ML for Process Industry Series

Machine Learning in Python for Dynamic Process Systems

A practitioner's guide for building process modeling, predictive, and monitoring solutions using dynamic data

First Edition

Ankur Kumar, Jesus Flores-Cerrillo

`

Machine Learning in Python for Dynamic Process Systems

A practitioner’s guide for building process modeling, predictive, and monitoring solutions using dynamic data

Ankur Kumar
Jesus Flores-Cerrillo

`

Dedicated to our spouses, family, friends, motherland, and all the data-science enthusiasts

न चोर हार्यं न च राज हार्यं न भात्रू भाज्यं न च भारकारि

व्ययं कृते वर्धत एव नित्यं विद्याधनं सर्वधनप्रधानम

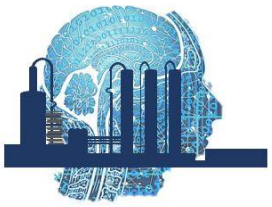
No one can steal it, no king can snatch it,
It cannot be divided among the brothers and it's not heavy to carry,
As you consume or spend, it increases; as you share, it expands,
The wealth of knowledge is the most precious wealth you can have.

- *A popular Sanskrit shloka*

`

Machine Learning in Python for Dynamic Process Systems

www.MLforPSE.com



Plant image on cover page obtained from https://pixabay.com/.

.

To request permissions, contact the authors at MLforPSE@gmail.com

First published: June 2023

`

About the Authors

Ankur Kumar holds a PhD degree (2016) in Process Systems Engineering from the University of Texas at Austin and a bachelor's degree (2012) in Chemical Engineering from the Indian Institute of Technology Bombay. He currently works at Linde in the Advanced Digital Technologies & Systems Group in Linde's Center of Excellence, where he has developed several in-house machine learning-based monitoring and process control solutions for Linde's hydrogen and air-separation plants. Ankur's tools have won several awards both within and outside Linde. Most recently, one of his tools, PlantWatch (a plantwide fault detection and diagnosis tool), received the 2021 Industry 4.0 Award by the Confederation of Industry of the Czech Republic. Ankur has authored or co-authored more than 10 peer-reviewed journal papers (in the areas of data-driven process modeling and optimization), is a frequent reviewer for many top-ranked research journals, and has served as Session Chair at several international conferences. Ankur also served as an Associate Editor of the Journal of Process Control from 2019 to 2021.

Jesus Flores-Cerrillo is currently an Associate Director - R&D at Linde and manages the Advanced Digital Technologies & Systems Group in Linde's Center of Excellence. He has over 20 years of experience in the development and implementation of monitoring technologies and advanced process control & optimization solutions. Jesus holds a PhD degree in Chemical Engineering from McMaster University and has authored or co-authored more than 40 peer-reviewed journal papers in the areas of multivariate statistics and advanced process control among others. His team develops and implements novel plant monitoring, machine learning, IIOT solutions to improve the efficiency and reliability of Linde's processes. Jesus's team received the Artificial Intelligence and Advanced Analytics Leadership 2020 award from the National Association of Manufacturers' Manufacturing Leadership Council.

`

Note to the readers

Jupyter notebooks and Spyder scripts with complete code implementations are available for download at https://github.com/ML-PSE/Machine_Learning_for_DPS. Code updates when necessary, will be made and updated on the GitHub repository. Updates to the book's text material will be available on Leanpub (www.leanpub.com) and Google Play (https://play.google.com/store/books). We would greatly appreciate any information about any corrections and/or typos in the book.

`

Series Introduction

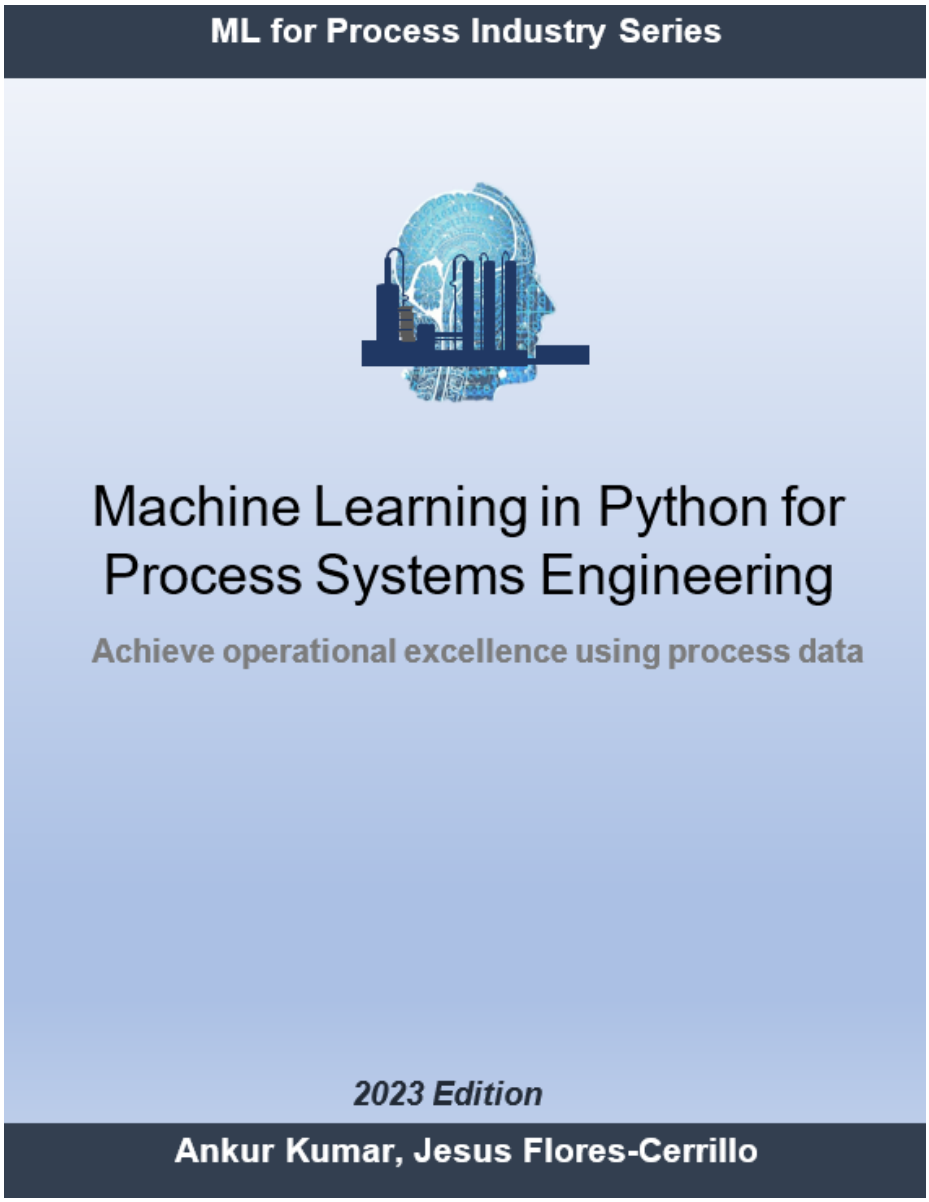
In the 21st century, data science has become an integral part of the work culture at every manufacturing industry and process industry is no exception to this modern phenomenon. From predictive maintenance to process monitoring, fault diagnosis to advanced process control, machine learning-based solutions are being used to achieve higher process reliability and efficiency. However, few books are available that adequately cater to the needs of budding process data scientists. The scant available resources include: 1) generic data science books that fail to account for the specific characteristics and needs of process plants 2) process domain-specific books with rigorous and verbose treatment of underlying mathematical details that become too theoretical for industrial practitioners. Understandably, this leaves a lot to be desired. Books are sought that have process systems in the backdrop, stress application aspects, and provide a guided tour of ML techniques that have proven useful in process industry. This series 'Machine Learning for Process Industry' addresses this gap to reduce the barrier-to-entry for those new to process data science.

The first book of the series '***Machine Learning in Python for Process Systems Engineering***' covers the basic foundations of machine learning and provides an overview of broad spectrum of ML methods primarily suited for static systems. Step-by-step guidance on building ML solutions for process monitoring, soft sensing, predictive maintenance, etc. are provided using real process datasets. Aspects relevant to process systems such as modeling correlated variables via PCA/PLS, handling outliers in noisy multidimensional dataset, controlling processes using reinforcement learning, etc. are covered. This second book of the series is focused on dynamic systems and provides a guided tour along the wide range of available dynamic modeling choices. Emphasis is paid to both the classical methods (ARX, CVA, ARMAX, OE, etc.) and modern neural network methods. Applications on time series analysis, noise modeling, system identification, and process fault detection are illustrated with examples. Future books of the series will continue to focus on other aspects and needs of process industry. It is hoped that these books can help process data scientists find innovative ML solutions to the real-world problems faced by the process industry.

Books of the series will be useful to practicing process engineers looking to 'pick up' machine learning as well as data scientists looking to understand the needs and characteristics of process systems. With the focus on practical guidelines and real industrial case studies, we hope that these books lead to wider spread of data science in the process industry.

`

Previous book(s) from the series
(https://MLforPSE.com/books/)



`

Preface

Model predictive control (MPC) and real-time optimization (RTO) are among the most critical technologies that drive the process industry. Any experienced process engineer would vouch that the success of MPCs and RTOs depend heavily on the accuracy of the underlying process models. Similarly, the key requirement for other important process technologies like dynamic data reconciliation, process monitoring, etc. is availability of accurate process models. Modeling efforts during commissioning of these tools can easily consume up to 90% of the project cost and time. Modern data revolution, whereby abundant amount of process data are easily available, and practical difficulties in building high-fidelity first-principles dynamic models for complex industrial processes have popularized the usage of empirical data-driven/machine learning (ML) models. Building dynamic models using process data is called system identification (SysID) and it's a very mature field with extensive literature. However, it is also very easy for a process data scientist (PDS) new to this field to get overwhelmed with the SysID mathematics and 'drowned' in the sea of SysID terminology. Several noteworthy ML books have been written on time-series analysis; however, in process industry, input-output models are of greater import. Unfortunately, there aren't many books that cater to the needs of modern PDSs interested in dynamic process modeling (DPM) without weighing them down with too much mathematical details and therein lies our motivation for authoring this book: specifically, a reader-friendly and easy to understand book that provides a comprehensive coverage of ML techniques that have proven useful for building dynamic process models with focus on practical implementations.

It would be clear to you by now that this book is designed to teach working process engineers and budding PDSs about DPM. While doing so, this book attempts to avoid a pitfall that several generic ML books fall into: overemphasis on 'modern' and complex ML techniques such as artificial neural networks (ANNs) and undertreatment of classical DPM methods. Classical techniques like FIR and ARX still dominate the dynamic modeling solutions offered by commercial vendors of industrial solutions and are no less 'machine-learning' than the ANNs. These two along with other classical techniques (such as OE, ARIMAX, CVA) predate the ANN-craze era and have stood the test of time in providing equal (if not superior) performance compared to ANNs. Correspondingly, along with modern ML techniques like RNNs, considerable portion of the book is devoted to classical dynamic models with the emphasis on understanding the implications of modeling decisions such as the impact of implicit noise model assumption with ARX model, the implication of differencing data, etc.

`

Guided by our own experience from building process models for varied industrial applications over the past several years, this book covers a curated set of ML techniques that have proven useful for DPM. The broad objectives of the book can be summarized as follows:

- reduce barrier-to-entry for those new to the field of SysID
- provide working-level knowledge of SysID techniques to the readers
- enable readers to make judicious selection of a SysID technique appropriate for their problems through intuitive understanding of the advantages and drawbacks of different methods
- provide step-by-step guidance for developing tools for soft sensing, process monitoring, predictive maintenance, etc. using SysID methods

This book adopts a tutorial-style approach. The focus is on guidelines and practical illustrations with a delicate balance between theory and conceptual insights. Hands-on-learning is emphasized and therefore detailed code examples with industrial-scale datasets are provided to concretize the implementation details. A deliberate attempt is made to not weigh readers down with mathematical details, but rather use it as a vehicle for better conceptual understanding. Complete code implementations have been provided in the GitHub repository. Although most of the existing literature on SysID use MATLAB as the programming environment, we have adopted Python in this book due to its immense popularity among the broad ML community. Several Python libraries are now available which makes DPM using Python convenient.

We are quite confident that this text will enable its readers to build dynamic models for challenging problems with confidence. We wish them the best of luck in their career.

Who should read this book

The application-oriented approach in this book is meant to give a quick and comprehensive coverage of dynamic modeling methodologies in a coherent, reader-friendly, and easy-to-understand manner. The following categories of readers will find the book useful:

1) Data scientists new to the field of system identification
2) Regular users of commercial process modeling software looking to obtain a deeper understanding of the underlying concepts
3) Practicing process data scientists looking for guidance for developing process modeling and monitoring solutions for dynamic systems
4) Process engineers or process engineering students making their entry into the world of data science

`

Pre-requisites

No prior experience with machine learning or Python is needed. Undergraduate-level knowledge of basic linear algebra and calculus is assumed.

Book organization

Under the broad theme of ML for process systems engineering, this book is an extension of the first book of the series (which dealt with fundamentals of ML and its varied applications in process industry); however, it can also be used as a standalone text. To give due treatment to various aspects of SysID, the book has been divided into three parts**. Part 1** of the book provides a perspective on the importance of ML for dynamic process modeling and lays down the basic foundations of ML-DPM (machine learning for dynamic process modeling). **Part 2** provides in-detail presentation of classical ML techniques and has been written keeping in mind the different modeling requirements and process characteristics that determine a model's suitability for a problem at hand. These include, amongst others, presence of multiple correlated outputs, process nonlinearity, need for low model bias, need to model disturbance signal accurately, etc. **Part 3** is focused on artificial neural networks and deep learning. While deep learning is the current buzzword in ML community, we would like to caution the reader against the temptation to deploy a deep learning model for every problem at hand. For example, the models covered in Part 2 still dominate the portfolio of models used in industrial controllers and can often provide comparable (or even superior) performance compared to ANNs with relatively less hassle.

Symbol notation

The following notation has been adopted in the book for representing different types of variables:

- lower-case letters refer to vectors ($x \in \mathbb{R}^{m\times 1}$) and upper-case letters denote matrices ($X \in \mathbb{R}^{n\times m}$)
- individual element of a vector and a matrix are denoted as x_j and x_{ij}, respectively.
- any i^{th} vector in a dataset gets represented as subscripted lower-case letter (e.g., $x_i \in \mathbb{R}^{m\times 1}$). Its distinction from an individual element x_j would be clear from the corresponding context.

`

Installing Required Packages

In this book, SIPPY package (https://github.com/CPCLAB-UNIPI/SIPPY) is used for classical dynamic modeling of input-output systems. Below are a few remarks regarding the installation of the package

- The package code can be downloaded from its GitHub repository. You may put the ‘sippy’ folder provided in the package’s distribution in your working directory to use the provided models
- To run the code provided in this book, Slycot package is not required
- Control package’s version should be < 0.9 (https://github.com/CPCLAB-UNIPI/SIPPY/issues/48)

`

Table of Contents

`

`

Part 1

Introduction & Fundamentals

Chapter 1

Machine Learning and Dynamic Process Modeling: An Introduction

Process industry operations are dynamic in nature. In complex process plants such as oil refineries, 1000s of process measurements may be recorded every second to capture crucial process trends. Plant engineers often employ dynamic process models to predict future values of process variables in applications such as process control, process monitoring, etc. Machine learning (ML) provides a convenient mechanism to bring together the dynamic process modeling (DPM) needs and the large data resources available in modern process plants.

This chapter provides an overview of what ML has to offer for DPM. This chapter also addresses a dichotomy between ML community and DPM community. While the former generally tends to claim that ML has been reduced to mere execution of 'model-fitting' actions, the later vouches for the need of 'experts' to build 'successful' models. We will attempt to reconcile these two differing viewpoints.

Overall, this chapter provides a whirlwind tour of how the power of machine learning is harnessed for dynamic process modeling. Specifically, the following topics are covered

- Introduction to dynamic process modeling and the need for machine learning
- Typical workflow in a ML-based DPM (ML-DPM) project
- Taxonomy of popular ML-DPM methods and models
- Applications of DPM in process industry

Let's now tighten our seat-belts as we embark upon this exciting journey of de-mystifying machine learning for dynamic process modeling.

`

1.1 Process Systems Engineering, Dynamic Process Modeling, and Machine Learning

Process industry is a parent term used to refer to industries like petrochemical, chemical, power, paper, cement, pharmaceutical, etc. These industries use processing plants to manufacture intermediate or final consumer products. As emphasized in Figure 1.1, the prime concern of the management of these plants is optimal design and operations, and high reliability through proactive process monitoring, quality control, data reconciliation, etc. All these tasks fall under the ambit of process systems engineering (PSE).

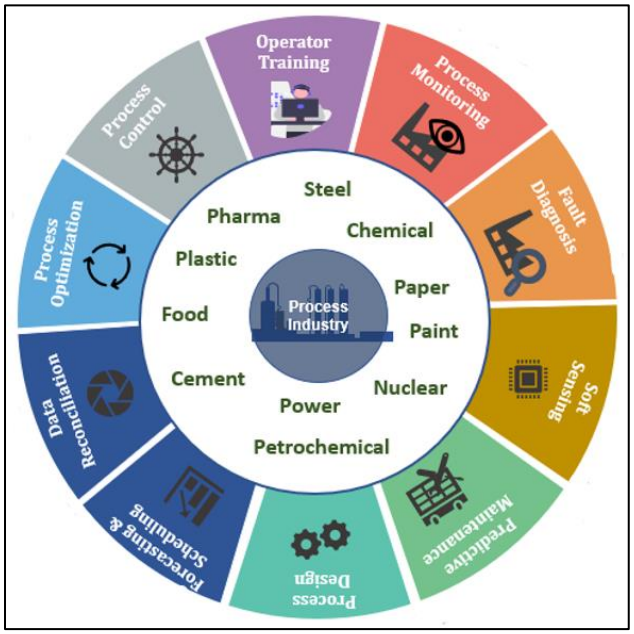


Figure 1.1: Overview of industries that constitute process industry and the tasks process systems engineers perform

The common theme among the PSE tasks is that they all rely on reliable dynamic mathematical process models that can accurately predict the future state of the process using current and past process measurements. Therefore, DPM is one of the defining skills of process systems engineers. Process industry has historically utilized both first principles/phenomenological and empirical/data-based models for DPM. While the former models provide higher fidelity, the later models are easier to build for complex systems. The immense rise in popularity of machine learning/data science in recent years and exponential increase in sensor measurements collected at plants have led to renewed interest in ML-based DPM. Several classical and ‘modern’ ML methods for DPM are at a process data scientist’s disposal. The rest of the book will take you on a whirlwind tour of these methods. Let’s now jump straight into the nitty-gritty of ML-DPM.

`

Components of a dynamic process model

In the context of DPM, there are three types of variables/signals in a process system as shown in Figure 1.2. The measured signals that we desire to predict are termed outputs while the measured signals that can be manipulated to influence the outputs are called inputs. In industrial processes, these outputs are seldom solely influenced or completely explained by inputs. This unexplained component is attributed to the noise signals which can manifest as measurement noise or process noise (unmeasured disturbances or unmeasured signals that impact the system are often the cause of process noise). In the pH neutralization process example shown below, process noise corresponds to unmeasured fluctuations in the wastewater acidity and/or flow which causes disturbances in the process output.

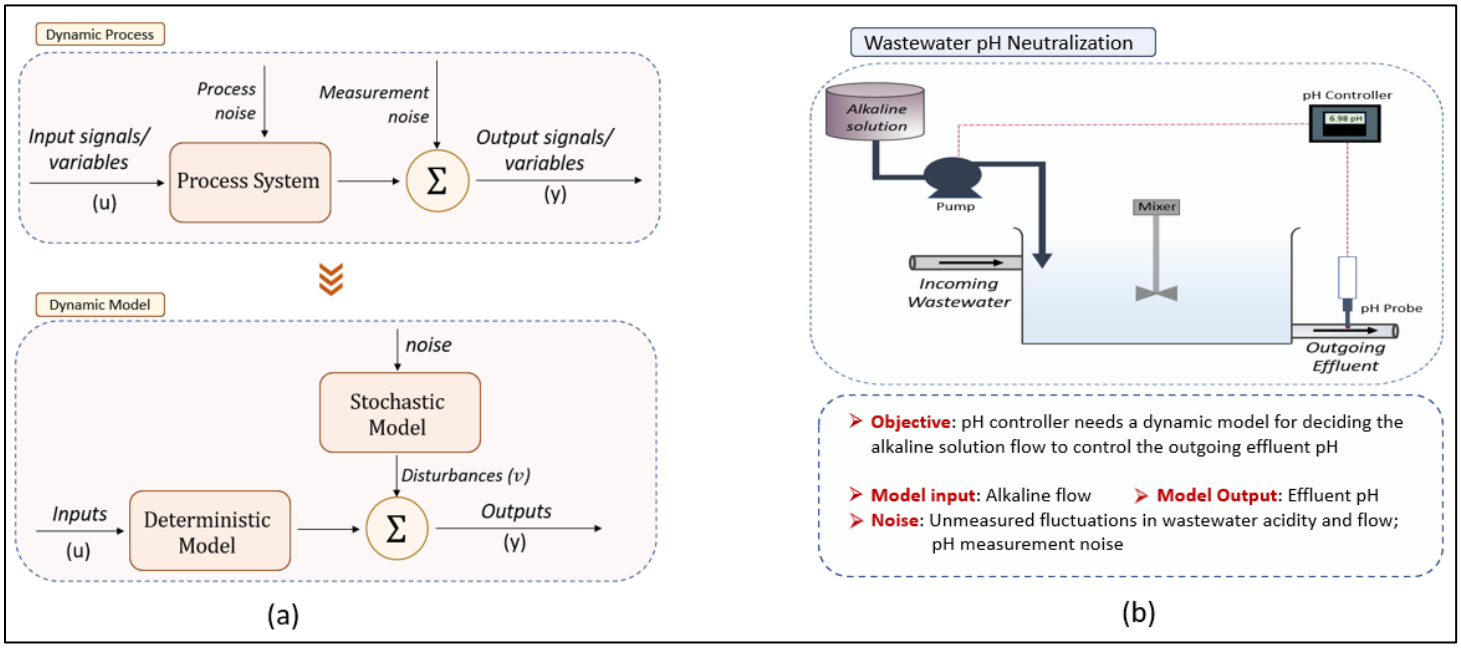


Figure 1.2: (a) Process system (and its dynamic model) with input, output, and noise signals (b) A pH neutralization process broken down into its dynamic components

The impact of measurement and process noise can be clubbed together and explained via a stochastic model as shown in the figure above. The resulting disturbance signal summarizes all the uncertain characteristics of the process. The stochastic and deterministic models can be estimated simultaneously as well as separately.

In the engineering community, the task of building mathematical models of a dynamic system using measured data is called system identification (SysID). Additionally, the task of modeling dynamic systems without input variables is termed time-series analysis.

`

Dynamic modeling notation

In this book, we will focus on time-invariant discrete-time models wherein the signals will be assumed to be recorded at regular sampling interval of *T* time units. The output *y* at time *kT* (*k* is any integer) will be denoted as *y(k)*. System identification therefore entails finding the relationship between the three signals *y(k), u(k), and v(k)*. For the pH example, a simple discrete-time linear model could look like the following

$$\hat{y}_{pH}(k) = \alpha\hat{y}_{pH}(k-1) + \beta u_{alkaline}(k-1)$$
$$y_{pH}(k) = \hat{y}_{pH}(k) + v(k) \qquad \text{eq. 1}$$

where α and β are estimable model parameters and the disturbance variable, *v(k)*, summarizes all the uncertainties. Here, the value of the output at the k^{th} time instant is predicated upon the past values of both output and input, and the disturbance. In the later chapters, we will study how to characterize *v(k)* using stochastic models.

We cannot emphasize enough the importance of proper handling of process disturbance signals. When in hurry, you may be tempted to ignore the stochastic component. However, doing so would only be inviting disappointment as you may end up with biased deterministic models with unsatisfactory performance. This interlink between stochastic and deterministic models would probably not be obvious to beginner PDSs. By the time you finish this book, this connection will become obvious.

Static vs dynamic model

Before we proceed further, let's take a quick look at how a dynamic model differs from a static model. Consider a SISO process in Figure 1.3 where a step change in input is induced and the corresponding change in output is observed.

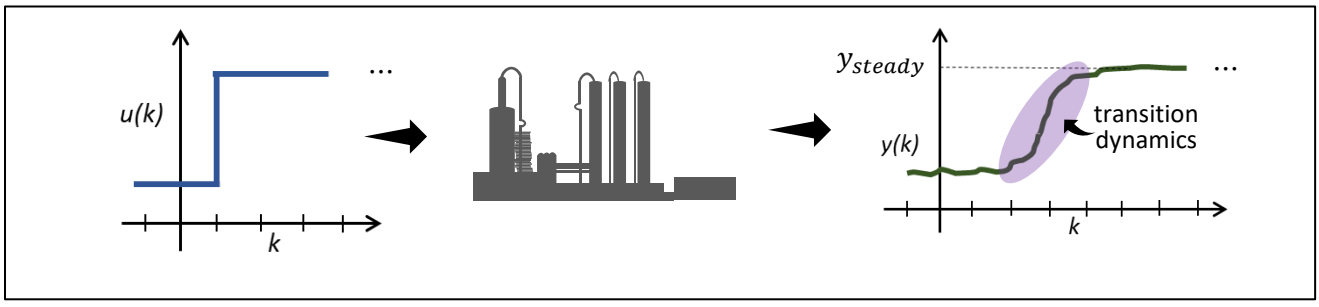


Figure 1.3: Representative dynamic changes in a SISO process output upon a step change in input

`

- In static model, we are only concerned with being able to predict y_{steady} as the impact of the step change in u.
- On the other hand, in dynamic model, we care about the transition dynamics as well.
- It is obvious that a dynamic model estimation is a more demanding problem and the complexity only increases for multivariable systems with stochastic components.

Validity of discrete-time models

It won't be wrong to say that continuous-time description of process systems (which are inherently continuous in nature) is more natural than discrete-time description. However, in computer controlled industrial plants, signals from the underlying continuous process are sampled and made available at discrete time-points. Therefore, there is sound rationale for focusing on discrete-time models.

When using first principles approach, discrete-time model can be obtained as an approximation as well as an exact description of the underlying process. For example, consider the following differential equation

$$\frac{dy}{dt} = f(y(t), u(t))$$

Approximating the derivative via forward difference gives,

$$\frac{y(t+T) - y(t)}{T} = f(y(t), u(t)); \quad T \; is \; sampling \; interval$$

$$\Rightarrow y(t+T) = y(t) + Tf\big(y(t), u(t)\big)$$

or

$$y(k+1) = y(k) \; + \; T \, f(y(k), u(k)); \quad k = \frac{t}{T}$$

Discrete-time model

In an alternate scenario, if the input variables are constant between samples, then an exact analytical discrete-time representation can be derived for linear processes.

`

1.2 ML-DPM Workflow

Figure 1.4 shows the typical steps involved in system identification. As you can see, SysID is more than just curve fitting. Overall, there are five broad tasks: data collection, exploratory data analysis, data pre-treatment, model identification, and model validation. Although we will study each step in detail in Chapter 4, let's take a quick overview:

- ***Data collection and exploratory data analysis****:* Availability of 'proper' data is absolutely critical and the requirement of 'richness' of training data is indispensable. In the previous section we saw that dynamic modeling is more demanding than static modeling and consequently, there are more stringent requirements on training data for dynamic modeling. Training data may be taken from historical database or fresh experiments may be performed. The training inputs should be such that the output data contains the dynamic variations of interest.

  Exploratory data analysis (EDA) involves preliminary (and usually manual) investigation of data to get a 'feel' of the system's characteristics. The activities may include generating some graphical plots to check the presence of trends, seasonality, and non-stationarity in data. Inferences made during EDA help make the right choices in the subsequent steps of SysID.

- ***Data pre-treatment***: This step consists of several activities that are designed to remove the portions of training data that are unimportant (or even detrimental) to model identification. It may entail removal of outliers and noise, removal of trends, etc. Alternatively, training data can also be massaged to manipulate the model's accuracy as suited for the end purpose of the model. For example, if the model is to be used for control purpose, then the training data may be pre-filtered to bolster model's accuracy for high frequency signals. Overall, the generic guideline is that you have better chances of a successful SysID with a better conditioned dataset.

- ***Model training***: Model training is the most critical step in SysID and entails a few sub-steps. First, a choice must be made on the type of disturbance model and deterministic model. The end use of the model, the available *a priori* system knowledge, and the desired degree of modeling complexity dictate these selections. For example, if the model is to be used for simulation purposes, then OE[1] structure may be preferred over ARX; if process disturbance has different dynamics compared to process input, then

[1] We will cover these model structures in the upcoming chapters

ARMAX will be preferable; if computational convenience is sought, then ARX is usually the first choice and is preferred over other complex structures such as recurrent neural networks. Model structure selection is followed by model parameter estimation. This sub-step is mostly 'hands-off' due to the availability of several specialized libraries that perform parameter estimation.

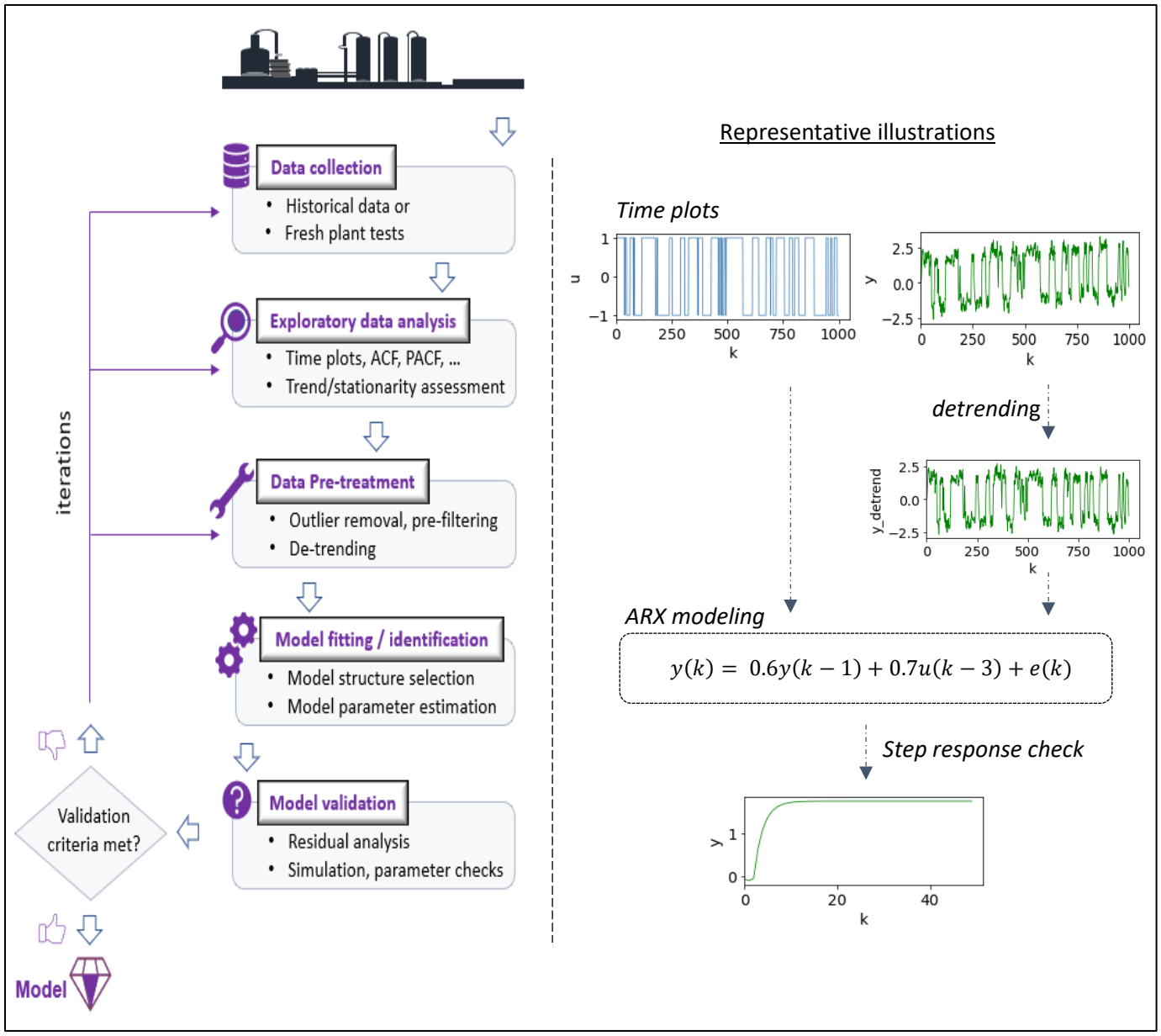


Figure 1.4: Steps (with sample sub-steps) involved in a typical ML-based dynamic modeling

`

- ***Model validation***: Once a model has been fitted, the next step is to check if the model truly represents the underlying process. Several techniques are at a modeler's disposal. For example, you could check the model's performance on a dataset that has not been used during parameter estimation, plot the modeling errors to look for leftover patterns, or check if the model agrees with the *a priori* information about the system. Often, your first model will fail the validation procedure. An expert modeler can use the validation results to infer the reasons for the failure. Some examples of the cause could be

    - Training data was not 'rich' enough
    - Training data was not pre-treated adequately
    - Choice of model structure was wrong
    - Estimation algorithm did not converge

    Once diagnosed, appropriate corrections are made, and the iterative procedure continues.

We hope that you get the understanding that SysID is more than just black-box application of modeling algorithm for estimation of model parameters. There are several practical aspects that require active participation of the modeler during the SysID process. Without exaggeration, we can say that SysID is an art, and the rest of the book will help you obtain the necessary skills to become the SysID artist!

1.3 Taxonomy of ML-based Dynamic Models

The field of system identification is overwhelmingly extensive owing to the decades of research that has led to the development of several specialized techniques. Figure 1.5 gives an overview of some of the popular methods and models that we will cover in this book. These models represent only a subset of all the SysID models out there and this selection is based on our experience regarding the relevance of these models for process systems modeling. We largely focus on time-invariant[2], discrete-time models and these shown models can help you handle a large majority of DPM problems you will encounter in process industry.

[2] Models where time variable does not appear as an explicit variable.

`

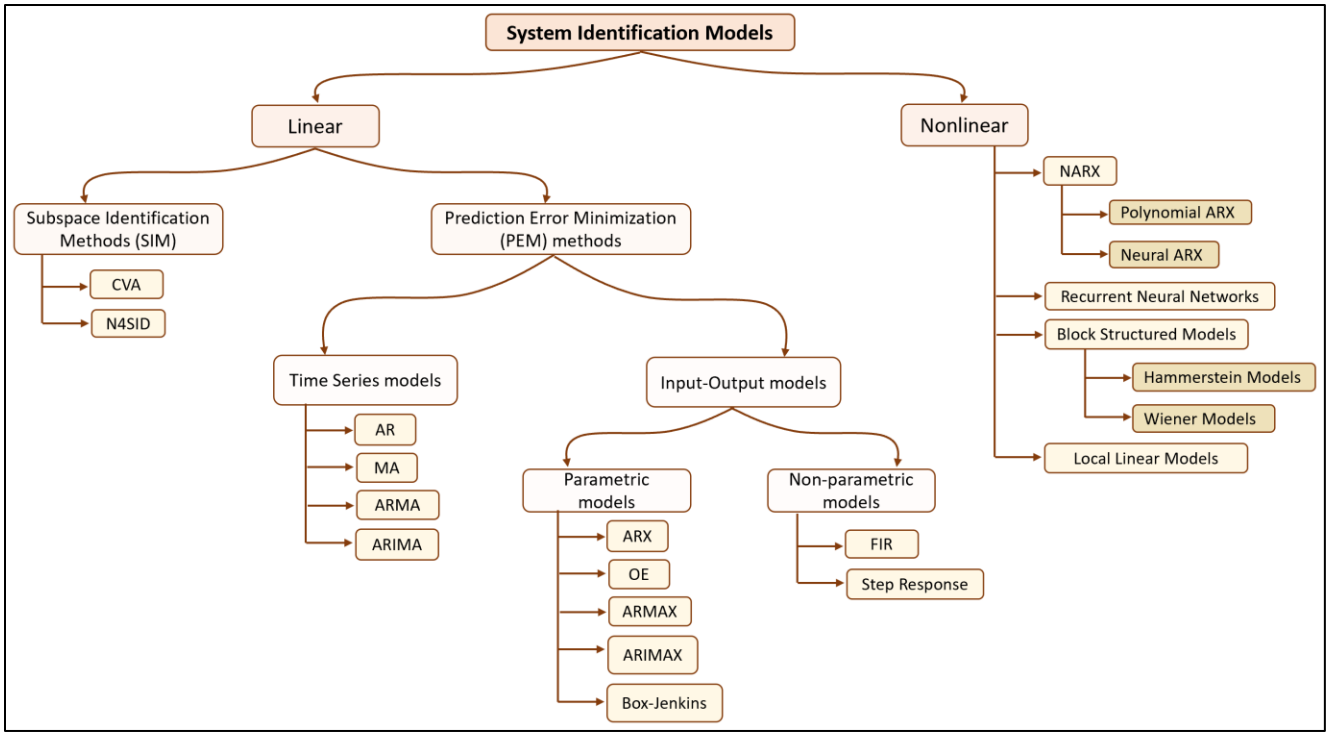


Figure 1.5: Some popular SysID methods and models covered in this book

There are several noteworthy points in Figure 1.5. First, the apparent domination of linear models: you may have expected nonlinear models to be preferable for modeling complex industrial process systems, however, as it turns out, linear models are often justified for DPM. The justification stems from the fact that industrial processes often operate around an optimal point making the linear models pretty good approximation for usage in process control and process monitoring applications. If the end use of the model is process simulation or design where the system response over wide range of input variables is of interest, then nonlinear models can prove to be more suitable. Within linear category, different modeling options exist to cater to process systems with different characteristics, viz, multivariable outputs, presence of correlated noise, disturbances sharing dynamics with inputs, presence of measurement noise only, presence of drifts or non-stationarities, etc.

The classification at the root of linear model sub-tree is based on the methodology used for model fitting. While PEM methods employ minimization of prediction errors, subspace methods are based on matrix algebra and do not involve any optimization for parameter optimization.[3] Do not worry if these terms do not make much sense right now; they will soon become 'obvious' to you. If not trained about the nuanced differences between these different models and methods, you may not have much idea at the outset about which model would be the best one for your system. This book will help you gain adequate conceptual understanding to become adept at making the right choice and obtaining your coveted model quickly.

[3] Another distinction is that PEM is used to obtain input-output models while SIM is used to obtain state-space models.

`

Types of models

In Eq. 1 we saw one way of representing a dynamic process model. The models from Figure 1.5 can be used to generate other representations of your dynamic systems; the figure below shows the different forms/types of models that we will learn to derive in this book using machine learning. We will also understand the pros and cons of these different model forms.

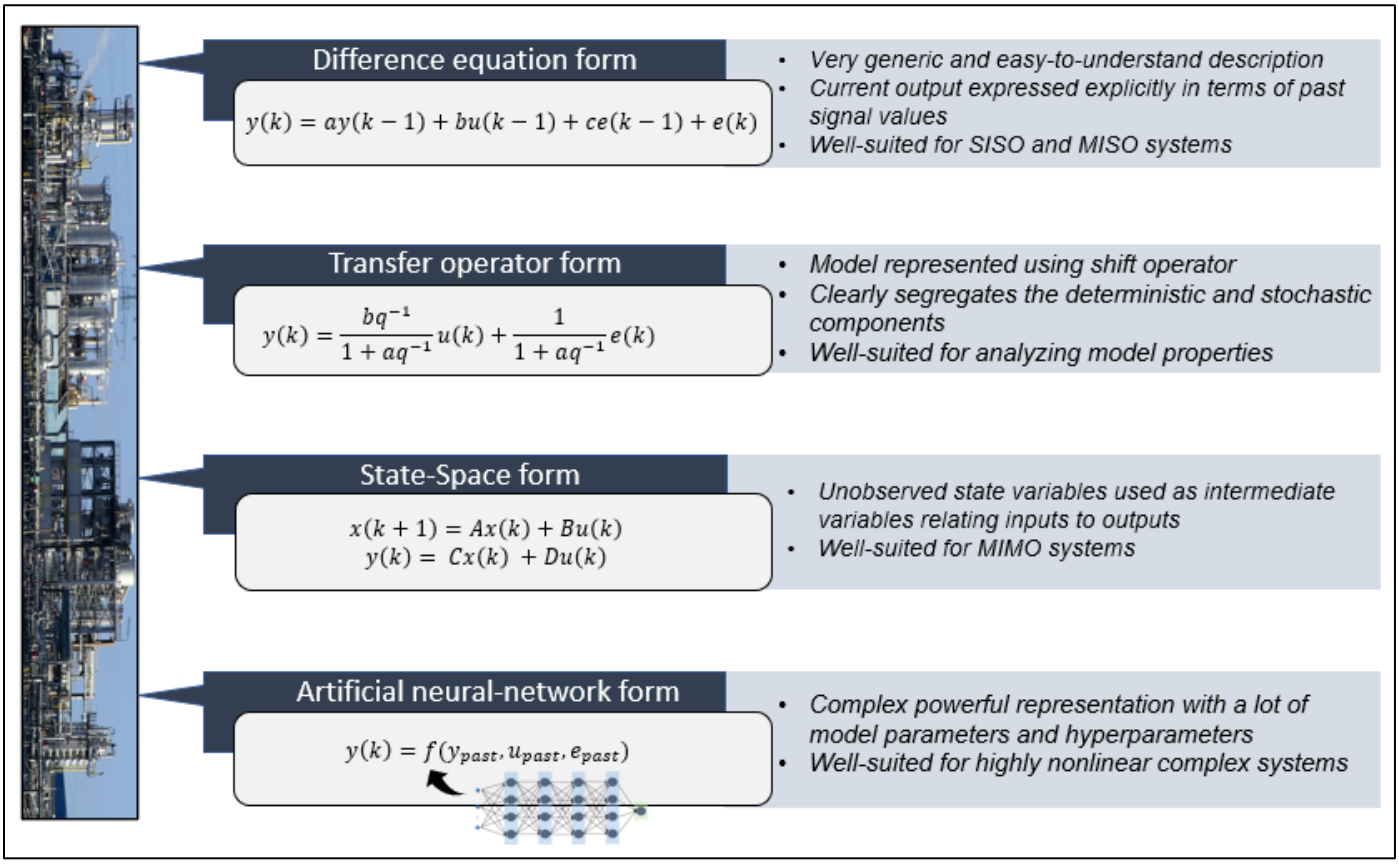
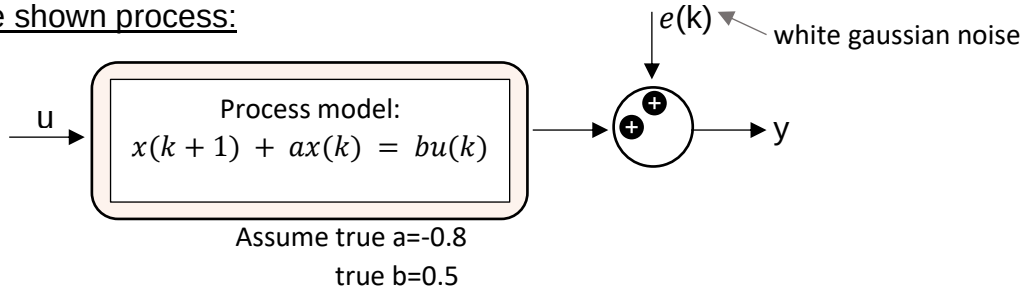


Figure 1.6: Different forms of dynamic process model (that we will learn to generate in this book), their defining characteristics, and corresponding representative examples

`

Why careful choice of model matters: A simple illustration

Consider the shown process:



If not careful, you may just ignore the presence of measurement noise and attempt to fit the following input-output model to estimate the model parameters *a* and *b*.

Fitted Model (ARX form): $y(k+1) + a\,y(k) = bu(k)$

Data file 'simpleProcess.csv' contains 1000 samples of *u* and *y* obtained from the true process. Below are the parameter estimates we get using this data,

$$\hat{a} = -0.6628\ (\pm\ 0.016)\ ,\ \hat{b} = 0.7174\ (\pm 0.035)$$

values in bracket denote standard errors

Two things are striking here: the estimates are grossly inaccurate, and the parameter error estimates seem to suggest high confidence in these wrong values! What went wrong in our approach? The reason is that the SNR (signal-to-noise ratio) value is not very high (data was generated with SNR ~ 10) and the fitted ARX model is wrong input-output form of the true process. The correct form would be the following

Correct input-output model

$$x(k+1) + ax(k) = bu(k)$$
$$y(k) = x(k) + e(k)$$
$$\Rightarrow \quad y(k+1) + a\,y(k) = bu(k) + e(k+1) + ae(k)$$

Equation error is not white (as assumed in ARX model) but colored!

Ignoring the presence of correlated equation error leads to biased (inaccurate) parameter estimates. Hopefully, this simple illustration has convinced you against blind application of 'convenient' models. Part II of this book will add several modeling tools to your ML-DPM arsenal to help avoid such modeling mistakes.

`

1.4 Applications of DPM in Process Industry

In Figure 1.1, we saw some of the applications of dynamic models in process industry. To provide further perspectives into how a typical plant operator or management may use these varied applications, Figure 1.7 juxtaposes DPM applications alongside the typical decision-making hierarchy in a process plant. Every step of plant operation is now-a-days heavily reliant on DPM-based tools and machine learning has proven to be a useful vehicle for quickly building these tools. You can use the models from Figure 1.5 for building these tools or if you use commercial vendor solutions, you can find them employing these models in their products. For example, in the process control field, FIR models have been the bedrock of industrial MPC controllers. In the last few years, commercial vendors have inducted CVA models in their offering due to the advantages provided by subspace models. The latest offering by Aspen, DMC3[4], incorporates neural networks for MPC and inferential modeling. ARX, BJ models are also used for industrial MPC[5].

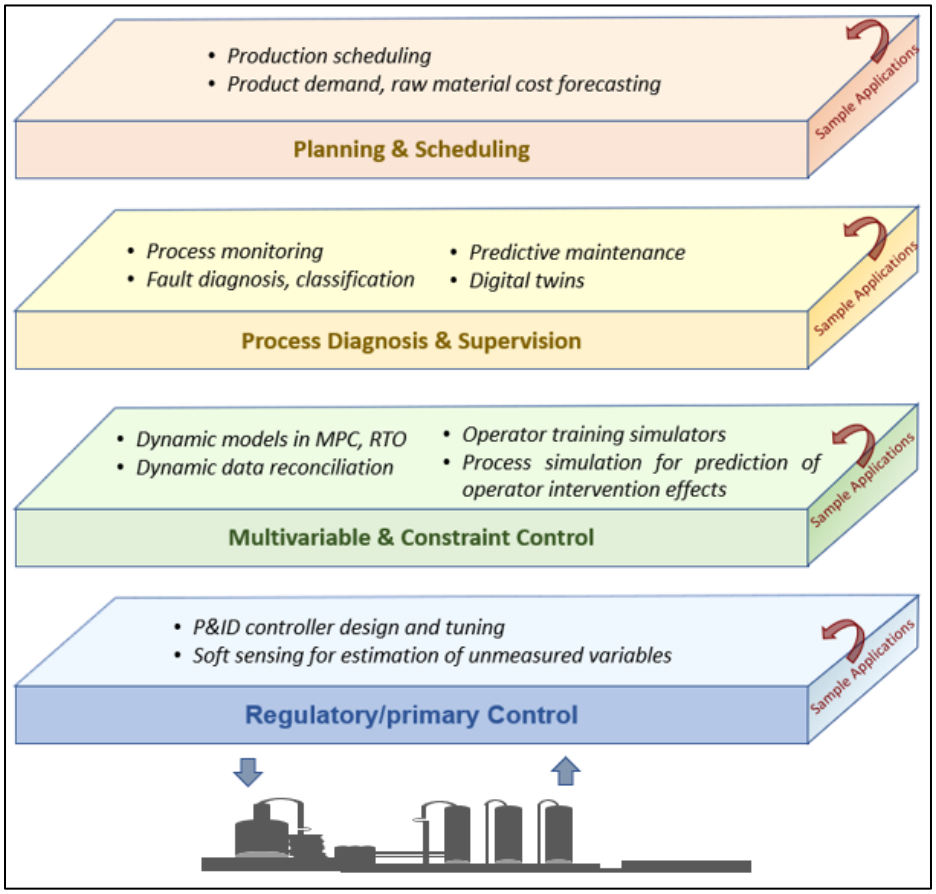


Figure 1.7: DPM applications in different layers of operational hierarchy

[4] https://www.aspentech.com/en/products/msc/aspen-dmc3

[5] Qin and Badgwell, A survey of industrial model predictive control technology. *Control Engineering Practice*, 2003

`

- The regulatory control layer primarily comprises of (PID) control valves. Potential DPM applications may involve usage of soft-sensor for estimation of difficult-to-measure controlled variables.

- The multivariable control layer usually consists of MPC and RTO modules. Here, dynamic models representing multivariable relationships between plant variables are used to ensure optimal operations of the plant. Another interesting application is development of operator training simulators (OTSs) for training plant operators.

- The process diagnostics layer ensures timely fault detection and diagnostics. Here, process measurements can be compared against predictions from dynamic models to check for presence of process abnormalities.

- The production scheduling layer has dynamic models to determine short-term optimal production schedules using forecasts of product demand and/or raw-material cost.

This concludes our quick attempt to establish the connection between process industry, dynamic process modeling, and machine learning. It must now be obvious to you that modern process industry relies heavily on dynamic process modeling to achieve its objectives of reducing maintenance costs, and increasing productivity, reliability, safety, and product quality. ML-based DPM helps build tools quickly to facilitate achieving these objectives.

This introductory chapter has also cautioned against blind application of ‘convenient’ ML models for dynamic modeling. Your process data will throw several questions at you at each stage of system identification. No straightforward answers exist to these questions and only some time-tested guiding principles are available. While the rest of the book will familiarize you with these principles, the onus still lies on you to use your process insights and SysID

`

understanding to make the right modeling choices. And yes, remember the age-old advice[6], “All models are wrong, but some are useful.”

Summary

This chapter impressed upon you the importance of DPM in process industry and the role machine learning plays in it. We familiarized ourselves with the typical SysID workflow, explored its different tasks, and looked at different ML models available at our disposal for model identification. We also explored the application areas in process industry where ML has proved useful. In the next chapter we will take the first step and learn about the environment we will use to execute our Python scripts containing SysID code.

[6] Attributed to the famous statistician George E. P. Box. It basically implies that your (SysID) model will seldom exactly represent the real process. However, it can be close enough to be useful for practical purposes.

`

Chapter 2

The Scripting Environment

In the previous chapter we studied the various aspects of system identification and learned about its different uses in process industry. In this chapter we will quickly familiarize ourselves with the Python language and the scripting environment that we will use to write ML codes, execute them, and see results. This chapter won't make you an expert in Python but will give you enough understanding of the language to get you started and help understand the several in-chapter code implementations in the upcoming chapters. If you already know the basics of Python, have a preferred code editor, and know the general structure of a typical SysID script, then you can skip to Chapter 3.

If you skim through the system identification literature, you will find almost exclusive usage of MATLAB software as the computing environment. This is mostly attributed to the System Identification toolbox, a very powerful MATLAB toolbox developed by Prof. Lennart Jung (a legend in system identification). Unfortunately, this tool not freely available, and MATLAB is not yet as popular as Python among the ML community. Luckily, several good souls in the Python community have developed specialized libraries for all aspects of SysID. Most of the popular SysID models can now be generated using off-the-shelf Python libraries. Considering the dominance of Python for deep learning, Python becomes an excellent choice for SysID scripting.

In the above context, we will cover the following topics to familiarize you to Python

- Introduction to Python language
- Introduction to Spyder and Jupyter, two popular code editors
- Overview of Python data structures and scientific computing libraries
- Python libraries for system identification
- Overview of a typical ML-DPM/SysID script

`

2.1 Introduction to Python

Python is a high-level general-purpose computer programming language that can be used for application development and scientific computing. If you have used other computer languages like Visual Basic, C#, C++, Java, then you would understand the fact that Python is an interpreted and dynamic language. If not, then think of Python as just another name in the list of computer languages. What is more important is that Python offers several features that sets it apart from the rest of the pack making it the most preferred language for machine learning. Figure 2.1 lists some of these features. Python provides all tools to conveniently carry out all steps of an ML-DPM project, namely, data collection, data exploration, data pre-processing, model ID, visualization, and solution deployment to end-users. In addition, freely available tools make writing Python code very easy[7].

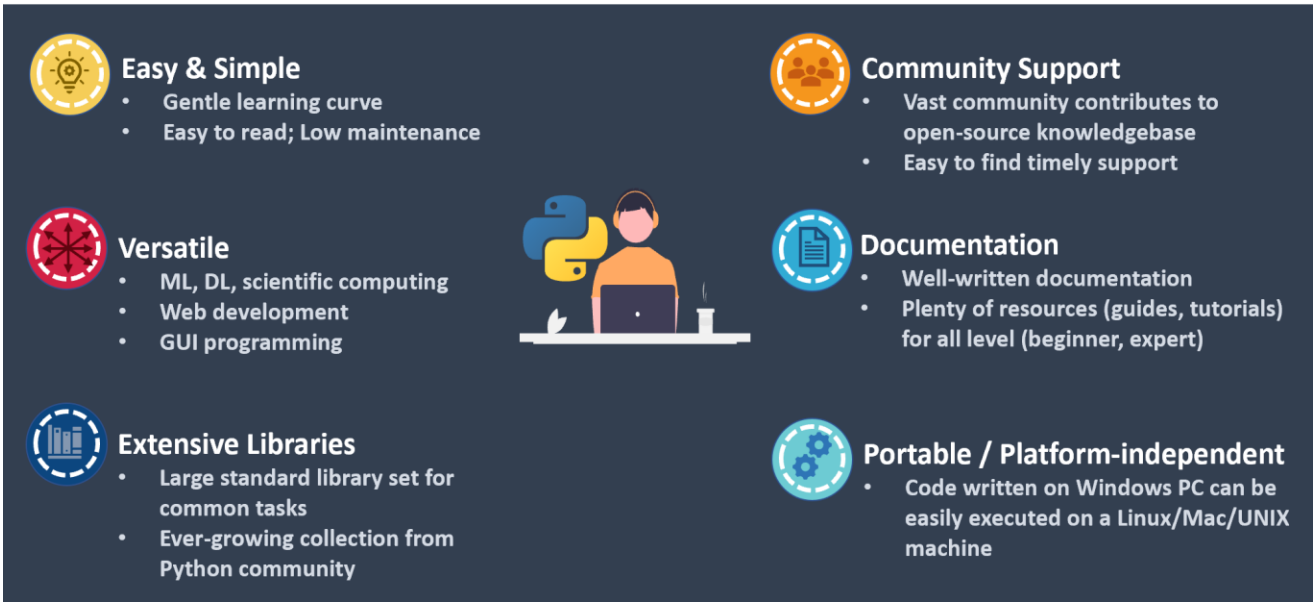


Figure 2.1: Features contributing to Python language's popularity

Installing Python

One can download official and latest version of Python from the *python.com* website. However, the most convenient way to install and use Python is to install Anaconda (*www.anaconda.com*) which is an open-source distribution of Python. Along with the core Python, Anaconda installs a lot of other useful packages. Anaconda comes with a GUI called Anaconda Navigator (Figure 2.2) from where you can launch several other tools.

[7] Most of the content of this chapter is similar to that in Chapter 2 of the book 'Machine Learning in Python for Process Systems Engineering' and have been re-produced with appropriate changes to maintain the standalone nature of this book.

`

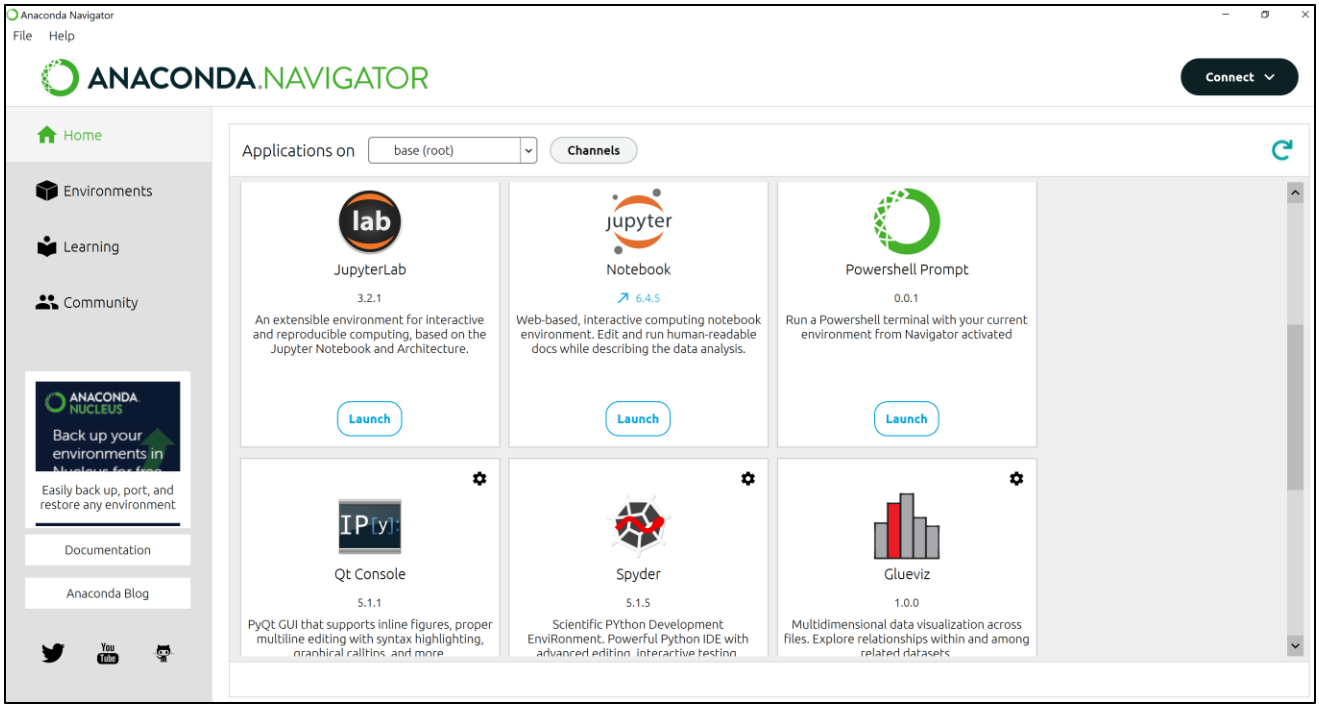


Figure 2.2: Anaconda Navigator GUI

Running/Executing Python code

Once Anaconda is installed, you can write your code in any text editor (like Notepad), save it in *.py* format, and then run via Anaconda command terminal. However, a convenient alternative is to use IDEs (integrated development environments). IDEs not just allow creating *.py* scripts, but also execute them on the fly, i.e., we can make code edits and see the results immediately, all within an integrated environment. Among the several available Python IDEs (PyCharm, VS Code, Spyder, etc.), we found Spyder to be the most convenient and functionality-rich, and therefore is discussed in the next section.

Jupyter Notebooks are another very popular way of writing and executing Python code. These notebooks allow combining code, execution results, explanatory text, and multimedia resources in a single document. As you can imagine, this makes saving and sharing complete data analysis very easy.

In the next section, we will provide you with enough familiarity on Spyder and Jupyter so that you can start using them.

`

2.2 Introduction to Spyder and Jupyter

Figure 2.3 shows the interface[8] (and its different components) that comes up when you launch Spyder. These are the 3 main components:

- *Editor*: You can type and save your code here. Clicking ▶ button executes the code in the active editor tab.
- *Console*: Script execution results are shown here. It can also be used for executing Python commands and interact with variables in the workspace.
- *Variable explorer*: All the variables generated by running editor scripts or console are shown here and can be interactively browsed.

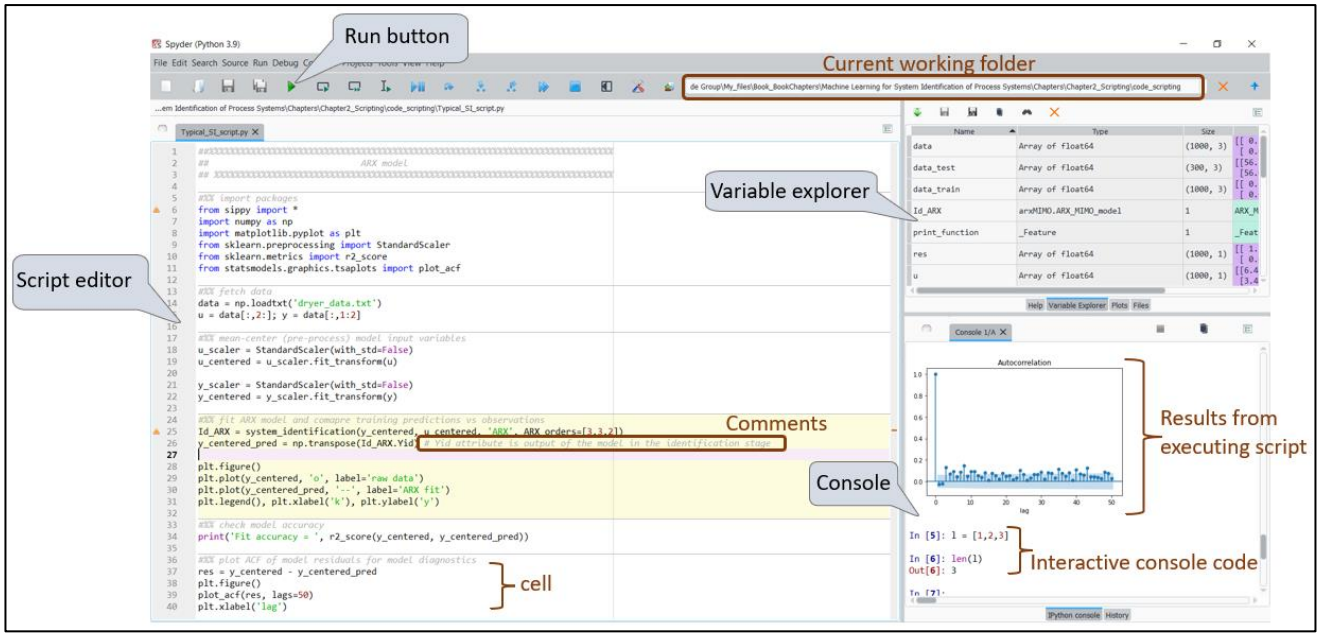


Figure 2.3: Spyder interface

Like any IDE, Spyder offers several convenience features. You can divide your script into cells and execute only selected cell if you choose to (by pressing Ctrl + Enter buttons). Intellisense allows you to autocomplete your code by pressing Tab key. Extensive debugging functionalities make troubleshooting easier. These are only some of the features available in Spyder. You are encouraged to explore the different options (such as pausing and canceling script execution, clearing out variable workspace, etc.) on the Spyder GUI.

[8] If you have used MATLAB, you will find the interface very familiar

`

With Spyder, you have to run your script again to see execution results if you close and reopen your script. In contrast to this, consider the Jupyter interface in Figure 2.4. Note that the Jupyter interface opens in a browser. We can save the shown code, the execution outputs, and explanatory text/figures as a (.ipnb) file and have them remain intact when we reopen the file in Jupyter notebook.

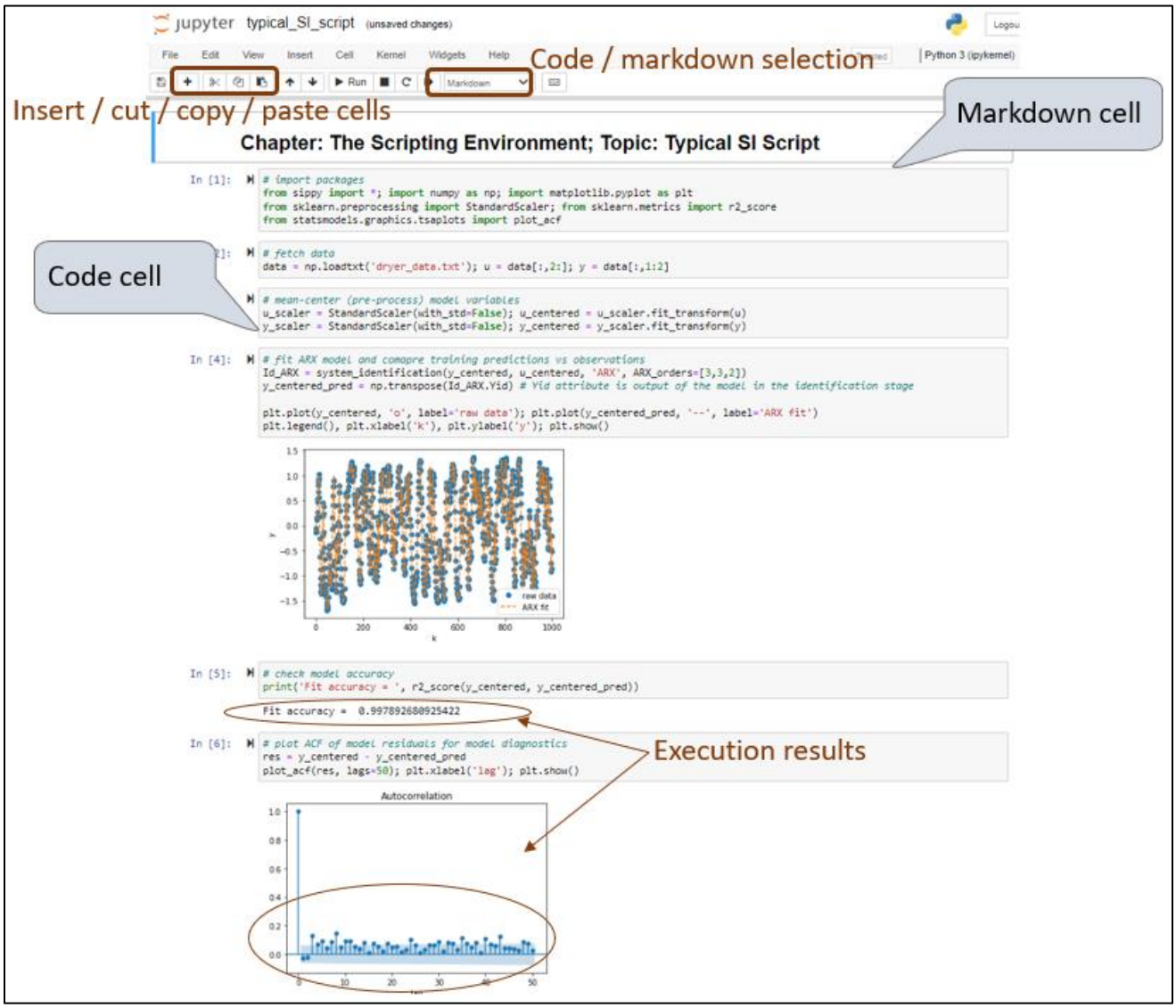


Figure 2.4: Jupyter interface

You can designate any input cell as a code or markdown (formatted explanatory text). You can press Ctrl + Enter keys to execute any active cell. All the input cells can be executed via the *Cell* menu.

This completes our quick overview of Spyder and Jupyter interfaces. You can choose either of them for working through the codes in the rest of the book.

`

2.3 Python Language: Basics

In the current and next sections, we will see several simple examples of manipulating data using Python and scientific packages. While these simple operations may seem unremarkable (and boring) in the absence of any larger context, they form the building blocks of more complex scripts presented later in the book. Therefore, it will be worthwhile to give these atleast a quick glance.

Note that you will find '#' used a lot in these examples; these hash marks are used to insert explanatory comments in code. Python ignores (does not execute) anything written after # on a line.

Basic data types

In Python, you work with 4 types of data types

```
# 4 data types: int, float, str, bool
i = 2 # integer; type(i) = int
f = 1.2 # floating-point number; type(f) = float
s = 'two' # string; type(s) = str
b = True # boolean; type(b) = bool
```

With the variables defined you can perform basic operations

```
# print function prints/displays the specified message
print(i+2) # displays 4
print(f*2) # displays 2.4
```

Lists, tuples as ordered sequences

Related data (not necessarily of the same data type) can be arranged together as a sequence in a list as shown below

```
# different ways of creating lists
list1 = [2,4,6]
list2 = ['air',3,1,5]
list3 = list(range(4)) # equals [0,1,2,3]; range function returns a sequence of numbers starting
                       from 0 (default) with increments of 1 (default)
list3.append(8) # returns [0,1,2,3,8];  append function adds new items to existing list
list4 = list1 + list2 # equals [2,4,6,'air',3,1,5]
list5 = [list2, list3] # nested list [['air', 3, 1, 5], [0, 1, 2, 3,8]]
```

`

Tuples are another sequence construct like lists, with a difference that their items and sizes cannot be changed Since tuples are immutable/unchangeable, they are more memory efficient.

```
# creating tuples
tuple1 = (0,1,'two')
tuple2 = (list1, list2) # equals ([2, 4, 6, 8], ['air', 3, 1, 5])
```

A couple of examples below illustrate list comprehension which is a very useful way of creating new lists from other sequences

```
# generate powers of individual items in list3
newList1 = [item**2 for item in list3] # equals [0,1,4,9,64]

# nested list comprehension
newList2 = [item2**2 for item2 in [item**2 for item in list3]] # equals [0,1,16,81,4096]
```

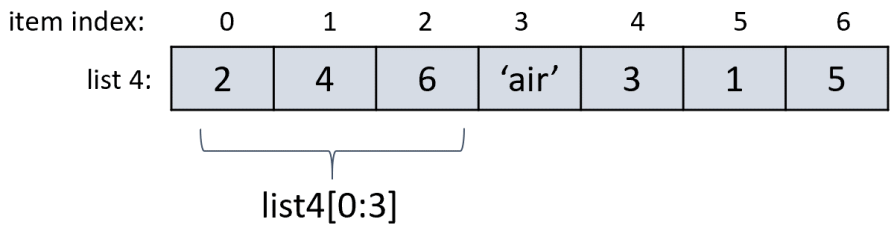
Indexing and slicing sequences

Individual elements in a list can be accessed and modified as follows

```
# working with single item using positive or negative indexes
print(list1[0]) # displays 2, the 1st item in list1
list2[1] = 1 # list2 becomes ['air',1,1,5]
print(list2[-2]) # displays 1, the 2nd last element in list2
```

Note that Python indexing starts from zero. Very often, we need to work with multiple items of the list. This can be accomplished easily as shown below.

```
# accessing multiple items through slicing
# Syntax: givenList[start,stop,step]; if unspecified, start=0, stop=list length, step=1
print(list4[0:3]) # displays [2,4,6], the 1st, 2nd, 3rd items; note that index 3 item is excluded
```




```
print(list4[:3]) # same as above
```

`

```
print(list4[4:len(list4)]) # displays [3,1,5]; len() function returns the number of items in list
print(list4[4:]) # same as above
print(list4[::3]) # displays [2, 'air', 5]
print(list4[::-1]) # displays list 4 backwards [5, 1, 3, 'air', 6, 4, 2]
list4[2:4] = [0,0,0] # list 4 becomes [2, 4, 0, 0, 0, 3, 1, 5]
```

Execution control statements

These statements allow you to control the execution sequence of code. You can choose to execute any specific parts of the script selectively or multiple times. Let's see how you can accomplish these

Conditional execution

```
# selectively execute code based on condition
if list1[0] > 0:
    list1[0] = 'positive'
else:
    list1[0] = 'negative'

# list1 becomes ['positive', 4, 6]
```

Loop execution

```
# compute sum of squares of numbers in list3
sum_of_squares = 0
for i in range(len(list3)):
    sum_of_squares += list3[i]**2

print(sum_of_squares) # displays 78
```

Custom functions

Previously we used Python's built-in functions (len(), append()) to carry out operations pre-defined for these functions. Python allows defining our own custom functions as well. The advantage of custom functions is that we can define a set of instructions once and then re-use them multiple times in our script and project.

For illustration, let's define a function to compute the sum of squares of items in a list

```
# define function instructions
def sumSquares(givenList):
    sum_of_squares = 0
    for i in range(len(givenList)):
        sum_of_squares += givenList[i]**2

    return sum_of_squares

# call/re-use the custom function multiple times
print(sumSquares(list3)) # displays 78
```

`

You might have noticed in our custom function code above that we used different indentations (number of whitespaces at beginning of code lines) to separate the 'for loop' code from the rest of the function code. This practice is actually enforced by Python and will result in errors or bugs if not followed. While other popular languages like C++, C# use braces ({}) to demarcate a code block (body of a function, loop, if statement, etc.), Python uses indentation. You can choose the amount of indentation but it must be consistent within a code block.

This concludes our extremely selective coverage of Python basics. However, this should be sufficient to enable you to understand the codes in the subsequent chapters. Let's continue now to learn about specialized scientific packages.

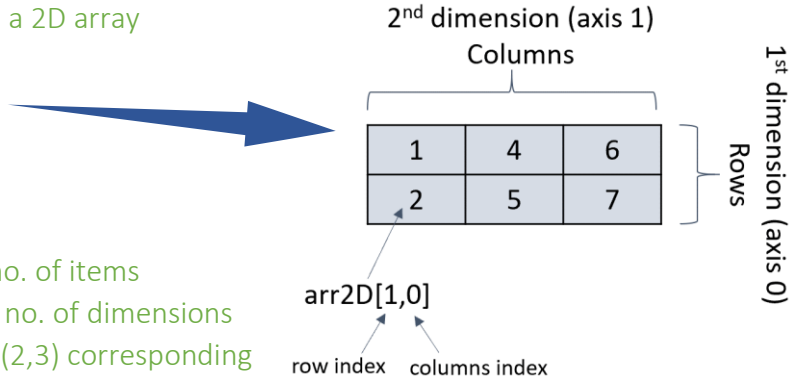
2.4 Scientific Computing Packages: Basics

While the core Python data-structures are quite handy, they are not very convenient for the advanced data manipulations we require for machine learning tasks. Fortunately, specialized packages like NumPy, SciPy, Pandas exist which provide convenient multidimensional tabular data structures suited for scientific computing. Let's quickly make ourselves familiar with these packages.

NumPy

In NumPy, ndarrays are the basic data structures which put data in a grid of values. Illustrations below show how 1D and 2D arrays can be created and their items accessed

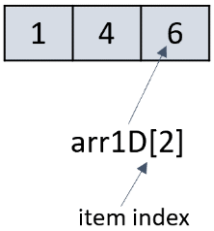
```
# import numpy package & create a 2D array
import numpy as np
arr2D = np.array([[1,4,6],[2,5,7]])
```




```
# getting information about arr2D
print(arr2D.size) # returns 6, the no. of items
print(arr2D.ndim) # returns 2, the no. of dimensions
print(arr2D.shape) # returns tuple(2,3) corresponding
                   to 2 rows & 3 columns
```

`

```
# create a 1D array
arr1D = np.array([1,4,6])
```




```
# getting information about arr1D
print(arr1D.size) # returns 3, the no. of items
print(arr1D.ndim) # returns 1, the no. of dimensions
print(arr1D.shape) # returns tuple (3,) corresponding
                   to 3 items
```

Note that the concept of rows and columns do not apply to a 1D array. Also, you would have noticed that we imported the NumPy package before using it in our script ('np' is just a short alias). Importing a package makes available all its functions and sub-packages for use in our script.

Creating NumPy arrays

Previously, we saw how to convert a list to a NumPy array. There are other ways to create NumPy arrays as well. Some examples are shown below

```
# creating sequence of numbers
arr1 = np.arange(3, 6) # same as Python range function; results in array([3,4,5])
arr2 = np.arange(3, 9, 2) # the 3rd argument defines the step size; results in array([3,5,7])
arr3 = np.linspace(1,7,3) # creates evenly spaced 3 values from 1 to 7; results in
                          array([1., 4., 7.])

# creating special arrays
arr4 = np.ones((2,1)) # array of shape (2,1) with all items as 1
arr5 = np.zeros((2,2)) # all items as zero; often used as placeholder array at beginning of script
arr6 = np.eye(2) # diagonal items as 1

# adding axis to existing arrays (e.g., converting 1D array to 2D array)
print(arr1[:, np.newaxis])
>>>[[3]
    [4]
    [5]]

arr7 = arr1[:, None] # same as above
```

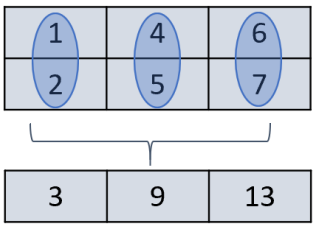
`

```
# combining / stacking arrays
print(np.hstack((arr1, arr2))) # horizontally stacks passed arrays
>>> [3 4 5 3 5 7]

print(np.vstack((arr1, arr2))) # vertically stacks passed arrays
>>> [[3 4 5]
     [3 5 7]]

print(np.hstack((arr5,arr4))) # array 4 added as a column into arr5
>>> [[0. 0. 1.]
     [0. 0. 1.]]

print(np.vstack((arr5,arr6))) # rows of array 6 added onto arr5
>>> [[0. 0.]
     [0. 0.]
     [1. 0.]
     [0. 1.]]
```

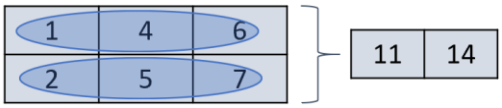
Basic Numpy functions

NumPy provides several useful functions like mean, sum, sort, etc., to manipulate and analyze NumPy arrays. You can specify the dimension (axis) along which data needs to be analyzed. Consider the sum function for example

```
# along-the row sum
arr2D.sum(axis=0) # returns 1D array
                  with 3 items
```




```
# along the column sum
arr2D.sum(axis=1) # returns 1D array
                  with 2 items
```



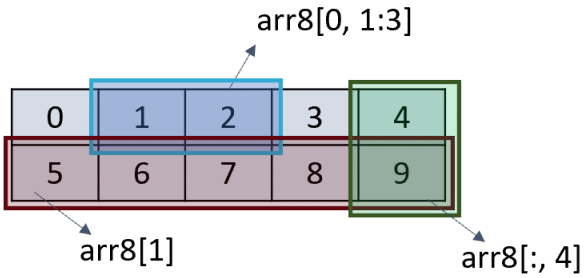
Executing arr2D.sum() returns the scalar sum over the whole array, i.e., 25.

Indexing and slicing arrays

Accessing individual items and slicing NumPy arrays work like that for Python lists

`

```
# accessing individual items
print(arr2D[1,2]) # returns 7

# slicing
arr8 = np.arange(10).reshape((2,5)) # rearrange the 1D array into shape (2,5)
print((arr8[0:1,1:3]))
>>> [[1 2]]

print((arr8[0,1:3])) # note that a 1D array is returned here instead of the 2D array above
>>> [1 2]

# accessing entire row or column
print(arr8[1]) # returns 2nd row as array([5,6,7,8,9]); same as arr8[1,:]
print(arr8[:, 4]) # returns items of 5th column as a 1D array
>>> [4 9]
```



An important thing to note about NumPy array slices is that any change made on sliced view modifies the original array as well! See the following example

```
# extract a subarray from arr8 and modify it
arr8_sub = arr8[:, :2] # columns 0 and 1 from all rows
arr8_sub[1,1] = 1000
print(arr8) # arr8 gets modified as well!!
>>> [[   0    1    2    3    4]
     [   5 1000    7    8    9]]
```

This feature becomes quite handy when we need to work on only a small part of a large array/dataset. We can simply work on a leaner view instead of carrying around the large dataset. However, situation may arise where we need to actually work on a separate copy of subarray without worrying about modifying the original array. This can be accomplished via the copy method.

`

```
# use copy method for a separate copy
arr8 = np.arange(10).reshape((2,5))
arr8_sub2 = arr8[:, :2].copy()
arr8_sub2[1, 1] = 100 # arr8 won't be affected here
```

Fancy indexing is another way of obtaining a copy instead of a view of the array being indexed. Fancy indexing simply entails using integer or boolean array/list to access array items. Examples below clarify this concept

```
# combination of simple and fancy indexing
arr8_sub3 = arr8[:, [0, 1]] # note how columns are indexed via a list
arr8_sub3[1, 1] = 100 # arr8_sub3 becomes same as arr8_sub2 but arr8 is not modified here

# use boolean mask to select subarray
arr8_sub4 = arr8[arr8 > 5] # returns array([6,7,8,9]), i.e., all values > 5
arr8_sub4[0] = 0 # again, arr8 is not affected
```

Vectorized operations

Suppose you need to perform element-wise summation of two 1D arrays. One approach is to access items at each index at a time in a loop and sum them. Another approach is to sum up items at multiple indexes at once. The later approach is called vectorized operation and can lead to significant boost in computational time for large datasets and complex operations.

```
# vectorized operations
vec1 = np.array([1,2,3,4])
vec2 = np.array([5,6,7,8])
vec_sum = vec1 + vec2 # returns array([6,8,10,12]); no need to loop through index 0 to 3

# slightly more complex operation (computing distance between vectors)
vec_distance = np.sqrt(np.sum((vec1 - vec2)**2)) # vec_distance = 8.0
```

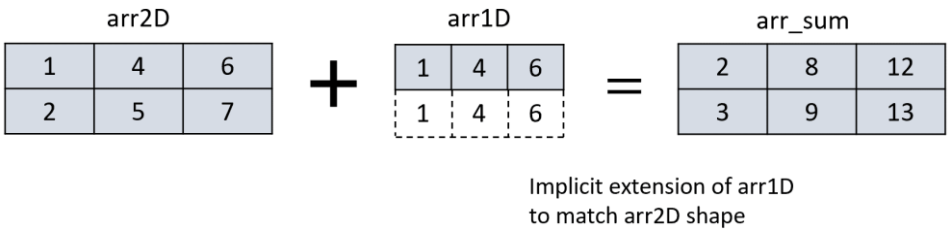
Broadcasting

Consider the following summation of arr2D and arr1D arrays

```
# item-wise addition of arr2D and arr1D
arr_sum = arr2D + arr1D
```

`

NumPy allows item-wise operation between arrays of different dimensions, thanks to 'broadcasting' where smaller array is implicitly extended to be compatible with the larger array, as shown in the illustration below. As you can imagine, this is a very convenient feature and for more details on broadcasting rules along with different example scenarios, you are encouraged to see the official documentation[9].



Pandas

Pandas is another very powerful scientific package. It is built on top of NumPy and offers several data structures and functionalities which make (tabular) data analysis and pre-processing very convenient. Some noteworthy features include label-based slicing/indexing, (SQL-like) data grouping/aggregation, data merging/joining, and time-series functionalities. Series and dataframe are the 1D and 2D array like structures, respectively, provided by Pandas

```
# Series (1D structure)
import pandas as pd

data = [10,8,6]
s = pd.Series(data)
print(s)
>>>
0   10
1    8
2    6
```

default row index

```
# Dataframe (2D structure)
data = [[1,10],[1,8],[1,6]]
df = pd.DataFrame(data, columns=['id', 'value'])
print(df)
>>>
   id  value
0   1     10
1   1      8
2   1      6
```

labeled columns (becomes 0 and 1 if no labels given)

```
# dataframe from series
s2 = pd.Series([1,1,1])
df = pd.DataFrame({'id':s2, 'value':s}) # same as above
```

Note that s.values and df.values convert the series and dataframe into corresponding NumPy arrays.

[9] numpy.org/doc/stable/user/basics.broadcasting.html

`

Data access

Pandas allows accessing rows and columns of a dataframe using labels as well as integer locations. You will find this feature pretty convenient.

```
# column(s) selection
print(df['id']) # returns column 'id' as a series
print(df.id) # same as above
print(df[['id']]) # returns specified columns in the list as a dataframe
>>>    id
     0  1
     1  1
     2  1

# row selection
df.index = [100, 101, 102] # changing row indices from [0,1,2] to [100,101,102] for illustration
print(df)
>>>    id  value
  100  1    10
  101  1     8
  102  1     6

print(df.loc[101]) # returns 2nd row as a series; can provide a list for multiple rows selection
print(df.iloc[1]) # integer location-based selection; same result as above

# individual item selection
print(df.loc[101, 'value']) # returns 8
print(df.iloc[1, 1]) # same as above
```

Data aggregation

As alluded to earlier, Pandas facilitates quick analysis of data. Check out one quick example below for group-based mean aggregation

```
# create another dataframe using df
df2 = df.copy()
df2.id = 2 # make all items in column 'id' as 2
df2.value *= 4 # multiply all items in column 'value' by 4
print(df2)
>>>    id  value
  100  2    40
```

`

```
  101   2    32
  102   2    24

# combine df and df2
df3 = df.append(df2)  # a new object is retuned unlike Python's append function
print(df3)
>>>     id  value
  100   1    10
  101   1     8
  102   1     6
  100   2    40
  101   2    32
  102   2    24

# id-based mean values computation
print(df3.groupby('id').mean()) # returns a dataframe
>>>    value
   id
  1     8.0
  2    32.0
```

File I/O

Conveniently reading data from external sources and files is one of the strong forte of Pandas. Below are a couple of illustrative examples.

```
# reading from excel and csv files
dataset1 = pd.read_excel('filename.xlsx') # several parameter options are available to customize
                                          what data is read
dataset2 = pd.read_csv('filename.xlsx')
```

This completes our very brief look at Python, NumPy, and Pandas. If you are new to Python (or coding), this may have been overwhelming. Don't worry. Now that you are atleast aware of the different data structures and ways of accessing data, you will become more and more comfortable with Python scripting as you work through the in-chapter code examples.

`

2.5 SysID-relevant Python Libraries

In the previous chapter we saw several dynamic modeling options for SysID. We also saw some of the pre-processing steps involved in data preparation and in the subsequent residual assessment step. Technically, one can employ the previously described scientific packages to implement the SysID algorithms, but it can be cumbersome and inconvenient. Fortunately, the Python user-community has made available several libraries that make implementation of the SysID algorithms very convenient. Figure 2.5 shows some of these libraries.

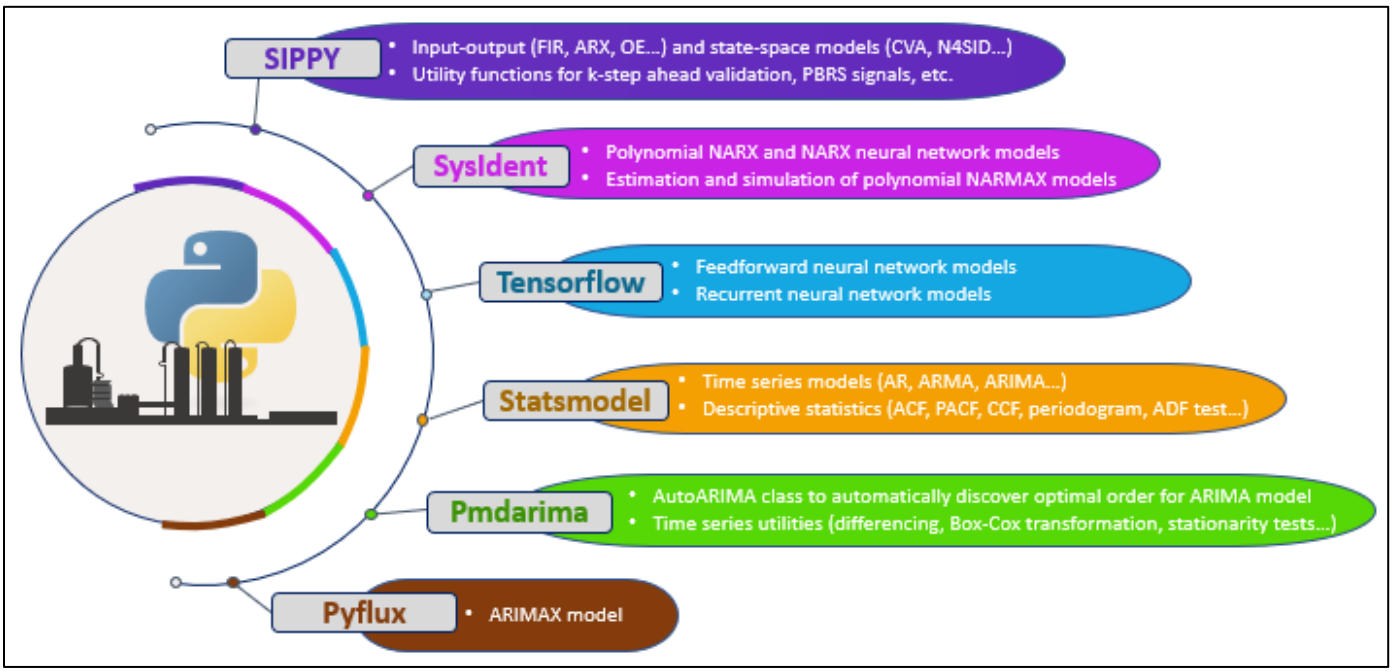


Figure 2.5: SysID-relevant Python packages[10] and the corresponding available functionalities

You will find us using these packages heavily in the upcoming chapters. We understand that many of the terms in the above figure may be alien to you right now, but we figured that it would be good to give you a 'feel' of what Python has to offer for SysID.

Note that we did not include the celebrated Sklearn library in Figure 2.5 that offers several generic ML-related functionalities such as dataset splitting, standardization, scoring, etc. Our SysID scripts will make extensive use of Sklearn library as well.

[10] SIPPY: https://github.com/CPCLAB-UNIPI/SIPPY
SysIdent: https://sysidentpy.org/; https://github.com/wilsonrljr/sysidentpy
Pyflux: https://pyflux.readthedocs.io/en/latest/

`

The concluding message is that Python provides most of the tools for SysID that you may find in other languages such as R and MATLAB.

2.6 Typical SysID Script

In Figure 1.4 in Chapter 1, we graphically portrayed different stages of a SysID exercise. Let's now see what a typical SysID script looks like. You will also understand how Python, NumPy, and advanced packages are utilized for ML-DPM scripting. We will study several SysID aspects in much greater detail in the next few chapters, but this simple script is a good start. The objective of this simple script is to take data for an input and an output variable from a file and build an ARX model between them. The first few code lines take care of importing the libraries that the script will employ.

```
# import packages
from sippy import system_identification
import numpy as np
import matplotlib.pyplot as plt
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import r2_score
from statsmodels.graphics.tsaplots import plot_acf
```

SIPPY provides ARX models

Used for model diagnostics

We studied NumPy before. SIPPY, Sklearn, and Statsmodels were introduced in previous section. Another library[11] that we see here is matplotlib[12] which is used for creating visualization plots. The next few lines of code fetch raw data and mean-center them.

```
# fetch data
data = np.loadtxt('InputOutputData.txt')
u = data[:,2:]; y = data[:,1:2] # first column is timestamp

# mean-center (pre-process) model variables
u_scaler = StandardScaler(with_std=False)
u_centered = u_scaler.fit_transform(u)
y_scaler = StandardScaler(with_std=False)
y_centered = y_scaler.fit_transform(y)
```

[11] If any package/library that you need is not installed on your machine, you can get it by running the command *pip install <package-name>* on Spyder console

[12] Seaborn is another popular library for creating nice-looking plots

`

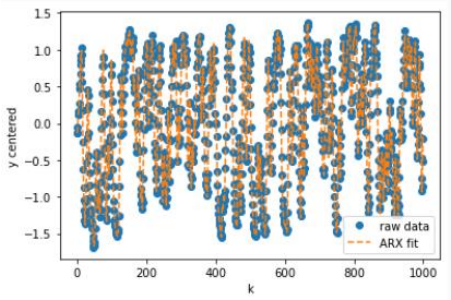
Here NumPy's loadtxt function is used to read space-separated data in the file InputOutputData.txt. The data get stored in a 2D NumPy array, *data*, where the 2nd and 3rd columns contain data for the output and input variables, respectively. NumPy slicing is used to separate the *u* and *y* data. Thereafter, variables are pre-processed to mean-center them. Next, an ARX model[13] is fitted and used to make predictions.

```
# fit ARX model and compare training predictions vs observations
Id_ARX = system_identification(y_centered, u_centered, 'ARX', ARX_orders=[3,3,2])
y_centered_pred = np.transpose(Id_ARX.Yid) # Yid gives predictions during identification
```

fitted model ➡

$$y(k) - 0.98y(k-1) + 0.02y(k-2) + 0.12y(k-3) = 0.02u(k-1) + 0.06u(k-2) + 0.07u(k-3)$$

```
plt.figure()
plt.plot(y_centered, 'o', label='raw data')
plt.plot(y_centered_pred, '--', label='ARX fit')
plt.legend(), plt.xlabel('k'), plt.ylabel('y')
```

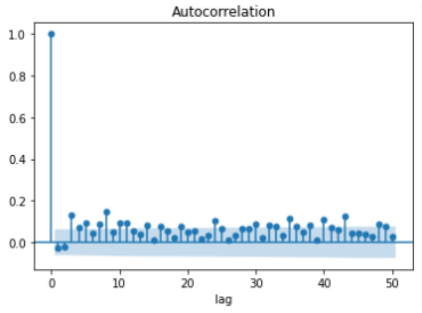


In the last step, model quality assessment is done.

```
# Assess model accuracy
print('Fit accuracy = ', r2_score(y_centered, y_centered_pred))
>>> Fit accuracy =  0.998
```

```
# plot ACF of model residuals
res = y_centered - y_centered_pred
plot_acf(res, lags=50)
plt.xlabel('lag')
```

➡



The residual plot suggests that further model refinement is warranted. In the later chapters, we will learn how this inference has been made and what recourses are at our disposal for model refinement.

[13] SIPPY: Model coefficients can be obtained via the *G* (or NUMAERATOR and DENOMINATOR) attribute(s) of the model object. See SIPPY's manual for more details.

`

Although there are many more advanced ML and SysID aspects, the above code is very representative of what goes in a ML-DPM script. Hopefully, you also got a feel of the powerful capabilities of the Sklearn and SIPPY libraries that help to encapsulate complex ML algorithms within user-friendly functions.

Summary

In this chapter we made ourselves familiar with the scripting environment that we will use for writing and executing ML scripts. We looked at two popular IDEs for Python scripting, the basics of Python language, and learnt how to manipulate data using NumPy and Pandas. We also looked at SysID-relevant libraries available in Python and saw what a typical SysID Python script may look like. In the next chapter you will learn several graphical tools that are utilized for preliminary investigation and visualization of dynamic dataset.

`

Chapter 3

Exploratory Analysis and Visualization of Dynamic Dataset: Graphical Tools

In Chapter 1 we had remarked that system identification is an art. A part of the art lies in making right inferences from exploratory analysis of data and model residuals via visual plots. Yeah, you read that right: even with all the fancy algorithms at our disposal, sometimes visual inspection of data is still the best tool for a quick assessment of dataset and model validity. Visual plotting can often provide crucial clues about model structure and model troubleshooting, and forms an important component of deductive phase of SysID. These plots will be used repeatedly in the in-chapter illustrations in this book and therefore we thought it would be best to introduce the concepts behind these graphical tools right away.

Visual plots can range from simple time plots to advanced spectral density plots. These may help us make some quick educated judgement about data stationarity, deterministic vs stochastic trends, presence of colored noise, whiteness of model residuals, etc. Don't worry if you don't understand these dynamic modeling-specific jargons for now. We will focus on generation and conceptual understanding of these plots in this chapter and learn inference-making using these plots in the later chapters.

Specifically, we will cover these topics

- Autocorrelation and autocovariance plots
- Partial autocorrelation plots
- Cross-correlation and cross-covariance plots
- Spectrum, spectral density plots, periodogram

`

3.1 Visual Plots: Simple Yet Powerful Tools

In system identification, careful scrutiny of data plots forms a critical component right from the initial step of exploratory data analysis (EDA) to the last step of model validation. Consider plots *a* and *b* in Figures 3.1 which are time plots of output signals in two different scenarios. Just a visual inspection makes it apparent that the raw output signals contain trends. One can further infer that while plot *a* exhibits a deterministic trend, plot *b* exhibits a stochastic trend[14]. Such assessment during EDA can help a process modeler select appropriate class of models for their systems.

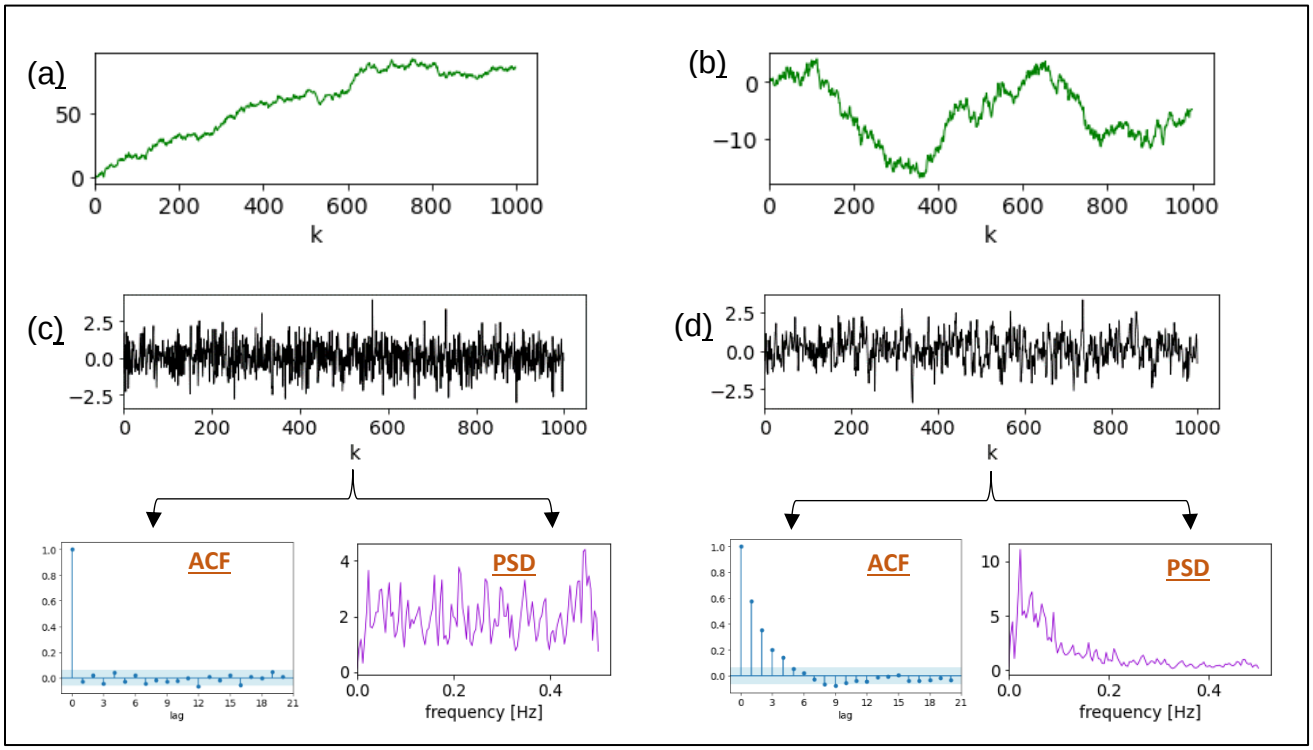


Figure 3.1: Visual plots for time series data

Consider further the plots 3.1*c* and *d*. Here, the shown noisy signals could correspond to the disturbance signals or the model residuals (difference between observations and model predictions). Here, unlike the previous example, the time plots don't provide any quick obvious assessment. However, the derived visualizations, autocorrelation and spectral density plots make the distinction between the two scenarios very evident. These plots confirm that the shown signals have white noise and colored noise properties, respectively. If these were model residuals, then non-zero autocorrelations in plot 3.1*d* immediately suggest inadequate modeling. Moreover, the spectral density plot looks qualitatively similar to that of an autoregressive (AR) process and therefore, an AR model could be attempted for the colored

[14] We will see the details on the distinction between these two types of trends in Chapter 4.

`

residuals. Alternatively, if these were process disturbance signals, then the white-noise like disturbance in plot 3.1*c* would encourage a process modeler to employ OE models[15].

These were just a sample of the kind of inferences one can draw using visual plots. Before we begin making use of these plots, it would be wise to understand the concepts behind them and this chapter is devoted to this purpose. If you are already familiar with these plots, then you can skip the rest of this chapter. Let's begin with the autocorrelation plots.

Time-domain vs frequency-domain analysis

Broadly, there are two distinct approaches to modeling dynamic systems and analyzing dynamic dataset: time-domain approach and frequency-domain approach. The models that we have discussed so far are time-domain models where the focus is on modeling future values as a function of the past and present values. Such models are primarily employed for predictions/forecasting and are popular among general DPM practitioners.

In the frequency-domain approach, the time-series signals are decomposed into their Fourier representations (summation of sinusoidal components) and the focus is on analyzing the output behavior of a system as a function of input signal frequency. Frequency-domain approach renders itself quite useful for characterizing bias and variance properties of models. Time-domain models are easier to understand for new entrants to DPM and will be the subject of focus in this book. Relevant frequency-domain concepts will be touched upon wherever needed.

3.2 Autocorrelation Function (ACF)

In Figure 3.1, we saw how ACF plots can make the difference between time-series signals very apparent. The inferences that can be drawn from these two ACF plots is even more interesting: in series (*d)*, the future values can be predicted using the past measurements while in series (*c)*, prediction of future values is not possible. How does ACF plots allow us to make such inferences? To understand this, let's revisit the classical concept of correlation. You probably already know that correlation coefficient quantifies the linear relationship between two variables and is given as follows

[15] The rationale behind this statement will become clear to you in Chapter 7.

`

$$\rho = \frac{\sum_{k=0}^{N-1}(y(k)-\bar{y})\,(u(k)-\bar{u})}{\sqrt{\sum_{k=0}^{N-1}(y(k)-\bar{y})^2\,(u(k)-\bar{u})^2}}$$

N is the number of samples; $\bar{u}$ and $\bar{y}$ are sample means of $u(k)$ and $y(k)$, respectively

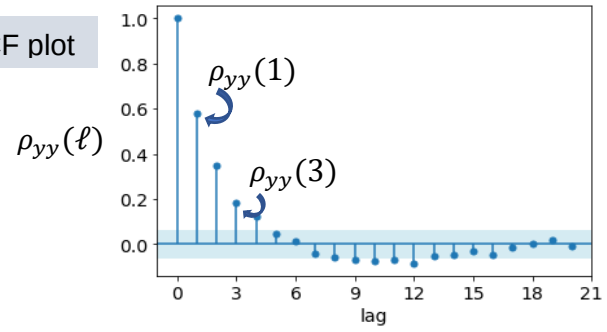
For a single time series *y(k),* correlation takes the form of autocorrelation

$$\rho_{yy}(\ell) = \frac{\sum_{k=\ell}^{N-1}(y(k)-\bar{y})(y(k-\ell)-\bar{y})}{\sum_{k=0}^{N-1}(y(k)-\bar{y})^2} \qquad \text{eq. 1}$$

N is the length of time series

Here, $\rho_{yy}(\ell)$ quantifies the linear relationship between lagged values of the time series ℓ distance apart. For example, $\rho_{yy}(2)$ measures the correlation between *y(k)* and *y(k-2)*. The plot (sample plot below) of different correlation coefficients versus lag ℓ is called autocorrelation function or ACF.

ACF plot



ACF properties

- $\rho_{yy}(0) = 1$
- $-1 \leq \rho_{yy}(\ell) \leq 1$
- $\rho_{yy}(\ell)$ is symmetric ($\rho_{yy}(\ell) = \rho_{yy}(-\ell)$) and therefore, ACF is plotted usually only for $\ell \geq 0$.

In Eq. 1, there is an implicit assumption that y(k) and y(k- ℓs) are related in the same way as y(k+q) and y(k+q- ℓ) for any q. Another assumption is that the mean $\bar{y}$ is invariant to the sampling location k. These assumptions result in ACF being a function of only the distance ℓ and Eq. 1 being valid for only stationary processes. We will learn more about stationarity in Chapter 4.

`

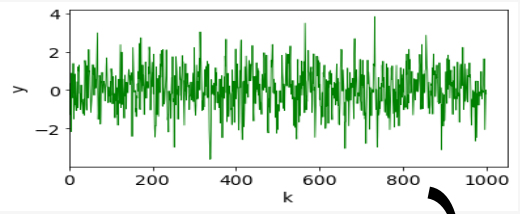
Example 3.1:

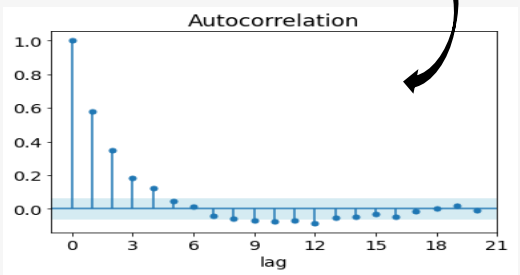
Consider the stochastic process below that generates time series *y(k)*

$$y(k) = 0.6y(k-1) + e(k)$$

Data-file SimpleTimeSeries.txt contains 1000 observations of *y(k)*. The code below generates the ACF plot.

```
# import packages
import numpy as np
from statsmodels.graphics.tsaplots import plot_acf
import matplotlib.pyplot as plt


# read data
y = np.loadtxt('simpleTimeSeries.csv', delimiter=',')


# generate ACF plot
conf_int = 2/np.sqrt(len(y))
plot_acf(y, lags= 20, alpha=None)
plt.gca().axhspan(-conf_int, conf_int, facecolor='lightblue') # confidence interval
```





The shaded region in the above ACF plot represents the (95%) confidence interval where the autocorrelation coefficients are expected to fall into for a white noise signal (signals whose values are uncorrelated). For the signal *y(k)*, values are obviously correlated and therefore, the coefficients are significant (lie outside the shaded region).

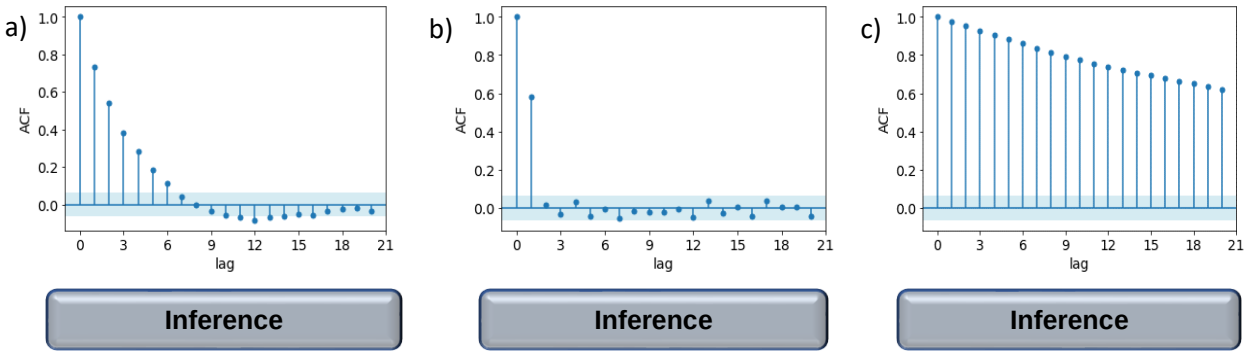
Significant autocorrelations indicate linear dependency of future values of a time series on its past values. Hopefully, now you can answer the question we posed at the beginning of this section about how ACF allows us to make inferences about predictability in Figure 3.1.

Examples on inferences drawn using ACF

- *Model structure selection*:

  Illustrations below show some sample ACF plots and the inferences that can be drawn regarding potential model structure

`



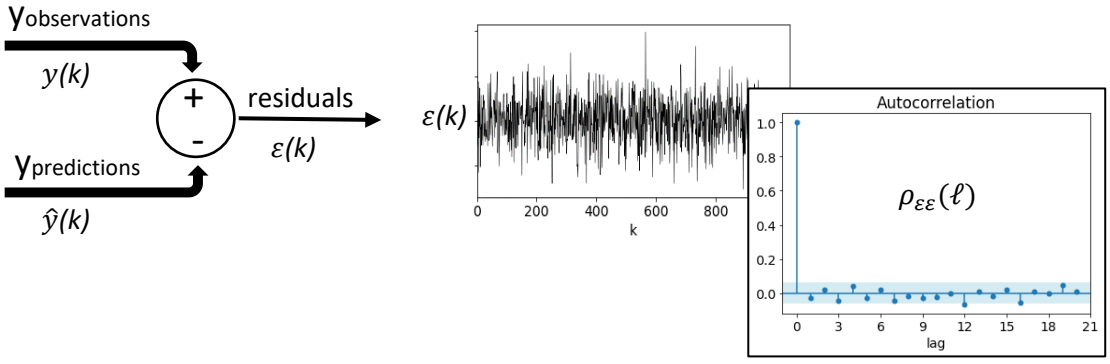
- Exponential decay of ACF values
- Signal is potentially from an AR process

- Sudden drop in ACF values
- Signal is potentially from a MA process

- ACF values fall very slowly
- Process is close to non-stationarity

- *Residual diagnostics*:

Residuals are differences between observed and predicted values of a signal. Consider the residual signal time plot and its ACF below



Here, we observe that the ACF values are not significant and therefore the model (that generated the predictions, $\hat{y}(k)$) has been able to capture all the linear relationships in data, leaving no leftover pattern in the residuals.[16]

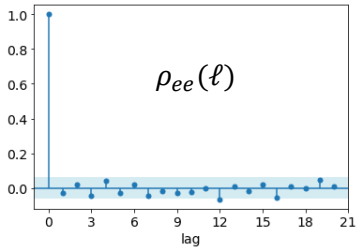
[16] An un-normalized version of ACF, called autocovariance function (ACVF), also exists and is gives as $\sigma_{yy}(\ell) = \frac{1}{N}\sum_{k=\ell}^{N-1}(y(k)-\bar{y})(y(k-\ell)-\bar{y})$. However, the covariance coefficients are not bounded and therefore, utilized less compared to correlation coefficients.

`

White Noise

Now that we understand ACFs, let's formally describe the term 'white noise' that we have already used a few times. A stochastic sequence *e(k)* is referred to as white noise if its samples have a probabilistic distribution with zero mean and finite variance (say σ^2), and the following theoretical ACF

$$\rho_{ee}(\ell) = \begin{cases} 1, & \ell = 0 \\ 0, & \ell \neq 0 \end{cases} \quad \ggg$$

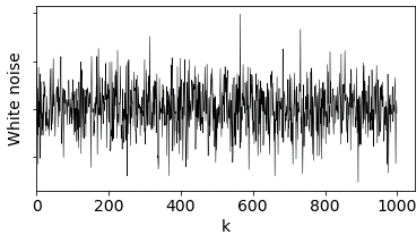


Therefore, samples of *e(k)* are serially uncorrelated random variables. In practice, the autocorrelation coefficients may not be exactly zero but will be close to zero. Nonetheless, it can be expected that 95% of the coefficients will lie within the $\pm 2/\sqrt{N}$ band as shown in Example 3.1. Also note that although gaussian white noise is commonly used, there are no constraints on the probability distribution of the samples. Accordingly, you can encounter the following notation in SysID literature

$e(k) \sim WN(0, \sigma^2)$ (white noise) or $e(k) \sim GWN(0, \sigma^2)$ (gaussian white noise)

A related term is iid (independent and identically distributed) noise. Every iid sequence is WN but not conversely. The easiest way to generate a white noise signal is to use Numpy's random normal function as follows

```
# import packages
import numpy as np, matplotlib.pyplot as plt

# generate GWN (1000 observations, mean 0, variance 1)
e = np.random.normal(loc=0, scale=1, size=1000)
plt.plot(e, 'black', linewidth=0.5)
plt.ylabel('White noise'), plt.xlabel('k')
```

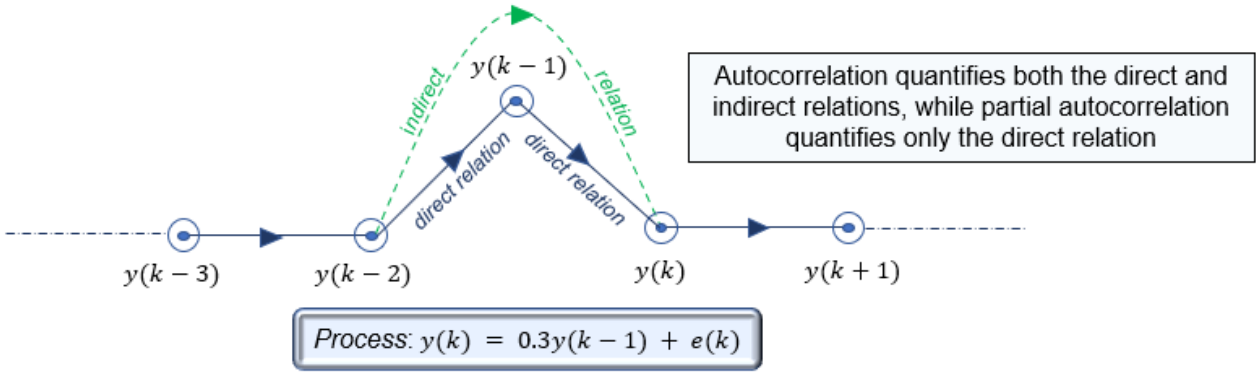


The use of the term 'white' to describe such a noise stems from its spectral properties which we will study shortly. Any noise signal that is not white is a 'colored' noise signal. In essence, a white noise is perfectly unpredictable, i.e., no amount of past observations can provide any clues about future observations. Nevertheless, such signals are the basic building blocks for building complex models of disturbance signals (more on this in Chapter 5).

`

3.3 Partial Autocorrelation Function (PACF)

Consider again the process in Example 3.1. The ACF plot suggests that *y(k)* and *y(k-2)* are correlated (and so are *y(k)* and *y(k-3)* and so on). This seems to indicate some inconsistency with the governing difference equation which relates only the successive samples *y(k)* and *y(k-1)*. However, it is easy to see that *y(k)* and *y(k-2)* end up being correlated because *y(k-1)* and *y(k-2)* are directly correlated.



In time series analysis, knowing direct correlation helps to decide what terms should be part of the governing difference equation during model structure selection. To compute the direct correlation between *y(k)* and *y(k-2)*, the effects of *y(k-1)* are 'discounted' or removed from both *y(k)* and *y(k-2)*. The correlation between the discounted variables gives the direct correlation or, more formally, partial correlation. Algorithm below shows the procedure for computation of partial (auto)correlation

Algorithm: Partial autocorrelation between *y(k)* and *y(k-2)*

1. Perform least squares fit between *y(k)* and *y(k-1)* as well as between *y(k-2)* and *y(k-1)* separately
2. Generate predictions from the least squares models and compute the following residuals

$$\epsilon_{y_k} = y_k - \hat{y}_k \quad \text{and} \quad \epsilon_{y_{k-2}} = y_{k-2} - \hat{y}_{k-2}$$

discounted variables

3. Partial autocorrelation coefficient $\varphi_{yy}(2) \equiv$ correlation between ϵ_{y_k} and $\epsilon_{y_{k-2}}$

The partial autocorrelation coefficient for lag 3 can be computed following the above algorithm wherein the effects of *y(k-1)* and *y(k-2)* are discounted from *y(k)* and *y(k-3)*. Other coefficients

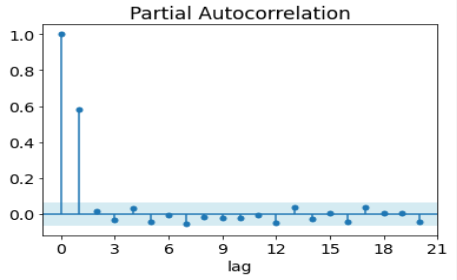
`

can be computed similarly and plotted versus lags to give partial autocorrelation function (PACF). Let's revisit our earlier example.

Example 3.1 continued:

The code below generates the PACF plot.

```
# generate PACF plot
conf_int = 2/np.sqrt(len(y))
plot_pacf(y, lags= 20, alpha=None)
plt.gca().axhspan(-conf_int, conf_int, facecolor='lightblue')
```



The PACF plot is along the expected lines. Only the coefficient at lag=1 is significant. Note that at lag=1, PACF and ACF values are the same as there are no intermediate variables. Moreover, PACF value at lag=0 is not defined but is often set to unity for visualization purposes.

The utility of PACF should be clear to you now. Another interpretation of PACF is worth mentioning. Suppose you fit the following (p^{th} order) autoregressive model.

$$y(k) = a_1 y(k-1) + a_2 y(k-2) \cdots + a_p y(k-p) + e(k)$$

The coefficient or weight (a_p) corresponding to the p^{th} lagged regressor (*y(k-p)*) equals the PACF value for lag *p* or $\varphi_{yy}(p)$. Therefore, the lag corresponding to the last significant PACF value in a PACF plot can indicate the number of terms to be used in an autoregressive model. More on this later in the chapter on time series analysis.

`

3.4 Cross-correlation Function (CCF)

Cross-correlation measures the linear relationship between two sequences, say, *u(k)* and *y(k)*, and is estimated as

$$\rho_{yu}(\ell) = \frac{\sum_{k=\ell}^{N-1} (y(k)-\bar{y})(u(k-\ell)-\bar{u})}{\sqrt{\sum_{k=0}^{N-1}(y(k)-\bar{y})^2 \; (u(k)-\bar{u})^2}} \qquad \text{eq. 2}$$

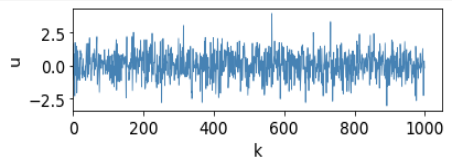
The plot of cross-correlation coefficients versus lag ℓ is called cross-correlation function.

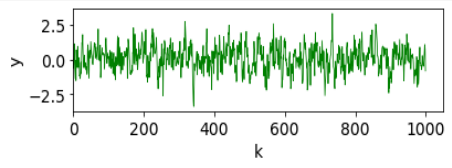
Example 3.2: Consider the input-output process below

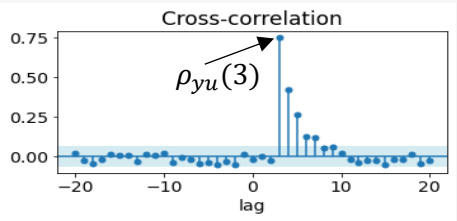
$$y(k) = 0.6y(k-1) + 0.7u(k-3) + e(k)$$

Data-file SimpleInputOutput.txt contains 1000 observations of *y(k) and u(k)*. The code below generates the CCF plot.

```
# import packages
import numpy as np, matplotlib.pyplot as plt
from statsmodels.tsa.stattools import ccf

# read data
data = np.loadtxt('simpleInputOutput.csv', delimiter=',')
u = data[:,0]; y = data[:,1]

# compute CCF values for lags = -20 to 20
ccf_vals_ = ccf(y, u) # ccf for lag >= 0
ccf_vals_neg_lags = np.flip(ccf(u, y)) # ccf for lag -1000 to 0
ccf_vals = np.hstack((ccf_vals_neg_lags[-21:-1], ccf_vals_[:21]))

# generate CCF plot
conf_int = 2/np.sqrt(len(y))
lags = np.arange(-20,21)
plt.plot(lags, ccf_vals) # no in-built analogue of plot_acf is provided by Stattools
plt.gca().axhspan(-conf_int, conf_int) # confidence interval
```







This property is used in the code above to compute ccf values for lags <=0

CCF Properties

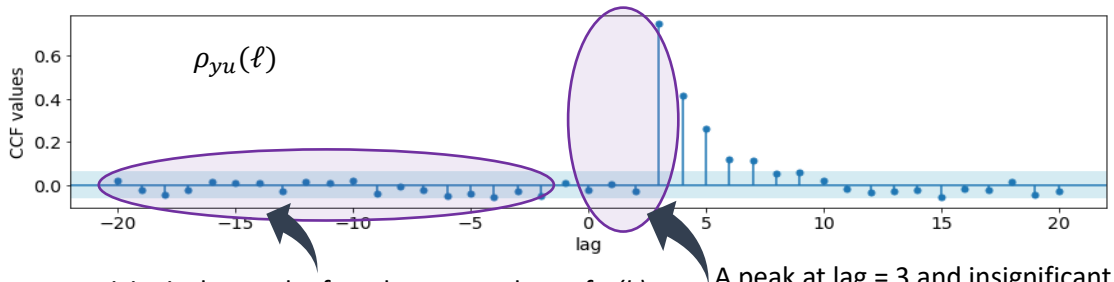
- $-1 \leq \rho_{yu}(\ell) \leq 1$
- $\rho_{yy}(\ell)$ *is asymmetric, i.e.,* $\rho_{yu}(\ell) \neq \rho_{yu}(-\ell)$ *and* $\rho_{yu}(\ell) = \rho_{uy}(-\ell)$

`

Examples on inferences drawn using CCF[17]

- *Model structure selection*:

  The CCF between *u(k)* and *y(k)* can give some crucial hints about the time delay (dead time (*d*)) of the process[18]. Consider the following scenario and the fact that dead time is the smallest value *d* with which *u(k-1-d)* influences *y(k)* directly.



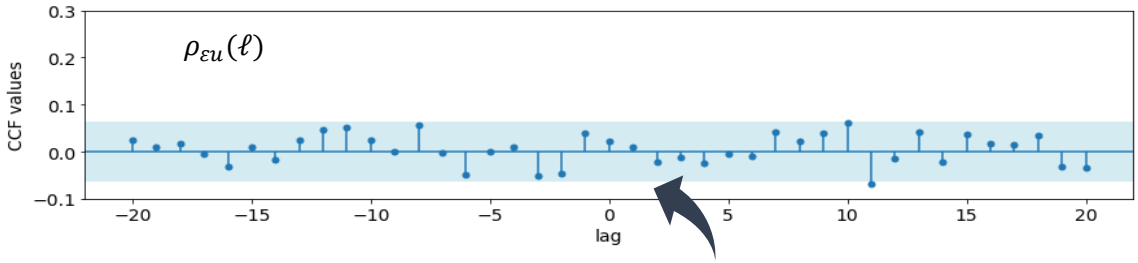
Asymmetricity is due to the fact that past values of *u(k)* are related to *y(k)* differently than the future values

A peak at lag = 3 and insignificant CCF values for lag <=2

⇒ A good guess for *d* is 2

- *Residual diagnostics*:

  If a model has adequately captured the relationship between input and output, then $\rho_{\varepsilon u}(\ell)$ between the residual ($\epsilon(k)$) and the input sequences (*u(k)*) should be nearly zero for positive[19] lags because $\epsilon(k)$ should not have any relationship with past input values.[20]



Model residuals are uncorrelated with input ⇒ no effect of input remains in the residuals

[17] Like ACF, CCF also has a covariance counterpart called cross-covariance function (CCVF) and is given as $\sigma_{yu}(\ell) = \frac{1}{N}\sum_{k=\ell}^{N-1}(y(k)-\bar{y})(u(k-\ell)-\bar{u})$. Note that *N*-$\ell$ could be set as denominator instead of *N* to get unbiased estimates.

[18] We will formally introduce the term *dead time* in Chapter 6. But we figured that if you already knew what dead time is then you may appreciate this example.

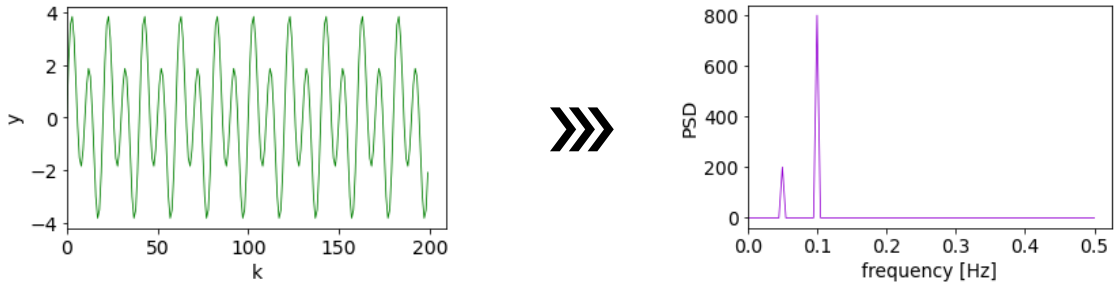
[19] Significant correlation for negative lags implies relation between residuals and future input values which is an indication of closed-loop feedback (and not necessarily model inadequacy). We will learn more on this later.

[20] A small caveat here is that either $\epsilon(k)$ or *u(k)* should be white to impose this condition

`

3.5 Power Spectral Density (PSD) and Periodogram

A dynamic signal's properties can also be described in terms of its frequency contents, i.e., by decomposing the signal into sinusoidal components with different frequencies. The analysis in frequency domain is also called spectral analysis and for the stochastic discrete-time signals, the tool of choice is (power) spectral density plot that can provide a quick glimpse of strength of different frequency components. For example, consider the following time series and its spectral density plot. The spectral plot makes it easy to infer that the components with frequencies 0.1Hz and 0.05Hz dominate the signal.



In our motivating illustration in Figure 3.1, we saw that the white noise signal has approximately uniform contributions from all frequencies. Infact, this is a defining characteristic of any white-noise signal (analogous to white light which is a mixture of all visible frequencies of light). The common practice to generate PSD plots for sampled time-series data is to use periodograms. Without going into the mathematical details, it suffices to state that a periodogram plots the square of the magnitude of the Fourier transform as a function of frequency, and therefore, shows how the intensity or power of a dynamic signal is distributed across frequencies. Let's see an instance of a PSD-based inference in the following example.

Example 3.3:

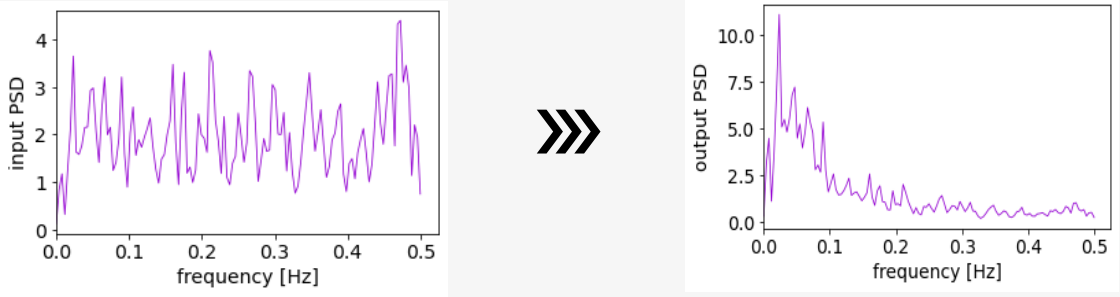
We will consider the input and output time-series signals from Example 3.2, study their spectral properties, and attempt to make some modeling-related inferences. The code below generates the periodograms for the input and output signals

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

# read data
data = np.loadtxt('simpleInputOutput.csv', delimiter=',')
u = data[:,0]; y = data[:,1]
```

`

```
# input PSD
freq, PSD = signal.welch(u)
plt.figure(), plt.plot(freq, PSD)
plt.ylabel('input PSD'), plt.xlabel('frequency [Hz]')

# output PSD
freq, PSD = signal.welch(y)
plt.figure(), plt.plot(freq, PSD)
plt.ylabel('output PSD'), plt.xlabel('frequency [Hz]')
```

Welch method estimates PSD as averaged periodograms of overlapping segments of a signal; Averaging reduces noisy fluctuations in PSD values



It is apparent from the comparison of the spectral densities that the process has low-pass filter characteristics and therefore a model (such as ARX) with such characteristics can be a good initial choice .

This concludes our quick look at some of the popular graphical tools for analyzing time series data. Hopefully, you now have sufficient knowledge of these tools to understand their specific applications in the upcoming chapters.

Summary

In this chapter we studied the concepts behind autocorrelation function (ACF), partial autocorrelation function (PACF), cross-correlation function (CCF), and periodograms. We also looked at some of the applications of these graphical tools. These tools come in handy for analyzing time series and input-output data, and are a crucial part of the fundamentals upon which you will develop your ML-DPM expertise. In the next chapter we will continue building the fundamentals and look at some established guidelines for SysID.

`

Chapter 4

Machine Learning-based Dynamic Modeling: Workflow and Best Practices

In Figure 1.4 in Chapter 1 we looked at a typical workflow for system identification. What is noteworthy is that while model ID is only a part of this workflow, very often too much focus goes into application of model ID to estimate model parameters at the expense of other aspects of SysID. Dumping raw data into model ID module will invariably generate unsatisfactory model if data is not pre-treated appropriately. Additionally, you will seldom be right in your modeling choices in the very first attempt. The take-home message is that knowledge about how to prepare raw data to enhance its information content about the system, how to assess validity of obtained model, how to analyze modeling results critically, and how to iterate judiciously is absolutely critical for successful SysID. Fortunately, several guidelines and best practices have been devised to guide a process modeler at each step of the SysID workflow. We will learn these guidelines and best practices in this chapter.

We will not cover the best practices associated with generic machine learning workflow. Concepts like feature extraction, feature engineering, cross-validation, regularization, etc. have already been covered in detail in our first book of the series. In this chapter our focus will be on the unique challenges presented by dynamic processes and the specific best practices to deal with them. Specifically, the following topics are covered

- Identification test design using PRBS and GBN signals
- Pre-treatment of raw data for removal of measurement noise, offsets, trends, and drifts
- Guidelines around selection of model structure
- Model order selection via AIC and cross-validation
- Model quality assessment via residual analysis, simulation response analysis, etc.

`

4.1 System Identification Workflow

So far in this book we have been consciously emphasizing the difference between model identification and system identification. While model ID is all about estimating model parameters, SysID is about ensuring that the model truly represents the underlying system and stands consistent w.r.t. any prior available knowledge and any modeling assumptions. Figure 4.1 reproduces the SysID workflow we had seen in Chapter 1 and shows the several steps that serve to achieve the stated goals. The rest of the chapter will sensitize you about these varied aspects of SysID and help you gain working-level understanding about how to successfully execute a SysID project.

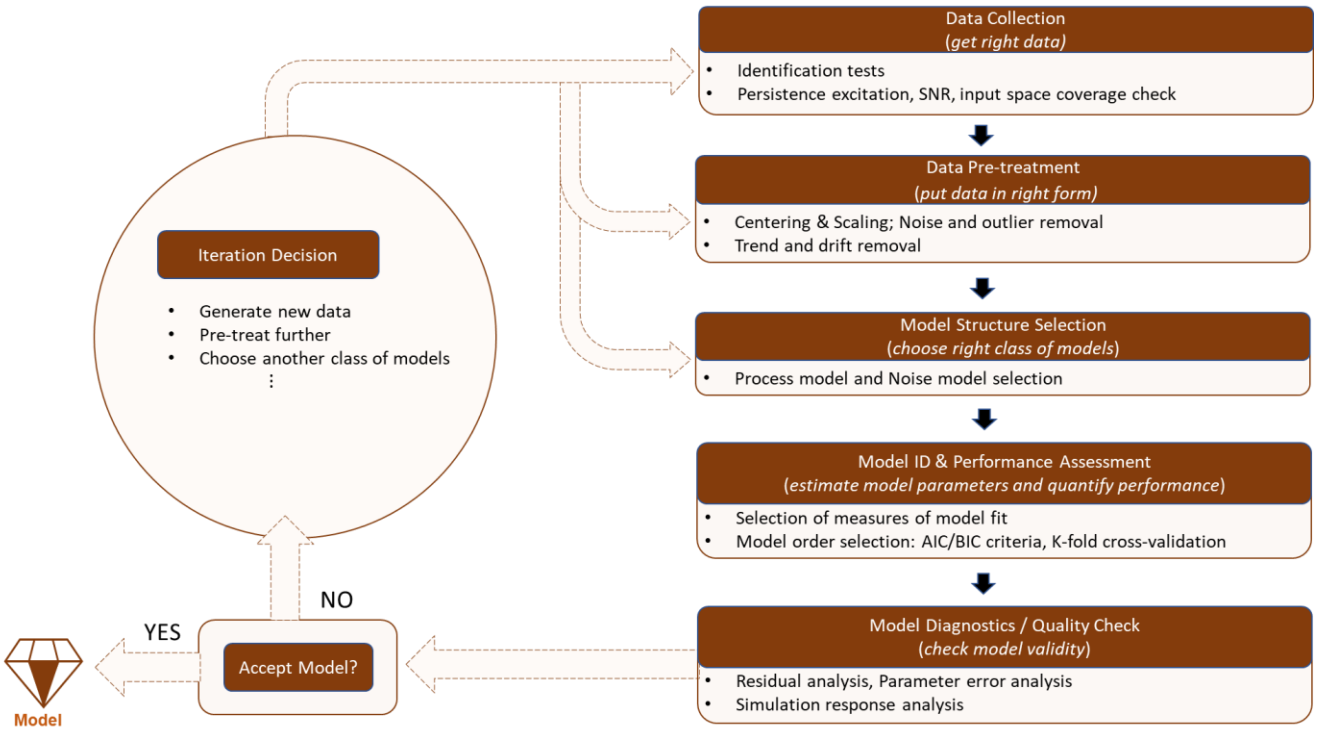


Figure 4.1: Typical steps in a SysID workflow that we will cover in this chapter

The first step of the workflow is data collection. Whether you are using pre-existing historical data or conducting fresh identification tests, you must ensure that the dataset has enough 'juice' in it to enable identification of the desired model. We will soon see some guidelines around this task. The next step corresponds to sanitizing the data to remove the unwanted components that can negatively impact model ID. For example, elimination of any slow drift in the output signals is essential to avoid incorrect model parameters. After the first two steps, our dataset is ready and we come to model structure selection where the onus is on the process modeler to choose the right class of process and noise models. We saw in Chapter 1 that an incorrect choice can lead to biased parameters estimates. Your expert domain knowledge and any prior information on process disturbances can come in quite handy at this

`

step. For example, this is the step where you make a decision between ARX versus ARMAX, or ANN versus CVA models. A generic thumb rule is to start with a simple model (like AR, ARX) and then use inferences from model diagnostics to shortlist the next candidate model(s).

Once a model's selection has been made, the model parameters can be estimated using off-the-shelf Python packages. These packages also include functionalities for automatic selection of optimal model order, e.g., the number of auto-regressive terms in an ARX model. The estimated model must always undergo extensive check to ensure that the model is right for your process system and is consistent with the modeling assumptions and prior process knowledge. We will learn about several of these model quality checks later in the chapter. Any model deficiency encountered will necessitate appropriate correction and another pass-through the whole workflow. Let's now learn about each of these tasks one-by-one.

4.2 Identification Test/Input Signal Design

The first and foremost requirement for a successful model ID is that the model training data (whether historical or newly generated) must be informative. The traditional practice for fresh data collection as observed in industry (especially for model development for control purposes) is to induce step changes in each input separately and record the step responses. Figure 4.2 shows an example. While these step tests are simple enough to be understood by plant operators and provide good information on steady-state behavior of the process, there are several shortcomings. When the process has several inputs, the step tests take a lot of time and manpower. Moreover, step tests are known to not excite all the plant dynamics significantly and therefore the risk of inaccurate model ID is high.

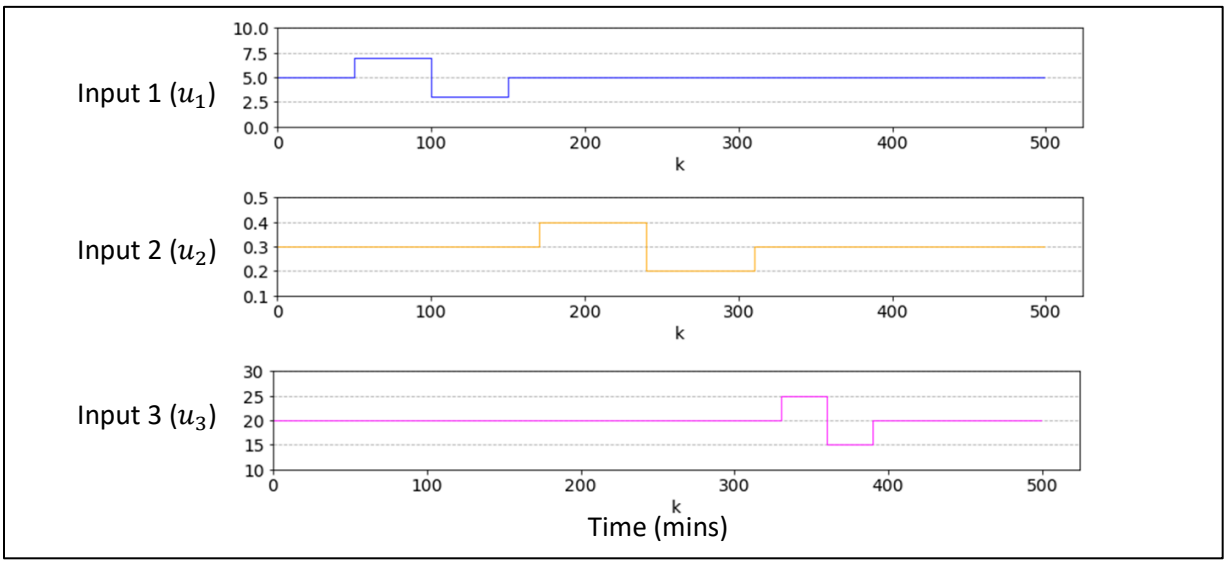


Figure 4.2: Traditional open loop identification test strategy

`

The ideal identification test[21] input signals should show a lot of fluctuations or excitations. Let's study two such signals, namely, PRBS and GBN.

Persistent Excitation

Consider the following process

$$y(k) = b_1 u(k-1) + b_2 u(k-2) + b_3 u(k-3) + b_4 u(k-4) + e(k)$$

Suppose that the input signal is hypothetically of the form $u(k) = u(k-1) + 1$. Under such an input signal, the process can be re-written as

$$\begin{aligned} y(k) &= b_1 u(k-1) + b_2 u(k-1) - b_2 + b_3 u(k-1) - 2b_3 + b_4 u(k-1) - 3b_4 + e(k) \\ &= \acute{b_1} u(k-1) + b_0 \end{aligned}$$

The above derivation implies that the process behaves virtually like a two-parameter system and therefore the four original model parameters can't be uniquely determined with output data collected with the aforementioned input signal. In other words, the true process model is not identifiable due to non-informative data. To ensure identifiability, the input variables must be sufficiently excited. Mathematically, a discrete-time input signal *u(k)* is said to be persistently exciting of order *n* (required to uniquely identify a PEM model with $\leq n$ model parameters) if the following conditions are met

The following limit exists

$$r_u(\tau) = \lim_{N \to \infty} \frac{1}{N} \sum_{t=0}^{N} u_{t+\tau} u_t$$

The following matrix is invertible

$$R_u(n) = \begin{bmatrix} r_u(0) & r_u(1) & \cdots & r_u(n-1) \\ r_u(-1) & r_u(0) & \cdots & r_u(n-2) \\ \vdots & \vdots & \ddots & \vdots \\ r_u(1-n) & r_u(2-n) & \cdots & r_u(0) \end{bmatrix}$$

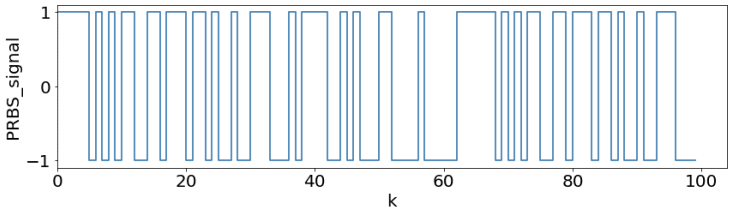
You will mostly not use these theoretical conditions directly, but it's worthwhile to know these: a step input has persistent excitation (p.e.) of order 1; a full length PRBS has p.e. of order *M;* a GBN signal and a white noise signal are persistently exciting with any finite order. Alternatively, in frequency domain, *u(k)* is persistently exciting of order *n* if its spectral density is not zero at *n* points in the interval (-π, π].

[21] Do not confuse between the process control literature term 'identification test' as used in the context of generating data for model ID and the machine learning term 'test data' used in the context of model validation.

`

Pseudo Random Binary Sequence (PRBS)

A PRBS signal switches between two levels with varying pulse widths as shown below. While the long width pulses help in estimation of steady-state gains, short pulses help in capturing transient dynamics.



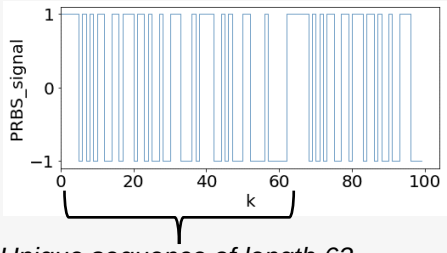
A PRBS signal of order 6

With order *n*, a unique sequence of length $M = 2^n-1$ can be generated. For larger signal length, the unique sequence repeates itself and therefore, has a period of length *M* as shown in the example below. Although a PRBS sequence is deterministic, for large *M* (and signal length $\geq M$), the signal has white-noise like properties and hence the term 'pseudo-random' in its name. Scipy library provides a function for PRBS generation. An example is shown below.

Example 4.1:

Let's generate a PRBS signal for an input fluctuating between values -1 and 1.

```
# import packages
from scipy.signal import max_len_seq
import matplotlib.pyplot as plt

# generate PRBS input signal of order 6 and plot
PRBS_signal = max_len_seq(6, length=100)[0]*2-1 # converting
sequence of 0's and 1's to sequence of -1's and 1's

plt.plot(PRBS_signal, 'steelblue')
plt.ylabel('PRBS_signal'), plt.xlabel('k')
```



Unique sequence of length 63

Generalized Binary Noise (GBN)

A GBN signal is similar to PRBS but non-periodic. It switches between two values *-a* and *a*, and the sequence is generated as follows

$$P[u(k) = -u(k-1)] = p_{sw}$$
$$P[u(k) = u(k-1)] = 1 - p_{sw}$$

; $p_{sw} \in (0,1)$ is the switching probability

`

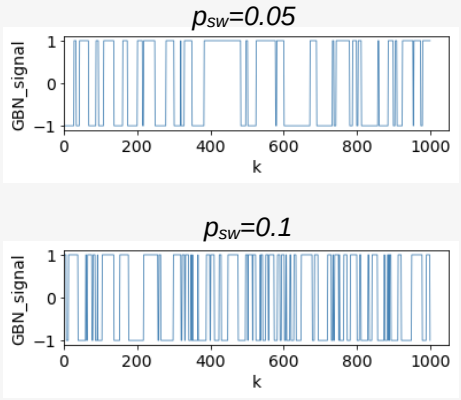
For p_{sw} = 0.5, GBN signal has white-noise like properties. However, a lower value of p_{sw} (0.1 or 0.05) is often recommended. SIPPY provides a function for GBN signal generation. An example is shown below.

Example 4.2:

Let's generate a GBN signal for an input fluctuating between values -1 and 1.

```
# import packages
from sippy import functionset as fset
import matplotlib.pyplot as plt

# generate GBN input signal using SIPPY and plot
[gbn_signal, _, _] = fset.GBN_seq(1000, 0.1])

plt.plot(gbn_signal, 'steelblue', linewidth=0.8)
plt.ylabel('GBN_signal'), plt.xlabel('k')
```

p_{sw}



In the plots, note how lower switching probability leads to higher pulse widths.

Persistence excitation is only a theoretical guarantee for model identifiability. Other practical requirements also warrant careful attention.

- *The physical levels of the test signals should be chosen large enough so that variations in output is greater than the variations due to noise. This property is quantified by a measure termed signal-to-noise ratio (SNR). A low SNR leads to inaccurate parameter estimates. A few words of caution here: If you are only interested in a linear model around some operating point, then you need to be wary of unwanted nonlinearities due to high test signal levels.*
- *If you are looking to build a nonlinear model, then the binary variations in input signal levels will not expose system nonlinearity. The level/amplitude of the test signal must also take a range of values.*

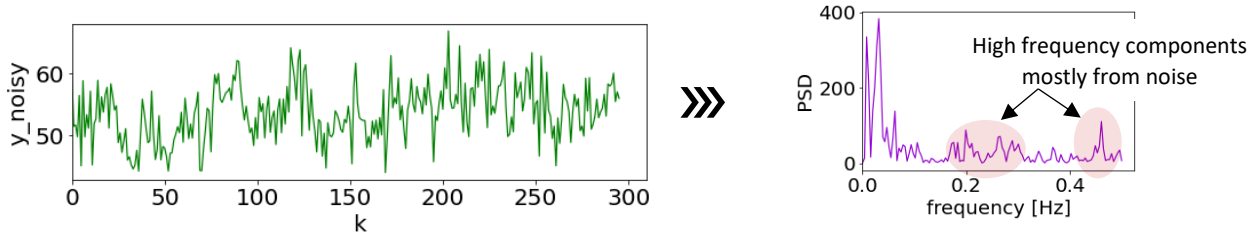
`

4.3 Data Pre-processing

Raw process data is seldom suitable for immediate use for model ID, even if obtained through well-designed identification tests. In this section, we will learn about some of the common practices used to handle the impurities that inadvertently creep into our dataset. Let's start with removal of high-frequency measurement noise.

(Measurement) Noise removal

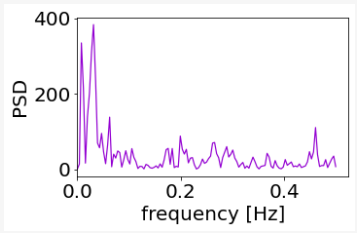
Consider the plot below which shows the observations of a temperature variable. Here, the high frequency wiggles on top of the apparently systematic variations are measurement noise and need to be removed before model ID. In time-domain, common methods for noise removal are moving-average smoothing and SG filtering.[22]



Another approach for noise removal is to work directly in frequency domain and filter out the undesirable high-frequency components of the signal. A common choice for such a filter is Butterworth filter[23]. The example below shows how to implement one such filter.

Example 4.3: Take the signal shown in the above motivating plot and clean it by removing its high-frequency components. Note that the noisy signal was generated by adding a white noise to the CO_2 concentration signal in the celebrated Box-Jenkins gas furnace dataset (https://openmv.net/info/gas-furnace).

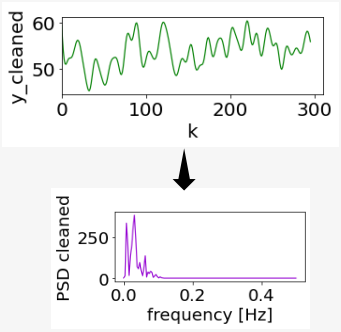
```
# import packages
import numpy as np, matplotlib.pyplot as plt
from scipy import signal

# read data and generate its periodogram
y_noisy = np.loadtxt('noisySignal.csv', delimiter=',')
freq, PSD = signal.welch(y_noisy)
plt.plot(freq, PSD), plt.ylabel('PSD'), plt.xlabel('frequency [Hz]')
```



[22] These were covered in detail in the first book of the series.
[23] Such filters are called low-pass filters as they allow/retain low-frequency components in the cleaned signal.

`

```
# apply Butterworth filter of order 5 with critical frequency as 0.2 Hz
b, a = signal.butter(5, 0.2, 'low') # create the filter
y_cleaned = signal.filtfilt(b, a, y_noisy) # apply the filter
plt.figure(), plt.plot(y_cleaned, 'g'), plt.ylabel('y_cleaned'), plt.xlabel('k')

# periodogram of cleaned data
freq, PSD = signal.welch(y_cleaned)
plt.figure(), plt.plot(freq, PSD)
plt.ylabel('PSD cleaned'), plt.xlabel('frequency [Hz]')
```



We can see how efficiently the noisy component has been removed. In practice, you will have to tune filter parameters cautiously so as to not filter out the useful components.

Centering and scaling

Dynamic models are often built to describe the behavior of the process around some desired operating point. Therefore, in SysID literature, you will find the usage of deviation variables which are defined as follows

$$y(k) = y_m(k) - \bar{y}\ ;\ u(k) = u_m(k) - \bar{u}$$

Where y_m, u_m are the actual raw measurements and $\bar{y}$, $\bar{u}$ are the nominal values around which the deviations are calculated. Models between *y(k)* and *u(k)* therefore relate the changes in inputs to the changes in outputs. An obvious option is to assign values to $\bar{y}$, $\bar{u}$ (also called offsets) corresponding to the desired steady-state operating point; however, this operating point may not be known beforehand. The recommended and common approach is to use the sample means as the offsets, i.e.,

$$\bar{y} = \frac{1}{N}\sum_{k=0}^{N-1} y_m(k) \quad ; \quad \bar{u} = \frac{1}{N}\sum_{k=0}^{N-1} u_m(k)$$

Another pre-treatment best practice is to scale the deviation variables to account for the differences in physical units of different variables. Scaling brings all the variables on an equal basis and prevents any variable being given undue high weightage due to its numerically high spread in values. A common choice is to use the standard deviation to scale any variable. Both centering and scaling can be accomplished using Sklearn.[24]

[24] Remember to transform the validation and test data using the same offset and scale determined using the model fitting data

`

Trend and drift removal

Consider the signals from Figure 3.1 in Chapter 3. These could be output signals from a purely stochastic process or disturbance signals from an input-output process as shown in the figure below. What is noteworthy about these signals is that they exhibit meandering behavior and don't seem to oscillate around some mean values. Therefore, these qualify as non-stationary signals. While signal *A* is said to have a trend, signal *B* shows a drift. In industrial processes, changes in ambient temperature, presence of process leaks, etc., could cause such disturbance signals. If these non-stationarities are not accounted for during SysID, inaccurate parameter estimates may result. However, methods exist to make the non-stationary signals stationary or remove the effect of non-stationarities from the input-output data. Let's learn these methods now.

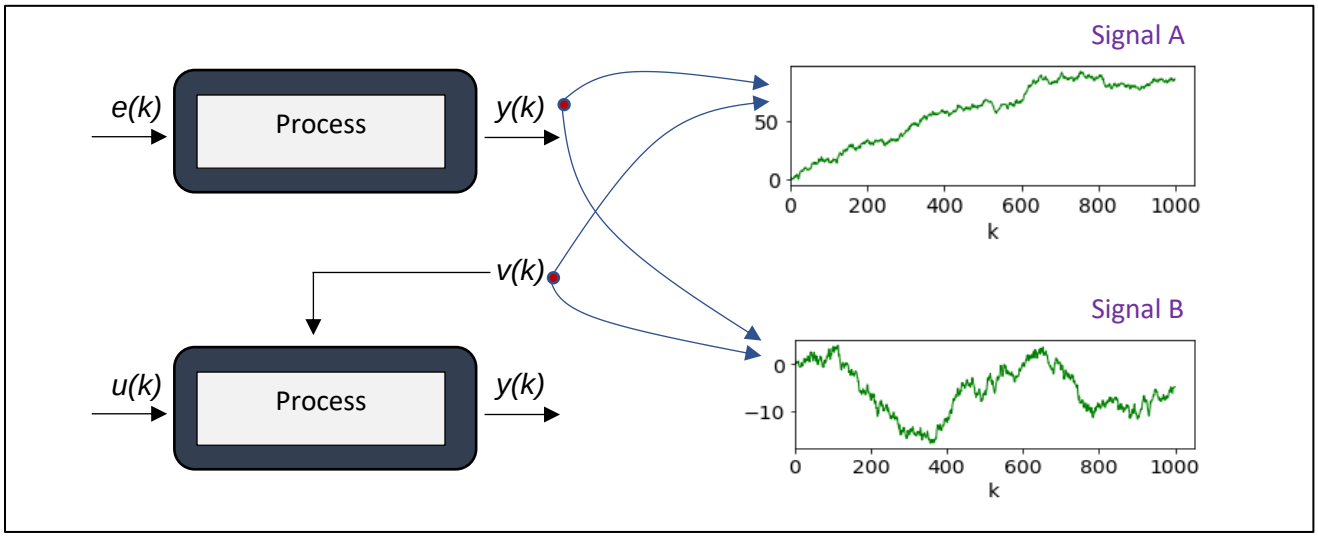


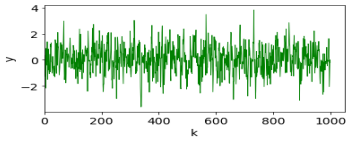
Figure 4.3: Non-stationary signals in a stochastic system and an input-output system

`

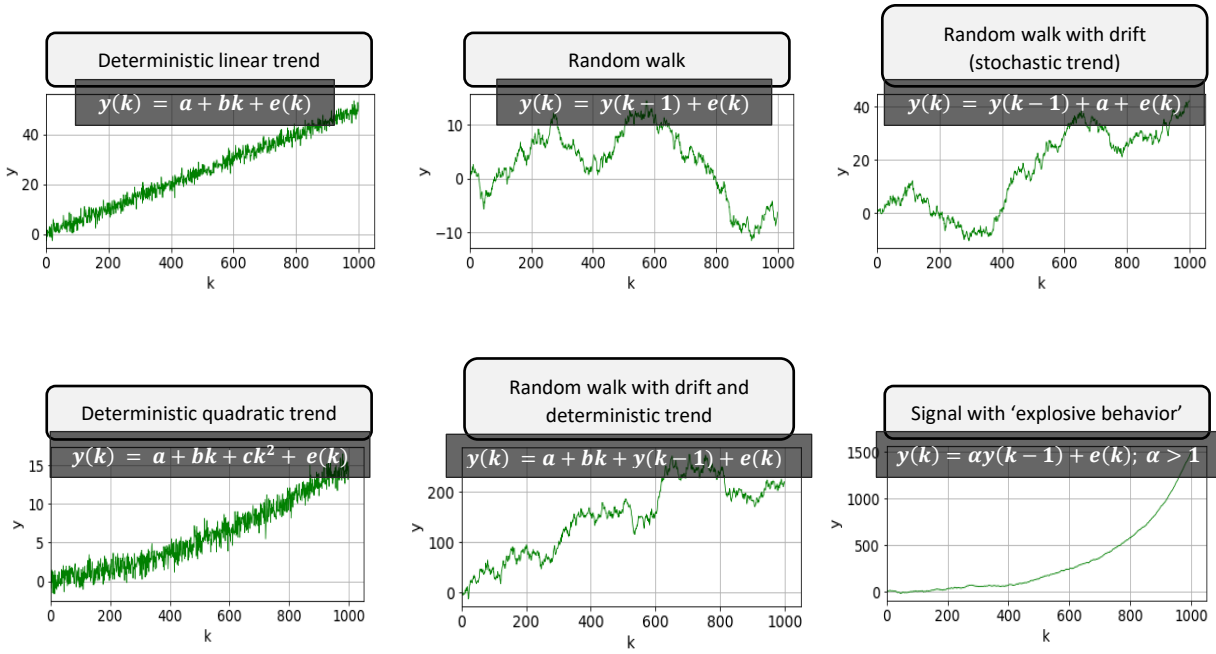
A quick primer on stationarity

Until now we have mentioned the term 'stationarity' a couple of times but haven't defined it rigorously. To understand this concept. Let's consider the following difference equation model fitted to the given data

eq. 1 $y(k) = 0.6y(k-1) + e(k)$ »»



A casual process modeler may not realize an implicit assumption being made on the validity of the model for any value of *k*, i.e., there is some sort of consistency in the behavior of the process irrespective of time. However, for the validity of such time-invariant behavior, the random signal *y(k)* must exhibit identical statistical properties; specifically, the mean and autocovariance (and therefore the variance as well) are constant*. Such signals are called stationary signals.** To see the behavior on the other side of the spectrum, consider the following non-stationary signals



Visually, signal (a) has an increasing mean and therefore is non-stationary. Signal (b) has a constant mean but an increasing variance that tends to infinity.*** Its visual characteristic is that it doesn't revert to its mean as frequently as a stationary signal does. Random walk is a simple process often used to demonstrate properties of non-stationary signals. Therefore, remember this term and the process. You will see later that *y(k) = y(k-1) + e(k)* can also be written as $y(k) = \sum_{j=0}^{k} e(j)$ and therefore such processes are also called integrating processes or processes with integrating type non-stationarity. The first five signals are further classified as homogeneous non-stationary because they can be transformed into stationary signals by suitable differencing or transformations.

`

In summary, stationarity imparts some sort of 'predictability' and enable development of models that could be used for prediction and forecasts. Attempts to fit stationary models to nonstationary signals will lead to spurious results and therefore this aspect warrants due caution from the process modeler. We will later study several qualitative and quantitative tests of stationarity.

---

* Eq. 1 can be used to derive the properties of *y(k)*. To see how, let's derive the mean ($\mu(k)$) of the signal.

$$y(k) = 0.6y(k-1) + e(k) = 0.6\big(0.6y(k-2) + e(k-1)\big) + e(k) = 0.6^2 y(k-2) + 0.6e(k-1) + e(k)$$

$$= 0.6^j y(k-j) + \sum_{i=0}^{j-1} 0.6^i e(k-i) \text{ [after j recursions]}$$

$$= 0 + \sum_{i=0}^{j-1} 0.6^i e(k-i) \text{ [as j } \to \infty]$$

$$\mu(k) = E[y(k)] = \sum_{i=0}^{\infty} 0.6^i E[e(k-i)] \text{ [E[.] denotes expectation of a signal]}$$

But $E[e(k-i)] = 0\ \forall i\ as\ e \sim WN \Rightarrow \mu(k) = 0 = constant!$

** Theoretically, signals as described are called weakly stationary or covariance stationary as only 2nd order moments have been considered. Strict stationarity occurs when all the statistical properties are constant. However, for most of the practical purposes, weak stationarity suffices and the qualification 'weakly' is omitted.

*** Repeated substitutions give $var(y(k) = var(e(k)) + var\big(e(k-1)\big) + \cdots = k\sigma^2 + var\big(y(0)\big) \Rightarrow$ variance of *y(k)* increases with k.

Detrending

As the method's name suggests, this method aims to remove trends from the non-stationary signals of the following type

$$y(k) = m(k) + z(k)$$

trend component — stationary component

`

The common practice for detrending is to fit (via least squares) a polynomial of an appropriate order (say $m(k) = a + bk + ck^2$ for quadratic order). The fitted polynomial is subtracted from the time-series signal and then a stationary time-series model is built for the residuals. For, the signal in Figure 4.4, the approach would look as follows

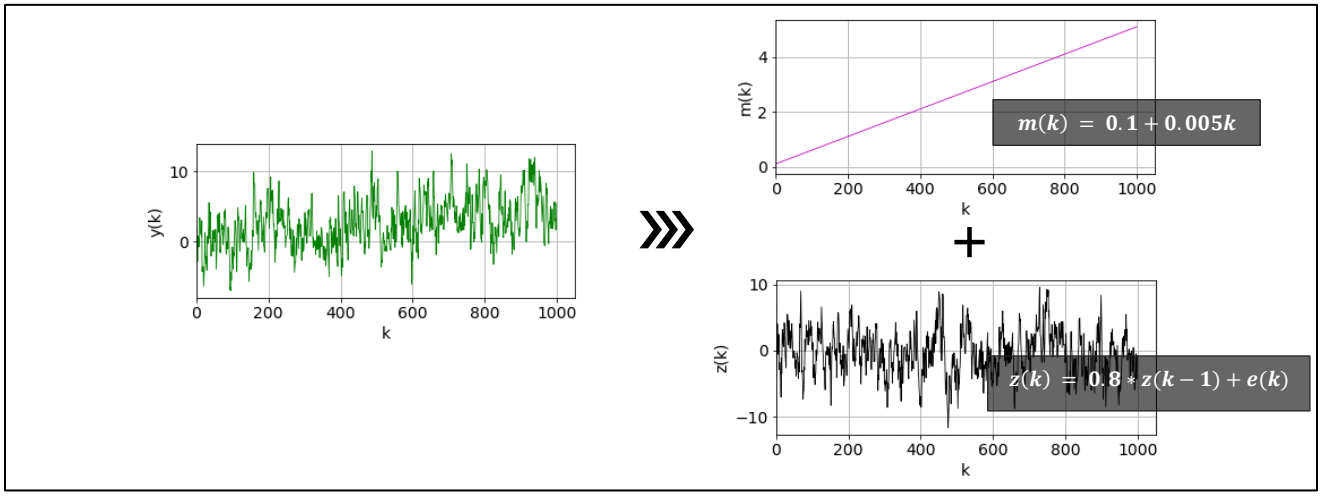


Figure 4.4: Breaking a signal into deterministic trend and stochastic residuals

Non-parametric method of detrending also exists and entail usage of moving average or low-pass filter to remove the slow dynamic (trend) components from the non-stationary signals.

Classical time-series decomposition

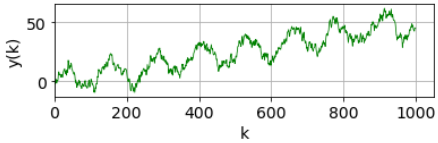
The method of breaking down a signal into its trend and random component belongs to a broader method called classical decomposition. Here, a signal *y(k)* is assumed to take the following form

$$y(k) = m(k) + s(k) + z(k)$$

trend component

random component

Here, the term *s(k)* represents the seasonal component to handle seasonal/periodic variations in data. One such signal with all three components is shown below



Classical decomposition models are popular in financial data analysis. The prime focus in this book is on I/O process models and therefore, seasonality is not considered.

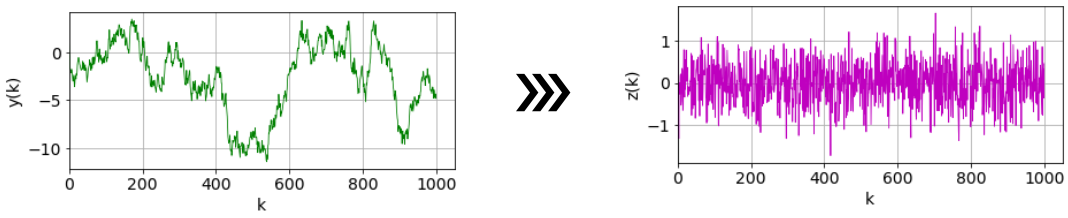
`

Differencing

An alternative and more preferable approach to dealing with non-stationary signals is differencing wherein consecutive samples of the signal are differenced to obtain stationary signals[25]. Post differencing, any suitable stationary model can be fitted to the differenced series. For example, consider a random walk process. Let *z(k)* be the differenced signal where

$$z(k) = y(k) - y(k-1)$$

$$\Rightarrow z(k) = e(k)$$ Stationary signal!



Sometimes differencing once is not enough and higher-order differencing may be needed to produce stationary signals. For example, consider the following

$$y(k) = 2y(k-1) - y(k-2) + e(k)$$

$$\Rightarrow z(k) = y(k) - y(k-1) = y(k-1) - y(k-2) + e(k)$$

$$\Rightarrow z(k) = z(k-1) + e(k)$$

$$\Rightarrow z(k) - z(k-1) = e(k)$$ ⇒ Stationary signal obtained after differencing 2 times

For input-output dataset, you won't have access to the disturbance signal directly. The recourse to get rid of non-stationarities in disturbance signals (*v(k)*) is to difference both the input and output signals and build a model between the difference signals. An example is shown below

$$\Delta y(k) = y(k) - y(k-1) \qquad \Delta u(k) = u(k) - u(k-1)$$

$$\Delta y(k) = 0.3\Delta y(k-1) + 1.2\Delta u(k-1) + e(k)$$ ⇐ ARX model fitted between the differenced signals

[25] Such non-stationarity is referred to as homogeneous non-stationarity.

`

The above approach implicitly differences the disturbances and it can be shown that the deterministic model between *y* and *u* is same as that between Δy and Δu. We will see how this happens in Chapter 7.

Detrending vs differencing

Differencing can also handle signals with deterministic trends. Two scenarios are shown below

First order/single degree differencing (handling linear trend)	Second order/second degree differencing (handling quadratic trend)
$y(k) = a + bk + e(k)$	$y(k) = a + bk + ck^2 + e(k)$
$\Rightarrow y(k) - y(k-1) = b + e(k)$	$\Rightarrow \underbrace{y(k) - y(k-1)}_{z(k)} = (b - c) + 2ck - e(k-1) + e(k)$
Stationary with mean *b*	$\Rightarrow z(k) - z(k-1) = 2c + e(k-2) - 2e(k-1) + e(k)$
	Stationary with mean 2c

In general, a polynomial trend of degree p can be handled through p^{th} order differencing.

Because differencing can handle trends and does not involve estimation of trend parameters, it is preferred over detrending. However, differencing makes signals noisier[26] and reduces the SNR values, and therefore can result in inaccurate models. Therefore, one should not go crazy with differencing! Over-differencing can do more harm than good. For process systems, you will rarely need to go beyond 2nd degree differencing.

Detrending and differencing are not the only transformations out there. Sometimes taking logarithms and then differencing can do the trick of making signals stationary. However, simple differencing works well with most of the datasets. With this, we complete our quick look at pre-treatment of dynamic signals.[27] Next, we will look at some guidelines around candidate model selection.

[26] Differencing is equivalent to high-pass filtering

[27] We have not covered outlier removal in this book. For dynamic dataset, outliers are handled via univariate outlier removal methods on individual signals or on residuals (obtained after model fitting). The first book of the series covers these methods in detail.

`

4.4 Model Structure Selection

Model structure selection refers to two subtasks:

a) Selection of the model type – for example, choosing between ARX, ARMAX, state-space, RNN models, etc.

b) Choosing the size of the model – for example, the number of terms/parameters in difference equation models, the order of the state-space model, the number of neurons in a neural network model.

While systematic procedures exist for the later subtask, the former is very subjective and relies heavily on the prior knowledge about the system. We had previously seen in Chapter 1 that a wrong selection of model type can lead to inaccurate model parameters. Therefore, it is good to be aware of some considerations that experienced process modelers make regarding model type selection.

Principle of parsimony

Principle of parsimony should be your guiding mantra during model structure selection. It states that you should not use any more parameters than necessary for adequate representation of your process. The simple rationale is that higher parameter dimensionality leads to greater chances of over-fitting and bad predictive ability, and higher computational complexity. This advice also translates to using the simplest model (and not necessarily the most generic model) first that meets the modeling purposes.

Figure below lists some aspects that you may consider during model structure selection. To keep our discussion simple, we will adopt a qualitative rather than quantitative approach. Hopefully, this discussion will help you appreciate why we have so many model types and make you pay more attention during model selection.

`

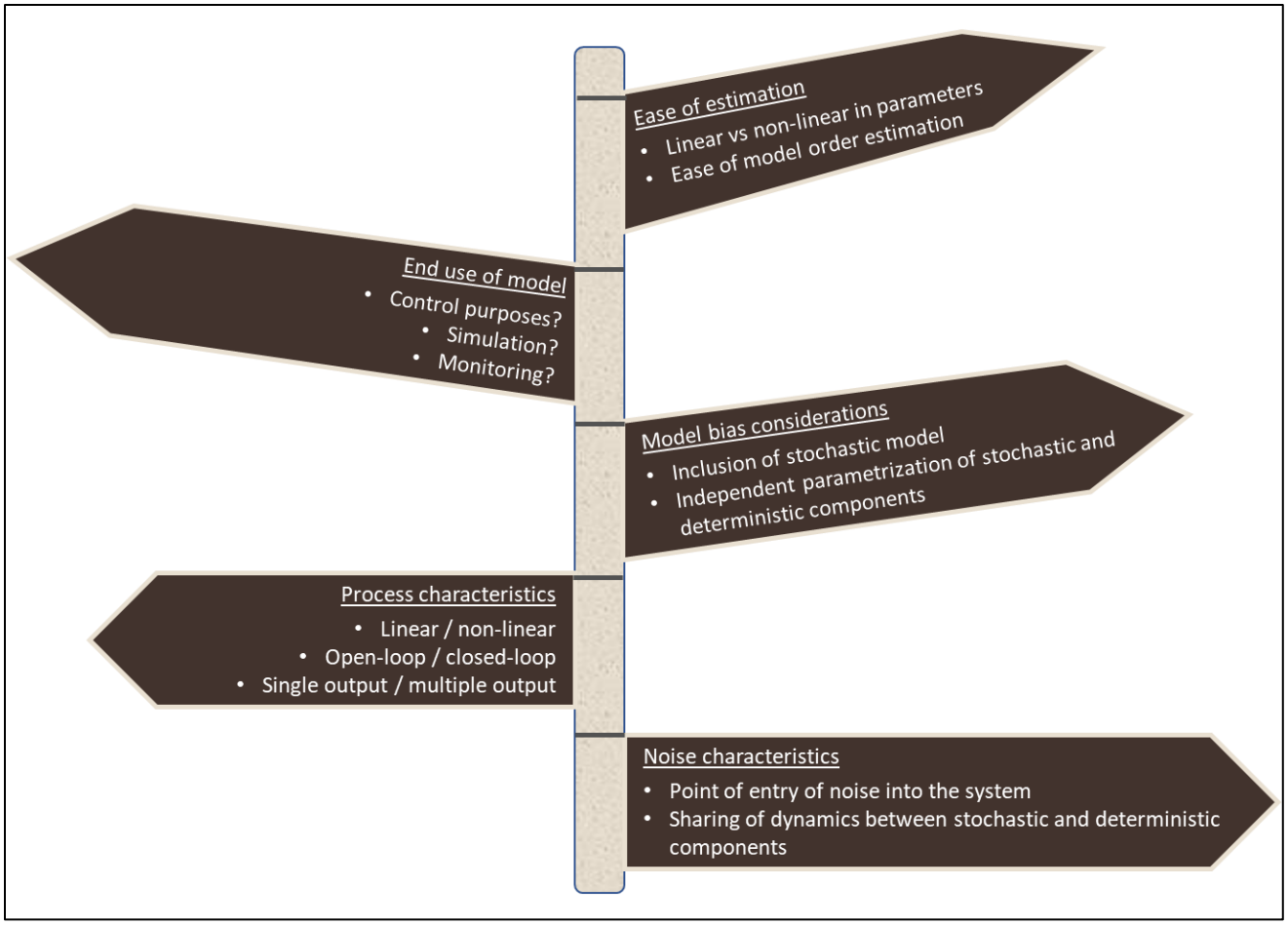


Figure 4.5: Considerations and insights that guide model structure selection[28]

Ease of estimation

Very often the choice of the model type is based on the ease with which a model can be quickly specified and fitted. For example, models like ARX and FIR are linear-in-parameters and therefore may be preferred over ARMAX or OE models which involve nonlinear optimization for parameter optimization. Difference equations like ARX and ARMAX need users to specify a number of model hyper-parameters such as the deadtime, the number of regressive terms, etc. Alternatively, state-space models do not require much *a priori* parameter specification.

In the upcoming chapters, you will learn the pros and cons of these different model types so that you can judiciously decide if the modeling convenience over-weighs the model's shortcomings.

[28] The meaning of independent and dependent parametrizations of stochastic and deterministic components will become clear in Chapter 6.

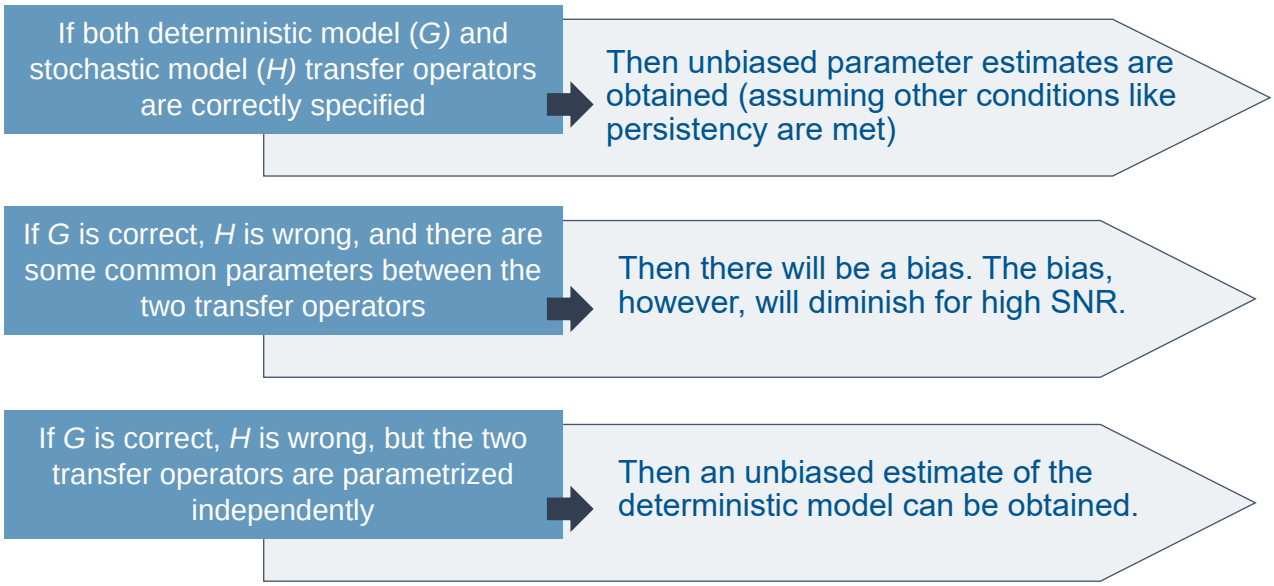
`

End use of model

We had previously alluded to data pre-treatment techniques being dictated by the end use of the model. However, the model's end use also dictates the model type selection. For example, if you are mainly interested in the deterministic portion of the process model (for say simulation purposes), then you are better-off with OE models rather than ARX models. If you are building a monitoring tool, then CVA models may be your preference. A good knowledge of a model's properties can help you decide if it suits your purpose or not.

Model bias considerations

As a process modeler, you should be aware of how your modeling choices influence the consistency/bias/accuracy of your model. An incorrect specification of stochastic component of the model can lead to biased parameter estimates of the deterministic component even if you have correctly specified the deterministic model (as seen before in Chapter 1). You should keep the following rules in mind (for open-loop and PEM models) to guide your modeling choice[29] [30].



The takeaway message for now is that paying attention to stochastic component of the model is not just some unnecessary exercise that can be ignored for convenience.

[29] Close-loop scenario is dealt specifically in Addendum A1.
[30] G and H transfer operators are introduced in Chapter 5.

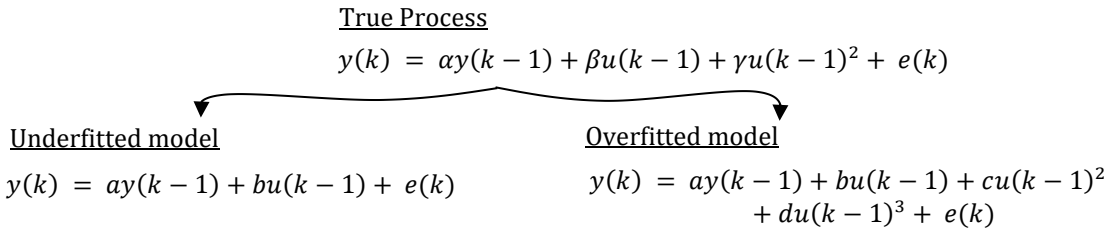
`

Model bias and variance

Errors during SysID is often characterized via two terms, viz, variance and bias. Variance refers to the variability in parameter estimates when model is fitted with different records of data. Variance occurs due to noise and limited records of data. Infact, theoretically, variance goes to zero for infinite amount data. You may have heard the statement “An overfitted model has high variance”: it essentially means that an overfitted model’s parameters may end up with significantly different values when trained with a different training dataset.

Bias refers to the systematic errors due to deficiencies in the model structure. The model cannot truly characterize the process and therefore, bias errors do not vanish even with large training dataset. The statement “An underfitted model has high bias” simply means that an underfitted model’s parameters are far away from their true values. In summary, while variance quantifies the precision of parameter estimates, bias quantifies the accuracy.

Consider the following scenario for illustration:



Process characteristics

Very often the process characteristics and your insights about the process will help in model type selection. For example, if you have reasons to believe that your process can be divided into distinct linear and nonlinear blocks, then block-structured nonlinear models may be preferred over NARX or ANNs. Furthermore, insights about the origin of nonlinearity can help you further decide between Hammerstein and Wiener models. If your process has multiple outputs, then state-space models may be preferred over difference-equation models for better parameter efficiency.

When you will be introduced to the different model structures later, special focus would be on highlighting specific properties of the models that make them more suitable to certain processes.

`

Noise characteristics

Our discussion till now must have impressed upon you the interplay between the stochastic and deterministic parts of a process model and the need to model process disturbances carefully. Insights on where exactly noise enters the system (at the input, middle, or output of the process) or whether the stochastic and deterministic parts share dynamics, etc. can help you describe the impact of noise on the plant better and therefore obtain more accurate models. If you don't have much prior information on the process noise, then don't worry; just make your best guess on the model type and then iterate using inferences from model quality checks.

As you work more with different model structures, you will naturally become better at model structure selection. You will also begin to appreciate why there is such a large collection of model types.

4.5 Model ID and Performance Assessment

Model ID entails estimation of parameters (denoted by vector θ) of the chosen model. This problem can formally be presented as follows: let the following input-output data be given

$$Y^N = \{y(k) = y(0), y(1), \dots, y(N-1)\}$$
$$U^N = \{u(k) = u(0), u(1), \dots, u(N-1)\}$$

and let $\hat{y}(k)$ be the sequence of predicted values and $\varepsilon(k)$ be the prediction error (also called residuals)

$$\varepsilon(k) = y(k) - \hat{y}(k) \qquad \text{eq. 1}$$

Parameters θ are estimated by imposing certain conditions on the error sequence. For example, PEM method tries to minimize the mean-squared values of $\varepsilon(k)$, i.e.,

$$\hat{\theta} = \min_{\theta} \frac{1}{N} \sum_{k} \varepsilon(k)^2$$

Mean-squared prediction errors are also used to measure the performance of the model on training data and validation/test data. Coefficient of determination (R^2) is another popular performance measure.

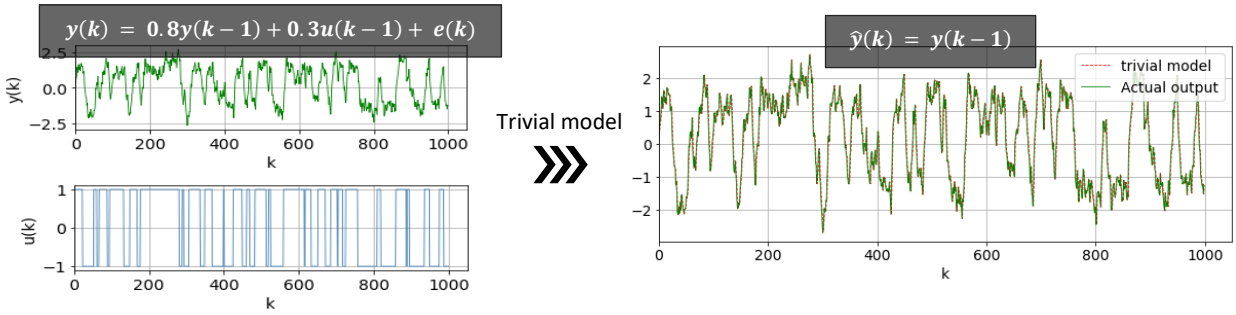
`

1-step ahead vs m-step ahead prediction

The error in Eq. 1 is formally defined as 1-step ahead error or $\varepsilon(k|k-1)$. This basically implies that all past measurements up to time *k-1* are utilized to generate predictions $\hat{y}(k|k-1)$ and then errors are computed. To see a potential problem with 1-step ahead errors, consider the following trivial model

$$\hat{y}(k) = y(k-1)$$ ⬅ Prediction = last output measurement

For a slow changing process (which most of the process systems are), the above trivial model can be deceptively impressive as shown below



***Conclusion**: Good match between plant observations and 1-step ahead predictions should not be the sole criteria of model adequacy check. Statistical analysis of errors is indispensable.*

An alternative to $\varepsilon(k|k-1)$ during parameter estimation and performance assessment is to use m-step ahead prediction errors, $\varepsilon(k|k-m)$, where output measurements only up to time *k-m* (i.e., *y(k-m), y(k-m-1), …*) are used and inputs up to time *k-1* are used as needed by the model. The model may use the predictions $\hat{y}$ recursively to arrive at the final $\hat{y}(k|k-m)$.

Analyzing the long-range or *m*-step ahead predictions is one of the components of model quality checks. As far as model fitting measure is concerned, 1-step ahead error is still the popular choice primarily due to the ease of computation and eventual minimization.

`

Model order selection

For most models, you need to specify the model size/order before you perform model ID. However, most likely, you would not know beforehand the correct size of your chosen model. The common recourse is to experiment with different model orders and select the one that minimizes the prediction errors. However, be careful that the prediction errors on the model fitting dataset will monotonically decrease as you increase the model order or the number of model parameters. The best practice, instead, is to use a fresh dataset, called validation dataset, to assess the model's performance on unseen data (not used during parameter estimation) as shown in Figure 4.6 below. This approach is called cross-validation.

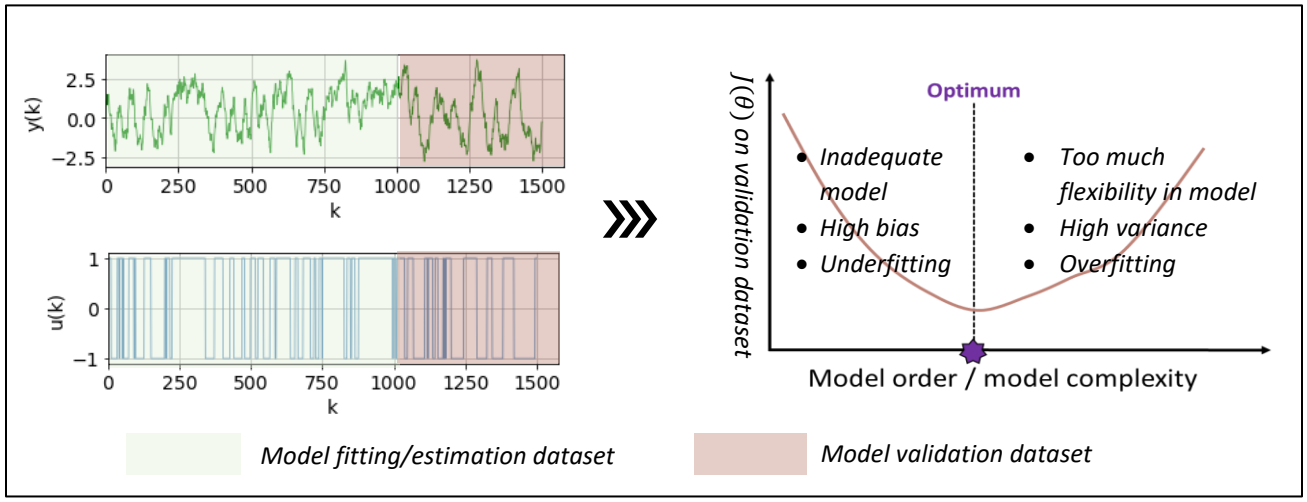


Figure 4.6: Cross-validation approach for dynamic modeling

You may notice that the samples in validation dataset are not obtained by shuffling and randomly choosing samples from the training dataset (as is done typically for static modeling). As should be obvious, this is done to maintain the chronological order of observation samples in fitting and validation datasets.

Another related approach is to carve out multiple sets of validation datasets from your original dataset and compute the performance measure as the average performance over these sets as shown in Figure 4.7. This is called *k*-fold cross-validation.

`

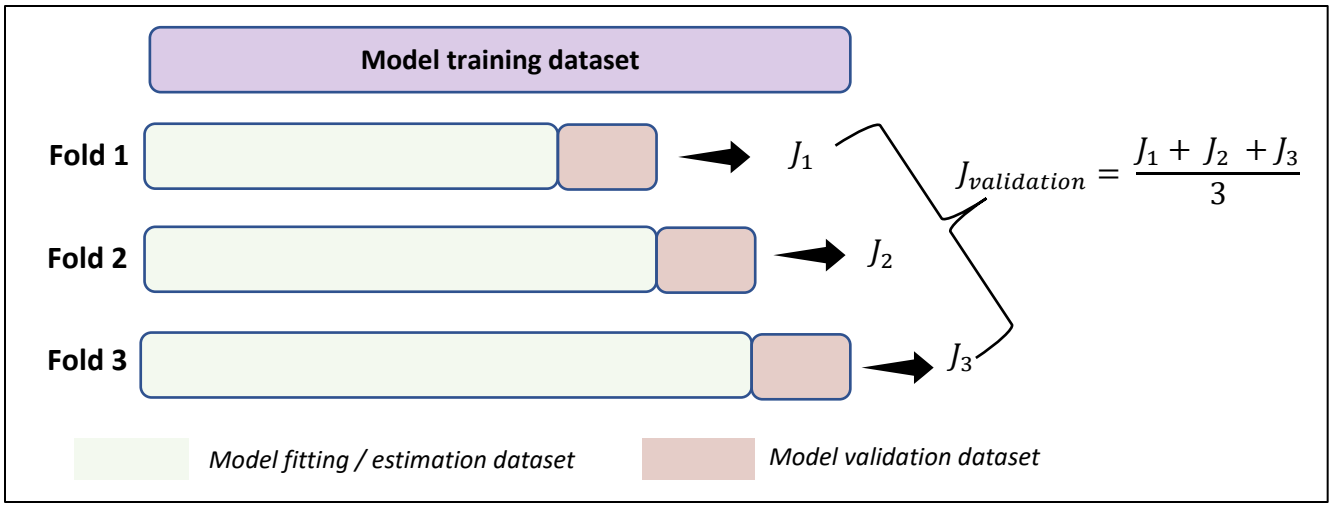


Figure 4.7: Illustration of 3-fold cross-validation procedure

Model order selection via information criteria

A popular alternative to cross-validation for model order selection is to use performance measure like Akaike information criteria (AIC) or Bayesian information criteria (BIC).

model performance model size

$$AIC(\theta) = N \ln\left[\frac{1}{N}\sum_{k}\varepsilon(k,\theta)^2\right] + 2n$$

$$BIC(\theta) = N\, ln\left[\frac{1}{N}\sum_{k}\varepsilon(k,\theta)^2\right] + n \ln(N)$$

Where, n = number of model parameters
N = number of observations in training dataset

As you can see, these criteria try to find a balance between model accuracy and model complexity. No separate validation dataset is used here, and the model order corresponding to the minimum AIC (or BIC) is selected. Most of the SysID packages provide native support for automated model order selection using these information criteria. You will see applications of AIC-based SysID in the later chapters.

`

4.6 Model Quality Check and Diagnostics

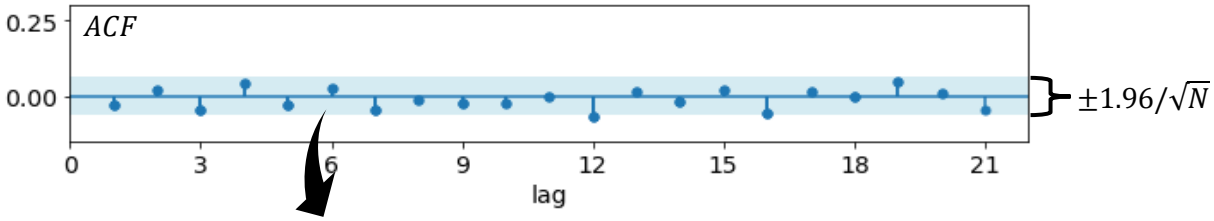
The discussion in the previous section has taught us that 'closeness' between predicted and measured outputs should not be construed as the sole definite proof of a 'good' model. Instead, the model needs to be further scrutinized carefully to ascertain that the model has truly picked up all the 'predictable' variations in the training dataset and adequately represents the process. If otherwise, you can look for clues for further improvement. Let's now look at some of the model quality check activities.

Residual analysis

Residuals are essentially the part of model fitting data that could not be explained by the model. Therefore, residual analysis entails checking that there remains no systematic variation in the residual sequence; if it does then the leftover systematic variations can further be modeled to improve the overall model predictions. The following two properties are desired for $\varepsilon(k)$

1) $\varepsilon(k)$ should be white noise, i.e., autocorrelation coefficients should be close to zero[31]
2) $\varepsilon(k)$ should be uncorrelated to input *u(k)*[32]

For condition (1)[33], ACF plot is generated (with about 20 lags). The correlation coefficients (except for lag zero) are expected to be within some threshold - usually corresponding to 95% significance level which is given by $\pm 1.96/\sqrt{N}$. If condition (1) is violated, then it is implied that the model could be improved further; however, no specific clues are provided about the specific recourse for improvement. A generic advice is to try more complex stochastic model for the disturbance signal.

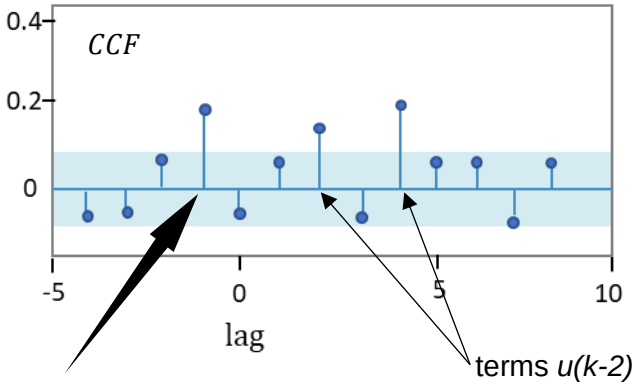


No more than 5% of correlations can be outside this band for condition (1) to be considered met

[31] If the disturbance signals are not modeled (such as for OE models), then this condition is not checked.
[32] Either of ε(k) or *u(k)* should be white for this condition.
[33] Quantitative statistical tests (such as Ljung-Box test) also exist to check if the residuals indicate a lack of fit

`

For condition (2), CCF plot is generated along with thresholds (again, $\pm 1.96/\sqrt{N}$ for 95% confidence interval). If greater than 5% correlations (for lags > 0) violate the thresholds, then it implies that there still exist some variations in $\varepsilon(k)$ that stem from *u(k)* and therefore the deterministic part of the model can further be improved. A hypothetical CCF plot below show some examples of inferences drawn when condition (2) is not met[34].



Significant correlations at negative lags imply process feedback or that residuals are correlated to the future inputs. This is an indication that data was collected from a closed-loop process where output noise influences future process inputs. [More on closed-loop ID in Addendum A1]

terms *u(k-2)* and *u(k-4)* are missing from the model

Residual analysis is often performed on the fitting dataset itself. It can be done on a fresh validation dataset as well; however it will be more demanding for the model.

Transient response checks

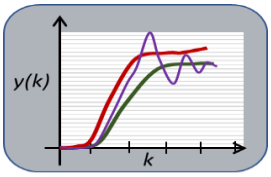
It is a good practice to check the predicted step response or impulse response[35] after model fitting. If you have some prior insights about the process gain and/or time constant, then you can compare your prior beliefs against what you observe in the transient response plots. Mentioned below are some other scenarios where these plots can come in handy:

[34] The rules of thumb: sharp peaks indicate missing input terms; slow-varying and consistent correlations outside the confidence region indicate missing output terms in the model.

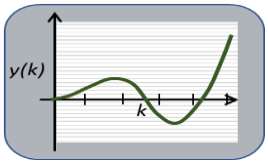
[35] Step and impulse response of a process are introduced in Chapter 6.

`

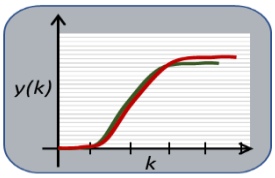
- Assume you have several candidate models who provide similar prediction responses. It is possible that the step responses from these models end up being very different. This would indicate that some of the models are obviously inaccurate and can help you reject them. Process insights, if available, can be used to select the model(s) whose transient response(s) seem more reasonable.



- Your model may give good predictive responses, but the step response may end up showing unbounded steady-state values! If you know that your process is inherently stable, then this implies wrong model.



- A good agreement between transient responses from a parametric model and a non-parametric model can build confidence that essential features of the process have been picked up by the model



Simulation response checks

Previously, we talked about *m*-step ahead predictions being preferable over the conventional 1-step ahead predictions for model adequacy analysis. This idea can be taken to the extreme, resulting in infinite-step ahead predictions – also called as simulation. Simulation is often performed on fresh data set and the output at any time *k* is generated using only the past input measurements and initial conditions of the output. If model includes past output terms, then simulated outputs are used recursively. Illustration below clarifies this concept.

Model: $y(k) = 0.8y(k-1) + 0.6u(k-1) + e(k)$
Initial condition: $y(0) = 0$
To predicted: $\hat{y}_{SIM}(3)$

$$\hat{y}_{SIM}(1) = 0.8y(0) + 0.6u(0)$$

$$\hat{y}_{SIM}(2) = 0.8\hat{y}_{SIM}(1) + 0.6u(1)$$

$$\hat{y}_{SIM}(3) = 0.8\hat{y}_{SIM}(2) + 0.6u(2)$$

Disturbance contributions to output are ignored in generating $\hat{y}_{SIM}$.[36] The utility of $\hat{y}_{SIM}$ is that if SNR is high and $\hat{y}_{SIM}(k)$ matches closely with *y(k)*, then a good (deterministic) model is implied. Do note that poor simulation performance may imply high disturbance levels and not

[36] For models where disturbance signal is not modeled (such as OE, FIR), simulation is same as predictions

`

necessarily bad (deterministic) model. Nonetheless, if simulation does not catch the major systematic variations in *y(k)*, then the model is probably not good.

Example 4.4:

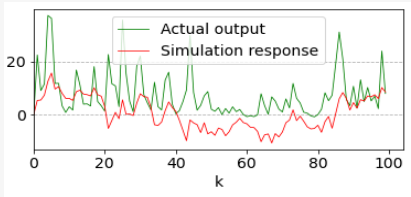
Consider the following actual process and candidate model. We will generate some output data from the process and then analyze the simulated and 1-step ahead predictions from the model.

Process: $y(k) = 0.3y(k-1) + 0.8u(k-1) + 0.2u^2(k-1) + e(k)$

Model: $y(k) = 0.8y(k-1) + 0.6u(k-1) + e(k)$

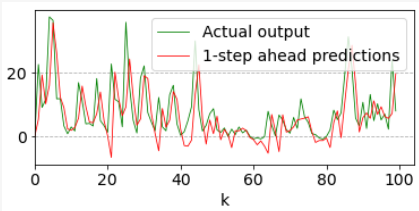
Initial condition: $y(0) = 0$

```
# import packages
import numpy as np, matplotlib.pyplot as plt

 # define input signal (for illustration, let this be white noise)
N = 100
u = np.random.normal(0, 5, N)

# compute actual process output
y = np.zeros((N, ))
e = np.random.normal(0, 0.1, N) # measurement noise
for k in range(1, N):
   y[k] = 0.3*y[k-1] + 0.8*u[k-1] + 0.2*u[k-1]*u[k-1] + e[k]

# compute model's simulated output and compare
y_sim = np.zeros((N, ))
for k in range(1, N):
   y_sim[k] = 0.8*y_sim[k-1] + 0.6*u[k-1]

plt.plot(y, 'g', linewidth=0.8, label='Actual output')
plt.plot(y_sim, 'r', linewidth=0.8, label='1-step ahead predictions')
plt.ylabel('y simulated'), plt.xlabel('k'), plt.xlim(0) , plt.legend()
```



(expectedly, very bad model performance)

```
# compute model's 1-step ahead predictions and compare
y_pred = np.zeros((N, ))
for k in range(1, N):
   y_pred[k] = 0.8*y[k-1] + 0.6*u[k-1]

plt.plot(y, 'g', linewidth=0.8, label='Actual output')
plt.plot(y_pred, 'r', linewidth=0.8, label='1-step ahead predictions')
plt.ylabel('y predictions'), plt.xlabel('k'), plt.xlim(0)
```



(deceptively) pretty good model performance!)

`

Simulation can sometimes throw surprising results as a result of model inaccuracy. For example, simulated outputs may show divergent trend while 1-step ahead predictions may be perfect! Furthermore, predicted outputs from two models may be very similar, but the simulated responses can be significantly different.

Parameter error checks

Post model ID, the value of the parameetrs should be checked against their estimation error. Ideally, the errors should be insignificant relative to the estimated parameter values. In practice, a 95% confidence interval is generated for each parameter and if the confidence interval includes 0, then that parameter is removed from the model. A new model excluding the removed variables is re-identified.

This completes our quick look at the model diagnostics. There are a few other checks that we did not cover such as pole-zero cancellation, stationarity checks using poles, invertibility checks using zeros, etc. Overall, this chapter has equipped you with several tools to help you along the SysID journey. We hope that you already feel comfortable that you have all the necessary weapons in your arsenal to handle each step of the SysID process. We will end this chapter with two advices. First, don't forget that SysID is an iterative process and therefore don't get disheartened if your model fails in first attempt. Second, remember the famous quote[37] we alluded to in Chapter 1, "*All models are wrong, but some are useful!*"

Summary

In this chapter we learnt the best practices of setting up a system identification workflow. We familiarized ourselves with practice-relevant concepts around identification test design, data pre-treatment, model structure and order selection, and model quality checks. Next, we will begin learning about arguably the most important and interesting task of SysID, i.e., model identification.

[37] Attributed to the famous statistician George E. P. Box.

`

Part 2

Classical Machine Learning Methods for Dynamic Modeling

`

Chapter 5

Time Series Analysis: Concepts and Applications

As alluded to before, time series analysis (TSA) refers to study of time series signals from processes without exogenous inputs. Before we dive into the relatively more complex world of input-output modeling, it would be prudent to first get acquainted with how to model stochastic variations in signals and compactly describe the dependencies among adjacent observations in a time series. The study of time series is not just for pedagogical convenience. SysID derives many concepts from time series analysis and therefore, a strong foundation in modeling time series is important for mastering the art of SysID.

Time series analysis can help us to characterize I/O model residuals and get clues regarding further model refinement. If your process has measured disturbance signals (inputs that can't be manipulated; for example, ambient temperature), then they can be modeled via TSA to provide better forecasts and control. Furthermore, TSA can be used to model controlled process variables and build process monitoring tools. In this chapter, we will work through case-studies illustrating these applications.

Apart from setting a strong foundation of digital signal processing, this chapter will also introduce the backshift notation and transfer function operator. These constructs are quite useful for compact description and algebraic manipulation of the difference equations. Specifically, the following topics are covered

- Introduction to AR, MA, and ARMA models for stationary signals
- Using ACF and PACF for model structure selection
- Monitoring controlled process variable in a CSTR using ARMA models
- Introduction to ARIMA models for nonstationary signals
- Forecasting measured signals using ARIMA models

`

5.1 Time Series Analysis: An Introduction

Time series analysis refers to the study and modeling of dependencies among sequential data points in time series signals. Several approaches exist for modeling time series. In this chapter, we will look at four of the most common models as shown in Figure 5.1. While AR, MA, and ARMA models are used to describe stationary signals, ARIMA is used for (homogeneous) nonstationary signals. In this chapter, we will understand the differences, similarities, and the inherent connections between these models.

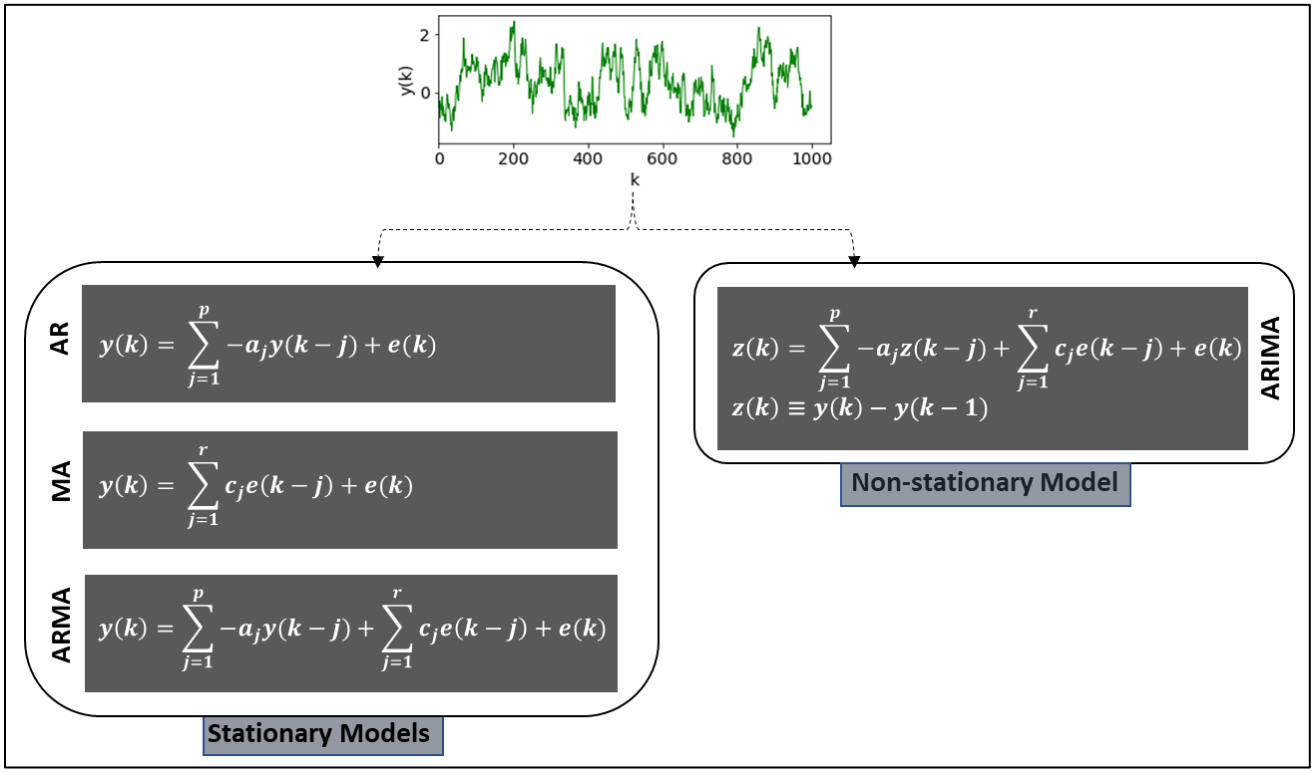
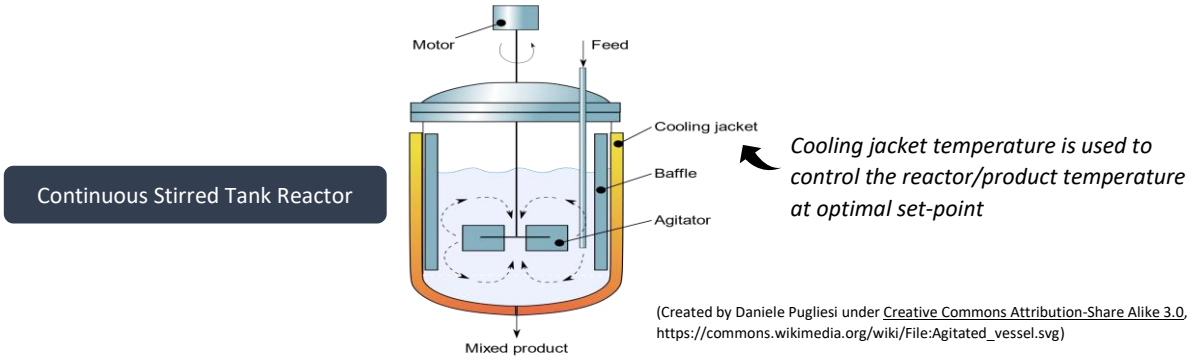


Figure 5.1: Commonly employed univariate time-series models

To further motivate the study of time series analysis, let's briefly look at two of the use cases that were mentioned previously. The illustration below shows a CSTR unit that has measured disturbance signals (feed concentration and temperature). An accurate model for these signals can help generate accurate forecasts and control the vessel temperature more effectively. Also, it is easy to show that if the setpoint of the controllers do not change, then the controlled variables (here, output product temperature and concentration) depend on process disturbances alone. Therefore, a TSA model for these variables can be built to eventually look for significant mismatch between measurements and model predictions as an indicator of process faults.

`



5.2 Autoregressive (AR) Models: An Introduction

AR models are among the simplest and easiest to understand time series models, wherein the current state is written as a linear combination of past measurements as shown below[38]

$$y(k) = -a_1 y(k-1) - a_2 y(k-2) \cdots - a_p y(k-p) + e(k) \quad \text{eq. 1}$$
$$= \sum_{j=1}^{p} -a_j y(k-j) + e(k)$$

The above model uses *p* lagged/past values and therefore is referred to as an AR(p) model or an autoregressive model of order *p*. In spite of its simplicity, an AR model can generate varied patterns of signals as shown below. In process industry, processes with inherent autoregressive behavior are fairly common.

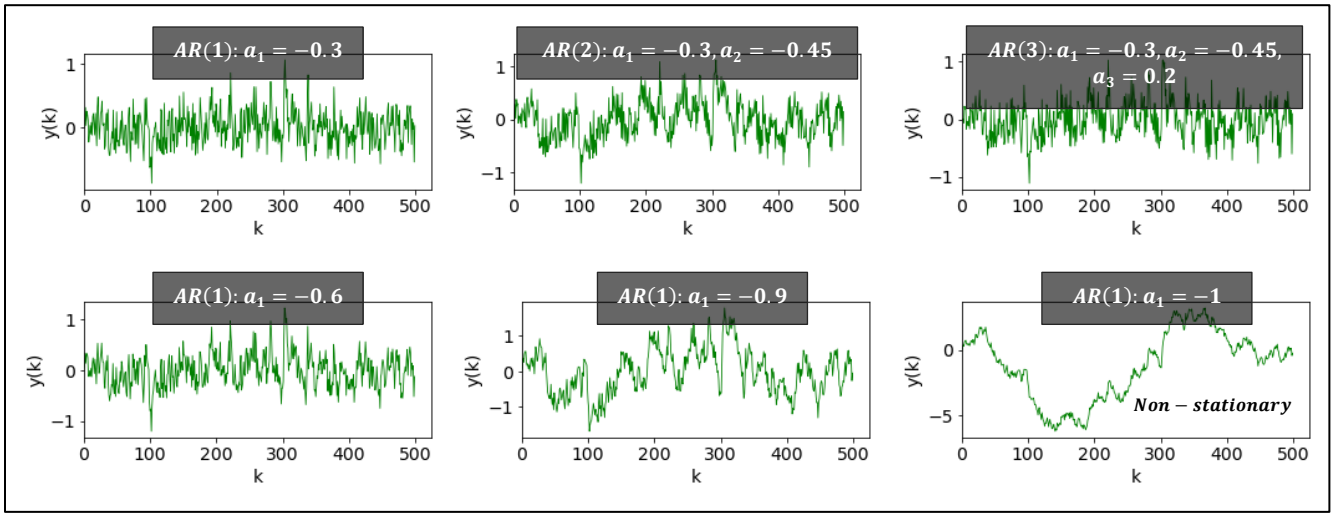


Figure 5.2: Time-series signals from AR processes

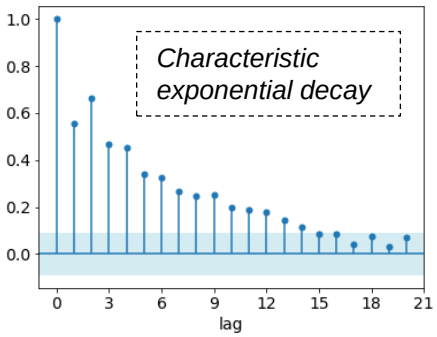
[38] You will see soon why we have used '$-a_j$' and not 'a_j' in the equation

`

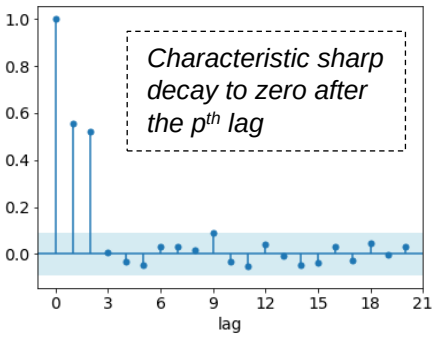
Model order selection

Given any time series data, the suitability of an AR model and its potential order can be judged using the ACF[39] and PACF plots as shown below for the AR(2) plot in Figure 5.2.

ACF plot



PACF

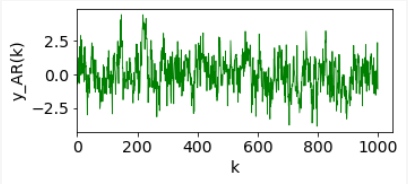


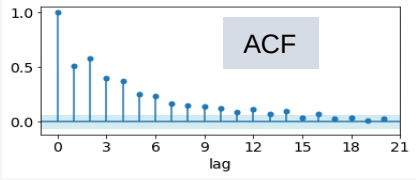
Let's see an example which shows how to generate data from an AR process and how to fit an AR model.

Example 5.1:

We will create an AR(2) signal with 1000 samples. Thereafter, guided by ACF/PACF, a model will be fitted to the data followed by residual diagnostics.

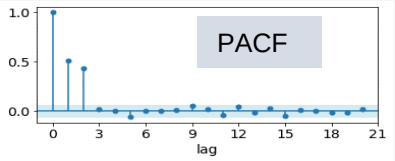
```
# import packages
import numpy as np, matplotlib.pyplot as plt
from statsmodels.tsa.arima_process import ArmaProcess
from statsmodels.tsa.arima.model import ARIMA
from statsmodels.graphics.tsaplots import plot_acf, plot_pacf

# generate data for AR(2) process using statsmodels.tsa package
ar_coeffs = np.array([1, -0.3, -0.45]) # a1 = -0.3, a2 = -0.45
ARprocess = ArmaProcess(ar_coeffs)
y_AR = ARprocess.generate_sample(nsample=1000)

# generate ACF and PACF plots for y_AR
conf_int = 2/np.sqrt(len(y_AR))
plot_acf(y_AR, lags= 20, alpha=None, title='')
plt.gca().axhspan(-conf_int, conf_int)
```





[39] In general, the ACF plot for an AR process may also show damped sine waves in addition to damped exponentials

```
plot_pacf(y_AR, lags= 20, alpha=None, title='')
plt.gca().axhspan(-conf_int, conf_int)
```



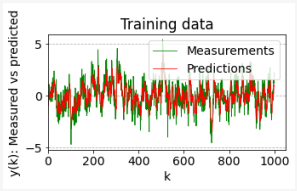
The PACF plot clearly indicates an AR(2) process. We will again use the *statsmodels.tsa* package to fit an AR(2) model.

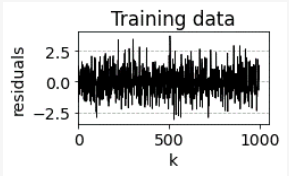
```
# Fit an AR(2) model
y_AR_centered = y_AR - np.mean(y_AR)
model = ARIMA(y_AR_centered, order=(2, 0, 0)) # order = (p,d,r)
results = model.fit()

# Print out the estimate for the parameters a1 and a2
print('[a1, a2] = ', -results.arparams)

>>> [a1, a2] =  [-0.27 -0.50]
```

AR model is linear-in-parameters and therefore, OLS method can be used to estimate the unknown model parameters.

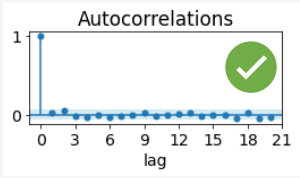
The estimated parameters are close to the true values (the inaccuracy in the estimates diminishes for larger sample size). With the obtained model, let's generate the model residuals and confirm whiteness of the residual sequence.

```
# get model predictions and residuals on training dataset
y_AR_centered_pred = results.predict()
residuals = y_AR_centered - y_AR_centered_pred

plt.figure(), plt.title('Training data')
plt.plot(y_AR_centered, 'g', label='Measurements')
plt.plot(y_AR_centered_pred, 'r', label='Predictions')
plt.ylabel('y(k): Measured vs predicted'), plt.xlabel('k'), plt.legend()

plt.figure(), plt.plot(residuals, 'black', linewidth=0.8)
plt.title('Training data'), plt.ylabel('residuals'), plt.xlabel('k')
```






```
# ACF residuals
plot_acf(residuals, lags= 20, alpha=None, title='')
plt.gca().axhspan(-conf_int, conf_int, facecolor='lightblue', alpha=0.5)
plt.xlabel('lag'), plt.title('Autocorrelations')
```



`

Shift operator and transfer function operator

The original form of the difference equations that we have worked with so far is not very convenient when combining different models and analyzing model properties. For a more compact and useful representation, the concept of backshift operator (q^{-1}) is introduced. It is defined as

$$y(k-1) = q^{-1}y(k)$$

Evidently, the backshift operator moves the signal one step back. Multiple applications shift the signal back further

$$q^{-1}\left(q^{-1}y(k)\right) = q^{-2}y(k) = y(k-2)$$

Let's rewrite Eq. 1 using the shift operator as follows

$$\begin{aligned} & y(k) + \sum_{j=1}^{p} a_j y(k-j) = e(k) \\ \Rightarrow\ & y(k) + \left(\sum_{j=1}^{p} a_j\, q^{-j}\right) y(k) = e(k) \\ \Rightarrow\ & \left(1 + \sum_{j=1}^{p} a_j\, q^{-j}\right) y(k) = e(k) \\ \Rightarrow\ & y(k) = \frac{1}{\left(1+\sum_{j=1}^{p} a_j q^{-j}\right)}\, e(k) \equiv H(q^{-1})e(k) \end{aligned}$$

Let's see how we would represent a simple ARX model $y(k) + a_1 y(k-1) + a_2 y(k-2) = b_1 u(k-1) + b_2 u(k-2) + e(k)$ using the q notation

$$(1 + a_1 q^{-1} + a_2 q^{-2}) y(k) = q^{-1}(b_1 + b_2 q^{-1}) u(k) + e(k)$$

$$\Rightarrow y(k) = G(q^{-1})u(k) + H(q^{-1})e(k) \qquad \text{eq. 2}$$

$$G(q^{-1}) = \frac{q^{-1}(b_1 + b_2 q^{-1})}{1 + a_1 q^{-1} + a_2 q^{-2}} \qquad H(q^{-1}) = \frac{1}{1 + a_1 q^{-1} + a_2 q^{-2}}$$

Note that $G(q^{-1})$ and $H(q^{-1})$ are (input and noise transfer) operators (and not multipliers) that operate on $u(k)$ and $e(k)$, respectively, and determine how the incoming signals are 'transferred' to the output. Therefore, $G(q^{-1})$ and $H(q^{-1})$ are called transfer operators. Equation 2 can be written further as

$$y(k) = y_u(k) + v(k)$$

deterministic output

disturbance signal

`

where, $y_u(k) = G(q^{-1})u(k)$ eq. 3

$v(k) = H(q^{-1})e(k)$

Note that Eq. 3 clearly shows that in an ARX model, the process disturbance *v(k)* is a colored signal and not a white noise (*e(k)*). It is evident, therefore, that the transfer operator form makes it easy to dissect the stochastic and deterministic parts of the model.

The *q* notation may not seem appealing to you immediately, but soon enough you will begin to appreciate the convenience it offers! For now, just note that $G(q^{-1})$ and $H(q^{-1})$ can be manipulated like polynomials of q^{-1} as you will soon see.

5.3 Moving Average (MA) Models: An Introduction

In MA models the current state is a summation of current and past white noise error values as shown below

$$y(k) = c_1 e(k-1) + c_2 e(k-2) \cdots + c_r e(k-r) + e(k) \quad \text{eq. 4}$$

$$= \sum_{j=1}^{r} c_j e(k-j) + e(k)$$

$$= \left(1 + \sum_{j=1}^{r} c_j q^{-j}\right) e(k)$$

The above model is referred to as an MA(r) model. An MA model can be interpreted as a counterpart of FIR models for time series signals where the effect of past random shocks/noise are propagated directly to future values of the time series. Plots below provide a sample of the patterns generated from MA models.

`

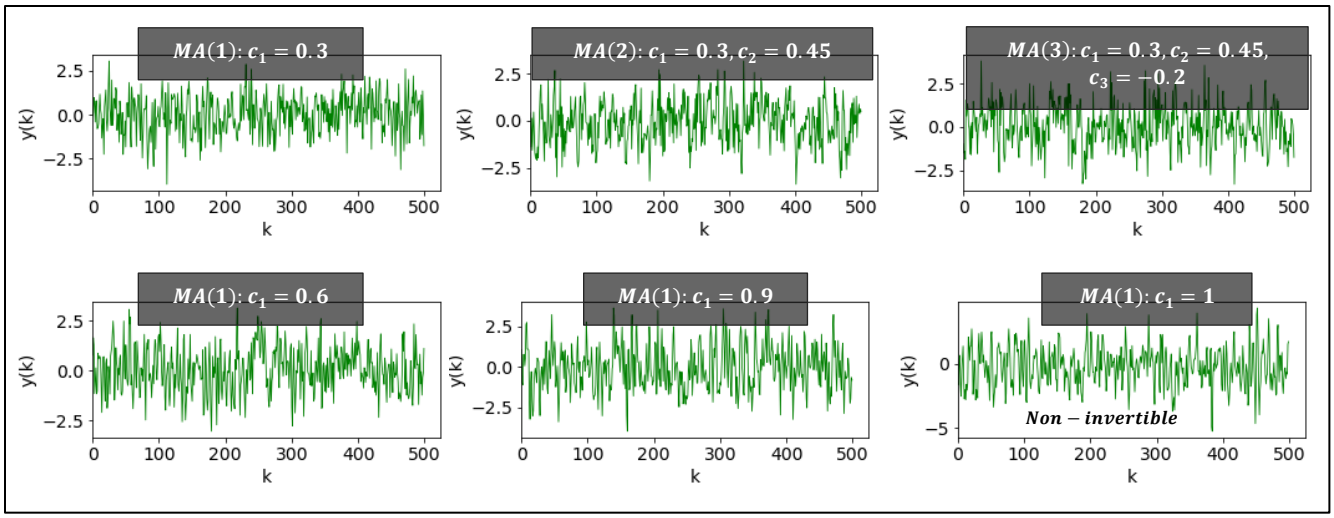
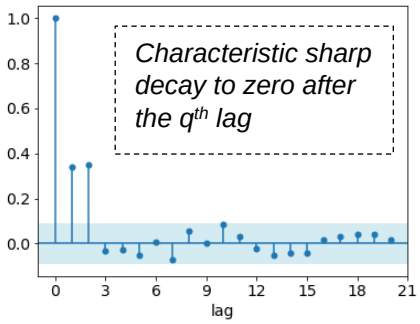


Figure 5.3: Time-series signals from MA processes

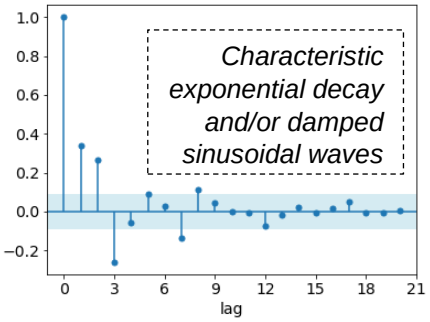
Model order selection

Just like for AR model, ACF and PACF plots provide direct clues about the order of the model as shown below for the MA(2) plot in Figure 5.3.

ACF plot



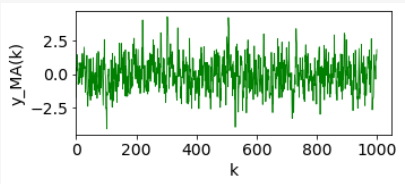
PACF



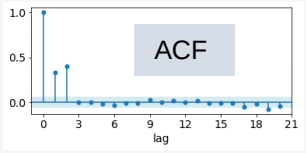
Let's concretize our understanding of the MA process through an example.

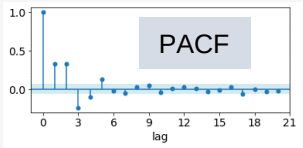
Example 5.2: We will create an MA(2) signal with 1000 samples. Thereafter, guided by ACF/PACF, a model will be fitted to the data followed by residual diagnostics. The imported packages are the same as that in Example 5.1

```
# generate data for MA(2) process using statsmodels.tsa package
ma_coeffs = np.array([1, 0.3, 0.45]) # [1, c1, c2]
MAprocess = ArmaProcess(ma = ma_coeffs)
y_MA = MAprocess.generate_sample(nsample=1000)
```



```
# generate ACF and PACF plots for y_MA
conf_int = 2/np.sqrt(len(y_MA))
plot_acf(y_MA, lags= 20, alpha=None, title='')
plt.gca().axhspan(-conf_int, conf_int)

plot_pacf(y_MA, lags= 20, alpha=None, title='')
plt.gca().axhspan(-conf_int, conf_int)
```





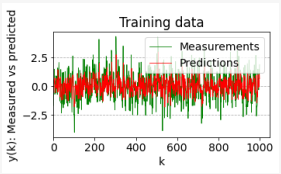
The ACF plot clearly indicates an MA(2) process. We will again use the *statsmodels.tsa* package to fit an MA(2) model.

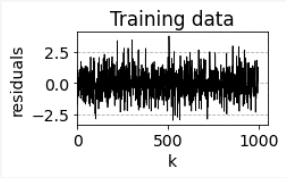
```
# Fit an MA(2) model
y_MA_centered = y_MA - np.mean(y_MA)
model = ARIMA(y_MA_centered, order=(0, 0, 2)) # order = (p,d,r)
results = model.fit()

# Print out the estimate for the parameters c1 and c2
print('[c1, c2] = ', results.maparams)

>>> [c1, c2] =  [0.30 0.53 ]
```

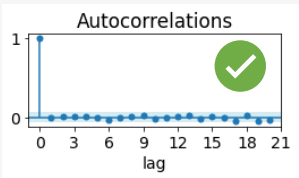
Estimation of MA parameters is not as easy as that for AR parameters because the unknowns, past shocks (*e*) and coefficients (*c*), are non-linearly present in the model. Therefore, iterative numerical procedure is employed for parameter estimation.

The estimated parameters are close to the true values (the inaccuracy in the estimates diminishes for larger sample size). With the obtained model, let's generate the model residuals and confirm whiteness of the residual sequence.

```
# get model predictions and residuals on training dataset
y_MA_centered_pred = results.predict()
residuals = y_MA_centered - y_MA_centered_pred

plt.figure(), plt.title('Training data')
plt.plot(y_MA_centered, 'g', label='Measurements')
plt.plot(y_MA_centered_pred, 'r', label='Predictions')
plt.ylabel('y(k): Measured vs predicted'), plt.xlabel('k'), plt.legend()

plt.figure(), plt.plot(residuals, 'black', linewidth=0.8)
plt.title('Training data'), plt.ylabel('residuals'), plt.xlabel('k')

# ACF residuals
plot_acf(residuals, lags= 20, alpha=None, title='')
plt.gca().axhspan(-conf_int, conf_int, facecolor='lightblue', alpha=0.5)
plt.xlabel('lag'), plt.title('Autocorrelations')
```







`

Equivalence between AR and MA models

AR and MA models may appear to be very different, but, under some conditions, they are inter-convertible. Any stationary AR model can be rewritten as an MA(∞) model and any invertible (definition of invertibility will be provided soon) MA model can be equivalently rewritten as an AR(∞) model. For example, consider the MA(1) model $y(k) = c_1 e(k-1) + e(k)$, which can be rewritten as

$$\begin{aligned} y(k) &= c_1(y(k-1) - c_1 e(k-2)) + e(k) \\ &= c_1 y(k-1) - c_1^2 e(k-2) + e(k) \end{aligned}$$

$\vdots$ after repeated substitutions

AR(∞) model! $$y(k) = -\sum_{j=1}^{\infty} (-c_1)^j \, y(k-j) + e(k)$$ eq. 5

An alternative and quicker derivation:

$$y(k) = (1 + c_1 q^{-1}) e(k)$$
$$\Rightarrow (1 + c_1 q^{-1})^{-1} y(k) = e(k)$$
$$\Rightarrow \left(\sum_{j=0}^{\infty} (-c_1)^j q^{-j}\right) y(k) = e(k)$$

using polynomial long division

Note that Eq. 5 represents a valid/sensible model only if $|c_1| < 1$ (otherwise c_1^j grows unbounded). This is also the condition of invertibility for a MA(1) model. An AR(1) model can be rewritten as MA(∞) model via recursive substitution of *y(k-1)*, *y(k-2)*, and so on.

ACF/PACF plots may sometime suggest both AR and MA models as adequate representations of the underlying process. Given the equivalence between the two class of models, you can choose either. Just keep a note that while AR parameters are easy to estimate, MA model may be more parsimonious (i.e., require much fewer parameters).

`

5.4 Autoregressive Moving Average (ARMA) Models: An Introduction

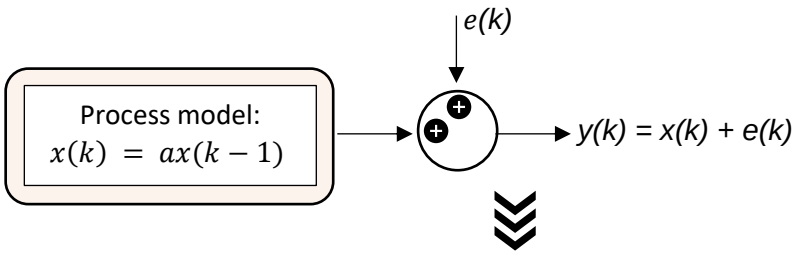
Sometimes a more compact and parsimonious model (compared to AR or MA models) can be built by combining the autoregressive and moving average terms. Such models are called ARMA models and take the following form

$$y(k) = -a_1 y(k-1) \cdots - a_p y(k-p) + c_1 e(k-1) \cdots + c_r e(k-r) + e(k)$$

$$= \underbrace{\left(-\sum_{j=1}^{p} a_j q^{-j}\right) y(k)}_{\text{AR terms}} + \underbrace{\left(\sum_{j=1}^{r} c_j q^{-j}\right) e(k)}_{\text{MA terms}} + e(k) \qquad \text{eq. 6}$$

The above model is called ARMA(p,r) due to the p^{th} order autoregressive component and r^{th} order moving average component. It is apparent that AR and MA models are special cases of ARMA model. Additionally, any stationary and invertible ARMA model can equivalently be rewritten an AR(∞) or a MA(∞) model.

ARMA models are not just for mathematical convenience but are commonly encountered in process industry. To see the reason, consider an example scenario below where output from a pure AR process is contaminated by a random white noise disturbance



$$y(k) = a\big(y(k-1) - e(k-1)\big) + e(k)$$

$$\Rightarrow y(k) = ay(k-1) - ae(k-1) + e(k)$$

Overall, an ARMA(1,1) process!

`

Plots below provide a sample of patterns generated from ARMA models.

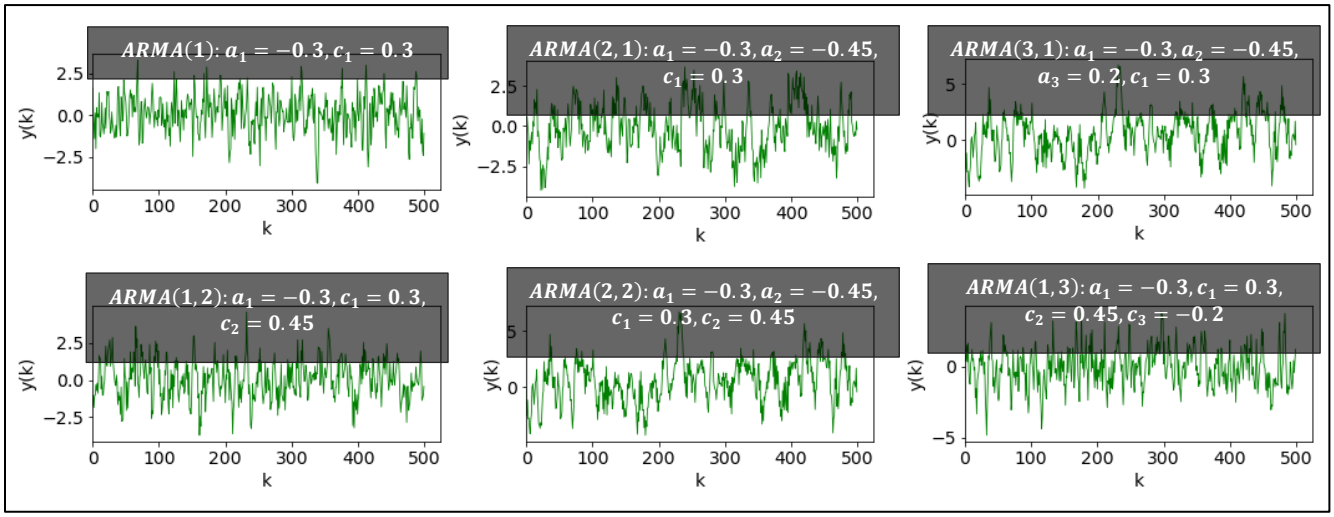
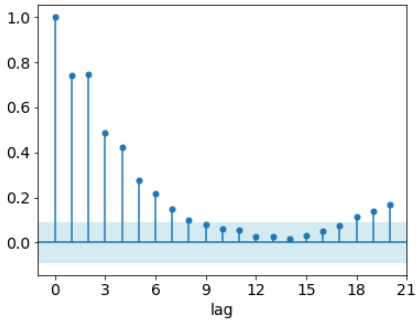


Figure 5.4: Time-series signals from ARMA processes

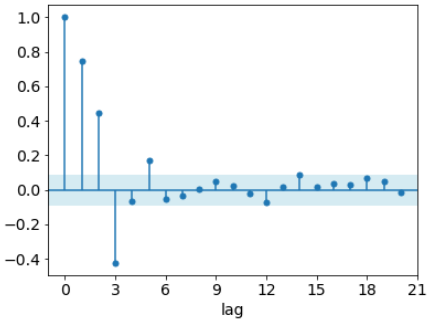
Model order selection

ACF and PACF plots for the ARMA(2,2) plot in Figure 5.5 are provided below.

ACF plot



PACF

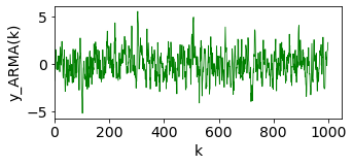


Both ACF and PACF plots tails off in exponential-like pattern (with mixture of damped sine waves) which is characteristic of ARMA process

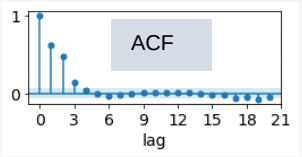
As is apparent, not much help is provided regarding the specific values of *p* and *r*. The general recourse is to use model order selection techniques introduced in Chapter 4.

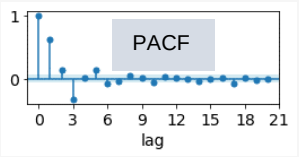
Example 5.3: We will create an ARMA(1,2) signal with 1000 samples. Thereafter, the AR and MA orders will be automatically determined via AIC and the corresponding model will be fitted to the data. The imported packages are the same as that in Example 5.1

```
# generate data for ARMA(1,2) process
ARMAprocess = ArmaProcess(ar=[1, -0.3], ma=[1, 0.3, 0.45])
y_ARMA = ARMAprocess.generate_sample(nsample=1000)
```




```
# generate ACF and PACF plots for y_ARMA
plot_acf(y_ARMA, lags= 20, alpha=None, title='')
plot_pacf(y_ARMA, lags= 20, alpha=None, title='')
```





The ACF and PACF plots are not very conclusive. Fortunately, the *tsa* package provides a function arma_order_select_ic that can help estimate the orders via AIC.

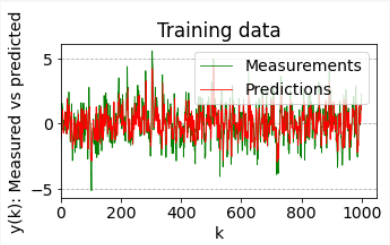
```
# Determine the optimal AR and MA orders
from statsmodels.tsa.stattools import arma_order_select_ic

y_ARMA_centered = y_ARMA - np.mean(y_ARMA)
res = arma_order_select_ic(y_ARMA_centered, max_ma=4, ic=["aic"])
p, r = res.aic_min_order
print('(p, r) = ', res.aic_min_order)
```

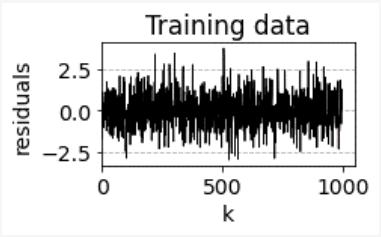
```
>>> (p, r) =  (1, 2)
```

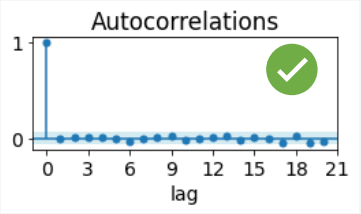
Orders correctly identified!

```
# Fit an ARMA(p,r) model
model = ARIMA(y_ARMA_centered, order=(p, 0, r))
results = model.fit()
print('[a1] = ', -results.arparams); print('[c1, c2] = ', results.maparams)

>>> [a1] =  [-0.29]
[c1, c2] =  [0.30 0.53]
```

The estimated parameters are close to the true values. Next, the residuals are generated as in the previous examples and whiteness of the residual sequence is confirmed.







`

5.5 Monitoring Controlled Variables in a CSTR using ARMA Models

To illustrate a practical application of time series analysis, we will consider the continuous stirred-tank reactor described in the chapter introduction. Here, the product temperature will be monitored to ensure it is not deviating away from its optimal setpoint due to any process fault. The datafile *CSTR_controlledTemperature.csv* contains 4 hours of temperature measurements[40] sampled at 0.1 seconds (totaling 2401 samples). A process fault occurs at 3.5 hours (around sample # 2100) causing a 20% decrease in the heat transfer coefficient. Let's see if this process fault causes any deviations in the product temperature and if our monitoring methodology can detect these deviations.

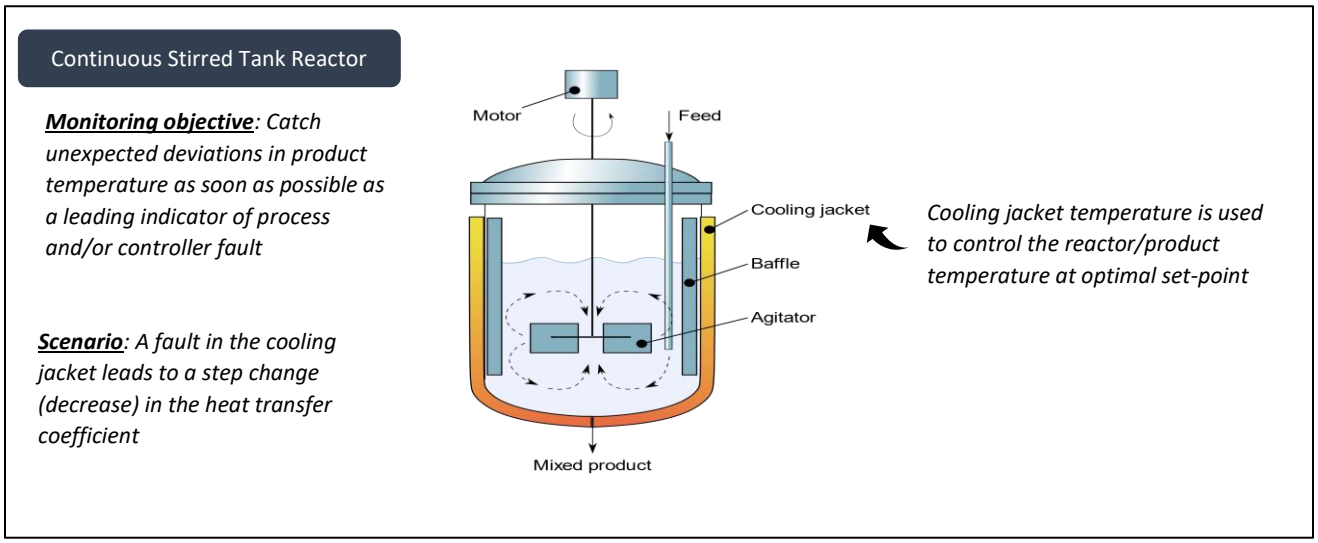


Figure 5.5: Process fault detection using TSA: case-study set-up[41]

Let's start by importing the required packages and exploring the provided I/O dataset. The first 3 hours of data will be used to build a model and the last 1 hour will be used as test data.

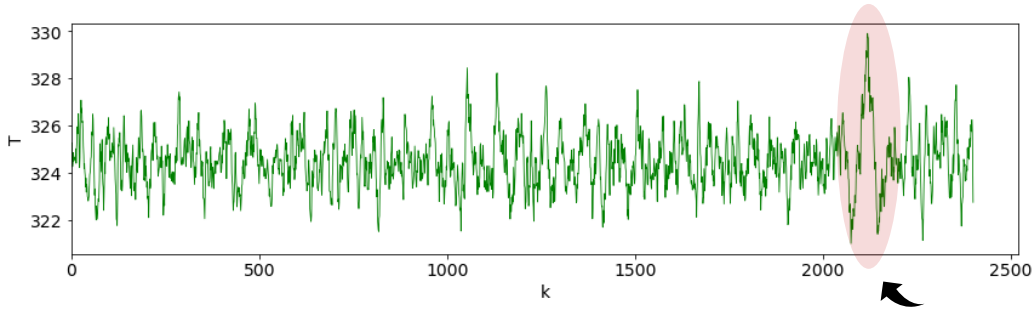
```
# import packages
import matplotlib.pyplot as plt, numpy as np, control
from statsmodels.graphics.tsaplots import plot_acf, plot_pacf
from statsmodels.tsa.stattools import arma_order_select_ic
from statsmodels.tsa.arima.model import ARIMA
```

[40] The CSTR model provided at https://github.com/APMonitor/pdc/blob/master/CSTR_Control.ipynb is used to generate the data.

[41] CSTR diagram created by Daniele Pugliesi under Creative Commons Attribution-Share Alike 3.0, https://commons.wikimedia.org/wiki/File:Agitated_vessel.svg

`

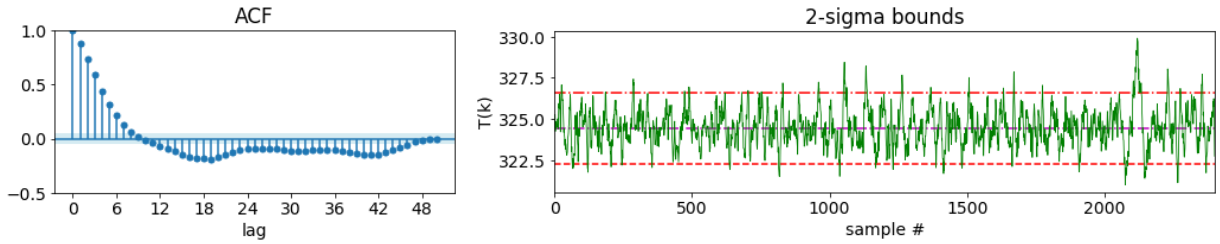
```
# read data and split into training and test data
T = np.loadtxt('CSTR_controlledTemperature.csv', delimiter=',')
T_train = T[:1800]; T_test = T[1800:]
```



Short-lived deviation in absolute value at start of the process-fault

The plot above shows that the temperature seems to show only a short-lived[42] deviation in its absolute value due to the process fault. Therefore, on the basis of this time-plot alone, plant operators could very well ignore this as an unimportant fluctuation and the process fault could go unattended.

The trivial univariate monitoring approach of using $mean \pm 2 * standard\ deviation$ bounds would also be inadequate because this approach assumes that the samples are independent of each-other, which as shown by the below ACF plot of *T_train* data, is certainly not true for the product outlet temperature (and dynamic processes in general). The control chart shown below shows the failure of this approach as the controlled variable does not show any sustained unusual violations of the control bounds (bounds are computed using the training data only).



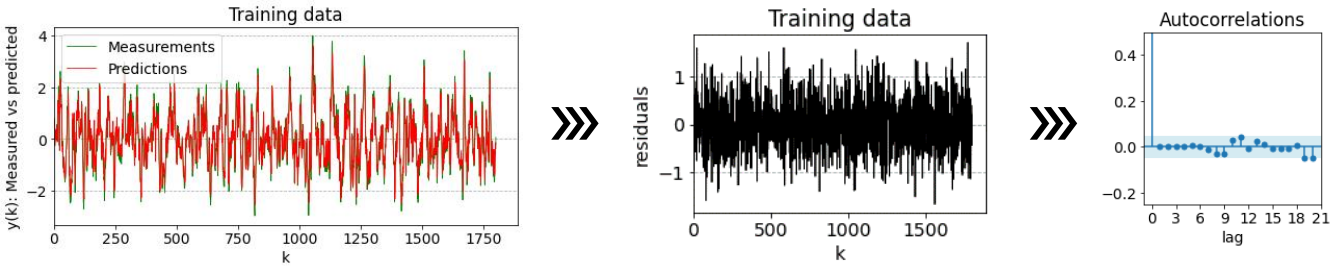
Let's now see if the time-series models can do a better job of fault detection. We will attempt to fit an ARMA model.

[42] The controller adjusts the jacket temperature to compensate for the reduction in the heat transfer coefficient.

`

```
# Determine the optimal AR and MA orders
T_train_centered = T_train - np.mean(T_train); T_test_centered = T_test - np.mean(T_train)
res = arma_order_select_ic(T_train_centered, max_ar=5, max_ma=5, ic=["aic"])
p, r = res.aic_min_order
print('(p, r) = ', res.aic_min_order)

>>> (p, r) =  (2, 4)

# Fit an ARMA(p,r) model and get residuals
model = ARIMA(T_train_centered, order=(p, 0, r))
results = model.fit()
T_train_centered_pred = results.fittedvalues # same as results.predict()
residuals_train = T_train_centered - T_train_centered_pred # same as results.resid
```

The residuals pass the model quality check.



It is time now to use the fitted ARMA model for fault detection. The rationale is straight-forward: if any process-fault leads to changes in process behavior then the model fitted using 'normal' operating condition data won't be very accurate with faulty data and therefore, the residuals will show higher values. Consequently, the residuals are monitored for fault detection as summarized in Figure 5.6.

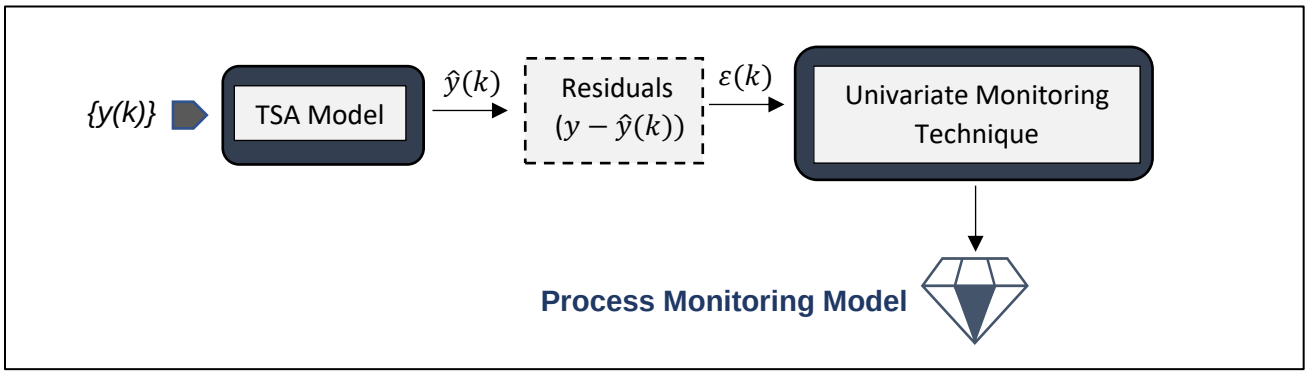
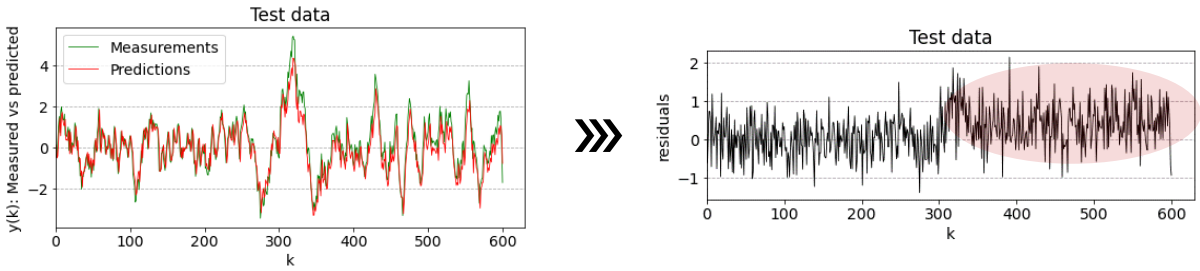


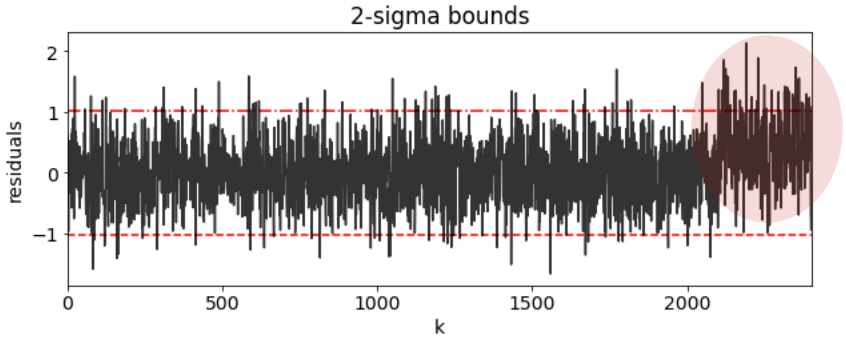
Figure 5.6: Process monitoring methodology via monitoring of time-series analysis-based residuals.

`

```
# 1-step ahead predictions for test data
results_test = results.apply(T_test_centered)
T_test_centered_pred = results_test.fittedvalues # same as results_test.predict()
residuals_test = T_test_centered - T_test_centered_pred
```



The above plots show that the residuals exhibit an increase in the mean value after the onset of process fault (after sample # 300 for test dataset). The control chart for the whole dataset makes the sustained unusual values of the residuals more evident. The fault diagnosis exercise can now be carried out to isolate the root cause of the fault.



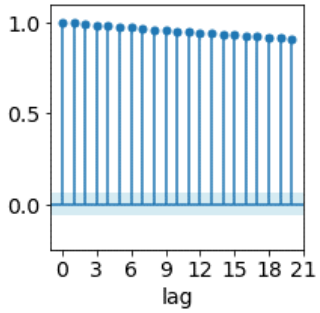
Fault detection via monitoring of model parameter estimates

We saw in the above example how process faults lead to change in the behavior of process variables. Therefore, as alternative to residual monitoring, another popular approach for process fault detection is to build new dynamic models for the process at regular intervals and check for significant changes in model parameters. Significant parameter changes can be good indicators of process fault.

Stationarity revisited and introduction to invertibility

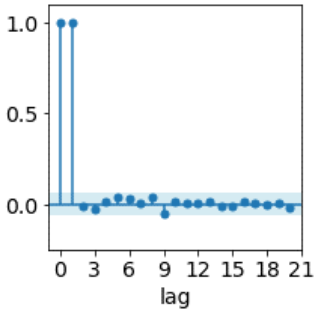
In Chapter 4, we had seen how homogeneous non-stationary signals can be made stationary by differencing. However, it is not always easy to infer homogeneous non-stationarity from simple time-plots. Fortunately, ACF, PACF, and PSD plots can provide crucial clues. The characteristic indicators of non-stationary behavior for these graphical plots are provided below. You should be easily able to explain the rationale behind these characteristics based on the concepts we have covered so far.

ACF plot



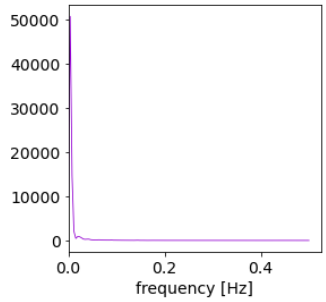
Slowly decaying/nearly constant

PACF plot



Near-unity value at lag 1

PSD



Significant power at low frequencies

Checking stationarity using noise transfer operator *H(q^{-1})*

Transfer operator for the ARMA model in Eq. 6 can be written as

$$H(q^{-1}) = \frac{1 + \sum_{j=1}^{r} c_j q^{-j}}{1 + \sum_{j=1}^{p} a_j q^{-j}} = \frac{C(q^{-1})}{A(q^{-1})}$$

numerator polynomial in q^{-1}

denominator polynomial in q^{-1}

An ARMA model represents a stationary process if the absolute of the roots of $A(q^{-1})$ is greater than 1. For example, for the AR(1) process *y(k) = ay(k-1) + e(k)*,

$$A(q^{-1}) = 1 - aq^{-1} \Rightarrow root = \frac{1}{a}$$

$$\Rightarrow stationarity\ if\ \left|\frac{1}{a}\right| > 1 \Rightarrow |a| < 1$$

In SysID literature, roots of $A(q^{-1})$ are called poles. Note that if degree of $A(q^{-1})$ or the order of autoregressive part is > 1, then poles can be complex numbers. Therefore, 'poles outside the unit circle' is the correct way of stating the condition of stationarity.

`

Poles inside the unit circle lead to explosive non-stationarity and poles on the unit circle causes homogeneous non-stationarity (such as random walk process). MA processes have $A(q^{-1}) = 1$ and are always stationary (for finite *r* and c_j).

Invertibility

Another condition imposed on disturbance model to represent realistic processes is invertibility. The condition requires that the roots of the numerator polynomial ($C(q^{-1})$), also called zeros of the transfer operator, lie outside the unit circle. AR models have $C(q^{-1}) = 1$ and therefore, are always invertible. Without going into mathematical details, it suffices to know that for non-invertible models, it is possible to write *e(k)* as a summation of future values of *y(k)*; such models are called non-causal models and do not represent realistic processes.

5.6 Autoregressive Integrated Moving Average (ARIMA) Models: An Introduction

In Chapter 4 we saw that non-stationary signals may be modeled via stationary ARMA models after suitable differencing. Such difference-stationary processes are formalized as ARIMA processes[43] and are represented as

$$\underbrace{(1-q^{-1})^d y(k)}_{\text{Integrating part with } d \text{ differences}} = \frac{C(q^{-1})}{A(q^{-1})}e(k) = \underbrace{\frac{1+\sum_{j=1}^{r} c_j q^{-j}}{1+\sum_{j=1}^{p} a_j q^{-j}}\, e(k)}_{\text{ARMA part}}$$

$$\Downarrow$$

$$z(k) = \frac{C(q^{-1})}{A(q^{-1})}e(k) \qquad \text{where } z(k) = \underbrace{(1-q^{-1})^d y(k)}_{\text{differenced signal}}$$

[43] Sometimes, modelers include a constant term θ_o in ARIMA model which then takes the following form $(1-q^{-1})^d y(k) = \theta_o + \frac{C(q^{-1})}{A(q^{-1})}e(k)$. In Chapter 4, we had shown that a deterministic trend of degree *d* can be removed with differencing *d* times. Therefore, the inclusion of θ_o in the model is equivalent to explicitly including a deterministic trend in the model.

`

For example, consider a signal that becomes ARMA(1,1) stationary after first differencing

$$(1-q^{-1})y(k) = \frac{1-cq^{-1}}{1-aq^{-1}}e(k) \quad \longleftarrow \text{transfer operator form}$$

$$\Rightarrow (1-aq^{-1})(y(k)-y(k-1)) = e(k) - ce(k-1)$$

$$\Rightarrow y(k) = (1+a)y(k-1) - ay(k-2) - ce(k-1) + e(k) \quad \longleftarrow \text{difference equation form}$$

It should have occurred to you by now that *q* notation makes it much easier to represent and work with complicated models.

The above model is called ARIMA(p,d,r) due to the d^{th} degree of differencing and ARMA(p,r) modeling. You can see that with suitable choice of *p, d*, and *r*, MA, AR, and ARMA models become special cases of ARIMA model.

Random walk model

An ARIMA(p,1,r) disturbance model is equivalent to saying that there are random (and temporally correlated) step changes occurring in the disturbance signal. A pure integrating model of order 1 (ARIMA(0,1,0), a.k.a. random walk model) takes the following form

$$(1-q^{-1})y(k) = e(k) \Rightarrow y(k) = y(k-1) + e(k)$$

The integrating model is called as such because the noise values are continuously added to the output signal.

Model order selection

The graphical tools can be used to check if differencing is needed. If the differenced series shows signs of AR or MA or ARMA, then no further differencing is required. Very rarely, you would need to choose *p, d*, or *r* greater than 2.

Unit-root tests

A rigorous way of checking the presence of unit pole (and therefore the need of differencing) is unit-root test. To understand this test, consider a simple AR(1) process *y(k) = ay(k-1) + e(k)*. To warrant differencing, '*a*' needs to be equal to 1 and therefore, in the regression model *z(k) = (a-1)y(k-1) + e(k)* where *z(k) = y(k) – y(k-1)*, the coefficient *a-1* would equal 0. The unit-

`

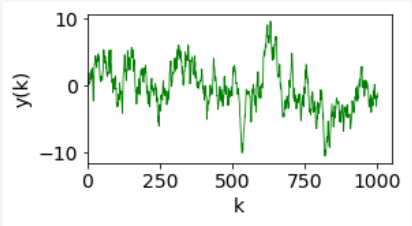
root test is simply a hypothesis test on this coefficient being equal to zero. This is called Dickey-Fuller test.

If you have some experience with hypothesis testing, then you would know that if the *p* value of the test is above a specified threshold/significance level[44], then the (null) hypothesis (on the series under study being non-stationary) is not rejected. For more generic series of integrating order > 1, a more generic augmented Dickey-Fuller test is used. However, the test process remains the same.

Example 5.4:

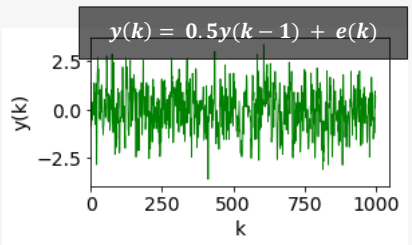
Generate data from AR process and perform ADF test to check for unit root.

```
# import packages
import numpy as np, matplotlib.pyplot as plt
from statsmodels.tsa.arima_process import ArmaProcess

# generate data
ar_coeffs = np.array([1, -0.96]) # y(k) = 0.96y(k-1) + e(k)
ARprocess = ArmaProcess(ar=ar_coeffs)
y = ARprocess.generate_sample(nsample=1000)

# perform augmented Dickey-Fuller test
from statsmodels.tsa.stattools import adfuller
result = adfuller(y)
print('p-value: %f' % result[1])
```

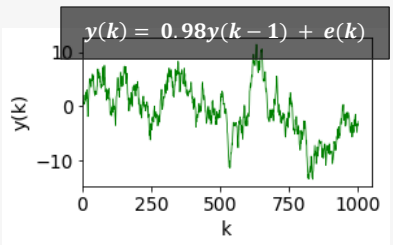


>>> p-value: 0.000187

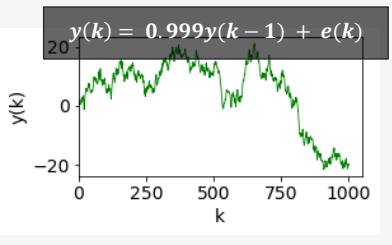
Since the p-value is less than 0.05, the null hypothesis is rejected ⇒ the time series is stationary



p-value: 0



p-value: 0.027



p-value: 0.904

[44] The threshold is commonly set to 5% or 0.05

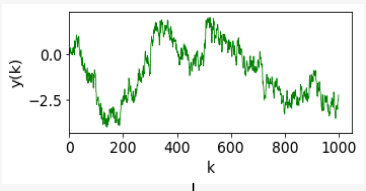
Example 5.5:

In this example we will utilize a given a time series generated from the following process. With guidance from the graphical plots and ADF test, we will infer the correct model orders. Thereafter, an ARIMA model will be fitted to find the model parameters.

$$(1 - q^{-1})y(k) = \frac{1 - 0.7q^{-1}}{1 - 0.5q^{-1}}e(k)$$

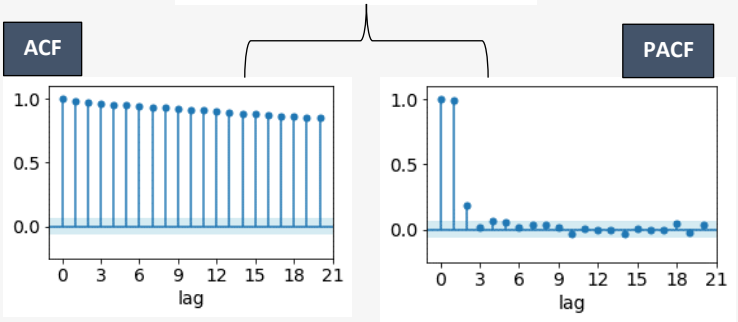
Note that packages like *pmdarima* exist which can be used to automatically estimate the order of differencing and the AR/MA orders.

```
# import packages and read data
import numpy as np, matplotlib.pyplot as plt
from statsmodels.tsa.arima.model import ARIMA
y = np.loadtxt('ARIMA_data.txt')
```

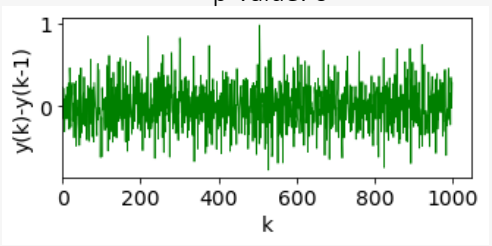


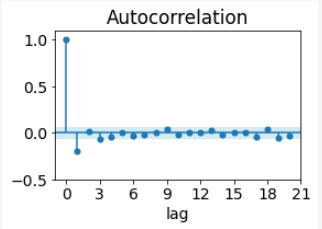
p-value: 0.325

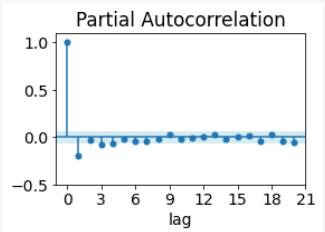
The correlation plots and the ADF test clearly indicate the presence of unit-root. Therefore, we know that differencing is required atleast once. Let's check the ACF and PACF plots of the difference signal.



p-value: 0







The ADF test indicates that the differenced signal is stationary and therefore no further differencing is required. Moreover, the ACF and PACF plots clearly indicate the AR and MA orders to be 1 (both plots show sharp decline in value after lag 1). We can therefore fit an ARIMA(1,1,1) model to the given data.

```
# fit ARIMA(1,1,1) model
y_centered = y - np.mean(y)
model = ARIMA(y_centered, order=(1, 1, 1))
results = model.fit()
print(results.summary())
```

```
==================================
                 coef    std err
----------------------------------
ar.L1          0.4883      0.096
ma.L1         -0.6723      0.082
```

Parameter estimates are close to actual

`

5.7 Forecasting Signals using ARIMA

Previously we saw how time series models can be used to build process monitoring solutions. As alluded to before, another common use of such models is in forecasting of stochastic signals. Although the *Statsmodels* package provides native support for generating forecasts, it would be instructive to see what the forecasting work process actually entails. We will use a simple ARIMA(1,1,1) model for this illustration. Let the model be the following

$$(1 - q^{-1})y(k) = \frac{1 - 0.7q^{-1}}{1 - 0.5q^{-1}}e(k)$$

Let's expand the above into a difference equation with only *y(k)* on the L.H.S.

$$y(k) = 1.5y(k-1) - 0.5y(k-2) - 0.7e(k-1) + e(k) \qquad \text{eq. 7}$$

To derive the expression for the forecasts from Eq. 7, the following notation/rules apply:

- $\hat{y}(k+l|k) \equiv \hat{y}(l)$ denotes the *l*th step-ahead forecast/prediction using all information available at time *k*
- Any *y* on R.H.S. of Eq. 7 that has not yet occurred is replaced with its forecast y
- Any future noise term is replaced with zero and past noise term with the 1-step ahead residuals (for example, $e(k-1) = y(k-1) - \hat{y}(k-1|k-2)$)

Let's see how the forecast expressions look like for $l = 1, 2, 3, \ldots$

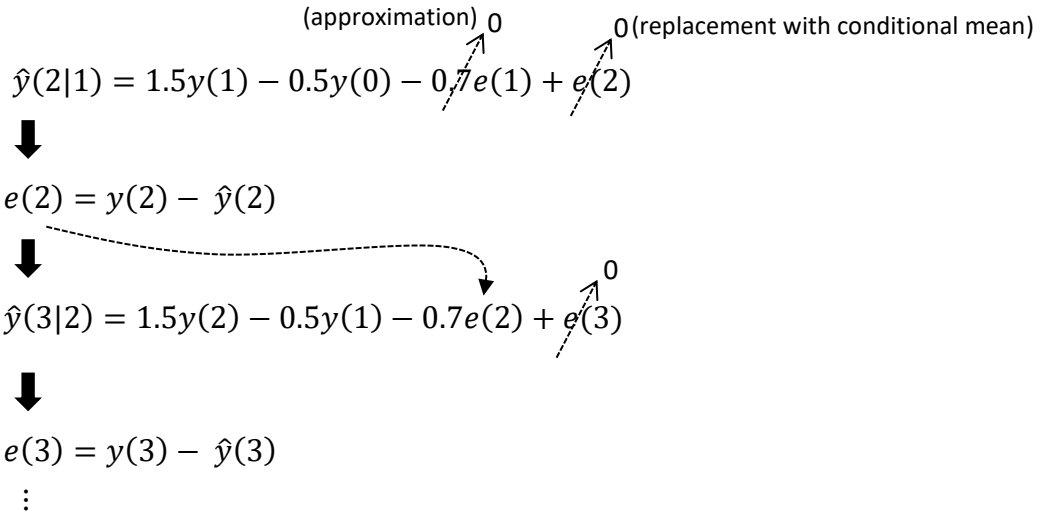
$$\hat{y}(1) = 1.5y(k) - 0.5y(k-1) - 0.7e(k)$$
$$\hat{y}(2) = 1.5\hat{y}(1) - 0.5y(k)$$
$$\hat{y}(3) = 1.5\hat{y}(2) - 0.5\hat{y}(1)$$
$$\vdots$$

to be replaced with $y(k) - \hat{y}(k|k-1)$

both error terms in Eq. 7 replaced with zero

A somewhat incomplete aspect in the above forecast expressions is computation of *e(k)*. The annotation text suggests making use of $\hat{y}(k|k-1)$, but it would involve the term *e(k-1)*. This leads to a recursive approach to find the past noise values. The established approach is to start from time *p* + *d*, set estimated noise values for times ≤ p + d to zero and then recursively estimate the noise values until time *k* using the forecast expression for the 1-step ahead forecast. For our ARIMA(1,1,1) model, the computation would look like the following:

`



This completes our quick overview of time-series analysis and its applications. Hopefully, you have gained an in-depth understanding of the concepts behind time-series analysis and can appreciate its usefulness in describing process systems.

Summary

In this chapter we looked at the four common models used to describe time series, namely, AR, MA, ARMA, and ARIMA models. We studied their model forms and procedures behind the corresponding model order selection. We looked at how ACF and PACF tools provide useful clues regarding model structure. We also looked into more details the stationarity requirements of TSA models and ways of assessing stationarity of signals. We will build upon the concepts covered in this chapter and look at models used to describe input-output dynamic signals in the next chapter.

`

Chapter 6

Input-Output Modeling - Part 1: Simple Yet Popular Classical Linear Models

There is beauty is simplicity - this age-old truth perfectly sums up the reason behind the popularity of two simple input-output models, FIR and ARX models, that we will learn about in this chapter. In the age of artificial neural networks, these models still hold on their own and are an essential part of any PDS's toolkit. As alluded to before, I/O processes have inputs signals that can be manipulated to control the output signals. The objective, therefore, of I/O modeling is to find the deterministic relationship between the input and output signals, and, optionally, find a suitable model for the stochastic disturbances affecting the outputs.

FIR model is a non-parametric model and is used in thousands of MPC solutions in process industry globally. The remarkable success of the FIR-based MPCs is testament to the fact that simple models can be quite useful representation of complex industrial processes. FIR models are intuitive and simple enough to be understood by plant operators. However, nothing is all rosy with anything. FIR models, owing to their flexibility often lead to overfitting. ARX models, on the other hand, are parametric models and require much lower number of parameters. Consequently, the resulting models are 'smoother'. ARX models are often the initial choice in any SysID exercise. Nonetheless, ARX models can easily suffer from high bias errors. Since these two models are so critical, this chapter is devoted to understanding their various aspects, along with their pros and cons, in detail.

Specifically, the following are covered in the chapter.

- Introduction to FIR and ARX models
- SysID of industrial furnaces using FIR and ARX models
- Hyperparameter selection for FIR and ARX models
- Stochastic component of ARX models
- Model bias in FIR and ARX models

`

6.1 FIR Models: An Introduction

Impulse response of a system is its response to a special impulse input as shown in Figure 6.1. The impulse response model approximates the system output as a linear combination of several impulse responses due to past inputs. Since the impulse responses usually decay to zero, it suffices to include only the past few values (say *M*) of *u(k)* to approximate *y(k)*. The value '*M*' is called the model order and the resulting model is called the finite impulse response (FIR) model. You will notice that there is no attempt to model stochastic disturbances and a non-parametric modeling form (there is no compact mathematical structure) is adopted. Consequently, FIR model is among the simplest SysID model.

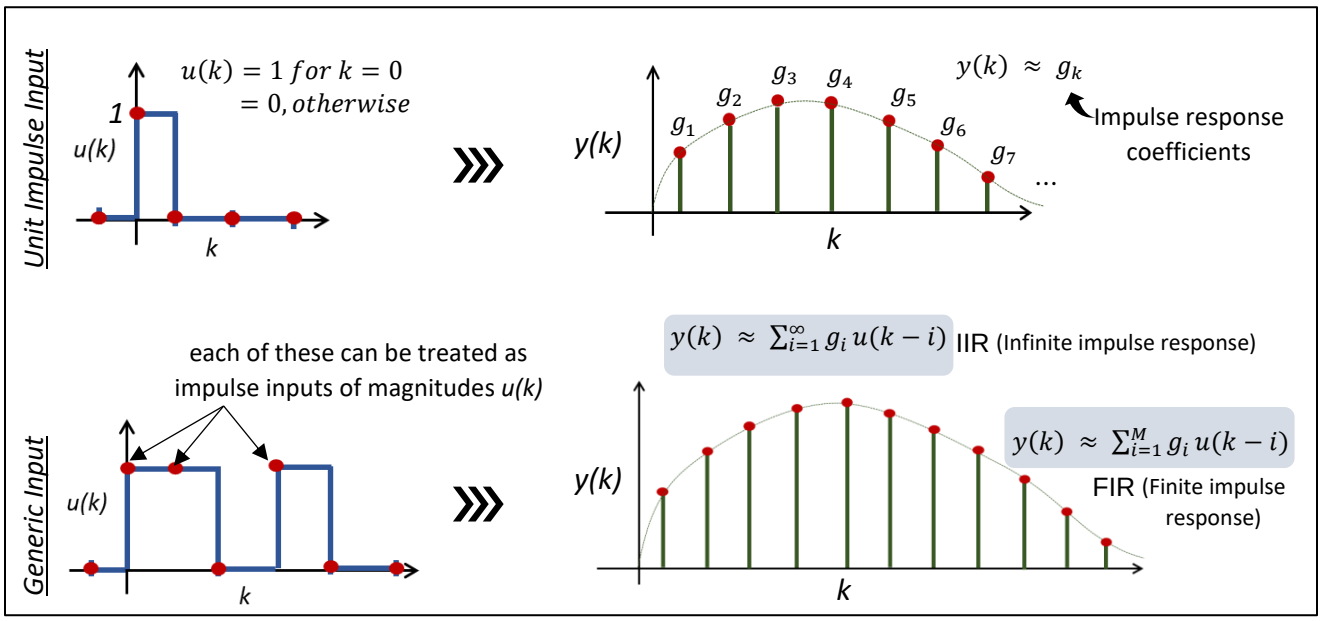


Figure 6.1: Impulse response model for a SISO system

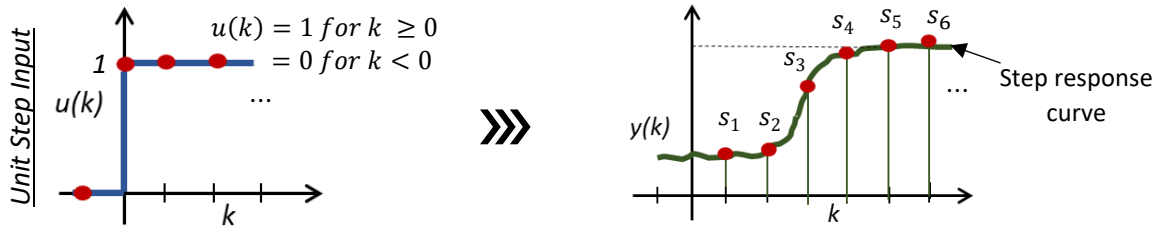
Fitting a FIR model entails estimation of the impulse response coefficients. Although a FIR model's structure is quite simple, it is very flexible and given sufficiently large *M*, any type of complex impulse response can be fitted. It is common to employ model orders of 90 to 120 to get all the impulse coefficients. However, such large parameters dimensionality often leads to overfitting (impulse response curves show wiggles or spikes due to output signal corruption by process disturbances). Apart from model flexibility, another advantage of FIR model is that an estimate of I/O delay is automatically obtained after model fitting. For a system with delay *d*, the first *d* impulse coefficients will be zero or close to zero. This is in fact a classical method for delay estimation. FIR models also provide unbiased parameter estimates as the deterministic part of the model is independently parametrized (recall our discussion from Chapter 4). These favorable properties of FIR models make them the dominant SysID model deployed in industrial MPCs[45].

[45] Darby and Nikolaou, MPC: Current practice and challenges. *Control Engineering Practice*, 2012

`

Step-response curves

Step response is response of a system to a step change in its input as shown below



As should be apparent, a step response curve can provide a quick assessment of process delay, time-constant of the process, and steady-state gain. System nonlinearity can also be quickly assessed by looking at response curves to steps of different magnitudes. Due to these favorable properties and the ease of inducing step inputs (compared to impulse inputs), step response curves are popularly used to visualize a system's behavior in process industry.

The step response coefficient, *s(k)*, can be easily estimated from the impulse coefficients as follows

$$s(k) = \sum_{j=1}^{k} g(j)$$

Model estimation

An FIR model[46] can be conveniently estimated using least squares method. Consider an FIR model of order *M* and the following data sequence

$$\left.\begin{matrix} y(0), y(1), \dots, y(N-1) \\ u(0), u(1), \dots, u(N-1) \end{matrix}\right\}$$ Available historical SISO process data

[46] A FIR model can compactly be written as $y(k) = \varphi_k^T \theta_0 + v(k)$ where φ_k is regressor vector ($[u(k-1), u(k-2), \dots, u(k-M)]^T$), θ_0 is parameter vector ($[g(1), g(2), \dots, g(M)]^T$), and $v(k)$ is disturbance signal which is ignored during estimation of θ_0. Note that $v(k)$ is not restricted to be white noise.

`

With $e(k) = y(k) - \sum_{i=1}^{M} g(i)u(k-i)$ as equation error, the data can be put in the following matrix form

$$\begin{bmatrix} y(M) \\ y(M+1) \\ \vdots \\ y(N-1) \end{bmatrix} = \begin{bmatrix} u(M-1) & u(M-2) & \cdots & u(0) \\ u(M) & u(M-1) & \cdots & u(1) \\ \vdots & \ddots & \cdots & \vdots \\ u(N-2) & \cdots & \cdots & u(N-1-M) \end{bmatrix} \begin{bmatrix} g_1 \\ g_2 \\ \vdots \\ g_M \end{bmatrix} + \begin{bmatrix} e(M) \\ e(M+1) \\ \vdots \\ e(N-1) \end{bmatrix}$$

$$\Downarrow$$

$$Y = \Phi\theta + E$$

Estimate of the parameter vector, $\hat{\theta}$, is obtained by minimizing the error vector. You probably would already know that an analytical solution for $\hat{\theta}$ exists and is given as $\hat{\theta} = [\Phi^T\Phi]^{-1}\Phi^T Y$.

Model order selection

Correct order selection is important in FIR modeling. Too low model order won't catch complex process dynamics and lead to inaccurate process gain estimates, and too high model order can lead to overfitting resulting in noisy impulse/step response curves. If any approximate prior knowledge is available about the process settling time or the time to steady state (TTSS), then it can be used to set the model order. For example, if TTSS is 120 then as many coefficients would be required in the model.

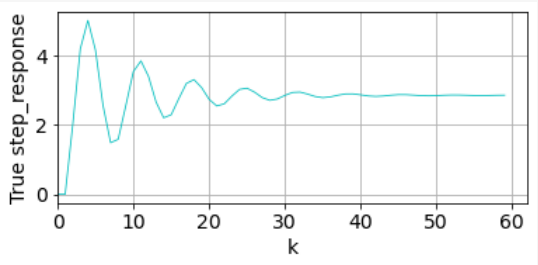
Example 6.1:

File *FIR_illustrate.csv* contains I/O data for the below process whose shown step response suggests a TTSS of ~30. The variance of the noise signal was adjusted to provide signal-to-noise ratio of 20. We will fit three FIR models with model orders 3, 30, and 60, respectively, and compare their step responses with the true curve.

$$y(k) = \frac{2q^{-2}}{1 - 1.1q^{-1} + 0.8q^{-2}}\, u(k) + v(k)$$

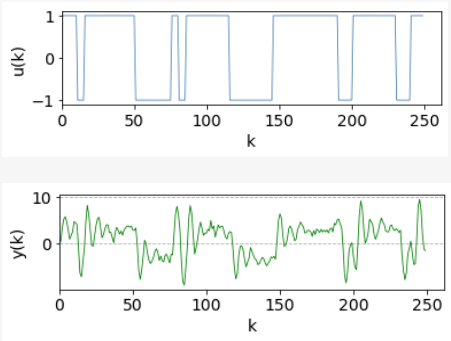
$$v(k) = \frac{1}{1 - 0.6q^{-1}}\, e(k)$$

An AR signal as disturbance



We will use the SIPPY package to fit our FIR models and employ Control package (https://pypi.org/project/control/) to generate step responses. Let's start with loading the data

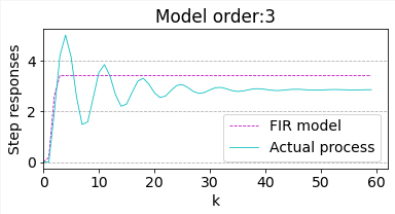
```
# import packages
import matplotlib.pyplot as plt, numpy as np
from sklearn.preprocessing import StandardScaler
from sippy import system_identification as SysID
import control

# read data
data = np.loadtxt('FIR_illustrate.csv', delimiter=',')
u = data[:,0, None]; y = data[:,1, None]

# center data before model fitting
u_scaler = StandardScaler(with_std=False)
u_centered = u_scaler.fit_transform(u)

y_scaler = StandardScaler(with_std=False)
y_centered = y_scaler.fit_transform(y)

# fit FIR model
model_order = 30
FIRmodel = system_ SysID (y_centered, u_centered, 'FIR', FIR_orders=[model_order,0])

# generate step response of FIR model
step_response_model, _ = control.matlab.step(FIRmodel.G, T=60)
plt.plot(step_response_model, 'm--', linewidth=0.8)
plt.ylabel('step_response_model'), plt.xlabel('k')
```



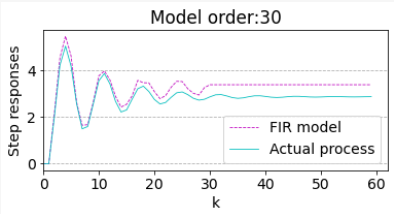
Time delay

Duration of step response

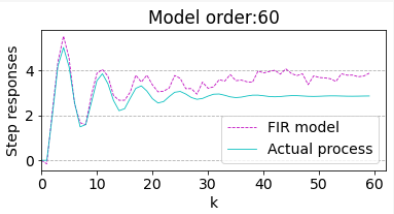
The plots below show the obtained step responses (along with the true step response) for the three different choices of the model order. [The reader can check out the code provided in the online repository to see how the true step response is generated.]



- Under-fitted
- Oscillations not captured



- Seemingly the right fit
- Oscillatory nature captured
- Gain estimate slightly off due to noise and limited training data



- Over-fitted
- Noisy response curve

`

6.2 FIR Modeling of Industrial Furnaces

To showcase an industrial application of FIR models, we will consider fired heaters used in petroleum refineries wherein, as shown in the figure below, fuel is combusted to heat feed oil and vaporize it for subsequent fractionation[47]. In the SISO version of the system, we will consider the task of keeping the outlet temperature of the heated oil (*TO*) in control. Changes in feed oil temperature, *TI*, or the feed flow will impact *TO*. The controller adjusts the flow of the fuel gas (although indirectly, by manipulating the set-point of the flue-gas PID controller. The PID controller, not shown in the schematic, adjusts the fuel-gas stream's valve opening to meet the specified flow)[48]. The process controller, however, needs a model to understand the relationship between flue gas flow set-point (FGSP) and *TO*.

To obtain the model, we will use data provided in the file *IndustrialFiredHeater_SISO.csv* which contains 1000 samples of TO and FGSP obtained from an open-loop simulation of the system wherein FGSP was excited using a GBN signal and the system was subjected to disturbances in *TI*.[49] Note that the time interval here is in minutes and not seconds.

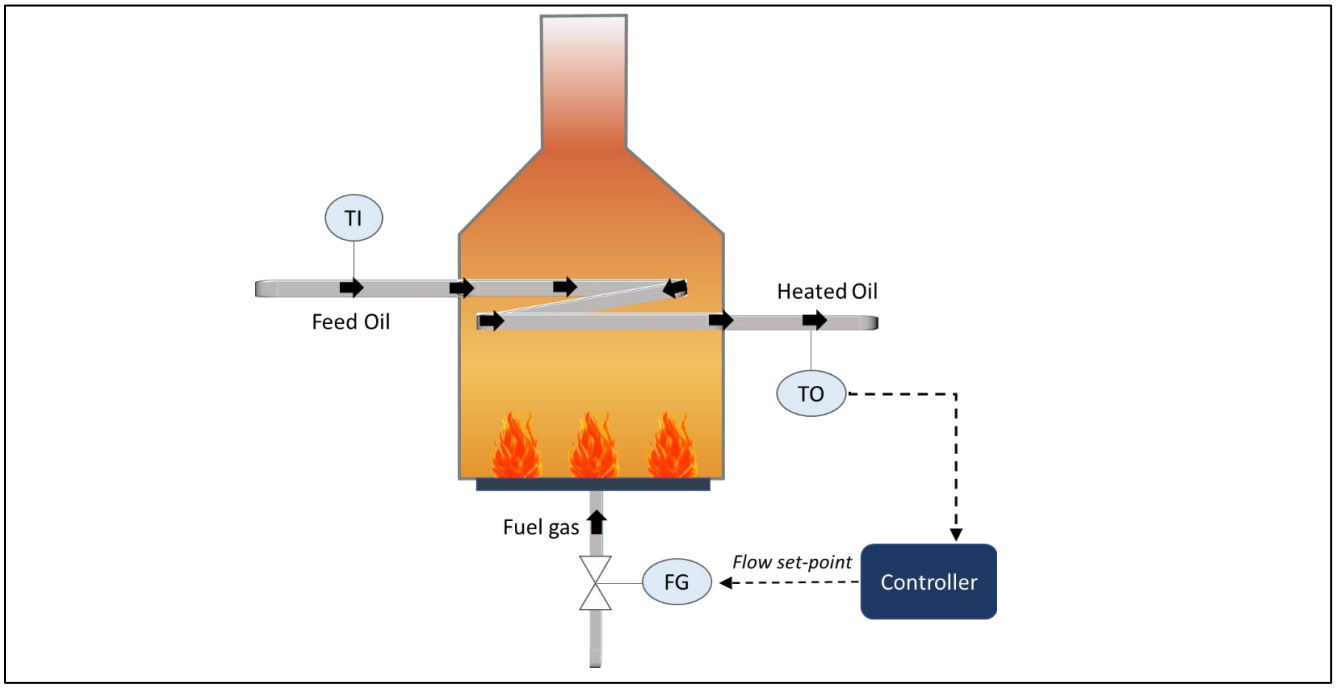


Figure 6.2: Representative schematic of a fired heater used in petroleum refineries

[47] The reader is referred to https://apmonitor.com/dde/index.php/Main/FiredHeaterSimulation for more details on the system. At this link, readers can also find the system model that is used to generate the simulation dataset used in this case-study.

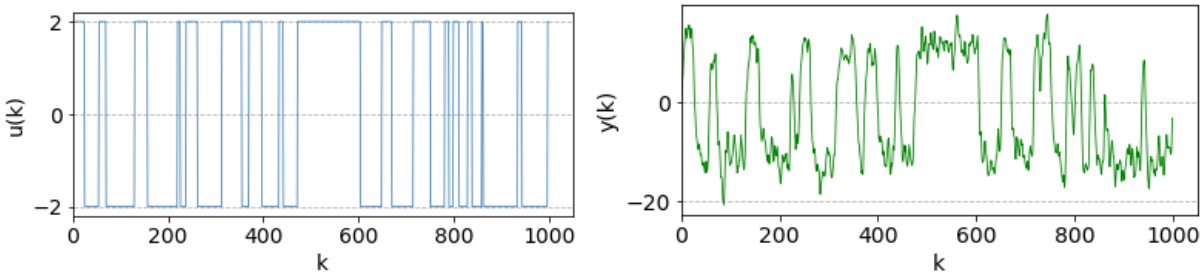
[48] Using PID controller's setpoints as manipulated variables is a very standard approach in modern (MPC) controllers.

[49] Online code repository shows how the identification test data was generated

`

Let's start by importing the required packages and exploring the provided I/O dataset.

```
# import packages
import matplotlib.pyplot as plt, numpy as np, control
from sklearn.preprocessing import StandardScaler
from sippy import system_identification as SysID
from statsmodels.tsa.stattools import ccf


# read fired heater identification test data and plot
data = np.loadtxt('IndustrialFiredHeater_SISO.csv', delimiter=',')
FG, TO = data[:,0, None], data[:,1, None]

plt.figure(), plt.plot(FG, 'steelblue'), plt.ylabel('u(k)'), plt.xlabel('k')
plt.figure(), plt.plot(TO, 'g'), plt.ylabel('y(k)'), plt.xlabel('k')
```



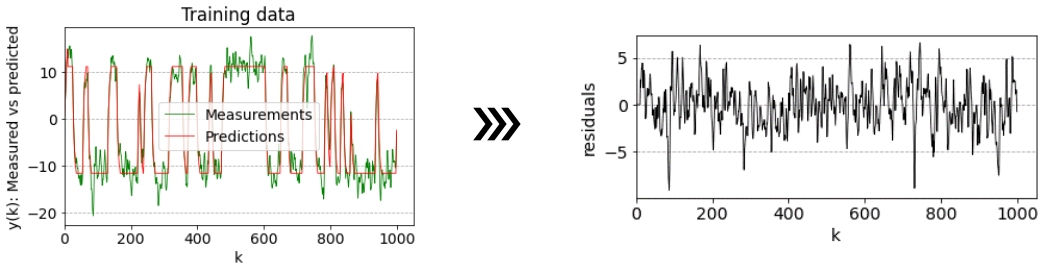
Next, we will center the data and fit an FIR model using AIC for model order selection.

```
# center data and fit FIR model
u_scaler = StandardScaler(with_std=False)
FG_centered = u_scaler.fit_transform(FG)

y_scaler = StandardScaler(with_std=False)
TO_centered = y_scaler.fit_transform(TO)

FIRmodel = SysID(TO_centered, FG_centered, 'FIR', IC='AIC', nb_ord=[1,20])
 # a range of 1 to 20 is specified for optimal order search

>>> suggested orders are: Na= 0 ; Nb= 11 Delay:  0
```

As shown, an FIR model with eleven coefficients has been found to provide the optimal fit. With the model fitted, we will conduct the model quality checks, starting with residual analysis.

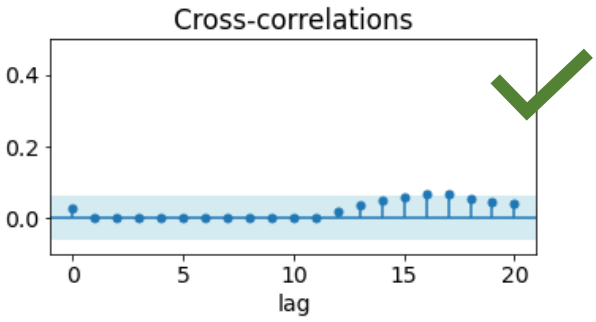
`

```
# get model predictions and residuals on training dataset
TO_predicted_centered = FIRmodel.Yid
# Yid attribute provides the output of the identified model during model ID
TO_predicted = np.transpose(y_scaler.inverse_transform(TO_predicted_centered))
residuals = TO - TO_predicted

plt.figure(), plt.plot(TO, 'g', label='Measurements')
plt.plot(TO_predicted, 'r', label='Predictions')
plt.title('Training data'), plt.ylabel('y(k): Measured vs predicted'), plt.xlabel('k'), plt.legend()
```



Note that we will not perform any autocorrelation check on the above residuals because FIR model's residuals are not constrained to be white noise. However, cross-correlations between the residual and the input sequences need to be checked.

```
# CCF b/w residual and input sequences
ccf_vals = ccf(residuals, FG, adjusted=False) # ccf for lag >= 0
ccf_vals = ccf_vals[:21] # ccf for lag 0 to 20

lags, conf_int = np.arange(0,21), 1.96/np.sqrt(len(residuals))
plt.figure(), plt.vlines(lags, [0], ccf_vals), plt.axhline(0, 0, lags[-1])
plt.plot(lags, ccf_vals, marker='o', linestyle='None')
plt.gca().axhspan(-conf_int, conf_int, facecolor='lightblue') # shaded confidence interval
```

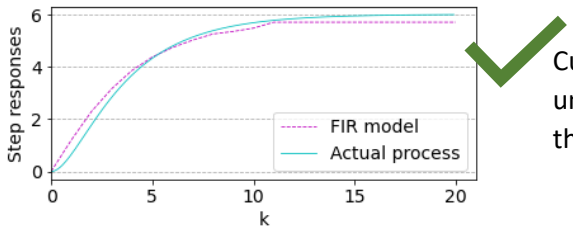


Cross-correlation coefficients are within the confidence interval

`

Next, we will check if the fitted model has any unexpected characteristics; for this we will generate the model's step response curve and compare with the true process curve.

```
# generate step response of FIR model
step_response_model, _ = control.matlab.step(FIRmodel.G, T=21)

# generate step response of original process
s = control.matlab.tf('s')
sys_G  = 6/(3*s**2 + 4*s + 1) # https://apmonitor.com/dde/index.php/Main/FiredHeaterSimulation
step_response_process, T = control.matlab.step(sys_G, T=20)

# overlay the step responses
plt.figure(), plt.plot(step_response_model, 'm--', label="FIR model")
plt.plot(T, step_response_process, 'c', label="Actual process")
plt.ylabel('Step responses'), plt.xlabel('k'), plt.legend()
```



Curves compare well; no unexpected characteristics in the model's curve

In practice, you will often not be so lucky as to have access to the true process step response curve. Nonetheless, you will probably have some process insights to judge if the model's step response curve is reasonable. Overall, our model seems good to be accepted. Note that we did not perform any simulation checks here because for an FIR model, simulation equals predictions. We will perform simulation checks for the ARX model later in this chapter.

`

6.3 ARX Models: An Introduction

ARX (autoregressive with eXogenous input) model is extension of AR time series model to I/O systems. Here, as shown in Figure 6.3, the current output state is written as a linear combination of past inputs as well as past output measurements[50]. The FIR model may be seen as a special case of ARX model with $n_a = 0$. However, the inclusion of past measurements in the ARX model allows for more compact description of the process and helps to overcome the high parameter-dimensionality issue faced by FIR model. The price for parsimonious representation is paid for by having to specify more hyperparameters (d, n_a, n_b).

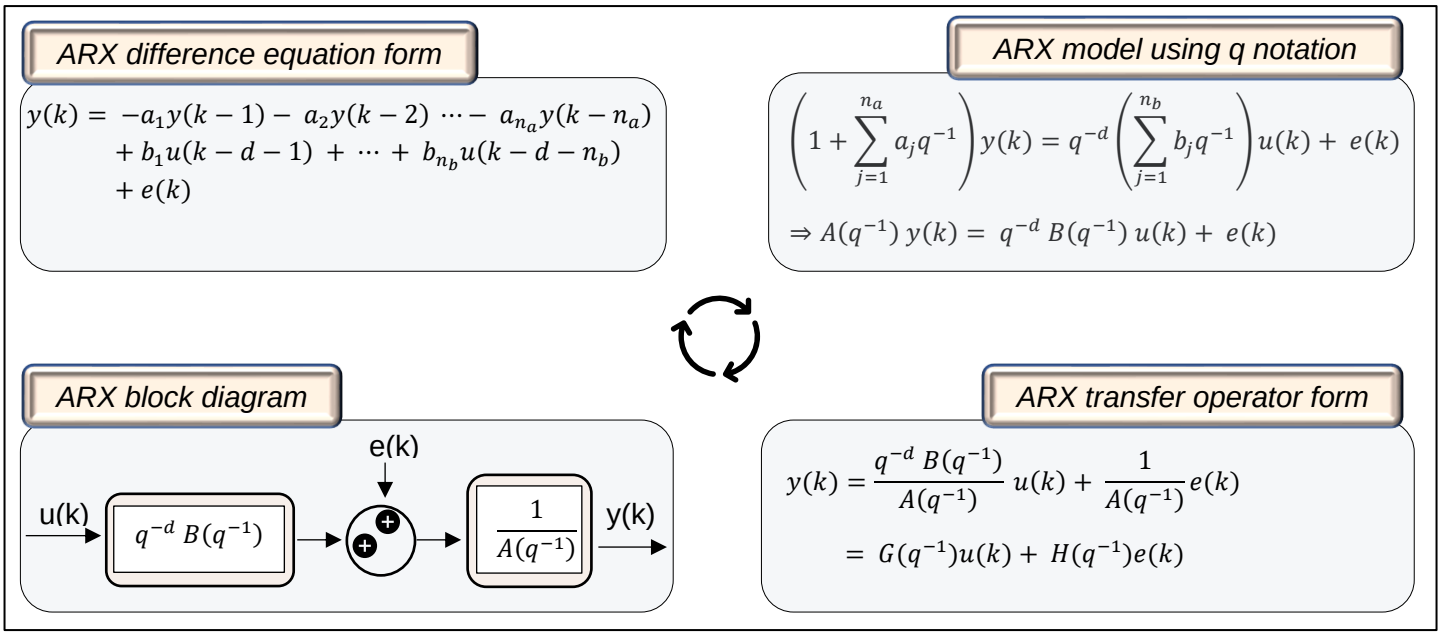


Figure 6.3: ARX model structure in different forms

A salient (and often overlooked) feature of ARX model is the implicit assumption that the noise enters the system at such a point that it is subjected to the same process dynamics ($\frac{1}{A(q^{-1})}$) as the deterministic signal *u(k)*. The block diagram in Figure 6.3 emphasizes this aspect. In real world, an example scenario could be perturbations affecting the process feed which then passes through the entire process dynamics. Not many systems that you will encounter will show such noise characteristics. Moreover, the dependent parametrization of *G* and *H* operators imply that an ARX model is biased for incorrect noise model specification even if you choose your model orders for the deterministic part correctly. Despite the shortcomings, the convenience of estimating ARX models often makes them the first choice for an initial solution in any SysID exercise.

[50] We have not included the term *u(k)* in the difference equation forms of FIR and ARX models. This is because we are working with discrete-time models and any changes in *u(k)* most often does not have instantaneous impact on *y(k)*.

`

Model estimation

Least squares method can again be employed here. For illustration, let's consider a 2nd order ARX[51] model with delay = 1: $y(k) = -a_1 y(k-1) - a_2 y(k-2) + b_1 u(k-2) + b_2 u(k-3) + e(k)$. Let $\varphi_k = [-y(k-1), -y(k-2), u(k-2), u(k-3)]^T$ be the regressor vector and $\theta_0 = [a_1, a_2, b_1, b_2]^T$ be the parameter vector. Therefore, $y(k) = \varphi_k^T \theta_0 + e(k)$ and the matrix form would look as follows

$$\begin{bmatrix} y(3) \\ y(4) \\ \vdots \\ y(N-1) \end{bmatrix} = \begin{bmatrix} -y(2) & -y(1) & u(1) & u(0) \\ -y(3) & -y(2) & u(2) & u(1) \\ \vdots & \ddots & \cdots & \vdots \\ -y(N-2) & \cdots & \cdots & u(N-4) \end{bmatrix} \begin{bmatrix} a_1 \\ a_2 \\ \vdots \\ b_2 \end{bmatrix} + \begin{bmatrix} e(3) \\ e(4) \\ \vdots \\ e(N-1) \end{bmatrix}$$

$$\Downarrow$$

$$Y = \Phi\theta + E$$

The analytical solution, $\hat{\theta} = [\Phi^T \Phi]^{-1} \Phi^T Y$, exists if the matrix $\Phi^T \Phi$ is invertible which is guaranteed if the input signal *u(k)* is persistent exciting of sufficient order.

Model bias in ARX and FIR models

We have noted that the ARX model's assumption on noise signal is pretty restrictive. If the actual process has a stochastic part different from $\frac{e(k)}{A(q^{-1})}$, *then* $\hat{\theta}$ *estimated via least squares will be biased[52] even if G(q-1)'s structure is correct because G(q-1) and H(q-1) are not independently parameterized.*

We can look at this aspect from another perspective: one of the requirements in least squares method for bias-free estimates is that the equation error is uncorrelated with the regressor vector. Now let's consider an example scenario where the actual process is given by $y(k) = -a_{10} y(k-1) + b_{10} u(k-1) + e(k) + c_{10} e(k-1)$ *and the fitted model is 1st order ARX model* $y(k) = -a_1 y(k-1) + b_1 u(k-1)$. *Since the equation error* $\epsilon(k)$ *equals* $e(k) + c_{10} e(k-1)$, *it becomes correlated to the regressor vector as y(k-1) is*

[51] An ARX model requires specification of two orders: order of the numerator ($B(q^{-1})$) and denominator ($A(q^{-1})$) polynomials. Specification of only a single order value usually implies $n_a = n_b$.

[52] A simple way to understand bias is that if multiple estimates of a parameter is obtained from different sets of I/O data, then the difference between the average of the estimates and the true parameter value is the bias.

correlated to e(k-1). Correspondingly, parameter estimates are biased[53]. The amount of bias in parameter estimates depend on the level of noise. For high SNR, the bias can be negligible.

In FIR model, $H(q^{-1})$ is not modeled and therefore, FIR model's least-squares estimates are unbiased. Note that these discussions assume that the disturbance signal is not correlated to input which is true for open-loop processes.

Example 6.2:

In this example we will study the impact of SNR on ARX model bias. We will assume the following true process

$$y(k) = \frac{0.7q^{-1}}{1-0.7q^{-1}}u(k) + \frac{1}{1-0.2q^{-1}}e(k)$$

We will use SNRs of 10 and 100. The data files ARX_illustrate_SNR10.csv and ARX_illustrate_SNR100.csv contain I/O data with 1000 samples each. We will fit 1st order ARX models using least squares and compare the model's step responses and estimated parameters with those from the true process.

```
# import packages
import matplotlib.pyplot as plt, numpy as np
from sklearn.preprocessing import StandardScaler
from sippy import system_identification as SysID
import control

# read data
data = np.loadtxt('ARX_illustrate_SNR10.csv', delimiter=',')
u = data[:,0, None]; y = data[:,1, None]

# center data before model fitting
u_scaler = StandardScaler(with_std=False); u_centered = u_scaler.fit_transform(u)
y_scaler = StandardScaler(with_std=False); y_centered = y_scaler.fit_transform(y)

# fit ARX model
ARXmodel = SysID(y_centered, u_centered, 'ARX', ARX_orders=[1,1,0])
print(ARXmodel.G) # shows the transfer operator G (in terms of q); Note that
symbol z is used instead of q in the package
```

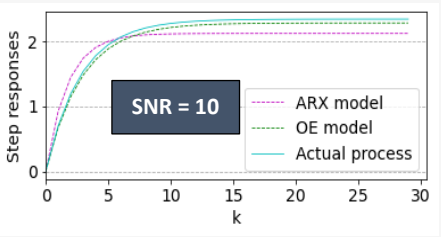
$$G(q^{-1}) = \frac{0.92q^{-1}}{1-0.56q^{-1}}$$

[53] Alternatives to least squares method such as instrumental variable (IV) method exist which can provide bias-free estimates for ARX model. However, expectedly, these alternative methods are computationally more involved.

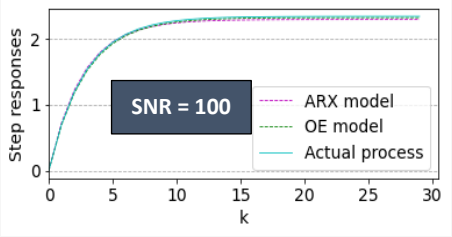
We can see that with significant level of disturbance, the parameter estimates are not very accurate. Let's see how the model's step response compares against that of the true process.

```
# generate  step response of ARX model
step_response_model, _ = control.matlab.step(ARXmodel.G, T=30)

# generate step response of original process
NUM_G, DEN_G = [0.7], [1, -0.7]
sys_G = control.matlab.tf(NUM_G, DEN_G, 1)
step_response_process, _ = control.matlab.step(sys_G, T=30)

# overlay the step responses
plt.plot(step_response_model, 'm--', linewidth=0.8, label="ARX model")
plt.plot(step_response_process, 'c', linewidth=0.8, label="Actual process")
plt.ylabel('Step responses'), plt.xlabel('k'), plt.legend()
```

Plots below shows the step response curves and the estimated parameters. As expected, for SNR=10, the estimated step response curve is farther away from the true curve compared to those from SNR=100 dataset. However, high noise level is not solely responsible for this inaccuracy. Incorrect noise model contributes significantly as well. To illustrate this fact, the plots show curves from an OE model (it is covered in the next chapter, but it suffices to know that an OE model uses similar structure for operator *G* as an ARX model but ignores *H*) as well. Theoretically, OE model provides unbiased estimated and therefore its curves are much closer to the true curves.

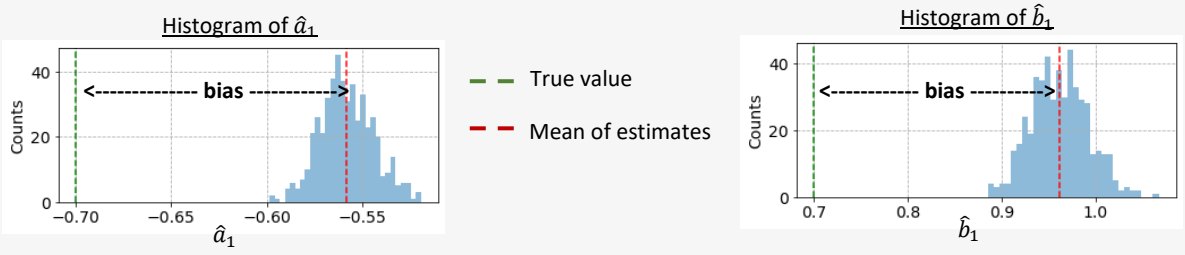


$$G_{ARX} = \frac{0.92q^{-1}}{1 - 0.56q^{-1}}$$



$$G_{ARX} = \frac{0.72q^{-1}}{1 - 0.68q^{-1}}$$

To further clarify the bias aspect, 500 different sets of I/O data was generated for SNR=10 and least squares estimation method was used. The histograms below show the distribution of the parameters. The systematic bias is evident in the plots.



`

Unbiased high-order ARX modeling

In SysID literature, you may encounter statements stating that high-order ARX models are unbiased. To understand this, let's consider an example where the true process is given by

$$y(k) = \frac{B(q^{-1})}{A(q^{-1})}u(k) + \frac{e(k)}{A(q^{-1})D(q^{-1})}$$

Now instead of fitting an ARX model $y = \frac{B}{A}u + \frac{e}{A}$, we can fit the model $y = \frac{BD}{AD}u + \frac{e}{AD}$ [q^{-1} and *k* not explicitly specified for brevity]. Therefore, if the degree of $D(q^{-1})$ is *r* then we are basically fitting an ARX model with orders n_a+*r* and n_b+*r* instead of an ARX model with orders n_a and n_b. The higher order ARX model will be unbiased because both *G* and *H* operators are correctly specified! Furthermore, your model residuals will be white now. One can recover the true transfer operator $\frac{B}{A}$ from the estimated transfer operator $\frac{BD}{AD}$.

It has been shown that a sufficiently high-order ARX model can approximate any linear system arbitrarily well. Model order reduction techniques exist to subsequently reduce the order of the estimated ARX model to make it more tractable.

Hyperparameter selection

Complete specification of an ARX model structure requires values of the input-output delay (*d*) and the orders (n_a and n_b which are the number of poles and zeros of the input transfer operator *G*). The most common way of estimating these values is to select a range of potential values for *d*, n_a, and n_b, and use information criteria such as AIC or cross-validation to find the most optimal combination. SIPPY implements the AIC-based methodology. As a thumb-rule, for industrial plants, n_a is usually three or less.

Approaches alternative to the above brute-force approach exist as well. For example, you can first build an FIR model to get an estimate of the delay, or a CCF plot between *u(k)* and *y(k)* can be utilized. A process delay of *d* will imply that the CCF possesses its first significant peak at lag $d + 1$, while the coefficients will be close to zero for lag $\leq d$. With the delay estimate in hand, you may then proceed for order estimation via brute-force. Here again, an approximate order value may be obtained using the Hankel Singular values obtained via subspace identification procedure (you will learn about this in Chapter 8). This approximate value can help you decide the potential range for n_a (and n_b) in your brute-force search.

`

Once your model has been fitted, model diagnostics may provide you clues regarding misspecification of the model structure. For example, isolated significant peaks in CCF plot between *u(k)* and $\epsilon(k)$ indicate missing input terms in the model. Too small delay specification will lead to insignificant parameter estimates of the first few input term coefficients. Too large value of *d* will show up as several contiguous CCF coefficients (*u(k)* and $\epsilon(k)$) outside the confidence bounds.

Let's put these guidelines into practice in a realistic case-study presented next.

6.4 ARX Modeling of Industrial Furnaces

Previously, we built a parametric FIR model using the fired-heater dataset; the model ended up with 11 parameters. In this section, we will attempt to build a parametric ARX model and see if it ends up providing a more parsimonious description of the system. An ARX model can be fitted as easy as it was for fitting an FIR model using SIPPY as shown below

```
# fit ARX model with centered signals
ARXmodel = SysID(TO_centered, FG_centered, 'ARX', IC='AIC', na_ord=[1,20], nb_ord=[1,20], delays=[0,5])

>>> suggested orders are: Na= 4 ; Nb= 3 Delay:  0
```

The fitted model ends up with 7 parameters and the following structure

$$G_{ARX} = \frac{0.8405q^3 - 0.1983q^2 + 0.1147q}{q^4 - 1.541q^3 + 0.9914q^2 - 0.4023q + 0.0877}$$

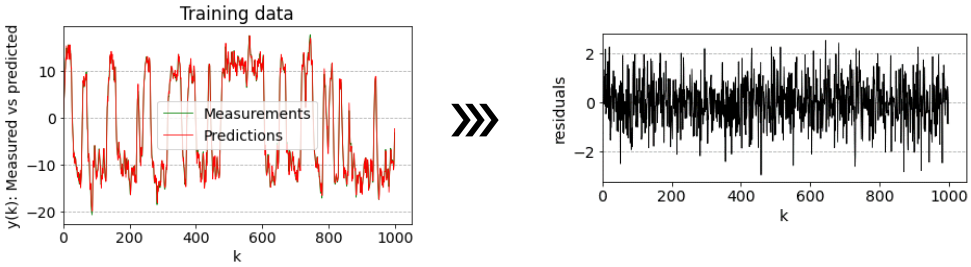
$$= \frac{0.8405q^{-1} - 0.1983q^{-2} + 0.1147q^{-3}}{1 - 1.541q^{-1} + 0.9914q^{-2} - 0.4023q^{-3} + 0.0877q^{-4}}$$

Let's see if this parsimonious model provides as good performance as the FIR model and passes the diagnostic checks. As before, we start with residual analysis.

```
# get model predictions and residuals on training dataset
TO_predicted_centered = ARXmodel.Yid
TO_predicted = np.transpose(y_scaler.inverse_transform(TO_predicted_centered))
residuals = TO - TO_predicted
```

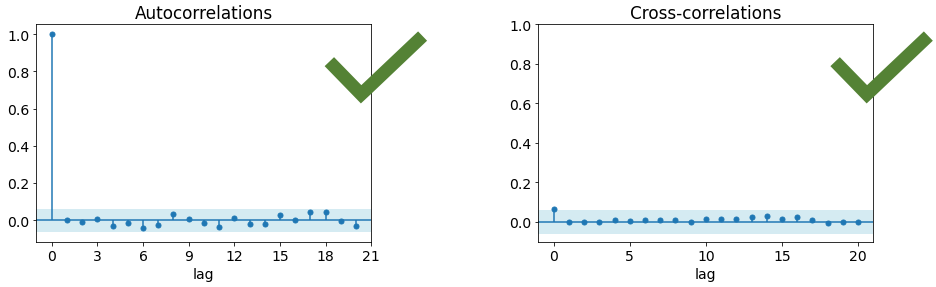
`

```
plt.figure(), plt.plot(TO, 'g', label='Measurements')
plt.plot(TO_predicted, 'r', label='Predictions')
plt.title('Training data'), plt.ylabel('y(k): Measured vs predicted'), plt.xlabel('k'), plt.legend()
```



The model predictions on the training dataset seems to be almost perfect. However, we know that 1-step ahead predictions on training dataset can be misleading and therefore, we will hold onto our judgement for a bit. Let's look at the ACF and CCF plots of the residuals.

```
# ACF of residuals
from statsmodels.graphics.tsaplots import plot_acf
conf_int = 1.96/np.sqrt(len(residuals))
plt.figure(), plot_acf(residuals, lags= 20, alpha=None, title='')
plt.gca().axhspan(-conf_int, conf_int, facecolor='lightblue') # shaded confidence interval

# CCF b/w residual and input sequences
ccf_vals = ccf(residuals, FG, adjusted=False) # ccf for lag >= 0
ccf_vals = ccf_vals[:21] # ccf for lag 0 to 20

lags = np.arange(0,21)
plt.figure(), plt.vlines(lags, [0], ccf_vals), plt.axhline(0, 0, lags[-1])
plt.plot(lags, ccf_vals, marker='o', linestyle='None')
plt.gca().axhspan(-conf_int, conf_int, facecolor='lightblue') # shaded confidence interval
```

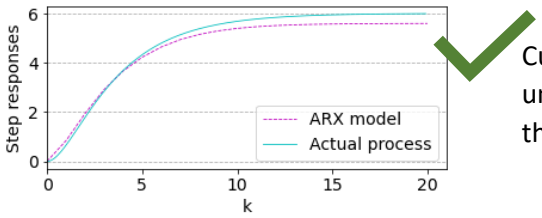


Coefficients of interest are within the confidence intervals

`

Next, let's see the model's step response curve (and compare with the true process curve).

```
# generate step response of ARX model
step_response_model, _ = control.matlab.step(ARXmodel.G, T=21)

# generate step response of original process
s = control.matlab.tf('s')
sys_G  = 6/(3*s**2 + 4*s + 1)
step_response_process, T = control.matlab.step(sys_G, T=20)

# overlay the step responses
plt.figure(), plt.plot(step_response_model, 'm--', label="ARX model")
plt.plot(T, step_response_process, 'c', label="Actual process")
plt.ylabel('Step responses'), plt.xlabel('k'), plt.legend()
```



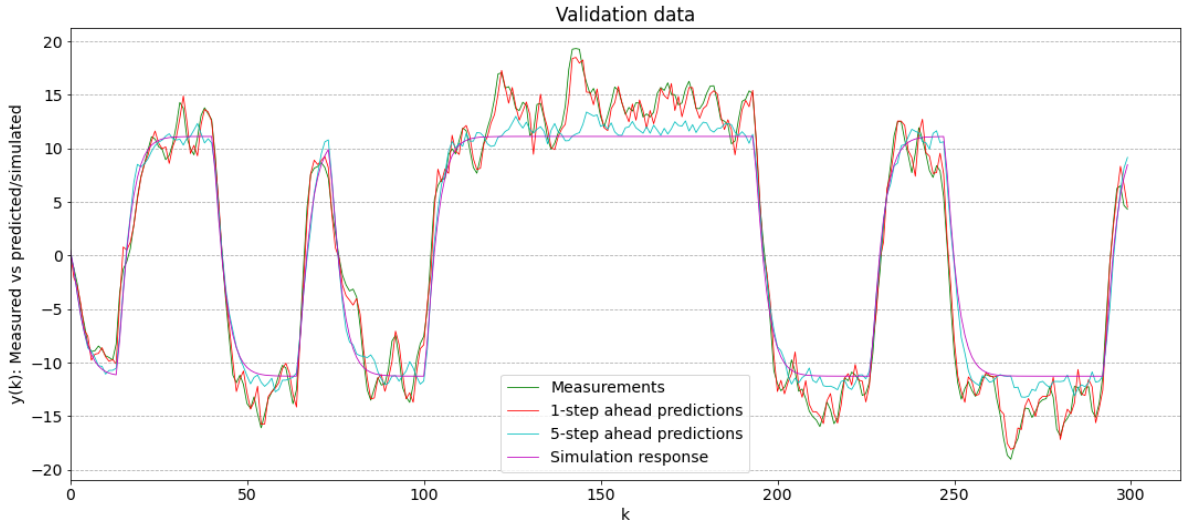
Curves compare well; no unexpected characteristics in the model's curve

The model has done reasonably well so far. We will do one last check and analyze the model's *m*-step ahead responses and simulation responses on a separate validation dataset.

```
# read fired heater validation data and center
data_val = np.loadtxt('IndustrialFiredHeater_SISO_Validation.csv', delimiter=',')
FG_val,  TO_val  = data_val[:,0, None], data_val[:,1, None]
FG_val_centered, TO_val_centered = u_scaler.fit_transform(FG_val), y_scaler.fit_transform(TO_val)

# get model's 1-step ahead and 5-step ahead predictions, and simulation responses
from sippy import functionset as fset

TO_val_predicted_1step_centered = np.transpose(fset.validation(ARXmodel, FG_val_centered,
                                                TO_val_centered, np.linspace(0,299,300), k=1))
TO_val_predicted_5step_centered = np.transpose(fset.validation(ARXmodel, FG_val_centered,
                                                TO_val_centered, np.linspace(0,299,300), k=5))
TO_val_simulated_centered, _, _ = control.matlab.lsim(ARXmodel.G, FG_val_centered[:,0],
                                                np.linspace(0,299,300))
```

`

```
TO_val_predicted_1step = y_scaler.inverse_transform(TO_val_predicted_1step_centered)
TO_val_predicted_5step = y_scaler.inverse_transform(TO_val_predicted_5step_centered)
TO_val_simulated = y_scaler.inverse_transform(TO_val_simulated_centered)

plt.figure()
plt.plot(TO_val, 'g’, label='Measurements')
plt.plot(TO_val_predicted_1step, 'r', label='1-step ahead predictions')
plt.plot(TO_val_predicted_5step, 'c', label='5-step ahead predictions')
plt.plot(TO_val_simulated, 'm', label='Simulation response')
plt.title('Validation data'), plt.ylabel('y(k): Measured vs predicted/simulated'), plt.xlabel('k')
plt.legend()
```



The plot above again corroborates the adequacy of the model. The 5-step ahead predictions and the simulated responses seem to have captured the systematic variations in the temperature signal which suggests that the deterministic component of the model has been accurately modeled.

`

6.5 FIR and ARX MIMO Models

While we have used SISO systems to illustrate the concepts, in industrial practice, you are more likely to deal with MISO (multi-input single output) systems where, as the name suggests, there are multiple input variables affecting a single output variable. Extension of SISO ARX and FIR models to MISO system is straightforward as shown below

ARX MISO model: $$y(k) = \frac{B_1(q^{-1})}{A(q^{-1})}u_1(k) + \frac{B_2(q^{-1})}{A(q^{-1})}u_2(k) + \cdots + \frac{e(k)}{A(q^{-1})}$$

FIR MISO model: $$y(k) = \sum_{i=1}^{M_1} g_{1,i}u_1(k-i) + \sum_{i=1}^{M_2} g_{2,i}u_2(k-i) + \cdots + v(k)$$

In the expression above for the ARX MISO model, the denominator polynomial is assumed to be the same, $A(q^{-1})$, in the transfer operators for all the inputs. This is a special case of general MISO ARX model which uses different denominator polynomials (which however leads to nonlinear parameter estimator problem). For MIMO (multi-input multi-output) systems, the usual approach is to build separate MISO models for each output. For example, for a 2-output 2-input system, your ARX model could look as follows

$$A_1(q^{-1})y_1(k) = B_{11}(q^{-1})u_1(k) + B_{12}(q^{-1})u_2(k) + e_1(k)$$

$$A_2(q^{-1})y_2(k) = B_{21}(q^{-1})u_1(k) + B_{22}(q^{-1})u_2(k) + e_2(k)$$

It is not difficult to see that the above framework of independent parametrization of y_1's and y_2's ARX MISO models would be sub-optimal if there are correlated parameters among the two MISO models. It would definitely be wiser to identify models for all the outputs simultaneously. Moreover, as is apparent, both FIR and ARX models employ a large number of parameters for MIMO case which makes model fitting prone to overfitting and necessitates large dataset. Fortunately, specialized techniques for MIMO SysID do exist. However, we will hold off our discussion of MIMO modeling techniques for a bit and instead look at other popular I/O modeling techniques in the next chapter that overcome the shortcomings of ARX models.

This completes our quick look at two of the most popular classical linear dynamic models. Negligible computational burden, global optimal solution, and only a little demand for prior process knowledge have contributed to these models' sustained popularity. By being exposed to their respective pros and cons, we hope that you have a good understanding of what you are signing up for (e.g., the implicit noise model in ARX model) when employing these models. If you find yourself in an inadvertent situation where ARX and FIR models fail to provide

`

satisfactory performances, then you might need to play with your noise model. The next chapter aims to equip you with the requisite tools necessary to accomplish this.

Summary

In this chapter we studied in detail the concepts behind FIR and ARX modeling and looked at the pros and cons of these models. Industrially relevant case-study was illustrated to showcase the implementation details. Next, we will move on to learn about more generic linear modeling framework (of which ARX and FIR models can be considered special cases) which will add some powerful techniques to your SysID arsenal.

`

Chapter 7

Input-Output Modeling - Part 2: Handling Process Noise the Right Way

In the previous chapter we saw that the FIR and high-order models can provide unbiased and linear-in-parameter models, but, unfortunately, suffer from high variance errors due to large number of parameters. Therefore, in this chapter, we will look at other popular model structures, namely, ARMAX, OE, and Box-Jenkins, that can provide more parsimonious description of the underlying process and generate unbiased models. These models allow greater flexibility in noise dynamics and capture the fact that noise may enter (or originate in) the system at different points which, in turn, determines the structure of the noise transfer operator. However, there is no free lunch! These models generate non-linear-in-parameter predictors and therefore the parameter estimation procedures are computationally more complex. Nonetheless, addition of these models to your ML-DPM toolkit will allow you to choose model structure specific to the problem at hand.

In the later part of this chapter, we will relax the stationarity conditions on disturbance signals and look at how to model processes with non-stationary disturbances. We will see how differencing input and output signals helps in removing disturbance non-stationarities. Using simulated illustrations, we will try to understand the implications of choosing these model structures. Specifically, we will study the following topics.

- Introduction to ARMAX and Box-Jenkins models
- Modeling distillation columns using ARMAX models
- Introduction to OE model structure and its comparison to ARX model structure
- Introduction to ARIMAX models
- Modeling gas furnace systems using Box-Jenkins models

`

7.1 PEM Models

In Chapter 6, we saw how the ARX model provides a very restrictive treatment of the process disturbances. We also saw how this shortcoming can lead to significant model bias. Accordingly, the focus in this chapter is to overcome this shortcoming and look at other popular alternatives to ARX model, namely, the output error (OE) model, the autoregressive moving average with eXogenous (ARMAX) inputs model, and the Box-Jenkins (BJ) model. As shown in Figure 7.1, these parametric models provide a broad spectrum of options regarding the parametrization of the noise transfer operator (*H*). For example, while disturbances are not modeled and only input transfer operator is sought in OE framework, completely independent parametrization of both *G* and *H* operators are allowed in the BJ model. In the next few sections, we will explore these models' properties and learn how to use them judiciously.

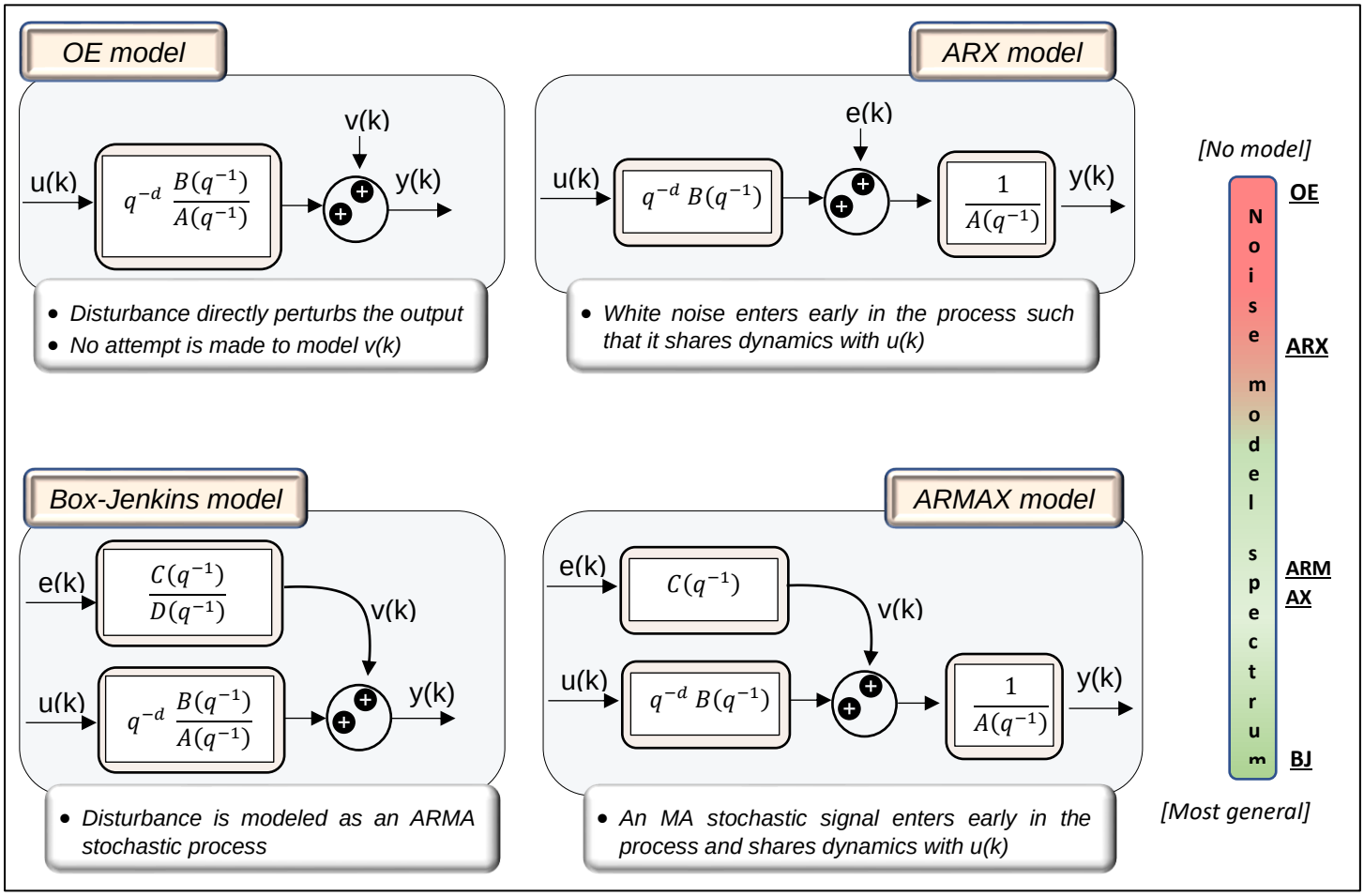


Figure 7.1: Model structure of PEM models

All the models in Figure 7.1 belong to the family of prediction error methods wherein the model parameters are estimated by minimizing the one-step ahead prediction errors ($\varepsilon(k)$) where,

`

$$\varepsilon(k) = y(k) - \hat{y}(k|k-1)$$

Optimal model prediction based on information up to time *k-1*

Using the transfer operator model form, an expression for $\hat{y}(k|k-1)$ in terms of past (input and output) measurements (and predictions in some cases) can be easily derived to give the following expression

$$\hat{y}(k) = [1 - H^{-1}]\, y(k) + H^{-1} G\, u(k) \qquad \text{eq. 1}$$

We will use Eq. 1[54] in the coming sections to derive the difference equation expressions for the model predictions.

PEM predictors

To derive the expression for PEM model predictions, we can follow the below steps.

$$\begin{aligned} & y(k) = G\, u(k) + H\, e(k) \\ \Rightarrow\ & H^{-1}\, y(k) = H^{-1} G\, u(k) + e(k) \\ \Rightarrow\ & y(k) + H^{-1}\, y(k) = y(k) + H^{-1} G\, u(k) + e(k) \\ \Rightarrow\ & y(k) = [1 - H^{-1}]\, y(k) + H^{-1} G\, u(k) + e(k) \end{aligned}$$

eq. 1 $\Rightarrow \hat{y}(k) = [1 - H^{-1}]\, y(k) + H^{-1} G\, u(k)$

Since *e(k)* is white noise, it's most probable value at time *k* is zero

Let's use the above predictor for ARX model where $G = \frac{q^{-d} B}{A}$ and $H = \frac{1}{A}$. Eq. 1 gives

$$\hat{y}(k) = [1 - A]\, y(k) + q^{-d} B\, u(k)$$

$$\begin{aligned} \Rightarrow\ \hat{y}(k) = & -a_1 y(k-1) - a_2 y(k-2) \cdots - a_{n_a} y(k-n_a) \\ & + b_1 u(k-d-1) + \cdots + b_{n_b} u(k-d-n_b) \end{aligned}$$

The above expression for ARX predictor could also have been obtained by simply setting *e(k)* to zero in the ARX model's difference equation.

[54] To be read as $\hat{y}(k) = [1 - H^{-1}(q^{-1})]\, y(k) + H^{-1}(q^{-1})\, G(q^{-1})\, U(k)$. We will skip the q notation in certain expressions for presentation clarity.

`

7.2 ARMAX Models: An Introduction

Autoregressive moving average with eXogenous inputs (ARMAX) model is extension of ARMA time series model to I/O systems. As shown in Figure 7.2, the current output value is related to the past measurements of inputs and outputs, and the past noise values as well! With ARMA disturbances, greater flexibility is provided to the noise model (compared to ARX) and therefore, ARMAX is a popular model for industrial processes. Note that if n_c= 0, then ARMAX model reduces to an ARX model, and if additionally, n_a= 0, then a FIR model results. Also note that the *G* and *H* operators are not independently parametrized due to the common denominator polynomial, and therefore, the potential for model bias (due to inaccurate noise model) remains.

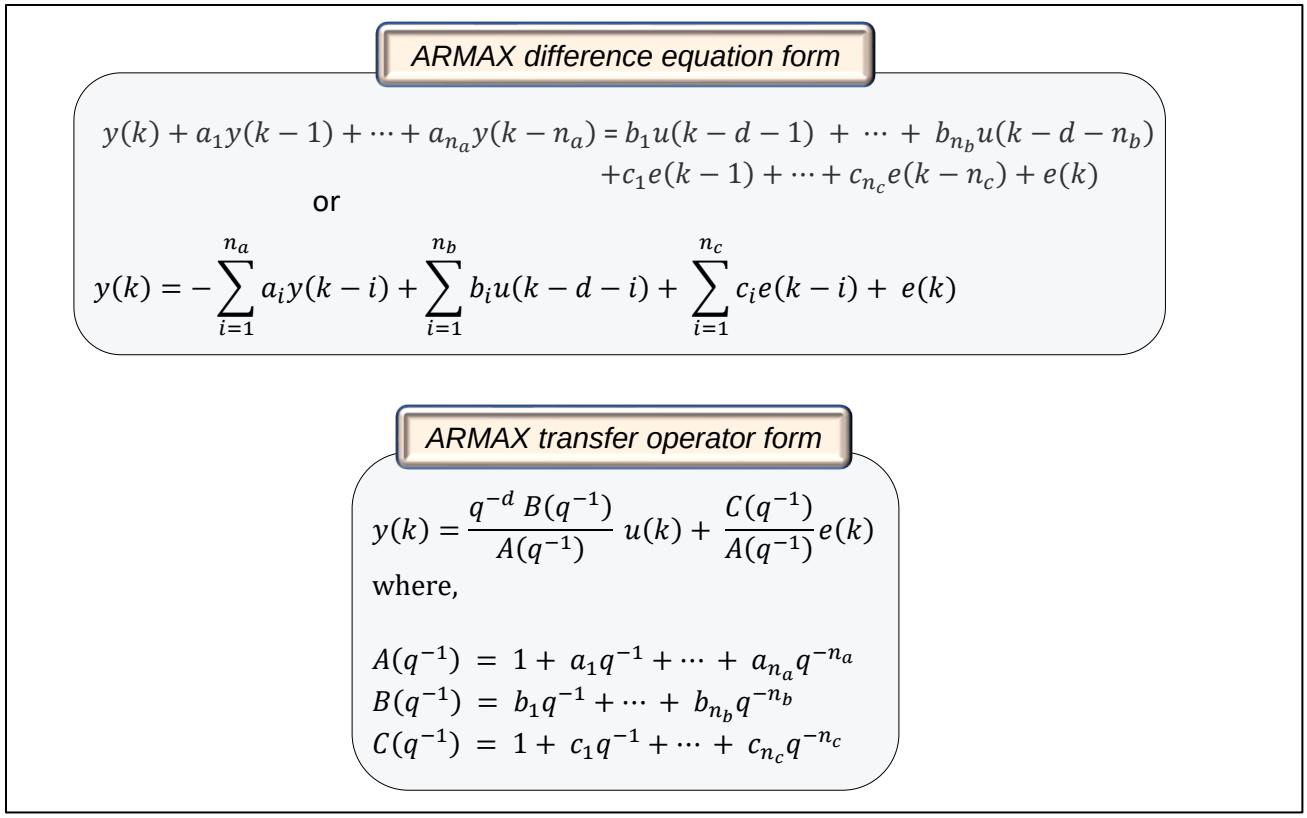


Figure 7.2: ARXMAX (n_a, n_b, n_c) with delay d model structure for SISO system

`

ARMAX predictor

To derive an expression for the 1-step ahead prediction, we may simply replace the unpredictable part *e(k)* in the difference equation form of ARMAX to zero and get the following

$$\hat{y}(k) = -\sum_{i=1}^{n_a} a_i y(k-i) + \sum_{i=1}^{n_b} b_i u(k-d-i) + \sum_{i=1}^{n_c} c_i e(k-i) \qquad \text{eq. 2}$$

Alternatively, Eq. 1 can be used as well:

$$\hat{y}(k) = \left[1 - \frac{A(q^{-1})}{C(q^{-1})}\right] y(k) + \frac{A(q^{-1})}{C(q^{-1})} \frac{q^{-d}\, B(q^{-1})}{A(q^{-1})}\, u(k)$$

$$\Rightarrow C(q^{-1})\hat{y}(k) = [C(q^{-1}) - A(q^{-1})]\, y(k) + \; q^{-d}\, B(q^{-1}) u(k)$$

$$\Rightarrow \hat{y}(k) = \sum_{i=1}^{n_c} c_i y(k-i) \; - \sum_{i=1}^{n_a} a_i y(k-i) - \sum_{i=1}^{n_c} c_i \hat{y}(k-i) + \sum_{i=1}^{n_b} b_i u(k-d-i)$$

Current prediction is a function of past predictions as well

The above expression can also be derived from Eq. 2 by substituting $e(k) = y(k) - \hat{y}(k)$. Another noteworthy point is that the ARMAX predictor is nonlinear-in-unknowns during model fitting due to the product terms in $\sum_{i=1}^{n_c} c_i \hat{y}(k-i)$ and therefore, nonlinear optimization is needed for parameter estimation.

`

7.3 ARMAX Modeling of Distillation Columns

To showcase an industrial application of ARMAX models, we will use a simulated dataset from a distillation column. The ubiquity and importance of distillation columns in process industry cannot be overstated. Figure 7.3 shows a schematic of one such column wherein the incoming feed is separated into a top product (distillate) and a bottoms product. Strict control of the product flows and concentrations is critical for optimal column operations and the heating power (*QB*) supplied to the reboiler which drives the vapor flow in the column is a commonly used manipulated variable. Therefore, we will attempt to build a dynamic model between the distillate concentration (X_D) and *QB* to help the column controller operate the unit efficiently. To obtain the model, we will use data provided in the file *DistillationColumn_SNR10.csv* which contains 1000 samples of *QB* and X_D obtained from an open-loop simulation of the system wherein *QB* was excited using a GBN signal.[55]

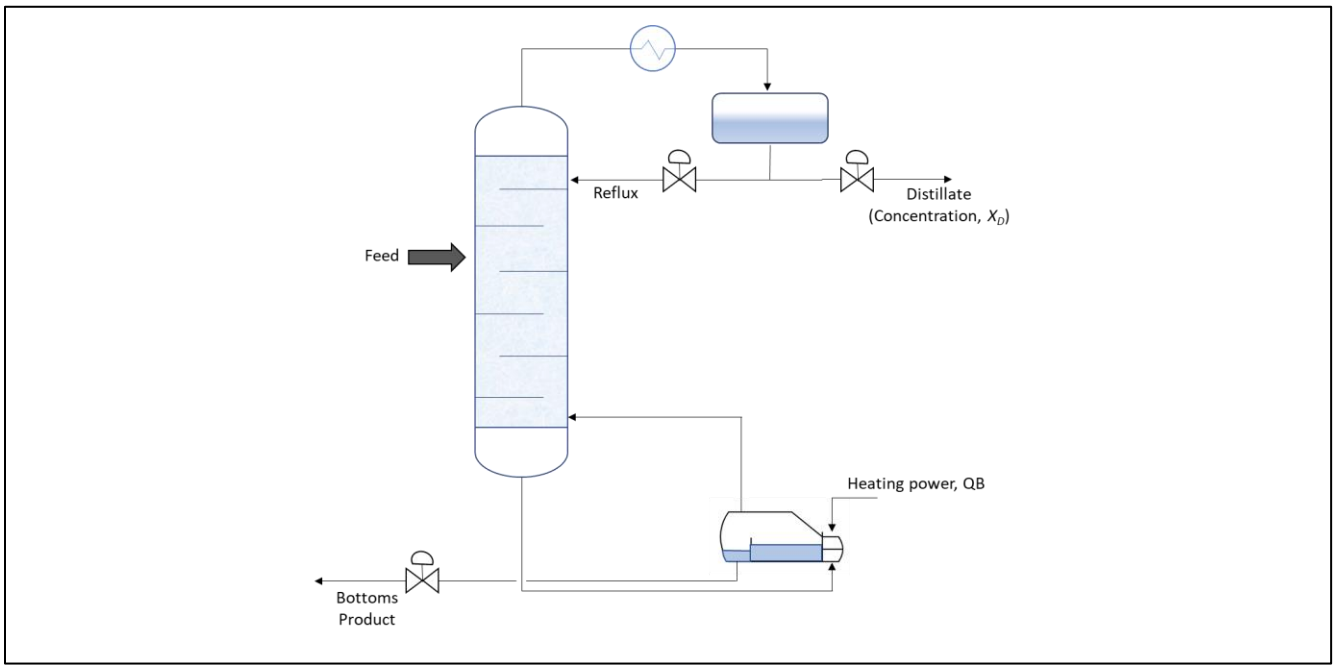


Figure 7.3: Representative schematic of a distillation column used in process industry

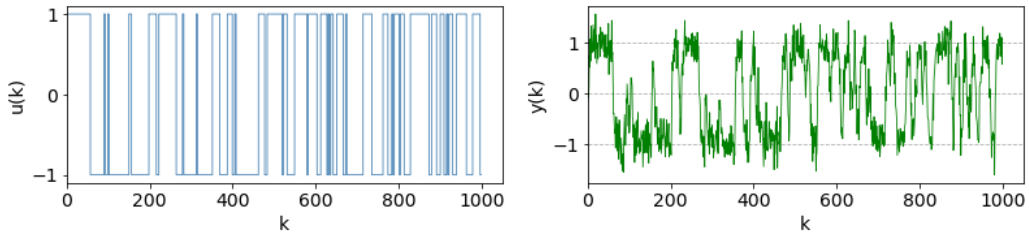
Let's start by importing the required packages and exploring the provided I/O dataset.

```
# import packages
import matplotlib.pyplot as plt, numpy as np, control
```

[55] The model for the distillation column was taken from '*Digital control systems: design, identification and implementation. Springer, 2006*'.

`

```
from sklearn.preprocessing import StandardScaler
from sippy import system_identification as SysID
from statsmodels.graphics.tsaplots import plot_acf
from statsmodels.tsa.stattools import ccf

# read distillation column data and plot
data = np.loadtxt(DistillationColumn_SNR10.csv', delimiter=',')
u = data[:,0, None] # QB
y = data[:,1, None] # XD

plt.figure(), plt.plot(u, 'steelblue'), plt.ylabel('u(k)'), plt.xlabel('k')
plt.figure(), plt.plot(y, 'g'), plt.ylabel('y(k)'), plt.xlabel('k')
```



Next, we will center the data and fit an ARX model using AIC.

```
# center data and fit ARX model
u_scaler = StandardScaler(with_std=False); u_centered = u_scaler.fit_transform(u)
y_scaler = StandardScaler(with_std=False); y_centered = y_scaler.fit_transform(y)

ARXmodel = SysID(y_centered, u_centered, 'ARX', IC='AIC', na_ord=[1,10], nb_ord=[1,10], delays=[0,2])
# delay range obtained from FIR model

>>> suggested orders are: Na= 5 ; Nb= 5 Delay:  1
```

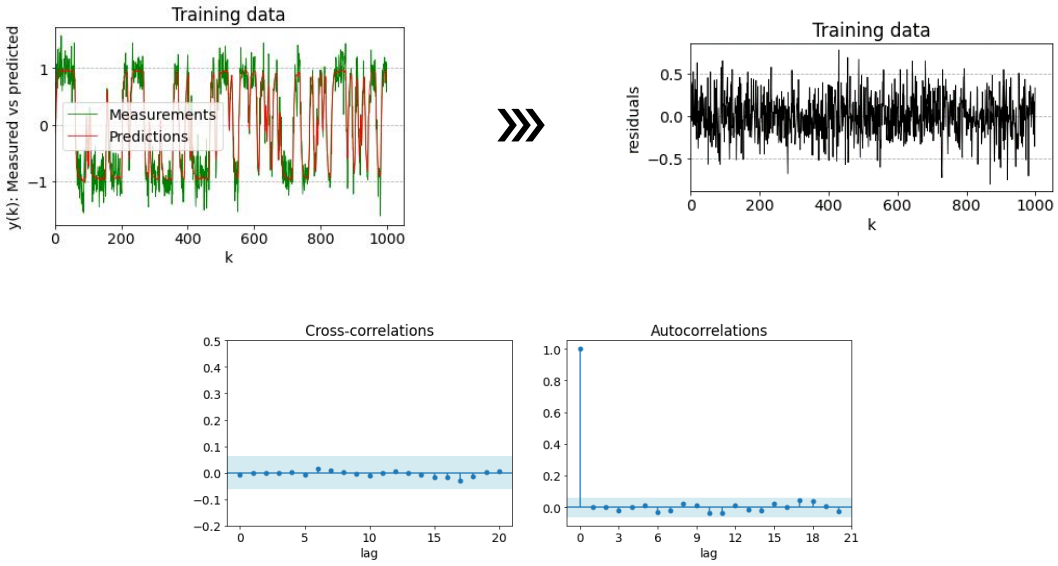
The obtained ARX model is of pretty high order. We have studied previously that high-order ARX models become necessary to negate the impact of inaccurate specification of disturbance dynamics. Therefore, it is reasonable to attempt an ARMAX model with the hope of obtaining a more parsimonious model.

```
# fit ARMAX model
ARMAXmodel = SysID(y_centered, u_centered, 'ARMAX', IC='AIC', na_ord=[1,10], nb_ord=[1,10],
nc_ord=[1,10], delays=[0,2], max_iterations=1000)
```

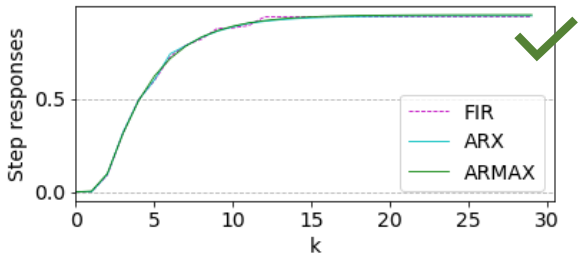
`

>>> Armax model:
- Params:
  na: 2 (1, 10)
  nb: 3 (1, 10)
  nc: 2 (1, 10)
  delay: 0 (0, 2)

As hoped, the ARMAX model's size is reasonable. A quick residual check (plots shown below) build some confidence into the model.



To build further confidence into the model, we check the step responses from FIR, ARX, and the ARMX models. The close match of the step responses further validates the model.



Step responses from different models match.

`

7.4 OE Models: An Introduction

Output error (OE) models lie at the extreme end of PEM modeling spectrum wherein no model is postulated for the disturbance signal and only the effects of inputs on the output variable are modeled as shown in Figure 7.4. Accordingly, *v(k)* is completely ignored for model predictions. Since no past output information is used and past predictions are used for future predictions, one-step ahead prediction equals *m*-step ahead predictions (for any *m*) and infinite-step prediction as well. In other words, simulation and prediction responses coincide for OE model![56] Figure 7.4 also tells us that the predictor is nonlinear-in-unknowns during model fitting and therefore, like ARMAX[57], nonlinear optimization is required for parameter optimization.

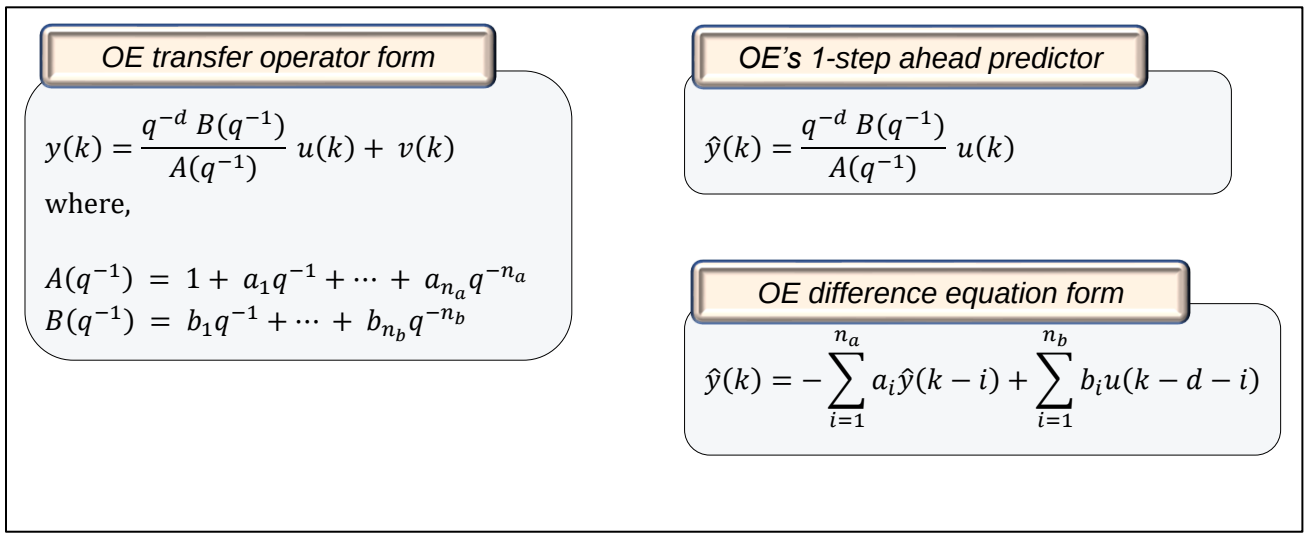


Figure 7.4: OE model structure and one-step ahead predictor for SISO system

There is no issue of parametrization dependency between *G* and *H* operators and therefore, OE models generate unbiased deterministic models. Therefore, when one is not sure of noise characteristics, OE models are preferred to provide unbiased input transfer operator. The corresponding residuals ($y(k) - \hat{y}_{OE}(k)$) are of course not bound to satisfy whiteness conditions and therefore, ACF test is not imposed during model diagnostics; CCF test, however, would still apply. Very often, a time-series model is fit to the residual sequence to approximate the noise transfer operator as shown in Figure 7.5 below. You can take the approximated model as your final model or use this as an initial estimate in a Box-Jenkins model estimation procedure to obtain a better model.

---

[56] For any PEM model, $\hat{y}(k) = \frac{B(q^{-1})}{A(q^{-1})}u(k)$ gives infinite-step predictions/simulation responses.

[57] If *v(k) = e(k)*, then OE becomes special case of ARMAX model with $C(q^{-1}) = A(q^{-1})$

`

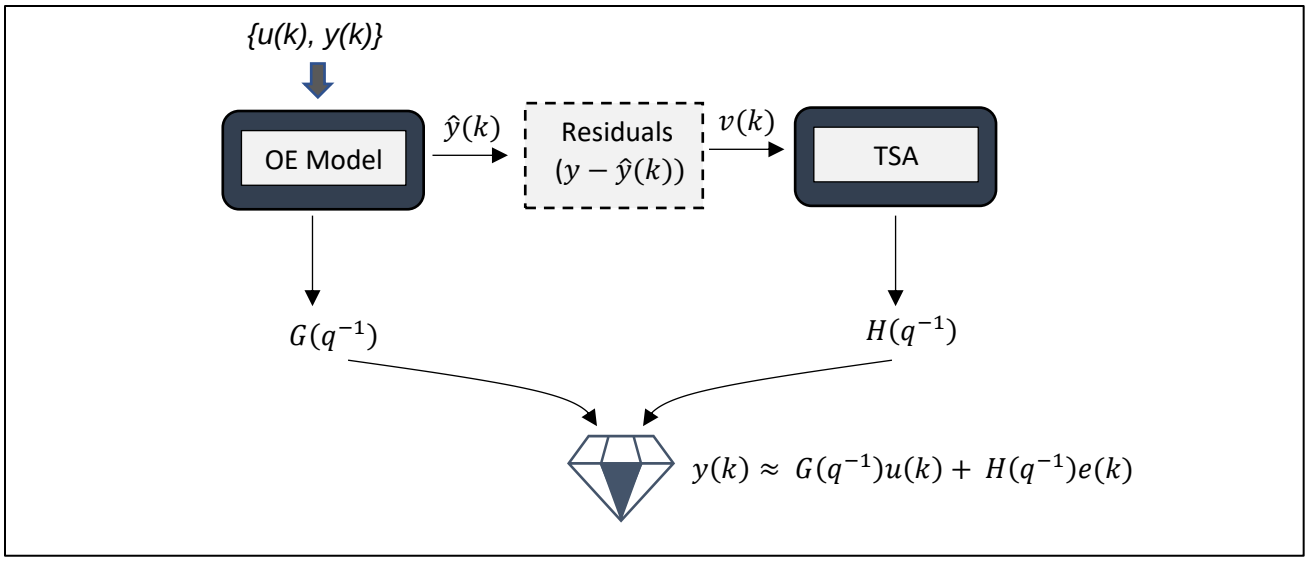


Figure 7.5: Approximating a process model through separate estimation of deterministic and stochastic components.

Example 7.1:

In example 6.2 in the previous chapter, we provided a comparison between ARX and OE model performances for modeling the below process. In this example, we will see how those OE models were fitted.

$$y(k) = \frac{0.7q^{-1}}{1 - 0.7q^{-1}}u(k) + \frac{1}{1 - 0.2q^{-1}}e(k)$$

We will also see how the overall process model can be obtained through separate estimation of deterministic and stochastic components. The code for importing packages and reading and pe-processing the dataset remains the same as in Example 6.2. We will use the dataset with SNR = 10.

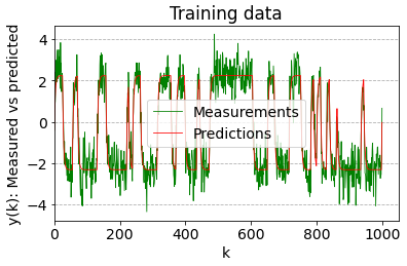
```
# fit ARX model
OEmodel = SysID(y_centered, u_centered, 'OE', IC='AIC', nb_ord=[1,5], nf_ord=[1,5], delays=[0,5])
# SIPPY uses the symbol F to denote denominator polynomial of transfer operator G
print(OEmodel.G)
```

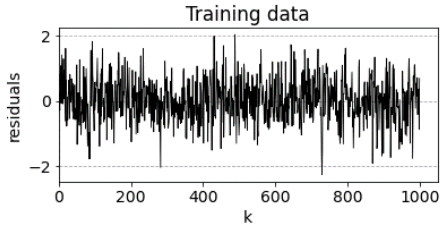
$$G(q^{-1}) = \frac{0.6704q^{-1}}{1 - 0.7052q^{-1}}$$

OE model approximated the deterministic component much better than the ARX model and therefore, it is not surprising that its step response was closer to the true step response. Let's see if the residuals pass the diagnostics check. As alluded to before, OE models need not be tested for whiteness of residuals.

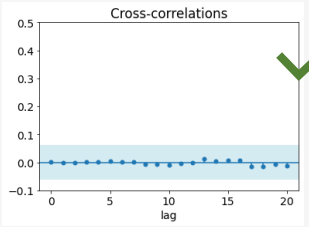
`

```
# get model predictions and residuals on training dataset
y_predicted_centered = OEmodel.Yid
y_predicted = np.transpose(y_scaler.inverse_transform(y_predicted_centered))
residuals = y - y_predicted
```



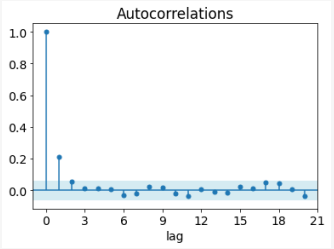


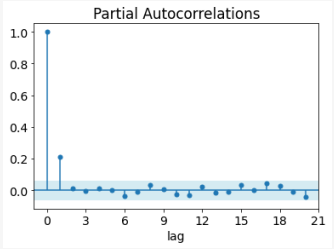

```
# CCF b/w residuals and input sequence
ccf_vals = ccf(residuals, u, adjusted=False) # ccf for lag >= 0
ccf_vals = ccf_vals[:21] # ccf for lag 0 to 20

# plot
plt.figure(figsize=(6,4)), plt.vlines(lags, [0], ccf_vals), plt.axhline(0, 0, lags[-1])
plt.plot(lags, ccf_vals, marker='o', markersize=5, linestyle='None')
plt.gca().axhspan(-conf_int, conf_int, facecolor='lightblue', alpha=0.5) # shaded confidence interval
plt.xlabel('lag'), plt.title('Cross-correlations')
```



No correlations exist between the residual and the input sequences

Alright, now that we have successfully obtained the deterministic component, let's see if the model residuals can help us obtain the stochastic component. For this, we will need to fit a time-series model to the residuals as shown in Figure 7.5 and we already know that ACF/PACF plots can provide clues about the candidate models.





The exponential decay of ACF and sharp cut-off of PACF at lag *1* suggests an AR(1) model which is consistent with the true process!

`

7.5 Box-Jenkins Models: An Introduction

Box-Jenkins (BJ) model relaxes the restriction of (common $A(q^{-1})$ polynomial in G and H operators in) ARMAX model and uses disturbance signal dynamics completely different from the deterministic dynamics. As shown in Figure 7.6, G and H operators are independently parameterized which ensures unbiased deterministic model estimation. Expectedly, the model expressions are more complex and requires specification of 5 hyperparameters, vis-a-vis, n_a, n_b, n_c, n_d, d. Moreover, nonlinear optimization is required for model fitting. Let's see a quick example of modeling an industrial gas-furnace system via BJ model.

BJ transfer operator form

$$y(k) = \frac{q^{-d}\,B(q^{-1})}{A(q^{-1})}\,u(k) + \frac{C(q^{-1})}{D(q^{-1})}\,e(k)$$

where,

$$A(q^{-1}) = 1 + a_1 q^{-1} + \cdots + a_{n_a} q^{-n_a}$$
$$B(q^{-1}) = b_1 q^{-1} + \cdots + b_{n_b} q^{-n_b}$$
$$C(q^{-1}) = 1 + c_1 q^{-1} + \cdots + c_{n_c} q^{-n_c}$$
$$D(q^{-1}) = 1 + d_1 q^{-1} + \cdots + d_{n_d} q^{-n_d}$$

BJ's 1-step ahead predictor

$$\hat{y}(k) = \left[1 - \frac{D(q^{-1})}{C(q^{-1})}\right] y(k) + \frac{q^{-d}\,D(q^{-1})B(q^{-1})}{C(q^{-1})A(q^{-1})}\,u(k)$$

BJ difference equation form

To see the difference equation form, let's consider the following for convenience

$$A(q^{-1}) = 1 + a_1 q^{-1},\ B(q^{-1}) = b_1 q^{-1}, C(q^{-1}) = 1 + c_1 q^{-1}, D(q^{-1}) = 1 + d_1 q^{-1}$$
$$d = 0$$

Then,

$$y(k) = \frac{b_1 q^{-1}}{1 + a_1 q^{-1}}\,u(k) + \frac{1 + c_1 q^{-1}}{1 + d_1 q^{-1}}\,e(k)$$

$$\Rightarrow y(k) + (a_1 + d_1)y(k-1) + a_1 d_1 y(k-2) = b_1 u(k-1) + b_1 d_1 u(k-2) + (a_1 + c_1)e(k-1) + a_1 c_1 e(k-2) + e(k)$$

Figure 7.6: Box-Jenkins model structure for SISO system

`

Example 7.2:

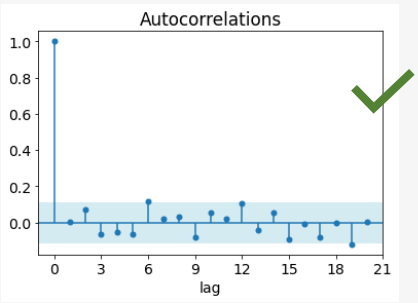
In this illustration, we will use the popular Box-Jenkins gas furnace dataset (https://www.itl.nist.gov/div898/handbook/pmc/section4/pmc451.htm). The data comes from a SISO gas furnace wherein methane was combusted. The methane feed rate is the input, the CO_2 concentration in the gases exiting the furnace is the output, and 296 samples recorded at 9-second intervals are provided. We will build a BJ model for this dataset using SIPPY. The code is almost identical to that in the previous example.

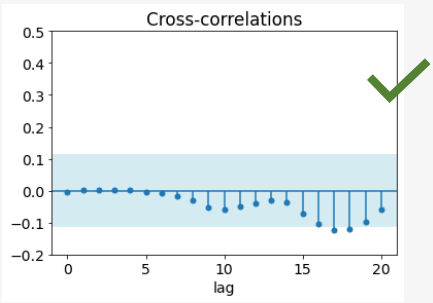
```
# read data
data = np.loadtxt('gas-furnace.csv', delimiter=',')
u = data[:,0, None]; y = data[:,1, None]

# center data
u_scaler = StandardScaler(with_std=False); u_centered = u_scaler.fit_transform(u)
y_scaler = StandardScaler(with_std=False); y_centered = y_scaler.fit_transform(y)

# fit BJ model
BJmodel = SysID(y_centered, u_centered, 'BJ', IC='AIC', nb_ord=[3,3], nc_ord=[1,1], nd_ord=[2,2],
                                                    nf_ord=[2,2], delays=[2,2], max_iterations=500)
# see SIPPY user-guide for details on symbols used for transfer operator polynomials
print(BJmodel.G)
print(BJmodel.H)
```

$$G(q^{-1}) = \frac{q^{-2}(-0.53q^{-1} - 0.38q^{-2} - 0.51q^{-3})}{1 - 0.56q^{-1} + 0.01q^{-2}}$$

$$H(q^{-1}) = \frac{1 + 0.05q^{-1})}{1 - 1.50q^{-1} + 0.60q^{-2}}$$

The obtained polynomials of the transfer operators are very close to those reported in the original source (*Box et al., Time Series Analysis: Forecasting and Control, Wiley*). The above model also (almost) passes the residual diagnostics check as shown below by the ACF and CCF plots of the residuals.





`

7.6 ARIMAX Models: An Introduction

So far we have worked under the assumption that the process disturbances (*v(k)*) are stationary. However, this may not always be true. Nonetheless, if *v(k)* is difference stationary, then it can be modeled as an ARIMA process: $v(k) = \frac{C(q^{-1})}{(1-q^{-1})^d A(q^{-1})}\, e(k)$ where *d* is the differencing order[58], and the final model looks as shown in Figure 7.7.

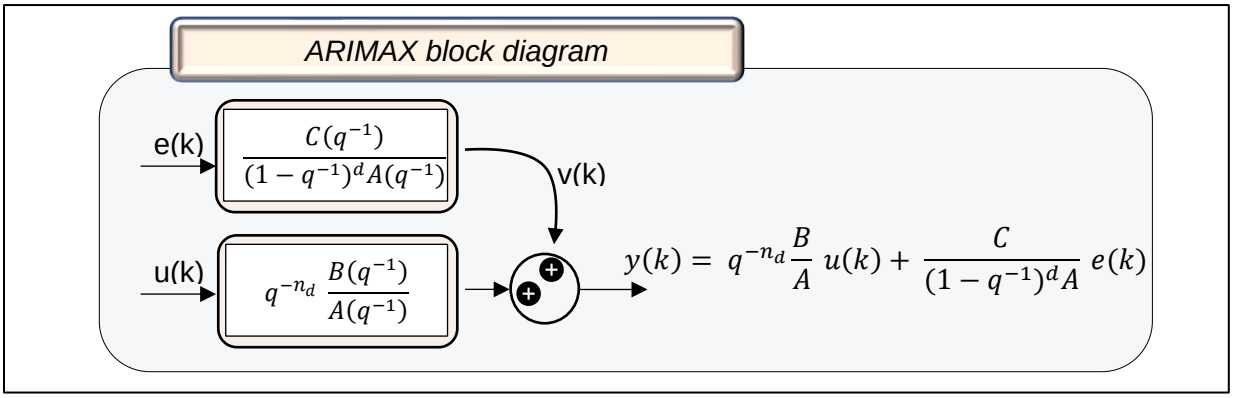


Figure 7.7: Model structure of ARIMAX. Note that here *d* denotes the differencing order and not the process delay.

Let's do some quick algebraic manipulation to understand the implications of employing this model.

ARIMAX model: $y(k) = q^{-n_d}\frac{B(q^{-1})}{A(q^{-1})}u(k) + \frac{C(q^{-1})}{(1-q^{-1})^d A(q^{-1})}e(k)$

$$\Rightarrow (1-q^{-1})^d y(k) = q^{-n_d}\frac{B(q^{-1})}{A(q^{-1})}(1-q^{-1})^d u(k) + \frac{C(q^{-1})}{A(q^{-1})}e(k)$$

$$\Rightarrow \tilde{y}(k) = q^{-n_d}\frac{B(q^{-1})}{A(q^{-1})}\tilde{u}(k) + \frac{C(q^{-1})}{A(q^{-1})}e(k)$$

ARMAX model!

where $\tilde{y}(k) = (1-q^{-1})^d y(k)$

$\tilde{u}(k) = (1-q^{-1})^d u(k)$

[58] As remarked in Chapter 5, *d* is usually ≤ 2

`

The expression above shows that ARIMAX model is equivalent to ARMAX model on differenced input and output signals! Moreover, getting ARMAX model on differenced data provides an unbiased estimate of the deterministic model or the input transfer operator *G* between *u(k)* and *y(k)*. Like ARIMA models, ARIMAX should be used with caution or only when necessary. If *v(k)* does not have any integration non-stationarity, then differencing *u(k)* and *y(k)* will only amplify noise and reduce SNR. One way to ascertain disturbance non-stationarity is to fit an OE model on undifferenced data and then assess the properties of the residuals (via ACF/PACF or unit root tests). Let's work out an example to understand the methodology better.

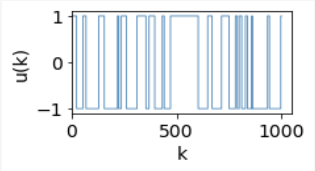
Example 7.3:

In this example, we will re-use the true process from Example 7.1 with a modification of the AR coefficient in the disturbance model that now has an integrating term as well as shown below.

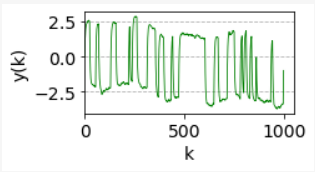
$$y(k) = \frac{0.7q^{-1}}{1-0.7q^{-1}}u(k) + \frac{1}{(1-q^{-1})(1-0.7q^{-1})}e(k)$$

We will see how the true deterministic component can be recovered by working with differenced input and output signals.

```
# import packages
import matplotlib.pyplot as plt, numpy as np, control
from sklearn.preprocessing import StandardScaler
from sippy import system_identification as SysID
from statsmodels.graphics.tsaplots import plot_acf
```



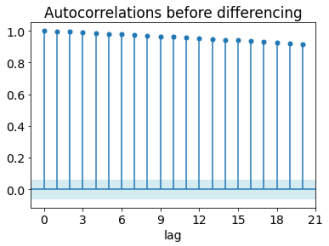

```
# read data
data = np.loadtxt('ARIMAX_illustrate_SNR15.csv', delimiter=',')
u = data[:,0, None]; y = data[:,1, None]
```



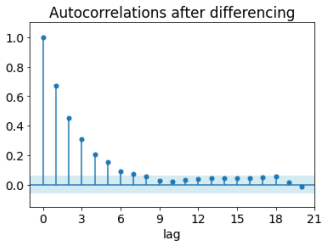

```
# center data
u_scaler = StandardScaler(with_std=False); u_centered = u_scaler.fit_transform(u)
y_scaler = StandardScaler(with_std=False); y_centered = y_scaler.fit_transform(y)

#%% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
#       Fit OE model and check residual non-stationarity
#% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
OEmodel = SysID(y_centered, u_centered, 'OE', OE_orders=[1,1,0])
y_predicted_OE = y_scaler.inverse_transform(np.transpose(OEmodel.Yid))
OEresiduals = y - y_predicted_OE
```

`

The ACF plots below show the autocorrelation of the residual sequence before and after differencing. It is clear that the process is impacted by non-stationary disturbance and the non-stationarity goes away after the first differencing of the residuals. Guided by these observations, we will build an ARMAX model after differencing the input and output signals once.

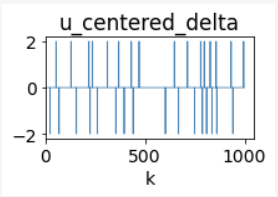


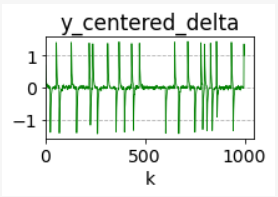
plot_acf(OEresiduals)



plot_acf(np.diff(OEresiduals, axis=0))

```
#%% %%%%%%%%%%%%%%%%%%%%%%%%
#                 Fit ARIMAX model
#% %%%%%%%%%%%%%%%%%%%%%%%%%
# difference I/O data once
u_centered_delta = np.diff(u_centered, axis=0)
y_centered_delta = np.diff(y_centered, axis=0)
```






```
# fit ARMAX model on differenced data
ARMAXmodel = SysID(y_centered_delta, u_centered_delta, 'ARMAX', ARMAX_orders=[1,1,0,0])
print(ARMAXmodel.G)
print(ARMAXmodel.H)
```

$$G(q^{-1}) = \frac{0.6999q^{-1}}{1-0.6988q^{-1}} \quad ; H(q^{-1}) = \frac{1}{1-0.6988q^{-1}}$$

$$\Delta y(k) = \frac{0.6999q^{-1}}{1-0.6988q^{-1}}\Delta u(k) + \frac{1}{1-0.6988q^{-1}}e(k)$$

$$y(k) = \frac{0.6999q^{-1}}{1-0.6988q^{-1}}u(k) + \frac{1}{(1-q^{-1})(1-0.6988q^{-1})}e(k)$$

We have obtained a very close estimate of the true process! The differenced input and output sequences may seem very different, but as shown in this example, they obey the same deterministic relationship.

`

What model structure to choose?

With quite a few models at your disposal, you have all the rights to ask, 'how to choose the best model for my system?' Unfortunately, there is no definite answer. However, the concepts covered in the current and the previous chapters can help you choose the right path to your answer and, most importantly, understand the implications of choosing any model structure.

A trivial approach may be to try and fit a generic ARIMAX model, but this won't be a wise approach because you may end up with nonexistent dynamic in your G and H operators due to noisy data. Obtaining the most simple and parsimonious model should be your guiding mantra! In general, the selection of model structure will largely depend on the extent of prior information/insights you have about the system, specifically about the disturbance signals such as if the noise enters early or late in the process or if the disturbances have drift tendencies. Other major selection criterion is the end goal: you may be interested only in the deterministic model (to be used for simulation for example) or also in the noise model (for better predictions). Of course, SNR, amount of available data, and ease of estimation are also crucial determining factors.

One school of thought recommends employing an OE model to estimate G, studying the residual characteristics to approximate H, and then using these estimates as initial guess for a BJ model for obtaining improved estimates. In another school of thought, ARX is the preferred initial model choice.

Do note that we remained focused on SISO systems in Chapters 6 and 7. You can imagine that it would be quite a tedious task to find the optimal combination of model orders for the polynomials for all the G and H operators for all the input and outputs in a MIMO model. The state-space models introduced in the next chapter will make your life very easy for modeling MIMO systems as you won't need to pre-specify all those model orders. Therefore, hold your horses tight, more modeling options are coming your way!

`

Summary

In this chapter we looked at classical PEM models for modeling linear SISO systems. We learnt the pros and cons of ARMAX, OE, BJ, and ARIMAX models. We saw their applications for modeling distillation columns and gas furnaces. You now have all the tools available to make educated selection of your model structure. To further strengthen your modeling capabilities, let's move on to the study of state-space and subspace models for MIMO systems.

`

Chapter 8

State-Space Models: Efficient Modeling of MIMO Systems

In the previous chapter we alluded to the problems related to the classical methodology of modeling a MIMO system through separate MISO models for each output. In a physical system, the different output variables seldom behave independently; they often share common parameters and/or correlated noise. Therefore, identifying all the output models simultaneously while being cognizant of shared dynamics between them leads to better and more robust models. This alternative approach to SysID is provided by state-space (SS) models. State-space models add an intermediate layer of 'state variables' between the input and output variables, wherein, the inputs determine the states, and the outputs are derived as linear combination of the states. This modeling paradigm leads to parsimonious models capable of fitting complex processes with both fast and slow dynamics.

In this chapter, we will cover in detail the subspace identification techniques which are utilized for fitting the SS models. While several popular subspace identification methods (SIMs) such as N4SID, CVA, MOESP exist, CVA will be our primary focus. Subspace identification offers several additional advantages over I/O models. SIMs involve only linear algebra for model ID (no iterative nonlinear optimization schemes are needed). SIMs also provide in-situ mechanism for automated optimal model order selection. Subspace models are more robust to noise and provide smooth (step and impulse) response curves. Infact, it is not uncommon to fit SIM model to I/O data and then derive FIR model using the SIM model for usage in MPC applications. Due to such superior properties of SIM, they are among the 'mainstream' models provided by advanced process control (APC) solution vendors.

CVA models are also inherently suitable for computing fault detection indices, performing fault diagnosis, and therefore building process monitoring tools. We will build one such application in this chapter. Specifically, the following topics will be covered

- Introduction to state-space models and CVA modeling
- Modeling MIMO glass furnace via CVA
- Process monitoring of Industrial Chemical Plants using CVA

`

8.1 State-Space Models: An Introduction

Consider the distillation column in Figure 8.1 where we have four inputs and four outputs. Let's focus on the relationship between the reboiler duty Q_B and distillate composition x_D. We can build an I/O model between Q_B and x_D, but it is clear that Q_B doesn't affect x_D directly. There are internal dynamics or states (e.g., arising from the liquid holdup at each column tray) that changes in Q_B have to pass through before impacting x_D. Similar argument could be made for Q_B vs D relationship. The unobserved intermediate variables which characterize the internal dynamics or 'states' of the column are termed state variables and, as argued before, the outputs share these states. The modeling framework that reflects such interrelationship between the inputs and outputs is the state-space model whose structure[59] is shown in Figure 8.1. In a generic modeling exercise, the estimated state variables may not always have any physical meanings and even the choice of the states is not unique.

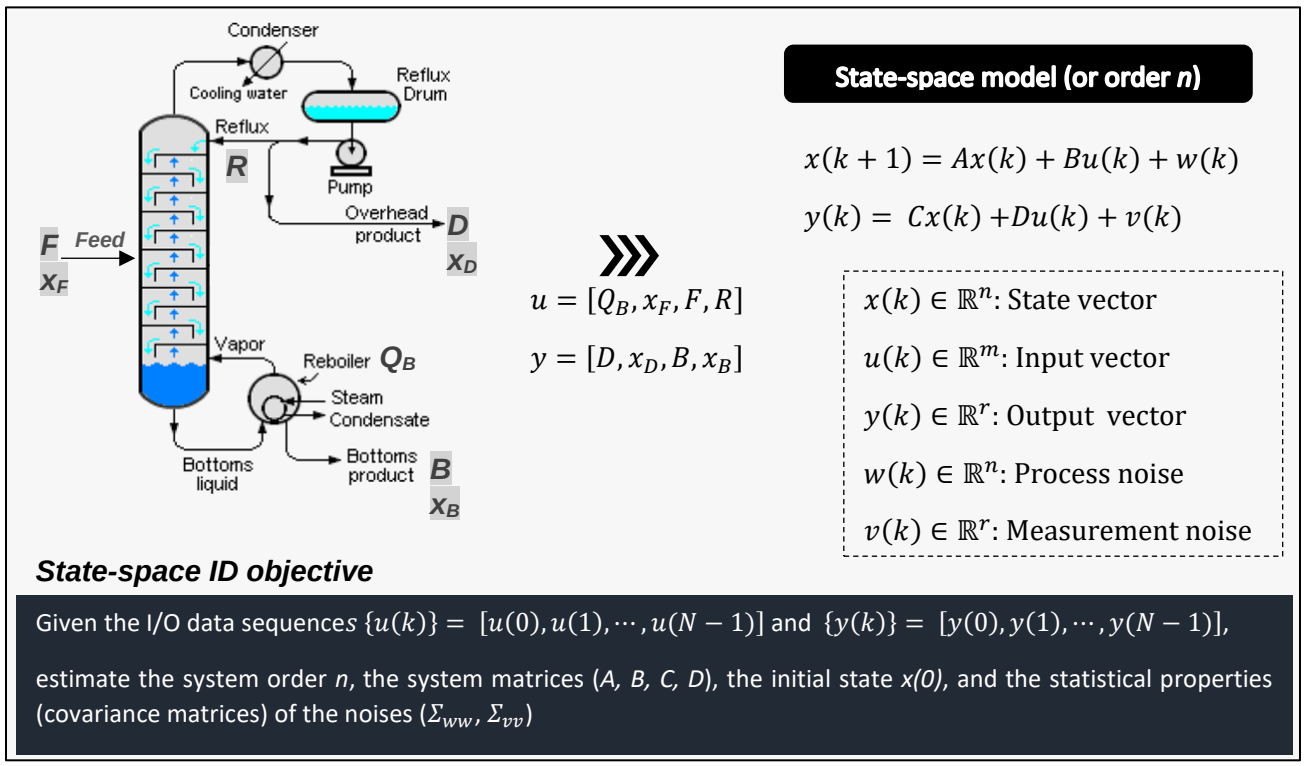


Figure 8.1: State-space representation of a process[60]

Our familiar PEM methodology could be used to estimate SS models but that would involve solving a nonlinear optimization problem. The better alternative is the subspace identification method.[61] Specifically, in the CVA (canonical variate analysis) approach, the states are

[59] The matrix *D* is often kept equal to zero due to no immediate direct impact of *u(k)* on *y(k)*

[60] Column diagram created by Milton Beychok under Creative Commons Attribution-Share Alike 3.0 [http://commons.wikimedia.org/wiki/File:Continuous_Binary_Fractional_Distillation.PNG]

[61] A drawback of subspace identification is that the model fit may not be as accurate as that obtained via PEM because optimality is not guaranteed in SIM. However, the difference is accuracy is usually not significant.

`

estimated via linear algebra as a linear combination of past outputs and inputs that maximizes the explained variability in the future output measurements. Without further ado, let's now delve deeper into the CVA concepts and master the art of CVA-based state-space modeling.

Forms of state-space models

The form of state-space model we saw in Figure 8.1 is called process form. Two other common forms are shown below

Innovation form

$$x(k+1) = Ax(k) + Bu(k) + Ke(k)$$
$$y(k) = Cx(k) + Du(k) + e(k)$$

Kalman gain

Predictor form

$$x(k+1) = A_k x(k) + B_k u(k) + Ky(k)$$
$$y(k) = Cx(k) + Du(k) + e(k)$$

where $A_k = A - KC$
$B_k = B - KD$

Any of the three forms can be chosen to model any MIMO system.

8.2 State-Space Modeling via CVA

CVA is among the most popular technique for state-space modeling of dynamic MIMO systems. Figure 8.2 shows the general approach used by CVA (and other SIM techniques) for estimating the SS models using historical plant data: the state sequence {*x(k)*} for the historical observations are first estimated using the {*y(k), u(k)*} data sequences, then the system matrices (*A, B, C, D*) are estimated via least-squares regression, and thereafter the covariance matrices are found. We will see more on this procedure soon. All SIM algorithms follow this procedure and differ primarily in the way the state sequence is estimated.

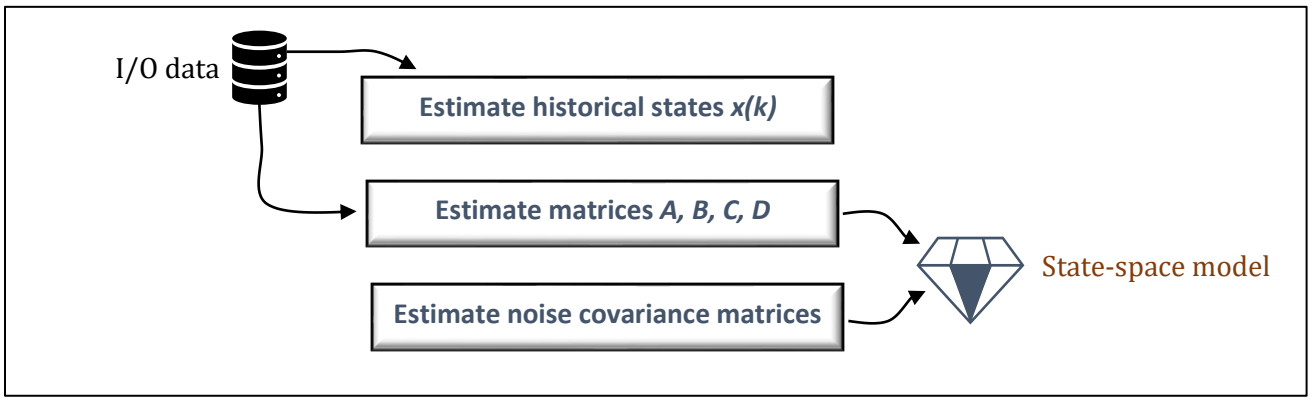


Figure 8.2: State-space modeling workflow

`

Let's look at the model fitting procedure in more details.

Mathematical background

Using the state-space model and some smart re-arrangement[62] of input-output data sequence, it can be shown that state vector *x(k)* can be approximated as a linear combination of past inputs and outputs[63]

unknown matrix

$$\hat{x}(k) = J\,p(k) \qquad \text{eq. 1}$$

defined as $\left[y^T(k-1)\ y^T(k-2)\ \cdots\ y^T(k-l_y)\ u^T(k-1)\ u^T(k-2)\ \cdots\ u^T(k-l_u)\right]^T$

In Eq. 1, l_y and l_u denote the number of lagged outputs and inputs used to build the *p(k)* vector. Usually, $l_y = l_u = l$ and it is a model hyperparameter. The CVA method utilizes CCA (canonical covariate analysis), a multivariate statistical technique, to find *J*. A quick primer on the CCA technique is provided below.

CCA: A quick primer

CCA is a dimensionality reduction technique which aims to find linear combinations of input variables ($u \in \mathbb{R}^m$) that are maximally correlated with linear combinations of outputs variables ($y \in \mathbb{R}^p$)*. You may relate this to PLS wherein covariance between latent variables is maximized. These linear combinations are called canonical variables and are given as

Observations in respective latent spaces

$c = Ju$ ← *c* comprises of uncorrelated latent variables

$d = Ly$ ← *d* comprises of uncorrelated latent variables

The covariance matrix between *c* and *d* is (rectangular) diagonal

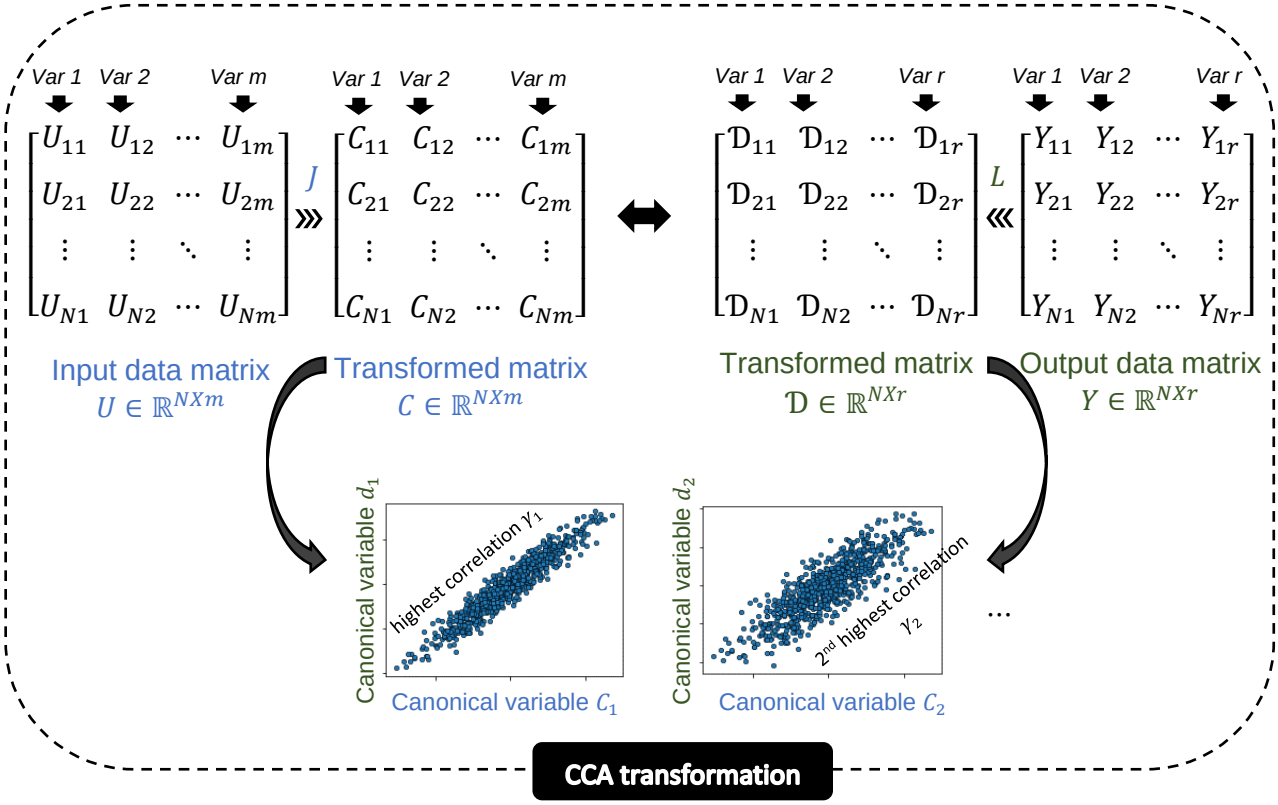
latent variables are only pairwise correlated

$$\Sigma_{cd} = J\Sigma_{uy}L^T = D = diag(\gamma_1, \gamma_2, \dots, \gamma_o, 0, \dots, 0)$$

The elements γ_i of matrix D are called canonical correlations with ($\gamma_1 \geq \gamma_2 \geq \dots \geq \gamma_o$) and quantify the degree of correlations between the i^{th} canonical variate pair c_i *and* d_i.

[62] Chen et. al., A canonical variate analysis based process monitoring scheme and benchmark study. 2014

[63] All SIM algorithms follow this procedure and differ primarily in the way the matrix *J* is estimated



The matrices $J \in \mathbb{R}^{mXm}$ and $L \in \mathbb{R}^{rXr}$ are found via singular value decomposition

$$\Sigma_{uu}^{-1/2}\Sigma_{uy}\Sigma_{yy}^{-1/2} = U\Sigma V^T$$

where, $J = U^T\Sigma_{uu}^{-1/2}, L = V^T\Sigma_{yy}^{-1/2}, D = \Sigma$

One may retain only those canonical variate pairs (say, top *n* pairs) that show significant correlations and discard the others, and therefore achieve dimensionality reduction. Let $\hat{c}$ and $\hat{d}$ be the n-dimensional ($n < m$ and r) vectors, then,

$$\hat{c} = J_n u$$
$$\hat{d} = L_n y$$

where, J_n and L_n are first n rows of J and L

---

* It is assumed that *u* and *y* are centered. Scaling, however, is not required as CCA is scale-invariant, i.e., the estimated latent variables and the correlations remain the same with or without scaling. Also note that we have not used the *'k'* notation as CCA is designed for static data.

`

CVA states

In CVA, the two sets of vectors between which CCA is performed are *p(k)* (defined in Eq. 1) and *f(k)* where[64]

$$f(k) = [y^T(k)\ y^T(k+1)\ \cdots\ y^T(k+l)]^T$$

↖ current and future outputs

CVA therefore solves the following singular value decomposition problem and finds linear combinations of past inputs and outputs that are maximally correlated with linear combinations of the current and future outputs.

$$\Sigma_{pp}^{-1/2}\Sigma_{pf}\Sigma_{ff}^{-1/2} = U\Sigma V^T \qquad \text{eq. 2}$$

We know that Σ is a diagonal matrix. Its elements are also called (Hankel) singular values. If *n* is assumed to be the model order, then the optimal state at time *k* that is most predictive of the future outputs is given by[65]

$$\hat{x}(k) = J_n p(k)$$

↖ Contains first *n* rows of matrix $J = U^T\Sigma_{pp}^{-1/2}$

System matrices

With the states estimated, the system matrices (*A, B, C, D*) can be estimated via multivariable linear least-squares regression[66]. The final expression can be derived as

$$\begin{bmatrix}\hat{A} & \hat{B}\\ \hat{C} & \hat{D}\end{bmatrix} = cov\left(\begin{bmatrix}\hat{x}(k+1)\\ y(k)\end{bmatrix}, \begin{bmatrix}\hat{x}(k)\\ u(k)\end{bmatrix}\right) cov^{-1}\left(\begin{bmatrix}\hat{x}(k)\\ u(k)\end{bmatrix}, \begin{bmatrix}\hat{x}(k)\\ u(k)\end{bmatrix}\right)$$

The noise covariance matrices, *Q* and *R*, can be estimated as the covariance matrices of the state and output residuals, respectively.

---

[64] The number of lead values used to define *f(k)* is taken equal to the number of lags used in *p(k)* which is a common practice.

[65] Alternatively, $\hat{x}(k) = U_n^T\Sigma_{pp}^{-1/2}p(k)$ where U_n contains the first *n* columns of *U*.

[66] The multivariable linear model is $\begin{bmatrix}x(k+1)\\ y(k)\end{bmatrix} = \begin{bmatrix}A & B\\ C & D\end{bmatrix}\begin{bmatrix}x(k)\\ u(k)\end{bmatrix} + \begin{bmatrix}w(k)\\ v(k)\end{bmatrix}$

`

$$\varepsilon_x(k) = \hat{x}(k+1) - \left(\hat{A}\hat{x}(k) + \hat{B}u(k)\right) \Rightarrow \hat{Q} = \Sigma_{\varepsilon_x\varepsilon_x}$$
$$\varepsilon_y(k) = y(k) - \left(\hat{C}\hat{x}(k) + \hat{D}u(k)\right) \Rightarrow \hat{R} = \Sigma_{\varepsilon_y\varepsilon_y}$$

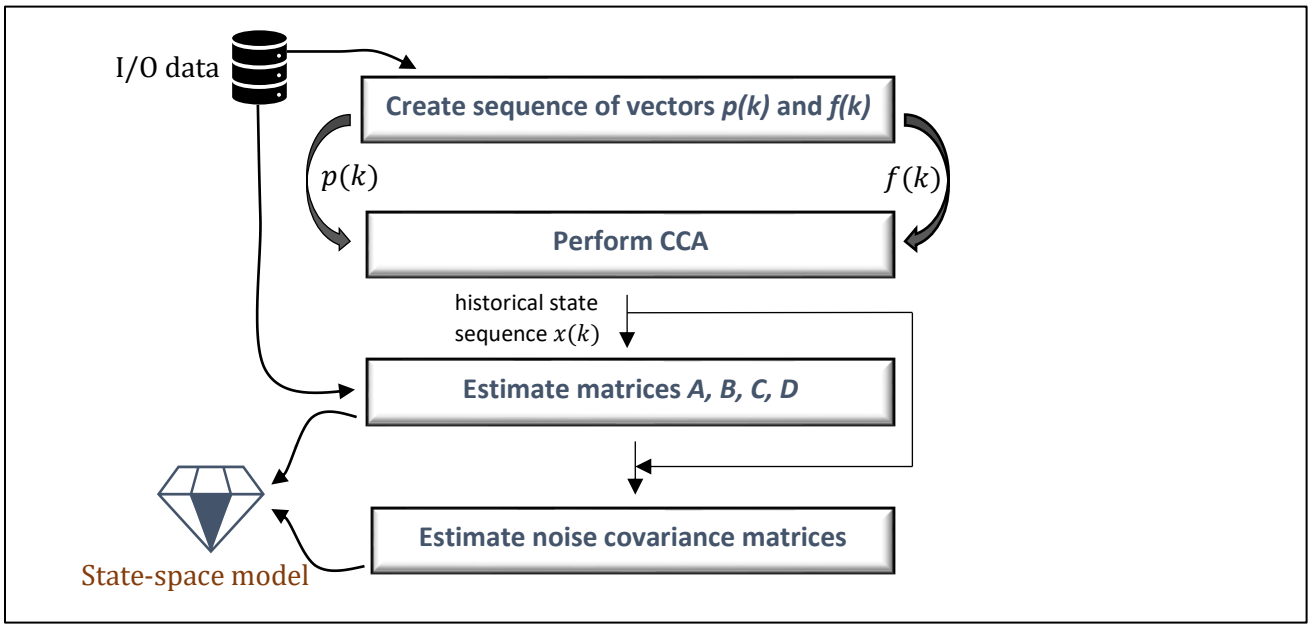


Figure 8.3: CVA model identification workflow

Hyperparameter selection

The mathematical background of the CVA algorithm suggests specification of two hyperparameters: the lag order (l) and the model order (n). While model order specification is not required prior to state estimation, the lag order needs to be known beforehand. Let's see the common approaches for estimation of these hyperparameters.

Lag order (l)

The guidance for assigning a value to the lag l is that it should be large enough to capture data autocorrelation. A commonly recommended strategy is to fit several multivariate autoregressive models with different values of l and choose the l that minimizes the AIC criterion. For example, let $\tilde{l}$ be one of the trial values. Then the following relationship is set up

`

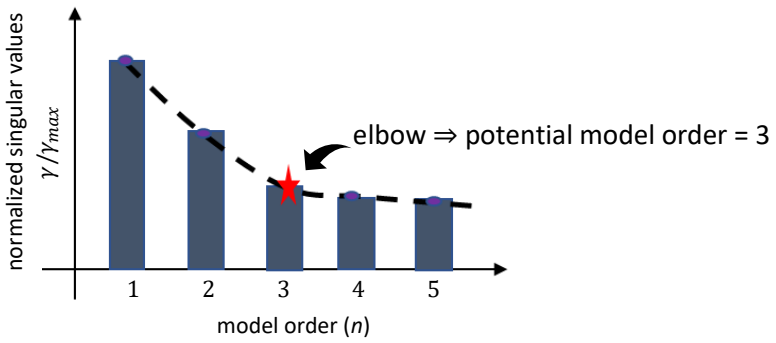
$$\begin{bmatrix} y(\tilde{l}+1) & y(\tilde{l}+2) & \cdots & y(N-1) \\ u(\tilde{l}+1) & u(\tilde{l}+2) & \cdots & u(N-1) \end{bmatrix}_{(r+m)X\,(N-\tilde{l})} = \theta \begin{bmatrix} y(\tilde{l}) & y(\tilde{l}+1) & \cdots & y(N-1) \\ u(\tilde{l}) & u(\tilde{l}+1) & \cdots & u(N-1) \\ y(\tilde{l}-1) & y(\tilde{l}) & \cdots & \vdots \\ u(\tilde{l}-1) & u(\tilde{l}) & \cdots & \vdots \\ \vdots & \vdots & \cdots & \vdots \\ y(0) & y(1) & \cdots & y(N-\tilde{l}-1) \\ u(0) & u(1) & \cdots & u(N-\tilde{l}-1) \end{bmatrix}_{\tilde{l}(r+m)\,X\,(N-\tilde{l})}$$

Each observation of an output variable is regressed against past values of all output and input variables

The above model is fitted (via OLS) for $\hat{l} = 1,2,\ldots,l_{max}$ and AICs are noted. Here, l_{max} is some sufficiently large value. The lag value corresponding to the minimum AIC is then chosen.

Model order (n)

Correct selection of the value for *n* is important: while too large a value leads to model overfitting, too small value for *n* leads to underfitting. One approach for model order selection is based on Hankel singular values. If the plot of the singular values vs model order has a prominent 'elbow', then the model order corresponding to the elbow can be chosen as shown below



Another practice is to choose *n* such that the normalized values of the *(n+1)*th onwards singular values are below some threshold (SIPPY take a default value of 0.1 for this threshold). However, very often there is no elbow or the singular values decrease slowly. Therefore, the more extensively used criteria for model order selection is AIC[67].

[67] Let $\hat{n}$ be a trial model order in some range 0 to n_{max}. The outputs $\hat{y}(k)$ are predicted using the fitted SS model and the AIC is computed. The value of $\hat{n}$ that gives minimum AIC becomes the optimal model order.

`

The flowchart below summarizes the aforementioned steps of CVA-based state-space model estimation.

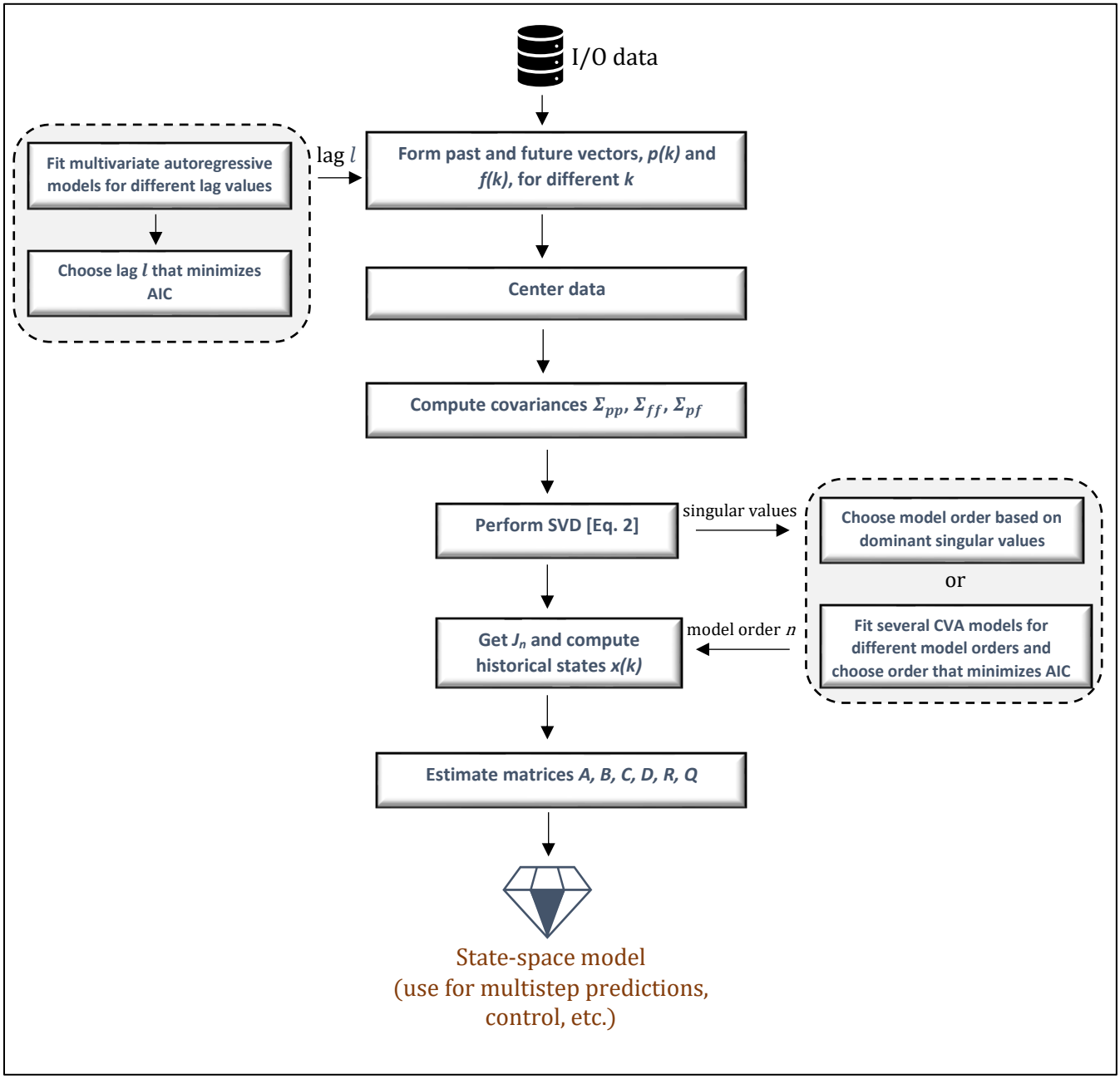


Figure 8.4: Complete workflow for dynamic process modeling via CVA[68]

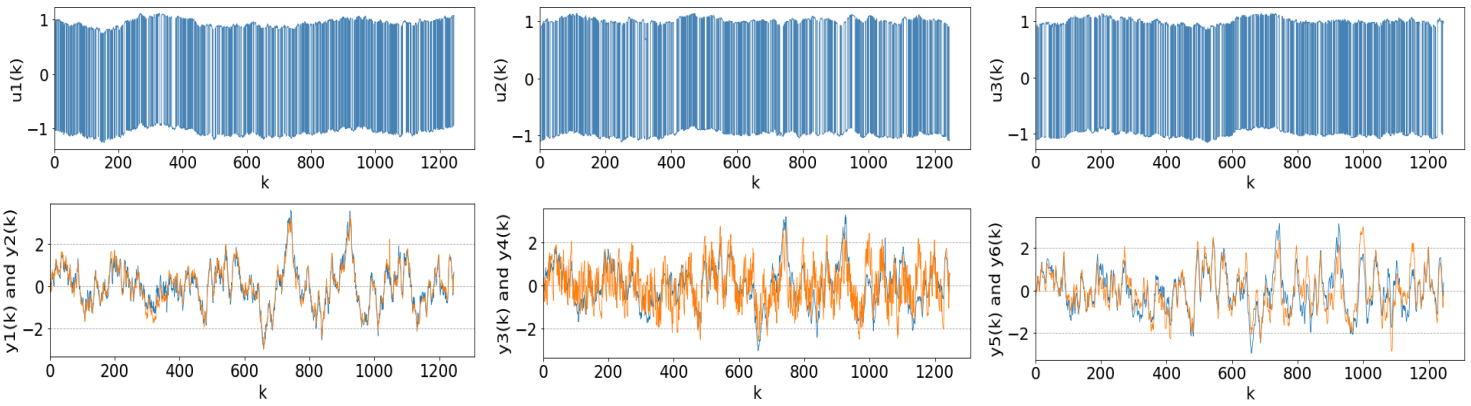
[68] Russell et al., Data-Driven Methods for Fault Detection and Diagnosis in Chemical Processes. *Springer*, 2001. Russell et al., Fault detection in industrial processes using canonical variate analysis and dynamic principal component analysis. *Chemometrics and intelligent laboratory systems*, 2000

`

8.3 Modeling Glass Furnaces via CVA

To demonstrate the ease of building state-space models using SIPPY, we will use data from a glass furnace.[69] The process has 3 inputs sources (two burners and one ventilator) and six outputs (temperature sensors in a cross section of the furnace)[70]. The provided dataset contains 1247 samples of the nine variables. The dataset can be downloaded from the DaISy datasets repository. Let's start with exploring the dataset.

```
# import packages
import matplotlib.pyplot as plt, numpy as np
from sklearn.preprocessing import StandardScaler
from sippy import system_identification as SysID

# read data
data = np.loadtxt('glassfurnace.dat'); U = data[:,1:4]; Y = data[:,4:]
```



From the above plots it is apparent that the output variables are highly correlated and therefore the attempt to build a MIMO model is justified. We will split the dataset into training and test datasets followed by data centering and model fitting.

```
# split into training and test dataset; center them
from sklearn.model_selection import train_test_split
U_train, U_test, Y_train, Y_test = train_test_split(U, Y, test_size=0.2, shuffle=False)

U_scaler = StandardScaler(with_std=False)
```

[69] De Moor B.L.R. (ed.), DaISy: Database for the Identification of Systems, Department of Electrical Engineering, ESAT/STADIUS, KU Leuven, Belgium, URL: http://homes.esat.kuleuven.be/~smc/daisy/

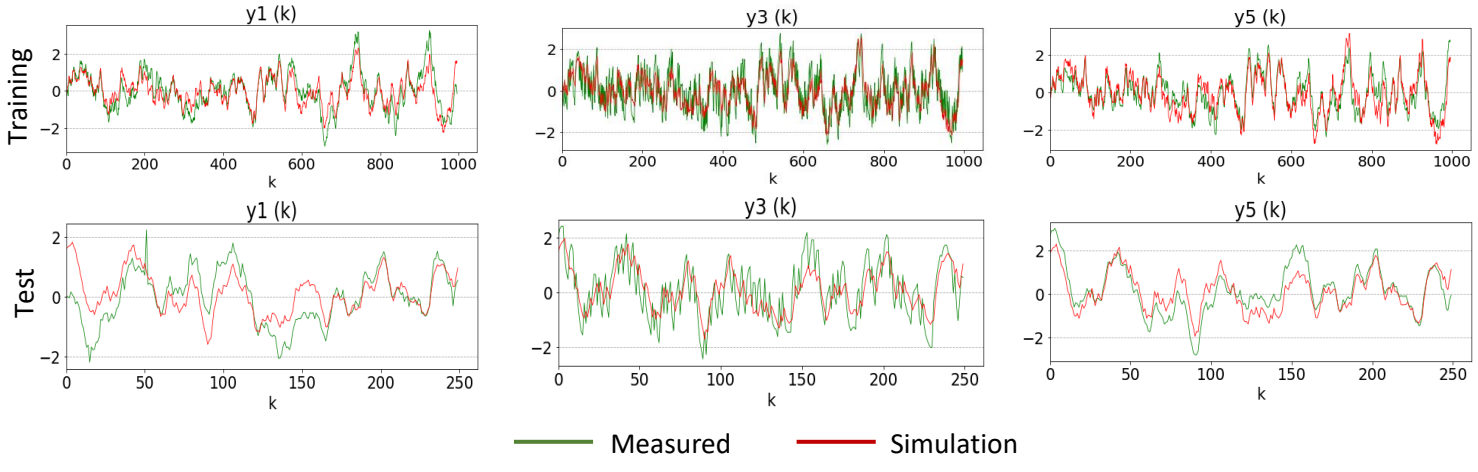
[70] Overschee and Moor, N4SID*: Subspace Algorithms for the Identification of Combined Deterministic-Stochastic Systems. Automatica, 1994

`

```
U_train_centered = U_scaler.fit_transform(U_train); U_test_centered = U_scaler.transform(U_test)
Y_scaler = StandardScaler(with_std=False)
Y_train_centered = Y_scaler.fit_transform(Y_train); Y_test_centered = Y_scaler.transform(Y_test)

# fit CVA model
model = SysID(Y_train_centered, U_train_centered, 'CVA', IC='AIC', SS_f=20, SS_orders=[1,20])

>>> The suggested order is: n= 7
```

Let's interpret the above line of code used for model fitting. Here, we have specified a lag of 20 via the *SS_f* parameter and the Akaike information criteria is being used for optimal model order selection. The order is constrained to be in the range of 1 to 20 as specified via the *SS_orders* parameter. SIPPY returned the optimal order as seven. With the model in place, let's check out the model's simulation performance to assess its acceptability.

```
# compare simulation responses vs measurements on training and test data
from sippy import functionsetSIM as fsetSIM
Xid_train, Yid_train_centered = fsetSIM.SS_lsim_process_form(model.A, model.B, model.C, model.D,
                                    np.transpose(U_train_centered), model.x0) # Xid → states
Yid_train_centered = np.transpose(Yid_train_centered)

Xid_test, Yid_test_centered = fsetSIM.SS_lsim_process_form(model.A, model.B, model.C, model.D,
                                    np.transpose(U_test_centered), Xid_train[:,-1, None]) # passing the
                                    state at the end of training as the starting state for test data
Yid_test_centered = np.transpose(Yid_test_centered)
```



The comparison of simulation responses and measurements suggests that although the model tends to underpredict for some instances, overall, the model does well to capture the broad variations in the outputs.

`

8.4 Monitoring Industrial Chemical Plants using CVA

In this section we will see another application of CVA. We will learn the details of CVA-based monitoring tool development through step-by-step application for a large-scale industrial chemical plant. The process under study is called Tennessee Eastman process (TEP) and Figure 8.5 shows the process flowsheet. This process consists of several unit operations: a reactor, a condenser, a separator, a stripper, and a recycle compressor. There are 22 continuous process measurements, 19 composition measurements, and 11 manipulated variables. A dataset for this process has been made available at https://github.com/camaramm/tennessee-eastman-profBraatz.[71] The dataset contains training and test data from normal operation period and 21 faulty periods with distinct fault causes. For each fault class, training dataset contains 480 samples collected over 24 operation hours and test dataset contains 960 samples collected over 48 operation hours. For the faulty data, faulty operation starts from sample 160 onwards.

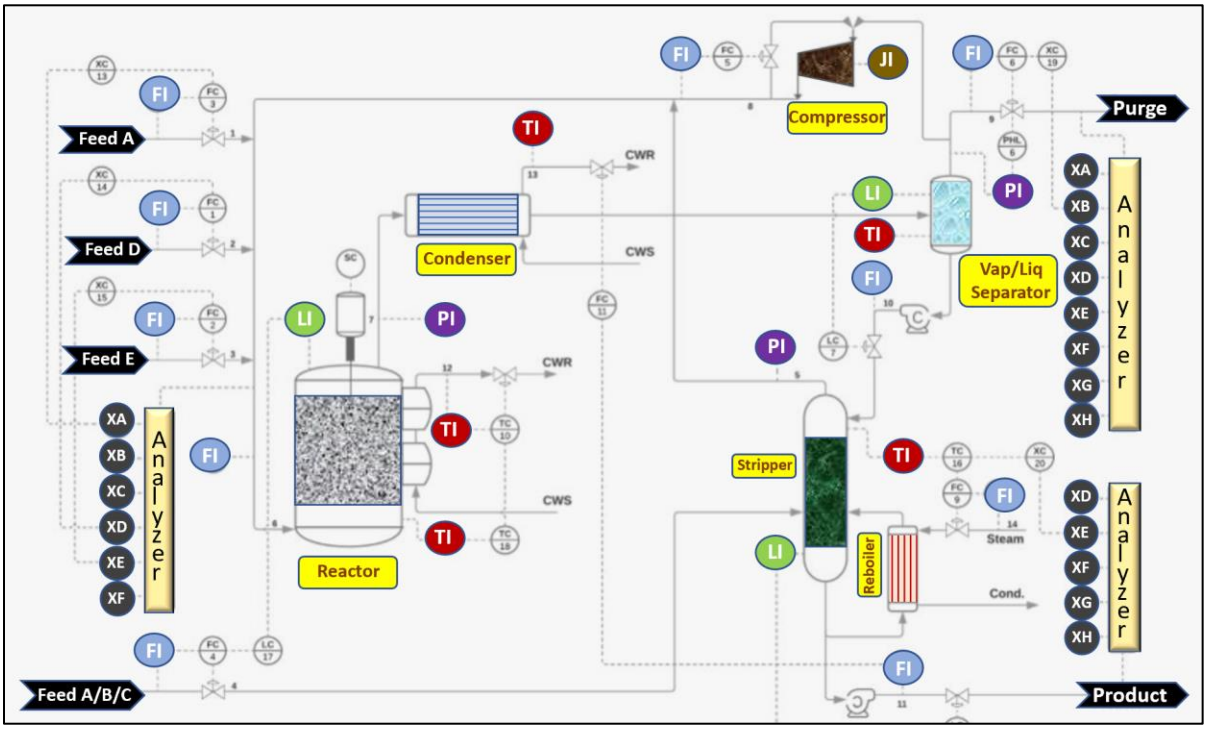


Figure 8.5: Tennessee Eastman process flowsheet[72] with flow (FI), temperature (TI), pressure (PI), composition (analyzers), level (LI), power (JI) sensors.

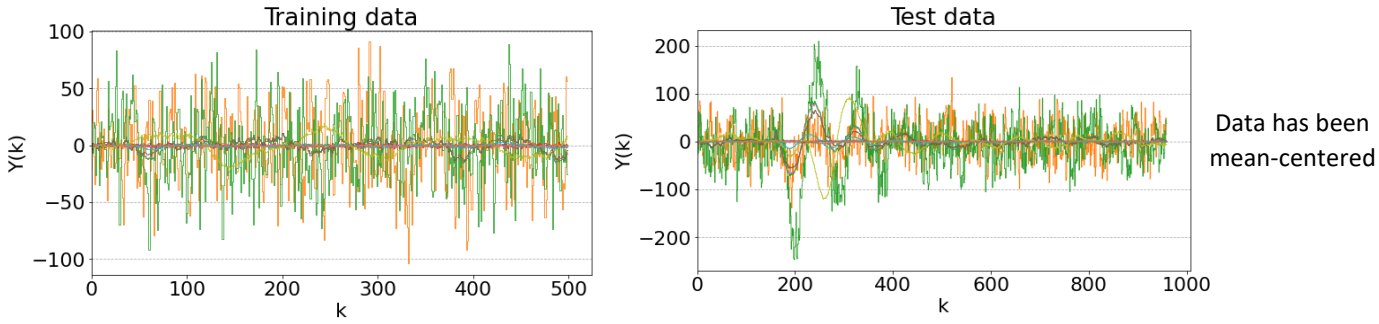
[71] Detailed information about the process and the faults can be obtained from the original paper by Downs and Vogel titled 'A plant-wide industrial process control problem'

[72] Adapted from the original flowsheet by Gilberto Xavier (https://github.com/gmxavier/TEP-meets-LSTM) provided under Creative-Commons Attribution 4.0 International License (https://creativecommons.org/licenses/by/4.0/).

`

As alluded to before, the CVA modeling and monitoring procedures provided in this book mirror the description in the work[68] of Russell et al., and we will therefore attempt to replicate the CVA-based fault detection results from their paper titled '*Fault detection in industrial processes using canonical variate analysis and dynamic principal component analysis*'. Specifically, we will use the *Fault 5* dataset (file *d05_te.dat*) as our test data and normal operation dataset (file *d00.dat*) as our training data. We will use the manipulated variables as our inputs and the process measurements as our outputs. Our objective is to build a monitoring tool that can accurately flag the presence of a process fault with low frequency of false alerts. Let's begin with a quick exploration of the dataset.

```
# import packages
import matplotlib.pyplot as plt, numpy as np
from sklearn.preprocessing import StandardScaler

# read data and separate input-output variables
trainingData = np.loadtxt('d00.dat').T
FaultyData =  np.loadtxt('d05_te.dat')
uData_training, yData_training = trainingData[:,41:52], trainingData[:,0:22]
uData_test, yData_test = FaultyData[:,41:52], FaultyData[:,0:22]
```



It is obvious that continuously monitoring all the 22 output (and 11 input) variables is not practical. The single combined output plots above do seem to clearly indicate a process upset around sample number 200 in faulty dataset; however, they also seem to give a wrong impression of normal process conditions sample number 500 onwards. It is known that Fault 5 continues until the end of the dataset. Fortunately, CVA makes the monitoring task easy by summarizing the state of any complex multivariate process into three simple indicators or monitoring indices as shown in Figure 8.6. During model training, statistical thresholds are determined for the indices and for a new data-point, the new indices' values are compared against the thresholds. If any of the three thresholds are violated, then presence of abnormal process conditions is confirmed. Let's look into how these monitoring indices are computed.

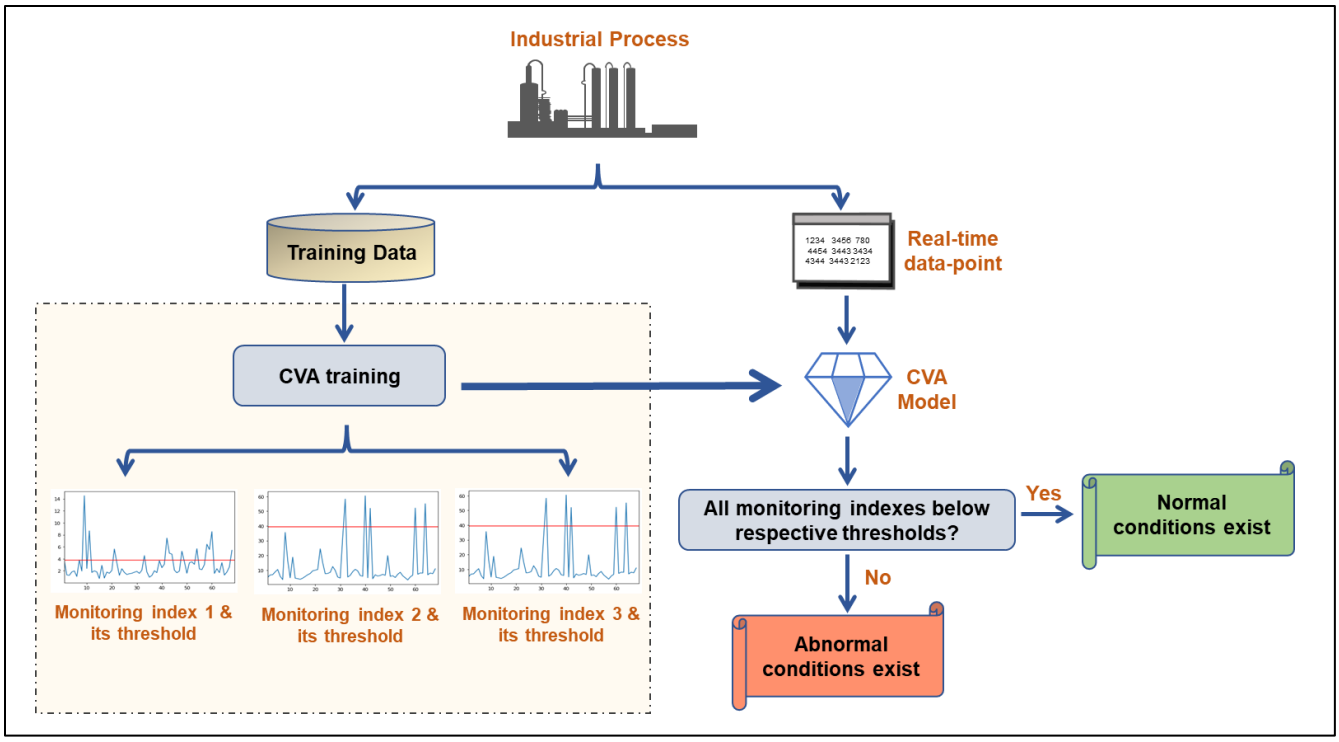


Figure 8.6: CVA-based process monitoring workflow

Process monitoring/fault detection indices

In CVA-based method, 3 indices - T_s^2, T_e^2, Q - are computed. The computation of these metrics and their corresponding threshold are shown below. Note that the threshold expressions are based on the assumption that process variables are gaussian distributed; if this assumption is invalid then empirical techniques such as KDE or percentile may be employed.

Metric 1: T_s^2

$$T_s^2(k) = x^T(k)x(k) = p^T(k)J_n^T J_n p(k)$$

Threshold @ significance level α

$$T_{s,CL}^2 = \frac{n(N^2-1)}{N(N-n)}F_\alpha(n, N-n)$$

Metric 2: T_e^2

$$T_e^2(k) = p^T(k)J_e^T J_e p(k)$$

Threshold @ significance level α

$$T_{e,CL}^2 = \frac{z(N^2-1)}{N(N-z)}F_\alpha(z, N-z)$$

Metric 3: Q

$$Q(k) = r^T(k)r(k)$$
$$r(k) = p(k) - J_n^T J_n p(k)$$

Threshold @ significance level α

$$Q_{CL} = \frac{\sigma}{2\mu} g\chi_\alpha^2\left(\frac{2\mu^2}{\sigma}\right)$$

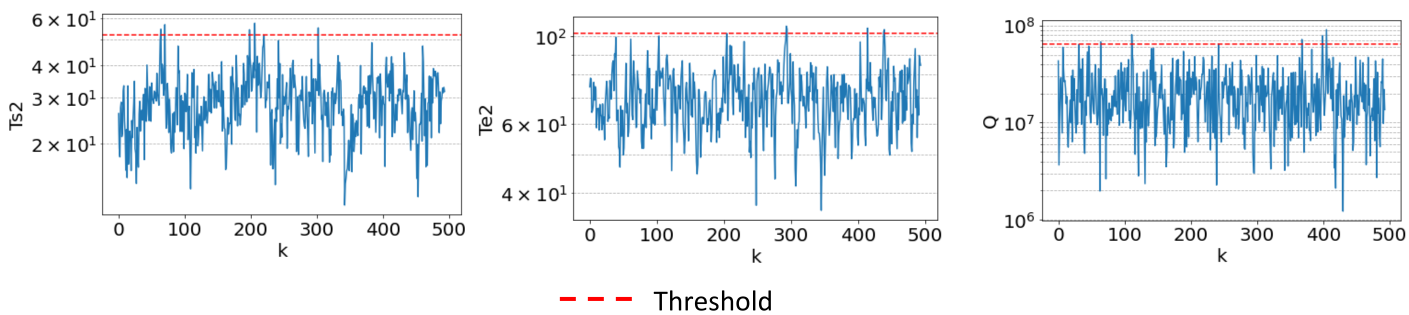
- J_e contains all except the first *n* rows of *J* i.e., the last $l(m+r)-n$ rows of *J*
- $z = l(m+r)-n$
- $F_\alpha(n, N\text{-}n)$ is the 1-α percentile of the *F* distribution with *n* and *N-n* degrees of freedom
- χ_α^2 is the (1-α) percentile of a chi-squared distribution with *h* degrees of freedom; μ denotes the mean value and σ denotes the variance of the Q metric
- Significance level of α = 0.05 would mean that there is a 5% chance that an alert is false

`

Let's compute the monitoring indices for the training dataset. To ensure a better understanding of the methodology, we will skip the usage of SIPPY package for CVA modeling and carry out each step from scratch.

```
#%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
##      generate past (p) and future (f) vectors for training dataset and center them
#%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
N = trainingData.shape[0]
l = 3 # as used by Russell et. al
m = uData_training.shape[1]
r = yData_training.shape[1]

pMatrix_train = np.zeros((N-2*l, l*(m+r)))
fMatrix_train = np.zeros((N-2*l, (l+1)*r))
for i in range(l, N-l):
    pMatrix_train[i-l,:] = np.hstack((yData_training[i-l:i,:].flatten(), uData_training[i-l:i,:].flatten()))
    fMatrix_train[i-l,:] = yData_training[i:i+l+1,:].flatten()

# center data
p_scaler = StandardScaler(with_std=False); pMatrix_train_centered = p_scaler.fit_transform(pMatrix_train)
f_scaler = StandardScaler(with_std=False); fMatrix_train_centered = f_scaler.fit_transform(fMatrix_train)

#%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
##                  computes covariances and perform SVD
#%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
import scipy

sigma_pp = np.cov(pMatrix_train_centered, rowvar=False)
sigma_ff = np.cov(fMatrix_train_centered, rowvar=False)
sigma_pf = np.cov(pMatrix_train_centered, fMatrix_train_centered,
                                                rowvar=False)[:len(sigma_pp),len(sigma_pp):]

matrixProduct = np.dot(np.dot(np.linalg.inv(scipy.linalg.sqrtm(sigma_pp).real), sigma_pf),
                                                np.linalg.inv(scipy.linalg.sqrtm(sigma_ff).real))
U, S, V = np.linalg.svd(matrixProduct)
J = np.dot(np.transpose(U), np.linalg.inv(scipy.linalg.sqrtm(sigma_pp).real))

# get the reduced order matrices
n = 29 # as used by Russell et. al
Jn = J[:n,:]
Je = J[n:,:]
```

```
#%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
##                    get fault detection metrics for training data
#%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
Xn_train = np.dot(Jn, pMatrix_train_centered.T)
Ts2_train = [np.dot(Xn_train[:,i], Xn_train[:,i]) for i in range(pMatrix_train_centered.shape[0])]

Xe_train = np.dot(Je, pMatrix_train_centered.T)
Te2_train = [np.dot(Xe_train[:,i], Xe_train[:,i]) for i in range(pMatrix_train_centered.shape[0])]

r_train = pMatrix_train_centered.T - np.dot(Jn.T, Xn_train)
Q_train = [np.dot(r_train[:,i], r_train[:,i]) for i in range(pMatrix_train_centered.shape[0])]

# determine thresholds as 99th percentile of respective metrics
Ts2_CL = np.percentile(Ts2_train, 99)
Te2_CL = np.percentile(Te2_train, 99)
Q_CL = np.percentile(Q_train, 99)
```



The T_s^2 metric measures the systematic variations 'inside' the state space defined by the state variables. Correspondingly, T_r^2 quantifies the variations 'outside' the state space. When only T_s^2 violates the threshold, it indicates that the state variables are 'out-of-control' but the process can still be well explained by the estimated state-space model. When T_s^2 or Q violates the threshold, it indicates that the characteristics of noise affecting the system has changed or the estimated model cannot explain the new faulty observations.

NOTES

CVA vs PCA (or PLS) for process monitoring

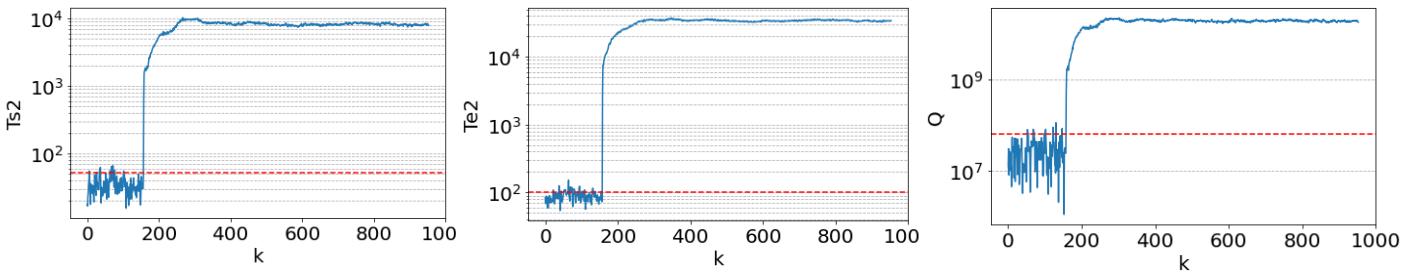
You may notice the similarity between CVA-based and PCA-based monitoring methodologies in terms of metric definitions. While dynamic PCA (and dynamic PLS) are sometimes used to monitor dynamic processes, several studies have shown the superiority of CVA over DPCA (and DPLS) for dynamic process monitoring.

`

Fault detection

It's time now to check whether our monitoring charts can help us detect the presence of process abnormalities. For this, we will compute the monitoring statistics for the test data.

```
#%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
##     generate past vectors (p(k)) for test dataset and center
#%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
Ntest = FaultyData.shape[0]

pMatrix_test = np.zeros((Ntest-2*l, l*(m+r)))
for i in range(l,Ntest-l):
    pMatrix_test[i-l,:] = np.hstack((yData_test[i-l:i,:].flatten(), uData_test[i-l:i,:].flatten()))

pMatrix_test_centered = p_scaler.transform(pMatrix_test)


#%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
##                  get fault detection metrics for training data
#%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
Xn_test = np.dot(Jn, pMatrix_test_centered.T)
Ts2_test = [np.dot(Xn_test[:,i], Xn_test[:,i]) for i in range(pMatrix_test_centered.shape[0])]

Xe_test = np.dot(Je, pMatrix_test_centered.T)
Te2_test = [np.dot(Xe_test[:,i], Xe_test[:,i]) for i in range(pMatrix_test_centered.shape[0])]

r_test = pMatrix_test_centered.T - np.dot(Jn.T, Xn_test)
Q_test = [np.dot(r_test[:,i], r_test[:,i]) for i in range(pMatrix_test_centered.shape[0])]
```



The control charts for the test data shows successful detection of process fault. The T_e^2 metric is known to be overly sensitive with high incidence of false alerts which can explain why it seems to often breach the threshold before the onset of the actual fault. The following flowchart summarizes all the steps for building a CVA-based monitoring tool.

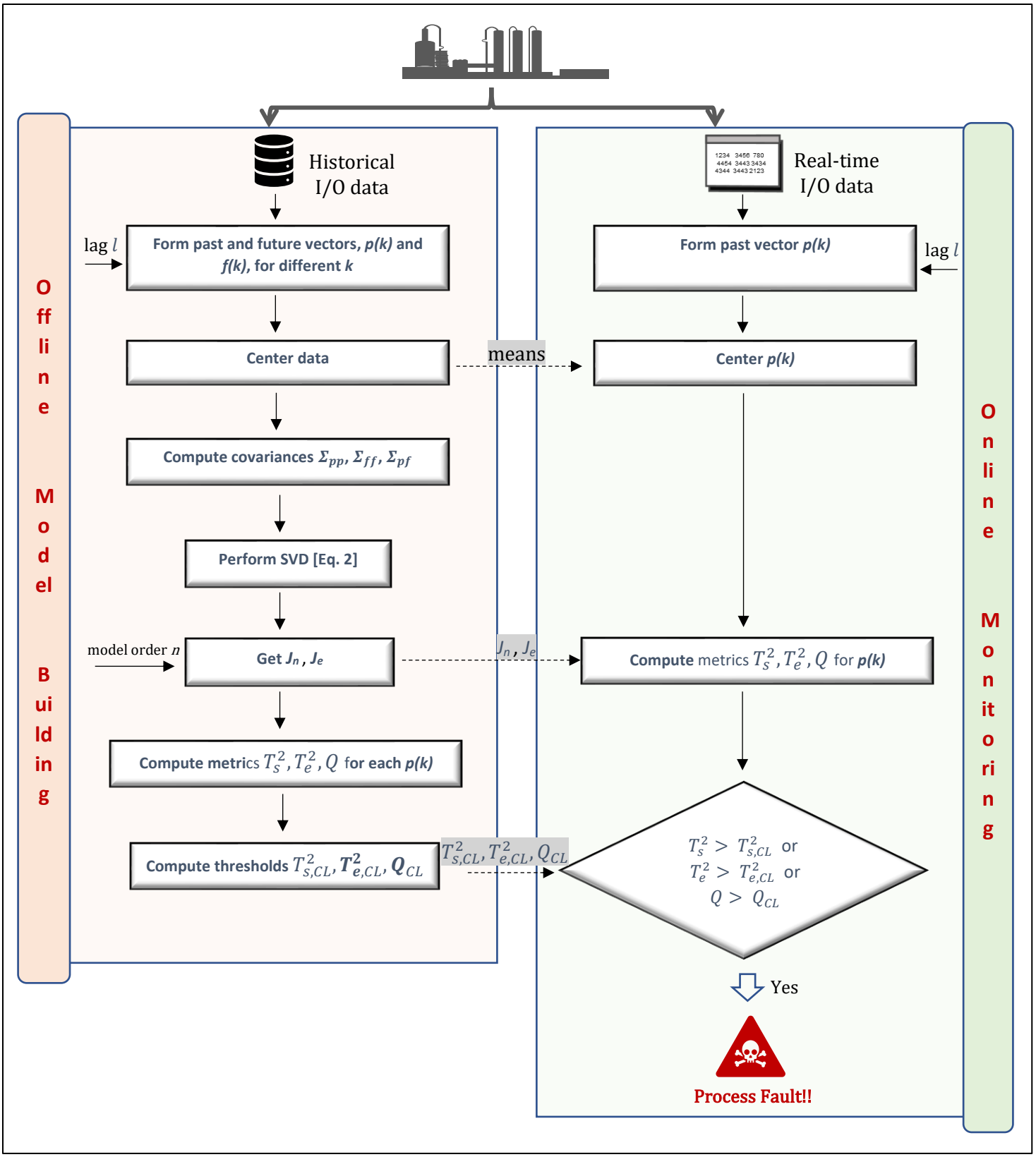


Figure 8.7: Complete workflow for process monitoring via CVA

`

Equivalence of state-space and ARMAX models

Although state-space and input-output models may seem very different, they are related! Given any state-space model, an equivalent ARMAX model can be determined and *vice-versa*. Therefore, other linear structures like ARX or OE can be re-written as state-space model as well.

SISO I/O difference equation form to SS form

Consider

$$y(k) + a_1 y(k-1) + \cdots + a_{n_a} y(k-n) = b_1 u(k-1) + \cdots + b_n u(k-n)$$

⇅ equivalent

$$x(k+1) = Ax(k) + Bu(k)$$

$$y(k) = Cx(k)$$

where, $x(k) \in \mathbb{R}^{n\,X\,1}, C = [1, 0, \ldots, 0]$

$$B = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix}, \quad A = \begin{bmatrix} -a_1 & 1 & 0 & \cdots & 0 \\ -a_2 & 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & 1 \\ -a_n & 0 & 0 & \cdots & 0 \end{bmatrix},$$

SS form to transfer operator form

Consider

$$x(k+1) = Ax(k) + Bu(k)$$

$$y(k) = Cx(k) + Du(k)$$

⇊ using shift operator

$$qx(k) = Ax(k) + Bu(k)$$

$$\Rightarrow (qI - A)x(k) = Bu(k)$$

$$\Rightarrow x(k) = (qI - A)^{-1} Bu(k)$$

$$\Rightarrow y(k) = \underbrace{[C(qI - A)^{-1}B + D]}_{G(q)} u(k)$$

Nonetheless, we have seen in this chapter that SS model enjoys some significant advantages vis-à-vis ARMAX models such as no requirement of *a prior* parametrization (i.e., delay and orders for all the inputs need not be pre-specified) and usage of robust SVD calculations for model fitting.

Hopefully, this chapter has sufficiently impressed upon you the powerful capabilities of CVA. Due to its favorable properties, it is not surprising that CVA is one of the SysID methods provided in Aspen's DMCplus software[73]. With this we complete our brief foray into the world of MIMO dynamic process modeling. Be mindful that we have only scratched the surface of the vast literature on SIM and CVA. We hope that you have gained sufficient understanding of how CVA works and the confidence to use it to build predictive and monitoring tools. You now have all the tools to handle SISO and MIMO linear systems. Moving ahead, we will now

[73] Chaiwatanodom, Paphonwit, Fault Detection and Identification of Large-scale Dynamical Systems. *Diss. Massachusetts Institute of Technology*, 2021

`

relax the restriction of linearity and learn how to handle nonlinear processes in the next chapter.

Summary

In this chapter, we looked at state-space models and saw how it offers more insights into the internal process dynamics compared to input-output models. For fitting state-space models we studied subspace identification techniques with the focus on CVA models. Through a couple of industrial-scale case-studies, we learnt how to build dynamic models for usage in process control and process monitoring. In the next chapter we will see some classical techniques for modeling nonlinear processes.

`

Chapter 9

Nonlinear System Identification: Going Beyond Linear Models

In the previous chapters we restricted our attention to only linear processes. However, most industrial processes exhibit linear behavior only in a limited range of operating conditions. If a global model of complex processes (such as high-purity distillation, pH neutralization, polymerization, etc.) is sought, then the demon of nonlinearity would inadvertently surface. It won't be an exaggeration to state that nonlinearity is not an exception, rather the rule in process industry. We won't be surprised if you are tempted to always resort to artificial neural networks-based structures which have become synonymous with nonlinear modeling now-a-days. However, there exist simple classical nonlinear model structures which are more interpretable and equally powerful. We will explore these models in this chapter.

Nonlinear models can be powerful tools but there are certain costs to their usage. These include greater demand for variability in training data (which means more laborious experiment design), more complex model selection procedure, and more computationally intensive parameter estimation. Therefore, the decision to use nonlinear models should be judiciously made. If you do decide to go ahead, then the concepts covered in this chapter may help you greatly. Specifically, the following topics will be covered

- Introduction to NARX models
- Modeling heat exchanger process using NARX models
- Introduction to block-structured models, specifically, Hammerstein models and Wiener models

---

`

9.1 Nonlinear System Identification: An Introduction

Nonlinear SysID becomes a necessary evil when accurate predictions are sought over wide range of operating conditions. Although nonlinear SysID literature is not as mature as linear SysID, SysID community has developed several useful approaches for nonlinear modeling. Figure 9.1 shows a few popular approaches. In the 1st approach, a nonlinear relationship is directly fitted between *y(k)* and past data The nonlinear form could be polynomial-based or more complex such as neural network-based. We will focus on the polynomial form in this chapter. Just like linear SysID, different model structures can be postulated, namely, nonlinear ARX (NARX), NOE, NARMAX, etc. In the 2nd approach, the linear and nonlinear parts are defined in separate blocks, leading to block-structured models. Depending upon whether the nonlinear block comes before or after the linear block, one could have the Hammerstein model or the Wiener model (the two most popular models from this class). In the 3rd approach, several linear models are built for different regions of the operating space and then they are combined to give one global model. Numerous industrial applications of these approaches have been reported for predictive modeling, process control, and process monitoring.

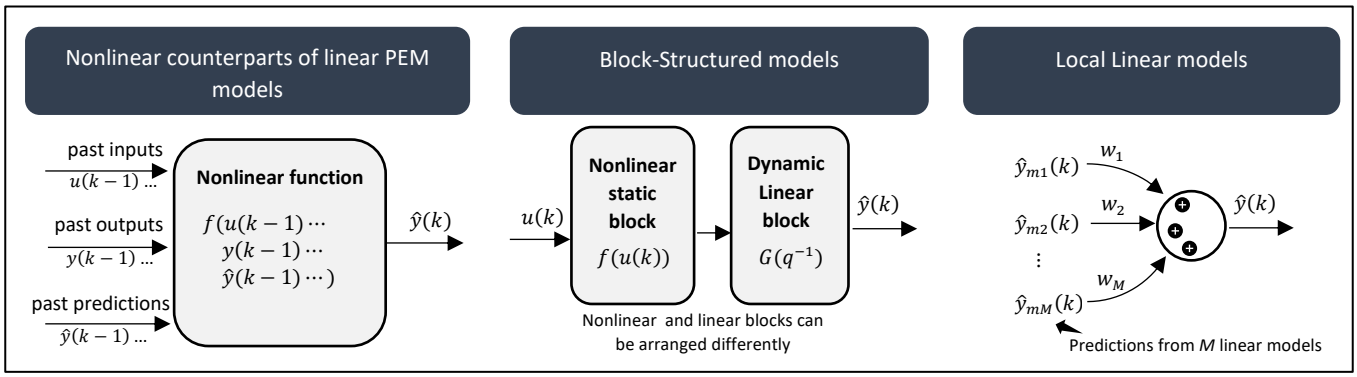
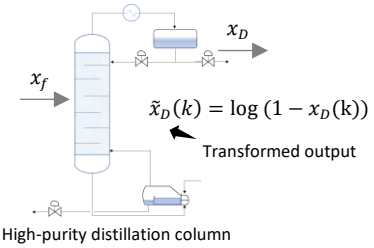


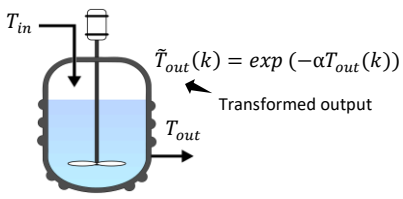
Figure 9.1: Popular classical approaches for nonlinear system identification

Nonlinear transformation

Considering high computational burden of nonlinear SysID, if the nature of nonlinearity is known beforehand, then a nonlinear transformation of the input or output variables may be utilized to avoid explicit nonlinear fitting. Illustrations below provide some example scenarios.



High-purity distillation column



Non-isothermal CSTR
(commons.wikimedia.org/wiki/File:Continuous_bach_reactor_CSTR.svg)

`

In practice the required nonlinear transformation is more likely to be unknown. Therefore, let's start with arguably the default choice for nonlinear modeling: the NARX models.

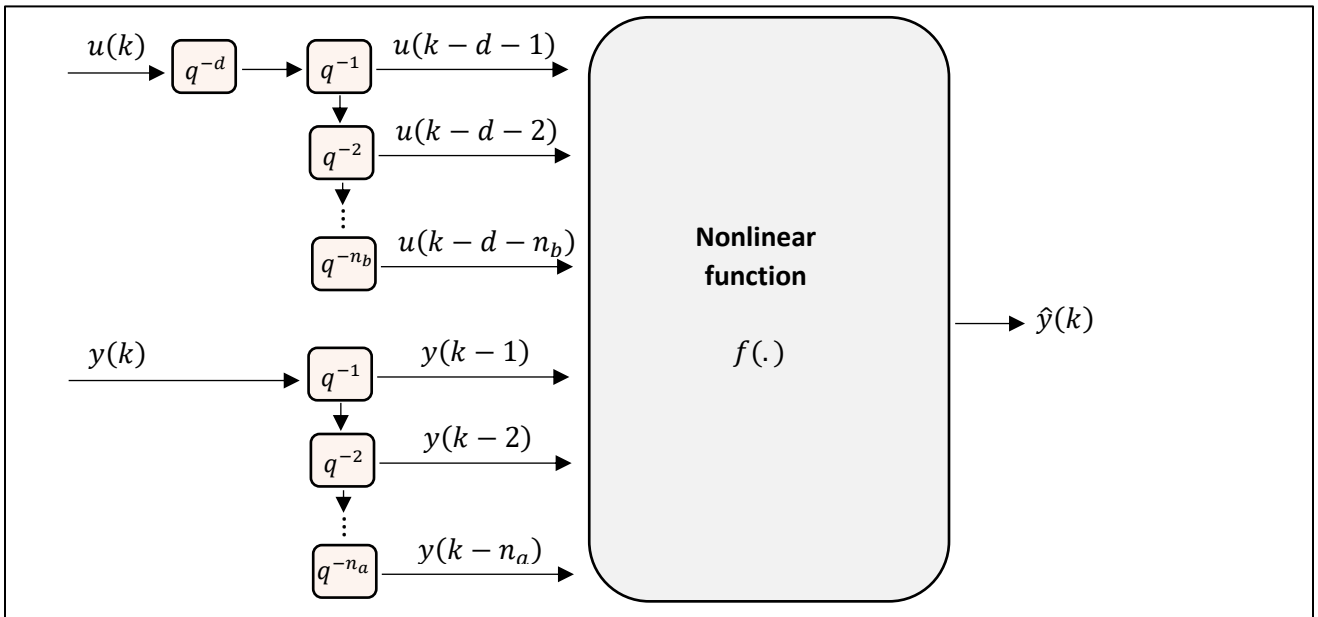
9.2 NARX Models: An Introduction

Nonlinear autoregressive with eXogenous inputs (NARX) model is, as the name suggests, the nonlinear analogue of ARX model, wherein, the past inputs and outputs of the process are used for future predictions. As shown in Figure 9.2, the model takes the following form

$$y(k) = f(u(k-d-1), \ldots, u(k-d-n_b), y(k-1), \ldots, y(k-n_a), \theta) + e(k)$$

(θ: model parameters)

While the function *f* is commonly taken to be polynomial form, any other nonlinear structure (including the neural networks) could be used as deemed suitable. A feature of polynomial NARX is that the predictor, $\hat{y}$, is linear-in-the-parameters! Therefore, polynomial NARX parameter estimation becomes straight-forward if the specific terms/regressors (such as *y(k-1)u(k-1), u²(k-1)*) that are to be included in the model is known *a priori*. Of course, like with ARX, hyperparameters (*d*, n_b, n_a, *polynomial degree*) have to be specified.



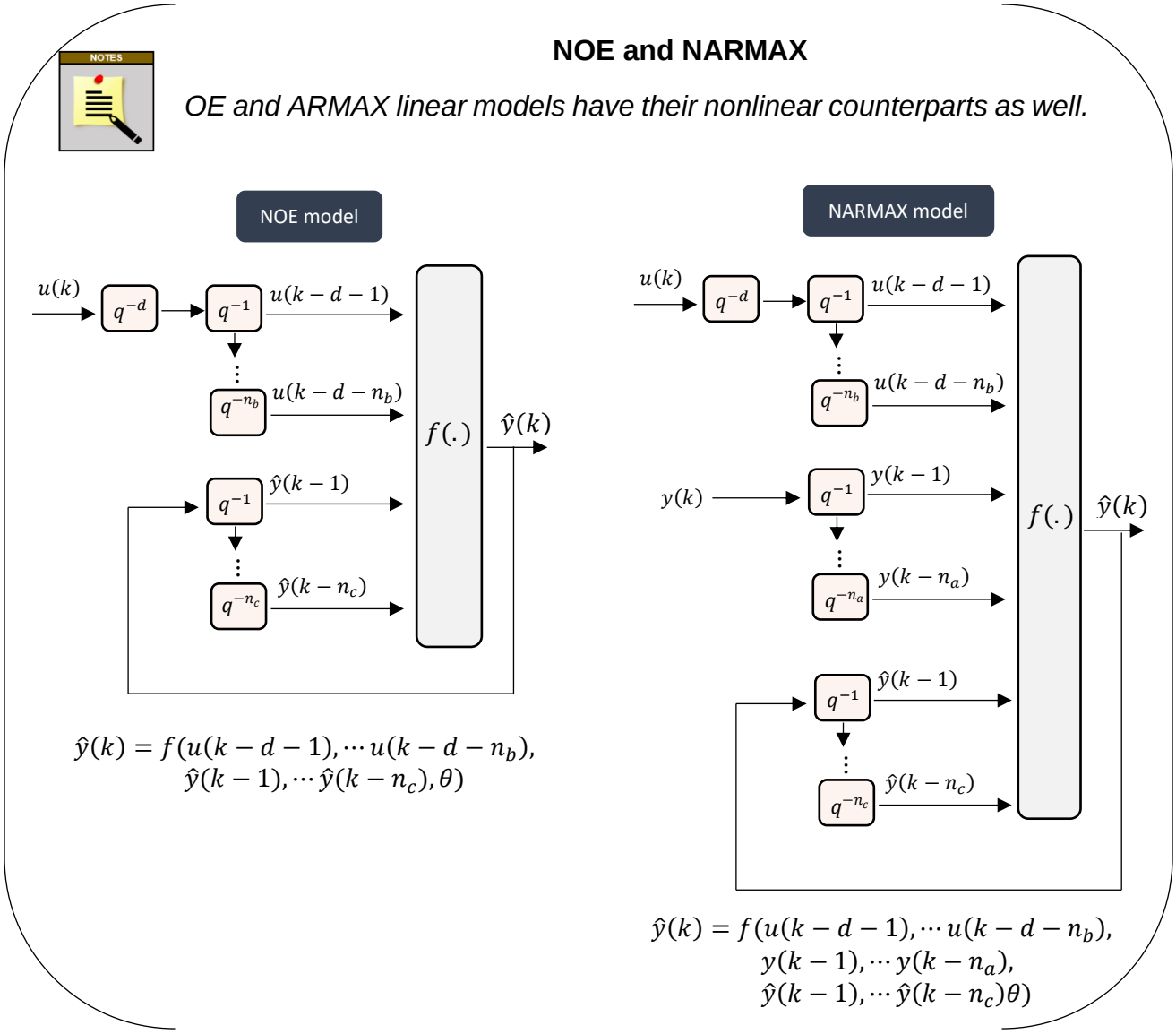
Example: A polynomial NARX model of degree 2, *d=0*, and $n_a = n_b = 1$

$$\hat{y}(k) = a_1 y(k-1) + a_2 y^2(k-1) + a_3 y(k-1)u(k-1) + b_1 u(k-1) + b_2 u^2(k-1)$$

Figure 9.2: NARX model structure

`

The primary challenge during polynomial NARX identification is model structure selection. The number of potential regressors can become very large for even modest value of lags (n_a, n_b) and polynomial degree[74], and inclusion of many terms increase chances of overfitting. Fortunately, this task is automated[75] using AIC-based algorithm in the Python package SysIdentPY. Let's see how this package can be used for nonlinear identification of an industrial heat exchanger.



[74] Very often, *d* is kept zero and only the input and output lags (*n* and *m*) are optimized for a given polynomial degree.

[75] In a recent work by Sun & Braatz (*ALVEN: Algebraic learning via elastic net for static and dynamic nonlinear system identification, Computers & Chemical Engineering, 2020*), a DALVEN methodology has been proposed that uses a combination of nonlinear transformations, univariate feature selection, and elastic net regression for NARX structure selection.

`

9.3 Nonlinear Identification of a Heat Exchanger Process Using NARX

To demonstrate the utility of nonlinear SysID, we will use data from a liquid-saturated steam heat exchanger[76] where hot saturated steam is used to heat cold liquid. The provided dataset contains 4000 samples (sampling time 1s) of input (liquid flow rate) and output (outlet liquid temperature) variables. For this benchmark nonlinear process, our objective will be to build a dynamic model to predict the outlet liquid temperature as a function of incoming liquid flowrate. The dataset can be downloaded from the DaISy datasets repository.

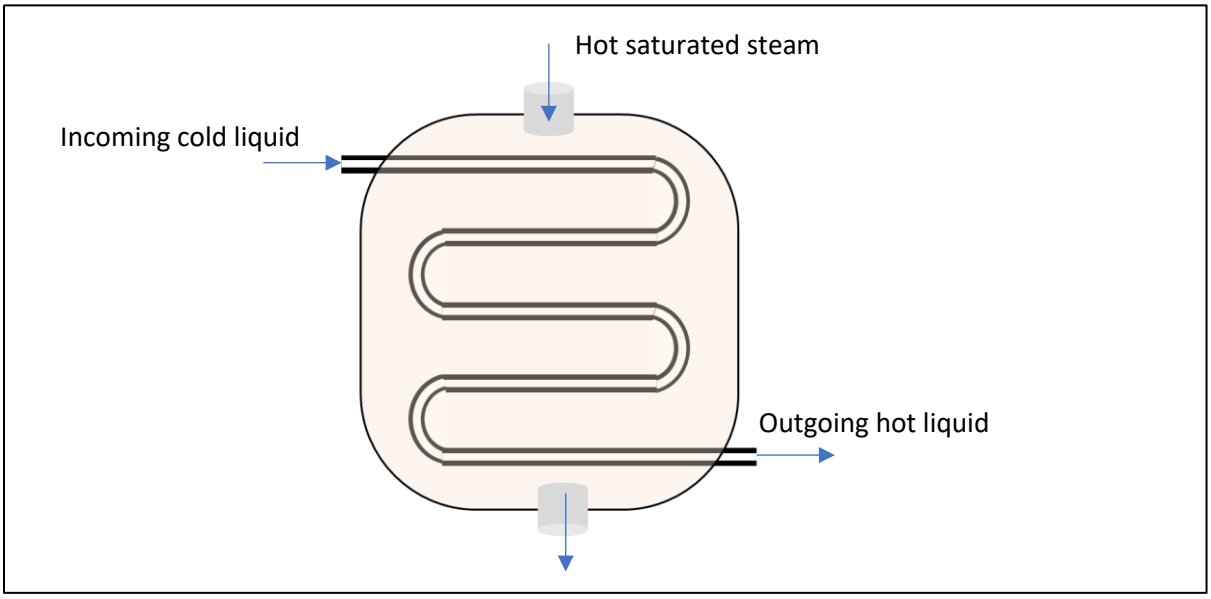


Figure 9.3: Liquid-saturated steam heat exchanger

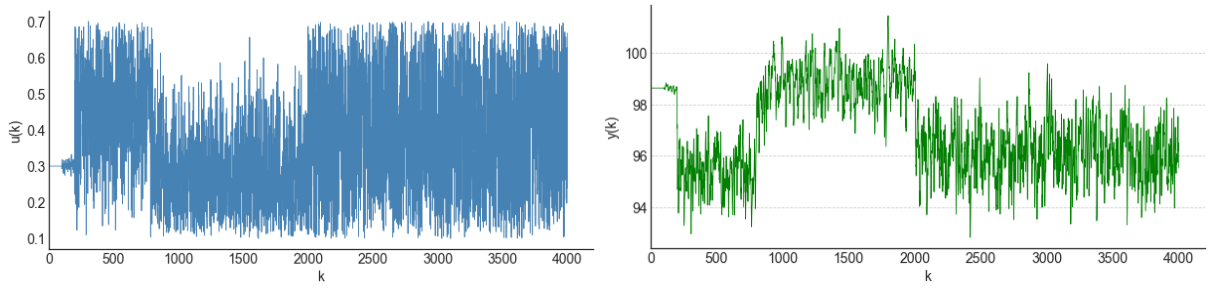
We will use the SysIdentPy package for this nonlinear SysID exercise. Let's begin by exploring the dataset.

```
# import packages
import matplotlib.pyplot as plt, numpy as np
from sklearn.preprocessing import StandardScaler
from statsmodels.graphics.tsaplots import plot_acf
from statsmodels.tsa.stattools import ccf
```

[76] De Moor B.L.R. (ed.), DaISy: Database for the Identification of Systems, Department of Electrical Engineering, ESAT/STADIUS, KU Leuven, Belgium, URL: http://homes.esat.kuleuven.be/~smc/daisy/

`

```
from sysidentpy.model_structure_selection import FROLS
from sysidentpy.basis_function._basis_function import Polynomial
from sysidentpy.utils.display_results import results

# read heat-exchanger data
data = np.loadtxt(exchanger.dat')
u = data[:,1, None]; y = data[:,2, None]

plt.figure(), plt.plot(u, 'steelblue'), plt.ylabel('u(k)'), plt.xlabel('k')
plt.figure(), plt.plot(y, 'g'), plt.ylabel('y(k)'), plt.xlabel('k')
```



Next, we will split the dataset into training (75%) and test (25%) datasets, and center them.

```
# split into training and test dataset
u_fit = u[:3000,0:1]; u_test = u[3000:,0:1]
y_fit = y[:3000,0:1]; y_test = y[3000:,0:1]

u_scaler = StandardScaler(with_std=False)
u_fit_centered = u_scaler.fit_transform(u_fit); u_test_centered = u_scaler.transform(u_test)
y_scaler = StandardScaler(with_std=False)
 y_fit_centered = y_scaler.fit_transform(y_fit); y_test_centered = y_scaler.transform(y_test)
```

To fit the polynomial NARX, we need to specify the polynomial degree and the lags. The authors of the dataset recommend[77] "a lag of 3 for the 'AR part' and 9 for the '*X* part'". For polynomial degree, a value of 2 is chosen to start with. SysIdentPY uses AIC to determine which specific polynomial terms become part of the regressor set in the model. The following code achieves this.

[77] Bittanti and L. Piroddi, "Nonlinear identification and control of a heat exchanger: a neural network approach", Journal of the Franklin Institute, 1996.

`

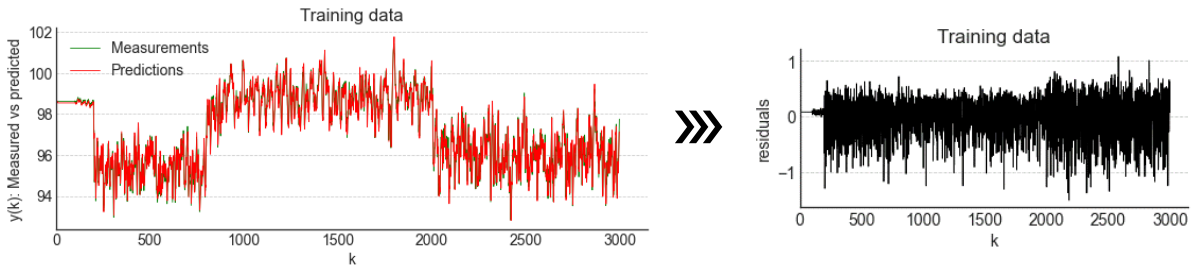
```
# fit NARX model
basis_function = Polynomial(degree=2)
model = FROLS(
    order_selection=True,
    n_info_values=75,
    ylag=3, xlag=9,
    info_criteria='aic',
    estimator='least_squares',
    basis_function=basis_function
)
model.fit(X=u_fit_centered, y=y_fit_centered)
```

A total of 57 polynomial terms are selected by SysIdentPy as part of the regressor set. The snippet below shows some of the terms ['x1' to be read as 'u'].

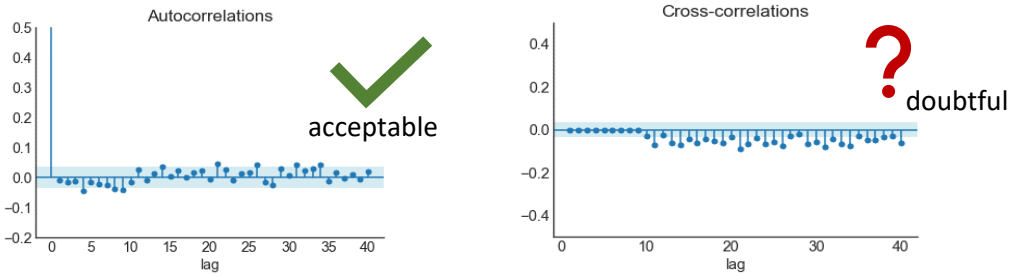
	Regressors	Parameters	ERR
0	y(k-1)	6.0490E-01	9.45813556E-01
1	y(k-2)	-5.5109E-01	2.63576840E-03
2	x1(k-5)y(k-1)	5.6995E-01	1.19429351E-03
3	x1(k-3)	-2.1264E+00	7.19976894E-04
4	x1(k-2)	-2.4622E+00	6.25218348E-04
5	x1(k-4)	-1.9460E+00	6.39866753E-04
6	y(k-3)	2.3011E-01	5.98553280E-04
7	x1(k-5)^2	-2.2581E+00	4.96492047E-04
8	x1(k-6)y(k-2)	-4.3001E-01	2.22025182E-04
9	x1(k-2)^2	-3.0755E+00	3.18688092E-04
10	x1(k-1)	-1.3935E+00	3.00926941E-04
11	x1(k-5)	1.5733E+00	4.40167340E-04

With the model now available, let's start with residual analysis.

```
# get model (1-step ahead) predictions and residuals on training dataset
y_fit_predicted_centered = model.predict(X=u_fit_centered, y=y_fit_centered, steps_ahead=1)
y_fit_predicted = y_scaler.inverse_transform(y_fit_predicted_centered)
residuals_fit = y_fit - y_fit_predicted
```



`

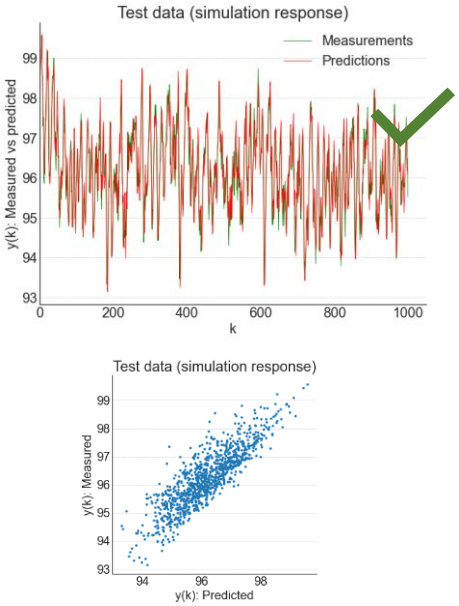


The CCF plot is not terrible but throws some doubt on the acceptability of the model. To check if the model can be useful, let's check the model's simulation response on the test dataset.

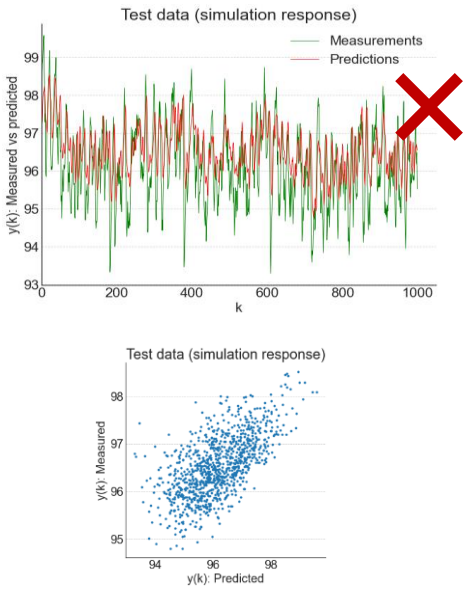
```
# infinite-step ahead predictions on test dataset
y_test_predicted_centered = model.predict(X=u_test_centered, y=y_test_centered, steps_ahead=None)
y_test_predicted = y_scaler.inverse_transform(y_test_predicted_centered)

plt.figure(), plt.plot(y_test, 'g', label='Measurements'), plt.plot(y_test_predicted, 'r', label='Predictions')
plt.title('Test data'), plt.ylabel('y(k): Measured vs predicted'), plt.xlabel('k'), plt.legend()
plt.figure(), plt.plot(y_test, y_test_predicted, '.')
plt.title('Test data (simulation response)'), plt.ylabel('y(k): Measured'), plt.xlabel('y(k): Predicted')
```

Plots below show the infinite-step ahead predictions (simulation response) for the NARX model. For comparison, an ARX model was also generated.

NARX model



ARX model



It is apparent that the model provides good simulation performance and therefore provides confidence that the model has managed to adequately capture the deterministic relationship

`

of the process. Moreover, it is not surprising that the linear model struggles to capture the relationship adequately. Polynomial degree of 2 provides reasonably good performance and therefore, we will not optimize it any further. In a generic exercise, an optimization scheme will need to be adopted for a more systematic optimal selection of model hyperparameters.

9.4 Introduction to Block-Structured Nonlinear Models

Block-structured models describe nonlinear processes through separate blocks of static nonlinearity and linear dynamics as shown in Figure 9.4. Two separate scenarios have been depicted in the figure: the nonlinearity is at the input for a Hammerstein process while for a Wiener process, the nonlinearity is at the output[78]. While this blocks-based construct may seem very simple (and restrictive), it has been successfully used to model several complex industrial processes. For example, an industrial scale MPC-controlled air separation plant was modeled via Hammerstein-Wiener model using real process data by Pattison et. al. and used for optimizing plant scheduling under varying product demands[79].

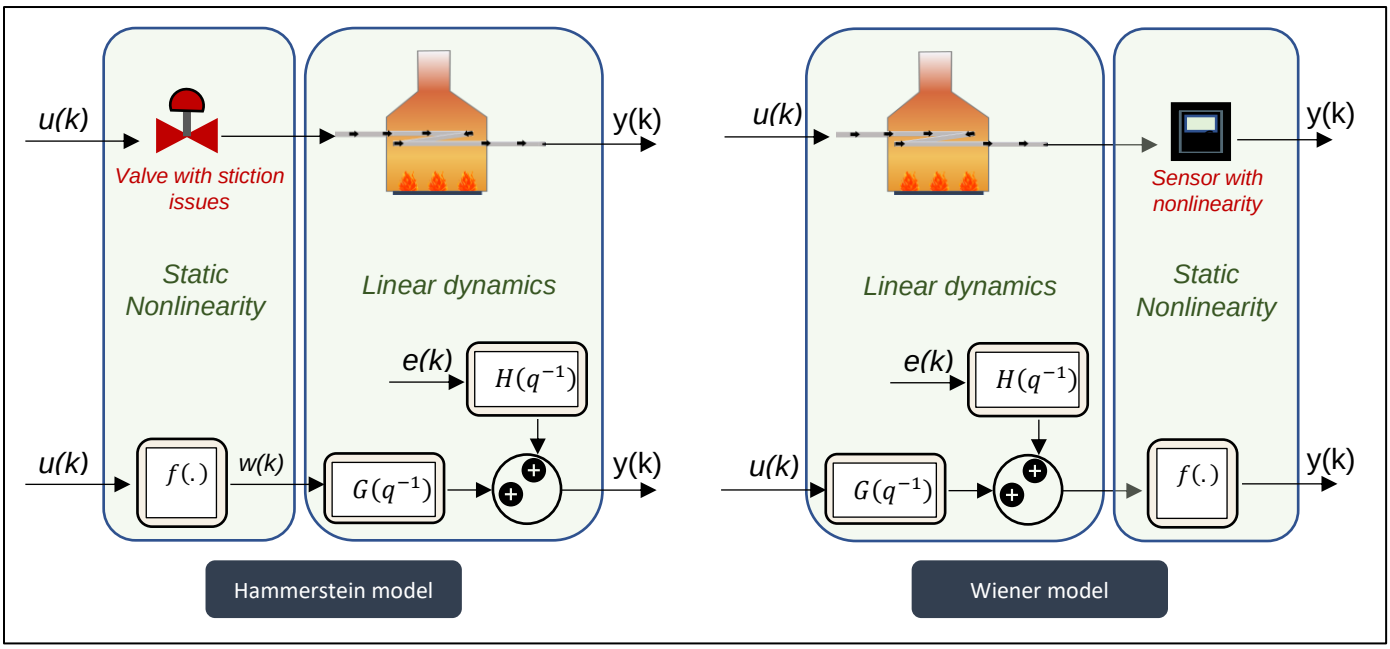


Figure 9.4: Linear-nonlinear relationships in Hammerstein and Wiener processes

[78] Other combinations of nonlinear and linear blocks (such as nonlinear-linear-nonlinear) have also been employed by SysID community.

[79] Pattison et. al., Optimal process operations in fast-changing electricity markets: framework for scheduling with low-order dynamic models and an air separation application, *Industrial & Engineering Chemistry Research*, 2016

`

Let's study Hammerstein model in more details to understand the general characteristics of block-structured models.

Hammerstein model

In Hammerstein model the inputs are first transformed by a nonlinear function and then passed through a linear dynamic process as shown in Figure 9.5. Polynomials are often used for the nonlinear function; piecewise linear functions being another popular choice. To understand this framework in more details, let's derive the difference equation form for a Hammerstein process with 2nd order polynomial and first order ARX dynamics

$$w(k) = \beta_1 u(k) + \beta_2 u^2(k)$$

$$y(k) = -a_1 y(k-1) + b_1 w(k)$$

$$\Rightarrow y(k) = -a_1 y(k-1) + b_1\beta_1 u(k-1) + b_1\beta_2 u^2(k-1) \qquad \text{eq. 1}$$

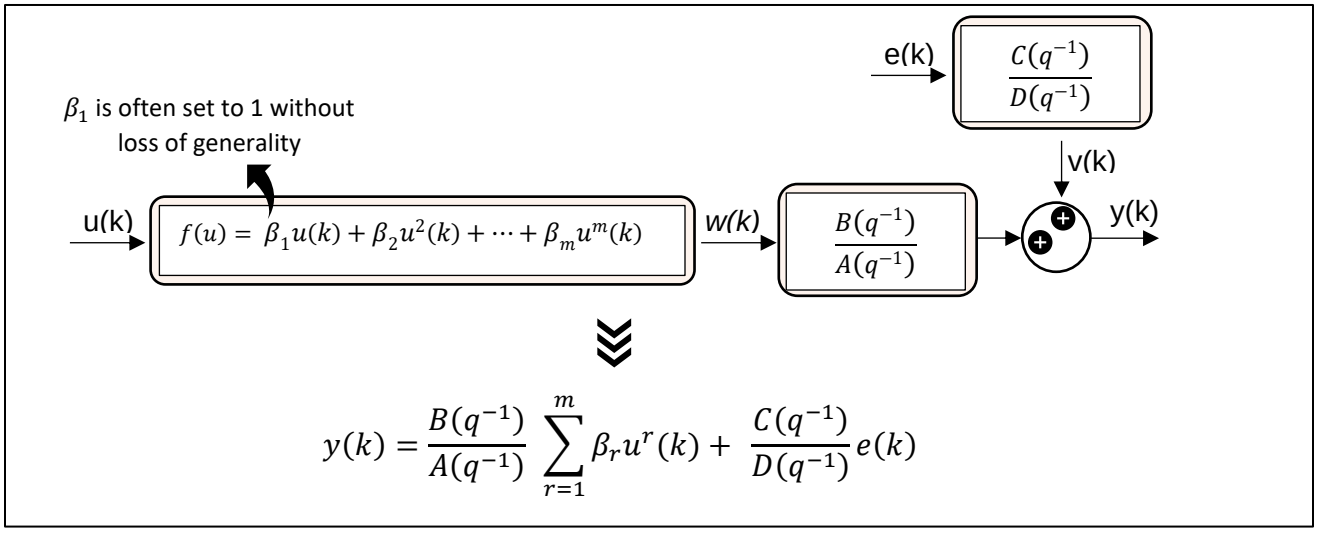


Figure 9.5: SISO Hammerstein model with polynomial nonlinearity. β_s are parameters of the nonlinear block.

There are different ways of estimating the parameters $a_1, b_1, \beta_1, \beta_2$ in Eq. 1 (and, in general, any Hammerstein model).

- *Approach 1*:

  Employ a nonlinear optimization scheme to find the parameters directly

`

- *Approach 2*:

  Redefine the parameters to derive a linear-in-the-parameters system. In Eq. 1, we can see that the system has one parameter more than necessary. Either β_1 or b_1 can be fixed to 1 without changing the relationship between *u(k)* and *y(k)*. Let's set $\beta_1 = 1$ and let $b_1\beta_2 = r_1$. Therefore,

$$y(k) = -a_1 y(k-1) + b_1 u(k-1) + r_1 u^2(k-1) \qquad \text{eq. 2}$$

  Eq. 2 can be fitted like a MISO ARX system (with $u^2(k-1)$ as the 2nd input to the system). Once b_1 and r_1 are estimated, β_2 is computed as $\frac{r_1}{b}$.

- *Approach 3*:

  This is an iterative approach wherein the linear and nonlinear block parameters are estimated separately at each iteration[80]. For our simple problem in Eq. 1, this may seem like an overkill, but it will help understand the approach clearly

  $\underline{Initialize: Set\ f(u) = u}$

  $\underline{At\ (i+1)^{th}\ iteration:}$ $\beta_1^{(i)}$ and $\beta_2^{(i)}$ *are available from i^th iteration*

  *Step 1: (*re-)estimate parameters of $A(q^{-1})$ and $B(q^{-1})$, i.e., a_1, b_1 for a fixed $f(u) = \beta_1^{(i)} u(k) + \beta_2^{(i)} u^2(k)$ by minimizing the following

$$V = \sum_{k=1}^{N} y(k) - [a_1 y(k-1) + b_1 \beta_1^{(i)} u(k-1) + b_1 \beta_2^{(i)} u^2(k-1)] \quad \leftarrow \textit{Linear problem}$$

  *Step 2: (*re-)estimate parameters of $A(q^{-1})$ and $f(u)$, i.e., a_1, β_1, β_2, for a fixed $B(q^{-1}) = b_1^{(i+1)} q^{-1}$ by minimizing the following

$$V = \sum_{k=1}^{N} y(k) - [a_1 y(k-1) + b_1^{(i+1)} \beta_1 u(k-1) + b_1^{(i+1)} \beta_2 u^2(k-1)] \quad \leftarrow \textit{Linear problem}$$

  Continue iteration if parameters not converged

[80] Y. Zhu, Multivariable System Identification for Process Control. *Elsevier*, 2001

`

Nonlinear SysID can inarguably be a daunting task for an inexperienced process modeler due to higher complexity of parameter estimation and large number of choices that lie at a modeler's discretion regarding the type of nonlinearity functions to choose from. Lack of systematic guidelines on nonlinear function selection makes things only worse. Therefore, in this chapter we looked at a couple of intuitive nonlinear modeling frameworks that have used for a variety of industrial processes. If linear models fail to provide satisfactory performance, then it can be worthwhile to try these simple nonlinear models to see if any significant improvements can be achieved. However, keep your expectations grounded - do not strive to obtain a perfect nonlinear model at the cost of project stagnation; the approximations can be good enough to be useful!

Summary

In this chapter we looked in detail two popular classical nonlinear SysID techniques, NARX and Hammerstein modeling. We worked through an application of NARX for modeling heat exchangers. With this we have completed our study of the classical modeling methods for dynamic processes. We are confident that by now you are comfortable with system identification jargon and can deftly handle most of the industrial dynamic modeling tasks with the knowledge gained so far. We will continue our study of nonlinear modeling in the next part of the book on neural networks and deep learning.

`

Addendum A1

Closed-Loop Identification: Modeling Processes Under Feedback

The modeling techniques discussed in the book till now have an assumption regarding independence between process inputs *u(k)* and disturbances *v(k)*. In other words, open-loop conditions were assumed. However, data collection for system identification under open-loop condition may not always be possible due to process safety and product quality concerns. Therefore, closed-loop identification (CLI) which refers to system identification using data collected under feedback control becomes a necessity. However, the feedback control leads to correlation between inputs and disturbances due to which some open-loop techniques (such as FIR, OE, SIM, etc.) that we have discussed so far will fail to provide unbiased models or more stringent conditions are imposed on model structure selection for unbiased parameter estimates.

In this short addendum, we will attempt to apprise ourselves of the challenges of CLI and look at ways to overcome these challenges. Specifically, the following topics will be covered

- Challenges with CLI
- Identifiability conditions for CLI
- Experiment design for CLI and direct identification approach

---

`

CLI Challenges

To guide our discussion on CLI challenges, consider Figure A1.1 which shows the conceptual differences between an open loop and a closed-loop process with a linear feedback controller.

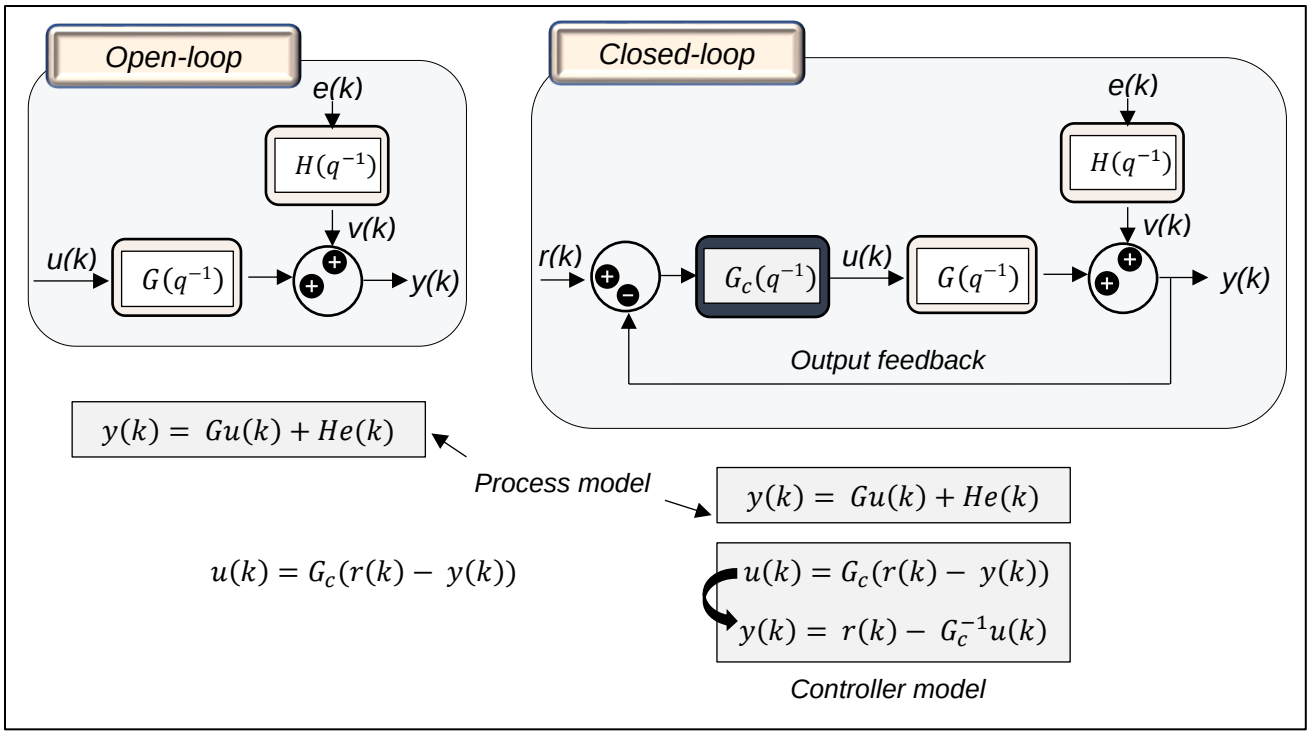


Figure A1.1: SISO open loop process and closed-loop process with linear controller

Let's further assume that there is no excitation in the reference setpoints, i.e., *r(k)* = 0. Under such conditions, there are two governing equations of the process which are competing for identification as shown below

The two equations are competing for identification

$y(k) = G_p u(k) + He(k)$ ← process model

$y(k) = r(k) - G_c^{-1}u(k) = -G_c^{-1}u(k)$ ← controller model

Due to the presence of noise disturbances in the process model, it is the (inverse of) controller model that will get 'picked up' during SysID[81]. Even if *r(k)* is externally excited, depending on the 'size' of *r(k)* and *e(k)*, SysID may pick G_p, G_c^{-1}, or some combination of the two model.

[81] Under certain conditions (such as the controller being nonlinear like MPC), it may be possible to identify G_p without external excitation in *r(k)*.

`

Data collection for CLI and Direct Identification Approach

In closed-loop process, *u(k)* is decided by the controller and therefore it may not be possible to manually induce persistent excitations in *u(k)*. To facilitate identifiability of process model, a common alternative is to persistently excite the reference signal *r(k)*. If even this is not possible then the only resort if to use the historical process data with the hope that reference setpoints have sufficient variability.

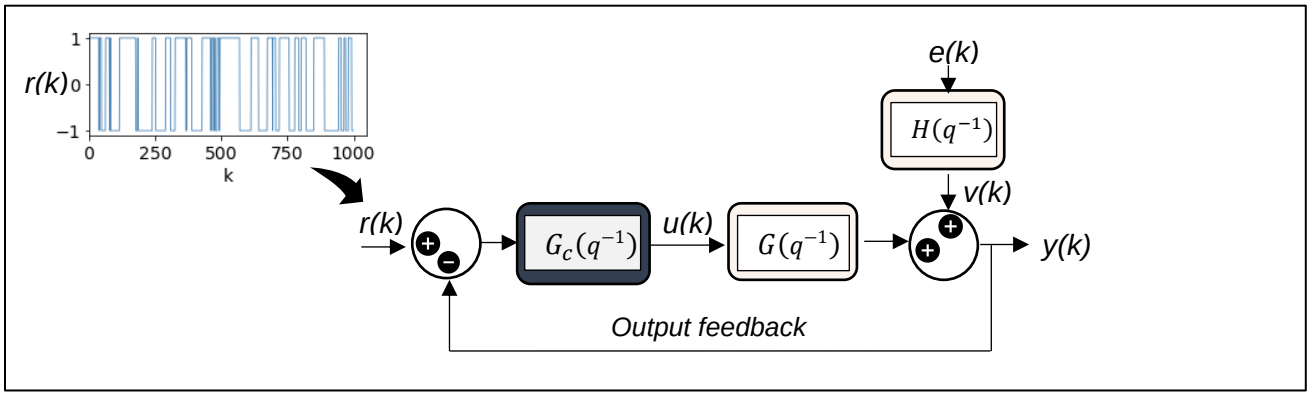


Figure A1.2: Closed-loop process with persistently excited reference signal

When the closed-loop process is excited externally through *r(k)*, the common CLI approach (called direct identification approach) directly builds a model between *y(k)* and *u(k)* (just like we did for open-loop systems) without utilizing any information about the controller or the reference signals.

Model structure selection for CLI

Assuming sufficiently excited r(k), PEM models provide unbiased models with closed-loop data if both $G(q^{-1})$ and $H(q^{-1})$ are correctly chosen. Note that the correct noise model structure was not a necessity for open-loop case.

`

Part 3

Artificial Neural Networks & Deep Learning

`

Chapter 10

Artificial Neural Networks: Handling Complex Nonlinear Systems

It won't be an exaggeration to say that artificial neural networks (ANNs) are currently the most powerful modeling construct for describing generic nonlinear processes. ANNs can capture any kind of complex nonlinearities, don't impose any specific process characteristics, and don't demand specification of process insights prior to model fitting. Furthermore, several recent technical breakthroughs and computational advancements have enabled (deep) ANNs to provide remarkable results for a wide range of problems. Correspondingly, ANNs have re(caught) the fascination of data scientists and the process industry is witnessing a surge in successful applications of ML-based process control, predictive maintenance, and inferential modeling involving ANN-based system identification.

For SysID, ANNs are commonly used in NARX framework wherein an ANN model is fitted to find the nonlinear relationship between current output and past input/output measurements. In such applications, the FFNN (feed-forward neural networks) is invariably employed. However, for dynamic modeling, a specialized type of network called RNN (recurrent neural network) exist which is designed to be aware of the temporal nature of dynamic process data. Consequently, RNNs tend to be more parsimonious than FFNN. We will see SysID applications of both these popular architectures in this chapter.

There is no doubt that ANNs have proven to be monstrously powerful. However, it is not easy to tame this monster. If the model hyperparameters are not set judiciously, it is very easy to end up with disappointing results. The reader is referred to Part 3 of Book 1 of this series for a detailed exposition on ANN training strategies and different facets of ANN models. In this chapter, SysID relevant concepts are sufficiently described to enable an uninitiated reader to learn how to use ANNs. Specifically, the following topics are covered

- Introduction to ANN, FFNN, RNN, LSTM
- Heat exchanger SysID via FFNN-based NARX
- Heat exchanger SysID via LSTM

`

10.1 ANN: An Introduction

Artificial neural networks (ANNs) are nonlinear empirical models which can capture complex relationships between input-output variables via supervised learning or recognize data patterns via unsupervised learning. Architecturally, ANNs were inspired by human brain and are a complex network of interconnected neurons as shown in Figure 10.1. An ANN consists of an input layer, a series of hidden layers, and an output layer. The basic unit of the network, neuron, accepts a vector of inputs from the source input layer or the previous layer of the network, takes a weighted sum of the inputs, and then performs a nonlinear transformation to produce a single real-valued output. Each hidden layer can contain any number of neurons.

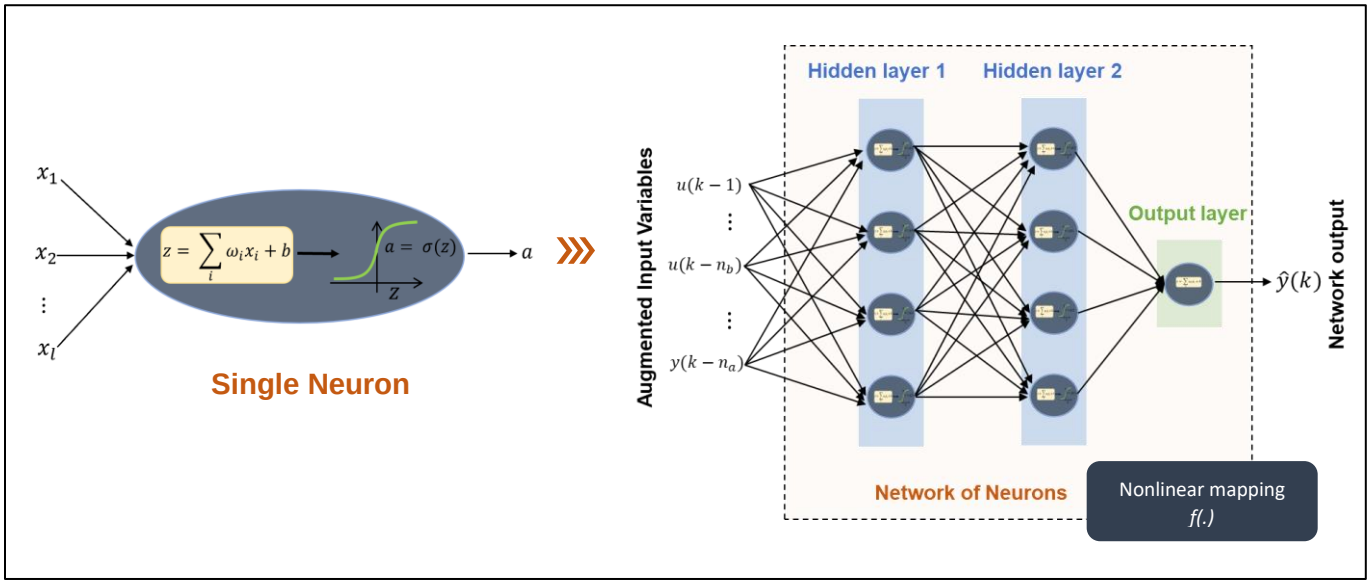


Figure 10.1: Architecture of a single neuron and feedforward neural network with 2 hidden layers for a SISO (single input single output) process within NARX framework

The network shown in Figure 10.1 is an example of a fully-connected feed-forward neural network (FFNN), the most common type of ANN. In FFNN, signals flow in only one direction, from the input layer to the output layer via hidden layers. Neurons between consecutive layers are connected fully pairwise and neurons within a layer are not connected.

What is deep learning

In a nutshell, using an ANN with a large number of hidden layers to find relationship/pattern in data is deep learning (technically, ≥ 2 hidden layers implies a deep neural network (DNN)). Several recent algorithmic innovations have overcome the model training issues for DNNs which have resulted in the DNN-led AI revolution we are witnessing today.

`

10.2 Modeling Heat Exchangers using FFNN-NARX

To learn how to setup and fit a feed-forward neural network, we will reuse the heat-exchanger dataset from Chapter 9 and build a FFNN-NARX model between the incoming liquid flow and outgoing liquid temperature. The code for importing basic packages and loading data into the workspace remains the same.

```
# split dataset into fitting and test dataset
u_fit = u[:3000, 0:1]; u_test = u[3000:,0:1]
y_fit = y[:3000, 0:1]; y_test = y[3000:, 0:1]

# scale data before model fitting
u_scaler = StandardScaler()
u_fit_scaled = u_scaler.fit_transform(u_fit); u_test_scaled = u_scaler.transform(u_test)
y_scaler = StandardScaler()
y_fit_scaled = y_scaler.fit_transform(y_fit); y_test_scaled = y_scaler.transform(y_test)
```

As shown in Figure 10.1, the input vector that gets supplied to the network for *y(k)*'s prediction includes all the lagged variables. Therefore, we need to now create the augmented input variables.

```
# rearrange u and y data into (# observation samples, # lagged regressors) form
u_lags = 9; y_lags = 3
u_augument_fit_scaled = []
y_augment_fit_scaled = []

for sample in range(max(u_lags, y_lags), u_fit.shape[0]):
    row = np.hstack((u_fit_scaled[sample-u_lags:sample,0], y_fit_scaled[sample-y_lags:sample,0]))
    u_augument_fit_scaled.append(row)
    y_augment_fit_scaled.append(y_fit_scaled[sample])

# conversion: convert list into array
u_augument_fit_scaled = np.array(u_augument_fit_scaled)
y_augment_fit_scaled = np.array(y_augment_fit_scaled)
```

We are now ready to define and fit our neural network.

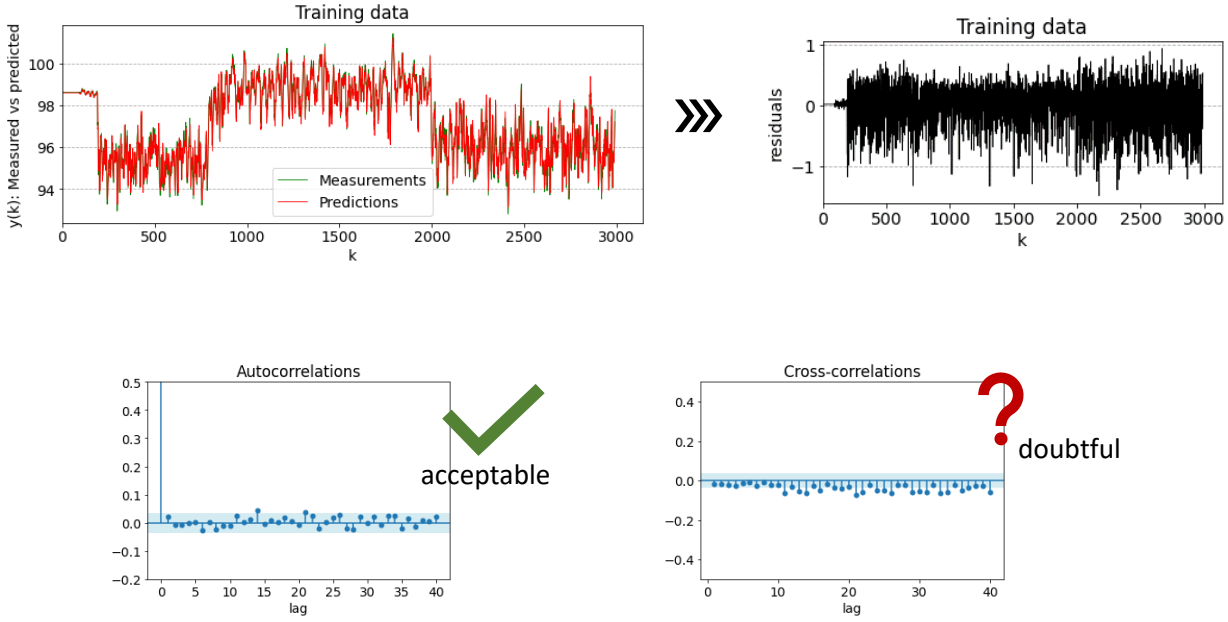
```
# import Keras libraries
from tensorflow.keras import Sequential
from tensorflow.keras.layers import Dense
```

`

```
# define model
model = Sequential()
model.add(Dense(14, activation='relu', kernel_initializer='he_normal', input_shape=(12,)))
 # 14 neurons in 1st hidden layer; this hidden layer accepts data from a 12-dimensional input
model.add(Dense(7, activation='relu', kernel_initializer='he_normal'))
# 7 neurons in 2nd layer
model.add(Dense(1)) # output layer

# compile and fit model
model.compile(loss='mse', optimizer='Adam') # mean-squared error is to be minimized
model.fit(u_augument_fit_scaled, y_augment_fit_scaled, epochs=250, batch_size=125)
```

Let's now get the model predictions and residuals on the training dataset and see how our residuals look like.

```
# get model (1-step ahead) predictions and residuals on training dataset
y_augment_fit_scaled_pred = model.predict(u_augument_fit_scaled)
y_augment_fit_pred =  y_scaler.inverse_transform(y_augment_fit_scaled_pred)
y_augment_fit = y_scaler.inverse_transform(y_augment_fit_scaled)
residuals_fit = y_augment_fit - y_augment_fit_pred
```



Results are similar to that seen for polynomial-NARX model. The slight breaches of the confidence interval in CCF plot indicates the need to add more lagged variables as model inputs. However, as a modeler, you have to make a judgement between making the model

`

more complex versus assessing the acceptability of slight model inaccuracies. To check if the model can be useful, let's again check the model's simulation response on the test dataset. We first need to arrange our test data in appropriate form.

```
# rearrange u and y data into (# sequence samples, # lagged regressors) form
u_augument_test_scaled = []
y_augment_test_scaled = []

for sample in range(max(u_lags, y_lags), u_test.shape[0]):
    row = np.hstack((u_test_scaled[sample-u_lags:sample,0], y_test_scaled[sample-y_lags:sample,0]))
    u_augument_test_scaled.append(row)
    y_augment_test_scaled.append(y_test_scaled[sample])

# conversion: convert list into array
u_augument_test_scaled = np.array(u_augument_test_scaled)
y_augment_test_scaled = np.array(y_augment_test_scaled)
```

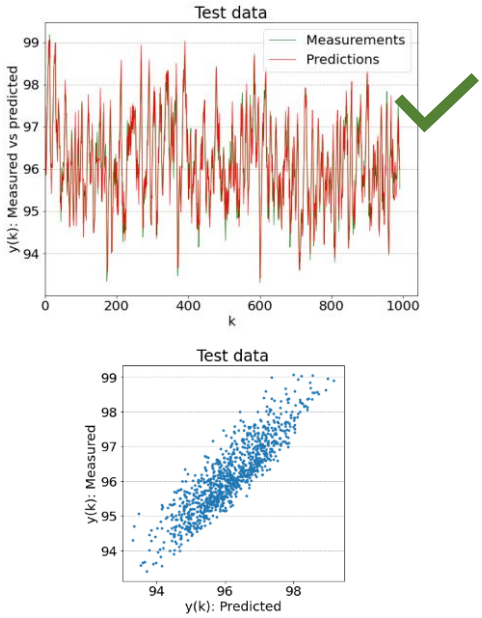
While generating the 1-step ahead predictions on validation dataset is straightforward, the code below shows how to generate simulation response or the infinite-step ahead predictions. It is easy to see that we are simply using the past model predictions to generate subsequent predictions.

```
# infinite-step ahead predictions on test dataset [first u_lags samples are used as initial conditions]
y_augment_test_scaled_sim = np.copy(y_test_scaled)

for sample in range(u_lags, len(u_test),1):
    regressorVector = np.hstack((u_test_scaled[sample-u_lags:sample,0],
                                 y_augment_test_scaled_sim[sample-y_lags:sample,0]))
    regressorVector = regressorVector[None,:]
    sim_response = model.predict(regressorVector)
    y_augment_test_scaled_sim[sample] = sim_response

y_augment_test_sim =  y_scaler.inverse_transform(y_augment_test_scaled_sim)
```

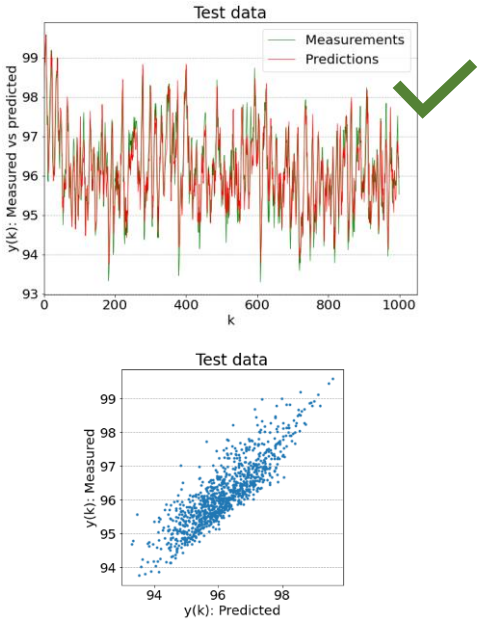
Plots below show the 1-step ahead and the infinite-step ahead predictions (simulation response) for the FFNN-NARX model.

`

1-step ahead



Infinite-step ahead



By now we are informed enough to not get very excited with good 1-step ahead predictions. However, it is encouraging to see the good simulation response, which although not as good as 1-step ahead predictions[82], is able to match the ups and downs in the raw measurements.

[82] You may notice some minor variations between your results and those shown in this chapter due to random nature of network fitting algorithm.

`

Understanding flow of data in a FFNN

If you execute the command *model.summary()* for the model developed in the previous section, you will be presented with the following information which details the number of parameters to be estimated in each layer of the network. Let's understand how these numbers are obtained which will help clarify the flow of information in FFNN.

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_3 (Dense)              (None, 14)                182

dense_4 (Dense)              (None, 7)                 105

dense_5 (Dense)              (None, 1)                 8

=================================================================
Total params: 295
Trainable params: 295
Non-trainable params: 0
_________________________________________________________________
```

We know that each augmented input sample is 12-dimensional ($u_{aug} \in R^{12}$). In the forward pass (also called forward propagation), the augmented input is first processed by the neurons of the first hidden layer. In the j^{th} neuron of this layer, the weighted sum of the inputs are non-linearly transformed via an activation function, *g*

$$a_j = g(w_j^T u_{aug} + b_j)$$

$w_j \in R^{12}$ are the weights applied to the inputs and b_j is the bias added to the sum. Thus, each neuron has 13 parameters (12 weights and a bias) leading to 182 parameters for all the 14 neurons of the 1st layer that need to be estimated. Outputs of all the 14 neurons form vector $a^{(1)} \in R^{14}$

$$a^{(1)} = g^{(1)}(W^{(1)} u_{aug} + b^{(1)})$$

where each row of $W^{(1)} \in R^{14\times12}$ contains the weights of a neuron. The same activation function is used by all the neurons of a layer. $a^{(1)}$ becomes the input to the 2nd hidden layer.

$$a^{(2)} \in R^7 = g^{(2)}(W^{(2)} a^{(1)} + b^{(2)})$$

where $W^{(2)} \in R^{7\times14}$. Each neuron in the 2nd layer has 14 weights and a bias parameter, leading to 105 parameters in the layer. The final output layer had a single neuron with 8 parameters and no activation function, giving the network output as follows

$$a^{(3)} = \hat{y} = (w^{(3)})^T a^{(2)} + b^{(3)}$$

where $w^{(3)} \in R^7$.

`

10.3 RNN: An Introduction

Recurrent neural networks (RNNs) are ANNs for dealing with sequential data, where the order of occurrence of data holds significance. In the FFNN-based NARX model that we studied in the previous section, there is no implicit or explicit specification of the fact that the data is temporal/sequential in nature, i.e., *u(k-2)* comes before *u(k-1)* for example. There is no efficient mechanism to specify this temporal order of data in a FFNN. RNNs accomplish this by processing elements of a sequence recurrently and storing a hidden state that summarizes the past information during the processing. The basic unit in a RNN is called a RNN cell which simply contains a layer of neurons. Figure 10.2 shows the architecture of a RNN consisting of a single cell and how it processes a data sequence with ten samples.

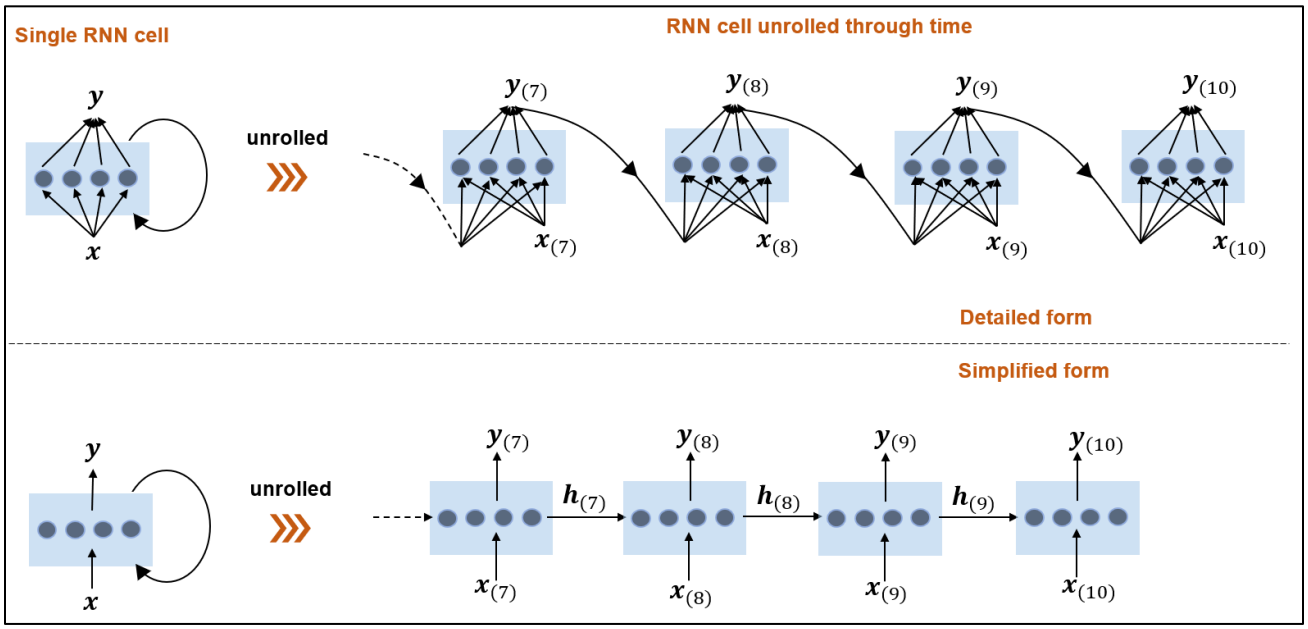


Figure 10.2: Representation of an RNN cell in rolled and unrolled format. The feedback signal in rolled format denotes the recurrent nature of the cell. Here, the hidden state (*h*) is assumed to be same as intermediate output (*y*). $h_{(0)}$ is usually taken as zero vector.

We can see that the ten samples are processed in the same order of their occurrence and not at once. An output, $y_{(i)}$, is generated at the i^{th} step and then fed to the next step for processing along with $x_{(i+1)}$. By way of this arrangement, $y_{(i+1)}$ is a function of $x_{(i+1)}$ and $y_{(i)}$. Since, $y_{(i)}$ itself is a function of $x_{(i)}$ and $y_{(i-1)}$, $y_{(i+1)}$ effectively becomes a function of $x_{(i+1)}$, $x_{(i)}$, and $y_{(i-1)}$. Continuing the logic further implies that the final sequence output, $y_{(10)}$, is a function of all ten inputs of the sequence, that is, $x_{(10)}, x_{(9)}, \ldots, x_{(1)}$. This 'recurrent' mechanism leads to efficient capturing of temporal patterns in data.

`

RNN outputs

If the neural layer in the RNN cell in Figure 10.2 contains *n* neurons (*n* equals 4 in the shown figure), then each $y_{(i)}$ or $h_{(i)}$ is a n-dimensional vector. For simple RNN cells, $y_{(i)}$ equals $h_{(i)}$. Let *x* be a m-dimensional vector. At any i^{th} step, we can write the following relationship

$$y_{(i)} = g(W_x x_{(i)} + W_y y_{(i-1)} + b)$$

where $W_x \in R^{n \times m}$ with each row containing the weight parameters of a neuron as applied to *x* vector, $W_y \in R^{n \times n}$ with each row containing the weight parameters of a neuron as applied to *y* vector, $b \in R^n$ contains the bias parameters, and g denotes the activation function. The same neural parameters are used at each step.

If all the outputs of the sequence are of interest, then it is called sequence-to-sequence or many-to-many network. However, very often only the last step output is needed, leading to sequence-to-vector or many-to-one network. Moreover, the last step output may need to be further processed and so a FC layer is often added. Figure 10.3 shows one such topology.

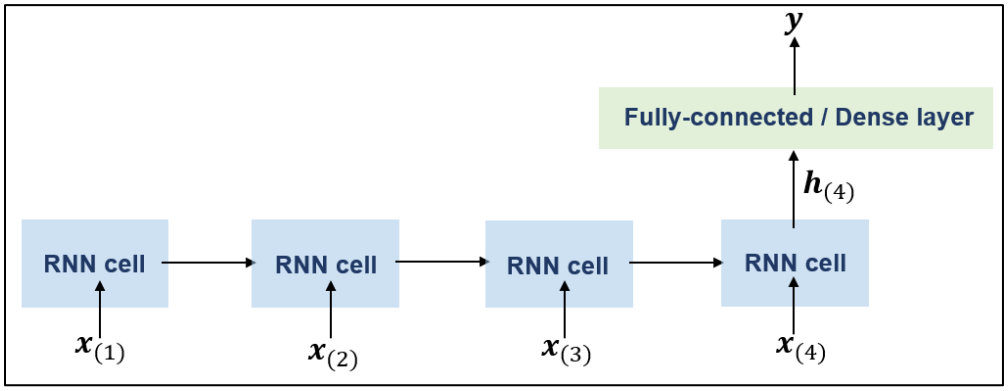


Figure 10.3: Sequence to vector RNN topology with a FC layer

RNN-based NARX topology

Figure 10.4 below shows how a potential RNN-based NARX architecture may look like for a SISO process where three lagged measurements are used to make one-step ahead predictions. We can observe that all the lagged measurements are not processed simultaneously by an RNN cell which often results in less number of model parameters when compared to a FFNN-based NARX model.

`

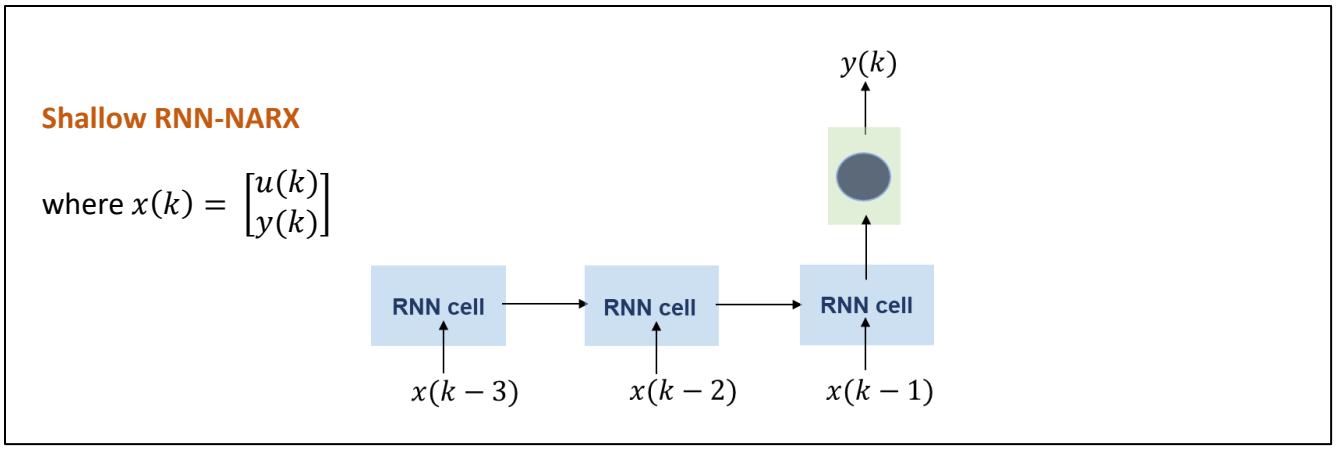


Figure 10.4: RNN-NARX topology for a SISO process. [Note that if the model input vectors (*x*) are simply the process inputs (*u*) then the architecture becomes NFIR (nonlinear FIR)].

LSTM networks

RNNs are powerful dynamic models, however, the vanilla RNN (with a single neural layer in a cell) introduced before faces difficulty learning long-term dependencies, i.e., when number of steps in a sequence is large (~ ≥10). This happens due to the vanishing gradient problem during gradient backpropagation. To overcome this issue, LSTM (Long Short-Term Memory) cells have been devised which are able to learn very long-term dependencies (even greater than 1000) with ease. Unlike vanilla RNN cells, LSTM cells have 4 separate neural layers as shown in Figure 10.5. Moreover, in a LSTM cell, the internal state is stored in two separate vectors, $h_{(t)}$ or hidden state and $c_{(t)}$ or cell state. Both these states are passed from one LSTM cell to the next during sequence processing. $h_{(t)}$ can be thought of as the short-term state/memory and $c_{(t)}$ as the long-term state/memory (we will see later why), and hence the name LSTM.

The vector outputs of the FC layers interact with each-other and the long-term state via three 'gates' where element-wise multiplications occur. These gates control what information go into the long-term and short-term states at any sequence processing step. While we will understand the mathematical details later, for now, if suffices to know the following about these gates:

- Forget gate determines what parts of long-term state, $c_{(t)}$, are retained and erased
- Input gate determines what parts of new information (obtained from processing of $x_{(t)}$ and $h_{(t-1)}$) are stored in long-term state
- Output gate determines what parts of long-term state are passed on as short-term state

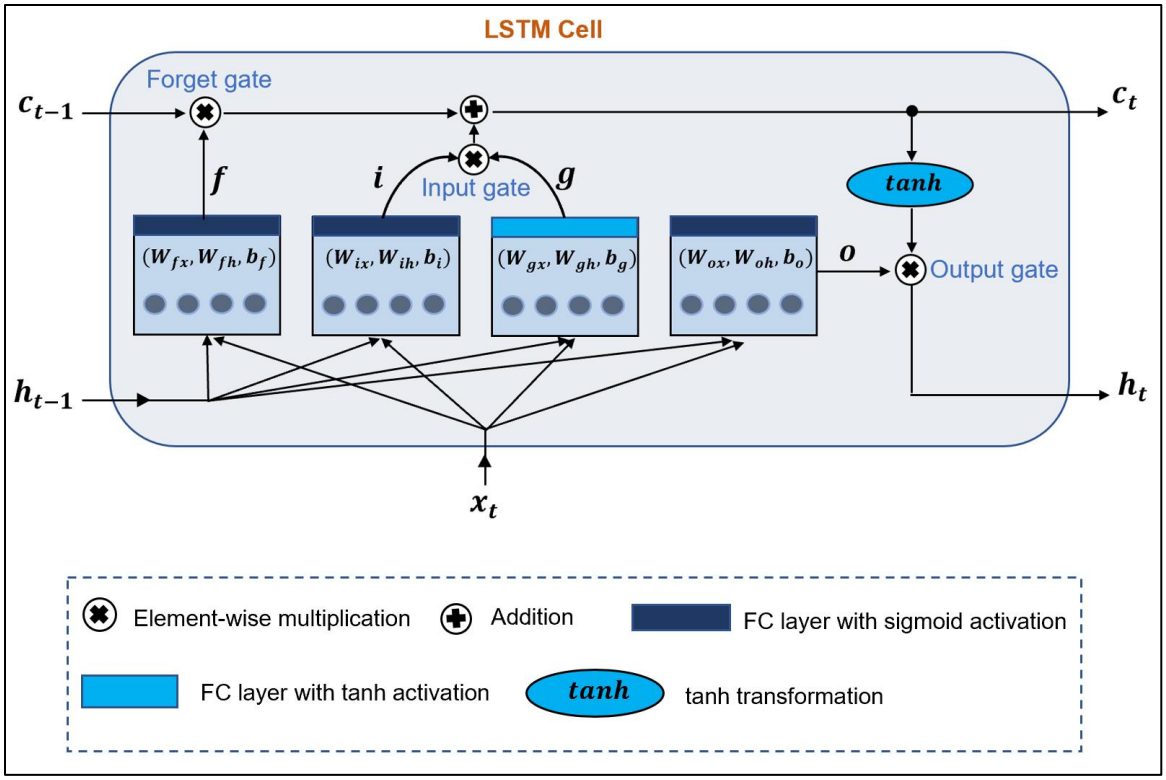


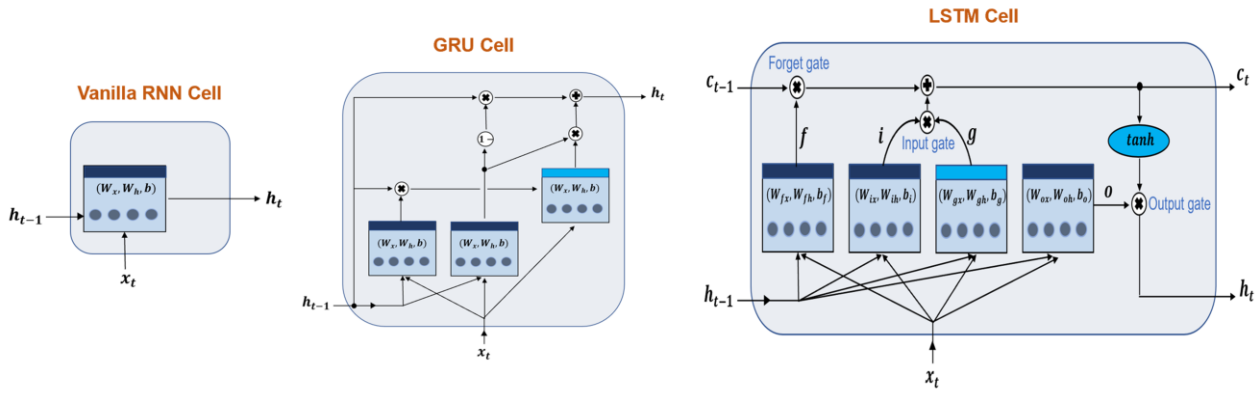
Figure 10.5: Architecture of an LSTM cell. Three FC layers use sigmoid activation functions and one FC layer uses tanh activation function. Each of these neural layers have its own parameters W_x, W_h, and b

This flexibility in being able to manipulate what information are passed down the chain during sequence processing is what makes LSTM networks so successful at capturing long-term patterns in sequential datasets. Consequently, LSTM networks are the default RNNs now-a-days. In the next section we will see a quick application of RNN for system identification using LSTM cells.

`

Vanilla RNN cell vs GRU cell vs LSTM cell

There is another popular variant of RNN cell, called GRU cell. As shown in the illustration below, GRU cell is simpler than LSTM cell. GRU cell has 3 neural layers and its internal state is represented using a single vector, $\boldsymbol{h}_{(t)}$. For several common tasks, GRU cell-based RNN seems to provide similar performance as LSTM cell-based RNN and therefore, it is slowly gaining more popularity.



10.4 Modeling Heat Exchangers using LSTM

To show how to train a LSTM network, we will re-use the heat-exchanger dataset. We will also use *early stopping* to optimize the number of *epochs* using validation dataset.

```
# read data
u_fit = u[:3000,0:1]; u_val = u[3000:,0:1]
y_fit = y[:3000,0:1]; y_val = y[3000:,0:1]

# scale data before model fitting
u_scaler = StandardScaler()
u_fit_scaled = u_scaler.fit_transform(u_fit); u_val_scaled = u_scaler.transform(u_val)
y_scaler = StandardScaler()
y_fit_scaled = y_scaler.fit_transform(y_fit); y_val_scaled = y_scaler.transform(y_val)

X_fit_scaled = np.hstack((u_fit_scaled, y_fit_scaled)) # @ Figure 10.4
X_val_scaled = np.hstack((u_val_scaled, y_val_scaled))
```

`

As shown in Figure 10.4, the input that gets supplied to the network for *y(k)*'s prediction is a sequence of the lagged measurements. Therefore, we need to now create these sequences. Currently, the shape of *X* array is {# of samples, # of features}. For RNN, this input data matrix needs to be converted into the shape {# of sequence samples, # of time steps, # of features} such that each entry (of shape {# of time steps, # of features}) along the 0^{th} dimension is a complete sequence of past data which is used to make prediction of temperature values. Here, the # of time steps is taken to be 9 and # of features equals 2.

```
# rearrange X data into (# sequence samples, # time steps, # features) form
nTimeSteps = 9
X_fit_scaled_sequence = []; X_val_scaled_sequence = []
y_fit_scaled_sequence = []; y_val_scaled_sequence = []

for sample in range(nTimeSteps, X_fit_scaled.shape[0]):
   X_fit_scaled_sequence.append(X_fit_scaled[sample-nTimeSteps:sample, :])
   y_fit_scaled_sequence.append(y_fit_scaled[sample])

for sample in range(nTimeSteps, X_val_scaled.shape[0]):
   X_val_scaled_sequence.append(X_val_scaled[sample-nTimeSteps:sample, :])
   y_val_scaled_sequence.append(y_val_scaled[sample])

# convert list of (time steps, features) arrays into (samples, time steps, features) array
X_fit_scaled_sequence, y_fit_scaled_sequence = np.array(X_fit_scaled_sequence),
                                               np.array(y_fit_scaled_sequence)
X_val_scaled_sequence, y_val_scaled_sequence = np.array(X_val_scaled_sequence),
                                               np.array(y_val_scaled_sequence)
```

Basically, each block of 9 continuous rows in (scaled) *X* array becomes a sequence. The topology of RNN that we will build is same as that shown in Figure 10.4, except for the number of time-steps. Like we did for FFNN modeling, we will import relevant Keras libraries. In the code below, a LSTM RNN layer is followed by a dense layer.

```
# import Keras libraries
from tensorflow.keras import Sequential
from tensorflow.keras.layers import LSTM
from tensorflow.keras.layers import Dense
from tensorflow.keras import regularizers

# define model
model = Sequential()
model.add(LSTM(units=6, kernel_regularizer=regularizers.L1(0.001), input_shape=(nTimeSteps,2)))
# LSTM cell with 6 neurons in each of the 4 neural layers
model.add(Dense(units=1))
# 1 neuron in output layer
```

`

The above 3-line code completely defines the structure of the required RNN. The single neuron in the output layer does the job of transforming the 6th dimensional hidden state vector from the last step of RNN layer into a scalar value. The model summary below shows the number of model parameters in each layer.

```
Layer (type)                 Output Shape              Param #
=================================================================
lstm (LSTM)                  (None, 6)                 216

dense (Dense)                (None, 1)                 7

=================================================================
Total params: 223
Trainable params: 223
Non-trainable params: 0
_________________________________________________________________
```

In vanilla RNN cell, we saw that $W_x \in R^{n\times m}$, $W_h \in R^{n\times n}$ and, $b \in R^n$. This leads to total number of parameters equal to $mn + n^2 + n$. In an LSTM cell, each of the 4 neural layers have their own set of these parameters. Therefore an LSTM cell has $4(mn + n^2 + n)$ model parameters. In this example, we have m = 2 and n = 6; therefore we ended up with 4(12 + 36 + 6) = 216 model parameters in the LSTM layer.

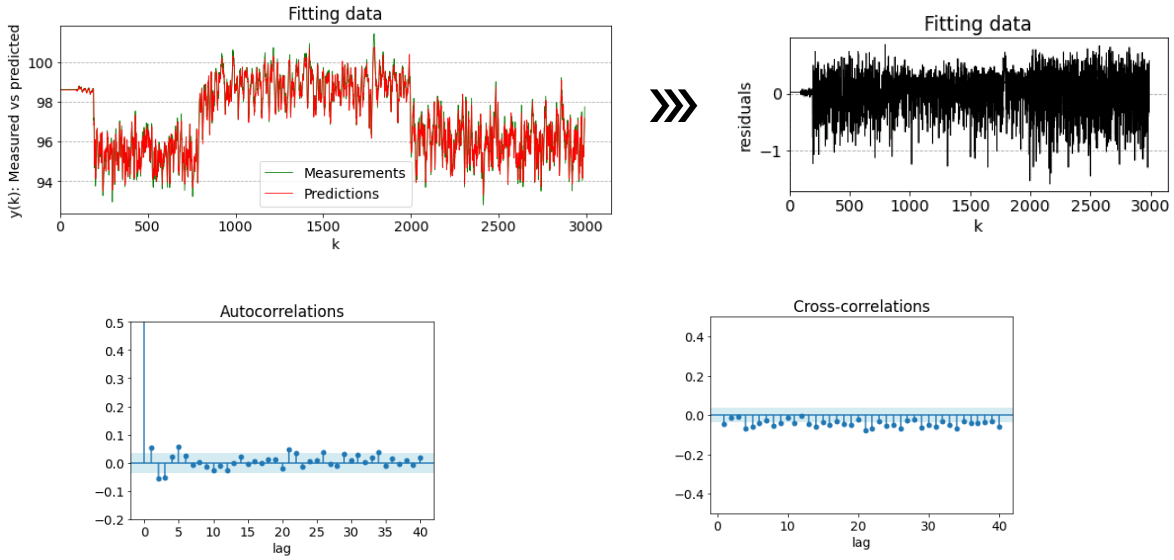
Next we compile and fit the model.

```
# compile and fit model with early stopping
from tensorflow.keras.callbacks import EarlyStopping

model.compile(loss='mse', optimizer='Adam')
es = EarlyStopping(monitor='val_loss', patience=10, restore_best_weights=True)
history = model.fit(X_fit_scaled_sequence, y_fit_scaled_sequence, epochs=250, batch_size=125,
                    validation_data=(X_val_scaled_sequence, y_val_scaled_sequence), callbacks=[es])
```

Let's now get the model predictions and residuals on the fitting dataset and see how our residuals look like.

```
# get model (1-step ahead) predictions and residuals on fitting dataset
y_fit_scaled_sequence_pred = model.predict(X_fit_scaled_sequence)
y_fit_sequence_pred =  y_scaler.inverse_transform(y_fit_scaled_sequence_pred)
y_fit_sequence = y_scaler.inverse_transform(y_fit_scaled_sequence)
residuals_fit = y_fit_sequence - y_fit_sequence_pred
```

`

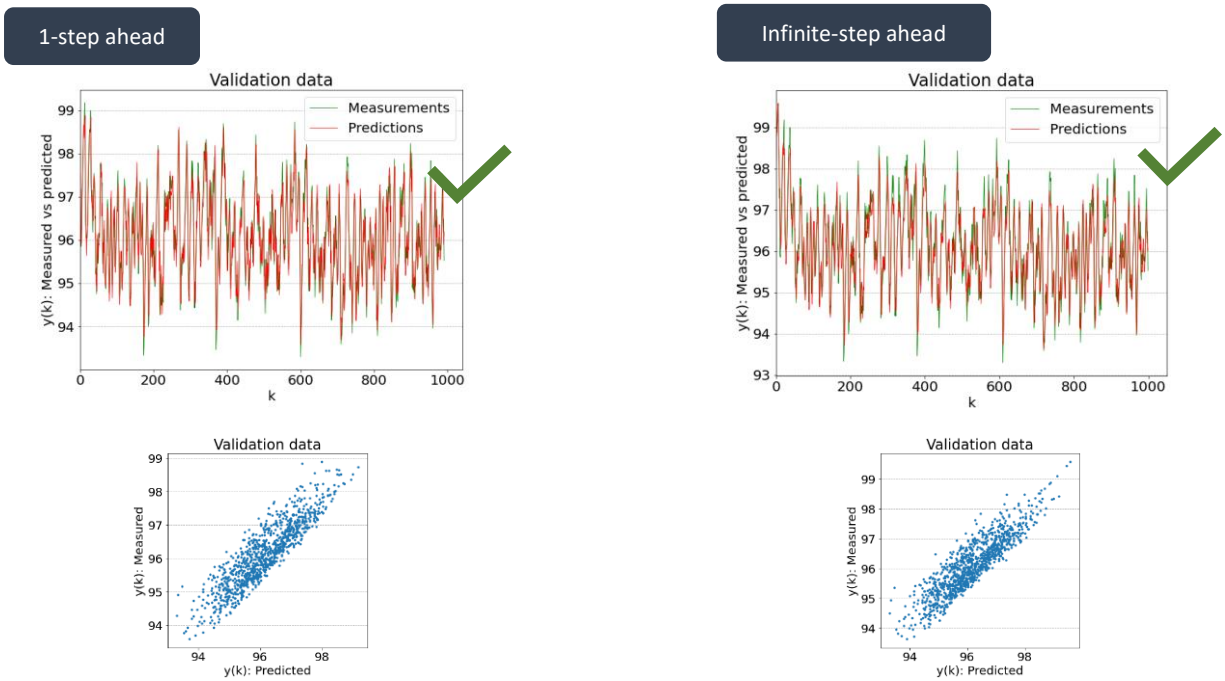


Results are similar to that seen for FFNN-NARX model. Let us now check its performance on validation dataset. We again pay emphasis to the infinite-step ahead predictions.

```
# infinite-step ahead predictions [first nTimeSteps samples are used as initial conditions]
y_val_scaled_sim = np.copy(y_val_scaled)

for sample in range(nTimeSteps, X_val_scaled.shape[0]):
    X_val_scaled_sim = np.hstack((u_val_scaled, y_val_scaled_sim))
    inputSequence = X_val_scaled_sim[sample-nTimeSteps:sample, :]
    inputSequence = inputSequence[None,:,:]
    sim_response = model.predict(inputSequence)
    y_val_scaled_sim[sample] = sim_response

y_val_pred_sim =  y_scaler.inverse_transform(y_val_scaled_sim)
```

Plots below show the 1-step ahead and the infinite-step ahead predictions (simulation response) for the LSTM-NARX model.

`



Ideally, we should check the predictions on a dataset completely separate from fitting and validation datasets; but we are checking it on validation dataset in the current case-study just to be consistent with the assessment dataset used for FFNN-NARX and polynomial-NARX (from Chapter 9).

This concludes our quick peep into the world of artificial neural networks. We are guilty of condensing a lot of concepts into a single chapter. However, you should hopefully now be comfortable developing ANN-based solutions for your nonlinear SysID needs. Do keep in mind that we only barely scratched the surface of the vast field of ANNs. Nonetheless, you should be able to navigate the ANN literature comfortably now and build upon the knowledge you have gained in this chapter.

Summary

In this chapter we studied the application of ANNs for dynamic systems. We first looked at how to build FFNN-based NARX models and saw its application for heat-exchanger modeling. Then, we learnt the remarkable capabilities of LSTM networks in handling temporal data. After working out these examples, you would have gained a very good understanding on how to setup the ANN-based solution for your specific problems. With this we will sign-off now and wish you all the best in your ML journey.

`

End of the book

`

By the process data scientists, for the process data scientists, of process data science

Machine Learning in Python for Dynamic Process Systems

This book is designed to help readers gain a working-level knowledge of machine learning-based modeling techniques for dynamic processes. Readers can leverage the concepts learned to build advanced solutions for process monitoring, soft sensing, predictive maintenance, and process control for dynamic systems. The application-focused approach of the book is reader friendly and easily digestible to the practicing and aspiring process engineers, and data scientists. Upon completion, readers will be able to confidently navigate the system identification literature and make judicious selection of modeling approaches suitable for their problems.

The following topics are broadly covered:

- *Exploratory analysis of dynamic dataset*
- *Best practices for dynamic modeling*
- *Linear and discrete-time classical parametric and non-parametric models*
- *State-space models for MIMO systems*
- *Nonlinear system identification and closed-loop identification*
- *Neural networks-based dynamic process modeling*

www.MLforPSE.com