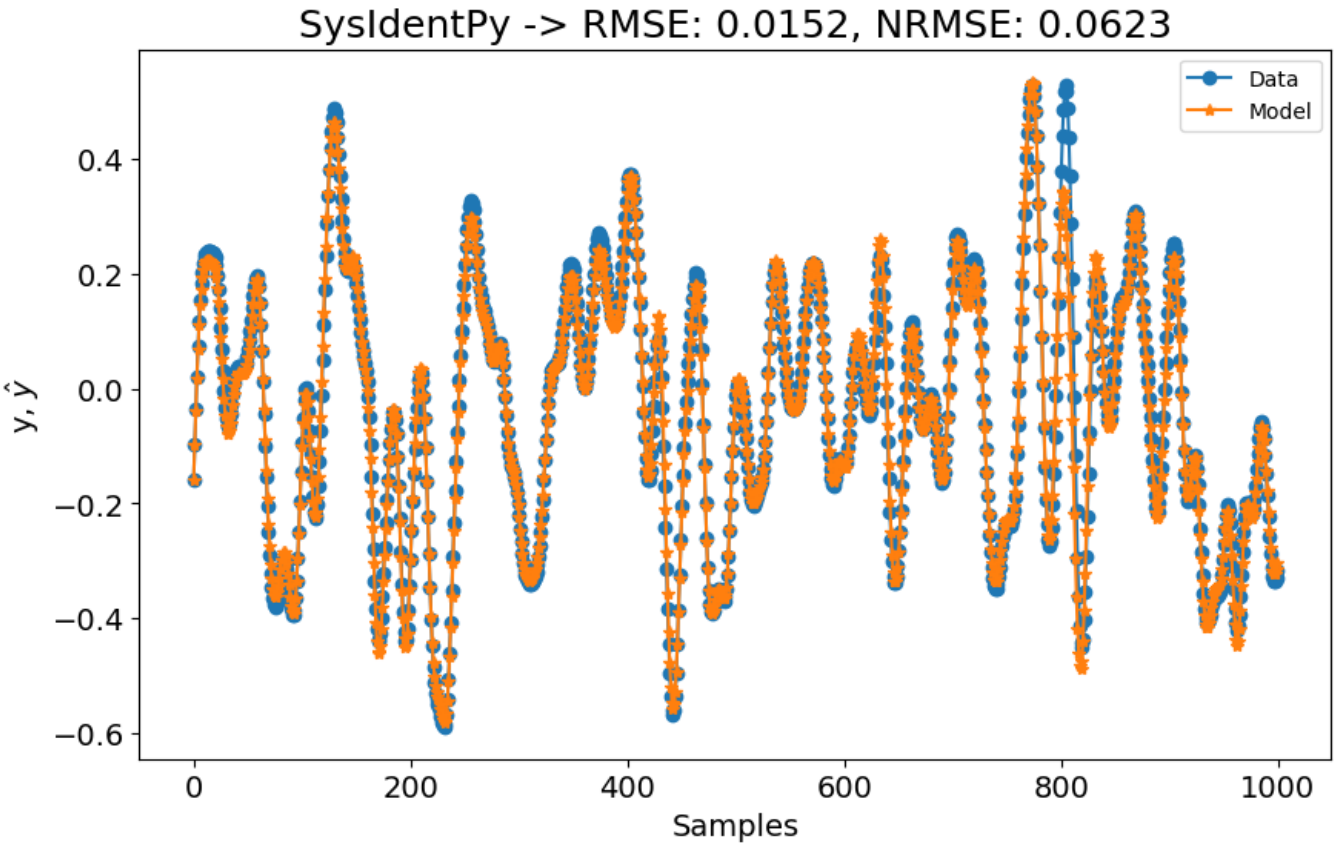
```
model.fit(X=x_train, y=y_train)
y_test = np.concatenate([y_train[-model.max_lag:], y_test])
x_test = np.concatenate([x_train[-model.max_lag:], x_test])
yhat = model.predict(X=x_test, y=y_test[:model.max_lag, :])
rmse = root_mean_squared_error(y_test[model.max_lag + n :], yhat[model.max_lag +
n:])
rmse_sota = rmse/y_test.std()
plot_results(y=y_test[model.max_lag :], yhat=yhat[model.max_lag :], n=1000,
title=f"SysIdentPy -> RMSE: {round(rmse, 4)}, NRMSE: {round(rmse_sota, 4)}")
```



The first configuration is already better than the **SOTA** models shown in the benchmark table! We started using `xlag=ylag=7` to have an idea of how well SysIdentPy would handle this dataset, but the results are pretty good already! However, the benchmarking paper indicates that they used higher lags for their models. Let's check what happens if we set `xlag=ylag=10`.

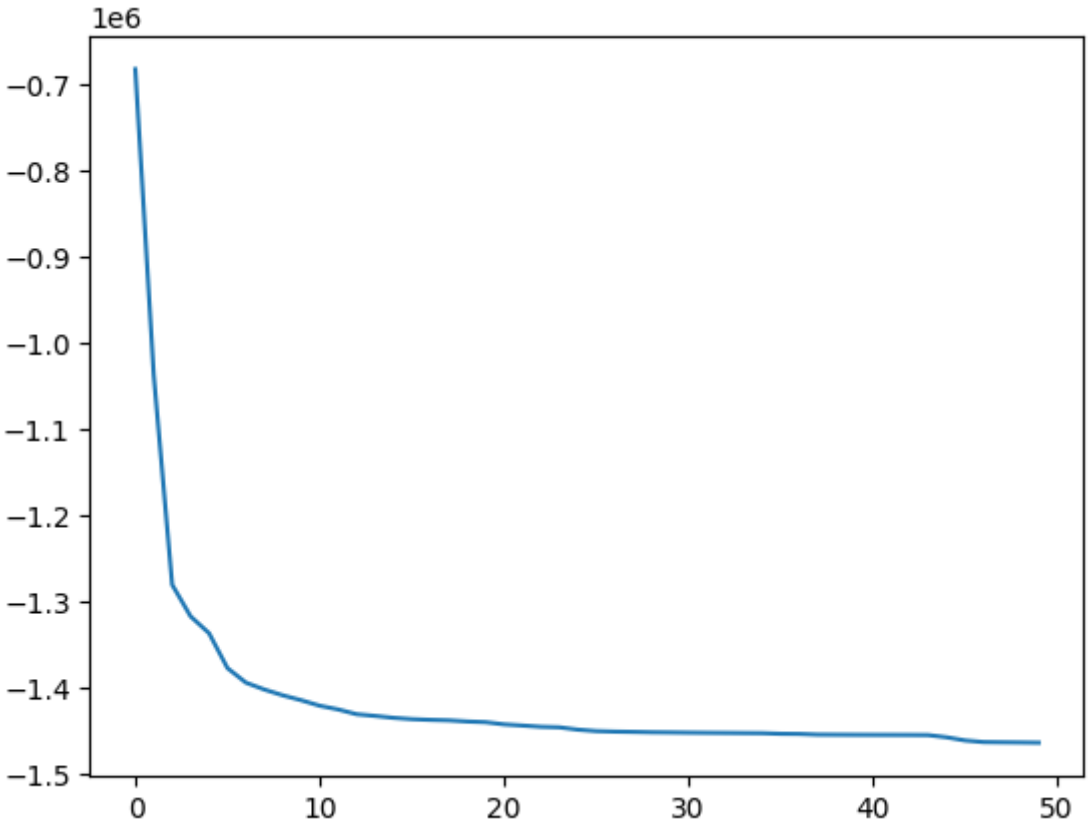
```
x_train, y_train = train_val
x_test, y_test = test

n = test.state_initialization_window_length

basis_function = Polynomial(degree=2)
model = FROLS(
    xlag=10,
    ylag=10,
```

```
    basis_function=basis_function,
    estimator=LeastSquares(unbiased=False),
    n_info_values=50,
)

model.fit(X=x_train, y=y_train)
y_test = np.concatenate([y_train[-model.max_lag:], y_test])
x_test = np.concatenate([x_train[-model.max_lag:], x_test])
yhat = model.predict(X=x_test, y=y_test[:model.max_lag, :])
rmse = root_mean_squared_error(y_test[model.max_lag + n :], yhat[model.max_lag +
n:])
rmse_sota = rmse/y_test.std()
plot_results(y=y_test[model.max_lag :], yhat=yhat[model.max_lag :], n=1000,
title=f"SysIdentPy -> RMSE: {round(rmse, 4)}, NRMSE: {round(rmse_sota, 4)}")
```



The performance is even better now! For now, we are not worried about the model complexity (even in this case where we are comparing to a deep state neural network...). However, if we check the model order and the `AIC` plot, we see that the model have 50 regressors , but the `AIC` values do not change much after each added regression.

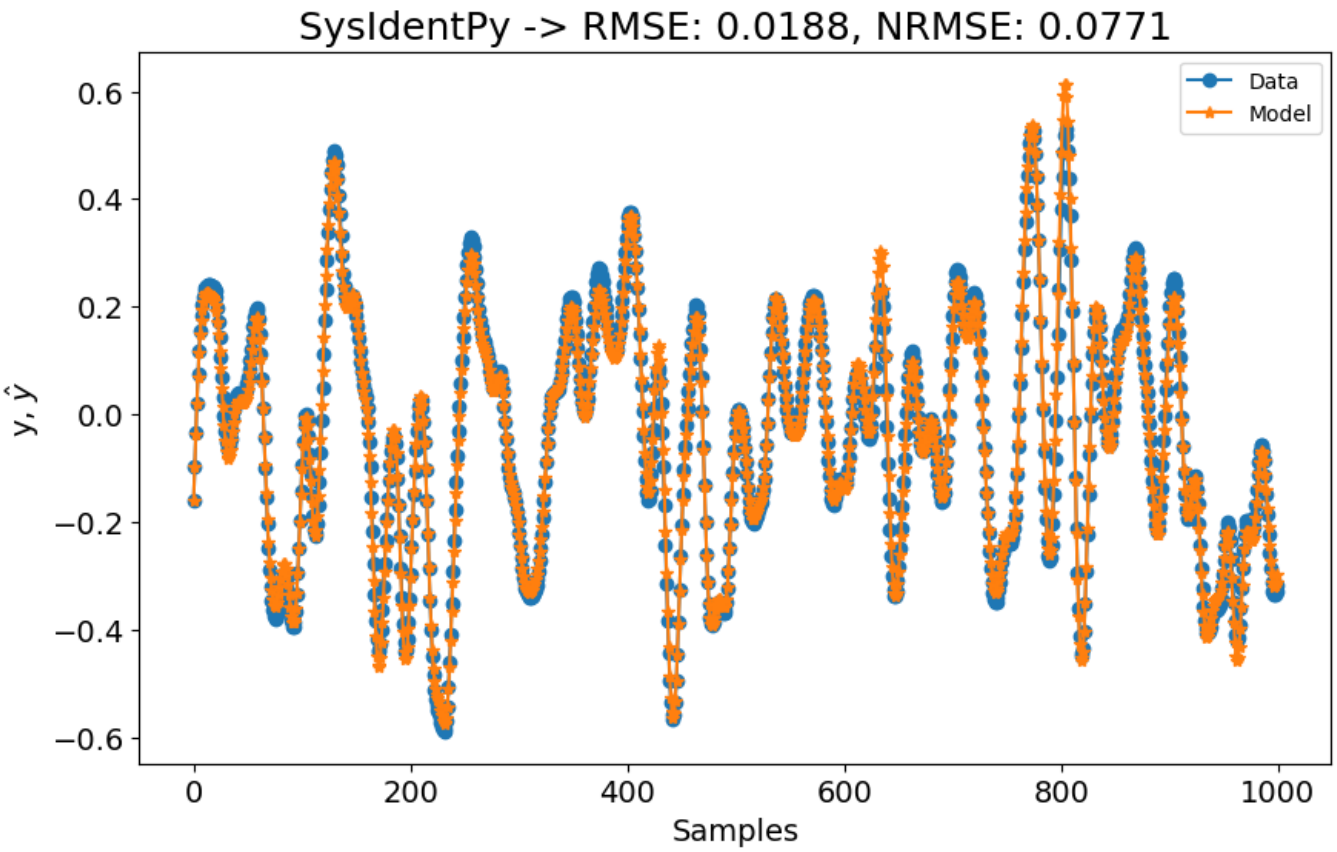
```
plt.plot(model.info_values)
```



So, what happens if we set a model with half of the regressors?

```
x_train, y_train = train_val
x_test, y_test = test

n = test.state_initialization_window_length

basis_function = Polynomial(degree=2)
model = FROLS(
    xlag=10,
    ylag=10,
    basis_function=basis_function,
    estimator=LeastSquares(unbiased=False),
    n_info_values=50,
    n_terms=25,
    order_selection=False
)

model.fit(X=x_train, y=y_train)
y_test = np.concatenate([y_train[-model.max_lag:], y_test])
x_test = np.concatenate([x_train[-model.max_lag:], x_test])
yhat = model.predict(X=x_test, y=y_test[:model.max_lag, :])
rmse = root_mean_squared_error(y_test[model.max_lag + n :], yhat[model.max_lag +
n:])
rmse_sota = rmse/y_test.std()
```

```
plot_results(y=y_test[model.max_lag :], yhat=yhat[model.max_lag :], n=1000,
title=f"SysIdentPy -> RMSE: {round(rmse, 4)}, NRMSE: {round(rmse_sota, 4)}")
```



As shown in the figure above, the results still outperform the SOTA models presented in the benchmarking paper. The SOTA results from the paper could likely be improved as well. Users are encouraged to explore the deepsysid package, which can be used to build deep state neural networks.

This basic configuration can serve as a starting point for users to develop even better models using SysIdentPy. Give it a try!

Air Passenger Demand Forecasting - A Benchmarking

In this case study, we explore the capabilities of SysIdentPy by applying it to the Air Passenger dataset, a classic time series dataset widely used for evaluating time series forecasting methods. The primary goal of this analysis is to demonstrate that SysIdentPy can serve as a strong alternative for time series modeling, rather than to assert that one library is superior to another.

Dataset Overview

The Air Passenger dataset consists of monthly totals of international airline passengers from 1949 to 1960. This dataset is characterized by its strong seasonal patterns, trend components, and variability,

making it an ideal benchmark for evaluating various time series forecasting methods. Specifically, the dataset includes:

- **Total Monthly Passengers**: The number of passengers (in thousands) for each month.
- **Time Period**: From January 1949 to December 1960, providing 144 data points.

The dataset exhibits clear seasonal fluctuations and a trend, which poses a significant challenge for forecasting methods. It serves as a well-known benchmark for assessing the performance of different time series models due to its inherent complexity and well-documented behavior.

Comparison with Other Libraries

We will compare the performance of SysIdentPy with other popular time series modeling libraries, focusing on the following tools:

- **sktime**: An extensive library for time series analysis in Python, offering various modeling techniques. For this case study, we will use:
  - `AutoARIMA`: Automatically selects the best ARIMA model based on the data.
  - `BATS` (Bayesian Structural Time Series): A model that captures complex seasonal patterns and trends.
  - `TBATS` (Trigonometric, Box-Cox, ARMA, Trend, and Seasonal): A model designed to handle multiple seasonal patterns.
  - `Exponential Smoothing`: A method that applies weighted averages to forecast future values.
  - `Prophet`: Developed by Facebook, it is particularly effective for capturing seasonality and holiday effects.
  - `AutoETS` (Automatic Exponential Smoothing): Selects the best exponential smoothing model for the data.
- **SysIdentPy**: A library designed for system identification and time series modeling. We will focus on:
  - `MetaMSS` (Meta-heuristic Model Structure Selection): Uses metaheuristic algorithms to select the best model structure.
  - `AOLS` (Accelerated Orthogonal Least Squares): A method for selecting relevant regressors in a model.
  - `FROLS` (Forward Regression with Orthogonal Least Squares, using polynomial base functions): A regression technique for model structure selection with polynomial terms.
  - `NARXNN` (Nonlinear Auto-Regressive model with Exogenous Inputs using Neural Networks): A flexible method for modeling nonlinear time series with external inputs.

Objective

The objective of this case study is to evaluate and compare the performance of these methods on the Air Passenger dataset. We aim to assess how well each library handles the complex seasonal and trend components of the data and to showcase SysIdentPy as a viable option for time series forecasting.

Required Packages and Versions

To ensure that you can replicate this case study, it is essential to use specific versions of the required packages. Below is a list of the packages along with their respective versions needed for running the case studies effectively.

To install all the required packages, you can create a `requirements.txt` file with the following content:

```
sysidentpy==0.4.0
pystan==2.19.1.1
holidays==0.11.2
fbprophet==0.7.1
neuralprophet==0.2.7
pandas==1.3.2
numpy==1.23.3
matplotlib==3.8.4
pmdarima==1.8.3
scikit-learn==0.24.2
scipy==1.9.1
sktime==0.8.0
statsmodels==0.12.2
tbats==1.1.0
torch==1.12.1
```

Then, install the packages using:

```
pip install -r requirements.txt
```

- Ensure that you use a virtual environment to avoid conflicts between package versions. This practice isolates your project's dependencies and prevents version conflicts with other projects or system-wide packages. Additionally, be aware that some packages, such as `sktime` and `neuralprophet`, may install several dependencies automatically during their installation. Setting up a virtual environment helps manage these dependencies more effectively and keeps your project environment clean and reproducible.
- Versions specified are based on compatibility with the code examples provided. If you are using

different versions, some adjustments in the code might be necessary.

Let's begin by importing the necessary packages and setting up the environment for this analysis.

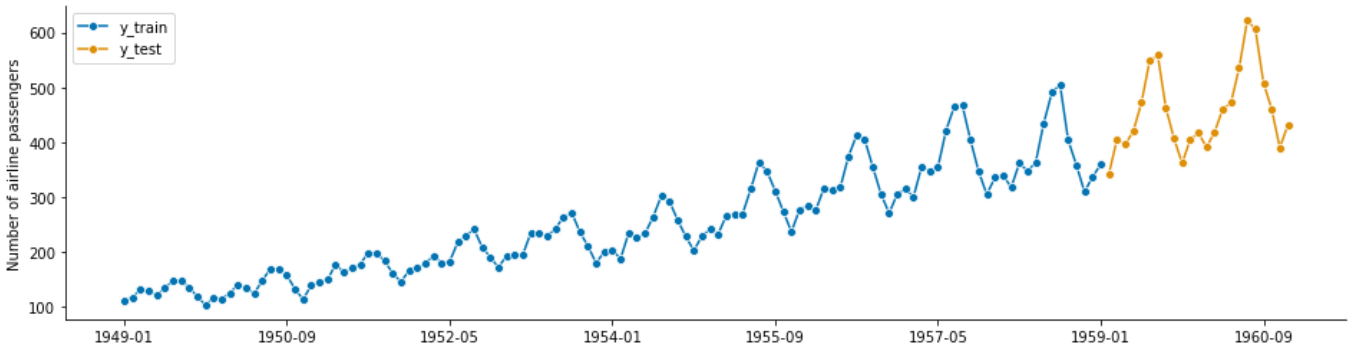
```
from warnings import simplefilter
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import scipy.signal.signaltools

def _centered(arr, newsize):
    # Return the center newsize portion of the array.
    # this is needed due a conflict error using the versions of the packages defined
    # for this example
    newsize = np.asarray(newsize)
    currsize = np.array(arr.shape)
    startind = (currsize - newsize) // 2
    endind = startind + newsize
    myslice = [slice(startind[k], endind[k]) for k in range(len(endind))]
    return arr[tuple(myslice)]


scipy.signal.signaltools._centered = _centered
from sysidentpy.model_structure_selection import FROLS
from sysidentpy.model_structure_selection import AOLS
from sysidentpy.model_structure_selection import MetaMSS
from sysidentpy.basis_function import Polynomial
from sysidentpy.utils.plotting import plot_results
from torch import nn
from sysidentpy.neural_network import NARXNN

from sktime.datasets import load_airline
from sktime.forecasting.ets import AutoETS
from sktime.forecasting.arima import ARIMA, AutoARIMA
from sktime.forecasting.base import ForecastingHorizon
from sktime.forecasting.exp_smoothing import ExponentialSmoothing
from sktime.forecasting.fbprophet import Prophet
from sktime.forecasting.tbats import TBATS
from sktime.forecasting.bats import BATS
from sktime.forecasting.model_selection import temporal_train_test_split
from sktime.performance_metrics.forecasting import mean_squared_error
from sktime.utils.plotting import plot_series

from neuralprophet import NeuralProphet
from neuralprophet import set_random_seed

simplefilter("ignore", FutureWarning)
np.seterr(all="ignore")
```

```
%matplotlib inline
loss = mean_squared_error
```

We use the `sktime` method to load the data. Besides, 23 samples is used as test data, following the definitions in the `sktime` examples.

```
y = load_airline()
y_train, y_test = temporal_train_test_split(y, test_size=23)  # 23 samples for
testing
plot_series(y_train, y_test, labels=["y_train", "y_test"])
fh = ForecastingHorizon(y_test.index, is_relative=False)
print(y_train.shape[0], y_test.shape[0])
```

The following image shows the data of the system to be modeled.



Results

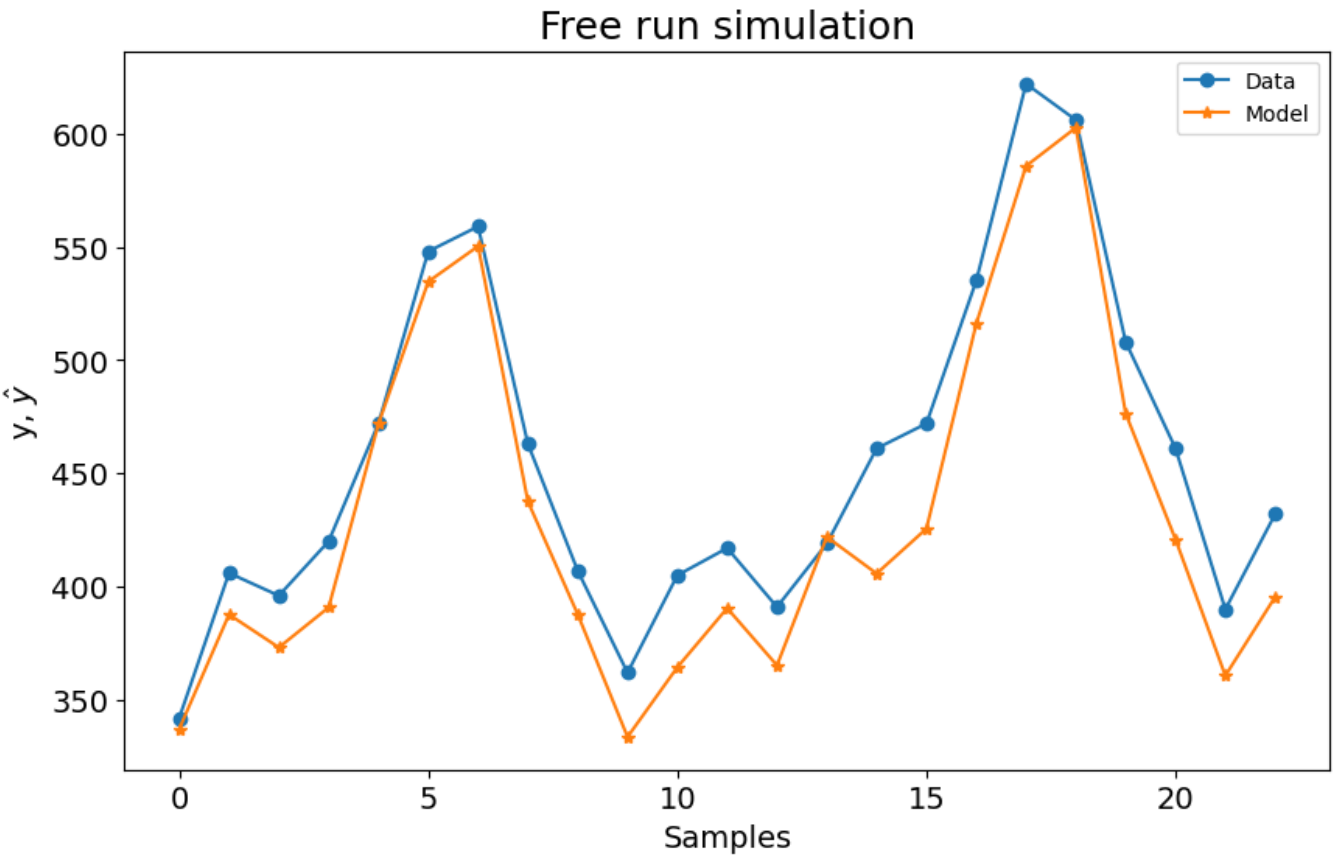
Because we have several different models to test, the results are summarized in the following table. The user you will see that no hyperparameter tuning was made for SysIdentPy model. The idea here is to show how simple it can be to build good models in SysIdentPy.

No.	Package	Mean Squared Error
1	SysIdentPy (Neural Model)	316.54
2	SysIdentPy (MetaMSS)	450.99
3	SysIdentPy (AOLS)	476.64
4	NeuralProphet	501.24
5	SysIdentPy (FROLS)	805.95
6	Exponential Smoothing	910.52
7	Prophet	1186.00
8	AutoArima	1714.47
9	Manual Arima	2085.42

No.	Package	Mean Squared Error
10	ETS	2590.05
11	BATS	7286.64
12	TBATS	7448.43

SysIdentPy: FROLS

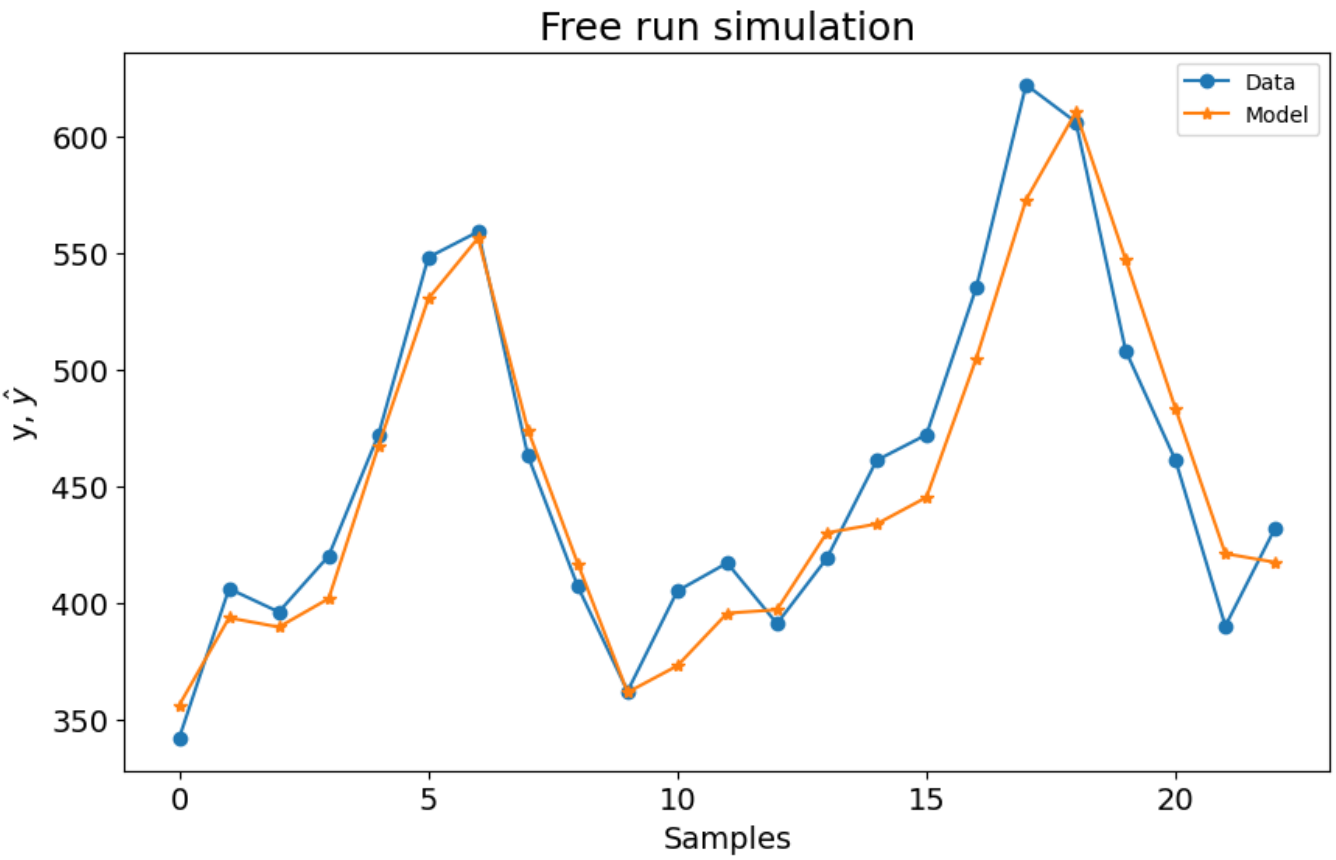
```
y = load_airline()
y_train, y_test = temporal_train_test_split(y, test_size=23)
y_train = y_train.values.reshape(-1, 1)
y_test = y_test.values.reshape(-1, 1)
basis_function = Polynomial(degree=1)
sysidentpy = FROLS(
    order_selection=True,
    ylag=13,  # the lags for all models will be 13
    basis_function=basis_function,
    model_type="NAR",
)

sysidentpy.fit(y=y_train)
y_test = np.concatenate([y_train[-sysidentpy.max_lag :], y_test])
yhat = sysidentpy.predict(y=y_test, forecast_horizon=23)
frols_loss = loss(
    pd.Series(y_test.flatten()[sysidentpy.max_lag :]),
    pd.Series(yhat.flatten()[sysidentpy.max_lag :]),
)
print(frols_loss)
plot_results(y=y_test[sysidentpy.max_lag :], yhat=yhat[sysidentpy.max_lag :])
>>> 805.95
```



SysIdentPy: AOLS

```
y = load_airline()
y_train, y_test = temporal_train_test_split(y, test_size=23)
y_train = y_train.values.reshape(-1, 1)
y_test = y_test.values.reshape(-1, 1)
df_train, df_test = temporal_train_test_split(y, test_size=23)
df_train = df_train.reset_index()
df_train.columns = ["ds", "y"]
df_train["ds"] = pd.to_datetime(df_train["ds"].astype(str))
df_test = df_test.reset_index()
df_test.columns = ["ds", "y"]
df_test["ds"] = pd.to_datetime(df_test["ds"].astype(str))

sysidentpy_AOLS = AOLS(
    ylag=13, k=2, L=1, model_type="NAR", basis_function=basis_function
)

sysidentpy_AOLS.fit(y=y_train)
y_test = np.concatenate([y_train[-sysidentpy_AOLS.max_lag :], y_test])
yhat = sysidentpy_AOLS.predict(y=y_test, steps_ahead=None, forecast_horizon=23)

aols_loss = loss(
    pd.Series(y_test.flatten()[sysidentpy_AOLS.max_lag :]),
```

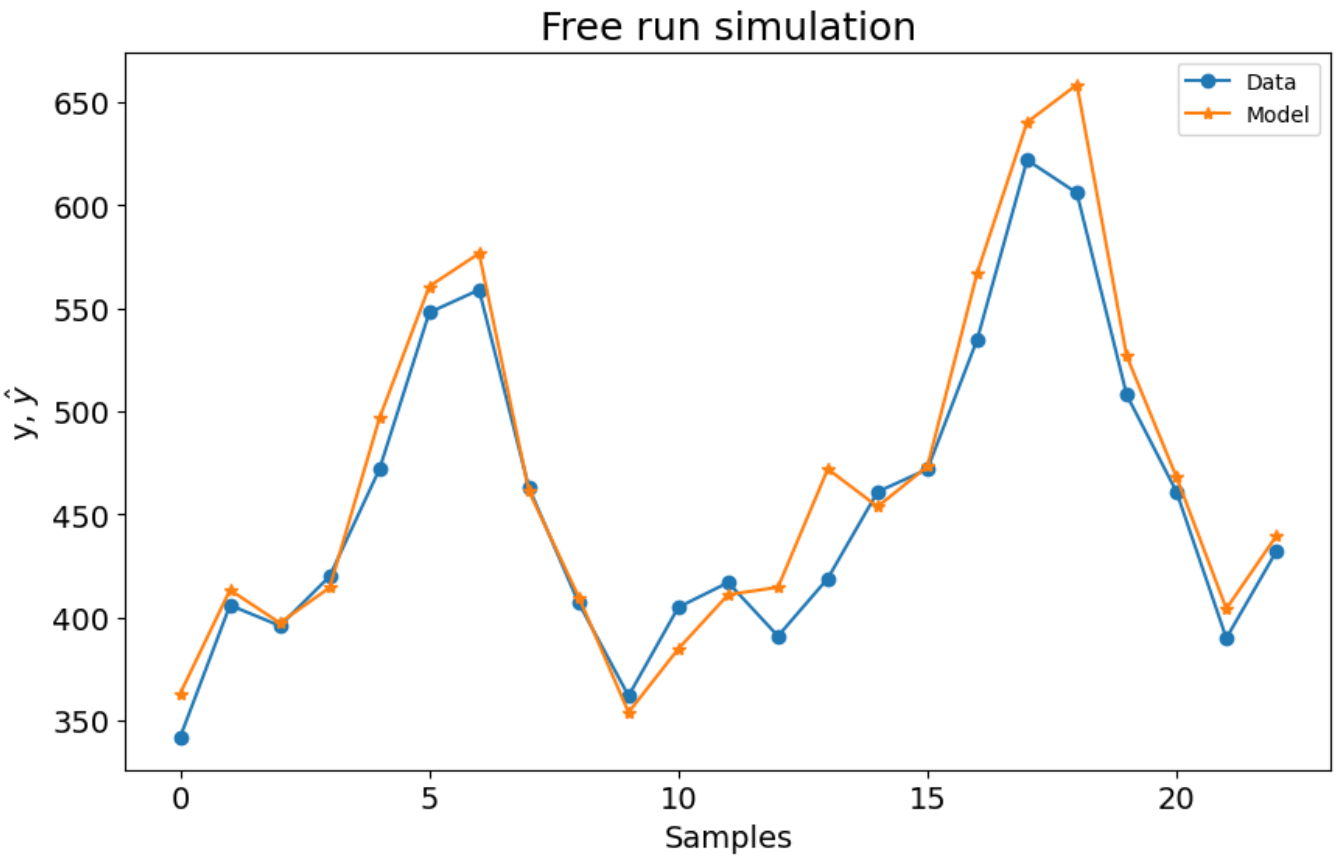
```
    pd.Series(yhat.flatten()[sysidentpy_AOLS.max_lag :]),
)

print(aols_loss)
plot_results(y=y_test[sysidentpy_AOLS.max_lag :], yhat=yhat[sysidentpy_AOLS.max_lag
:])
>>> 476.64
```



SysIdentPy: MetaMSS

```
set_random_seed(42)
y = load_airline()
y_train, y_test = temporal_train_test_split(y, test_size=23)
y_train = y_train.values.reshape(-1, 1)
y_test = y_test.values.reshape(-1, 1)

sysidentpy_metamss = MetaMSS(
    basis_function=basis_function, ylag=13, model_type="NAR", test_size=0.17
)

sysidentpy_metamss.fit(y=y_train)

y_test = np.concatenate([y_train[-sysidentpy_metamss.max_lag :], y_test])
```

```
yhat = sysidentpy_metamss.predict(y=y_test, steps_ahead=None, forecast_horizon=23)

metamss_loss = loss(
    pd.Series(y_test.flatten()[sysidentpy_metamss.max_lag :]),
    pd.Series(yhat.flatten()[sysidentpy_metamss.max_lag :]),
)

print(metamss_loss)
plot_results(
    y=y_test[sysidentpy_metamss.max_lag :], yhat=yhat[sysidentpy_metamss.max_lag :]
)

>>> 450.99
```



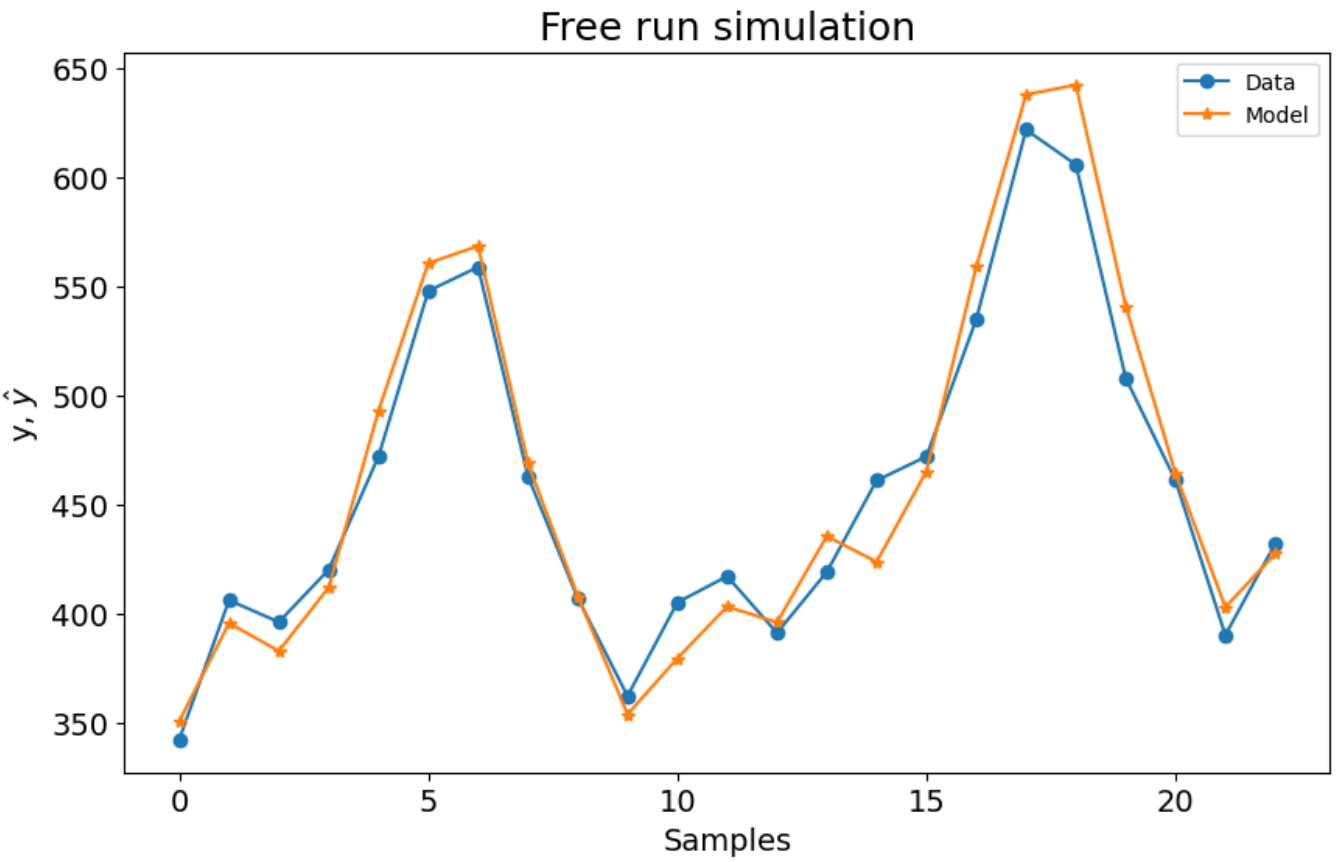
SysIdentPy: Neural NARX

The network architecture is just the same as the one used in to show how to build a Neural NARX model in SysIdentPy docs.

```
import torch
torch.manual_seed(42)

y = load_airline()
```

```
# the split here will use 36 as test size just because the network will use the
first values as initial conditions. It could be done like the others methods by
concatenating the values
y_train, y_test = temporal_train_test_split(y, test_size=36)
y_train = y_train.values.reshape(-1, 1)
y_test = y_test.values.reshape(-1, 1)
x_train = np.zeros_like(y_train)
x_test = np.zeros_like(y_test)

class NARX(nn.Module):
    def __init__(self):
        super().__init__()
        self.lin = nn.Linear(13, 20)
        self.lin2 = nn.Linear(20, 20)
        self.lin3 = nn.Linear(20, 20)
        self.lin4 = nn.Linear(20, 1)
        self.relu = nn.ReLU()


    def forward(self, xb):
        z = self.lin(xb)
        z = self.relu(z)
        z = self.lin2(z)
        z = self.relu(z)
        z = self.lin3(z)
        z = self.relu(z)
        z = self.lin4(z)
        return z

narx_net = NARXNN(
    net=NARX(),
    ylag=13,
    model_type="NAR",
    basis_function=Polynomial(degree=1),
    epochs=900,
    verbose=False,
    learning_rate=2.5e-02,
    optim_params={},  # optional parameters of the optimizer
)

narx_net.fit(y=y_train)
yhat = narx_net.predict(y=y_test, forecast_horizon=23)

narxnet_loss = loss(
    pd.Series(y_test.flatten()[narx_net.max_lag :]),
    pd.Series(yhat.flatten()[narx_net.max_lag :]),
)
```

```
print(narxnet_loss)
plot_results(y=y_test[narx_net.max_lag :], yhat=yhat[narx_net.max_lag :])
```



sktime models

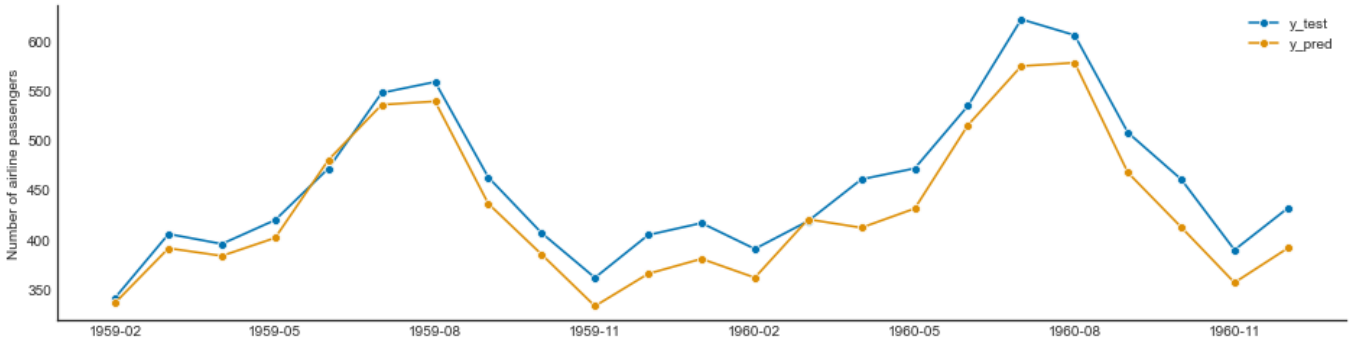
The following models are the ones available in the **sktime** package.

```
y = load_airline()
y_train, y_test = temporal_train_test_split(y, test_size=23)  # 23 samples for
testing
plot_series(y_train, y_test, labels=["y_train", "y_test"])
fh = ForecastingHorizon(y_test.index, is_relative=False)
```

sktime: Exponential Smoothing

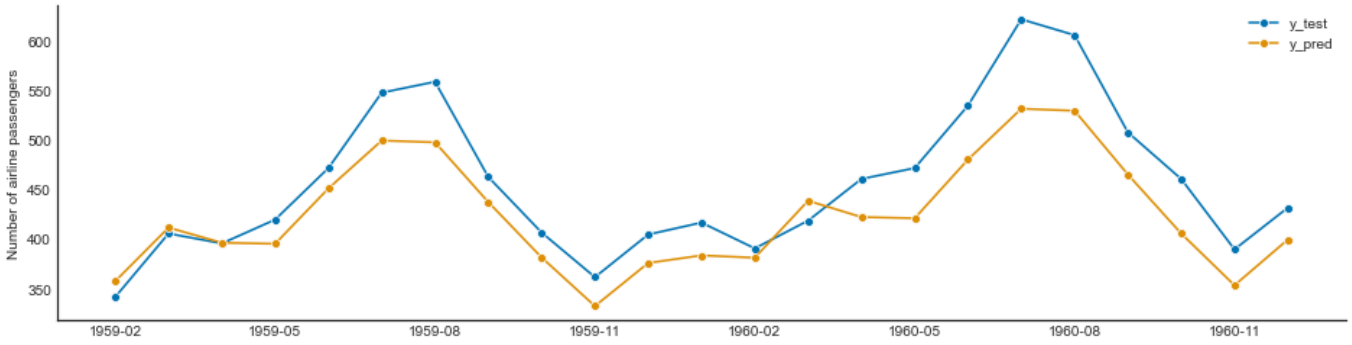
```
es = ExponentialSmoothing(trend="add", seasonal="multiplicative", sp=12)
y = load_airline()
y_train, y_test = temporal_train_test_split(y, test_size=23)
es.fit(y_train)
y_pred_es = es.predict(fh)
plot_series(y_test, y_pred_es, labels=["y_test", "y_pred"])
```

```
es_loss = loss(y_test, y_pred_es)
es_loss
>>> 910.46
```



sktime: AutoETS

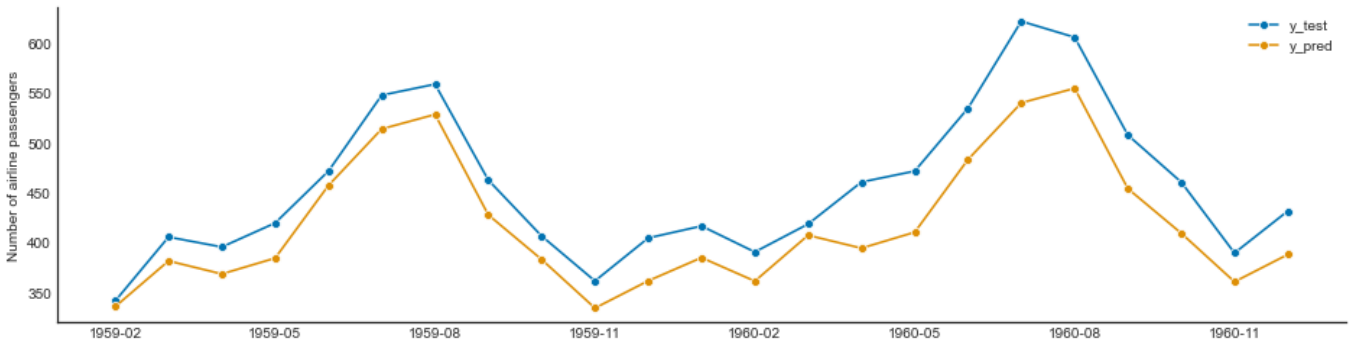
```
y = load_airline()
y_train, y_test = temporal_train_test_split(y, test_size=23)
ets = AutoETS(auto=True, sp=12, n_jobs=-1)
ets.fit(y_train)
y_pred_ets = ets.predict(fh)
plot_series(y_test, y_pred_ets, labels=["y_test", "y_pred"])
ets_loss = loss(y_test, y_pred_ets)
ets_loss
>>> 1739.11
```



sktime: AutoArima

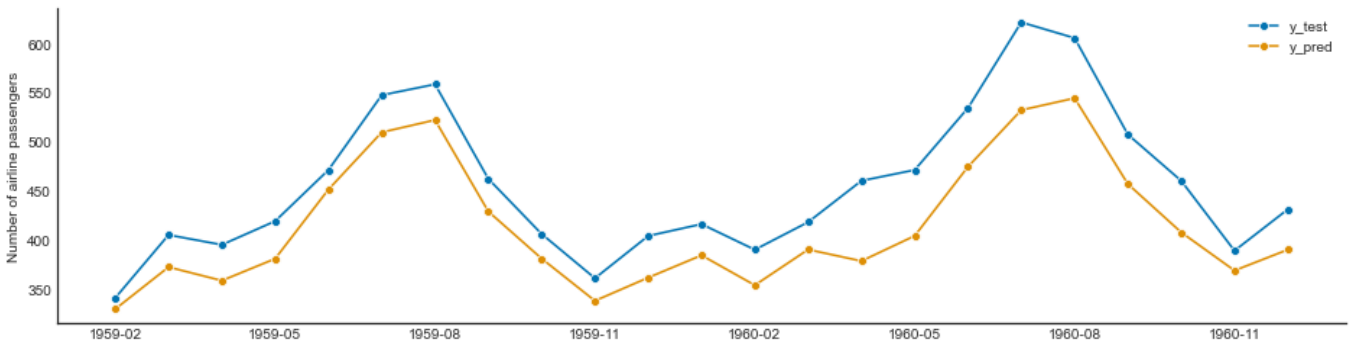
```
y = load_airline()
y_train, y_test = temporal_train_test_split(y, test_size=23)

auto_arima = AutoARIMA(sp=12, suppress_warnings=True)
auto_arima.fit(y_train)
y_pred_auto_arima = auto_arima.predict(fh)
```

```
plot_series(y_test, y_pred_auto_arima, labels=["y_test", "y_pred"])
autoarima_loss = loss(y_test, y_pred_auto_arima)
autoarima_loss
>>> 1714.47
```



sktime: Arima

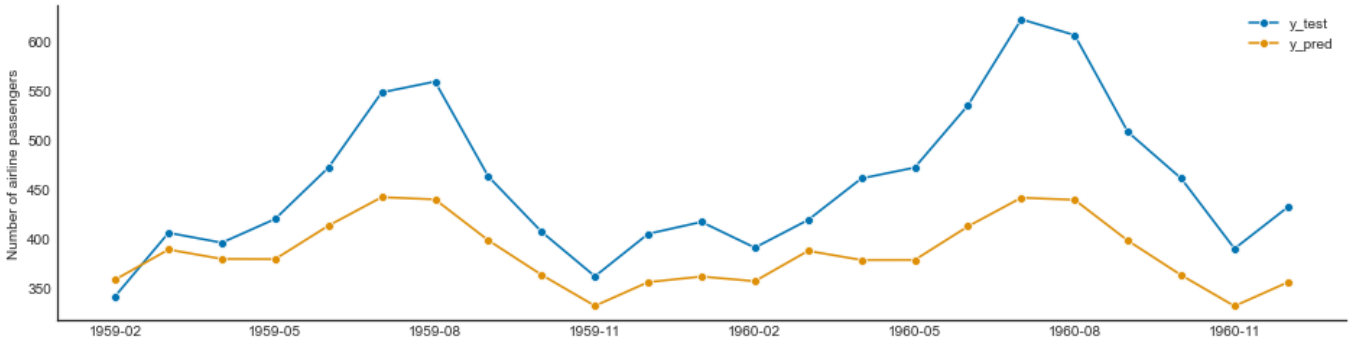
```
y = load_airline()
y_train, y_test = temporal_train_test_split(y, test_size=23)
manual_arima = ARIMA(
    order=(13, 1, 0), suppress_warnings=True
)  # seasonal_order=(0, 1, 0, 12)
manual_arima.fit(y_train)
y_pred_manual_arima = manual_arima.predict(fh)
plot_series(y_test, y_pred_manual_arima, labels=["y_test", "y_pred"])
manualarima_loss = loss(y_test, y_pred_manual_arima)
manualarima_loss
>>> 2085.42
```



sktime: BATS

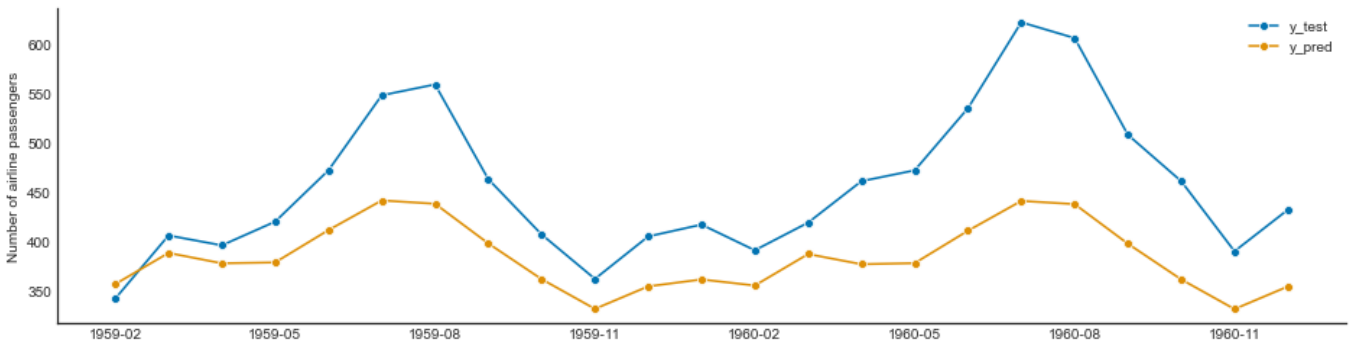
```
y = load_airline()
y_train, y_test = temporal_train_test_split(y, test_size=23)
bats = BATS(sp=12, use_trend=True, use_box_cox=False)
```

```
bats.fit(y_train)
y_pred_bats = bats.predict(fh)
plot_series(y_test, y_pred_bats, labels=["y_test", "y_pred"])
bats_loss = loss(y_test, y_pred_bats)
bats_loss
>>> 7286.64
```



sktime: TBATS

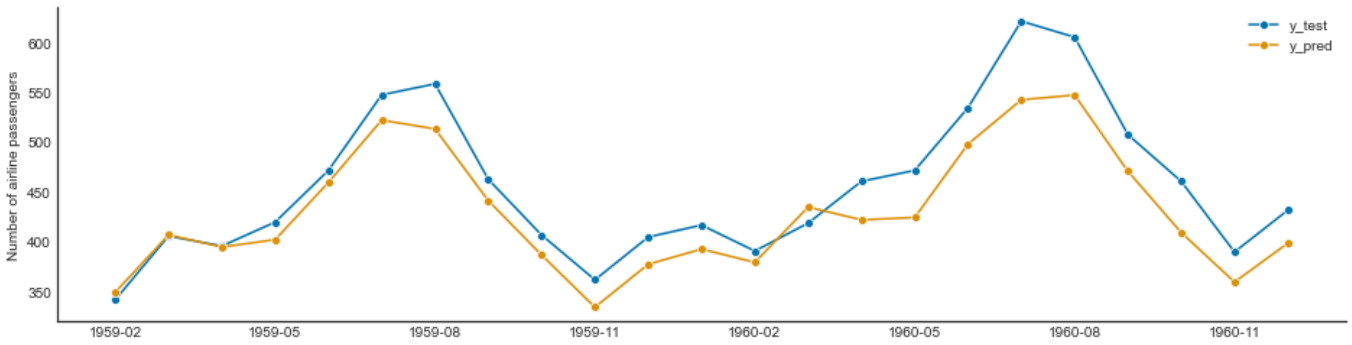
```
y = load_airline()
y_train, y_test = temporal_train_test_split(y, test_size=23)
tbats = TBATS(sp=12, use_trend=True, use_box_cox=False)
tbats.fit(y_train)
y_pred_tbats = tbats.predict(fh)
plot_series(y_test, y_pred_tbats, labels=["y_test", "y_pred"])
tbats_loss = loss(y_test, y_pred_tbats)
tbats_loss
>>> 7448.43
```



sktime: Prophet

```
set_random_seed(42)
y = load_airline()
y_train, y_test = temporal_train_test_split(y, test_size=23)
```

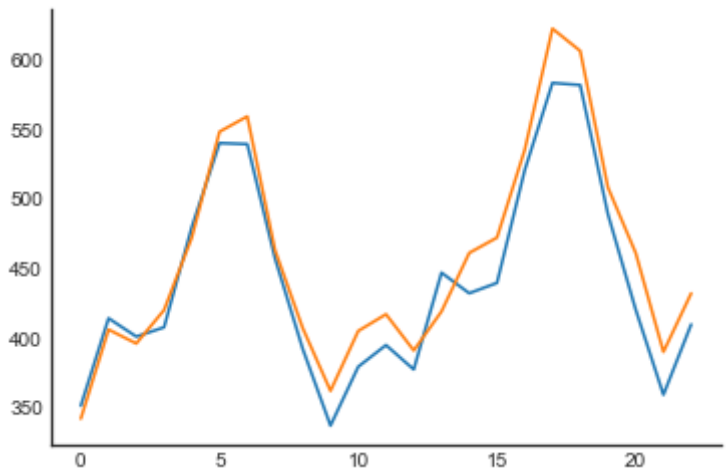
```
z = y.copy()
z = z.to_timestamp(freq="M")
z_train, z_test = temporal_train_test_split(z, test_size=23)
prophet = Prophet(
    seasonality_mode="multiplicative",
    n_changepoints=int(len(y_train) / 12),
    add_country_holidays={"country_name": "Germany"},
    yearly_seasonality=True,
    weekly_seasonality=False,
    daily_seasonality=False,
)
prophet.fit(z_train)
y_pred_prophet = prophet.predict(fh.to_relative(cutoff=y_train.index[-1]))
y_pred_prophet.index = y_test.index
plot_series(y_test, y_pred_prophet, labels=["y_test", "y_pred"])
prophet_loss = loss(y_test, y_pred_prophet)
prophet_loss
>>> 1186.00
```



Neural Prophet

```
set_random_seed(42)
df = pd.read_csv(r".\datasets\air_passengers.csv")
m = NeuralProphet(seasonality_mode="multiplicative")
df_train = df.iloc[:-23, :].copy()
df_test = df.iloc[-23:, :].copy()

m = NeuralProphet(seasonality_mode="multiplicative")
metrics = m.fit(df_train, freq="MS")
future = m.make_future_dataframe(
    df_train, periods=23, n_historic_predictions=len(df_train)
)
forecast = m.predict(future)
plt.plot(forecast["yhat1"].values[-23:])
plt.plot(df_test["y"].values)
neuralprophet_loss = loss(forecast["yhat1"].values[-23:], df_test["y"].values)
```

```
neuralprophet_loss
>>> 501.24
```



The final results can be summarized as follows, resulting in the table presented in the beginning of this case study:

```
results = {
    "Exponential Smoothing": es_loss,
    "ETS": ets_loss,
    "AutoArima": autoarima_loss,
    "Manual Arima": manualarima_loss,
    "BATS": bats_loss,
    "TBATS": tbats_loss,
    "Prophet": prophet_loss,
    "SysIdentPy (Polynomial Model)": frols_loss,
    "SysIdentPy (Neural Model)": narxnet_loss,
    "SysIdentPy (AOLS)": aols_loss,
    "SysIdentPy (MetaMSS)": metamss_loss,
    "NeuralProphet": neuralprophet_loss,
}
sorted(results.items(), key=lambda result: result[1])
```

System With Hysteresis - Modeling a Magneto-rheological Damper Device

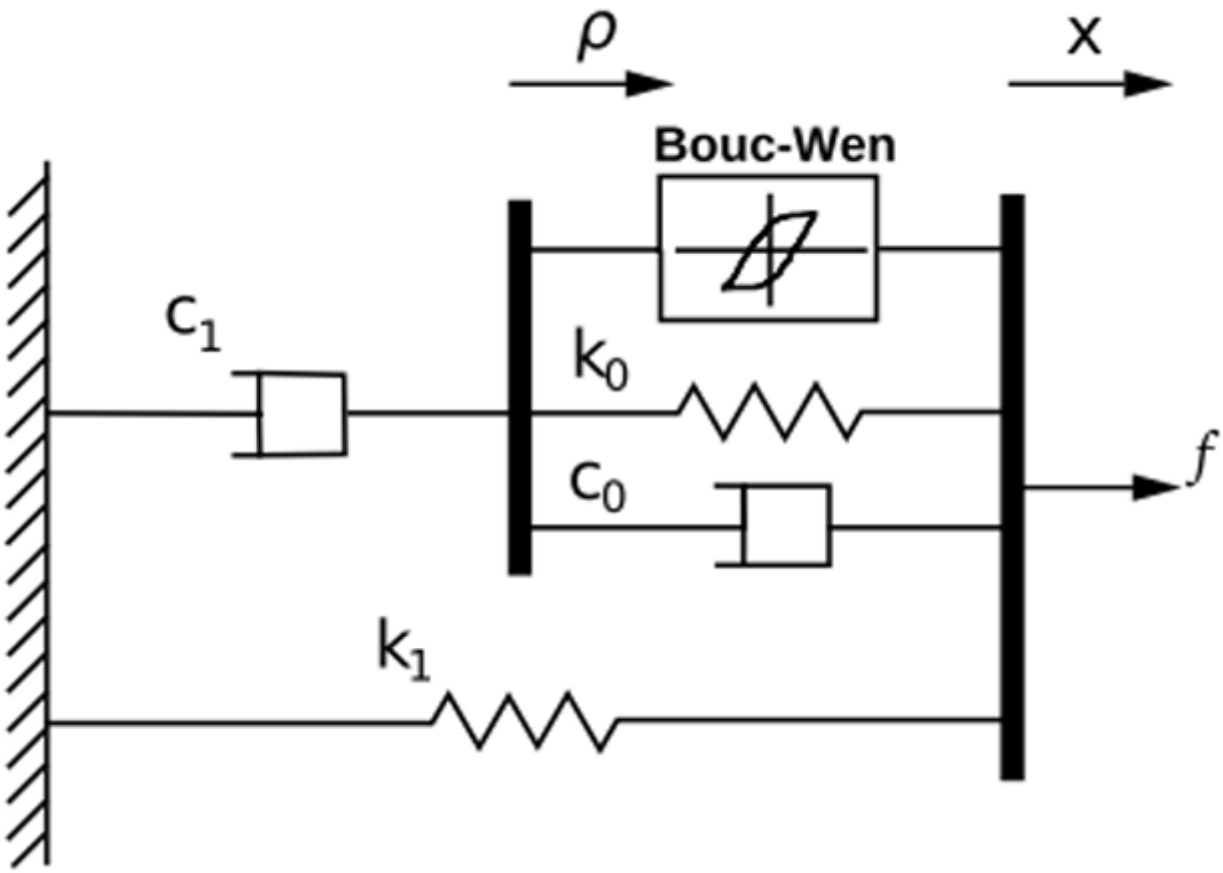
The memory effects between quasi-static input and output make the modeling of hysteretic systems very difficult. Physics-based models are often used to describe the hysteresis loops, but these models usually lack the simplicity and efficiency required in practical applications involving system characterization, identification, and control. As detailed in [Martins, S. A. M. and Aguirre, L. A.] (Sufficient conditions for rate-independent hysteresis in autoregressive identified models - ScienceDirect), NARX models have proven to be a feasible choice to describe the hysteresis loops.

See Chapter 8 for a detailed background. However, even considering the sufficient conditions for rate independent hysteresis representation, classical structure selection algorithms fails to return a model with decent performance and the user needs to set a multi-valued function to ensure the occurrence of the bounding structure $\mathcal{H}$ ([Martins, S. A. M. and Aguirre, L. A.](Sufficient conditions for rate-independent hysteresis in autoregressive identified models - ScienceDirect)).

Even though some progress has been made, previous work has been limited to models with a single equilibrium point. The present case study aims to present new prospects in the model structure selection of hysteretic systems regarding the cases where the models have multiple inputs and it is not restricted concerning the number of equilibrium points. For that, the MetaMSS algorithm will be used to build a model for a magneto-rheological damper (MRD) considering the mentioned sufficient conditions.

A Brief description of the Bouc-Wen model of magneto-rheological damper device

The data used in this study-case is the Bouc-Wen model ([Bouc, R](R. Bouc, “Forced Vibrations of a Mechanical System with Hysteresis,” Proceedings of the 4th Conference on Non-linear Oscillations, Prague, 5-9 September 1967, pp. 315. - References - Scientific Research Publishing (scirp.org)), [Wen, Y. X.](Method for Random Vibration of Hysteretic Systems | Journal of the Engineering Mechanics Division | Vol 102, No 2 (ascelibrary.org))) of an MRD whose schematic diagram is shown in the figure below.



> The model for a magneto-rheological damper proposed by [Spencer, B. F. and Sain, M. K.] (Controlling buildings: a new frontier in feedback | IEEE Journals & Magazine | IEEE Xplore).

The general form of the Bouc-Wen model can be described as ([Spencer, B. F. and Sain, M. K.] (Controlling buildings: a new frontier in feedback | IEEE Journals & Magazine | IEEE Xplore)):

$$\frac{dz}{dt} = g\left[x, z, sign\left(\frac{dx}{dt}\right)\right]\frac{dx}{dt},$$

where z is the hysteretic model output, x the input and $g[\cdot]$ a nonlinear function of x, z and $sign(dx/dt)$. [Spencer, B. F. and Sain, M. K.](Controlling buildings: a new frontier in feedback | IEEE Journals & Magazine | IEEE Xplore) proposed the following phenomenological model for the aforementioned device:

$$\begin{aligned}
f &= c_1\dot{\rho} + k_1(x - x_0),\\
\dot{\rho} &= \frac{1}{c_0 + c_1}[\alpha z + c_0\dot{x} + k_0(x - \rho)],\\
\dot{z} &= -\gamma|\dot{x} - \dot{\rho}|z|z|^{n-1} - \beta(\dot{x} - \dot{\rho})|z|^n + A(\dot{x} - \dot{\rho}),\\
\alpha &= \alpha_a + \alpha_b u_{bw},\\
c_1 &= c_{1a} + c_{1b}u_{bw},\\
c_0 &= c_{0a} + c_{0b}u_{bw},\\
\dot{u}_{bw} &= -\eta(u_{bw} - E).
\end{aligned}$$

where f is the damping force, c_1 and c_0 represent the viscous coefficients, E is the input voltage, x is the displacement and $\dot{x}$ is the velocity of the model. The parameters of the system (see table below)

were taken from [Leva, A. and Piroddi, L.](NARX-based technique for the modelling of magneto-rheological damping devices - IOPscience).

Parameter	Value	Parameter	Value
c_{0_a}	$20.2\,N\,s/cm$	α_a	$44.9\,N/cm$
c_{0_b}	$2.68\,N\,s/cm\,V$	α_b	$638\,N/cm$
c_{1_a}	$350\,N\,s/cm$	γ	$39.3\,cm^{-2}$
c_{1_b}	$70.7\,N\,s/cm\,V$	β	$39.3\,cm^{-2}$
k_0	$15\,N/cm$	n	2
k_1	$5.37\,N/cm$	η	$251\,s^{-1}$
x_0	$0\,cm$	A	47.2

For this particular study, both displacement and voltage inputs, x and E, respectively, were generated by filtering a white Gaussian noise sequence using a Blackman-Harris FIR filter with 6Hz cutoff frequency. The integration step-size was set to $h = 0.002$, following the procedures described in [Martins, S. A. M. and Aguirre, L. A.](Sufficient conditions for rate-independent hysteresis in autoregressive identified models - ScienceDirect). These procedures are for identification purposes only since the inputs of a MRD could have several different characteristics.

The data used in this example is provided by the Professor Samir Angelo Milani Martins.

The challenges are:

- it possesses a nonlinearity featuring memory, i.e. a dynamic nonlinearity;
- the nonlinearity is governed by an internal variable z(t), which is not measurable;
- the nonlinear functional form in the Bouc Wen equation is nonlinear in the parameter;
- the nonlinear functional form in the Bouc Wen equation does not admit a finite Taylor series expansion because of the presence of absolute values

Required Packages and Versions

To ensure that you can replicate this case study, it is essential to use specific versions of the required packages. Below is a list of the packages along with their respective versions needed for running the case studies effectively.

To install all the required packages, you can create a `requirements.txt` file with the following content:

```
sysidentpy==0.4.0
pandas==2.2.2
numpy==1.26.0
```

```
matplotlib==3.8.4
scikit-learn==1.4.2
```

Then, install the packages using:

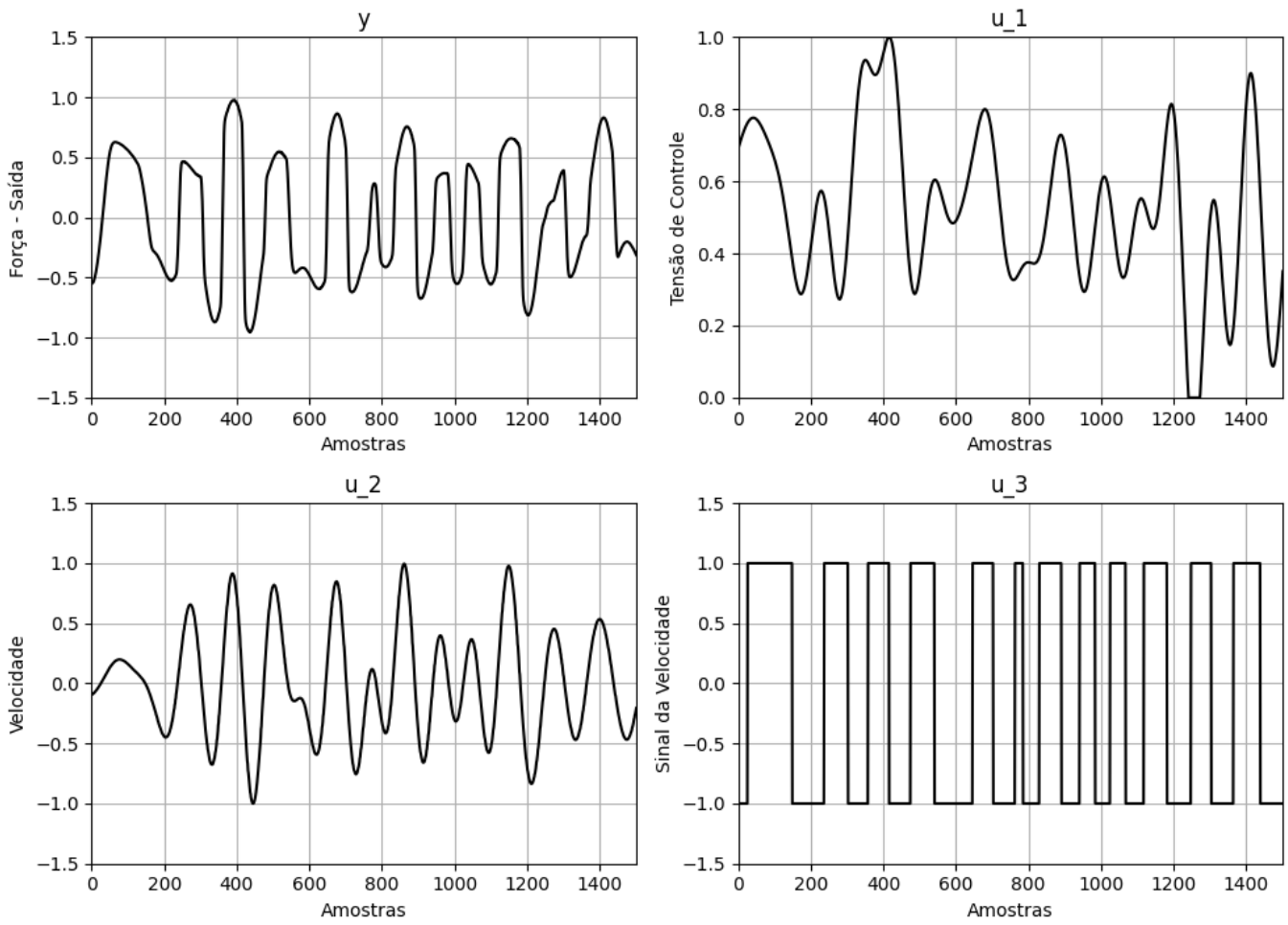
```
pip install -r requirements.txt
```

- Ensure that you use a virtual environment to avoid conflicts between package versions.
- Versions specified are based on compatibility with the code examples provided. If you are using different versions, some adjustments in the code might be necessary.

SysIdentPy Configuration

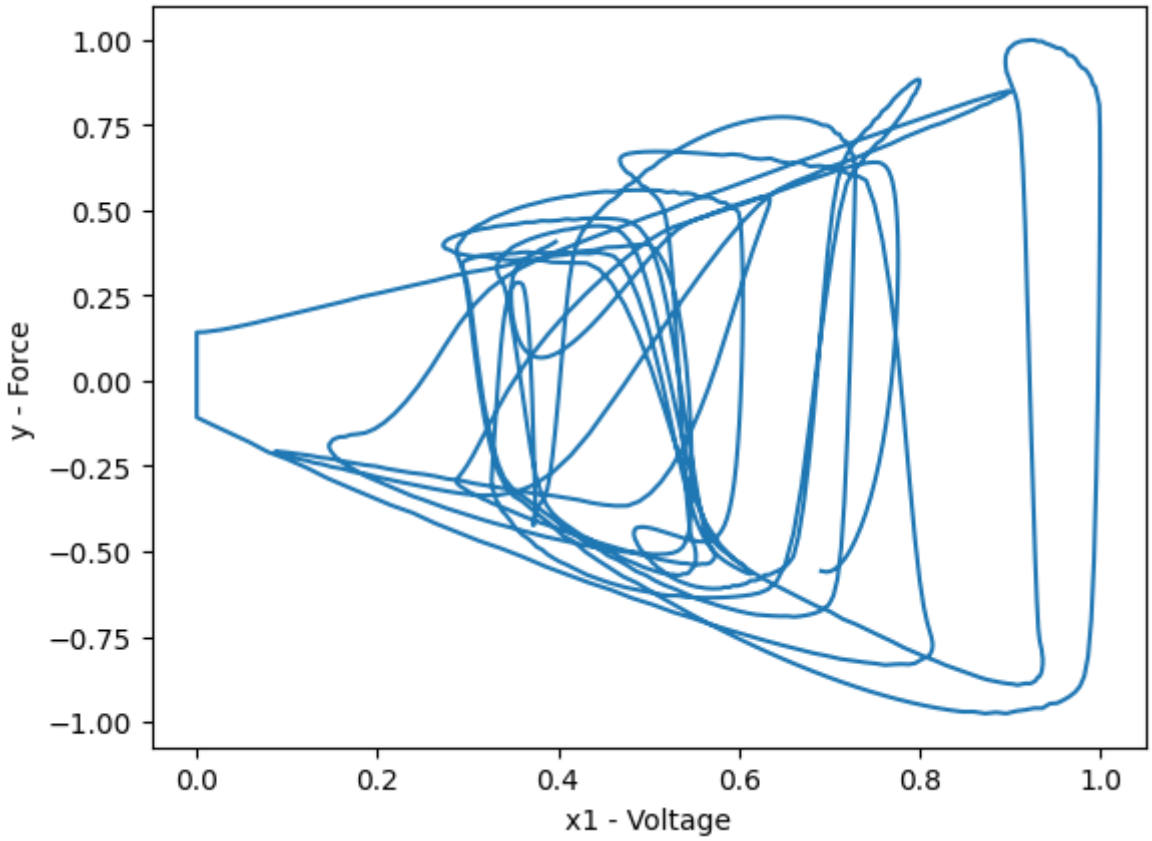
```
import numpy as np
from sklearn.preprocessing import MaxAbsScaler, MinMaxScaler
import pandas as pd
import matplotlib.pyplot as plt

from sysidentpy.model_structure_selection import FROLS
from sysidentpy.basis_function import Polynomial
from sysidentpy.utils.display_results import results
from sysidentpy.parameter_estimation import LeastSquares
from sysidentpy.metrics import root_relative_squared_error
from sysidentpy.utils.plotting import plot_results

df = pd.read_csv("boucwen_histeretic_system.csv")
scaler_x = MaxAbsScaler()
scaler_y = MaxAbsScaler()

init = 400
x_train = df[["E", "v"]].iloc[init:df.shape[0]//2, :]
x_train["sign_v"] = np.sign(df["v"])
x_train = scaler_x.fit_transform(x_train)

x_test = df[["E", "v"]].iloc[df.shape[0]//2 + 1:df.shape[0] - init, :]
x_test["sign_v"] = np.sign(df["v"])
x_test = scaler_x.transform(x_test)

y_train = df[["f"]].iloc[init:df.shape[0]//2, :].values.reshape(-1, 1)
y_train = scaler_y.fit_transform(y_train)

y_test = df[["f"]].iloc[df.shape[0]//2 + 1:df.shape[0] - init, :].values.reshape(-1,
1)
y_test = scaler_y.transform(y_test)
```

```
# Plotting the data
plt.figure(figsize=(10, 8))
plt.suptitle('Identification (training) data', fontsize=16)

plt.subplot(221)
plt.plot(y_train, 'k')
plt.ylabel('Force - Output')
plt.xlabel('Samples')
plt.title('y')
plt.grid()
plt.axis([0, 1500, -1.5, 1.5])

plt.subplot(222)
plt.plot(x_train[:, 0], 'k')
plt.ylabel('Control Voltage')
plt.xlabel('Samples')
plt.title('x_1')
plt.grid()
plt.axis([0, 1500, 0, 1])

plt.subplot(223)
plt.plot(x_train[:, 1], 'k')
plt.ylabel('Velocity')
plt.xlabel('Samples')
plt.title('x_2')
plt.grid()
plt.axis([0, 1500, -1.5, 1.5])

plt.subplot(224)
plt.plot(x_train[:, 2], 'k')
plt.ylabel('sign(Velocity)')
plt.xlabel('Samples')
plt.title('x_3')
plt.grid()
plt.axis([0, 1500, -1.5, 1.5])

plt.tight_layout(rect=[0, 0, 1, 0.95])
plt.show()
```

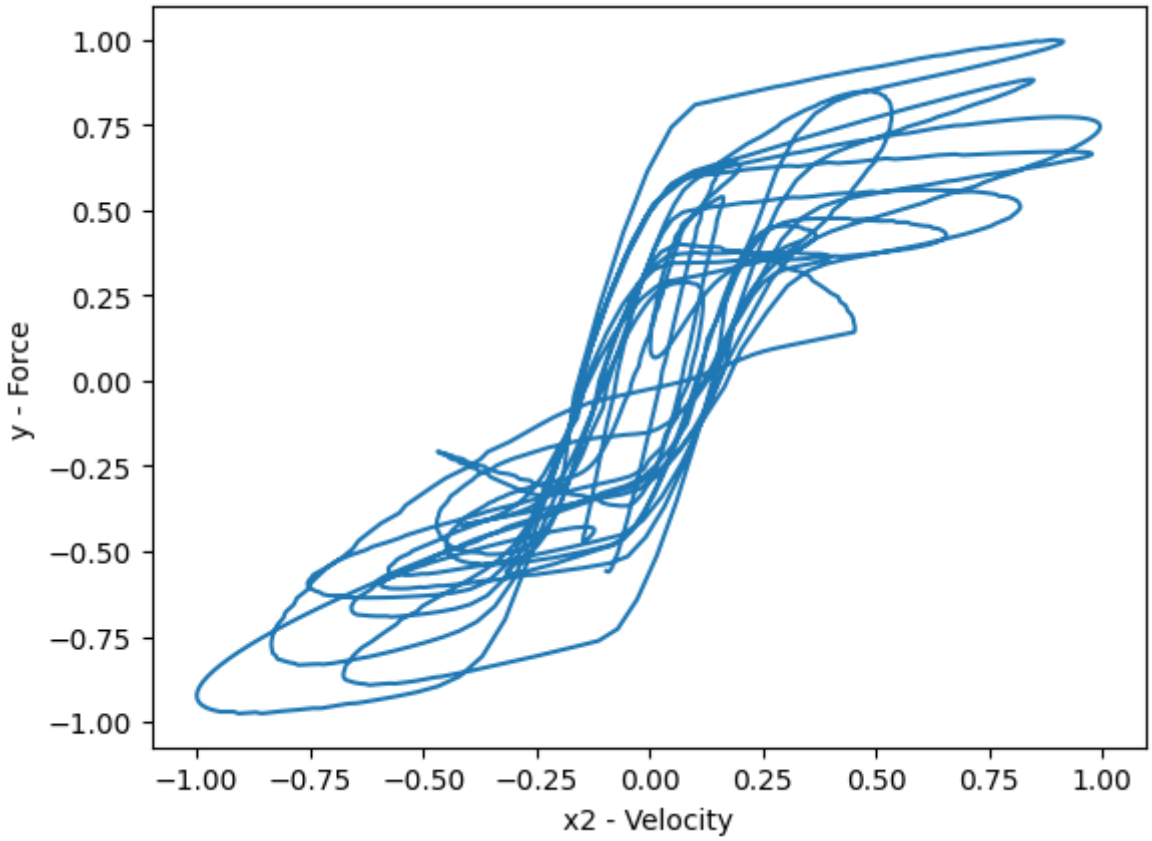


Let's check how is the histeretic behavior considering each input:

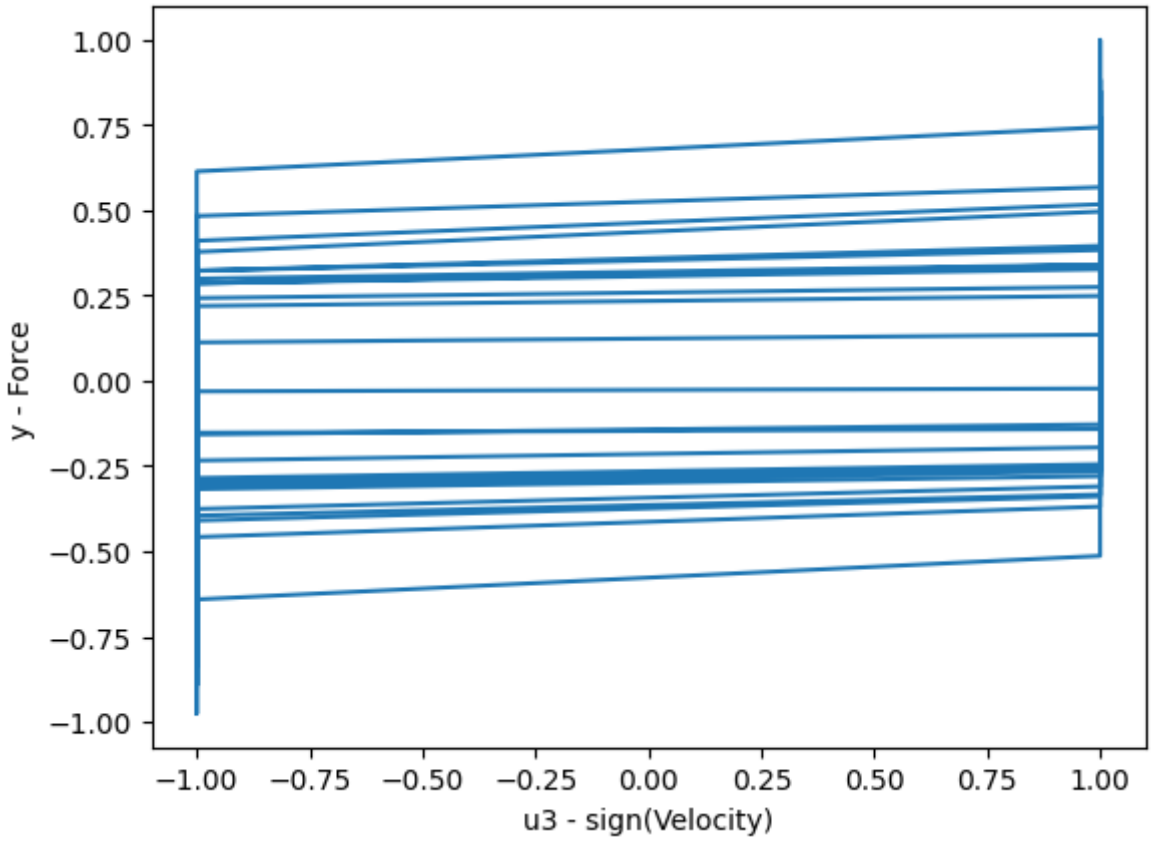
```
plt.plot(x_train[:, 0], y_train)
plt.xlabel("x1 - Voltage")
plt.ylabel("y - Force")
```




```
plt.plot(x_train[:, 1], y_train)
plt.xlabel("x2 - Velocity")
plt.ylabel("y - Force")
```

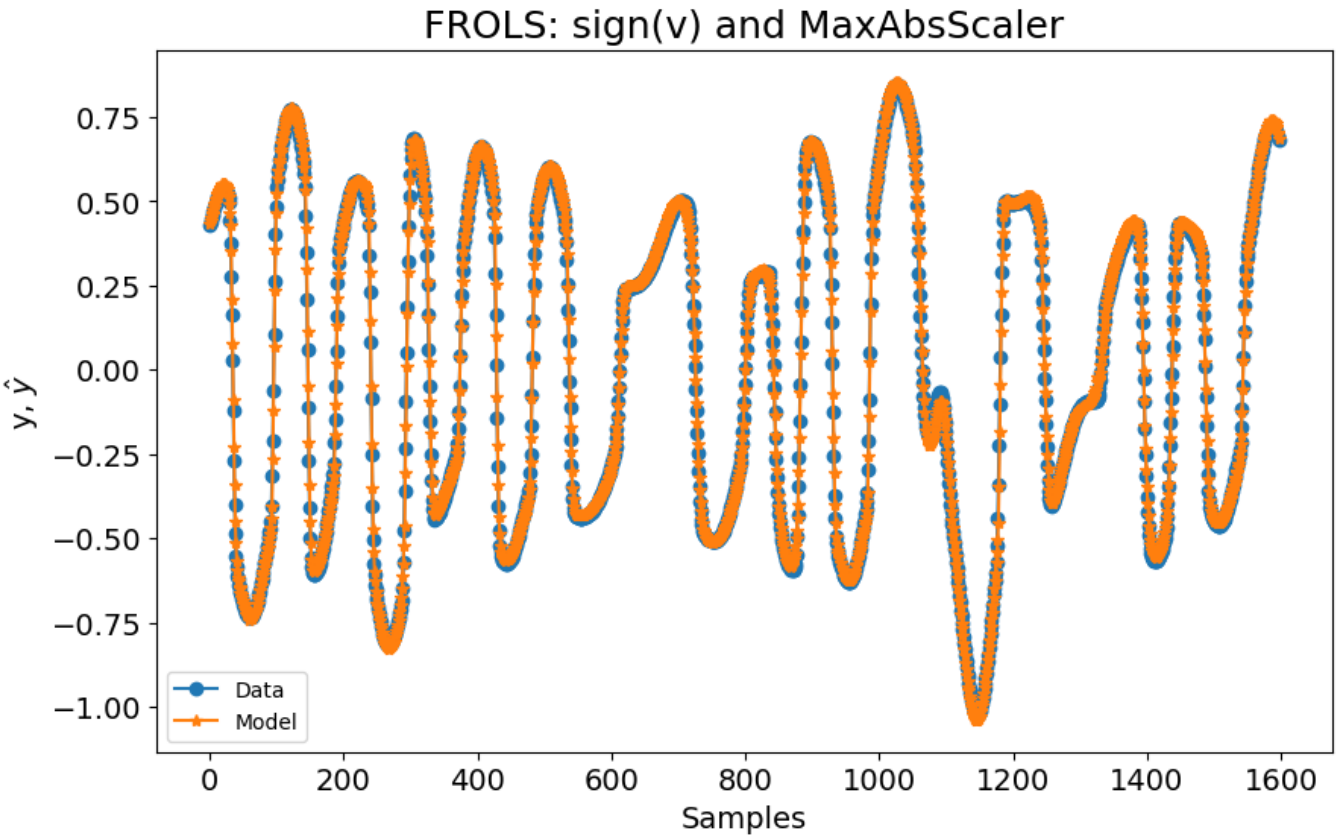



```
plt.plot(x_train[:, 2], y_train)
plt.xlabel("u3 - sign(Velocity)")
plt.ylabel("y - Force")
```



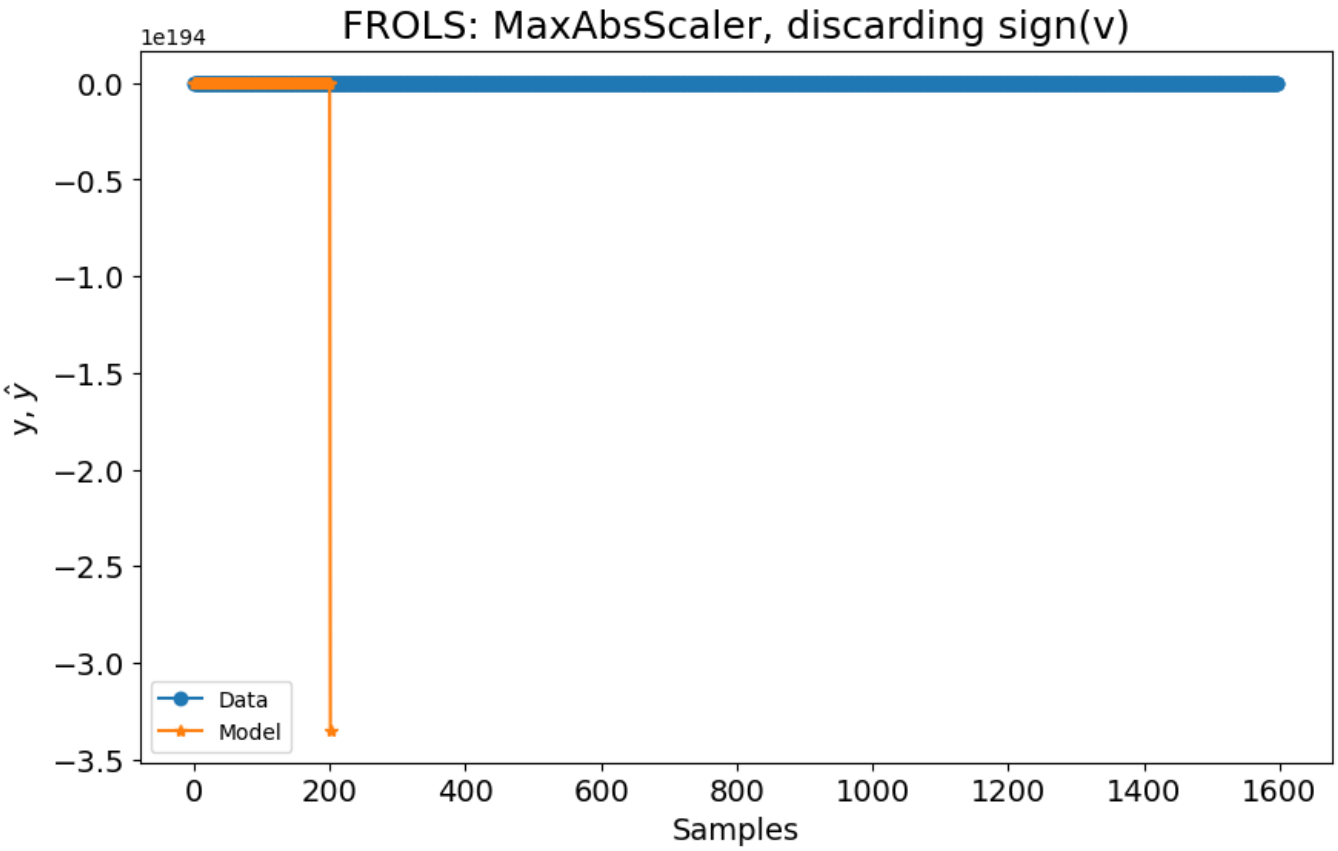
Now, we can just build a NARX model:

```
basis_function = Polynomial(degree=3)
model = FROLS(
    xlag=[[1], [1], [1]],
    ylag=1,
    basis_function=basis_function,
    estimator=LeastSquares(),
    info_criteria="aic",
)

model.fit(X=x_train, y=y_train)
yhat = model.predict(X=x_test, y=y_test[:model.max_lag :, :])
rrse = root_relative_squared_error(y_test[model.max_lag :], yhat[model.max_lag :])
print(rrse)
plot_results(y=y_test[model.max_lag :], yhat=yhat[model.max_lag :], n=10000,
title="FROLS: sign(v) and MaxAbsScaler")
>>> 0.0450
```



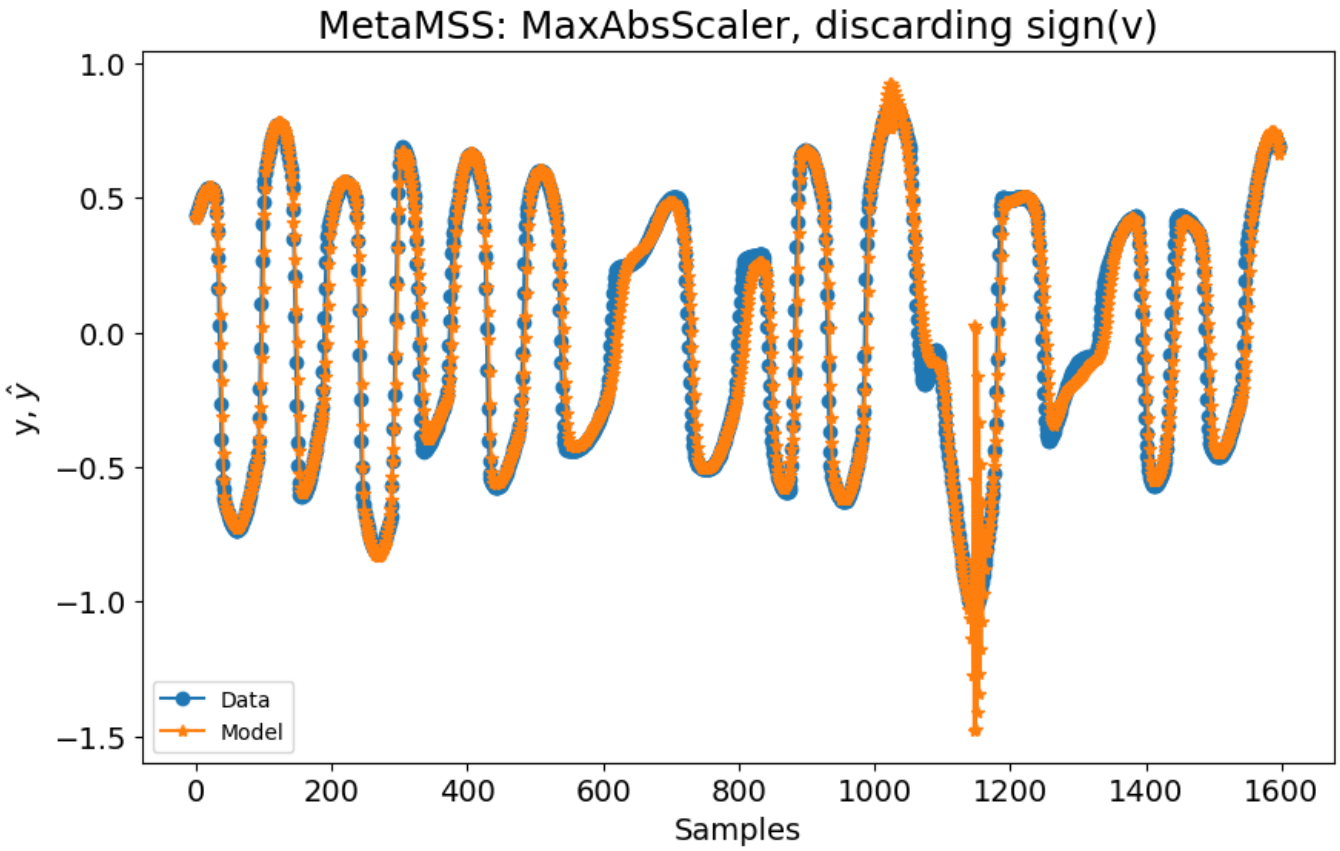
If we remove the `sign(v)` input and try to build a NARX model using the same configuration, the model diverge, as can be seen in the following figure:

```
basis_function = Polynomial(degree=3)
model = FROLS(
    xlag=[[1], [1]],
    ylag=1,
    basis_function=basis_function,
    estimator=LeastSquares(),
    info_criteria="aic",
)

model.fit(X=x_train[:, :2], y=y_train)
yhat = model.predict(X=x_test[:, :2], y=y_test[:model.max_lag :, :])
rrse = root_relative_squared_error(y_test[model.max_lag :], yhat[model.max_lag :])
print(rrse)
plot_results(y=y_test[model.max_lag :], yhat=yhat[model.max_lag :], n=10000,
title="FROLS: MaxAbsScaler, discarding sign(v)")
>>> nan
```



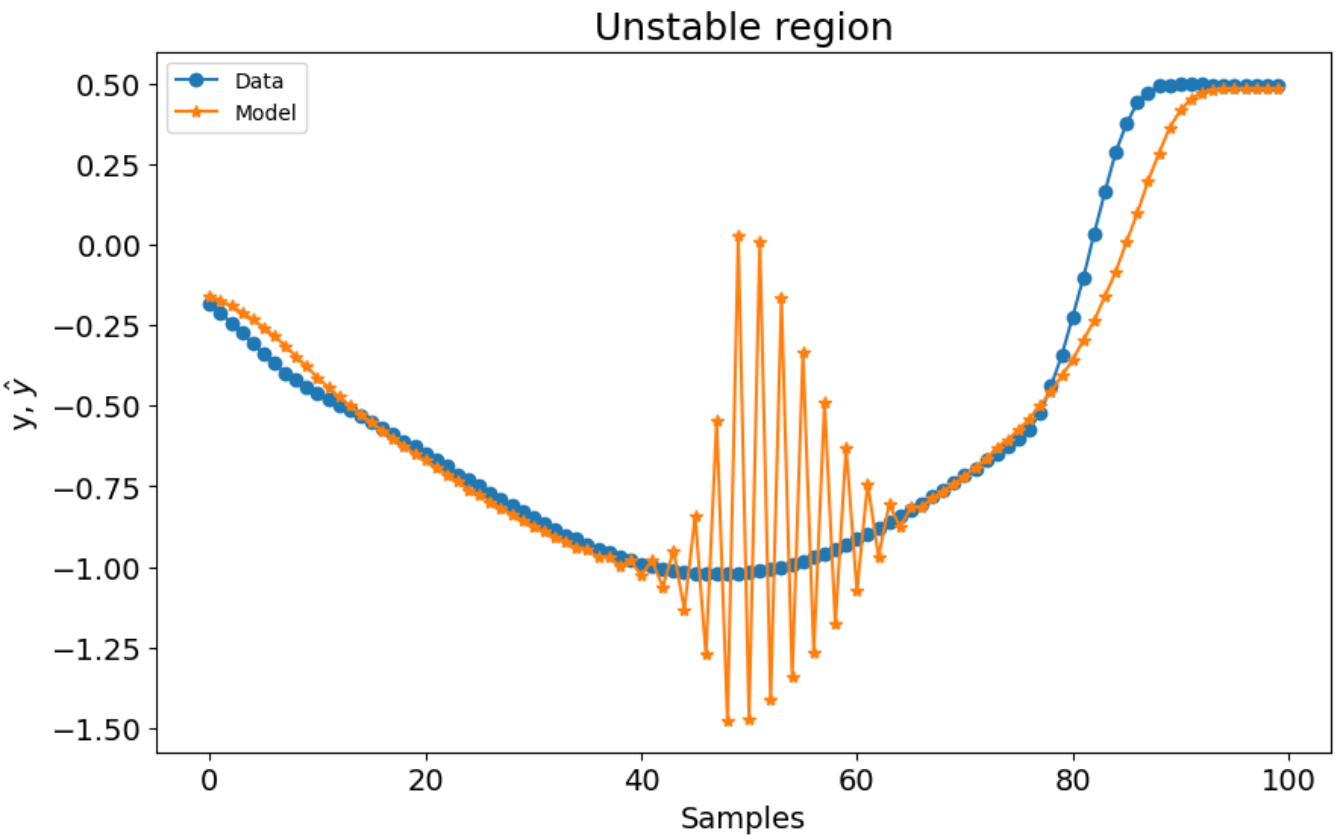
If we use the `MetaMSS` algorithm instead, the results are better.

```
from sysidentpy.model_structure_selection import MetaMSS

basis_function = Polynomial(degree=3)
model = MetaMSS(
    xlag=[[1], [1]],
    ylag=1,
    basis_function=basis_function,
    estimator=LeastSquares(),
    random_state=42,
)

model.fit(X=x_train[:, :2], y=y_train)
yhat = model.predict(X=x_test[:, :2], y=y_test[:model.max_lag :, :])
rrse = root_relative_squared_error(y_test[model.max_lag :], yhat[model.max_lag :])
print(rrse)
plot_results(y=y_test[model.max_lag :], yhat=yhat[model.max_lag :], n=10000,
title="MetaMSS: MaxAbsScaler, discarding sign(v)")
>>> 0.24
```



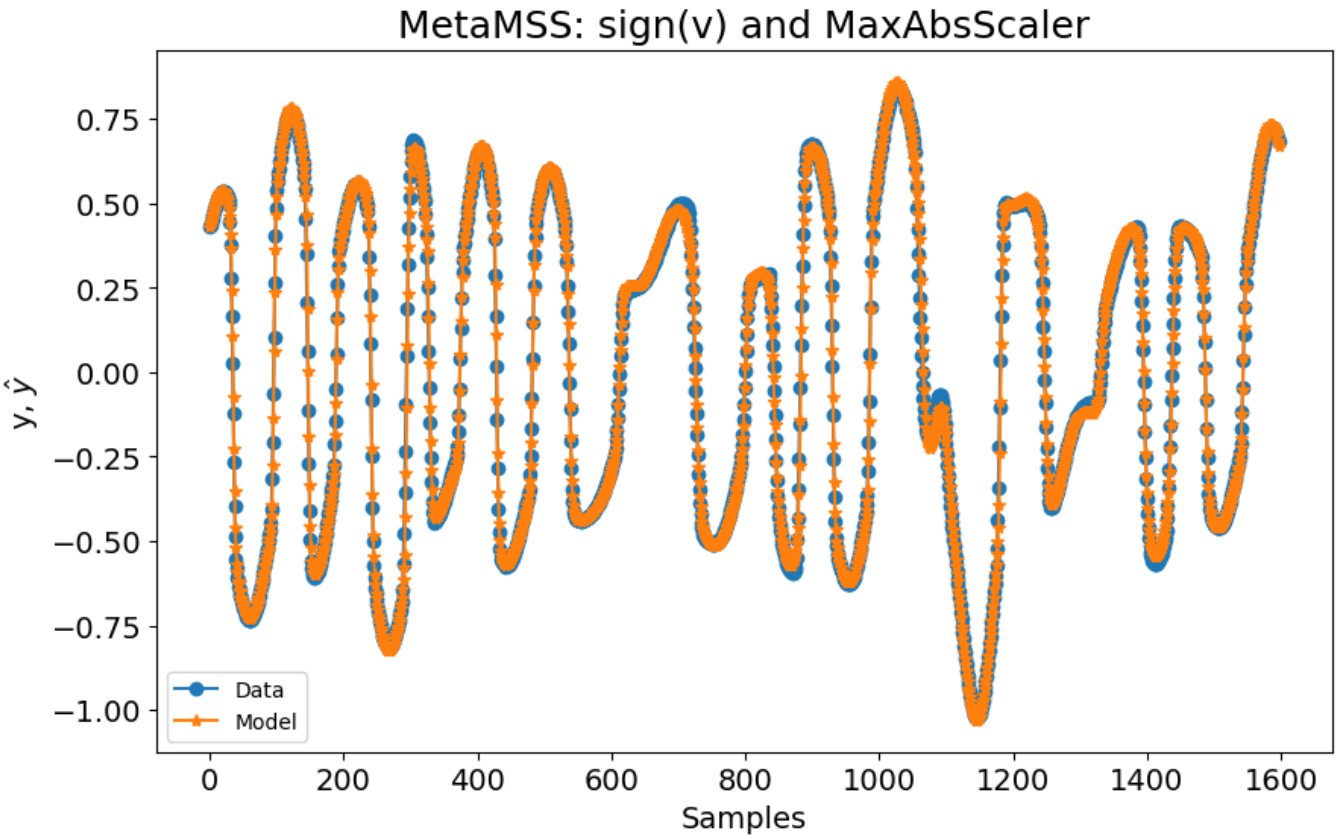
However, when the output of the system reach its minimum value, the model oscillate

```
plot_results(y=y_test[1100 : 1200], yhat=yhat[1100 : 1200], n=10000, title="Unstable region")
```



If we add the `sign(v)` input again and use `MetaMSS`, the results are very close to the `FROLS` algorithm with all inputs

```
basis_function = Polynomial(degree=3)
model = MetaMSS(
    xlag=[[1], [1], [1]],
    ylag=1,
    basis_function=basis_function,
    estimator=LeastSquares(),
    random_state=42,
)

model.fit(X=x_train, y=y_train)
yhat = model.predict(X=x_test, y=y_test[:model.max_lag :, :])
rrse = root_relative_squared_error(y_test[model.max_lag :], yhat[model.max_lag :])
print(rrse)
plot_results(y=y_test[model.max_lag :], yhat=yhat[model.max_lag :], n=10000,
title="MetaMSS: sign(v) and MaxAbsScaler")
>>> 0.0554
```

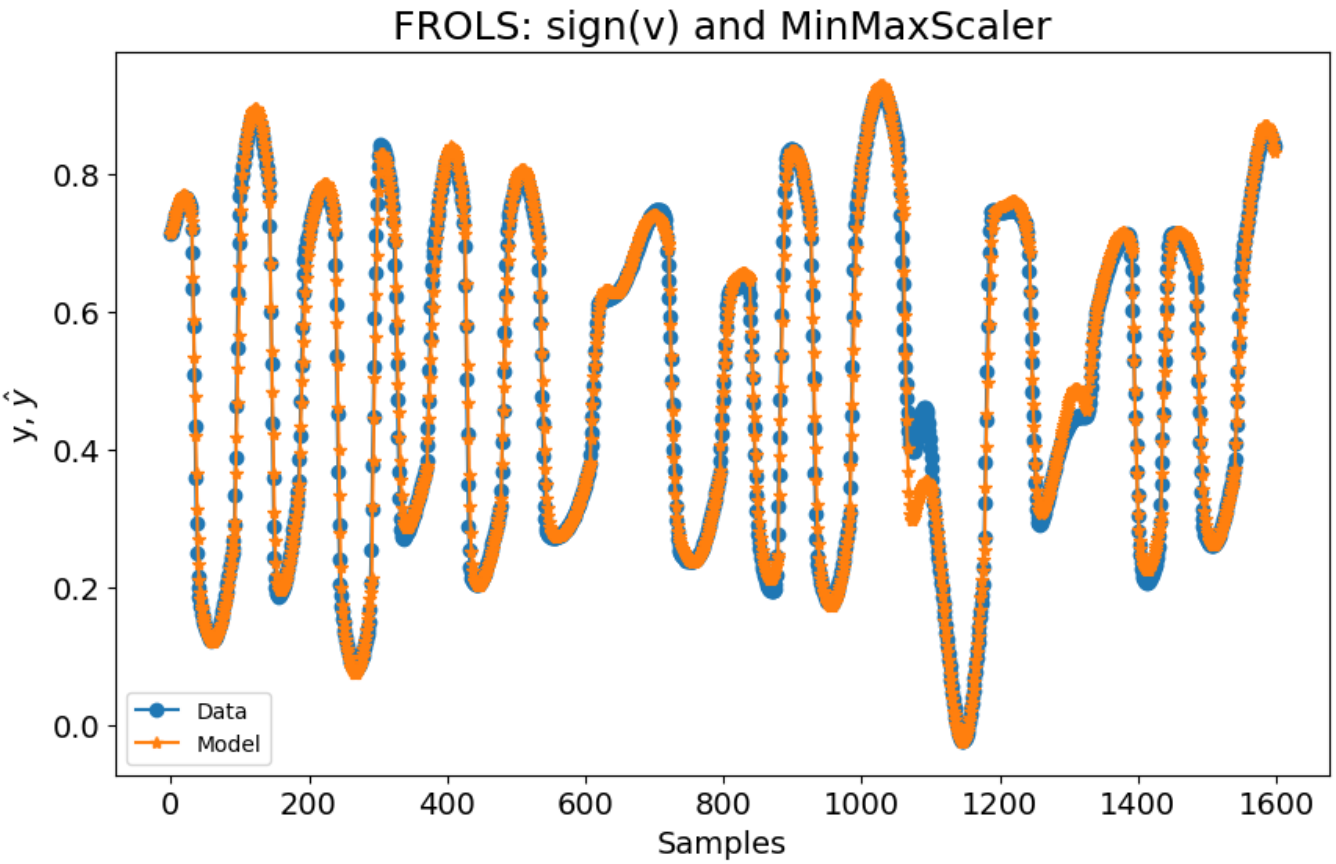


This case will also highlight the significance of data scaling. Previously, we used the `MaxAbsScaler` method, which resulted in great models when using the `sign(v)` inputs, but also resulted in unstable models when removing that input feature. When scaling is applied using `MinMaxScaler`, however, the overall stability of the results improves, and the model does not diverge, even when the `sign(v)` input is removed, using the `FROLS` algorithm.

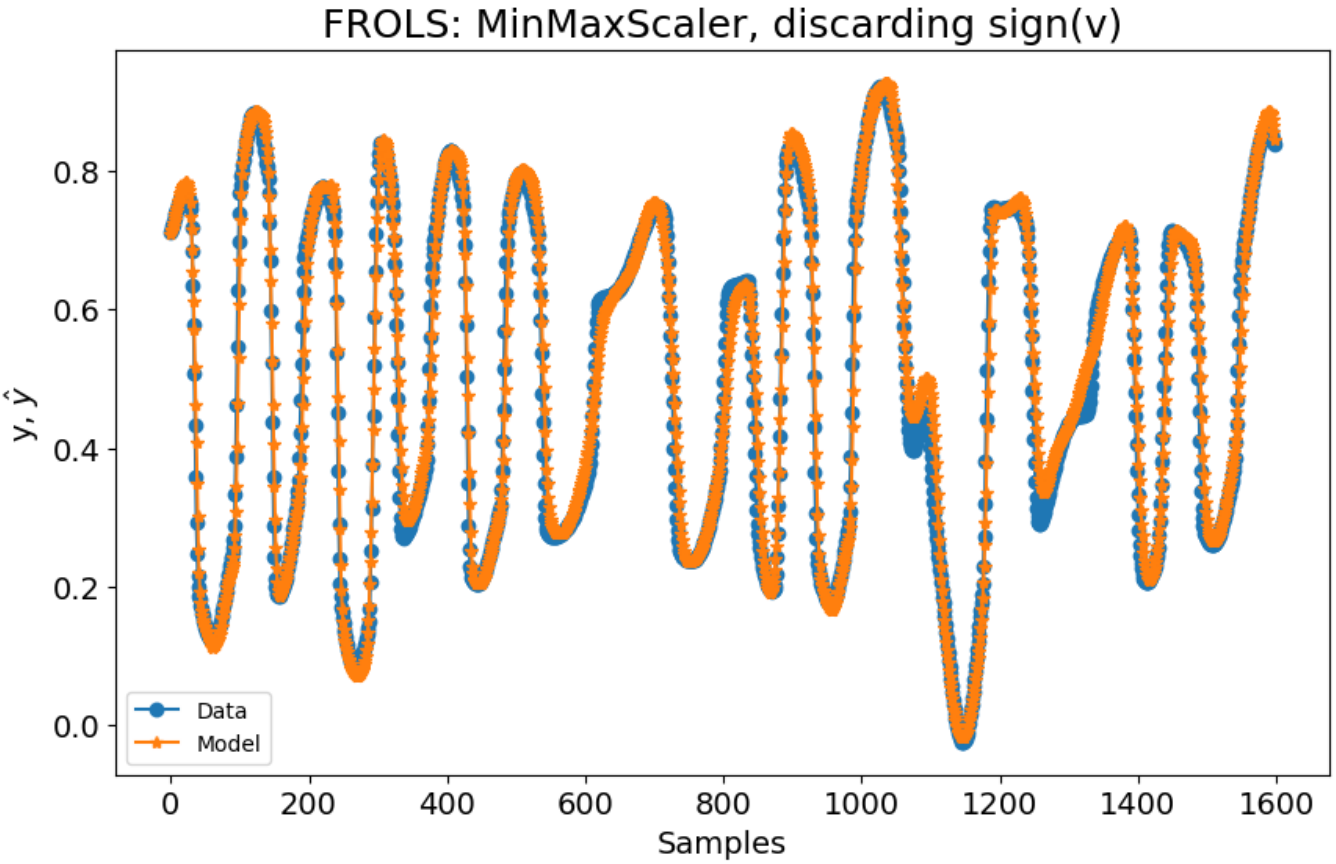
The user can get the results bellow by just changing the data scaling method using

```
scaler_x = MinMaxScaler()
scaler_y = MinMaxScaler()
```

and running the each model again. That is the only change to improve the results.

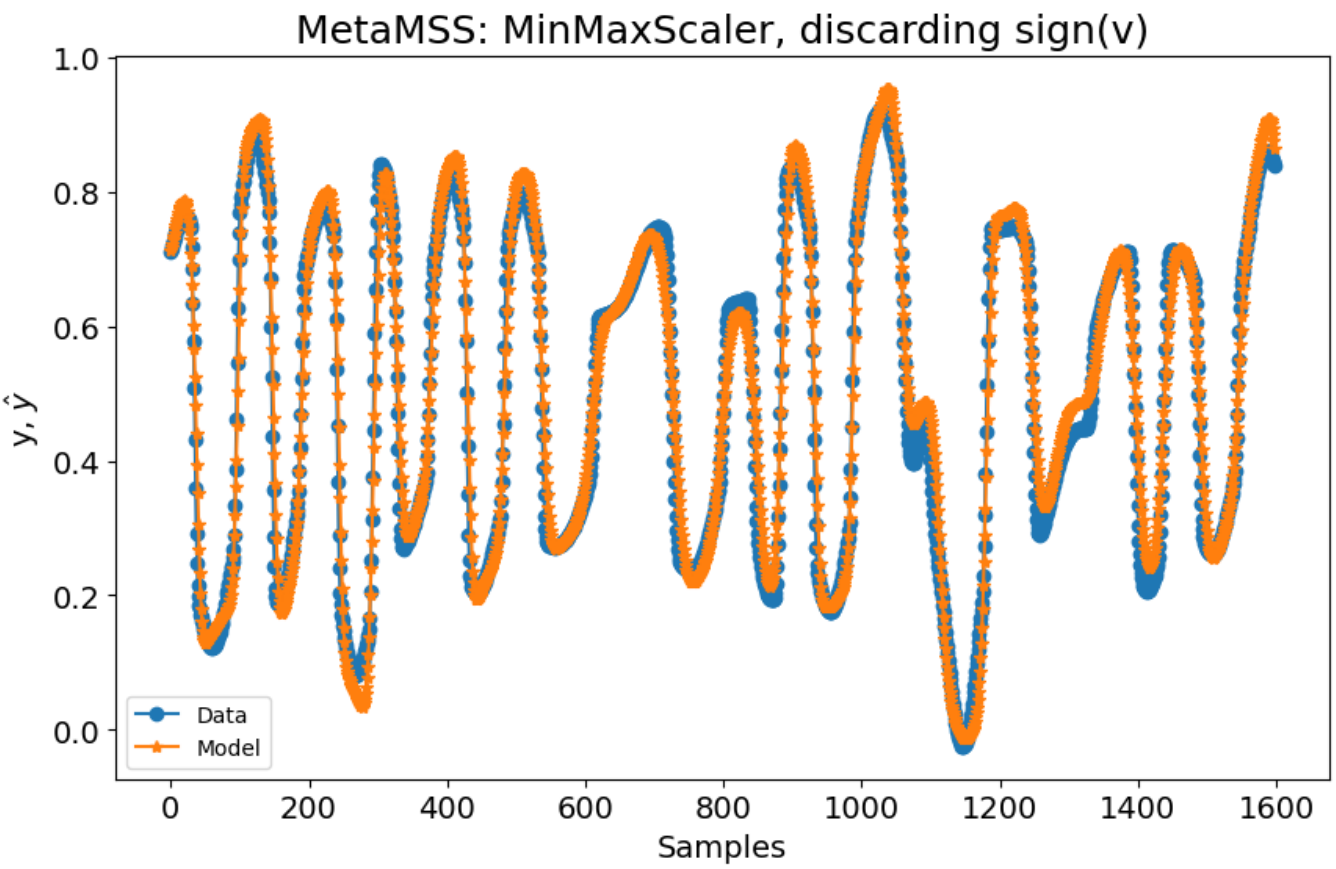


FROLS: with `sign(v)` and `MinMaxScaler`. RMSE: 0.1159

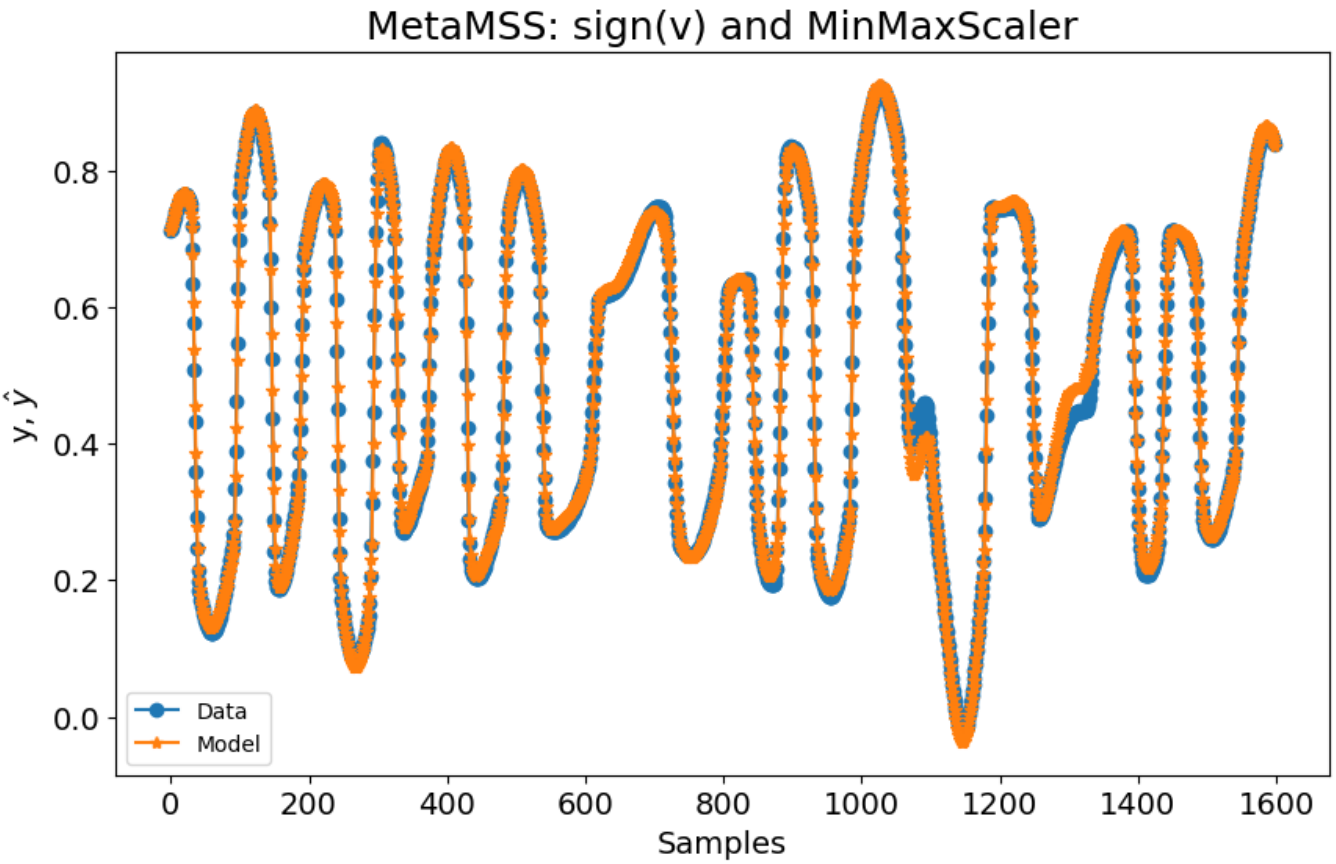


`sign(v)`

`MinMaxScaler`



> MetaMSS: discarding `sign(v)` and using `MinMaxScaler`. RMSE: 0.1762

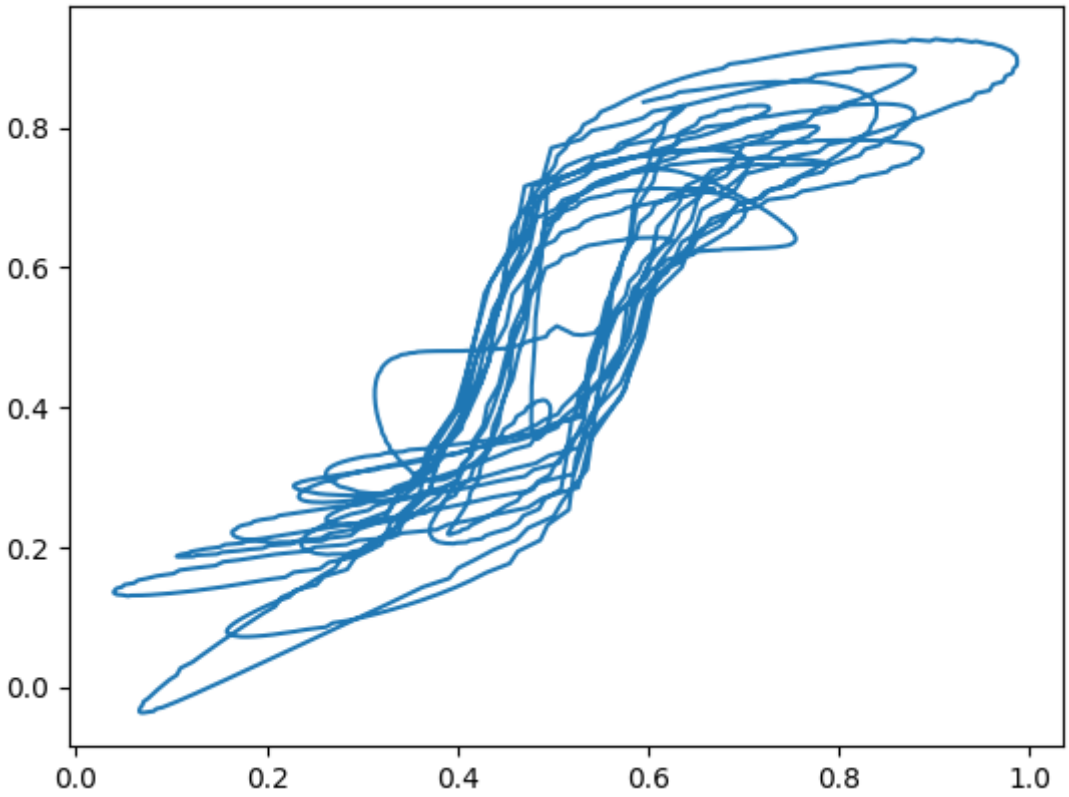


> MetaMSS: including `sign(v)` and using `MinMaxScaler`. RMSE: 0.0694

In contrast, the MetaMSS method returned the best model overall, but not better than the best `FROLS` method using `MaxAbsScaler`.

Here is the predicted histeretic loop:

```
plt.plot(x_test[:, 1], yhat)
```



Silver box

The description content mainly derives (copy and paste) from the [associated paper](Three free data sets for development and benchmarking in nonlinear system identification | IEEE Conference Publication | IEEE Xplore). For a detailed description, readers are referred to the linked reference.

> The Silverbox system can be seen as an electronic implementation of the Duffing oscillator. It is build as a 2nd order linear time-invariant system with a 3rd degree polynomial static nonlinearity around it in feedback. This type of dynamics are, for instance, often encountered in mechanical systems. [Nonlinear Benchmark](Nonlinear Benchmark - Silverbox)

In this case study, we will create a NARX model for the Silver box benchmark. The Silver box represents a simplified version of mechanical oscillating processes, which are a critical category of nonlinear dynamic systems. Examples include vehicle suspensions, where shock absorbers and progressive springs play vital roles. The data generated by the Silver box provides a simplified representation of such combined components. The electrical circuit generating this data closely approximates, but does not perfectly match, the idealized models described below.

As described in the original paper, the system was excited using a general waveform generator (HPE1445A). The input signal begins as a discrete-time signal $r(k)$, which is converted to an analog signal $r_c(t)$ using zero-order-hold reconstruction. The actual excitation signal $u_0(t)$ is then obtained by passing $r_c(t)$ through an analog low-pass filter $G(p)$ to eliminate high-frequency content around multiples of the sampling frequency. Here, p denotes the differentiation operator. Thus, the input is given by:

$$u_0(t) = G(p) r_c(t).$$

The input and output signals were measured using HP1430A data acquisition cards, with synchronized clocks for the acquisition and generator cards. The sampling frequency was:

$$f_s = \frac{10^7}{2^{14}} = 610.35\,\mathrm{Hz}.$$

The silver box uses analog electrical circuitry to generate data representing a nonlinear mechanical resonating system with a moving mass m, viscous damping d, and a nonlinear spring $k(y)$. The electrical circuit is designed to relate the displacement $y(t)$ (the output) to the force $u(t)$ (the input) by the following differential equation:

$$m\frac{d^2y(t)}{dt^2} + d\frac{dy(t)}{dt} + k(y(t))y(t) = u(t).$$

The nonlinear progressive spring is described by a static, position-dependent stiffness:

$$k(y(t)) = a + by^2(t).$$

The signal-to-noise ratio is sufficiently high to model the system without accounting for measurement noise. However, measurement noise can be included by replacing $y(t)$ with the artificial variable $x(t)$ in the equation above, and introducing disturbances $w(t)$ and $e(t)$ as follows:

$$\begin{aligned} &m\frac{d^2x(t)}{dt^2} + d\frac{dx(t)}{dt} + k(x(t))x(t) = u(t) + w(t), \\ &k(x(t)) = a + bx^2(t), \\ &y(t) = x(t) + e(t). \end{aligned}$$

Required Packages and Versions

To ensure that you can replicate this case study, it is essential to use specific versions of the required packages. Below is a list of the packages along with their respective versions needed for running the case studies effectively.

To install all the required packages, you can create a `requirements.txt` file with the following content:

```
sysidentpy==0.4.0
pandas==2.2.2
numpy==1.26.0
matplotlib==3.8.4
nonlinear_benchmarks==0.1.2
```

Then, install the packages using:

```
pip install -r requirements.txt
```

- Ensure that you use a virtual environment to avoid conflicts between package versions.
- Versions specified are based on compatibility with the code examples provided. If you are using different versions, some adjustments in the code might be necessary.

SysIdentPy configuration

In this section, we will demonstrate the application of SysIdentPy to the Silver box dataset. The following code will guide you through the process of loading the dataset, configuring the SysIdentPy parameters, and building a model for mentioned system.

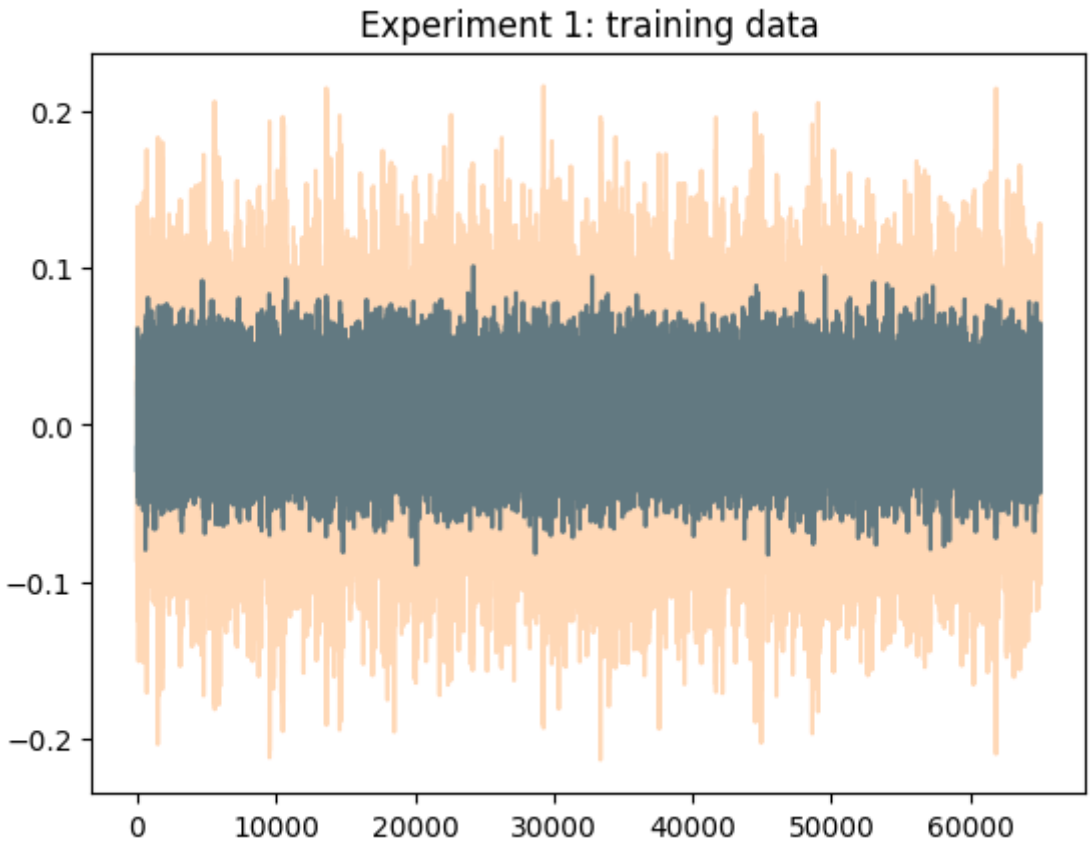
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

from sysidentpy.model_structure_selection import FROLS
from sysidentpy.basis_function import Polynomial, Fourier
from sysidentpy.parameter_estimation import LeastSquares
from sysidentpy.metrics import root_mean_squared_error
from sysidentpy.utils.plotting import plot_results

import nonlinear_benchmarks

train_val, test = nonlinear_benchmarks.Silverbox(atleast_2d=True)
```

We used the `nonlinear_benchmarks` package to load the data. The user is referred to the [package documentation](GerbenBeintema/nonlinear_benchmarks: The official dataload for http://www.nonlinearbenchmark.org/ (github.com)) to check the details of how to use it.

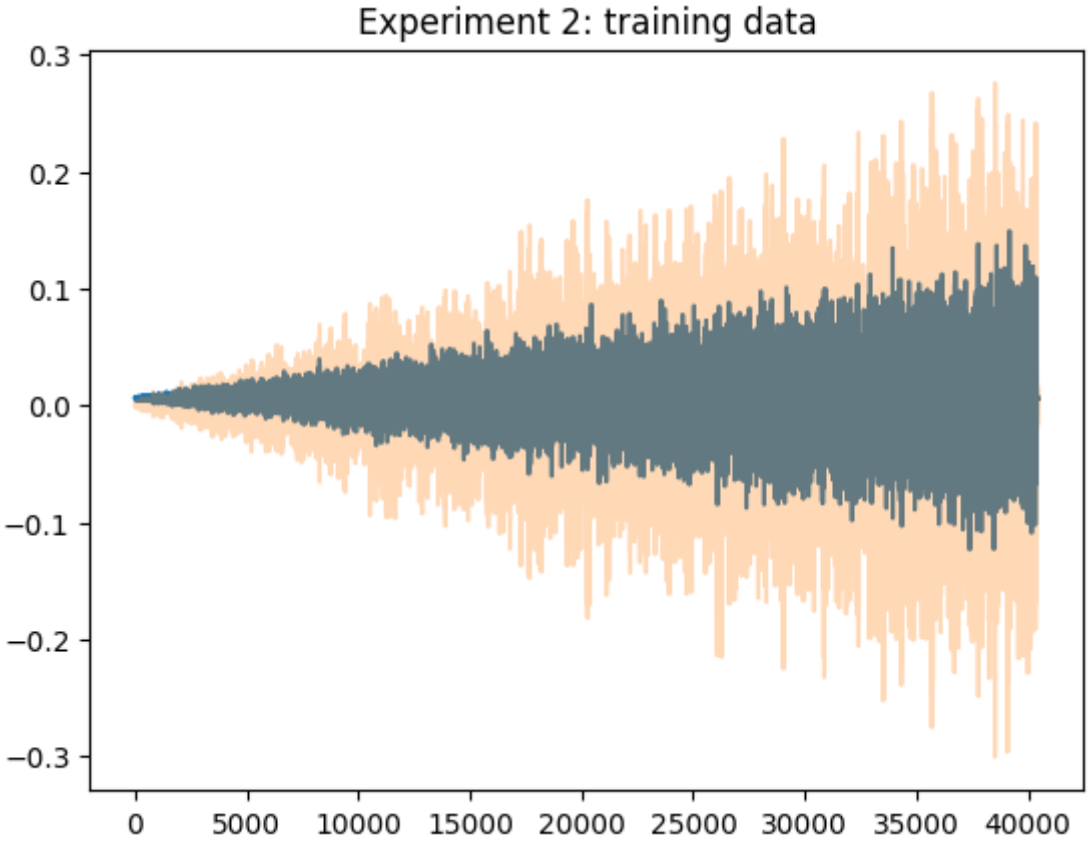
The following plot detail the training and testing data of the experiment.

```
plt.plot(x_train)
plt.plot(y_train, alpha=0.3)
plt.title("Experiment 1: training data")
plt.show()

plt.plot(x_test)
plt.plot(y_test, alpha=0.3)
plt.title("Experiment 1: testing data")
plt.show()

plt.plot(test_arrow_full.u)
plt.plot(test_arrow_full.y, alpha=0.3)
plt.title("Experiment 2: training data")
plt.show()
```

```
plt.plot(test_arrow_no_extrapolation.u)
plt.plot(test_arrow_no_extrapolation.y, alpha=0.2)
plt.title("Experiment 2: testing data")
plt.show()
```




Experiment 1: testing data
0.2
0.1
0.0
−0.1
−0.2
0
5000
10000
15000
20000


Experiment 2: training data
0.3
0.2
0.1
0.0
−0.1
−0.2
−0.3
0
5000
10000
15000
20000
25000
30000
35000
40000

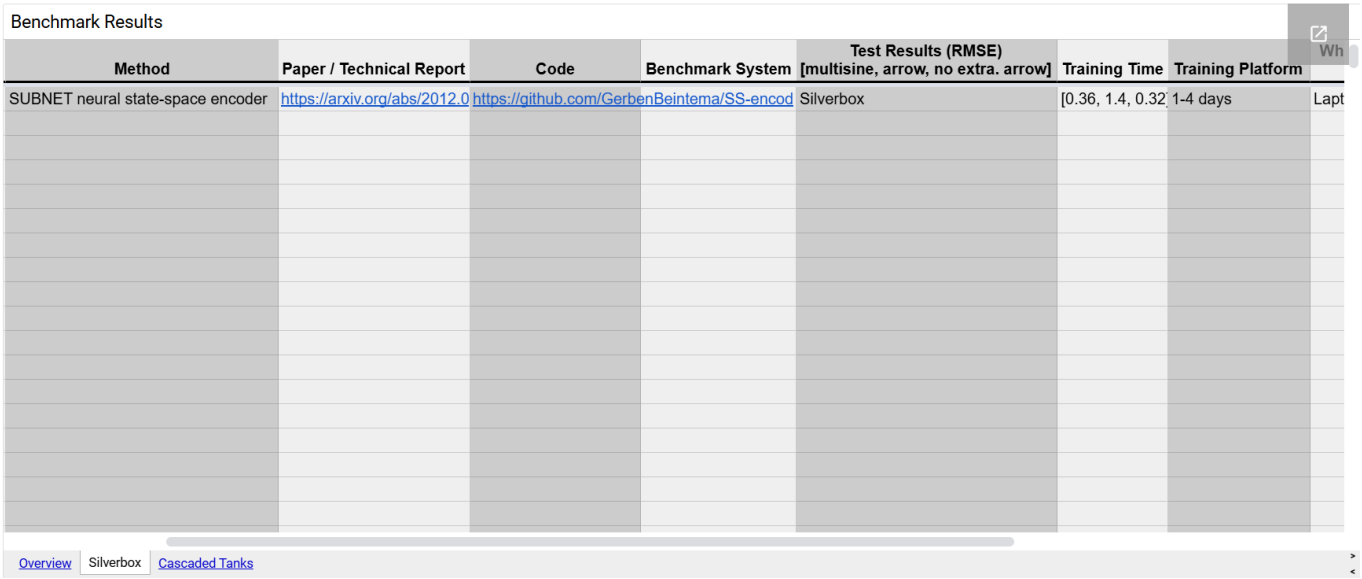
> Important Note

The goal of this benchmark is to develop a model that outperforms the state-of-the-art (SOTA) model presented in the benchmarking paper. However, the results in the paper differ from those provided in the GitHub repository.

nx	Set	NRMS	RMS (mV)
2	Train	0.10653	5.8103295
2	Validation	0.11411	6.1938068
2	Test	0.19151	10.2358533
2	Test (no extra)	0.12284	5.2789727
4	Train	0.03571	1.9478290
4	Validation	0.03922	2.1286373
4	Test	0.12712	6.7943448
4	Test (no extra)	0.05204	2.2365904
8	Train	0.03430	1.8707026
8	Validation	0.03732	2.0254112
8	Test	0.10826	5.7865255
8	Test (no extra)	0.04743	2.0382715

> Table: results presented in the github.

It appears that the values shown in the paper actually represent the training time, not the error metrics. I will contact the authors to confirm this information. According to the Nonlinear Benchmark website, the information is as follows:



Benchmark Results

Method	Paper / Technical Report	Code	Benchmark System	Test Results (RMSE) [multisine, arrow, no extra. arrow]	Training Time	Training Platform	Wh
SUBNET neural state-space encoder	https://arxiv.org/abs/2012.0	https://github.com/GerbenBeintema/SS-encod	Silverbox	[0.36, 1.4, 0.32	1-4 days		Lapt

Overview | Silverbox | Cascaded Tanks

where the values in the "Training time" column matches the ones presented as error metrics in the paper.

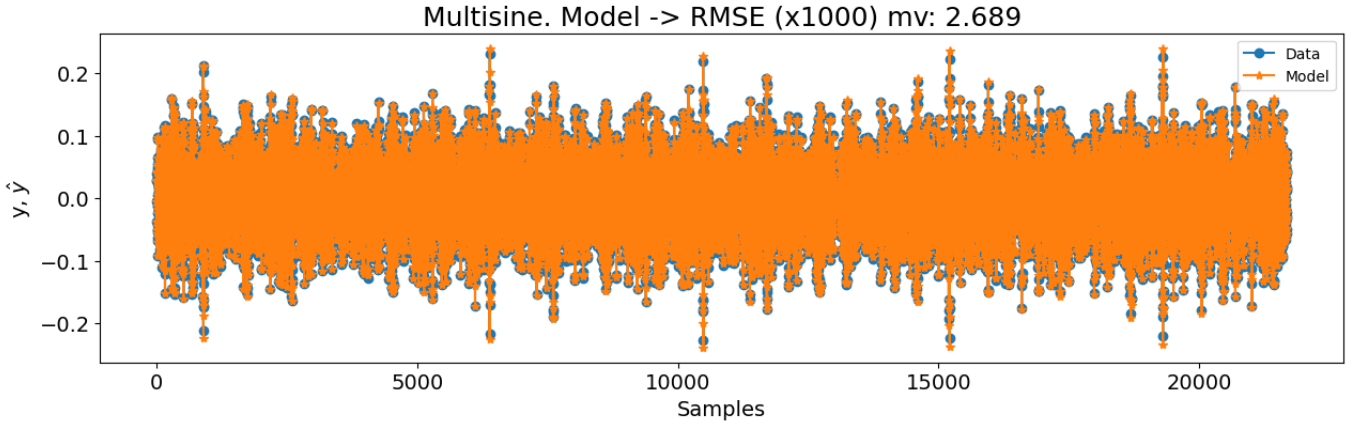
> While we await confirmation of the correct values for this benchmark, we will demonstrate the performance of SysIdentPy. However, we will refrain from making any comparisons or attempting to improve the model at this stage.

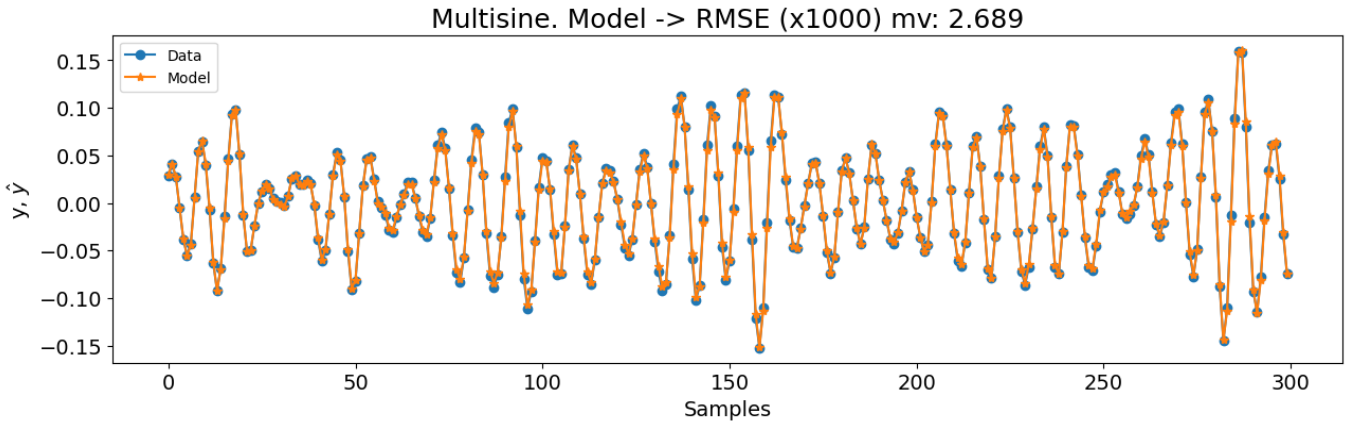
Results

We will start (as we did in every other case study) with a basic configuration of FROLS using a polynomial basis function with degree equal 2. The `xlag` and `ylag` are set to 7 in this first example. Because the dataset is considerably large, we will start with `n_info_values=40`. Because we dealing with a large training dataset, we will use the `err_tol` instead of information criteria to have a faster performance. We will also set `n_terms=40`, which means that the search will stop if the `err_tol` is reached or 40 regressors is tested in the `ERR` algorithm. While this approach might result in a sub-optimal model, it is a reasonable starting point for our first attempt. There are three different experiments: multisine, arrow (full), and arrow (no extrapolation).

```
x_train, y_train = train_val.u, train_val.y
test_multisine, test_arrow_full, test_arrow_no_extrapolation = test
x_test, y_test = test_multisine.u, test_multisine.y

n = test_multisine.state_initialization_window_length

basis_function = Polynomial(degree=2)
```

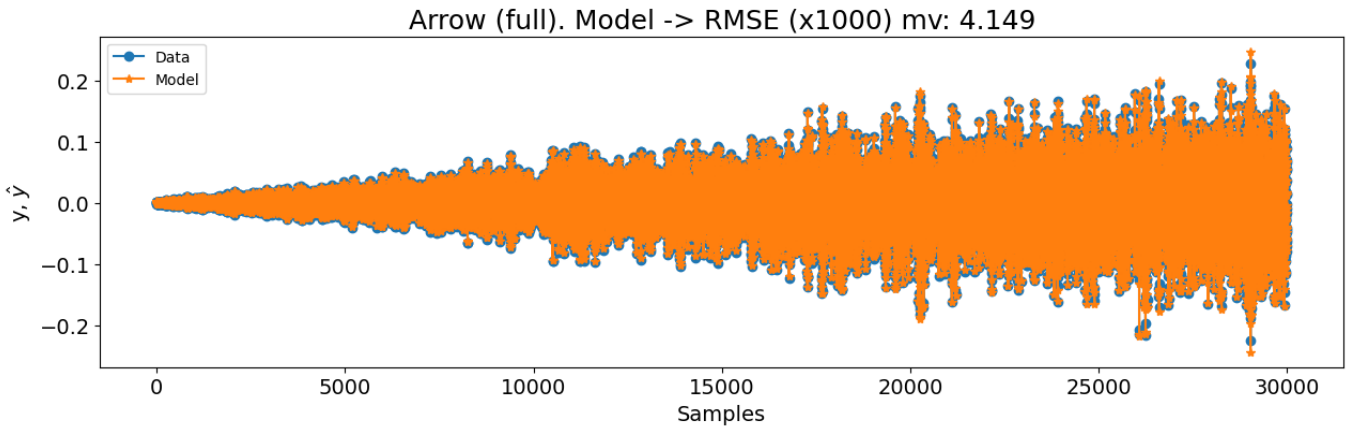
```
model = FROLS(
    xlag=7,
    ylag=7,
    basis_function=basis_function,
    estimator=LeastSquares(),
    err_tol=0.999,
    n_terms=40,
    order_selection=False
)

model.fit(X=x_train, y=y_train)
y_test = np.concatenate([y_train[-model.max_lag:], y_test])
x_test = np.concatenate([x_train[-model.max_lag:], x_test])
yhat = model.predict(X=x_test, y=y_test[:model.max_lag, :])
rmse = root_mean_squared_error(y_test[model.max_lag + n :], yhat[model.max_lag +
n:])
nrmse = rmse/y_test.std()
rmse_mv = 1000 * rmse
print(nrmse, rmse_mv)
plot_results(y=y_test[model.max_lag :], yhat=yhat[model.max_lag :], n=30000,
figsize=(15, 4), title=f"Multisine. Model -> RMSE (x1000) mv: {round(rmse_mv, 4)}")

plot_results(y=y_test[model.max_lag :], yhat=yhat[model.max_lag :], n=300, figsize=
(15, 4), title=f"Multisine. Model -> RMSE (x1000) mv: {round(rmse_mv, 4)}")

> 0.14238040337149 37
> 7.727682109791501
```

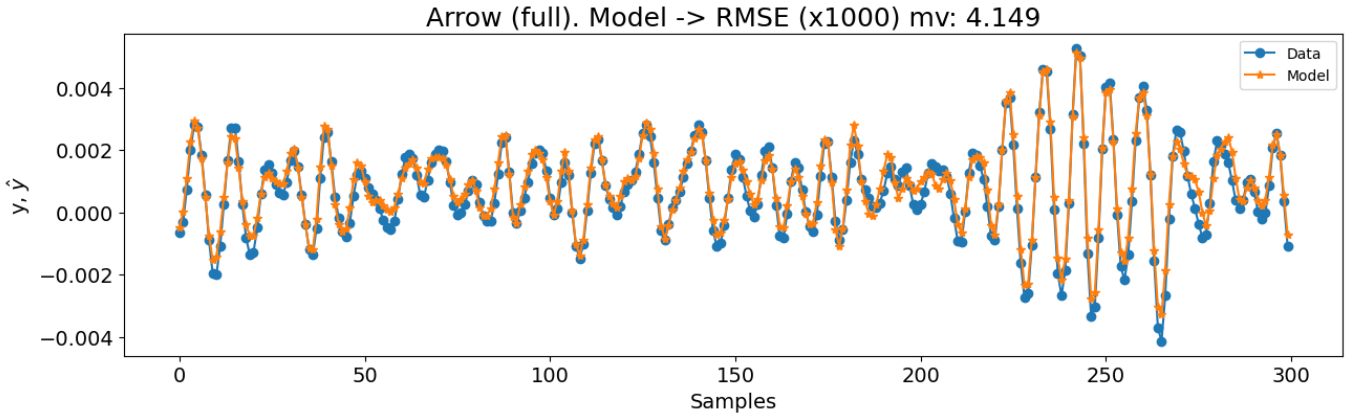





```
x_train, y_train = train_val.u, train_val.y
test_multisine, test_arrow_full, test_arrow_no_extrapolation = test
x_test, y_test = test_arrow_full.u, test_arrow_full.y

n = test_arrow_full.state_initialization_window_length

basis_function = Polynomial(degree=3)
model = FROLS(
    xlag=14,
    ylag=14,
    basis_function=basis_function,
    estimator=LeastSquares(),
    err_tol=0.9999,
    n_terms=80,
    order_selection=False
)

model.fit(X=x_train, y=y_train)
# we will not concatente the last values from train data to use as initial condition
here because
# this test data have a very different behavior.
# However, if you want you can do that and you will see that the model will still
perform
# great after a few iterations
yhat = model.predict(X=x_test, y=y_test[:model.max_lag, :])
rmse = root_mean_squared_error(y_test[model.max_lag + n :], yhat[model.max_lag +
n:])
nrmse = rmse/y_test.std()
rmse_mv = 1000 * rmse

print(nrmse, rmse_mv)

plot_results(y=y_test[model.max_lag :], yhat=yhat[model.max_lag :], n=30000,
figsize=(15, 4), title=f"Arrow (full). Model -> RMSE (x1000) mv: {round(rmse_mv,
4)}")
```

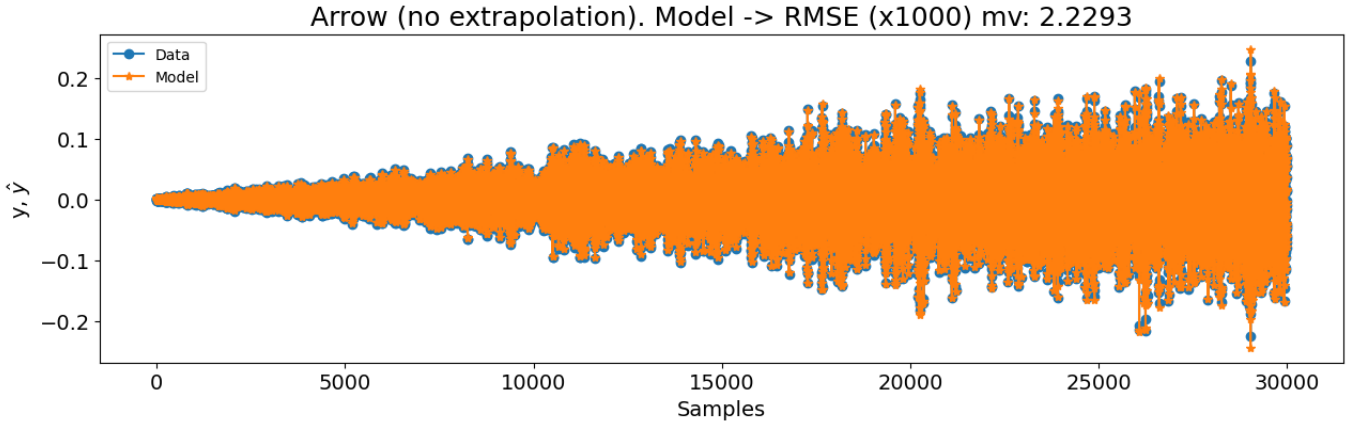
```
plot_results(y=y_test[model.max_lag :], yhat=yhat[model.max_lag :], n=300, figsize=
(15, 4), title=f"Arrow (full). Model -> RMSE (x1000) mv: {round(rmse_mv, 4)}")
```

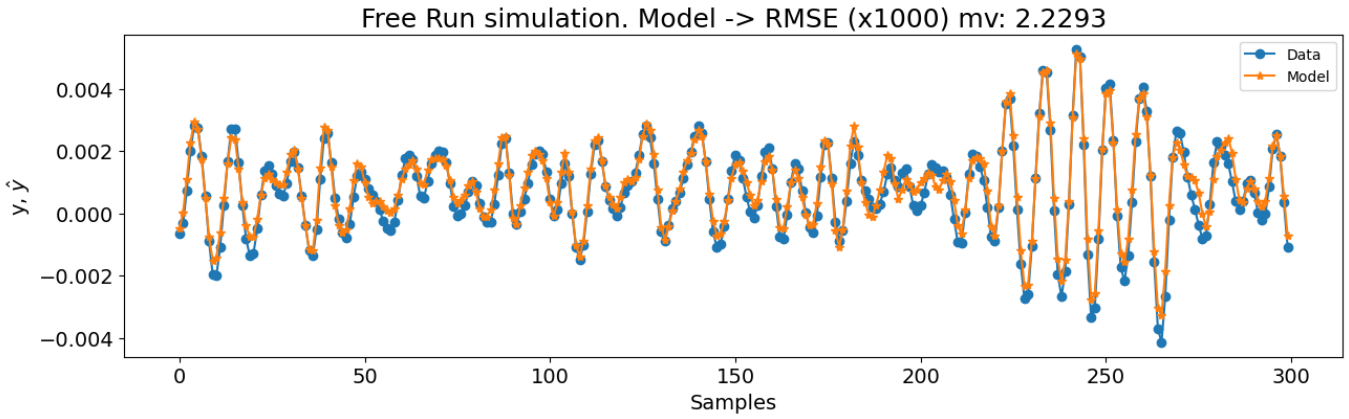





```
x_train, y_train = train_val.u, train_val.y
test_multisine, test_arrow_full, test_arrow_no_extrapolation = test
x_test, y_test = test_arrow_no_extrapolation.u, test_arrow_no_extrapolation.y

n = test_arrow_no_extrapolation.state_initialization_window_length

basis_function = Polynomial(degree=3)
model = FROLS(
    xlag=14,
    ylag=14,
    basis_function=basis_function,
    estimator=LeastSquares(),
    err_tol=0.9999,
    n_terms=40,
    order_selection=False
)

model.fit(X=x_train, y=y_train)
yhat = model.predict(X=x_test, y=y_test[:model.max_lag, :])
rmse = root_mean_squared_error(y_test[model.max_lag + n :], yhat[model.max_lag +
n:])
```

```
nrmse = rmse/y_test.std()
rmse_mv = 1000 * rmse
print(nrmse, rmse_mv)

plot_results(y=y_test[model.max_lag :], yhat=yhat[model.max_lag :], n=30000,
figsize=(15, 4), title=f"Arrow (no extrapolation). Model -> RMSE (x1000) mv:
{round(rmse_mv, 4)}")

plot_results(y=y_test[model.max_lag :], yhat=yhat[model.max_lag :], n=300, figsize=
(15, 4), title=f"Free Run simulation. Model -> RMSE (x1000) mv: {round(rmse_mv,
4)}")
```





F-16 Ground Vibration Test Benchmark

The following examples are intended to demonstrate the application of SysIdentPy on a real-world dataset. Please note that these examples are not aimed at replicating the results presented in the cited manuscripts. The model parameters, such as `ylag` and `xlag`, as well as the size of the identification and validation data sets, differ from those used in the original studies. Additionally, adjustments related to sampling rates and other data preparation steps are not covered in this notebook.

For a comprehensive reference regarding the F-16 Ground Vibration Test benchmark, please visit the nonlinear benchmark website.

> **Note:** This notebook serves as a preliminary demonstration of SysIdentPy's performance on the F-16 dataset. A more detailed analysis will be provided in a future publication. The nonlinear benchmark website offers valuable resources and references related to system identification and machine learning, and readers are encouraged to explore the papers and information available there.

Benchmark Overview

The F-16 Ground Vibration Test benchmark is a notable experiment in the field of system identification and nonlinear dynamics. It involves a high-order system with clearance and friction nonlinearities at the mounting interfaces of payloads on a full-scale F-16 aircraft.

Experiment Details:

- **Event:** Siemens LMS Ground Vibration Testing Master Class
- **Date:** September 2014
- **Location:** Saffraanberg military base, Sint-Truiden, Belgium

During the test, two dummy payloads were mounted on the wing tips of the F-16 to simulate the mass and inertia of real devices typically equipped on the aircraft during flight. Accelerometers were installed on the aircraft structure to capture vibration data. A shaker was placed under the right wing to apply input signals. The key source of nonlinearity in the system was identified as the mounting interfaces of the payloads, particularly the right-wing-to-payload interface, which exhibited significant nonlinear distortions.

Data and Resources:

- **Data Availability:** The dataset, including detailed system descriptions, estimation and test data sets, and setup images, is available for download in both .csv and .mat file formats.
- **Reference:** For in-depth information on the F-16 benchmark, refer to: J.P. Noël and M. Schoukens, "F-16 aircraft benchmark based on ground vibration test data," 2017 Workshop on Nonlinear System Identification Benchmarks, pp. 19-23, Brussels, Belgium, April 24-26, 2017.

The goal of this notebook is to illustrate how SysIdentPy can be applied to such complex datasets, showcasing its capabilities in modeling and analysis. For a thorough exploration of the benchmark and its methodologies, please consult the provided resources and references.

Required Packages and Versions

To ensure that you can replicate this case study, it is essential to use specific versions of the required packages. Below is a list of the packages along with their respective versions needed for running the

case studies effectively.

To install all the required packages, you can create a `requirements.txt` file with the following content:

```
sysidentpy==0.4.0
pandas==2.2.2
numpy==1.26.0
matplotlib==3.8.4
```

Then, install the packages using:

```
pip install -r requirements.txt
```

- Ensure that you use a virtual environment to avoid conflicts between package versions.
- Versions specified are based on compatibility with the code examples provided. If you are using different versions, some adjustments in the code might be necessary.

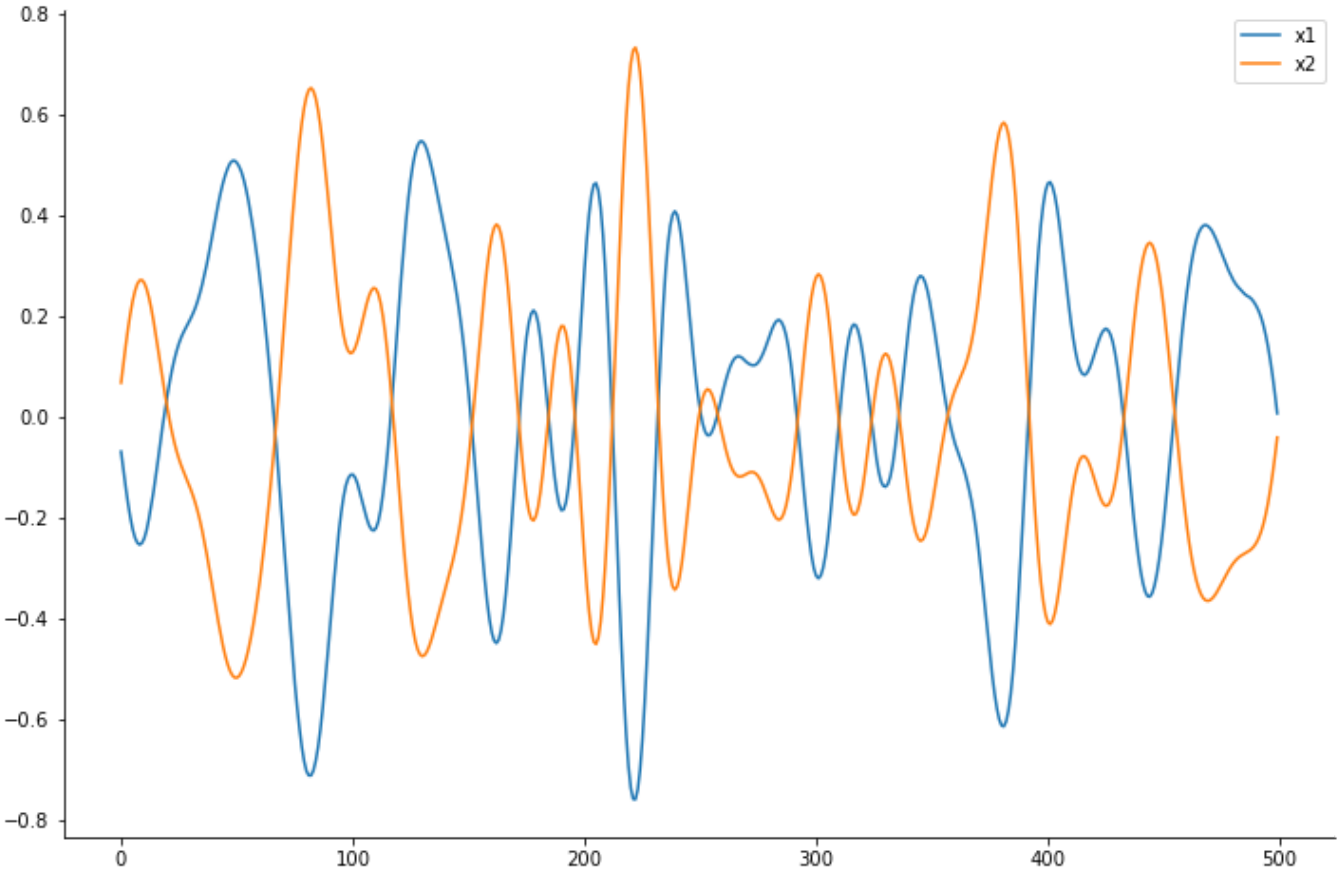
SysIdentPy Configuration

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sysidentpy.model_structure_selection import FROLS
from sysidentpy.basis_function._basis_function import Polynomial
from sysidentpy.parameter_estimation import LeastSquares
from sysidentpy.metrics import root_relative_squared_error
from sysidentpy.utils.display_results import results
from sysidentpy.utils.plotting import plot_residues_correlation, plot_results
from sysidentpy.residues.residues_correlation import (
    compute_residues_autocorrelation,
    compute_cross_correlation,
)
```

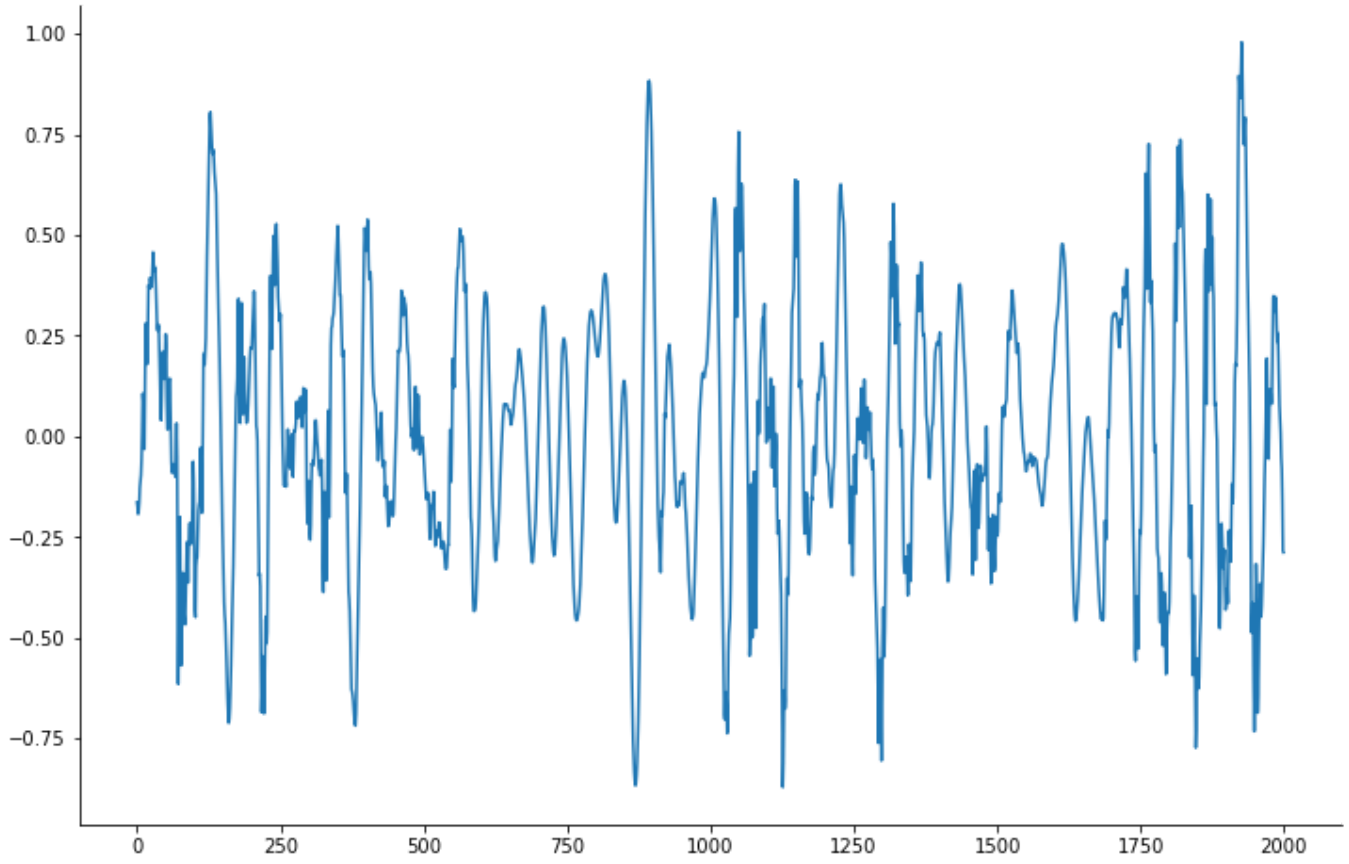
Procedure

```
f_16 = pd.read_csv(r"examples/datasets/f-16.txt", header=None, names=["x1", "x2",
"y"])
f_16.shape
f_16[["x1", "x2"]][0:500].plot(figsize=(12, 8))
```

```
>>> (32768, 3)
```




```
f_16["y"][0:2000].plot(figsize=(12, 8))
```



The following code is to split the dataset into training and test sets

```
x1_id, x1_val = f_16["x1"][0:16384].values.reshape(-1, 1), f_16["x1"][
    16384::
].values.reshape(-1, 1)
x2_id, x2_val = f_16["x2"][0:16384].values.reshape(-1, 1), f_16["x2"][
    16384::
].values.reshape(-1, 1)
x_id = np.concatenate([x1_id, x2_id], axis=1)
x_val = np.concatenate([x1_val, x2_val], axis=1)
y_id, y_val = f_16["y"][0:16384].values.reshape(-1, 1), f_16["y"][
    16384::
].values.reshape(-1, 1)
```

We will set the lags for both inputs as

```
x1lag = list(range(1, 10))
x2lag = list(range(1, 10))
```

and build a NARX model as follows

```
basis_function = Polynomial(degree=1)
estimator = LeastSquares()
```

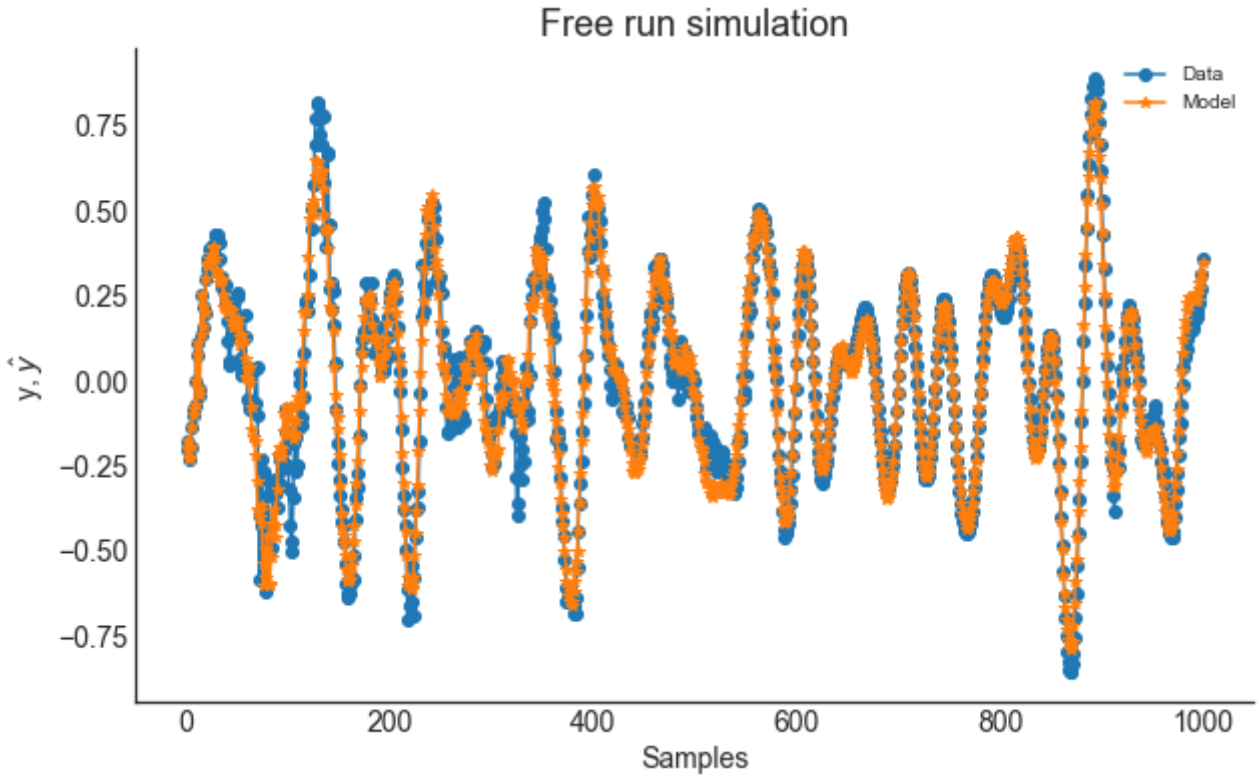
```
model = FROLS(
    order_selection=True,
    n_info_values=39,
    ylag=20,
    xlag=[x1lag, x2lag],
    info_criteria="bic",
    estimator=estimator,
    basis_function=basis_function,
)

model.fit(X=x_id, y=y_id)
y_hat = model.predict(X=x_val, y=y_val)
rrse = root_relative_squared_error(y_val, y_hat)
print(rrse)
r = pd.DataFrame(
    results(
        model.final_model,
        model.theta,
        model.err,
        model.n_terms,
        err_precision=8,
        dtype="sci",
    ),
    columns=["Regressors", "Parameters", "ERR"],
)

print(r)
```

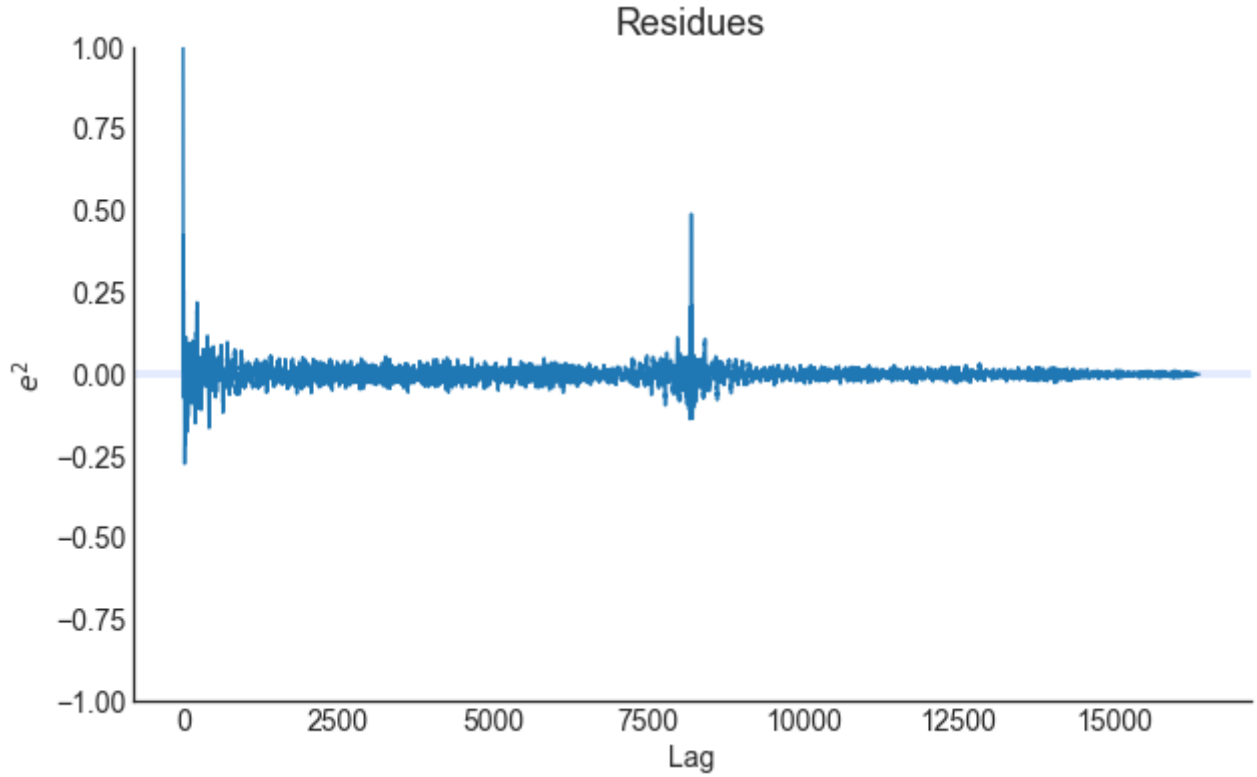
The RRSE is 0.2910

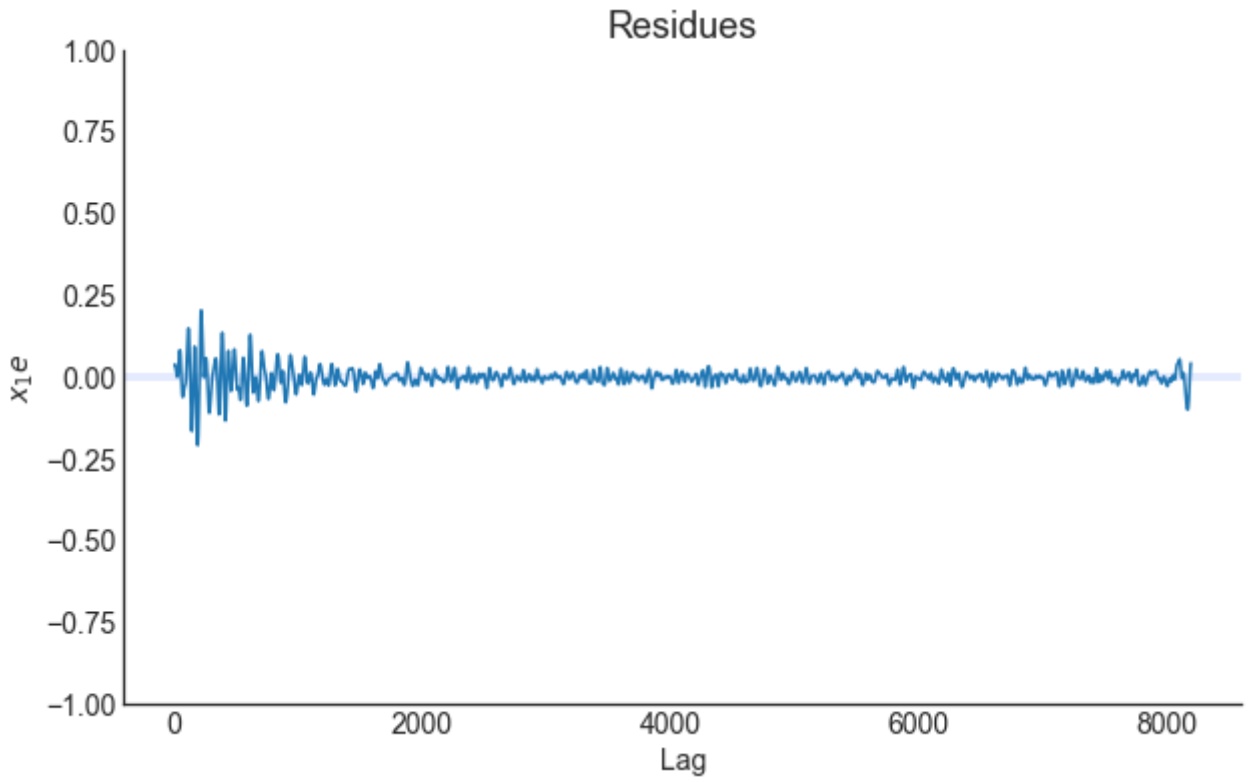
Regressors	Parameters	ERR
y(k-1)	1.8387E+00	9.43378253E-01
y(k-2)	-1.8938E+00	1.95167599E-02
y(k-3)	1.3337E+00	1.02432261E-02
y(k-6)	-1.6038E+00	8.03485985E-03
y(k-9)	2.6776E-01	9.27874557E-04
x2(k-7)	-2.2385E+01	3.76837313E-04
x1(k-1)	8.2709E+00	6.81508210E-04
x2(k-3)	1.0587E+02	1.57459800E-03
x1(k-8)	-3.7975E+00	7.35086279E-04
x2(k-1)	8.5725E+01	4.85358786E-04
y(k-7)	1.3955E+00	2.77245281E-04

Regressors	Parameters	ERR
y(k-5)	1.3219E+00	8.64120037E-04
y(k-10)	-2.9306E-01	8.51717688E-04
y(k-4)	-9.5479E-01	7.23623116E-04
y(k-8)	-7.1309E-01	4.44988077E-04
y(k-12)	-3.0437E-01	1.49743148E-04
y(k-11)	4.8602E-01	3.34613282E-04
y(k-13)	-8.2442E-02	1.43738964E-04
y(k-15)	-1.6762E-01	1.25546584E-04
x1(k-2)	-8.9698E+00	9.76699739E-05
y(k-17)	2.2036E-02	4.55983807E-05
y(k-14)	2.4900E-01	1.10314107E-04
y(k-19)	-6.8239E-03	1.99734771E-05
x2(k-9)	-9.6265E+01	2.98523208E-05
x2(k-8)	2.2620E+02	2.34402543E-04
x2(k-2)	-2.3609E+02	1.04172323E-04
y(k-20)	-5.4663E-02	5.37895336E-05
x2(k-6)	-2.3651E+02	2.11392628E-05
x2(k-4)	1.7378E+02	2.18396315E-05
x1(k-7)	4.9862E+00	2.03811842E-05

```
plot_results(y=y_val, yhat=y_hat, n=1000)
ee = compute_residues_autocorrelation(y_val, y_hat)
plot_residues_correlation(data=ee, title="Residues", ylabel="$e^2$")
x1e = compute_cross_correlation(y_val, y_hat, x_val[:, 0])
plot_residues_correlation(data=x1e, title="Residues", ylabel="$x_1e$")
```


Free run simulation
Data
Model
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
y, ŷ
0
200
400
600
800
1000
Samples


Residues
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
e²
0
2500
5000
7500
10000
12500
15000
Lag



PV Forecasting

In this case study, we evaluate SysIdentPy's capabilities for forecasting solar irradiance data, which can serve as a proxy for solar photovoltaic (PV) production. The objective is to demonstrate that SysIdentPy provides a competitive alternative for time series modeling, rather than claiming superiority over other libraries.

Dataset Overview

The dataset used in this analysis consists of solar irradiance measurements, which are crucial for predicting solar PV production. Solar irradiance refers to the power of solar radiation received per unit area at the Earth's surface, typically measured in watts per square meter (W/m²). Accurate forecasting of solar irradiance is essential for optimizing energy production and managing grid stability in solar power systems.

Dataset Details:

- **Source:** The dataset can be accessed from the NeuralProphet GitHub repository.
- **Time Frame:** The dataset covers a continuous period with frequent measurements.
- **Variables:** Solar irradiance values over time, which will be used to model and forecast future irradiance levels.

Comparison with Other Libraries

To assess the effectiveness of SysIdentPy, we will compare its performance with the NeuralProphet library. NeuralProphet is known for its flexibility and ability to capture complex seasonal patterns and trends, making it a suitable benchmark for this task.

For the comparison, we will use the following methods:

- **NeuralProphet:**
  - The configuration for NeuralProphet models will be based on examples provided in the NeuralProphet documentation. This library employs advanced techniques for capturing temporal patterns and forecasting.
- **SysIdentPy:**
  - **MetaMSS (Meta-heuristic Model Structure Selection):** Utilizes metaheuristic algorithms to determine the optimal model structure.
  - **AOLS (Accelerated Orthogonal Least Squares):** A method designed for selecting relevant regressors in a model.
  - **FROLS (Forward Regression with Orthogonal Least Squares, using polynomial base functions):** A regression technique that incorporates polynomial terms to enhance model selection.

Objective

The goal of this case study is to compare the performance of SysIdentPy's forecasting methods with NeuralProphet. We will specifically focus on:

- **1-Step Ahead Forecasting:** Evaluating the models' ability to predict the next time step in the series based on historical data.

We will train our models on 80% of the dataset and reserve the remaining 20% for validation purposes. This setup ensures that we test the models' performance on unseen data.

Required Packages and Versions

To ensure that you can replicate this case study, it is essential to use specific versions of the required packages. Below is a list of the packages along with their respective versions needed for running the case studies effectively.

To install all the required packages, you can create a `requirements.txt` file with the following content:

```
sysidentpy==0.4.0
pystan==2.19.1.1
holidays==0.11.2
fbprophet==0.7.1
neuralprophet==0.2.7
pandas==1.3.2
numpy==1.23.3
matplotlib==3.8.4
pmdarima==1.8.3
scikit-learn==0.24.2
scipy==1.9.1
sktime==0.8.0
statsmodels==0.12.2
tbats==1.1.0
torch==1.12.1
```

Then, install the packages using:

```
pip install -r requirements.txt
```

- Ensure that you use a virtual environment to avoid conflicts between package versions. This practice isolates your project's dependencies and prevents version conflicts with other projects or system-wide packages. Additionally, be aware that some packages, such as `sktime` and `neuralprophet`, may install several dependencies automatically during their installation. Setting up a virtual environment helps manage these dependencies more effectively and keeps your project environment clean and reproducible.
- Versions specified are based on compatibility with the code examples provided. If you are using different versions, some adjustments in the code might be necessary.

Procedure

1. **Data Preparation:** Load and preprocess the solar irradiance dataset.
2. **Model Training:** Apply the chosen methods from SysIdentPy and NeuralProphet to the training data.
3. **Evaluation:** Assess the forecasting accuracy of each model on the validation set.

By comparing these approaches, we aim to showcase SysIdentPy as a viable option for time series forecasting, highlighting its strengths and versatility in practical applications.

Let's start by importing the necessary libraries and setting up the environment for this analysis.

```
from warnings import simplefilter
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
from sysidentpy.model_structure_selection import FROLS
from sysidentpy.model_structure_selection import AOLS
from sysidentpy.model_structure_selection import MetaMSS
from sysidentpy.basis_function import Polynomial
from sysidentpy.utils.plotting import plot_results
from sysidentpy.neural_network import NARXNN
from sysidentpy.metrics import mean_squared_error
from neuralprophet import NeuralProphet
from neuralprophet import set_random_seed

simplefilter("ignore", FutureWarning)
np.seterr(all="ignore")
%matplotlib inline

loss = mean_squared_error
data_location = r".\datasets"
```

Neural Prophet

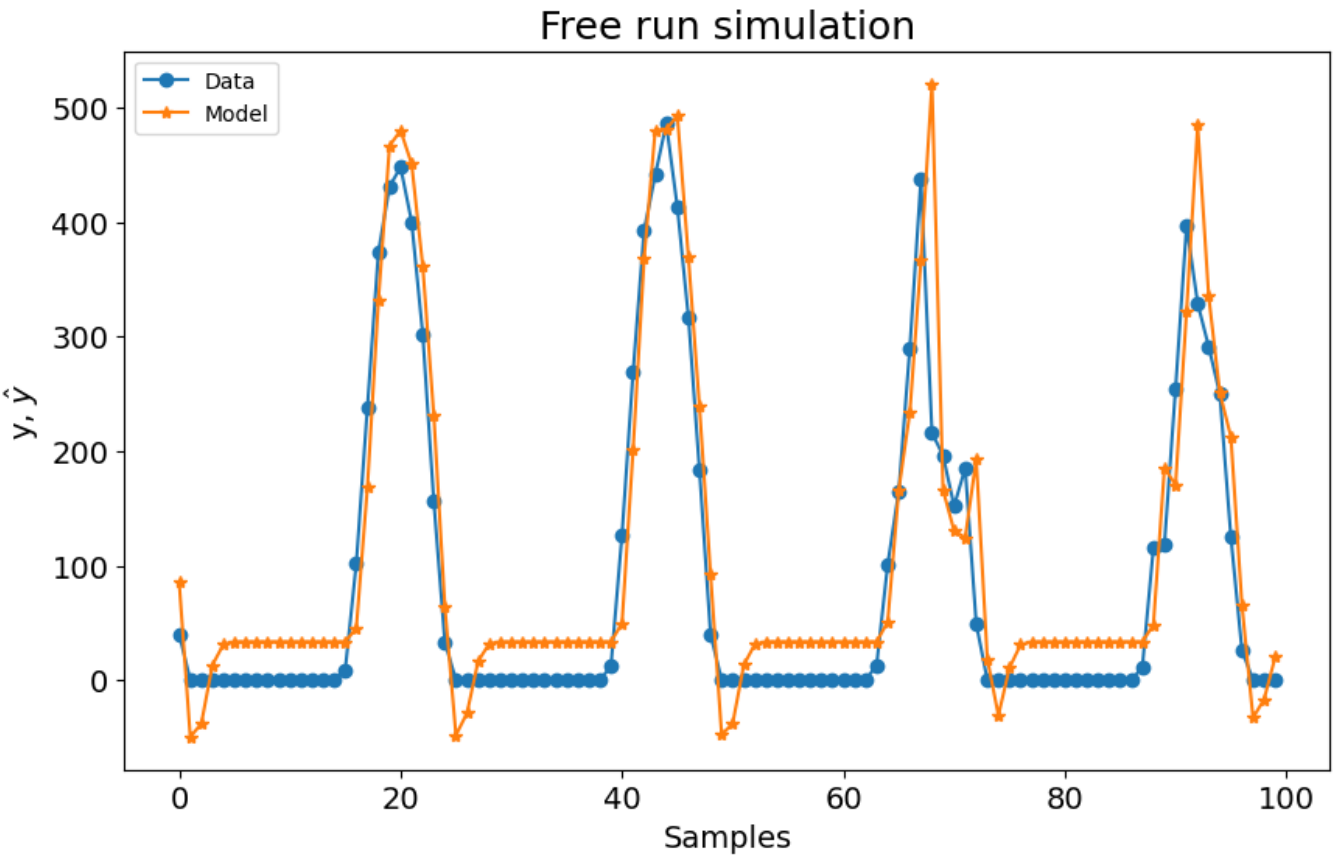
```
set_random_seed(42)
files = ["\SanFrancisco_PV_GHI.csv", "\SanFrancisco_Hospital.csv"]
raw = pd.read_csv(data_location + files[0])
df = pd.DataFrame()
df["ds"] = pd.date_range("1/1/2015 1:00:00", freq=str(60) + "Min", periods=(8760))
df["y"] = raw.iloc[:, 0].values

m = NeuralProphet(
    n_lags=24,
    ar_sparsity=0.5,
)

metrics = m.fit(df, freq="H", valid_p=0.2)
df_train, df_val = m.split_df(df, valid_p=0.2)
m.test(df_val)

future = m.make_future_dataframe(df_val, n_historic_predictions=True)
forecast = m.predict(future)

print(loss(forecast["y"][24:-1], forecast["yhat1"][24:-1]))

plt.plot(forecast["y"][-104:], "ro-")
plt.plot(forecast["yhat1"][-104:], "k*-")
```

The error is $MSE = 4642.23$ and will be used as baseline in this case. Lets check how SysIdentPy methods handle this data.

FROLS

```
files = ["\SanFrancisco_PV_GHI.csv", "\SanFrancisco_Hospital.csv"]
raw = pd.read_csv(data_location + files[0])
df = pd.DataFrame()
df["ds"] = pd.date_range("1/1/2015 1:00:00", freq=str(60) + "Min", periods=(8760))
df["y"] = raw.iloc[:, 0].values
df_train, df_val = df.iloc[:7008, :], df.iloc[7008:, :]
y = df["y"].values.reshape(-1, 1)
y_train = df_train["y"].values.reshape(-1, 1)
y_test = df_val["y"].values.reshape(-1, 1)
x_train = df_train["ds"].dt.hour.values.reshape(-1, 1)
x_test = df_val["ds"].dt.hour.values.reshape(-1, 1)

basis_function = Polynomial(degree=1)
sysidentpy = FROLS(
    order_selection=True,
    ylag=24,
    xlag=24,
    info_criteria="bic",
    basis_function=basis_function,
    model_type="NARMAX",
    estimator=LeastSquares(),
)

sysidentpy.fit(X=x_train, y=y_train)
x_test = np.concatenate([x_train[-sysidentpy.max_lag :], x_test])
y_test = np.concatenate([y_train[-sysidentpy.max_lag :], y_test])
yhat = sysidentpy.predict(X=x_test, y=y_test, steps_ahead=1)
sysidentpy_loss = loss(
    pd.Series(y_test.flatten()[sysidentpy.max_lag :]),
    pd.Series(yhat.flatten()[sysidentpy.max_lag :]),
)

print(sysidentpy_loss)
plot_results(y=y_test[-104:], yhat=yhat[-104:])
```

The $MSE = 3869.34$ for this case.

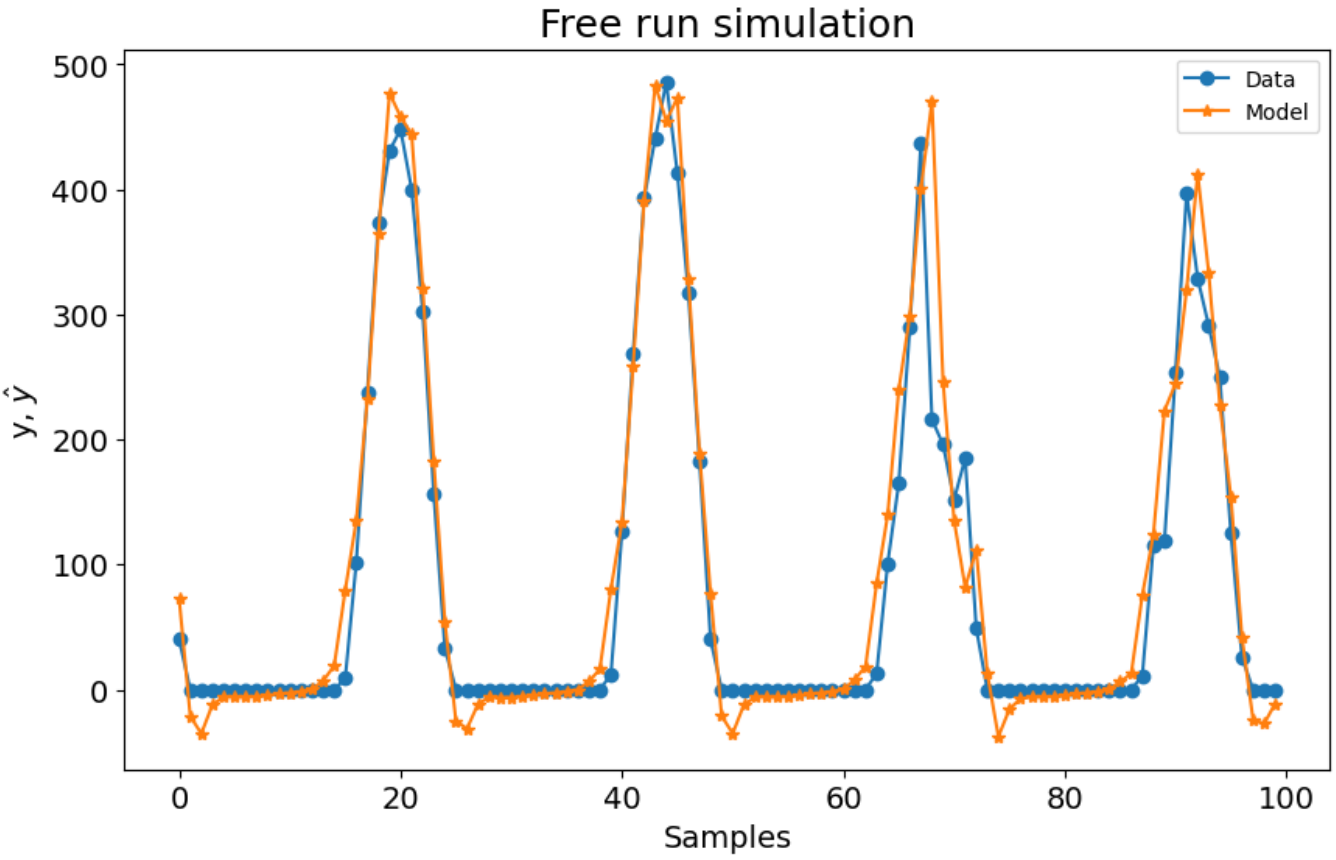


MetaMSS

```
set_random_seed(42)
files = ["\SanFrancisco_PV_GHI.csv", "\SanFrancisco_Hospital.csv"]
raw = pd.read_csv(data_location + files[0])
df = pd.DataFrame()
df["ds"] = pd.date_range("1/1/2015 1:00:00", freq=str(60) + "Min", periods=(8760))
df["y"] = raw.iloc[:, 0].values
df_train, df_val = df.iloc[:7008, :], df.iloc[7008:, :]
y = df["y"].values.reshape(-1, 1)
y_train = df_train["y"].values.reshape(-1, 1)
y_test = df_val["y"].values.reshape(-1, 1)
x_train = df_train["ds"].dt.hour.values.reshape(-1, 1)
x_test = df_val["ds"].dt.hour.values.reshape(-1, 1)

basis_function = Polynomial(degree=1)
estimator = LeastSquares()

sysidentpy_metamss = MetaMSS(
    basis_function=basis_function,
    xlag=24,
    ylag=24,
    estimator=estimator,
    maxiter=10,
```

```
    steps_ahead=1,
    n_agents=15,
    loss_func="metamss_loss",
    model_type="NARMAX",
    random_state=42,
)

sysidentpy_metamss.fit(X=x_train, y=y_train)
x_test = np.concatenate([x_train[-sysidentpy_metamss.max_lag :], x_test])
y_test = np.concatenate([y_train[-sysidentpy_metamss.max_lag :], y_test])
yhat = sysidentpy_metamss.predict(X=x_test, y=y_test, steps_ahead=1)
metamss_loss = loss(
    pd.Series(y_test.flatten()[sysidentpy_metamss.max_lag :]),
    pd.Series(yhat.flatten()[sysidentpy_metamss.max_lag :]),
)

print(metamss_loss)
plot_results(y=y_test[-104:], yhat=yhat[-104:])
```

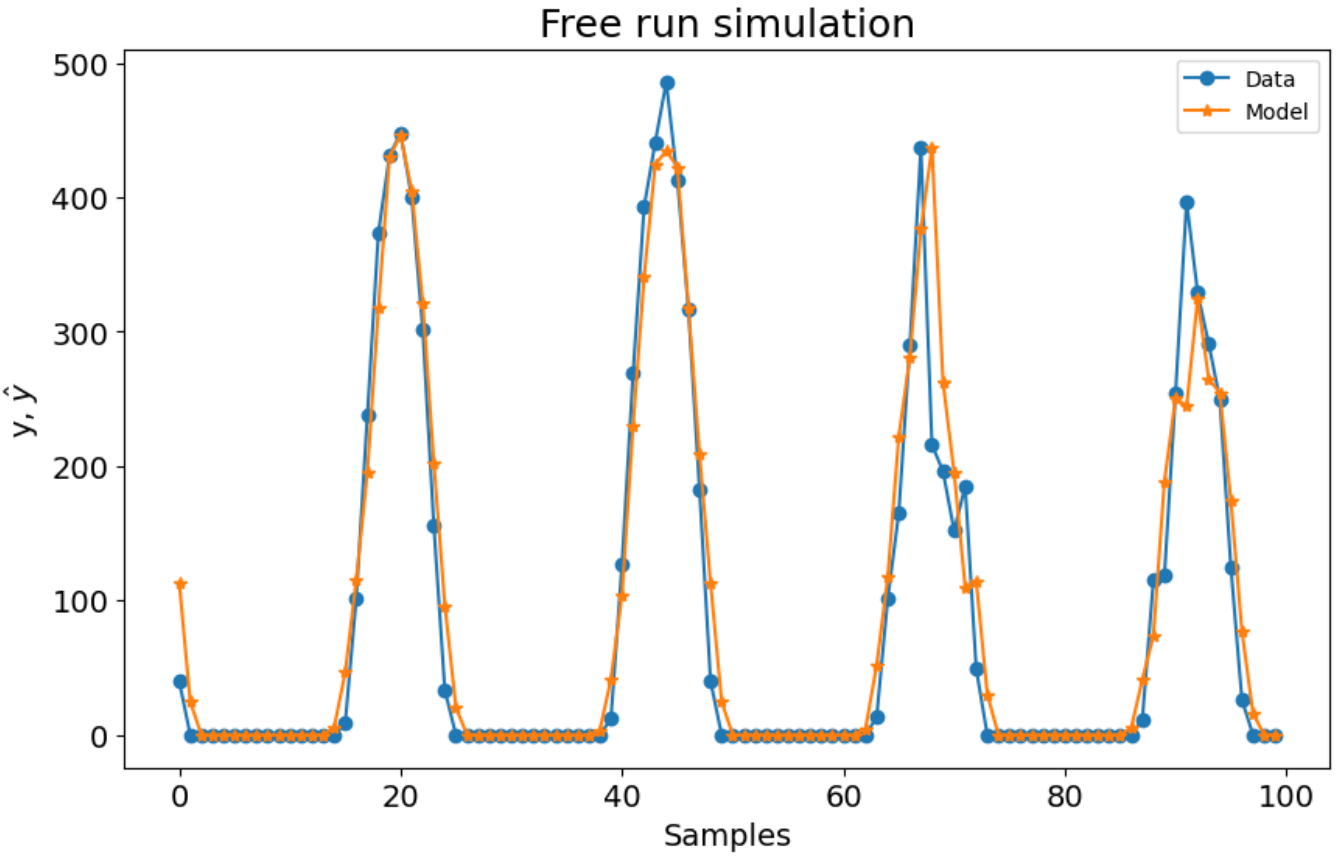
The MetaMSS algorithm was able to select a better model in this case, as can be observed in the error metric, $MSE = 2157.77$.



AOLS

```
set_random_seed(42)
files = ["\SanFrancisco_PV_GHI.csv", "\SanFrancisco_Hospital.csv"]
raw = pd.read_csv(data_location + files[0])
df = pd.DataFrame()
df["ds"] = pd.date_range("1/1/2015 1:00:00", freq=str(60) + "Min", periods=(8760))
df["y"] = raw.iloc[:, 0].values
df_train, df_val = df.iloc[:7008, :], df.iloc[7008:, :]
y = df["y"].values.reshape(-1, 1)
y_train = df_train["y"].values.reshape(-1, 1)
y_test = df_val["y"].values.reshape(-1, 1)
x_train = df_train["ds"].dt.hour.values.reshape(-1, 1)
x_test = df_val["ds"].dt.hour.values.reshape(-1, 1)

basis_function = Polynomial(degree=1)
sysidentpy_AOLS = AOLS(
    ylag=24, xlag=24, k=2, L=1, model_type="NARMAX", basis_function=basis_function
)

sysidentpy_AOLS.fit(X=x_train, y=y_train)
x_test = np.concatenate([x_train[-sysidentpy_AOLS.max_lag :], x_test])
y_test = np.concatenate([y_train[-sysidentpy_AOLS.max_lag :], y_test])
yhat = sysidentpy_AOLS.predict(X=x_test, y=y_test, steps_ahead=1)
aols_loss = loss(
    pd.Series(y_test.flatten()[sysidentpy_AOLS.max_lag :]),
    pd.Series(yhat.flatten()[sysidentpy_AOLS.max_lag :]),
)
print(aols_loss)
plot_results(y=y_test[-104:], yhat=yhat[-104:])
```

The error now is $MSE = 2361.56$.


Free run simulation
Data
Model
500
400
300
200
100
0
y, ŷ
0
20
40
60
80
100
Samples