Design of a Multi-Agent System for Process Monitoring and Supervision

Onur Dogu

Louisiana State University and Agricultural and Mechanical College

Follow this and additional works at: https://digitalcommons.lsu.edu/gradschool_theses

Part of the Chemical Engineering Commons

Recommended Citation
Dogu, Onur, "Design of a Multi-Agent System for Process Monitoring and Supervision" (2014). LSU Master's Theses. 125.
https://digitalcommons.lsu.edu/gradschool_theses/125

This Thesis is brought to you for free and open access by the Graduate School at LSU Digital Commons. It has been accepted for inclusion in LSU Master's Theses by an authorized graduate school editor of LSU Digital Commons. For more information, please contact gradetd@lsu.edu.
DESIGN OF A MULTI-AGENT SYSTEM FOR PROCESS MONITORING AND SUPERVISION

A Thesis

Submitted to the Graduate Faculty of the Louisiana State University and Agricultural and Mechanical College in partial fulfillment of the requirements for the degree of Master of Science

in

The Cain Department of Chemical Engineering

by

Onur Dogu
B.S., Middle East Technical University, 2011
May 2014
To my beloved parents, Serap and Mustafa Dogu, for providing me all the opportunities, love and support that made me the person I am.
ACKNOWLEDGEMENTS

I would like to thank my advisor Prof. Jose A. Romagnoli for his guidance, supervision, and support throughout my research process. I am grateful for the Turkish Fulbright Commission and Institute of International Education (IIE) for awarding me the Fulbright Scholarship during my studies in the United States. Finally, I want to thank my research group peers for all their help and support.
# TABLE OF CONTENTS

ACKNOWLEDGEMENTS ........................................................................................................ iii

ABSTRACT .......................................................................................................................... vi

1. INTRODUCTION ............................................................................................................ 1
   1.1. THESIS AIM ........................................................................................................... 3
   1.2. THESIS ORGANIZATION ...................................................................................... 4

2. PROCESS MONITORING ................................................................................................. 6
   2.1. FAULT DETECTION AND DIAGNOSIS .................................................................. 6
   2.2. MODEL-BASED PROCESS MONITORING METHODS ........................................... 17
   2.3. DATA-DRIVEN PROCESS MONITORING METHODS ............................................ 21
   2.4. CONCLUSIONS ...................................................................................................... 24

3. STATISTICAL PROCESS MONITORING METHODS ....................................................... 26
   3.1. MULTIVARIATE STATISTICS ................................................................................. 27
       3.1.1. DATA PRETREATMENT ................................................................................... 28
       3.1.2. UNIVARIATE STATISTICAL MONITORING .................................................. 30
       3.1.3. $T^2$ STATISTIC ............................................................................................. 32
   3.2. PATTERN CLASSIFICATION ................................................................................... 35
   3.3. PRINCIPAL COMPONENT ANALYSIS (PCA) ......................................................... 39
       3.3.1. PRINCIPAL COMPONENT ANALYSIS (PCA) FAULT DETECTION .............. 41
       3.3.2. PRINCIPAL COMPONENT ANALYSIS (PCA) FAULT IDENTIFICATION ...... 48
       3.3.3. PRINCIPAL COMPONENT ANALYSIS (PCA) FAULT DIAGNOSIS .......... 51
   3.4. FISHER DISCRIMINANT ANALYSIS (FDA) ......................................................... 54
       3.4.1. FISHER DISCRIMINANT ANALYSIS (FDA) FAULT DETECTION AND DIAGNOSIS ........................................................................................................ 58
   3.5. SELF-ORGANIZING MAPS (SOM) ......................................................................... 59
       3.5.1. SELF-ORGANIZING MAPS (SOM) FAULT DETECTION AND DIAGNOSIS .... 64
   3.6. CONCLUSIONS ...................................................................................................... 68

4. MULTI-AGENT BASED SYSTEMS .................................................................................. 69
4.1. JADE (Java Agent DEvelopment framework) ................................................................. 74
4.2. MULTI-AGENT SYSTEM ARCHITECTURE ................................................................. 80
  4.2.1. OPERATOR AGENT .............................................................................................. 84
  4.2.2. DATA PREPROCESSING AGENT .......................................................................... 86
  4.2.3. FAULT DETECTION AGENT ................................................................................ 86
  4.2.4. FAULT DIAGNOSIS AGENT ............................................................................... 91
4.3. CONCLUSIONS .......................................................................................................... 92

5. RESULTS AND DISCUSSION ......................................................................................... 93
  5.1. TENNESSEE EASTMAN PROCESS APPLICATION ............................................... 93
  5.2. RESULTS .................................................................................................................. 98
  5.3. CONCLUSIONS ....................................................................................................... 113

6. CONCLUSIONS & RECOMMENDATIONS .................................................................... 114

REFERENCES .................................................................................................................. 117

APPENDIX: AGENT JAVA CODES ................................................................................... 122
  A.1. OPERATOR AGENT ............................................................................................... 122
  A.2. DATA PREPROCESSING AGENT ........................................................................... 126
  A.3. FAULT DETECTION AGENT .................................................................................. 132
  A.4. FAULT DIAGNOSIS AGENT .................................................................................. 137
  A.5. OPERATOR AGENT USER INTERFACE ............................................................... 140
  A.6. MATLAB CONNECTION CLASS .......................................................................... 147

THE VITA ........................................................................................................................... 153
ABSTRACT

New process monitoring and control strategies are developing every day together with process automation strategies to satisfy the needs of diverse industries. New automation systems are being developed with more capabilities for safety and reliability issues. Fault detection and diagnosis, and process monitoring and supervision are some of the new and promising growth areas in process control. With the help of the development of powerful computer systems, the extensive amount of process data from all over the plant can be put to use in an efficient manner by storing and manipulation. With this development, data-driven process monitoring approaches had the chance to emerge compared to model-based process monitoring approaches, where the quantitative model is known as a priori knowledge. Therefore, the objective of this research is to layout the basis for designing and implementing a multi-agent system for process monitoring and supervision. The agent-based programming approach adopted in our research provides a number of advantages, such as, flexibility, adaptation and ease of use. In its current status, the designed multi-agent system architecture has the three different functionalities ready for use for process monitoring and supervision. It allows: a) easy manipulation and preprocessing of plant data both for training and online application; b) detection of process faults; and c) diagnosis of the source of the fault. In addition, a number of alternative data driven techniques were implemented to perform monitoring and supervision tasks: Principal Component Analysis (PCA), Fisher Discriminant Analysis (FDA), and Self-Organizing Maps (SOM). The process system designed in this research project is generic in the sense that it can be used for multiple applications. The process monitoring system is successfully tested with Tennessee Eastman Process application. Fault detection rates and fault diagnosis rates are compared amongst PCA, FDA, and SOM for different faults using the proposed framework.
1. INTRODUCTION

New process monitoring and control strategies are developing every day together with process automation strategies to satisfy the needs of diverse industries. Highly competitive economic measures demand lower cost and more efficient processes in manufacturing industries. Together with that, increasing environmental concerns and the effort to reduce the shutdown times of fabrication plants are pushing optimization and control to their achievable limits. The production industry is moving towards better overall system performance, high product quality and economic operation with environmental constraints. Together with all these new challenges, technology emerges too and new paradigms are developed to overcome emerging problems. New automation systems are being developed with more capabilities for safety and reliability issues. Within the automatic control of technical systems, supervisory functions serve to indicate undesired or not permitted process states. On top of that, the systems take appropriate actions in order to maintain the operation and to avoid damage or accidents. To accomplish or aid in accomplishing all these tasks, large amounts of data are collected in many chemical processes. The data can be analyzed to determine whether or not a fault has occurred in the process, where a fault is generally defined as abnormal process behavior. This abnormal process behaviour can be associated with equipment failure, equipment wear, or some process disturbances. This task of determining whether a fault has occurred is called fault detection. In addition to that, fault diagnosis is the task of determining which fault has occurred. Fault detection and diagnosis, and process monitoring and supervision are some of the new and promising growth areas in process control.
In order to have detailed information about a process, the optimization, control and monitoring of processes always involve employing models. In mechanistic models, the structure is based on fundamentals and whose parameters are estimated from plant data. On the other hand, in data-driven or empirical models, the structure and parameters are all identified from plant data. The important concern about the type of model to be used depends on the appropriateness of the model for the system in terms of its structure and assumptions. For instance, the structure established in mechanistic models contains many assumptions, some of which may not be entirely justified. The main concern about assumptions in a model depends on the structure of the disturbances in the system. This structure is rarely available from theory, and there exists many information rich variables cannot be incorporated into the mechanistic model, simply because the modeler cannot blend this information into the model. Thus, the seemingly unusable information is omitted and left out of the model.

Due to many different notions such as increased automation, faster sampling rates and advances in computing power, large amount of process data is available online and continuously stored. For many supervisory tasks such as process monitoring, diagnosis of process malfunctions, detection of mode change this stored data can be used in order to make use of the knowledge embedded in the data. However, in the majority of production plants all around the world, such supervisory tasks are left to the operator, in spite of having the measured data readily available at hand. This overwhelming task for the plant operators and engineers result in overload, and thus leading to erroneous decisions in some cases. Accordingly, it should be obvious that there is a need for approaches that can automatically capture and interpret knowledge. By doing this, the real-time decision-making burden on the operator can be substantially reduced.
In the first place, model-based fault detection and diagnosis schemes are relatively well established; however it is still very difficult to create the necessary nonlinear mathematical models for large scale processes. In addition to this, regular sensor measurements and records from historical data are kept in such industrial processes. With the help of the development of powerful computer systems, this extensive amount of process data from all over the plant can be put to use in an efficient manner by storing and manipulation. With this development, data-driven process monitoring approaches had the chance to emerge compared to model-based process monitoring approaches, where the quantitative model is known as a priori knowledge. Process operators are already keen on using this available data to minimize plant downtime and optimize process automation. However, without the use of helpful software development to aid this task, the task becomes rather overwhelming. By designing a superior and robust fault detection and diagnosis scheme using model-free and non-parametric methods the overall safety and reliability requirements of most critical factors in the process design can be successfully achieved.

1.1. THESIS AIM

One great challenge in the field of data-driven process monitoring is to develop an overall framework for advanced process monitoring with capabilities to organize, filter and manipulate data with the aim to detect abnormal events using an artificially-intelligent environment. Integration of these operational tasks in chemical plants will significantly help the automation of the system and this will be a positive step towards keeping the plant operational in the case of fault scenarios. The goal of the overall framework design and implementation is the automation of the processes of detecting faults and diagnosing their causes where possible.
Agent-based programming is a recent programming approach that offers various advantages for the implementation of the overall framework for advanced process monitoring. The agent-based approach helps for the coordination mechanism for each possible interaction which would have to be designed by hand. By using the agent-based programming approach the decisions being made in the system are automated, consistent and the approach reduces the overall development time. Therefore, the objective of this research is to layout the basis for designing and implementing a multi-agent system for process monitoring and supervision. The designed multi-agent system architecture has the three different functionalities ready for use for fault detection and diagnosis: Principal Component Analysis (PCA), Fisher Discriminant Analysis (FDA), and Self-Organizing Maps (SOM). The process system designed in this research project is generic in the sense that it can be used for multiple applications. The process monitoring system is successfully tested with Tennessee Eastman Process application. Fault detection rates and fault diagnosis rates are compared amongst PCA, FDA, and SOM for different faults.

1.2. THESIS ORGANIZATION

The thesis is organized as follows. Chapter 2 presents a brief introduction to process monitoring, and fault detection and diagnosis as well as a discussion of process faults and overview of process monitoring strategies are discussed. An introduction to model-based process monitoring methods and an introduction to data-driven process monitoring methods together with providing an overview of the methods.
Chapter 3 starts with describing multivariate statistics and giving an introduction to pattern classification. This section also covers statistical process monitoring methods (components of the proposed framework) including Principal Component Analysis (PCA), Fisher Discriminant Analysis (FDA), and Self-Organizing Maps (SOM). The section goes into detail about how each method is implemented and used in fault detection and diagnosis procedures.

Chapter 4 introduces multi-agent based systems. JADE, the middleware used in this research project, is discussed in detail in this section; together with the detailed description of the multi-agent system architecture designed in this research project for process monitoring.

Chapter 5 present the results for the research project together with introducing the Tennessee Eastman Process application. Finally, in Chapter 6 an overall discussion of the main findings and the conclusions for the thesis are given.
2. PROCESS MONITORING

2.1. FAULT DETECTION AND DIAGNOSIS

More complicated technical processes require advanced automation techniques to ensure safety and reliability issues. As automation gets more advanced the reliance on human operators to deal with abnormal events decrease considerably. Drawbacks of the reliance on human operators are thoroughly discussed in Venkatasubramanian et al. (Venkatasubramanian, Rengaswamy, Yin, & Kavuri, A Review of Process Fault Detection and Diagnosis Part I: Quantitative Model-Based Methods, 2003). The points can be summarized as below:

- Modern process plants are overwhelming to control by the help of human operators regarding their size and complexity; since there may be as many as 1500 process variables observed every few seconds, which leads to information overload.
- Time constraints for fault detection systems are of great importance in modern process plants for economic and safety issues. The correct decision making procedure for human operators lag fairly behind compared to automated systems.
- Process measurements of variables are insufficient, incomplete and/or unreliable more frequently than expected due to a variety of causes such as sensor biases or failures.

Considering all the disadvantages that human operators possess, it should be no surprise that human operators cause serious problems by making wrong decisions. Venkatasubramanian et al. claim that about 70% of the industrial accidents are caused by human errors, as revealed by industrial statistics. These abnormal events have significant impacts related to economy, safety
and environment. The estimates show that the petrochemical industry alone in the US is subjected to approximately 20 billion dollars in annual losses just because of poor abnormal event management. (Nimmo, 1995) Laser states that similar accidents costs British economy for a revenue loss up to 27 billion dollars every year (Laser, 2000). Together with that, another significant impact of abnormal events is accidents such as Union Carbide’s Bhopal, India accident and Occidental Petroleum’s Piper Alpha accident. (Lees, 1996)

Himmelblau defines a fault associated with a process as a departure from an acceptable range of an observed variable or a calculated parameter (Himmelblau, 1978). Similarly, a fault is defined as an unpermitted deviation of at least one characteristic property or variable of the system in Chiang (Chiang, Russell, & Braatz, 2001). Faults can be categorized according to their time dependencies as can be seen from Figure 2.1 (Isermann, 2005). Abrupt faults are stepwise faults, incipient faults are drift-like faults and intermittent are occasional faults.

![Figure 2.1](image)

**Figure 2.1:** Time dependency of faults: (a) abrupt; (b) incipient; (c) intermittent (Isermann, 2005).
Faults can be further classified with regard to the process models (Isermann, 2005). As can be seen from Figure 2.2, additive faults influence a variable $Y$ by an addition of the fault: $f$. Multiplicative faults influence a variable $Y$ by the product of another variable $U$ with the introduction of the fault: $f$. Additive faults appear as offsets of sensors, whereas multiplicative faults are parameter changes within a process. In addition, any kind of fault we come across in an industrial facility can be generally grouped in three categories according to their causes described by Venkatasubramanian et al.: gross parameter changes in a model, structural changes and malfunctioning sensors and actuators. (Venkatasubramanian, Rengaswamy, Yin, & Kavuri, A Review of Process Fault Detection and Diagnosis Part I: Quantitative Model-Based Methods, 2003)

![Basic models of faults](image)

Figure 2.2: Basic models of faults: (a) additive fault; (b) multiplicative fault (Isermann, 2005).

Gross parameter changes in a model are caused by the uncaptured dynamics that are occurring below the selected level of detail of the model. These detailed dynamics, which are highly nonlinear and cumbersome to model are typically lumped as parameters and these include interactions across the system boundary. Parameter failures arise when there is a disturbance entering the process from the environment through independent variables. An example of such a malfunction is the change in the heat transfer coefficient due to fouling in a heat exchanger.
As the name suggests, structural changes refer to changes in the process itself and occur due to hard failures in equipment. Structural malfunctions result in a change in the dynamic behavior and thus the information flow amongst various variables. In order to handle such a failure in a model-based approach, the system would require the removal of the appropriate model equations and restructuring the remaining equations in order to describe the current situation of the process. An example of a structural failure would be failure of a stuck valve or a broken or leaking pipe.

Malfunctioning sensors and actuators usually lead to the introduction of gross errors in the data. These failures could be due to a fixed failure, a constant bias in either positive or negative direction or an out-of range failure. A failure in one of the sensors and actuators could cause the plant state variables to deviate beyond acceptable limits since some of the instruments provide feedback signals which are essential for the control of the plant. For this reason, the failure must be detected promptly and corrective actions are accomplished in time. It is the purpose of fault detection and diagnosis system to detect any instrument fault as quickly as possible in order to prevent a possible serious degradation in the performance of the control system.

In addition to these faults; unstructured uncertainties, process noise and measurement noise exist in historical data, which are outside the scope of fault detection and diagnosis. Unstructured uncertainties are mainly faults that are not modelled as a priori knowledge. Process noise refers to the mismatch between the actual process and the predictions from model equations. Measurement noise refers to high frequency additive component in the sensor measurements. These abnormalities in the data should be filtered out in data preprocessing phase before starting fault detection and diagnosis.
The faults in the process need to be detected, diagnosed and then removed to ensure that the process operations satisfy the required performance specifications. The main goal of process monitoring is to ensure the success of the planned operations by recognizing anomalies in the behavior of the system (Chiang, Russell, & Braatz, 2001). The process monitoring system provides essential information about the status of the process and assists the plant operator and maintenance personnel to make appropriate remedial actions to remove the abnormal behavior from the process. Katipamula & Brambley defines the primary objective of a fault detection and diagnosis (FDD) system as early detection of faults and diagnosis of their causes (Katipamula & Brambley, 2005). This enables the correction of the faults before additional damage to the system or loss of service occurs. Fault detection is mainly accomplished by continuously monitoring the operations of a system, using FDD to detect and diagnose abnormal conditions and the faults associated with them, then evaluating the significance of the detected faults, and deciding how to respond. Successfully implemented process monitoring systems ensures that downtime is minimized, safety of plant operations is improved and manufacturing costs are reduced.

Chiang et al. defines the four procedures associated with monitoring as: fault detection, fault identification, fault diagnosis and process recovery; adopting the terminology given by Raich and Cinar (Chiang, Russell, & Braatz, 2001; Raich & Cinar, 1996). Fault detection is defined as determining whether a fault has occurred. The vital point of fault detection is early detection, which may provide invaluable warning on emerging problems, even before the problems start affecting the system directly. After the fault is detected, appropriate actions can be taken to avoid serious process upsets. Fault identification is defined as identifying the observation variables most relevant that can be used for diagnosing the fault. To eliminate the effect of the fault in a more efficient
manner, fault identification focuses the plant operator’s or engineer’s attention on the subsystems most applicable for the diagnosis of the fault. Fault diagnosis is defined as determining which fault occurred by determining the cause of the observed out-of-control status. In a more specific definition, Isermann defines fault diagnosis as determining the type, location, magnitude, and time of the fault (Isermann, Model Based fault Detection and Diagnosis Methods, 1995). Fault diagnosis procedure is an essential step for taking action against the fault and eliminating it decisively. The final step in the process monitoring loop is process recovery and it is defined as removing the effect of the fault. Process recovery can also be called intervention and is a necessary step to close the monitoring loop as can be seen from Figure 2.3. Whenever a fault is detected, fault identification, fault diagnosis, and process recovery procedures are implemented respectively. If the system does not come across a fault, only the fault detection procedure is repeated until a new fault is detected.

Figure 2.3: A schema of the process monitoring loop. (Chiang, Russell, & Braatz, 2001)

In a complete and successful process monitoring scheme, the procedures defined above are implemented as shown in Figure 2.3. On the other hand, in practice it may not always be necessary
to use the scheme as it is. For example, a fault diagnosis phase may be implemented without identifying the variables affected by the fault immediately. In addition to this, in an ideal process monitoring system the procedures should be automated as much as possible; however it is not necessary to automate all four procedures. Generally, assistance of the plant operators and engineers to diagnose the fault is the practical goal of process monitoring; in order to recover normal operation. For that reason, plant operators and engineers are incorporated into the process monitoring loop efficiently, rather than trying to automate the entire process monitoring scheme.

In-control operations need to be recovered generally after a fault occurs. Even if the fault diagnosis procedure is implemented successfully, the optimal approach to take in the process recovery phase may not be obvious. Process recovery can be done by reconfiguring the process, repairing the process, or retuning the controllers. Extensive amount of information about the topic can be seen in literature for retuning controllers and sensor reconstruction (Harris, 1989; Rhinehart, 1995; Dunia, Qin, F., & J., 1996). Although process recovery is a crucial and essential part of the process monitoring loop, it will not be further discussed in this thesis.

As mentioned before, the main idea lying behind process monitoring is to identify some measures to represent the state or behavior of the process. These measures can be developed using statistical theory, pattern classification theory, information theory, and systems theory (Chiang, Russell, & Braatz, 2001). The on-line data collected from the process can be converted into a few meaningful measures to assist the plant operators in determining the status of the operations and also diagnosing the faults where necessary. For fault detection purposes, certain limits can be defined for some of measures so that a fault can be detected whenever one of the evaluated measures goes
outside of the defined limits. By using this simple approach, the identified measures have the ability to define in-control behavior of the system as well as the out-of-control status. In fault identification, measures which accurately characterize the behavior of each observation variable can be developed. By doing so, the measure of one variable can be compared against the measure for other variables to determine the variable most effected by the corresponding fault. Fault diagnosis can be achieved by developing and comparing measures which accurately represent the different faults of a certain process.

An ideal process monitoring system must be sensitive and robust against all possible faults. This can be achieved by developing multiple measures, where each measure can be efficiently used to detect and diagnose a certain type of fault. This approach is handy and necessary since faults are manifested in several ways and it is impractical trying to detect and diagnose all these faults using only a few process measures. Using multiple process measures with the ability to detect and diagnose particular processes overcomes this problem and serves the goal of creating a sensitive and robust system against all possible faults.

In the general framework, fault detection and diagnosis requires two important components: a priori domain knowledge and a search strategy. The basic a priori knowledge required for process monitoring is a set of failures and the relationship between the observations and the failures (Venkatasubramanian, Rengaswamy, Yin, & Kavuri, A Review of Process Fault Detection and Diagnosis Part II: Qualitative Models and Search Strategies, 2003). The observations can also be referred as the symptoms of each particular fault. The a priori knowledge for a process monitoring system may be explicit as in a table look-up, or it may be inferred from some source of domain
knowledge. Firstly, the first-principles knowledge can be used to develop a fundamental understanding of the process to develop the a priori domain knowledge. The developed a priori domain knowledge is referred as deep, casual or model-based knowledge (Milne, 1987). The process monitoring methods which use model-based knowledge are simply called model-based process monitoring methods. Secondly, past experience with the process can be the basis for developing the a priori domain knowledge. The developed a priori domain knowledge is referred as shallow, compiled, evidential, process history-based knowledge (Venkatasubramanian, Rengaswamy, Yin, & Kavuri, A Review of Process Fault Detection and Diagnosis Part II: Qualitative Models and Search Strategies, 2003). The process monitoring methods which use process history-based knowledge are simply called data-driven process monitoring methods.

Data-driven process monitoring methods are directly derived from process data. Modern industrial systems are getting fairly larger every day and even a single unit in an industrial plant can be considered as a large-scale system. Modern processes require heavy instrumentation as large-scale systems and produce an exceptionally large amount of data continuously. Although this data is readily available for plant operators and engineers, it is beyond the capabilities of a human being to effectively assess process operations simply from observing the data. The strength of data-driven process monitoring techniques lies within their ability to transform the high-dimensional data produced by the large-scale industrial systems into a lower-dimensional data. In the process of lowering the dimension of the data, the important information hidden in the data is captured. By using the lower-dimension data created, with the help of some meaningful statistics a successful process monitoring system can be established for a large-scale system. The main drawback of data-
driven process monitoring techniques is that their strength is highly dependent on the quality and quantity of the available process data.

On the other hand, model-based process monitoring methods uses mathematical models often constructed from first principles. The model-based approach is applicable to rather simple and information-rich systems, where satisfactory models and enough sensors are available for the process. Most model-based methods are based on parameter estimation, observer-based design, and/or parity relations (Chiang, Russell, & Braatz, 2001). Most applications of model-based process monitoring systems have been to systems with a relatively small number of inputs, outputs, and states. It is challenging to apply this approach to systems containing a large number of inputs, outputs, and states, which is the case for most modern industrial plants and operations. This is mainly because of the fact that model-based systems require detailed models in order to be effective. Detailed models for large-scale systems are very hard and expensive to obtain given all the cross couplings and nonlinearities associated with a multivariable system (Chiang, Russell, & Braatz, 2001). The main advantage of model-based process monitoring systems is the ability to incorporate physical understanding of the process into the process monitoring scheme. It is model-based process monitoring systems’ requirement to have detailed analytical models readily available to outperform data-driven process monitoring methods.

In the general sense, a process monitoring scheme can be seen as a series of transformations or mappings on process measurements, which aids the diagnostic decision making procedure. The various transformations that process data go through during the fault detection and diagnosis process can be seen from Figure 2.4. The measurement space in Figure 2.4 is a space of
measurements with no a priori problem knowledge relating these measurements (Venkatasubramanian, Rengaswamy, Yin, & Kavuri, A Review of Process Fault Detection and Diagnosis Part I: Quantitative Model-Based Methods, 2003). These measurements are the input to the process monitoring system. The feature space in Figure 2.4 is a space of feature points obtained as a function of the measurements by utilizing a priori problem knowledge. In the feature space, in order to extract useful features about the process behavior with the goal of helping process monitoring; the measurements are analyzed and combined with the aid of a priori process knowledge. The mapping from the feature space to decision space is generally established by meeting an objective function; such as minimizing the misclassification rate. This transformation from the feature space to decision space is generally achieved by using a discriminant function. In some cases, the transformation can also be achieved using simple threshold functions. The decision space is a space of points having the dimension of the number of decision variables, which are obtained by suitable transformations of the feature space. The class space is a set of integers having the dimension of the number of failure classes. The failure class index categorically shows the belongingness of specific measurement patterns to a specific class; including the normal region. As can be seen from Figure 2.4, the class space is the final step in the process monitoring scheme. As a result, it is the final interpretation provided by the process monitoring system to be delivered to the plant operator or the engineer. The transformations from the decision space to the class space can be performed using threshold functions, template matching or symbolic reasoning (Venkatasubramanian, Rengaswamy, Yin, & Kavuri, A Review of Process Fault Detection and Diagnosis Part I: Quantitative Model-Based Methods, 2003).
2.2. MODEL-BASED PROCESS MONITORING METHODS

Model-based process monitoring techniques generate features using detailed mathematical models based on the measured input and output for the system. For model-based approaches, residuals, parameter estimates, and state estimates are commonly used features. Either directly or after some transformation, faults are detected and diagnosed by comparing the observed features with the features associated with normal operating conditions (Chiang, Russell, & Braatz, 2001).

Model-based fault detection and diagnosis techniques emerged in 1970s and developed extensively since then (Ding, 2008). All model-based fault detection and diagnosis techniques have implemented algorithms which explicitly use a process model to use with on-line process data, which is collected and recorded during the system operation. Detection of faults in the processes, actuators and sensors is achieved by using the dependencies between different measurable signals. Mathematical process models are developed in order to express these dependencies (Isermann, 2005). The so called strength of a model is simply based on its redundancy to the actual process. In other words, how well the model can simulate the behavior of the actual system designates the redundancy of the model. Redundancy can be constructed both by using hardware and software components (Ding, 2008).
Hardware redundancy requires redundant sensors to reconstruct the crucial components of a system using the identical hardware. Hardware redundancy based process control methods are well utilized in the control of such safety-critical systems as aircraft space vehicles and nuclear power plants (Venkatasubramanian, Rengaswamy, Yin, & Kavuri, A Review of Process Fault Detection and Diagnosis Part I: Quantitative Model-Based Methods, 2003). However, it is obvious that the applicability of hardware redundancy is limited due to the extra cost and additional space required in reconstruction. Analytical redundancy addresses these problems and is achieved from the functional dependence among the process variables. Analytical redundancy is usually provided by a set of algebraic or temporal relationships among the states, inputs and the outputs of the system. After analytical redundancy is readily available, fault detection can simply be achieved by calculating the residual for each data point. The residual is defined in Equation 2.1 as the difference between the measured process variables; \( y \), and their redundancy: \( \hat{y} \);

\[
\text{residual} = y - \hat{y} \quad \text{Eq. 2.1}
\]

Fault detection using analytical redundancy consists of residual generation and then evaluation. By checking the actual system behavior against the system model for consistency, fault detection can be achieved. Calculated residuals express inconsistencies and thus can be used for detection and isolation purposes as can be seen from Figure 2.5. The residuals should be close to zero when no fault occurs; but should show greater values when there is a change in the system. An explicit mathematical model of the system is required for the generation of the diagnostic residuals. For the creation of the model, either an analytical model derived using the first principles or an empirically obtained black-box model can be used.
Analytical models derived using the first principles are obtained based on a physical understanding of the process. In the development of model equations in chemical engineering processes mass, energy and momentum balances are used together with constitutive relationships such as equations of state. Models developed using first principles are generally very complex and often nonlinear, which makes the design of fault detection and diagnosis systems more challenging. With the improvement in more advanced computer systems and an improved understanding of nonlinear controller design analytical models for process monitoring are improving.

The problem that fault detection and diagnosis systems come across is to identify the state of a process based on its behavior. In the physical environment, the behavior of a process is monitored through its sensor outputs and actuator inputs as observed variables. Whenever a fault occurs, the relationship among these observed variables change and as a result a nonzero residual is generated.
Most process monitoring methods used nowadays have black-box plant models such as input-output or state-space models and assume linearity of the plant (Venkatasubramanian, Rengaswamy, Yin, & Kavuri, A Review of Process Fault Detection and Diagnosis Part I: Quantitative Model-Based Methods, 2003). The assumption of linearity gives birth to accuracy issues with the model. Unlike the black-box models, models created using first principles bear certain physical meanings. These physical meanings have great conveniences in fault detection and diagnosis, and controller design. Since most model-based approaches assume system linearity, their application to a non-linear system requires a model linearization around the operating point. Several factors such as system complexity, high dimensionality, process nonlinearity, and lack of useful data make it very difficult and often impractical to develop and accurate mathematical model for large-scale industrial systems. This fact limits the usefulness of model-based process monitoring systems in real industrial processes. Another problem related to model-based approaches is the simplistic approximation of the disturbances that include modelling errors. In most cases, model-based approaches only include additive modeling uncertainties for disturbances. However, in practice, multiplicative modeling uncertainties are caused by parameter drifts. This point is a serious limitation of all the model-based process monitoring schemes that have been developed so far (Venkatasubramanian, Rengaswamy, Yin, & Kavuri, A Review of Process Fault Detection and Diagnosis Part I: Quantitative Model-Based Methods, 2003). Based on the drawbacks of model-based methods stated above, it is obvious that data-driven process monitoring methods require more attention.
2.3. DATA-DRIVEN PROCESS MONITORING METHODS

As discussed before, model-based process monitoring methods require a priori knowledge about the system, whereas for data-driven process monitoring methods only the availability of large amount of historical process data is needed. The historical data provided by the process can be transformed and presented to the process monitoring system as a priori knowledge in different ways. This transformation and presentation is known as feature extraction (Venkatasubramanian, Rengaswamy, Yin, & Kavuri, A Review of Process Fault Detection and Diagnosis Part III: Process History Based Methods, 2003). The quantitative extraction techniques can be broadly group into two subcategories: statistical and non-statistical methods. Self-Organizing Maps (SOM), which are a form of neural networks, are an important example of non-statistical data classifiers. Examples for statistical feature extraction methods can be Principal Component Analysis (PCA) and Fisher Discriminant Analysis (FDA).

Limit sensing and discrepancy detection are amongst traditional process monitoring methods. Limit sensing is an easy to implement and understand method and it arises an alarm when observations cross predefined thresholds. However, since limit sensing ignores the interactions between the process variables for various sensors, it lacks sensitivity to some process upsets (Chiang, Russell, & Braatz, 2001). The way that discrepancy detection raises an alarm is by comparing simulated to actual observed values. Discrepancy detection highly depends on model accuracy. As discussed in model-based process monitoring methods, model inaccuracies are unavoidable in practice. Discrepancy detection can lack robustness since it is very difficult to distinguish genuine faults from errors in the model. Robust discrepancy detection statistics and
techniques have been studied; but it can be concluded that effective statistics are difficult to obtain especially for large-scale systems (Chiang, Russell, & Braatz, 2001).

Limit sensing is similar to univariate statistical techniques in the sense that it determines thresholds for each observation variable without using any information from other variables. Both methods ignore spatial correlations; which are the correlations among the observation variables, and serial correlations; which are the correlations among measurements of the same variable taken at different times. Limit sensing lacks sensitivity to many faults; because it does not take the spatial correlations. Limit sensing can also lack robustness; because it ignores serial correlations (Chiang, Russell, & Braatz, 2001).

Multivariate statistical techniques in process monitoring statistics emerged from the need to handle spatial correlations. Principal component analysis (PCA) is the most basic and widely used data-driven process monitoring technique for industrial systems. PCA is a dimensionality reduction technique which has been studied extensively over the last two decades. PCA accounts for correlations among variables and is an optimal dimensionality reduction technique in terms of capturing the variance of the data. Using multivariate statistics, the lower-dimensional representations of the data produced by PCA can improve the proficiency of fault detection and diagnosis. Principal component analysis can be useful for identifying either the variables responsible for the fault or the variables most affected by the fault. For applications where the desired amount of important information in the data can be captured in only two or three dimensions; the dominant process variability can be visualized in a single plot. Although this is not the case for the majority of process monitoring applications, other plots such as $T^2$ and $Q$ charts
can be used for visualization purposes. These charts are irrespective of how many dimensions required in the lower-dimension space and look similar to univariate charts but are based on multivariate statistics. These control charts can aid plant operators and engineers to observe and understand significant trends in process data. PCA will be discussed in detail in Section 3.3.

Another dimensionality reduction technique commonly used in process monitoring practice is Fisher Discriminant Analysis (FDA). FDA aids pattern classification by determining the portion of the observation space that is most effective in discriminating amongst several classes. For fault diagnosis purposes, discriminant analysis is applied to this portion of the observation space. An important feature of FDA is that it is applied to the data in all the classes simultaneously. When the discriminant function is evaluated for each class, all fault class information is utilized. For this theoretical superiority of FDA over PCA, better fault diagnosis performance is expected from FDA. Although this is case for many applications and their specific faults; there are still many cases that PCA over performs FDA in fault diagnosis. FDA will be discussed in detail in Section 3.4.

Self-Organizing Maps (SOM) are pattern recognition techniques that use the association between data patterns and fault classes without an explicit modelling of internal process states or structures. SOM is related to the data-driven process monitoring techniques, such as PCA and FDA, in terms of modeling the relationship between data patterns and fault classes. PCA and FDA are dimensionality reduction techniques based on rigorous multivariate statistics. Furthermore, Self-Organizing Maps are black box methods which learn the pattern based entirely from the training sessions for the specific map. SOM will be discussed in detail in Section 3.5. The process
monitoring measures for Principal Component Analysis (PCA), Fisher Discriminant Analysis (FDA), and Self-Organizing Maps (SOM) can be calculated based entirely on data. For the cases where a detailed first-principles or any other mathematical model is available, model-based process monitoring techniques can provide more effective fault detection and diagnosis than data-driven process monitoring techniques.

Venkatasubramanian et al. states that no single method used in data-driven fault detection and diagnosis techniques perform absolutely superior to other methods (Venkatasubramanian, Rengaswamy, Yin, & Kavuri, A Review of Process Fault Detection and Diagnosis Part III: Process History Based Methods, 2003). Thus it should be concluded that some of these methods can complement one another to give birth to better process monitoring systems. Integrating these different methods such as Principal Component Analysis (PCA), Fisher Discriminant Analysis (FDA), and Self-Organizing Maps (SOM) into a single overall process monitoring system by designing a multi-agent system architecture in order to develop a more capable process monitoring system became the motivation for this thesis. A multi-agent system architecture is designed in this research project in order to accomplish this goal in an efficient and user-friendly manner. Furthermore, integrating these complementary features in a way to develop hybrid methods can overcome the limitation of individual solution strategies of each method.

2.4. CONCLUSIONS

This Chapter of the thesis started with presenting a brief introduction to process monitoring, and fault detection and diagnosis as well as a discussion of process faults and overview of process
monitoring strategies. In addition, an introduction to model-based process monitoring methods and an introduction to data-driven process monitoring methods are provided together with an overview of the methods. To sum up, both data model-based and data-driven process monitoring techniques have their advantages and disadvantages for specific applications and it should be concluded that no single approach is best for all applications. Chiang et al. suggests that usually the best process monitoring scheme employs multiple statistics or methods for fault detection, identification, and diagnosis (Chiang, Russell, & Braatz, 2001). Incorporating several techniques for process monitoring is beneficial in many applications. Thus, this was an important motivation for designing an overall process monitoring framework that has the options for using PCA, FDA and SOM for fault detection and diagnosis. The details of the designed architecture will be explained in Section 4.2.
3. STATISTICAL PROCESS MONITORING METHODS

As discussed above, model-based process monitoring methods assume the quantitative model as a known priori. On the other hand, multivariate statistical process monitoring methods such as Principal Component Analysis (PCA), Fisher Discriminant Analysis (FDA), and Self-Organizing Maps (SOM) are only dependent on large amounts of historical data for detection and diagnosis of abnormal process behaviors. Statistical process monitoring methods are especially popular in large-scale industrial applications; because they are easy to design and simple to operate (Chiang, Russell, & Braatz, 2001). The common goal in PCA, FDA, and SOM is to present the desired process behavior to the plant operator or engineer in an efficient manner; without directly representing all the raw data produced by the system. This is done with the help of different statistical tools to carry out feature extraction, discriminant calculation and maximum selection procedures explained in Section 3.2. It should be obvious that multivariate methods are far more superior against univariate methods, which only monitor the magnitude and variation of single variable. Use of multivariate statistical process monitoring methods increase the reliability and robustness of the process monitoring system against plant-wide disturbances.

The remainder of this Section is organized in five parts. Section 3.1 describes multivariate statistics including an introduction about data pretreatment. The developmental procedure from univariate statistical monitoring to $T^2$ statistic is also presented in this Section. Section 3.2 describes pattern classification and its use in process monitoring strategies. Section 3.3 presents Principal Component Analysis (PCA) in detail, together with the use of PCA in fault detection and diagnosis. Section 3.4 presents Fisher Discriminant Analysis (FDA) in detail, together with how FDA is used
in fault detection and diagnosis. Finally, Section 3.5 presents Self-Organizing Maps (SOM) in detail, together with the use of SOM in fault detection and diagnosis.

3.1. MULTIVARIATE STATISTICS

Data-driven process monitoring methods can be effective when characterization of the process data variations are done successfully. Ogunnaike and Ray states that there are two types of variations for process data: common cause and special cause (Ogunnaike & Ray, 1994). The common cause variations are variations entirely due to random noise, such as noise associated with sensor readings. Special cause variations account for all the variations which are not attributed to common cause variations. Most special cause variations may be removed by standard process control strategies; however common cause variations generally cannot be removed just using standard process control strategies. The main reason is that common cause variations are inherent to process data and since variations in the process data are inevitable, multivariate statistics are used in most process monitoring schemes (Chiang, Russell, & Braatz, 2001).

The main assumption for using statistical theory for process monitoring is the realization is normal operation. That is, the characteristics of the data variations are relative unchanged unless a fault occurs in the system. This fault free data set is called the normal operating region. This assumptions is already backed up with various definitions of faults in industrial systems provided in the Section 2.1; namely fault being an abnormal process condition. This assumption leads to the realization that the properties of data variations, such as the mean and the variance of the data, are repeatable for same operating conditions, although the actual values of data may not be very predictable.
(Chiang, Russell, & Braatz, 2001). This feature of repeatability of the statistical properties is crucial for statistical process monitoring; since it allows thresholds for certain measures. By doing this, out-of-control status can be defined effectively and more importantly automatically. Making use of this feature is an important step in automating a data-driven process monitoring scheme.

3.1.1. DATA PRETREATMENT

Data pretreatment is often necessary for the data in the training set in order to extract the information in the data relevant to process monitoring efficiently (Chiang, Russell, & Braatz, 2001). The training set consists of the available off-line data for analysis before the process monitoring scheme is implemented on-line. This data is used to develop measures representing the in-control operations (normal operating region) together with different faults. This off-line data set is generally referred as unclean before data pretreatment. The pretreatment procedure to generate clean data generally consists of three tasks: removing variables, auto-scaling and removing outliers.

Removing variable refers to the variables that have no information relevant to process monitoring. An example for such a variable can be one which is known to exhibit extremely large measurement errors; as in the case of a poorly calibrated sensor. These type of variables should be removed before further analysis, since they can effect process monitoring measures negatively. Another case can be that some of the variables may be physically separate from the portion of the process which is being monitored. These type of in appropriate variables should be removed before further analysis in order to improve the proficiency of the process monitoring method.
Especially for the methods that are based on dimensionality reduction, like Principal Component Analysis (PCA) and Fisher Discriminant Analysis (FDA), process data often need to be scaled in order to avoid particular variables dominating certain process monitoring measures. In a simple example, the variation in temperatures values from 400 °F to 410 °F would be dominant in certain process monitoring measures compared to the variation in chemical concentration values from 0.1 to 0.2. To prevent this domination, all variables should be normalized using specific mean and variance values for each variable before further processing. Auto-scaling or normalization standardizes the process variables in a way that ensures each variable is given equal weight before the application of the certain fault detection and diagnosis method (Chiang, Russell, & Braatz, 2001). Auto scaling is the process of converting all the variables to the same scale so that each variable is given equal weight before its use. For normalization of data, each variable is first subtracted from its sample mean in order to capture the variation of the data from the mean. Then the mean-centered value obtained for each variable is divided by the specific standard deviation in order to scale each variable to unit variance to ensure that the process variables with high variances do not dominate as can be seen from Equation 3.1. The mean and standard deviation used in the data pretreatment process in obtained from historical training data for the process.

\[
\chi_{normal_i} = \frac{x_{raw_i} - \bar{x}}{\sigma_x}
\]

Eq. 3.1

Furthermore, outliers are defined as isolated measurement values that are erroneous by Chiang et al. (Chiang, Russell, & Braatz, 2001). Since these values are generally further away from the mean value, they may significantly influence the estimation of statistical parameters and other process
monitoring measures. Removing outliers from the training data set can significantly improve the estimation of statistical parameters and thus is an essential step in data pretreatment. Far distant outlier are obvious and thus can be easily removed by visual inspection as the most simplistic case. This approach is not practical and thus not suitable for large numbers of variables and large number of data points. For more detailed and rigorous outlier removal methods based on statistical thresholds, such as $T^2$ statistic can be employed. Detailed information about data preprocessing can be seen from Romagnoli and Sanchez (Romagnoli & Sanchez, 2000).

3.1.2. UNIVARIATE STATISTICAL MONITORING

Observation variables are process variables observed through a sensor reading. In the simplest process monitoring scheme, a univariate statistical approach to limit sensing can be used to determine the thresholds for each observation variable (Chiang, Russell, & Braatz, 2001). These determined thresholds define the boundary for in-control operations and any violation of these limits by the on-line data will alarm the system as a fault. A Shewhart chart is typically employed for this method, which is usually referred as limit sensing and limit value checking. An example Shewhart quality control chart can be seen in Figure 3.1. A process is referred to as in-control process if the measures are within the bounds of the upper control limit (UCL) and the lower control limit (LCL), otherwise the process is referred to as out-of-control.
Figure 3.1: Sample Shewhart quality control chart for Mooney viscosity data showing the desired target ($y_d$), upper control limit (UCL), and lower control limit (LCL) (Ogunnaike & Ray, 1994).

The most critical measures for a Shewhart chart is the upper control limit (UCL) and the lower control limit (LCL) for minimizing the rate of false alarms and the rate of missed detections. A false alarm, is an indication of a fault by the process monitoring system, when in actuality no fault has occurred. A missed detection is no indication of a fault by the process monitoring system, when in actuality a fault has already occurred. Inherently, for fault detection there is a trade-off between minimizing the false alarm rate and the missed detection rate. When the thresholds for UCL and LCL for an observation variable are too tight, the false alarm rate will be high and missed detection rate will be low. On the other hand, when the thresholds for UCL and LCL for an observation variable are too spread apart, a low false alarm rate and a high missed detection rate should be expected. The only ways to get around this inevitable trade-off is to either collect more
data or to reduce the normal process variability through installation of sensors with higher precision. When the certain threshold variables are available, statistical hypothesis theory can be applied to predict the false alarm and missed detection rates based on the statistics of the data in the training sets (Chiang, Russell, & Braatz, 2001). Using statistical hypothesis theory, the false alarm rate is equal to the type I error and missed detection rate for a particular fault is equal to the type II error.

### 3.1.3. $T^2$ STATISTIC

Recently, industrial systems are becoming more highly integrated and complex. As a result, faults occurring in modern processes present monitoring challenges which are not easily addressed by using simple univariate control charts. The main reason for this is, as discussed above, the thresholds determined for each observation variable in univariate charts do not consider the information contained in other variables. As a result, univariate methods ignore the correlation between variables and thus they do not accurately characterize the behavior of most modern industrial processes.

$T^2$ statistic is a very useful multivariate measure for data-driven process monitoring methods. Assuming the data in the training set to be consisting of $m$ observation variables and $n$ observations for each variable; a $X \in \mathbb{R}^{n \times m}$ matrix can be created by storing all the information available for the process (Chiang, Russell, & Braatz, 2001):
\[ X = \begin{bmatrix}
  x_{11} & x_{12} & \cdots & x_{1m} \\
  x_{21} & x_{22} & \cdots & x_{2m} \\
  \vdots & \vdots & \ddots & \vdots \\
  x_{n1} & x_{n2} & \cdots & x_{nm}
\end{bmatrix} \]

After the data is stacked into the \( X \) matrix, the sample covariance matrix of the training data set can be calculated as \( S \) using Equation 3.2:

\[
S = \frac{1}{n-1} X^T X
\]  
Eq. 3.2

Eigenvalue decomposition of the matrix \( S \) will yield:

\[
S = V \Lambda V^T
\]  
Eq. 3.3

where \( \Lambda \) is a diagonal matrix whose diagonal elements are the corresponding eigenvalues as in \( \Lambda_{ii} = \lambda_i \). \( V \) is an orthogonal \((V^T V = I)\) square matrix whose \( i^{th} \) column is the eigenvector \( v_i \) of \( S \). The projection \( y = V^T x \) of an observation vector \( x \in \mathbb{R}^m \) decouples the observation space into a set of uncorrelated variables corresponding to the elements of \( y \). The variance of the \( i^{th} \) element of \( y \) is equal to the \( i^{th} \) eigenvalue in the matrix \( \Lambda \). Assuming \( S \) in invertible, \( z \) can be defined as:

\[
z = \Lambda^{-1/2} V^T x
\]  
Eq. 3.4

By using the definition of \( x \), Hotelling’s \( T^2 \) statistic can be found by using Equation 3.5:

\[
T^2 = z^T z
\]  
Eq. 3.5
The $T^2$ statistic allows us to characterize the variability of the data in the entire $m$-dimensional observation space. For a given level of significance, by applying appropriate probability distributions appropriate scalar threshold values for the $T^2$ statistic can be determined automatically. Figure 3.2 shows the comparison of the in-control regions established using $T^2$ statistic and univariate statistics. As can be seen from Figure 3.2, the $T^2$ confidence region forms an ellipsoidal area compared to the rectangular area created by the univariate statistical region. The in-control region for operation is defined by the area remaining inside the ellipsoid in the case for $T^2$, as in $T < T^2_\alpha$. The leftover area outside the ellipsoid but inside the rectangle resembles the possible reported false alarms if only univariate statistical process control is used. Thus, this Figure 3.2 resembles the strength of multivariate statistical process control against univariate statistical process control in a simplistic manner.

![Figure 3.2: The comparison of the in-control regions in the observation space created using $T^2$ statistic and univariate statistics. (Chiang, Russell, & Braatz, 2001)](image)
In most cases, the actual covariance matrix for the in-control operation is not known and thus has to be estimated from the sample covariance matrix. When this is the case, faults can be detected for on-line data or any data collected outside the training set using the threshold given by Equation 3.6:

\[ T^2_\alpha = \frac{m (n - 1) (n + 1)}{n (n - m)} F_\alpha(m, n - m) \]  

where \( F_\alpha(m, n - m) \) is the upper 100\( \alpha \)% critical point of the \( F \)-distribution with \( m \) and \( n - m \) degrees of freedom (Chiang, Russell, & Braatz, 2001).

The upper control limits calculated from this equation assume that the data taken at one time instant is statistically independent to the data obtained at other time instances. For short sampling intervals, this can be a bad assumption. Nonetheless, this problem is overcome by having enough data in the training set to capture normal process variations. As the generalization for all data-driven process monitoring methods stated above, quality and quantity of the data set is crucial for monitoring proficiency. As long as sufficient and good quality historical data is available, mild deviations from normality or the statistical independence assumptions does not prevent \( T^2 \) statistic to be an effective tool in process monitoring.

3.2. PATTERN CLASSIFICATION

As discussed previously, modern industrial systems are heavily instrumented having massive amounts of data collected on-line and stored in computer databases. Together with the data
collected during in-control processes, there also exists many datasets constructed during out-of-control operations. Whenever the diagnosis decision for the data collected during the out-of-control operations is available, the data can be categorized into separate classes where each class pertains to a particular fault (Chiang, Russell, & Braatz, 2001). Cluster analysis is used for the cases where the data have not been previously diagnosed, for categorizing the data into separate classes. After the class information is established, whenever a fault is detected in the on-line process, the fault can be diagnosed by determining the fault region in which the new data points are located. The only requirement for successful fault diagnosis in this case is the fault detected being represented in the database. Figure 3.3 shows sample classes for an example process. As can be seen from the figure, the process data has been clustered into three distinct operation modes; namely classes ‘a’, ‘b’, and ‘c’, where each class represents a different operation status.

Figure 3.3: Certain type process statuses are shown as classes ‘a’, ‘b’, and ‘c’ for a sample process.
Pattern classification theory aids the assignment of data to one of the several categories or classes (Chiang, Russell, & Braatz, 2001). There are three main steps for a typical pattern classification system for the assignment of observation vector to one of several classes: feature extraction, discriminant analysis, and maximum selection. All these steps can be seen in Figure 3.4. Feature extraction step is implemented to increase the robustness of pattern classification system. This is done by reducing the dimensionality of the observation vector in a way that retains most of the information discriminating amongst different classes.

![Figure 3.4: A typical pattern classification system schema, where $f_i(x)$ are the feature extraction functions and $g_i(x)$ are the discriminant analysis functions (Chiang, Russell, & Braatz, 2001).](image)

Feature extraction is especially important when there is only a limited amount of quality process data is available. In the discriminant analysis step, the discriminant calculator computes the discriminant function value for each class; using the information in the reduced-dimensional space.
The discriminant function is a function quantifying the relationship between the observation vector and a particular class. Discriminant functions serve as indirect separating planes amongst classes by selecting the class with the maximum discriminant function value. A typical pattern classification schema including feature extraction, discriminant analysis, and maximum selection steps can be seen from Figure 3.4.

The pattern classification system assign an observation data point to the class \( i \), where the class \( i \) has the maximum discriminant function value as \( g_i(x) > g_j(x) \) (Chiang, Russell, & Braatz, 2001). \( g_j(x) \) is the discriminant function for class j given a data vector \( X \in \mathbb{R}^m \). For the case that the mean vector and covariance matrix are estimated, the discriminant function can be calculated using Equation 3.7:

\[
g_i(x) = -(x - \bar{x}_i)^T S_i^{-1} (x - \bar{x}_i) - \ln[\det(S_i)]
\]

where \( \bar{x}_i \) is the mean vector for class \( i \) and \( S_i \) is the sample covariance matrix for class \( i \). Since this discriminant function uses the entire data dimensionality for classification, it fall under multivariate statistical process control. This discriminant function will suffer from suboptimal classification if sufficient data are not available to accurately estimate the mean vector and covariance matrix for each class. In this case, in order to improve classification, dimensionality reduction can be used.

The misclassification rate is defined as the number of incorrect classifications divided by the total number of classifications whenever the pattern classification system is applied to some testing data
The objective of the pattern classification system is to minimize the misclassification rate. In minimizing the misclassification rate, the dimensionality reduction of the feature extraction step can play a key role. This is the case especially for datasets having a large dimension observation space and a small number of observations in each class. In the case where the mean and covariance of the classes are known exactly, the entire observation space should be maintained for discriminant analysis step. However in reality, this is seldom the case and inaccuracies exist in the statistical parameters of the classes. As a result, the amount of information obtained in some directions of the observation space may not outweigh the inaccuracies in the statistical parameters. This is the case especially for the variables that do not add much information in discriminating the data in the training set. Thus, elimination of these variables in the feature extraction step can decrease the misclassification rate when applied to the testing data (on-line data), which is independent of the training set. System identification theory also backs up this point of view by providing the fact that the accuracy of a model can be improved by decreasing the number of independent model parameters.

### 3.3. PRINCIPAL COMPONENT ANALYSIS (PCA)

As suggested before, dimensionality reduction techniques can greatly simplify and improve process monitoring applications by projecting the data into a lower-dimensional space that accurately characterizes the state of the process (Chiang, Russell, & Braatz, 2001). As mentioned before, Principal Component Analysis (PCA) is one such dimensionality reduction technique widely used both industry and academics. In order to capture the variability in the data, Principal
Component Analysis (PCA) produces a lower-dimensional representation in a way that preserves
the correlation structure between the process variables.

Principal Component Analysis has been applied to many different processes as a dimensionality
reduction tool for process monitoring (Piovoso, Kosanovic, & Pearson, 1992; Kresta, Marlin, &
F., 1991; Wise & Gallagher, 1996; Dunia & Qin, 1998; Raich & Cinar, 1996). As mentioned
before, most modern process metric that account for data variability cannot be visualized using a
single plot; in two or three dimensions. Principal Component Analysis (PCA) is a valuable tool
for different methods that automate the process monitoring procedure for the cases where most of
the data variations cannot be captured in two or three dimensions. There are many motivations for
using PCA as the dimensionality reduction technique in these methods (Chiang, Russell, & Braatz,
2001). Firstly, PCA has the ability to produce lower-dimensional representations of the data which
generalizes the required features better than the entire dimensionality of the data set. Thus, using
PCA can increase the strength and proficiency of the fault detection and diagnosis system.
Secondly, an important feature of PCA is the structure abstracted by PCA, which can be useful in
identifying either the variables responsible for the fault or the variables most affected by the fault.
Thirdly, PCA has the ability to separate the observation space into a subspace capturing the
systematic trends of the process and a subspace containing essentially the random noise. Chiang
et al. states that it is widely accepted that certain faults primarily affect one of the two subspaces
(Chiang, Russell, & Braatz, 2001). Thus, it is proposed that applying one measure developed for
one subspace and another measure developed for the other subspace can increase the sensitivity of
the process monitoring scheme to faults.
3.3.1. PRINCIPAL COMPONENT ANALYSIS (PCA) FAULT DETECTION

It is important to realize that Principal Component Analysis (PCA) is a linear dimensionality technique. PCA determines a set of orthogonal vectors named as loading vectors, ordered by the amount of variance explained in the loading vector directions (Chiang, Russell, & Braatz, 2001). Continuing the notation provided before, we can assume that the data in the training set to be consisting of \( m \) observation variables and \( n \) observations for each variable. Thus, a \( X \in \mathbb{R}^{n \times m} \) training matrix can be created by storing all the information available for process monitoring. The loading vectors can be calculated by solving the stationary points of the optimization problem using Equation 3.8:

\[
\max_{v \neq 0} \frac{v^T X^T X v}{v^T v}
\]

Eq. 3.8

where \( v \in \mathbb{R}^m \). Now the stationary points can be computed via a singular value decomposition using Equation 3.9:

\[
\frac{1}{\sqrt{n} - 1} X = U \Sigma V^T
\]

Eq. 3.9

where \( U \in \mathbb{R}^{n \times n} \) and \( V \in \mathbb{R}^{m \times m} \) are unitary matrices and the matrix \( \Sigma \in \mathbb{R}^{n \times m} \) contains the non-negative real singular values of decreasing magnitude along its main diagonal and zero off-diagonal elements. The loading vectors are the orthonormal column vectors in the matrix \( V \), and the variance of the training set projected along the \( i^{th} \) column of \( V \) is equal to \( \sigma_i^2 \). Solving Equation 3.9 is equivalent to solving an eigenvalue decomposition of the sample covariance matrix \( S \); as in Equation 3.10:
\[ S = \frac{1}{n-1} X^T X = V \Lambda V^T \quad \text{Eq. 3.10} \]

where the diagonal matrix \( \Lambda = \Sigma^T \Sigma \in \mathbb{R}^{m \times m} \) contains the non-negative real eigenvalues of decreasing magnitude and the \( i^{th} \) eigenvalue is equal to the square of the \( i^{th} \) singular value as in \( \lambda_i = \sigma_i^2 \). The projections of the observations in \( X \) into the lower-dimensional space are contained in the score matrix; \( T \), by selecting the columns of the loading matrix \( P \) to correspond to the loading vectors associated with the first \( a \) singular values as in Equation 3.11:

\[ T = X P \quad \text{Eq. 3.11} \]

The projection of \( T \) back into the \( m \)-dimensional observation space can be accomplished by using Equation 3.12:

\[ \hat{X} = T P^T \quad \text{Eq. 3.12} \]

Finally, the difference between \( X \) and \( \hat{X} \) is the residual matrix \( E \):

\[ E = X - \hat{X} \quad \text{Eq. 3.13} \]

To sum up, Principal Component Analysis (PCA) maps the measured variables into the Principal Component (PC) space, which is typically of a lower dimensionality, as can be seen from Figure 3.5. Chiang et al. states that the residual matrix captures the variations in the observation space spanned by the loading vectors associated with the \( m - a \) smallest singular values (Chiang,
Russell, & Braatz, 2001). The subspace spanned by $\hat{X}$ is called the score subspace and the subspace spanned by $E$ is called the residual subspace. Chiang et al. states that the subspace contained in the matrix $E$ has a small signal-to-noise ratio, and thus the removal of this space from $X$ can produce a more accurate representation of the process, $\hat{X}$.

Figure 3.5: Shows the Principal Component Analysis (PCA) transformation of measured data into the Principal Component (PC) Space.

A new observation vector in the testing set, $x \in \mathbb{R}^m$, can be projected into the lower-dimensional score space as in Equation 3.14:

$$t_i = x^T p_i$$  \hspace{1cm} \text{Eq. 3.14}$$

where $p_i$ is the $i^{th}$ loading vector. The transformed variable $t_i$ is also called the $i^{th}$ principal component of $x$. The projection of the observation vector $x$ into the score and residual spaces are
visualized in Figure 3.6. Chiang et al. states that to distinguish between the transformed variables and the transformed observation, the transformed variables are called principal components and the individual transformed observations are called scores (Chiang, Russell, & Braatz, 2001). The number of principal components is always less than or equal to the number of original measured variables and the transformation is defined in such a way that the first principal component has the largest possible variance. In order to further underline the strength of Principal Component Analysis (PCA) over univariate methods a comparison can be made. Using univariate statistical process monitoring methods, each of the scores mentioned here can be monitored separately. On the other hand, using PCA with the vectors projected into the lower-dimensional space; only \( a \) number of variables need to be monitored, compared to the \( m \) number of variables without making use of PCA.

Figure 3.6 Shows the projection of the observation vector \( x \) into the score and residual spaces, and the computation of the filtered observation \( \hat{x} \) (Chiang, Russell, & Braatz, 2001).
As mentioned in Section 3.1.3., $T^2$ statistic can be used to detect faults for multivariate process data (Chiang, Russell, & Braatz, 2001). Using the same notation used throughout the thesis, given that an observation vector $x$ is present and assuming again that $\Lambda = \Sigma^T \Sigma$ is invertible; the $T^2$ statistic can be directly calculated from the PCA representation as in Equation 3.15:

$$T^2 = x^T V (\Sigma^T \Sigma)^{-1} V^T x$$  \hspace{1cm} \text{Eq. 3.15}$$

As mentioned above, the $T^2$ statistic tends to be an inaccurate representation of the in-control process behavior, when the number of observation variables is large and the amount of data available is relatively small. By including in the matrix $P$ the loading vectors associated with the $\alpha$ largest singular values, the $T^2$ statistic for the lower-dimensional space can be directly computed as in Equation 3.16 (Chiang, Russell, & Braatz, 2001):

$$T^2 = x^T P \Sigma_{\alpha}^{-2} p^T x$$  \hspace{1cm} \text{Eq. 3.16}$$

where $\Sigma_{\alpha}$ contains the first $\alpha$ rows and columns of $\Sigma$. It is important to note that the $T^2$ statistic measures the variations in the score space only. For the cases when the actual covariance matrix is estimated from the sample covariance matrix, the $T^2$ statistic threshold can be calculated using Equation 3.17:

$$T^2_{\alpha} = \frac{a(n-1)(n+1)}{n(n-a)} F_{\alpha}(a, n-a)$$  \hspace{1cm} \text{Eq. 3.17}$$

$T^2$ statistic can also be used to detect the outliers in the training set using Equation 3.18:
\[
T^2_\alpha = \frac{(n - 1)^2 \left(\frac{a}{n - a - 1}\right)}{n \left(1 + \left(\frac{a}{n - a - 1}\right)\right)} F_a(a, n - a - 1)
\]

Eq. 3.18

It is important to note that the \(T^2\) statistic is overly sensitive to inaccuracies in the PCA space corresponding to the smaller singular values. The main reason for this is PCA directly measuring the variation along each of the loading vectors. PCA directly measures the scores corresponding to the smaller singular values. Using another process monitoring measure, namely the Q statistic, can overcome this problem in a degree. The Q statistic is a squared 2-norm measuring the deviation of the observations to the lower-dimensional PCA representation. Also known as the Squared Prediction Error (SPE), the Q statistic can monitor the portion of the observation space corresponding to the \(m - a\) smallest singular values. In order to define the Q statistic, a residual vector \(r\) can be determined as the projection of the observation \(x\) into the residual space by using Equation 3.19:

\[
r = (I - P P^T) x
\]

Eq. 3.19

Then, the Q statistic can be defined as in Equation 3.20:

\[
Q = r^T r
\]

Eq. 3.20

The Q statistic does not suffer from an oversensitivity to inaccuracies in the smaller singular values simply because of the fact that the Q statistic does not directly measure the variations along each loading vector but measures the total sum of variations in the residual space (Chiang, Russell, &
In order to calculate the threshold for the $Q$ statistic; $\theta_i$ and $h_0$ should be defined as in Equation 3.21 and 3.22 (Chiang, Russell, & Braatz, 2001):

$$\theta_i = \sum_{j=a+1}^{n} \sigma_j^{2i}$$  \hspace{1cm} \text{Eq. 3.21}

$$h_0 = 1 - \frac{2 \theta_1 \theta_3}{3 \theta_2^2}$$  \hspace{1cm} \text{Eq. 3.22}

Then the threshold to be used in fault detection and diagnosis for the $Q$ statistic, or the Squared Prediction Error (SPE), can be calculated from Equation 3.23; where $c_\alpha$ is the normal deviate corresponding to the $(1 - \alpha)$ percentile:

$$Q_\alpha = \theta_1 \left[ \frac{h_0 c_\alpha \sqrt{2 \theta_2}}{\theta_1} + 1 + \frac{\theta_2 h_0 (h_0 - 1)}{\theta_1^2} \right]^{1/h_0}$$  \hspace{1cm} \text{Eq. 3.23}

Compared to the $T^2$ statistic, the $Q$ statistic mainly measures the random variations of the process. An example can be the variations associated with the measurement noise. The $Q$ statistic threshold can be implemented in the process monitoring system in order to define the normal variations for the random noise and thus a violation of the threshold would indicate that the random noise has significantly changed. $T^2$ statistic and $Q$ statistic can detect different faults by using the appropriate thresholds. Thus, both methods should be employed together in a process monitoring scheme in order to make use of the advantages of both methods. In the process monitoring system created for this research, both methods are employed at the same time for fault detection and diagnosis using
the Principal Component Analysis functionality. Sample $T^2$ statistic and Q statistic (SPE) charts can be seen in Figure 3.7:

![Sample T² and Q statistic plots](image)

Figure 3.7: Sample $T^2$ statistic plot, and Q statistic, or Squared Prediction Error (SPE), plot.

### 3.3.2. PRINCIPAL COMPONENT ANALYSIS (PCA) FAULT IDENTIFICATION

After the fault detection phase of the process monitoring scheme is successfully applied and a fault is detected, determining the cause of the out-of-control status is the next step (Chiang, Russell, & Braatz, 2001). This task of fault diagnosis can be rather challenging for modern industrial systems; since the number of process variables is large and the processes are highly integrated. Another factor making fault diagnosis process challenging is the fact that many of the measured variables may deviate from their set-points for only a short point of time whenever a fault occurs. The reason
for this is the existing control loops in the systems bringing the variables back to their set-points even though the fault continues to persist in the whole system. This type of a system behavior leads to the status of the faults being disguised and thus it makes the process control system’s job harder to diagnose faults in a proper manner. The job to automate the process monitoring scheme gets harder trying to correctly isolate the correct fault acting on the system.

As mentioned before, the objective of fault identification is to determine which observation variables are most relevant to diagnosing the fault (Chiang, Russell, & Braatz, 2001). This requires the plant operators and engineers to focus on the subsystems most likely where the fault occurred. With the help of an automated fault diagnosis and detection system, process operators and engineers can be actively incorporated in the process monitoring scheme. This approach significantly reduces the time to recover in-control operations. As mentioned before, univariate statistical techniques do not account for correlations among the process variables. As a result, univariate statistical techniques for fault identification can leave out variables that are responsible for the fault (Chiang, Russell, & Braatz, 2001). Another drawback for using univariate statistical techniques for fault identification can be that the system can give alarm readings for so many variables that the engineer has little guidance on the main variables of concern.

These problems can be overcome by using multivariate statistical techniques for fault identification like contribution plots. Contribution plots are a Principal Component Analysis (PCA) approach to fault identification which improve upon the univariate statistical techniques by taking the spatial correlations into account (Chiang, Russell, & Braatz, 2001). Contribution plots are created in an effort to quantify the contribution of each process variable to the individual scores.
of the PCA representation. In order to accomplish this, the contributions of the scores responsible for out-of-control status are summed up for each process variable.

In order to create a contribution plot, the first step is to check the normalized scores \((t_i/\sigma_i)^2\) for the observation \(x\) and determine the scores \(r \leq a\) responsible for the out-of-control status (Chiang, Russell, & Braatz, 2001). As used before, \(t_i\) is the score of the observation projected onto the \(i^{th}\) loading vector, and \(\sigma_i\) is the corresponding singular value. The next step is calculating the contribution of each variable \(x_j\) to the out-of-control scores \(t_i\) using Equation 3.24:

\[
cont_{i,j} = \frac{t_i}{\sigma_i^2} p_{i,j} (x_j - \mu_j)
\]

where \(p_{i,j}\) is the \((i,j)^{th}\) element of the loading matrix \(P\). The calculated contribution should be set equal to zero in case the result turns out to be negative. In order to calculate the total contribution of the \(j^{th}\) process variable \(x_j\), Equation 3.25 should be used:

\[
CONT_j = \sum_{i=1}^{r} (cont_{i,j})
\]

After the total contribution for each variable is calculated, the values \(CONT_j\) for all \(m\) process variables, \(x_j\), should be plotted on a single graph for ease of use. A sample contribution plot can be seen in Figure 3.8. Using the total contribution values, \(CONT_j\), the variables responsible for a specific fault can be prioritized or ordered from high to low. Using this information, plant operators
and engineers can focus on the variables with high contribution values together with using their knowledge about the process in order to determine the cause of the out-of-control status.

Figure 3.8: Sample contribution plot from a sample process for faults in: (a) Reactor cooling water inlet temperature, and (b) Feed composition ratio.

3.3.3. PRINCIPAL COMPONENT ANALYSIS (PCA) FAULT DIAGNOSIS

As discussed above, fault identification techniques identify the variables associated with the faulty subsystem and thus assist in diagnosing the fault. However, in order to diagnose a fault properly the fault identification phase should be taken one more step forward; because it may take a considerable amount of time and process expertise on behalf of the plant operators and engineers before the fault is properly diagnosed (Chiang, Russell, & Braatz, 2001). By employing an automated fault diagnosis option in the process monitoring scheme, much of this required time and expertise can be eliminated. One approach to aid this goal is constructing separate PCA models for each process unit in the industrial system. By using this approach, whenever the PCA model for a particular unit indicates that the process is in out-of-control status, a fault associated with the
particular unit is assumed to occur. This approach helps to narrow down the cause of the abnormal process operations; however it does not clearly diagnose the cause of the fault. This realization leads to distinguishing fault isolation techniques, which are based on non-supervised classification, from fault identification techniques, which are based on supervised classification. Another approach to aid fault diagnosis is developing multiple PCA model, where each model is based on the data collected during a specific fault; rather than using a single PCA model. Multiple PCA model approach helps to handle a larger number of faults than a single PCA model. Raich and Cinar state using this method as a combination of principal component analysis and discriminant analysis (Raich & Cinar, 1995).

As mentioned above, a single PCA model created using all the data from all fault classes is the first approach for using PCA in fault diagnosis. For this method, all the data available for all fault classes is stacked into matrix $X$ and the loading matrix $P$ is calculated using Equation 3.9 or Eq. 3.10. The maximum likelihood classification for an observation $x$ is the fault class $i$ with the maximum score discriminant; $g_i(x)$. The score discriminant, $g_i(x)$, can be calculated using Equation 3.26 (Chiang, Russell, & Braatz, 2001):

$$g_i(x) = -\frac{1}{2} (x - \bar{x}_i)^T P (P^T S_i P)^{-1} P^T (x - \bar{x}_i) + \ln(p_i) - \frac{1}{2} \ln[\det(P^T S_i P)]$$  
Eq. 3.26

where $\bar{x}_i$ is the mean vector for class $i$ and can be calculated using:

$$\bar{x}_i = \frac{1}{n_i} \sum_{x_j \in X_i} x_j$$  
Eq. 3.27
where \( n_i \) is the number of data points in the fault class \( i \), \( \chi_i \) is the set of vectors \( \chi_j \) which belong to the fault class \( i \), and \( S_i \in \mathbb{R}^{m \times m} \) is the sample covariance matrix for class \( i \), which can be calculated using Equation 3.2. As mentioned above, the second approach for using PCA in fault diagnosis can be achieved by using multiple PCA models, assuming the PCA models retain the important variations in discriminating between the faults. Just like the single PCA model approach, the maximum likelihood classification for an observation \( x \) is the fault class \( i \) with the maximum score discriminant; \( g_i(x) \), for using multiple PCA models. The discriminant function for using with multiple PCA models can be calculated using (Chiang, Russell, & Braatz, 2001):

\[
g_i(x) = -\frac{1}{2} x^T P_i \Sigma_{\alpha,i}^{-2} P_i^T x + \ln(p_i) - \frac{1}{2} \ln[\det(\Sigma_{\alpha,i}^2)]
\]

Eq. 3.28

where \( P_i \) is the loading matrix for fault class \( i \). \( \Sigma_{\alpha,i} \) is the diagonal matrix shown in Equation 3.16, and also \( \Sigma_{\alpha,i}^2 \) is the covariance matrix of \( P_i x \). \( p_i \) is the overall likelihood of fault class \( i \). It is important to note that, in order to use Equation 3.28, the observation \( x \) has to be normalized and auto-scaled according to the mean and standard deviation of the training set for fault class \( i \). As mentioned above, the matrices \( P_i \), \( \Sigma_{\alpha,i} \), and \( p_i \) in Equation 3.28 only depend on the discriminant function of fault class \( i \). All these different discriminant functions are derived individually and thus the useful information for other classes is not utilized when each model is derived separately. On the other hand, Equation 3.26 utilizes information from all fault classes as mentioned above. Chiang et al. states that the discriminant function in Equation 3.26 can significantly outperform Equation 3.28 for fault diagnosis because of these properties (Chiang, Russell, & Braatz, 2001). If it is assumed that the overall likelihood for all fault classes is the same and the sample covariance
matrix, $P_l x$, of for all classes is the same; the use of the score discriminant in Equation 3.28 reduces to the use of $T^2_l$ statistic as:

$$T^2_l = x^T P_l \Sigma^{-2}_{\alpha_l} P_l^T x$$

Eq. 3.29

For this case, the score discriminant will select the fault class which has the minimum $T^2_l$ statistic (Chiang, Russell, & Braatz, 2001).

### 3.4. FISHER DISCRIMINANT ANALYSIS (FDA)

As stated before, the dimensionality reduction technique used in the feature extraction step plays an important role in terms of reducing the misclassification rate in fault detection and diagnosis. The dimensionality reduction step is critically important when the dimensionality of the observation space is large while the number of observations in each class is relatively small. Use of Principal Component Analysis (PCA) as a dimensionality reduction technique in fault detection and diagnosis is discussed in Section 3.3 in detail. PCA has several advantages in terms of properties for using in fault detection; however this is not exactly the case for fault diagnosis as stated before. The main reason for this is PCA not taking into account the information between classes when determining the lower-dimensional representation. An alternative to PCA can be Fisher Discriminant Analysis (FDA), which is another dimensionality reduction techniques that has been studied widely in literature (He, Qin, & Wang, 2005; Russell & Braatz, 1998; Chiang, Kotanchek, & Kordon, 2004; Yin, Ding, Haghani, Hao, & Zhang, 2012). Chiang et al. states that unlike Principal Component Analysis (PCA), Fisher Discriminant Analysis (FDA) takes into
account the information between the classes and thus has advantages over PCA fault diagnosis (Chiang, Russell, & Braatz, 2001).

As mentioned above, for fault detection and diagnosis purposes, data collected during plant operation under certain faults are categorized into different classes, where each class contains data representing a certain fault. Fisher Discriminant Analysis (FDA) is optimal in terms of maximizing the separation amongst these classes. This property of FDA makes it more apparent about why it can outperform PCA in fault diagnosis for some cases. FDA determines a set of linear transformation vectors, ordered in terms of maximizing the scatter between the classes while minimizing the scatter within each class (Chiang, Russell, & Braatz, 2001).

The notation used throughout the thesis is preserved in analyzing Fisher Discriminant Analysis (FDA) by defining the number of observations as $n$, the number of measurement variables as $m$, number of classes as $p$, and the number of observations in the $j^{th}$ class as $n_j$. $x_i$ is the representation of the vector of measurement variables for the $i^{th}$ observation. As used before, if the training data for all classes are stacked into the matrix $X \in \mathbb{R}^{n \times m}$, then the transpose of the $i^{th}$ row of $X$ is the column vector $x_i$. In order to understand Fisher Discriminant Analysis (FDA, the total scatter matrix, the within-scatter matrix, within-class-scatter matrix and the between-class-scatter matrix should be defined (Chiang, Russell, & Braatz, 2001). The total scatter matrix is defined in Equation 3.30, where $\bar{x}$ is the total mean vector defined in Equation 3.31:

$$S_t = \sum_{i=1}^{n} (x_i - \bar{x}) (x_i - \bar{x})^T$$  

Eq. 3.30
\[ \bar{x} = \frac{1}{n} \sum_{i=1}^{n} x_i \]  

Eq. 3.31

The within-scatter matrix for class \( j \) is defined in Equation 3.32, where \( \bar{x}_j \) is the mean vector for class \( j \) defined in Equation 3.33 and \( \chi_j \) defined as the set of vectors \( x_i \) which belong to the class \( j \):

\[ S_j = \sum_{x_i \in \chi_j} (x_i - \bar{x}_j)(x_i - \bar{x}_j)^T \]  

Eq. 3.32

\[ \bar{x}_j = \frac{1}{n_j} \sum_{x_i \in \chi_j} x_i \]  

Eq. 3.33

The within-class-scatter matrix is defined in Equation 3.34, and the between-class-scatter matrix is defined in Equation 3.35.

\[ S_w = \sum_{j=1}^{p} S_j \]  

Eq. 3.34

\[ S_b = \sum_{j=1}^{p} n_j (\bar{x}_j - \bar{x})(\bar{x}_j - \bar{x})^T \]  

Eq. 2.35

As can be seen from Equation 3.36, the total-scatter matrix is equal to the sum of the between-scatter matrix and the within-scatter matrix.
\[ S_t = S_b + S_w \]  

Eq. 3.36

The objective of the first FDA vectors can be seen from Equation 3.37 as to maximize the scatter between classes while minimizing the scatter within classes assuming \( S_w \) is invertible where \( \nu \in \mathbb{R}^m \):

\[
\max_{\nu \neq 0} \frac{\nu^T S_b \nu}{\nu^T S_w \nu}
\]

Eq. 3.37

Chiang et al. states that it can be shown that the linear transformation vectors for FDA can be calculated by computing the stationary points of the optimization problem in Equation 3.37 (Chiang, Russell, & Braatz, 2001). Thus, the FDA vectors are equal to the eigenvectors \( w_k \) of the generalized eigenvalue problem, where the eigenvalues \( \lambda_k \) indicate the degree of overall separability among the classes by projecting the data onto \( w_k \) as in Equation 3.38:

\[
S_b w_k = \lambda_k S_w w_k
\]

Eq. 3.38

As long as \( S_w \) is invertible, the FDA vectors can be computed from the generalized eigenvalue problem. \( S_w \) being invertible is most commonly the case when the number of observations \( n \) is significantly larger than the number of measurements \( m \) for the case in practice. It should be obvious that the first FDA vector is the eigenvector associated with the largest eigenvalue, the second FDA vector is the eigenvector associated with the second largest eigenvalue, and so on.
Chiang et al. states that a large eigenvalue $\lambda_k$ indicates that when the data in the classes are projected onto the associated eigenvector $w_k$; there is overall a large separation of the class means relative to the class variances, and consequently, a large degree of separation among the classes along the direction $w_k$ (Chiang, Russell, & Braatz, 2001). There will always be at most $p - 1$ eigenvalues which are not equal to zero; since the rank of $S_b$ is less than $p$, and thus FDA provides useful ordering of the eigenvectors in these directions.

3.4.1. FISHER DISCRIMINANT ANALYSIS (FDA) FAULT DETECTION AND DIAGNOSIS

As stated before, when Fisher Discriminant Analysis (FDA) is applied for pattern classification, the dimensionality reduction technique is applied to the data in all the classes simultaneously. The discriminant function for Fisher Discriminant Analysis (FDA) can be seen from Equation 3.39 with $W_a \in \mathbb{R}^{m \times a}$ as the matrix containing the eigenvectors (Chiang, Russell, & Braatz, 2001):

$$
g_j(x) = -\frac{1}{2}(x - \bar{x}_j)^T W_a \left(\frac{1}{n_j - 1} W_a^T S_j W_a \right)^{-1} W_a^T (x - \bar{x}_j) + \ln(p_i) - \frac{1}{2} \ln[\det\left(\frac{1}{n_j - 1} W_a^T S_j W_a \right)]
$$

Eq. 3.39

Chiang et al. states that since FDA uses the class information to compute the reduced-dimensional space, so that the discriminant function in Equation 3.39 exploits that class information to a far greater degree that can be done by PCA (Chiang, Russell, & Braatz, 2001). Unlike multiple PCA models, FDA utilizes all $p$ fault class information when evaluating the discriminant function for each class. By defining the normal operating region data collected during normal operating
conditions as an additional class of data, FDA can be used detect faults. Intuitively, the strength of fault detection using FDA depends on the similarity between the normal data and the faulty data; normal data being the data collected during normal operating conditions and faulty data being the data collected during faulty conditions. Chiang et al. states that when there exists a transformation $W$ such that the data from the normal operating conditions can be reasonably separated from the other fault classes, using FDA for fault detection will produce small missed detection rates for the known fault classes (Chiang, Russell, & Braatz, 2001). The discriminant function provided in Equation 3.39 does not take into account unknown faults; namely the faults associated with data outside of the lower dimensional space defined by FDA vectors. Thus, Fisher Discriminant Analysis (FDA) may not detect unknown faults, which do not exist in the training dataset.

Fault detection using PCA and FDA can be shortly compared again by stating that PCA is better able to separate the data as a whole and FDA is better able to separate the data among classes. FDA can generally do a better job than PCA in fault diagnosis at lower reduction orders (Chiang, Russell, & Braatz, 2001). However, as mentioned before, it is important to note that there is no single method that outperforms the other for all applications, so both methods can outperform the other one for different process operations. It should also be noted that by having an increased number of data points in the training sets, the overall misclassification rates for all methods tend to go down as mentioned before.

3.5. SELF-ORGANIZING MAPS (SOM)

As discussed in detail in Section 3.4, when developing a data-driven process monitoring method based on pattern classification a critical step is the feature extraction and dimensionality reduction.
Previously, we have discussed PCA and FDA as main tools for feature extraction and dimensionality reduction within a classification scheme. The main drawback of these methods is that they are based on linear projection of the data set. There exist some alternative methods for process monitoring, which can extract the nonlinear relationship among the process variables without modeling the internal process states or structure explicitly such as Artificial Neural Networks (ANN) and Self-Organizing Maps (SOM). Along these lines, pattern classification system can be set up using neural networks or Self-Organizing Maps (SOM) in order to capture the nonlinearities in the data. The non-linearity function of SOM is the main advantage over PCA and FDA as well as its visualization tools, and is incorporated into the proposed multi-agent system (Corona, Mulas, Baratti, & Romagnoli, 2012). Basically, SOM simultaneously performs data compression in terms of vector quantization, and dimensionality reduction in terms of topographical preservation.

SOM can be seen as a nonlinear version of PCA if PCA is considered to fit a hyperplane into a data cloud and points are encoded as coordinates in that hyperplane. If the data cloud is curved, the regarding plane should also be curved, not linear. SOM adjusts to the real shape of the data by replacing the hyperplane with a discrete representation consisting of connected nodes. The connected nodes are updated to bend to the data cloud’s shape as SOM is trained. Self-Organizing Map (SOM) is a type of Artificial Neural Network (ANN) which is trained using unsupervised learning. These techniques are very useful when data are abundant, but expert knowledge is lacking; since pattern recognition approaches are based on inductive reasoning through generalization from a set of stored or learned examples of process behaviors.
Firstly, the Artificial Neural Network (ANN) idea was motivated from the study of the human brain (Chiang, Russell, & Braatz, 2001). Since human brain is made up of millions of interconnected neurons, these interconnections allow humans to implement pattern recognition computations. In an attempt to mimic the computational structures of the human brain, Artificial Neural Networks (ANN) were developed. Chiang et al. states that ANN is a nonlinear mapping between input and output which consists of interconnected “neurons” arranged in layers and these layers are connected such that the signals at the input of the neural net are propagated through the network (Chiang, Russell, & Braatz, 2001). In literature, there are many applications of Artificial Neural Networks (ANN) in fault detection and diagnosis (Watanabe, Hirota, Hou, & Himmelblau, 1994; Tzafestas & Dalianis, 1994; Asakura, Kobayashi, & Hayashi, 1998; Suewatanakul & Himmelblau, 1996). One of the most popular configuration of ANN is the three-layer feed-forward ANN, which consists of an input layer, a hidden layer, and an output layer. Each of these layers contain neurons or nodes. Intuitively, the input layer neurons correspond to input variables and the output layer correspond to output variables. Each neuron in the hidden layer is connected to all input layer neurons and output layer neurons, and no connection is allowed within its own layer and the information flow is in one direction only (Chiang, Russell, & Braatz, 2001).

It is common to use ANN for fault diagnosis by assigning the input neurons to process variables and the output neurons to fault indicators (Chiang, Russell, & Braatz, 2001). Thus, the number of output neurons is equal to the number of different fault classes in the training data. If the input neurons are associated with the fault $j$, the $j^{th}$ output neuron is assigned to “1”; if the input neuron is not associated to any fault, it is assigned to “0”. Chiang et al. states that each neuron $j$ in the hidden and output layers receives a signal from the neurons of the previous layer $v^T =$
\[v_1 \ v_2 \ ... \ v_r\], scaled by the weight \(w_j^T = [w_{1j} \ w_{2j} \ ... \ w_{rj}]\). The strength of connection between two linked neurons is represented in the weights, which are determined via the training process (Chiang, Russell, & Braatz, 2001).

Secondly, neural network models can also be used for unsupervised learning using Self-Organizing Maps (SOM). SOM is also known as Kohonen Self-Organizing Map. In the case of SOM, the neural network learns some internal features of the input vectors \(x\). Chiang et al. states that a SOM maps the nonlinear statistical dependencies between high-dimensional data into simple geometric relationships, which preserve the most important topological and metric relationships of the original data. This allows the data to be clustered without knowing the class memberships of the input data (Chiang, Russell, & Braatz, 2001).

A Self-Organizing Map (SOM) consists of two layers; namely an input layer and an output layer (Chiang, Russell, & Braatz, 2001). The output layer is also called the feature map; since it represents the output vectors of the output space. In general, the feature map can be \(n\)-dimensional; but two-dimensional maps are preferred popularly. Chiang et al. states that the topology in the feature map can be organized in a rectangular grid, a hexagonal grid, or a random grid; and the number of the neurons in the feature map depends on the complexity of the problem, but the number must not be so large that too much training time is required (Chiang, Russell, & Braatz, 2001). Figure 3.9 shows the learning procedure demonstration of an SOM. In Figure 3.9, yellow point represents the training data point, which is being moved towards the white training data. The learning algorithm runs through the data finding the BMUs and updating all prototype vectors in order to create the map shown in the last phase of Figure 3.9.
An overview of the SOM algorithm can be summarized in a few steps discussed in Chiang et al. (Chiang, Russell, & Braatz, 2001). To begin with, the weight $w_j$ connects all the $m_x$ input neurons to the $j^{th}$ output neuron. The input values to the SOM can be discrete or continuous; but the output values are binary. The first step in creating the SOM is to assign small random numbers to the initial weight vector $w_j$ for each neuron $j$ from the output map as the first iteration; $t = 0$. Secondly, an input vector $x$ from the training data is retrieved, and the Euclidian distance between $x$ and each weight vector $w_j$ is calculated as: $\|x - w_j\|$. Then, the neuron closes to $x$ is declared as the best matching unit (BMU) and is denoted as neuron $k$. Then, each weight vector is updated so that the BMU and its topological neighbors are moved closer to the input vector in the input space by using Equation 3.40 for neuron $j$ as the update rule:

$$w_j(t + 1) = w_j(t) + \alpha(t) [x(t) - w_j(t)] \quad \text{for } j \in N_k(d) \quad \text{Eq. 3.40}$$

$$w_j(t + 1) = w_j(t + 1) = w_j(t) \quad \text{for } j \notin N_k(d)$$

where $N_k(d)$ is the neighborhood function around the winning neuron $k$ and $0 < \alpha(t) < 1$ is the learning coefficient. Both the learning coefficient and the neighborhood function are decreasing
functions of iteration number $t$. The iterative process is repeated for all the training samples until convergence is achieved and the final accuracy of the SOM depends on the number of iterations. Chiang et al. states that SOM has a fairly good recall ability when applied to new data and generally an increase in the number of neurons and the number of iterations would improve the clustering of classes while increasing computational time (Chiang, Russell, & Braatz, 2001). For fault detection purposes using Self-Organizing Maps (SOM), a SOM is trained to form a mapping of the input space during normal operating conditions, and then a fault can be detected by monitoring the distance between the observation vector and the BMU.

### 3.5.1. SELF-ORGANIZING MAPS (SOM) FAULT DETECTION AND DIAGNOSIS

In using the SOM algorithm, the first step is to train Self-Organizing Maps to the historical data, including all the different faults and the normal operating region. The created maps can be used for data analysis and point classification. Historical data analysis can be done using visualizations like component planes, and hit diagrams. The created maps can also be used as unsupervised clustered maps to identify variables important to clusters in the data. In order to train the SOM maps, data analysis is used for selecting variables and labeling clusters. Data analysis can be also used for the classification of new data points.

For fault detection, a SOM is trained to historical data from normal operating region of the process to create a non-parametric model proportional to the probability distribution of the data. Self-Organizing Map is a graph composed of an array of nodes connected together. The grids in the graph can be rectangular or hexagonal. Each node in the graph is called a prototype vector and they model the observation vectors in the data space. The prototype vectors have a representation
in both the input and output space. The Best Matching Unit (BMU) is defined as the closest prototype vector to a data point in the input space. The BMU can represent the referred data point on the map. The BMU of $i^{th}$ data point $v_i$ can be found by finding the closest prototype vector $m_k$ using Equation 3.41:

$$ pBMU_i = \arg \min_k (\partial(m_k, v_i)), \quad \forall k \quad \text{Eq. 3.41} $$

The first step in implementing the SOM procedure is selecting the shape of the map. Then, along the PCA hyperplane the prototypes vectors’ positions are initially embedded in the data space. After that, the map is trained to capture curves in the manifold as mentioned before. The prototype vectors’ positions are updated using each data point from a training set according to Equation 3.42:

$$ m_k(t + i) = m_k(t) + \alpha(t) h_{k,BMU_i}(v_i(t) - m_k(t)) \quad \text{Eq. 3.42} $$

where $t$ is the discrete time coordinate of the mapping steps and $\alpha$ is the monotonically decreasing learning rate. The scalar $h_{k,BMU_i}$ denotes a neighborhood kernel function centered at the BMU. Using the smallest Euclidean distance from the data vector of interest, data vectors are matched to the prototype vectors. A neighborhood kernel $h_{k,BMU_i}$ centered at $m_k(t)$ is usually chosen in the Gaussian form as in Equation 3.43:

$$ h_{m_k,BMU_i} = \exp \left( \frac{||m_k - BMU_i||^2}{2\sigma^2} \right) \quad \text{Eq. 3.43} $$
where $\sigma(t)$ denotes the monotonically decreasing width of the kernel that allows for a regular smoothing of the prototypes. The SOM algorithm continues to update the locations of the prototype vectors by passing over the data and it terminates after some predefined number of time steps have passed or prototype updating becomes negligible.

A well trained SOM can be used as a model for data and one example can be a Gaussian mixture model (GMM). As explored by Heskes, SOM can be used to create a Gaussian mixture model (GMM) using SOM’s vector quantization abilities (Heskes, 2001). The vector quantization abilities of SOM provide a compressed representation of a group of data. For a Self-Organizing Map (SOM) with $M$ map nodes, the Gaussian mixture model is the weighted sum of $M$ component Gaussian densities as can be seen from:

$$ p(x|\lambda) = \sum_{i=1}^{M} w_i g(x|\mu_i, \Sigma_i) \quad \text{Eq. 3.44} $$

where $x$ is a $D$-dimensional continuous-valued data vector, the $w_i$ are the mixture weights, and $g(x|\mu_i, \Sigma_i)$ are the component Gaussian densities. Each component density is a $D$-variate Gaussian function of the form:

$$ g(x|\mu_i, \Sigma_i) = \frac{1}{(2\pi)^{D/2} |\Sigma_i|^{1/2}} \exp\left(-\frac{1}{2} (x - \mu_i)' \Sigma_i^{-1} (x - \mu_i)\right) \quad \text{Eq. 3.45} $$

where $\mu_i$ is the mean vector, and $\Sigma_i$ is the covariance matrix. The mixture weights satisfy the following constraint:
The complete model is parameterized by the mean vectors, covariance, and mixture weights from all component densities. The SOM model is collectively represented in $\lambda$ (Reynolds, 2009):

$$\lambda = \{w_i, \mu_i, \Sigma_i\} \quad \text{for } i = 1, ..., M$$  \hspace{1cm} \text{Eq. 3.47}

By using this model, instead of using the data as the centers of Gaussian kernels, SOM map vectors are used. The discriminant function resulting is:

$$g(x)_i = \log(p(x|\lambda))$$  \hspace{1cm} \text{Eq. 3.48}

For discriminant analysis purposes; a normal point will have a small positive value of discriminant function and a faulty point will have a larger positive value of discriminant function. Once the probability of the faulty point occurring on each map is calculated, the maximum discriminant function assigns a point to an operating regime associated to a class:

$$g(x)_i > g(x)_j \quad \forall j \neq i$$  \hspace{1cm} \text{Eq. 3.49}
3.6. CONCLUSIONS

To sum up, this Chapter of the thesis started with describing multivariate statistics including an introduction about data pretreatment, together with the developmental procedure from univariate statistical monitoring to $T^2$ statistic, and pattern classification and its use in process monitoring strategies. This Section of the thesis also provides detailed information about Principal Component Analysis (PCA), Fisher Discriminant Analysis (FDA), and Self-Organizing Maps (SOM) and their use in fault detection and diagnosis. The important property of PCA is that it is the most widely used and simple one of these three approaches. PCA mainly exploits the information extracted from the difference within a class, whereas FDA exploits the information extracted from the difference amongst different classes. Both PCA and FDA are linear methods, whereas SOM is a non-linear method. As mentioned before no single process monitoring method is absolutely superior to another in all possible given conditions. Certain methods work better in fault detection compared to other methods for a given process and a given fault related to that process. In addition to that, some other methods might work better in fault diagnosis for a given process and a given fault related to that process. This is the reason for implementing all three different data-based process monitoring methods into the multi-agent system. In order to make use of all the helpful information and insight provided by all these different methods based solely on process data, all three methods are individually implemented.
4. MULTI-AGENT BASED SYSTEMS

The computational needs and capabilities for software engineering are getting more complex. The number of components and goals are increasing and thus creating new software design problems. These problems in software engineering are applicable for advanced process monitoring systems too; since they require extensive coding of all the features and functionalities. The problem of complexity can be overcome in a degree by the introduction of new programming paradigms. Multi-agent based systems is an important current example for a new approach in software programming in that sense.

In the agent based programming approach individual software entities are referred to as agents. Multi-agent systems (MAS) consist of two main elements: agents and their environment (Bellifemine, Caire, & Greenwood, 2007). Agents in a system are generally software based; however robots or human beings can also be a part of a MAS. An improvement that agent-oriented programming brings is the self-control of actions by the use of specific threads of individual agents. In an active agent environment, individual agents might exist as having their own responsibilities and goals while interacting with other agents and the user. In addition, agents can be active by means of monitoring their environment and taking actions based on the observations.

Multi-agent systems (MAS) were introduced as a concept in software engineering in 1980s and has been developing ever since (Shoham, 1993). MAS have been used both by academia for research purposes and in also practical industrial applications. Multi-agent systems can consist of different numbers of agents from tens to thousands of agents where necessary. For decades
researches from different fields like artificial intelligence, software engineering, robotics, social and biological sciences have focused on developing multi-agent systems (Lesser, 1999). A simple visualization for the characterization of a multi agent system can be seen from Figure 4.1:

![Figure 4.1: The characterization of a multi-agent system (Jennings N. R., 2000).](image)

General properties of agents can be summarized using the information provided by Jennings (Jennings N. R., 2000). Firstly, agents are clearly identifiable software entities with well-defined boundaries and interfaces. Secondly, being situated in a particular environment, agents receive inputs related to the state of their environment through sensors and they act on the environment through effectors. Thirdly, agents have particular objectives to achieve since they are designed to fulfill a specific purpose. In addition, agents are referred as autonomous; since they have control over their internal state and over their own behavior. Finally, agents are capable of exhibiting flexible problem solving behavior in order to achieve their design objectives. Agents need to be able to respond to changes that occur in their environment in a timely fashion and act in anticipation of future goals. Both of these last two properties can be summarized as agents being reactive. Mařík & Lažanský state that the agent paradigm developed from the notion of objects in
object-oriented programming (Mařík & Lažanský, 2007). Furthermore, Jennings state that, unlike objects, agents have their own internal state and are autonomous and proactive in that they can plan and execute their own activities (Jennings N. R., 2001). While object oriented approach is suitable for conventional decision making processes, agent-based approach gives an opportunity to effectively use cooperative decision making. The difference between conventional decision making and cooperative decision making can be seen from Figure 4.2. As can be seen from the figure, in conventional decision making even simple computations needs communication between the master and the slave. However, in cooperative decision making, agents can make their own decisions and communicate with the authority whenever necessary.

![Conventional decision making versus cooperative decision making](image)

**Figure 4.2: Conventional decision making versus cooperative decision making (Mařík & Lažanský, 2007).**

Using these properties, different definitions of agents can be given; all of which fall under the same generally accepted description of agents in the academia. Nwana states that an agent can be physical or software; and can be defined as an autonomous entity that is pro-active, reactive, social, and able to take part to an organized activity, in order to achieve its goals, by interacting with other
agents and users (Nwana, 1996). Russell and Norvig defines the agent as a software entity that can perceive its environment through sensors and act upon that environment through actuators (Russell & Norvig, 2003). Jennings states that an agent is an encapsulated computer system that is situated in some environment and that is capable of flexible, autonomous action in that environment in order to meet its design objectives (Jennings N. R., 2000).

Furthermore, Jennings states that agent-based approach as a programming paradigm is suitable for designing and building complex systems (Jennings N. R., 2000). Jennings defines complex systems as systems where the patterns and the outcomes of the interactions within the system are inherently unpredictable (Jennings N. R., 2000). Thus predicting the behavior of the overall complex system is extremely difficult or sometimes impossible because of the strong possibility of emergent behavior. In complex systems interactions will occur at unpredictable times, for unpredictable reasons, and between unpredictable entities. As a result, it is impossible to have information about all the interactions in the design phase of the multi-agent system. Therefore, agents are designed in a way to have the ability to make decision about the interactions they make while running. Jennings argues that agent-based approach is a natural and logical evolution of a range of contemporary approaches to software engineering (Jennings N. R., 2000). Accordingly, it is believed to succeed as a mainstream software engineering paradigm in the near-future. This is starting to be proven correct even further as many developed multi-agent systems show adaptability and robustness advantages over conventional process monitoring systems.

As stated above, multi-agent systems are computational systems that consist of two or more agents which interact or work together to perform some set of tasks or to satisfy some set of goals (Lesser,
Multi-agent systems introduce the possibility of agents having common or conflicting goals (Bellifemine, Caire, & Greenwood, 2007). Agents in an environment may interact with each other indirectly by acting on the environment; or directly via communication and negotiation. Agents in an environment can also communicate with each other when necessary using an agent communication language. Agents running in a Multi-agent systems may decide to cooperate for mutual benefit or may compete to serve their own interests. An agent in a multi-agent system operates asynchronously with respect to other agents to solve the problems defined for it and it has a certain level of autonomy. The autonomy aspect of an agent can be defined as the ability to make its own decisions about what activities to do, when to do them, what type of information should be communicated and to whom, and how to assimilate the information received. Autonomy of an agent is generally limited by the built-in design specifications of the agent, or as a result if dynamic agent organization in the environment such as coming to an agreement that specific agents should take on certain roles or adopt certain policies for some specified period of time. Another important feature of agents is adaptability. Adaptability is closely related to autonomy in the sense that the more autonomy an agent possesses the more adaptable it is to the emerging problem solving capabilities. The degree of autonomy and the range of adaptability are generally associated with the level of built-in intelligence/sophistication that an agent possesses (Lesser, 1999).

Bellifemine et al. states that industrial applications are very important for multi-agent systems because they are where the first multi-agent system techniques were experimented with and demonstrated their initial potential (Bellifemine, Caire, & Greenwood, 2007). There are many examples of the application of multi-agent systems in industrial settings. Jennings can be referred to for process control, Albert et al. can be referred to for system diagnostics, Parunak can be
referred to for manufacturing, Neagu et al. can be referred to for transportation logistics, and Greenwood et al. can be referred to for network management applications (Jennings N., 1994; Albert, Laengle, Woern, Capobianco, & Brighenti, 2003; Parunak, 1987; Neagu, Dorer, Greenwood, & Calisti, 2006; Greenwood, Vitaglione, Keller, & Calisti, 2006). There are also various applications developed using multi-agent systems in the area of process monitoring and control. One of the first and recognized example is the ARCHON (ARchitecture for Cooperative Heterogeneous ON-line Systems) Project (Cockburn & Jennings, 1996). The project is used in many different applications including electricity transport management and particle accelerator control (Corera, Laresgoiti, & Jennings, 1996; Perriolat, Skarek, Varga, & Jennings, 1996). Other recognized applications using multi-agent systems in the area of process monitoring and control include DaimlerChrysler’s Vision for Control project for process automation and Flavor Technology’s Manufacturing Agility Server (MAS) Project (Parunak, A Practitioners’ Review of Industrial Agent Applications, 2000).

The remainder of this Section is organized in two parts. Section 4.1 presents JADE (Java Agent DEvelopment framework) in detail. All the functionalities of JADE used in this research project are also described in this Section. Section 4.2 presents the multi-agent process monitoring system architecture designed in this research project. Detailed descriptions of all the agents designed and used are also presented in this Section.

4.1. JADE (Java Agent DEvelopment framework)

Woolridge and Jennings state that JADE (Java Agent DEvelopment framework) is a software platform that provides basic middleware-layer functionalities (Wooldridge & Jennings, 1995).
These functionalities are independent of the specific application and simplify the realization of distributed applications that exploit the software agent abstraction (Bellifemine, Caire, & Greenwood, 2007). One advantage of using multi-agent systems is that all the agents in the environment know how to execute many behaviors automatically after the design phase. An important instance of this automation feature is agent communication. Agents in an environment inherently know how to communicate with other agents in the environment using an agent communication language. This way, the developer of the system does not have to extensively code the communication protocols or messaging formats. Another important advantage is agents being multi-threaded. This way, the developer of the system can easily choose to run multiple agents on a single computer or a distributed system. These high-level abstractions provided by multi-agent systems make them require less time and effort to design the system itself. Making use of these properties via using a middleware, the designer of the system only needs to specify the behaviors of each agent and the communication between agents. One such agent-oriented middleware used in designing multi-agent systems is JADE (Java Agent DEvelopment framework). JADE is fully implemented in Java language and it is currently one of the most widely used middleware in creating multi-agent systems and it is a distributed middleware system with a flexible infrastructure (Bellifemine, Caire, & Greenwood, 2007). Bellifemine et al. state that the framework facilitates the development of complete agent-based applications by means of a run-time environment implementing the life-cycle support features required by agents, the core logic of agents themselves, and a rich suite of graphical tools (Bellifemine, Caire, & Greenwood, 2007). JADE is distributed as open source under the LGPL license and all the software relating to JADE can be found on the website: http://jade.tilab.com.
The Foundation of Intelligent Physical Agents (FIPA) was established in 1996 as an international non-profit association to develop a collection of standards relating to software agent technology. FIPA is an IEEE Computer Society standards organization that promotes agent-based technology and the interoperability of its standards with other technologies (IEEE Foundation for Intelligent Physical Agents, 2014). It is stated on their website that FIPA specifications represent a collection of standards which are intended to promote the interoperation of heterogeneous agents and the services that they can represent (IEEE Foundation for Intelligent Physical Agents, 2014).

Extensive information about FIPA specifications can be found in Bellifemine et al. (Bellifemine, Caire, & Greenwood, 2007). JADE is fully compliant with FIPA specifications and makes use of the complete Agent Management specification including the key services of AMS, DF, MTS and ACL; all of which will be explained in detail. Mainly due to its open source status and wide acceptance in industry and academics, JADE is now generally considered to be the leading FIPA-compliant open source agent framework (Bellifemine, Caire, & Greenwood, 2007).

JADE provides a run-time environment where the agents run actively on particular platforms. A JADE platform is composed of agent containers that can be hosted on a single machine or distributed over the network (Bellifemine, Caire, & Greenwood, 2007). Bellifemine et al. states that agents live in containers which are the Java process that provides the JADE run-time and all the services needed for hosting and executing agents (Bellifemine, Caire, & Greenwood, 2007). The specified main container must be started when JADE is initialized, must be active in the platform at all times, and all other containers must be registered with this main container on initialization. The configuration of the platforms and containers can be changed whenever required at run-time by creating new agents and moving agents from one machine to another one. JADE
also provides a library of classes for the programmers to use directly or by specifying certain properties. Another important functionality of JADE is having a set of graphical tools which allow the user to monitor and administer the activities of running agents.

Besides from the other properties mentioned above, the main container in each platform holds two special agents created automatically with initialization: AMS and DF (Grimshaw, 2010). The AMS (Agent Management System) provides the naming system in each platform to ensure each agent in a platform has a specific name and also represents authority. The AMS can also be used to create or kill agents on a remote machine. The DF (Directory Facility) provides a Yellow Pages service by providing agents the ability to find other agents by providing the service that agent requests in order to achieve its goals. The DF is used by any agent wishing to register its services or search for other available services.

Figure 4.3: Containers and platforms in JADE (Bellifemine, Caire, & Greenwood, 2007).
A sample container and platform structure in JADE can be seen from Figure 4.3. As can be seen from Figure 4.3, there are two platforms and four containers in total, each platform has a main container in it, and all the containers and platforms have distinctive names. Agents in all the containers have unique names and can communicate with each other regardless of where they are located. The DF and AMS agents can be seen in both of the main containers presented in Figure 4.3.

The communication between agents living on different platforms is based on modules called MTP (Message Transport Protocol). Bellifemine states that according to the FIPA specifications, a Message Transport Service (MTS) is one of the three most important services that an agent platform is required to provide; where the other two being the Agent Management Service and the Directory Facilitator (Bellifemine, Caire, & Greenwood, 2007). A MTS manages all message exchange within and between platforms. By using MTS, JADE agents are able to communicate with agents living on remote platforms regardless of whether these are other JADE platforms or different platforms provided that they are FIPA compliant (Grimshaw, 2010). With the use of MTS agent, communication between agents developed with a different programming language is also possible as long as they are FIPA compliant.

The class ‘agent’ defined in JADE class library represents the base of all the agents designed by the programmer. In other words, an agent is an instance if a user defined Java class that extends the ‘agent’ class mentioned above. Agent messages are the fundamental form of communication between agents and the structure of a message is a set of key values written in FIPA-ACL (Bellifemine, Caire, & Greenwood, 2007). The communication paradigm adopted is the
asynchronous message passing. The communication architecture offers flexible and efficient messaging, where JADE creates and manages a queue of incoming ACL messages, private to each agent. Each agent has a sort of mailbox as the agent message queue, where the JADE runtime posts messages sent by other agents (Bellifemine, Caire, & Greenwood, 2007). Bellifemine states that whenever a message is posted in the message queue, the receiving agent is notified (Bellifemine, Caire, & Greenwood, 2007). However, if and when the agent actually picks up the message from the message queue to process it is completely up to the programmer. Agents can access their queue via a combination of several modes: blocking, polling, timeout and pattern matching based.

![Internal architecture of a generic JADE agent](image)

Figure 4.4: Internal architecture of a generic JADE agent (Bellifemine F., 2002).

A task that an agent can carry out are defined as a ‘behaviour’ in JADE. In order to implement an agent-specific task, the programmer should define one or more ‘behaviour’ subclasses, instantiate
them and add the behaviour objects to the agent task list. A ‘Behaviour’ can be initialized, suspended, and spawned at any time (Bellifemine, Caire, & Greenwood, 2007). ‘Behaviours’ are logical execution threads and they can be composed in different ways to achieve complex execution patterns. It is the programmer call to define when an agent switches from the execution of one behaviour to the execution of another. The internal architecture of a generic JADE agent can be seen from Figure 4.4:

The process monitoring system designed in this research is completely implemented using JADE. The agents created are completely coded in Java programming language to be used in JADE. In addition to JADE functionalities, MATLAB was used to make all the necessary computations and matrix operations for its favorable ease of use with matrices. In order to connect JADE, which is in Java programming language, with MATLAB, an application programming interface (API) called ‘matlabcontrol’ was used. The freeware ‘matlabcontrol’ is a Java API that allows for calling MATLAB from Java. By using ‘matlabcontrol’, it is possible to evaluate functions in MATLAB, as well as getting variables from MATLAB, and setting variables in MATLAB. The interaction can be performed from either inside MATLAB or outside MATLAB. The API can be downloaded from the project website: https://code.google.com/p/matlabcontrol/

4.2. MULTI-AGENT SYSTEM ARCHITECTURE

The design of a data-driven process monitoring system is initiated by data acquisition. The data collected during plant operation is stored and thus is readily available for manipulations. The goal of the fault detection and diagnosis system is to make use of the historical data provided by the
process in order to assess the operational status of the process. For reaching this goal, the first step is to load the data to the process monitoring system. In order to provide a reliable result, the quality of the data provided to the process monitoring system should be high. Nonetheless, for most cases process data are contaminated with noise and outliers for different reasons. Narasimhan and Jordache state some of the common reasons for data contamination as: power supply fluctuations, analog input filtering, changes in ambient temperature, network transmission and signal conversion noise, etc. (Narasimhan & Jordache, 2000). Data preprocessing should be carried out in order to eliminate the contaminations in the data. Thus, data pretreatment is the second important step in the process monitoring system. Data pretreatment is briefly discussed in Section 3.1.1. After the data is loaded into the process monitoring system and it is preprocessed; statistical process monitoring procedure can be launched. The first step of process monitoring procedure is fault detection. Statistical fault detection is achieved by detecting the departure of a process from its normal operating behavior. Fault detection is initiated by the feature extraction step, as the important information is captured from a sample data by projecting the data from the input space to the feature space. In most cases, the feature space is lower in dimension compared to the input space and hence the task of dimensionality reduction is achieved. Due to its lower dimensionality, it is easier for an operator/engineer to visualize the process status in the feature space. Details of feature extraction and pattern classification is provided in Section 3.2. After a fault is detected, the next step is fault identification. Fault identification involves identifying the variables responsible for the fault. It brings the attention of the plant operators and engineers to the subsystem, which is most pertinent to the fault. Fault identification aids in eliminating and handling the abnormal situation in the most convenient manner. After the fault identification phase is successful, the process monitoring system can move on to fault diagnosis. Himmelblau states that fault diagnosis
involves determination of the equipment or the portion of process which is causing the fault (Himmelblau, 1978). For satisfying process performance specifications, fault diagnosis determines the subsystem or the environment that is violating its provided conditions. The fault identification phase is challenging simply because of the interrelation amongst the variables; thus it is difficult to find the actual cause of the fault. The process monitoring system should be expected to provide information about the possible causes of a fault with a specified degree of confidence. The fault diagnosis phase involves pattern classification in order to determine the belongingness of a data point to a possible fault class. Furthermore, for a given fault a priori information of the fault type either can be known or can be an unknown. The learning process of the process monitoring system in the case of priori knowledge being available is called supervised learning. This type of a system is not suitable for diagnosing novel faults; since the system would try to classify a given data point in one of the already trained classes. On the other hand, in the case of supervised learning, a system finds the appropriate classes without the knowledge of the fault class. Fault diagnosis phase is the final step for the process monitoring system.

All the different steps mentioned above should be incorporated into an overall framework for the sake of efficiency and ease of use. This can be done in an efficient manner by designing an overall multi-agent system architecture for process monitoring. The agent-based approach helps for the coordination mechanism for each possible interaction which would have to be designed by hand. By using the agent-based programming approach the decisions being made in the system are automated, consistent and the approach reduces the overall development time. Incorporating several agents for carrying out required tasks brings the advantage of ease of use and genericity. By designing the overall framework using a multi-agent system, the process monitoring system is
designed to be generic. Any new functionality or property can be easily incorporated into the system by adding a new agent or manipulating existing agents. Furthermore, as mentioned above, agent middleware provides many useful built-in functionalities like the agent communication language, which establishes agent communication. In the multi-agent architecture designed for this research project, there are four agents: operator agent, data preprocessing agent, fault detection agent, and fault diagnosis agent. In addition to the mentioned agents, a ‘MatlabConnect’ Java class is used to establish the communication between JADE and MATLAB. The code for all the agents and scripts used in this project is provided in the Appendix. The scheme for the overall framework of the multi-agent system designed in this research project can be seen in Figure 4.5:

Figure 4.5: The scheme for the overall framework of the multi-agent system.
4.2.1. OPERATOR AGENT

As the name suggest, the operator agent is the agent which communicates with the physical world, through communicating with the plant operators and engineers. The agent has a user interface, which prints out all the necessary information provided by the process monitoring system. The operator agent user interface, coded in Java programming language, can be seen from Figure 4.6:

![Operator Agent GUI](image)

Figure 4.6: The user interface for the operator agent.

As can be seen from Figure 4.5, the operator agent handles most of the communication among the agents. This is especially the case; since the operator agent is informed about all the processes going on in the system, in order to report to the plant operator or engineer if desired. The operator agent also handles the communication with the above mentioned ‘MatlabConnect’ Java class in
order to regulate the communication between JADE and MATLAB. Whenever the process monitoring system is run, the user interface for the operator agent pops up as in Figure 4.6. First, the system needs to be trained by selecting one of the functionalities provided by the process monitoring system: Principal Component Analysis (PCA), Fisher Discriminant Analysis (FDA), or Self-Organizing Map (SOM). Whenever a training method is selected on the user interface, the operator agent sends the specific message to the data preprocessing agent for the selected method to initiate the training process. The operator agent prints out a message saying the system is ready for online data manipulation after the training process is completed successfully.

The ‘Generate Data Online’ button, which can be seen in Figure 4.6, starts the on-line data supply for the process monitoring system. This functionality simply simulates a real-life plant environment, where a data point containing as many variables as the process outputs, is fed into the process monitoring system at a desired rate. The online data generator simply copies each data point from a specified data file and feeds it into the system one by one. By introducing this functionality, the system is made generic in a sense that the only input the system needs is a data file from a plant or process in any format. After plant data is readily available, it can be addressed to the system so the system can treat it as on-line. The system also supports real-time on-line process monitoring generically, since on-line data can be stored anywhere and provided to the online data generator loading database with a slight delay.

After the training procedure is completed successfully and on-line data generation is initiated, the system starts receiving data points continuously. At this point, on-line process monitoring can be started by selecting the corresponding functionality, which was selected for the training procedure.
After the corresponding ‘Online Data Manipulation’ button is selected on the user interface, the operator agent, the operator agent sends the specific message to the data preprocessing agent for the selected method to initiate the training process. After on-line process monitoring is initiated, the operator agent prints out necessary information on the user interface such as outlier removal results, fault detection results, and fault diagnosis results.

4.2.2. DATA PREPROCESSING AGENT

The data preprocessing agent is responsible for data pretreatment. Whenever a training procedure is selected using the operator agent user interface, the data preprocessing agent receives a message from the operator agent stating the selected method. Then, the data preprocessing agent starts historical data pretreatment and lets the operator agent know when the process is completed successfully. Furthermore, when an online data manipulation procedure is selected using the operator agent user interface, the data preprocessing agent again receives a message from the operator agent stating the selected method. Then, the data preprocessing agent starts on-line data pretreatment, the operator agent is informed whether the on-line point contains outliers or not. If the on-line point contains outliers, the data preprocessing agent removes the outlier and informs the operator agent.

4.2.3. FAULT DETECTION AGENT

Fault detection agent is firstly responsible for finishing off the historical training procedure initiated by the operator agent, after receiving a message from the data preprocessing agent stating the selected method. Percent of original variance preserved versus number of principal
components plots are created for each data class in order to show the plant operator or engineer, how much variance is preserved while dimensionality reduction. A sample plot can be seen in Figure 4.7:

![Figure 4.7: Sample percent of original variance preserved versus number of principal components plot.](image)

After the historical training procedure is completed successfully, fault detection agent sends a message to the operator agent for it to report to the user. For the on-line data manipulation property, the fault detection agent receives a message from the data preprocessing agent stating the selected method. Then, the data preprocessing agent starts online data manipulation using the selected method and constantly informs the operator agent about the status of the process. Whenever a fault is detected in the on-line data, the fault detection agent sends a message to the fault diagnosis agent to initiate fault diagnosis. In addition, operator agent is also informed by the fault detection agent, stating that a fault has been detected.
The Principal Component Analysis (PCA) functionality of the fault detection agent uses $T^2$ and $Q$ statistics in order to determine abnormal event conditions. Online $T^2$ and $Q$ statistic charts are also created by the fault detection agent for the plant operator or engineer to follow. The $T^2$ and $Q$ statistics statistic values and statistical limits are calculated using the equations mentioned in Section 3.3.1.

Figure 4.8: Screenshot of the user-interface for process monitoring using Principal Component Analysis (PCA).
The process monitoring system alarms for a detected fault when both $T^2$ and Q statistic charts indicate an out-of-control behavior. Whenever a fault is detected the fault detection agent creates a contribution plot for aiding fault identification. Contribution plots are also created using equations presented in Section 3.3.1. After the fault identification step is complete, the fault detection agent sends a message to the fault diagnosis agent to initiate fault diagnosis. A sample user-interface screen for process monitoring using Principal Component Analysis can be seen from Figure 4.8.

Figure 4.9: Sample plot for data projected into component planes created by the process monitoring system.
The Fisher Discriminant Analysis (FDA) functionality of the fault detection agent uses equations presented in Section 3.4.2 in order to calculate the first three FDA vectors for fault detection. In addition, an F.D.A. plot is created by data projected into component planes for the plant operator or engineer to observe process shifts easily. A sample FDA plot can be seen from Figure 4.9, where green dots represent historical data points, and red dots are introduced into the figure whenever a new data point is processed in the system. Whenever a fault is detected, the fault detection agent again sends a message to the fault diagnosis agent to initiate fault diagnosis.

The Self-Organizing Map (SOM) functionality of the fault detection agent starts with creating a quantization error plot using all the historical data for all fault classes. Quantization error plots reveal the nature of each fault presented in a single plot and can aid in a better physical understanding of faults. A sample quantization error plot can be seen from Figure 4.10, where the response curves of all faults can be distinguished one after another. Then, the fault detection agent uses equations presented in Section 3.5.1 in order to calculate the best matching units for fault detection. Firstly, the system trains an SOM using all the historical data for all classes. Then the system calculates required statistics to detect faults and corresponding contributions. Whenever a fault is detected, the fault detection agent again sends a message to the fault diagnosis agent to initiate fault diagnosis.
Figure 4.10: Sample quantization error plot created by the process monitoring system.

4.2.4. FAULT DIAGNOSIS AGENT

Fault diagnosis agent is the agent which finalizes the fault detection and diagnosis procedure carried out by the process monitoring system. As can be seen from Figure 4.5, the fault diagnosis agent does not have a historical training module; since it is not needed. Whenever a fault is detected in the system, the fault diagnosis agent receives a message from the fault detection agent stating to initiate fault diagnosis using a specified method. Then, the fault diagnosis agent calculates discriminant functions for each online data point using equations presented in Sections 3.3.3, 3.4.1, and 3.5.1 considering the desired method. After that, the fault diagnosis agent reports the on-line diagnosis result to the operator agent for it to report to the user.
4.3. CONCLUSIONS

In summary, this Chapter of the thesis started with presenting multi-agent systems and JADE (Java Agent DEvelopment framework) in detail. All the functionalities of JADE used in this research project and the multi-agent process monitoring system architecture designed in this research project are also described. Detailed descriptions of all the agents designed and used are also presented in this Section. The selection of multi-agent based programming for this research project enables easy modifications in every aspect of system from communication of agents to responsibilities and functionalities of each agent. Furthermore, the use of MATLAB scripts in executing various functionalities enables the easy add-on function for any new functionality planned to be incorporated into the system. Using these advantages for easy add-ons for the process monitoring system, new functionalities can be always added on top of the methods currently in use.
5. RESULTS AND DISCUSSION

5.1. TENNESSEE EASTMAN PROCESS APPLICATION

Tennessee Eastman process (TEP) model is a realistic simulation program of a chemical plant which is widely accepted as a benchmark for control and monitoring studies (Yin, Ding, Haghani, Hao, & Zhang, 2012). The code is first published by Downs and Fogel to provide a realistic problem for testing process control technology (Downs & Fogel, 1993). This test process problem is based on an actual industrial process, which is modelled in great detail for publication purposes. The process model has been coded by Downs and Fogel into a set of FORTRAN subroutines which describe the nonlinear relationships in the unit operations and the material and energy balances (Downs & Fogel, 1993). The updated versions of the FORTRAN code of the process is available over internet together with a MATLAB version which was created more recently. The code for Tennessee Eastman Problem for MATLAB can be downloaded from Arizona State University Control System Engineering Laboratory website: http://csel.asu.edu/?q=node/33.

The process flow sheet and controller scheme for the Tennessee Eastman process (TEP) can be seen from Figure 5.1. As can be seen from Figure 4.1, the process has five major units: reactor, condenser, compressor, separator and stripper. The process uses four gaseous reactants (A, C, D, and E) in order to produce two liquid products (G and H). There also exists an unwanted by-product (F), together with an inert (B). The chemical reactions taking place in the process are as follows:
\begin{align*}
A_{(g)} + C_{(g)} + D_{(g)} &\rightarrow G_{(liq)} & \text{Product 1} \\
A_{(g)} + C_{(g)} + E_{(g)} &\rightarrow H_{(liq)} & \text{Product 2} \\
A_{(g)} + E_{(g)} &\rightarrow F_{(liq)} & \text{By-product} \\
3 \; D_{(g)} &\rightarrow 2 \; F_{(liq)} & \text{By-product}
\end{align*}

Figure 5.1: Tennessee Eastman Process schematic together with the control scheme (Ricker, 1996).

The Tennessee Eastman process allows a total of 52 measurements; which consists of 41 process variables and 12 manipulated variables as a total of 53 process variables. The process variables can be seen from Table 5.1 and the manipulated variables can be seen from Table 5.2. The process
simulations of the Tennessee Eastman process introduced by Downs and Fogel contain 20 pre-
programmed faults, which can be seen in Table 5.3 (Downs & Fogel, 1993). The process monitoring
system for fault detection and diagnosis should be designed solely based on the process data, as
no priori information or knowledge about the mathematical model of Tennessee Eastman process
(TEP) is available. The data sets used in this research are created using the MATLAB simulation
provided by Ricker by using Simulink simulations (Ricker, 1996). The Simulink code provided by
the Ricker simulates the plant’s closed-loop behavior. Using the Simulink simulator, the operation
modes, measurement noise, sampling time and magnitudes of the faults can be easily modified and
data sets can be created with the desired properties. The training data sets are created for 48 hours
of plant operation. The process monitoring system is tested by using different data sets, created to
contain 96 hours of plant operation. 48 hours of plant operation corresponds to 480 data points and
96 hours of plant operation corresponds to 960 data points. After the process monitoring system is
trained using the training data sets, the test data sets are introduced as on-line data input, simulating
a real-life plant environment for the process monitoring system. An important point to note is that
the data sets used for training the multi-agent system are different from the data sets that the system
is trained in the online-process monitoring module. The on-line data input is fed to the system as
1 data point per second for simulating the real-time performance of the designed process
monitoring system. By using the same idea, any type of data file can be loaded into the multi-agent
system as if it was being fed to the process monitoring system as on-line plant data. For the first
set of runs, the training set includes the faults 1, 2, 4, 8, 11, 13, and 14 together with the normal
operating region.
Table 5.1: Description of Process Variables

<table>
<thead>
<tr>
<th>Block Name</th>
<th>Variable Number</th>
<th>Variable Name</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Input Feed</strong></td>
<td>XMEAS (1)</td>
<td>A feed (Stream 1)</td>
</tr>
<tr>
<td></td>
<td>XMEAS (2)</td>
<td>D feed (Stream 2)</td>
</tr>
<tr>
<td></td>
<td>XMEAS (3)</td>
<td>E feed (Stream 3)</td>
</tr>
<tr>
<td></td>
<td>XMEAS (4)</td>
<td>A and C feed (Stream 4)</td>
</tr>
<tr>
<td><strong>Recycle</strong></td>
<td>XMEAS (5)</td>
<td>Recycle flow (Stream 8)</td>
</tr>
<tr>
<td><strong>Reactor</strong></td>
<td>XMEAS (6)</td>
<td>Reactor feed rate (Stream 6)</td>
</tr>
<tr>
<td></td>
<td>XMEAS (7)</td>
<td>Reactor pressure</td>
</tr>
<tr>
<td></td>
<td>XMEAS (8)</td>
<td>Reactor level</td>
</tr>
<tr>
<td></td>
<td>XMEAS (9)</td>
<td>Reactor temperature</td>
</tr>
<tr>
<td><strong>Purge</strong></td>
<td>XMEAS (10)</td>
<td>Purge rate (Stream 9)</td>
</tr>
<tr>
<td><strong>Separator</strong></td>
<td>XMEAS (11)</td>
<td>Separator temperature</td>
</tr>
<tr>
<td></td>
<td>XMEAS (12)</td>
<td>Separator level</td>
</tr>
<tr>
<td></td>
<td>XMEAS (13)</td>
<td>Separator pressure</td>
</tr>
<tr>
<td></td>
<td>XMEAS (14)</td>
<td>Separator underflow (Stream 10)</td>
</tr>
<tr>
<td><strong>Stripper</strong></td>
<td>XMEAS (15)</td>
<td>Stripper level</td>
</tr>
<tr>
<td></td>
<td>XMEAS (16)</td>
<td>Stripper pressure</td>
</tr>
<tr>
<td></td>
<td>XMEAS (17)</td>
<td>Stripper underflow (Stream 11)</td>
</tr>
<tr>
<td></td>
<td>XMEAS (18)</td>
<td>Stripper temperature</td>
</tr>
<tr>
<td></td>
<td>XMEAS (19)</td>
<td>Stripper steam flow</td>
</tr>
<tr>
<td><strong>Miscellaneous</strong></td>
<td>XMEAS (20)</td>
<td>Compressor work</td>
</tr>
<tr>
<td></td>
<td>XMEAS (21)</td>
<td>Reactor water temperature</td>
</tr>
<tr>
<td></td>
<td>XMEAS (22)</td>
<td>Separator water temperature</td>
</tr>
<tr>
<td><strong>Reactor Feed Analysis</strong></td>
<td>XMEAS (23)</td>
<td>Component A</td>
</tr>
<tr>
<td></td>
<td>XMEAS (24)</td>
<td>Component B</td>
</tr>
<tr>
<td></td>
<td>XMEAS (25)</td>
<td>Component C</td>
</tr>
<tr>
<td></td>
<td>XMEAS (26)</td>
<td>Component D</td>
</tr>
<tr>
<td></td>
<td>XMEAS (27)</td>
<td>Component E</td>
</tr>
<tr>
<td></td>
<td>XMEAS (28)</td>
<td>Component F</td>
</tr>
<tr>
<td><strong>Purge Gas Analysis</strong></td>
<td>XMEAS (29)</td>
<td>Component A</td>
</tr>
<tr>
<td></td>
<td>XMEAS (30)</td>
<td>Component B</td>
</tr>
<tr>
<td></td>
<td>XMEAS (31)</td>
<td>Component C</td>
</tr>
<tr>
<td></td>
<td>XMEAS (32)</td>
<td>Component D</td>
</tr>
<tr>
<td></td>
<td>XMEAS (33)</td>
<td>Component E</td>
</tr>
<tr>
<td></td>
<td>XMEAS (34)</td>
<td>Component F</td>
</tr>
<tr>
<td></td>
<td>XMEAS (35)</td>
<td>Component G</td>
</tr>
<tr>
<td></td>
<td>XMEAS (36)</td>
<td>Component H</td>
</tr>
<tr>
<td><strong>Product Analysis</strong></td>
<td>XMEAS (37)</td>
<td>Component D</td>
</tr>
<tr>
<td></td>
<td>XMEAS (38)</td>
<td>Component E</td>
</tr>
<tr>
<td></td>
<td>XMEAS (39)</td>
<td>Component F</td>
</tr>
<tr>
<td></td>
<td>XMEAS (40)</td>
<td>Component G</td>
</tr>
<tr>
<td></td>
<td>XMEAS (41)</td>
<td>Component H</td>
</tr>
</tbody>
</table>
### Table 5.2: Description of Process Manipulated Variables

<table>
<thead>
<tr>
<th>Variable Number</th>
<th>Variable Name</th>
<th>Base Value (%)</th>
<th>Unit</th>
</tr>
</thead>
<tbody>
<tr>
<td>XMV (1)</td>
<td>D feed flow</td>
<td>63.053</td>
<td>kg/h</td>
</tr>
<tr>
<td>XMV (2)</td>
<td>E feed flow</td>
<td>53.980</td>
<td>kg/h</td>
</tr>
<tr>
<td>XMV (3)</td>
<td>A feed flow</td>
<td>24.644</td>
<td>kscmh</td>
</tr>
<tr>
<td>XMV (4)</td>
<td>A and C feed flow</td>
<td>61.302</td>
<td>kscmh</td>
</tr>
<tr>
<td>XMV (5)</td>
<td>Compressor recycle valve</td>
<td>22.210</td>
<td>%</td>
</tr>
<tr>
<td>XMV (6)</td>
<td>Purge valve</td>
<td>40.064</td>
<td>%</td>
</tr>
<tr>
<td>XMV (7)</td>
<td>Separator pot liquid flow</td>
<td>38.100</td>
<td>m³/h</td>
</tr>
<tr>
<td>XMV (8)</td>
<td>Stripper liquid product flow</td>
<td>46.534</td>
<td>m³/h</td>
</tr>
<tr>
<td>XMV (9)</td>
<td>Stripper steam valve</td>
<td>47.446</td>
<td>%</td>
</tr>
<tr>
<td>XMV (10)</td>
<td>Reactor cooling water flow</td>
<td>41.106</td>
<td>m³/h</td>
</tr>
<tr>
<td>XMV (11)</td>
<td>Condenser cooling water flow</td>
<td>18.114</td>
<td>m³/h</td>
</tr>
<tr>
<td>XMV (12)</td>
<td>Agitator speed</td>
<td>50.000</td>
<td>rpm</td>
</tr>
</tbody>
</table>

### Table 5.3: Description of Process Disturbances Introduced as Faults

<table>
<thead>
<tr>
<th>Fault Number</th>
<th>Process Variable</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fault 1</td>
<td>A/C feed ratio, B composition constant</td>
<td>Step</td>
</tr>
<tr>
<td>Fault 2</td>
<td>B composition, A/C ratio constant</td>
<td>Step</td>
</tr>
<tr>
<td>Fault 3</td>
<td>D feed temperature</td>
<td>Step</td>
</tr>
<tr>
<td>Fault 4</td>
<td>Reactor cooling water inlet temperature</td>
<td>Step</td>
</tr>
<tr>
<td>Fault 5</td>
<td>Condenser cooling water inlet temperature</td>
<td>Step</td>
</tr>
<tr>
<td>Fault 6</td>
<td>A feed loss</td>
<td>Step</td>
</tr>
<tr>
<td>Fault 7</td>
<td>C header pressure loss – reduced availability</td>
<td>Step</td>
</tr>
<tr>
<td>Fault 8</td>
<td>A,B,C feed composition</td>
<td>Random Variation</td>
</tr>
<tr>
<td>Fault 9</td>
<td>D feed temperature</td>
<td>Random Variation</td>
</tr>
<tr>
<td>Fault 10</td>
<td>C feed temperature</td>
<td>Random Variation</td>
</tr>
<tr>
<td>Fault 11</td>
<td>Reactor cooling water inlet temperature</td>
<td>Random Variation</td>
</tr>
<tr>
<td>Fault 12</td>
<td>Condenser cooling water inlet temperature</td>
<td>Random Variation</td>
</tr>
<tr>
<td>Fault 13</td>
<td>Reaction kinetics</td>
<td>Slow Drift</td>
</tr>
<tr>
<td>Fault 14</td>
<td>Reactor cooling water valve</td>
<td>Sticking</td>
</tr>
<tr>
<td>Fault 15</td>
<td>Condenser cooling water valve</td>
<td>Sticking</td>
</tr>
<tr>
<td>Fault 16</td>
<td>Unknown</td>
<td>Unknown</td>
</tr>
<tr>
<td>Fault 17</td>
<td>Unknown</td>
<td>Unknown</td>
</tr>
<tr>
<td>Fault 18</td>
<td>Unknown</td>
<td>Unknown</td>
</tr>
<tr>
<td>Fault 19</td>
<td>Unknown</td>
<td>Unknown</td>
</tr>
<tr>
<td>Fault 20</td>
<td>Unknown</td>
<td>Unknown</td>
</tr>
</tbody>
</table>
5.2. RESULTS

The process monitoring system designed in this research project is trained and tested with on-line data as explained in Section 5.1. After each test run, two important metric is calculated: fault detection rate and fault diagnosis rate. Fault detection rate is calculated as can be seen in Equation 5.1 as the number of detected points in the test data set over the total number of points in the test data set. Similarly, fault diagnosis rate is calculated as can be seen in Equation 5.2 as the number of correctly diagnosed points in the test data set over the total number of points in the test data set.

\[
Fault\ Detection\ Rate_{i} = \frac{N_{\text{detected}}}{N_{\text{total}}} \quad \text{Eq. 5.1}
\]

\[
Fault\ Diagnosis\ Rate_{i} = \frac{N_{\text{correct}}}{N_{\text{total}}} \quad \text{Eq. 5.2}
\]

Using the calculated fault detection rates and fault diagnosis rates, Principal Component Analysis (PCA), Fisher Discriminant Analysis (FDA), and Self-Organizing Maps (SOM) results from the process monitoring system are compared. The results of all the test runs for fault detection rates for PCA, FDA, and SOM are presented in Table 5.4. In addition, the results of all the test runs for fault diagnosis rates for PCA, FDA, and SOM are presented in Table 5.5.
Table 5.4: Fault Detection Rate Results for PCA, FDA and SOM on TEP

<table>
<thead>
<tr>
<th>TEP Fault</th>
<th>PCA Fault Detection Rate (%)</th>
<th>FDA Fault Detection Rate (%)</th>
<th>SOM Fault Detection Rate (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fault 1</td>
<td>99.79</td>
<td>99.79</td>
<td>99.48</td>
</tr>
<tr>
<td>Fault 2</td>
<td>96.67</td>
<td>99.79</td>
<td>96.36</td>
</tr>
<tr>
<td>Fault 4</td>
<td>29.63</td>
<td>99.90</td>
<td>99.69</td>
</tr>
<tr>
<td>Fault 8</td>
<td>99.38</td>
<td>99.90</td>
<td>98.65</td>
</tr>
<tr>
<td>Fault 11</td>
<td>79.21</td>
<td>99.90</td>
<td>98.13</td>
</tr>
<tr>
<td>Fault 13</td>
<td>99.48</td>
<td>99.90</td>
<td>98.44</td>
</tr>
<tr>
<td>Fault 14</td>
<td>29.52</td>
<td>99.90</td>
<td>99.79</td>
</tr>
</tbody>
</table>

Table 5.5: Fault Diagnosis Rate Results for PCA, FDA and SOM on TEP

<table>
<thead>
<tr>
<th>TEP Fault</th>
<th>PCA Fault Diagnosis Rate (%)</th>
<th>FDA Fault Diagnosis Rate (%)</th>
<th>SOM Fault Diagnosis Rate (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fault 1</td>
<td>89.19</td>
<td>97.82</td>
<td>99.06</td>
</tr>
<tr>
<td>Fault 2</td>
<td>0.00</td>
<td>81.60</td>
<td>96.36</td>
</tr>
<tr>
<td>Fault 4</td>
<td>0.00</td>
<td>79.21</td>
<td>95.84</td>
</tr>
<tr>
<td>Fault 8</td>
<td>67.05</td>
<td>94.60</td>
<td>91.89</td>
</tr>
<tr>
<td>Fault 11</td>
<td>28.86</td>
<td>70.79</td>
<td>95.11</td>
</tr>
<tr>
<td>Fault 13</td>
<td>99.38</td>
<td>40.02</td>
<td>94.60</td>
</tr>
<tr>
<td>Fault 14</td>
<td>2.39</td>
<td>1.46</td>
<td>9979</td>
</tr>
</tbody>
</table>
As can be seen from Table 5.4, fault detection rate for **Fault 1** is very high for all methods, which indicates successful fault detection. This is an expected result; since Fault 1 is a step change in \(A/C\) feed ratio in Stream 4; as can be seen from Table 5.3. Fault 1 results in an increase in the \(C\) feed and a decrease in the \(A\) feed in Stream 4. This results in a decrease in the \(A\) feed in the recycle Stream 5 and a control loop reacts to increase the \(A\) feed in Stream 1. These two effects counteract each other over time, which results in a constant \(A\) feed composition in Stream 6 after sufficient time. The controller action can be seen from the \(T^2\) and \(Q\)-statistic charts shown in Figure 5.2, similar to a controller response curve. The red dashed lines in Figure 4.2 part (a) and (b) represent \(T^2\) and \(Q\)-statistic threshold limits. Thus, nearly all the points are above the detection limits for both \(T^2\) and \(Q\)-statistic charts, which explains the high fault detection rate. Fault detection rates for FDA and SOM are also very close to unity and thus all methods can be considered to successfully detect Fault 1. This is an expected result since more than half of the variables monitored deviate significantly from their normal operating behavior.

![Figure 5.2](image)

**Figure 5.2:** \(T^2\)-statistic chart (a), and \(Q\)-statistic chart (b) for Fault 1 test run.
As can be seen from Table 5.5, fault diagnosis rate for Fault 1 is high for all methods, except being relatively low for PCA. PCA test run results suggest that 9.88% classification rate for Fault 8. This can be explained by analyzing the FDA chart for Fault 1 seen in Figure 5.3. As can be seen from Figure 5.3, the green dots represent historical training data, and the red dots represent on-line data processed in the system. The bigger and more concentrated region in the bottom represents the Normal Operating Region (NOR), together with some faults expanding the NOR. The cluster seen on the upper right hand side represents Fault 1, and as can be seen from Table 5.5, FDA successfully discriminates and diagnoses Fault 1; the same applies for SOM.

![Figure 5.3: Data Projected into component planes: a) green training data b) red Fault 1 test run.](image)

The misclassification rate of 9.88% is caused by the beginning of the shift of the operating region. As the process starts shifting from the NOR, until the new region is far enough, PCA classifies the
faulty points as Fault 8. Even with a relatively low fault diagnosis rate for PCA, the fault diagnosis rates for FDA and SOM are also very close to unity and thus all methods can be considered to successfully diagnose Fault 1.

Fault 2 is a step change in B composition in Stream 4; as can be seen from Table 5.3. As can be seen from Table 5.4, fault detection rate for Fault 2 is again high for all methods, which indicates successful fault detection. Fault 2 has the lowest fault detection rate among all the faults for SOM, whose value is close to the fault detection rate for PCA. Fault diagnosis results suggest that PCA cannot diagnose Fault 2; thus it should not be credited for diagnosis results on Fault 2. Fault diagnosis rate for FDA is relatively high at 81.60 %, where 16.94 % is misclassified as Fault 8. Like Fault 1, Fault 2 has an operating region shift as can be seen from Figure 5.4.

Figure 5.4: Data Projected into component planes: a) green training data b) red Fault 2 test run.
In the beginning of the test run, as the process shifts from the Normal Operating Region (NOR), the system misclassifies some points as Fault 8. As the system proceeds towards the new operating regime, misclassification rate decreases and the process monitoring system starts to diagnose the incoming on-line data points as Fault 2. The fault diagnosis rates for SOM are very successful with a value close to unity. Overall, Fault 2 can be successfully diagnosed mainly using only SOM.

Fault 4 is a step change in reactor cooling water inlet temperature; as can be seen from Table 5.3. As can be seen from Table 5.4, fault detection rate for Fault 4 is again high for FDA and SOM, but very low for PCA. A significant effect of Fault 4 is to induce a step change in the reactor cooling water flow rate. The sudden change in temperature increase in the reactor is compensated by the control loops. All the other measurement and manipulated variables do not vary significantly. This behavior can be seen from an average contribution plot created by PCA functionality of the designed process monitoring system shown in Figure 5.5. As can be seen from Figure 5.5, The significant contributing variables are seen as variables 2, 3, and 52, which are ‘D feed’, ‘E feed’, and ‘condenser cooling water flow’ respectively as can be seen from Table 5.1 and Table 5.2. The change in ‘condenser cooling water flow’ is directly caused by the fault and the most expected variable to contribute as it is the case. The changes in ‘D feed’, ‘E feed’ are mainly caused by the control action of the present control loop in the process. The contribution plots created by PCA can be misleading for some cases; since they are produced for a single data point and system behavior is dynamic. Contribution plots created using SOM outperform PCA, and thus should be included in the process monitoring system in the future versions (Robertson, 2013). The $T^2$-statistic and $Q$-statistic charts can be seen from Figure 5.6. Fault detection results for FDA and
SOM are both very close to unity, which resembles very successful fault detection using these techniques.

Figure 5.5: Average contribution plot for Fault 4 test run.

Figure 5.6: $T^2$-statistic chart (a), and Q-statistic chart (b) for Fault 4 test run.
Fault diagnosis results for PCA for Fault 4 are also very poor as can be seen from Table 5.5. PCA fails to diagnose Fault 2 classifying none of the points as Fault 2, and misclassifying 28.17 % of the points as Fault 13. FDA works relatively better for classifying Fault 2, by classifying 79.21 % of the test data for Fault 4, and 20.58 % for Fault 11. The FDA chart can be seen in detail in Figure 5.7:

![Data Projected into component planes for Fault 2 test run for: (a) the whole region, (b) zoomed towards NOR, and (c) zoomed more towards faulty points.](image)

As can be seen from Figure 5.7, Fault 4 slightly expands the Normal Operating Region (NOR), however staying very close. Part (c) of Figure 4.6 makes it clear that Fault 4 is hard to distinguish from NOR and also other faults which are represented in the area close to and surrounding NOR.
SOM is the only technique diagnosing Fault 4 very successfully by only misclassifying 3.85 % of the points as Fault 11.

**Fault 8** is a random variation in the A, B, C feed composition; as can be seen from Table 5.3. As can be seen from Table 5.4, fault detection rate for Fault 8 is very close to unity for PCA, FDA and SOM, which suggests successful fault detection using all three different methods. The FDA chart for Fault 8 can be seen in detail in Figure 5.8. As can be seen from Figure 5.8, Fault 8 expands the Normal Operating Region (NOR) considerably, together with having some points intersecting with the NOR. This explains the lower than expected fault diagnosis rates provided by PCA and SOM; since both methods misclassify some points as Fault 13. PCA, FDA and SOM has a misclassification rate of 32.33 %, 3.33 %, and 6.45 % respectively; where points misclassified for Fault 13. FDA does the best job in classifying Fault 8 as can be seen from Table 5.5, whereas it is the most problematic fault to diagnose for SOM with a fault diagnosis rate of 91.89 % as can be seen from Table 5.5.

![Data Projected into component planes: a) green training data b) red Fault 8 test run.](image)

Figure 5.8: Data Projected into component planes: a) green training data b) red Fault 8 test run.
**Fault 11** imposes a random variation in the temperature of the cooling water to the reactor as can be seen from Table 5.3. This causes large oscillations in the reactor cooling water flow rate, which results in a fluctuation of reactor temperature. This is caused by the control system trying to constantly adjust the cooling water flow to compensate for the reduced or increased cooling capacity of the water used. Nearly all the other variables remain around their set-points and behave similarly as in the normal operating conditions. This can be seen from Figure 5.9. An important point to note here is the resemblance of the contribution plots for Fault 4 and Fault 11 seen in Figures 5.5 and 5.9 respectively. Since both Fault 4 and Fault 11 affect the same process variable, this is an expected result.

![Figure 5.9: Average contribution plot for Fault 11 test run.](image)

As can be seen from Table 5.4, fault detection rate for Fault 11 is high for FDA and SOM, but relatively low for PCA. The low fault detection behavior is mainly caused again by the choice to
alert a fault alarm when both $T^2$-statistic and $Q$-statistic alert for a fault, which can be seen in Figure 5.10. As can be seen from Figure 5.10, although the $T^2$-statistic has considerably more faulty points beyond the threshold, the $Q$-statistic behaves more conservatively resulting in a lower overall fault detection rate. Furthermore, both FDA and SOM are very successful in detecting Fault 11.

![Figure 5.10: $T^2$-statistic chart (a), and $Q$-statistic chart (b) for Fault 11 test run.](image)

Fault diagnosis result are very low for PCA for Fault 11 as can be seen from Table 5.5, for which some portion is caused by the low fault detection rate. In addition, a significant amount, 50.73 %, of the points are classified as Fault 13 by PCA. Furthermore FDA misclassifies 13.10 % of points for Fault 13, 10.81 % of points for Fault 8, and 4.99 % of points for Fault 4, while classifying 70.79 % of points correctly as Fault 11. The high confusion rate can be understood by analyzing the FDA plot for Fault 11 provided in Figure 5.11:
As can be seen from Figure 5.11, Fault 11 expands the Normal Operating Region (NOR) in a specific direction with a close intersection with the NOR and other close-by faulty points. This causes the high confusion rate in PCA and FDA. SOM also misclassifies an insignificant 1.87% of the points for Fault 14. Even though Fault 4 and Fault 11 affect the same process variable, fault detection and diagnosis rates differ. This is mostly the situation for PCA, however FDA also differs for fault diagnosis rates. SOM detects and classifies Fault 4 and Fault 11 very closely; however the faults misclassified also differ for the SOM case. This adds to the discussion that no single method works best for all fault scenarios; results can even change when the same process variable is affected by different faults.

**Fault 13** is a slow drift in the kinetics of the reaction taking place in the reactor as can be seen from Table 5.3. Fault 13 requires many adjustments by the control system to accommodate for the changing composition of the reactor output. As can be seen from Table 5.4, fault detection rate for Fault 8 is very close to unity for PCA, FDA and SOM, which suggests successful fault detection using all three different methods. The FDA chart for Fault 8 can be seen in detail in Figure 5.12.
As can be seen from Figure 5.12, Fault 13 expands the Normal Operating Region (NOR) while still intersecting with the NOR. As mentioned above, Fault 13 is the most common fault to be confused with other faults. Faults 2, 4, 8, 11, and 14 is misclassified by PCA to an extent as being Fault 13; simply because of being in very close proximity of the NOR. FDA misclassified some points belonging to the test data for Faults 8 and 11, and SOM misclassified some points belonging to the test data for Fault 8 as Fault 13.

Fault diagnosis results for Fault 13 using PCA is very close to unity as can be seen from Table 5.5, which resembles successful fault diagnosis. This also explains how Faults 2, 4, 8, 11, and 14 can be misclassified as Fault 13; simply because PCA seems to classify a broad range of points close to the NOR as Fault 13. FDA is unsuccessful in classifying Fault 13 with a misclassification rate.
of 54.68% of the test points as Fault 8, and 4.78% of the test points as Fault 11. SOM is again successful in diagnosing Fault 13 with only a small misclassification rate of 3.33% for Fault 8.

Fault 14 is a sticky valve in the reactor cooling water as can be seen from Table 5.3. Its effect on the rest of the process is similar to the earlier random variation faults where it creates variations that would not be observed under normal operation. As can be seen from Table 5.4, fault detection rate for Fault 14 is very high for FDA and SOM, but very low for PCA. The low fault detection behavior is mainly caused again by the choice to alert a fault alarm when both T^2-statistic and Q-statistic alert for a fault, which can be seen in Figure 5.13. As can be seen from Figure 5.13, although the Q-statistic has considerably more faulty points beyond the threshold, the T^2-statistic behaves more conservatively resulting in a lower overall fault detection rate. This is the opposite case for Fault 11, where the T^2-statistic alerted more faults than the Q-statistic. Furthermore, both FDA and SOM produce very successful results in detecting Fault 11.

Figure 5.13: T^2-statistic chart (a), and Q-statistic chart (b) for Fault 14 test run.
Fault diagnosis result are very low for PCA for Fault 14 as can be seen from Table 5.5, for which a significant portion is caused by the low fault detection rate. In addition, a significant amount, 17.98 %, of the points are classified as Fault 13 by PCA, and 9.04 % of the points are classified as Fault 11. Furthermore, FDA is also very unsuccessful in diagnosing Fault 14 with a fault diagnosis rate of 1.46 %. In addition, FDA misclassifies 58.00 % of points for Fault 8, 32.85 % of points for Fault 11, and 6.24 % of points for Fault 13. The high confusion rate can again be understood by analyzing the FDA plot for Fault 14 provided in Figure 5.14. As can be seen from Figure 5.14, Fault 14 expands the Normal Operating Region (NOR) slightly while intersecting with the NOR considerably. Visually it is really challenging to differentiate the Fault 14 class without zooming further into the NOR in the plot. Finally SOM has a fault diagnosis rate very close to unity for Fault 14, which resembles successful diagnosis.

Figure 5.14: Data Projected into component planes for Fault 14 test run for: (a) the whole region; (b) zoomed towards NOR and faulty points.
5.3. CONCLUSIONS

In this Chapter of the thesis, we have started by giving a description of the Tennessee Eastman Process application. The designed multi-agent system for process monitoring and supervision is tested on the application and using various different fault scenarios. The results of the runs are presented also in this section and the results are discussed in many aspects. The results in general confirmed the expected case for no single process monitoring method being absolutely superior to another in all possible given conditions. Certain methods proved to work better in fault detection and diagnosis compared to other methods for a given process and a given fault related to that process. For the Tennessee Eastman application test case and for the set of faults considered in this research project, namely Fault 1, 2, 4, 8, 11, 13, and 14, the dominant method in terms of successful fault detection rates turned out to be Fisher Discriminant Analysis (FDA). In addition to that, comparing the methods used in terms of fault diagnosis reveals that the most successful technique as Self-Organizing Maps (SOM).
6. CONCLUSIONS & RECOMMENDATIONS

The advancements in computational technology leads to innovations and improvements in countless different fields. Control systems and instrumentation has benefited from these improvements considerably and continue to improve everyday by increasing the computational ability of processors to enhance process control, optimization, and process monitoring. In the field of process monitoring, on top of the information provided by the relatively well established model based process monitoring techniques, extensive amount of process data from all over the plant can be made use of more efficiently nowadays. In order to make use of this extensive data, data-driven process monitoring techniques can be exploited to get the most information from the data available about the system of interest. Therefore, the objective of this research was to layout the basis for designing and implementing a multi-agent system for process monitoring and supervision, which solely used process data. The architecture implemented various different techniques developed in history in order to make them useful in an overall process monitoring framework. The process monitoring system developed in this research project has three different functionalities ready for use for any process monitoring application; provided that the data sets for normal operating region and faults are available. The three different methods used for fault detection and diagnosis in the process monitoring system are Principal Component Analysis (PCA), Fisher Discriminant Analysis (FDA), and Self-Organizing Maps (SOM).

The process monitoring system designed in this research project is successfully tested with the Tennessee Eastman Process application explained in detail in Section 5.1. Fault detection rates and fault diagnosis rates are compared amongst PCA, FDA, and SOM for different faults using the
results produced by the process monitoring system. The comparison was done considering the fact
that no single process monitoring method is absolutely superior to another in all possible given
conditions. Certain methods work better in fault detection compared to other methods for a given
process and a given fault related to that process. In addition to that, some other methods might
work better in fault diagnosis for a given process and a given fault related to that process. This was
also the case observed in this research project with some exceptions. For the set of faults
considered in this research project, namely Fault 1, 2, 4, 8, 11, 13, and 14, the dominant method
in terms of successful fault detection rates turned out to be Fisher Discriminant Analysis (FDA) as
can be seen from Section 5.2. Self-Organizing Maps (SOM) also performed very close to FDA in
terms of successful fault detection rates with a slight decline in performance. Principal Component
Analysis performed relatively worse than FDA and SOM in detecting faults, especially for Faults
4, 11, and 14. Since Faults 11, 13, and 14 are dynamic faults, less successful results for both fault
detection and fault diagnosis were expected in certain methods. Comparing the methods used in
terms of fault diagnosis reveals that the most successful technique as Self-Organizing Maps (SOM)
in all of the faults except Fault 8. Fault 8 is the only fault FDA was able to outperform SOM in
fault diagnosis. SOM being the most successful fault diagnosing method is somewhat expected in
the set of faults selected for this research project; since there are many dynamic faults in the data
set. The main reason for this success is the aforementioned non-linear nature of Self-Organizing
Maps (SOM). Furthermore, as can be seen from Section 5.2, dynamic faults came out to be
problematic to diagnose successfully for FDA, whereas FDA always outperformed PCA except
Fault 13. PCA was not very successful in overall fault diagnosis compared to FDA and SOM
except for Fault 13. Fault 13 is very successfully diagnosed by PCA, even better than both FDA
and SOM. All these results confirm the fact that certain methods work better in fault detection and diagnosis compared to other methods for a given process and a given fault related to that process.

This research project sets up the foundation for a very promising generic process monitoring system designed using multi-agent systems. The selection of multi-agent based programming enables easy modifications in every aspect of system from communication of agents to responsibilities and functionalities of each agent. Furthermore, the use of MATLAB scripts in executing various functionalities enables the easy add-on function for any new functionality planned to be incorporated into the system. Using these advantages for easy add-ons for the process monitoring system, new functionalities can be always added on top of the methods currently in use. Some new methods to consider for enhancing the functionality of the fault detection and diagnosis aspects can be: Partial Least Squares (PLS), Independent Component Analysis (ICA), Artificial Neural Networks (ANN), and different hybrid methods. In addition, it is also possible to evolve the current process monitoring system into an intelligent system. This can be done by implementing an adaptability function to the system by making the system choose which functionality is performing best for the current mode of operation on the current system. This adaptability function can be based on historical data gathered by the process monitoring system judging the performance of the system in terms of correct fault detection rates and fault diagnosis rates. Using this data stored in the system, the system will know which method works better for a certain fault after consulting the plant operator or engineer. Using this adaptability function properly, the drawback for no ultimate superior functionality for every fault and every process can be overcome in a degree. Thus the overall strength and automation of the process monitoring system will be increased drastically.
REFERENCES


APPENDIX: AGENT JAVA CODES

A.1. OPERATOR AGENT

package Onur;
import jade.core.Agent;
import jade.core.AID;
import jade.core.behaviours.*;
import jade.lang.acl.ACLMessage;
import jade.lang.acl.MessageTemplate;
@SuppressWarnings("serial")
public class OperatorAgent extends Agent {

private OperatorGUI myGui; // The GUI
public final static String HistoricalDT = "HistoricalDT";
public final static String HistoricalDT1 = "HistoricalDT1";
public final static String HistoricalDT2 = "HistoricalDT2";
public final static String OnlineDM = "OnlineDM";
public final static String OnlineDM1 = "OnlineDM1";
public final static String OnlineDM2 = "OnlineDM2";
public final static String OnlineDG = "OnlineDG";
public final static String FaultDetected = "FaultDetected";
public final static String NewCluster = "NewCluster";

protected void setup()
{
    // Set up the GUI
    myGui = new OperatorGUI();
    myGui.setAgent(this);
    myGui.setVisible(true);
    System.out.println("LSU PSE Agent Research Project.");
    System.out.println("Agent \"" + getLocalName() + \"\"] + " has been initiated...
\n");
    addBehaviour(new ReceiveStatusOfModules());
}

OnlineDataManipulation odm = new OnlineDataManipulation(this,3000);
OnlineDataGeneration odg = new OnlineDataGeneration(this,3000);
OnlineDataManipulation1 odm1 = new OnlineDataManipulation1(this,3000);
OnlineDataManipulation2 odm2 = new OnlineDataManipulation2(this,3000);

public void behaSend1() {
    addBehaviour( new HistoricalDataTraining() );
}
public void behaSend2() {
    addBehaviour( odm );
}
public void behaSend3() {
    removeBehaviour( odm );
}
public void behaSend4() {
    addBehaviour( odg );
}
public void behaSend5() {
    removeBehaviour( odg );
}
public void behaSend6() {
    addBehaviour( new HistoricalDataTraining1() );
}
public void behaSend7() {
    addBehaviour( odm1 );
}
public void behaSend8() {
    removeBehaviour( odm1 );
}
public void behaSend9() {
    addBehaviour( new HistoricalDataTraining2() );
}
public void behaSend10() {
    addBehaviour( odm2 );
}
public void behaSend11() {
    removeBehaviour( odm2 );
}

public class HistoricalDataTraining extends OneShotBehaviour {
    public void action() {
        ACLMessage msg = new ACLMessage(ACLMessage.INFORM);
        msg.addReceiver(new AID("PreprocessingAgent", AID.ISLOCALNAME));
        msg.setContent(HistoricalDT);
    }
}
myAgent.send(msg);
System.out.println("P.C.A. Data Training Message sent to Preprocessing Agent.");
}

public class HistoricalDataTraining1 extends OneShotBehaviour {
    public void action() {
        ACLMessage msg = new ACLMessage(ACLMessage.INFORM);
        msg.addReceiver(new AID("PreprocessingAgent", AID.ISLOCALNAME));
        msg.setContent(HistoricalDT1);
        myAgent.send(msg);
        System.out.println("F.D.A. Data Training Message sent to Preprocessing Agent.");
    }
}

public class HistoricalDataTraining2 extends OneShotBehaviour {
    public void action() {
        ACLMessage msg = new ACLMessage(ACLMessage.INFORM);
        msg.addReceiver(new AID("PreprocessingAgent", AID.ISLOCALNAME));
        msg.setContent(HistoricalDT2);
        myAgent.send(msg);
        System.out.println("S.O.M. Data Training Message sent to Preprocessing Agent.");
    }
}

public class OnlineDataManipulation extends TickerBehaviour{
    private OnlineDataManipulation(Agent a, long period) {
        super(a, period);
    }
    public void onStart(){
        System.out.println("P.C.A. Online Fault Detection has been initialised...");
    }
    public void onTick() {
        ACLMessage msg = new ACLMessage(ACLMessage.INFORM);
        msg.addReceiver(new AID("PreprocessingAgent", AID.ISLOCALNAME));
        msg.setContent(OnlineDM);
        myAgent.send(msg);
    }
}

public class OnlineDataGeneration extends TickerBehaviour{
    private OnlineDataGeneration(Agent a, long period) {
        super(a, period);
    }
    public void onStart(){
        System.out.println("Online Data Generation has been initialised...");
    }
}
public void onTick() {
    ACLMessage msg = new ACLMessage(ACLMessage.INFORM);
    msg.addReceiver(new AID("PreprocessingAgent", AID.ISLOCALNAME));
    msg.setContent(OnlineDG);
    myAgent.send(msg);
    }

public class OnlineDataManipulation1 extends TickerBehaviour{
    private OnlineDataManipulation1(Agent a, long period) {
        super(a, period);
    }
    public void onStart(){
        System.out.println("F.D.A. Online Fault Detection has been initialised...");
    }
    public void onTick() {
        ACLMessage msg = new ACLMessage(ACLMessage.INFORM);
        msg.addReceiver(new AID("PreprocessingAgent", AID.ISLOCALNAME));
        msg.setContent(OnlineDM1);
        myAgent.send(msg);
    }
}

public class OnlineDataManipulation2 extends TickerBehaviour{
    private OnlineDataManipulation2(Agent a, long period) {
        super(a, period);
    }
    public void onStart(){
        System.out.println("S.O.M. Online Fault Detection has been initialised...");
    }
    public void onTick() {
        ACLMessage msg = new ACLMessage(ACLMessage.INFORM);
        msg.addReceiver(new AID("PreprocessingAgent", AID.ISLOCALNAME));
        msg.setContent(OnlineDM2);
        myAgent.send(msg);
    }
}

public class ReceiveStatusOfModules extends CyclicBehaviour {
    public void action() {
        MessageTemplate tpl1 = MessageTemplate.and(
            MessageTemplate.MatchContent(NewCluster),
            MessageTemplate.MatchSender( new AID("FaultDiagnosisAgent",AID.ISLOCALNAME) ));
        MessageTemplate tp2 = MessageTemplate.and(
            MessageTemplate.MatchContent(FaultDetected),
            MessageTemplate.MatchSender( new AID("FaultDetectionAgent",AID.ISLOCALNAME) ));
    }
ACLMessage msg1 = myAgent.receive(tp1);
ACLMessage msg2 = myAgent.receive(tp2);

if( msg1 != null ) {
    System.out.println("n[" + getLocalName() + "]"
+ " received a message from [Fault Diagnosis Agent]");
    System.out.println("An new Cluster has been detected");
    } else {block();}

if( msg2 != null ) {
    String msgsender = msg2.getSender().getName();
    System.out.println("n[" + getLocalName() + "]"
+ " received a message from [Fault Detection Agent]");
    System.out.println("An ONLINE FAULT has been detected");
    removeBehaviour( odm );
    System.out.println("Online Data Manipulation has been STOPPED.");
    removeBehaviour( odg );
    System.out.println("Online Data Generation has been STOPPED.");
    } else {block();}
}

A.2. DATA PREPROCESSING AGENT

package OMatlab;
import jade.core.Agent;
import jade.core.AID;
import jade.core.behaviours.*;
import jade.lang.acl.ACLMessage;
import jade.lang.acl.MessageTemplate;
import matlabcontrol.MatlabConnectionException;
import matlabcontrol.MatlabInvocationException;
import OMatlab.MatlabConnectVariable;
@SuppressWarnings("serial")
public class PreprocessingAgent extends Agent {
    public final static String NEWDATA = "NEWDATA";
    public final static String HistoricalDT = "HistoricalDT";
public final static String HistoricalDT1 = "HistoricalDT1";
public final static String HistoricalDT2 = "HistoricalDT2";
public final static String OnlineDM = "OnlineDM";
public final static String OnlineDM1 = "OnlineDM1";
public final static String OnlineDM2 = "OnlineDM2";
public final static String OnlineDG = "OnlineDG";
private MatlabConnectVariable OnurMConnect = new MatlabConnectVariable();

protected void setup() {
    System.out.println("LSU PSE Agent Research Project.");
    System.out.println("Agent " + "[" + getLocalName() + "]" + " has been initiated...
    addBehaviour(new ReceiveStatusOfModules());
}

public class historicalCheck extends OneShotBehaviour {
    public void action() {
        try {
            OnurMConnect.MConnect1();
            double result = OnurMConnect.result1;
            ACLMessage msg = new ACLMessage(ACLMessage.INFORM);
            msg.addReceiver(new AID("FaultDetectionAgent", AID.ISLOCALNAME));
            msg.addReceiver(new AID("OperatorAgent", AID.ISLOCALNAME));
            msg.setContent(HistoricalDT);
            if (result == 1.0){
                System.out.println("Existing outliers removed from historical data");
                myAgent.send(msg);
                System.out.println("Message sent to Fault Detection Agent.");
            }
            else { System.out.println("No outliers detected in the
data");
                myAgent.send(msg);
                System.out.println("Message sent to Fault Detection
Agent." );
            }
        }
    System.out.println("Agent received the Result from Matlab: " + OnurMConnect.getResult());
    } catch (MatlabConnectionException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
    } catch (MatlabInvocationException e) {
        // TODO Auto-generated catch block
public class historicalCheck1 extends OneShotBehaviour {

    public void action() {
        OnurMConnect.MConnect1();
        ACLMessage msg = new ACLMessage(ACLMessage.INFORM);
        msg.addReceiver(new AID("FaultDetectionAgent", AID.ISLOCALNAME));
        msg.setContent(HistoricalDT1);
        myAgent.send(msg);
    }
}

public class historicalCheck2 extends OneShotBehaviour {

    public void action() {
        OnurMConnect.MConnect1();
        ACLMessage msg = new ACLMessage(ACLMessage.INFORM);
        msg.addReceiver(new AID("FaultDetectionAgent", AID.ISLOCALNAME));
        msg.setContent(HistoricalDT2);
        myAgent.send(msg);
    }
}

public class OnlineDataGenerator extends OneShotBehaviour {

    public void action() {
        try {
            OnurMConnect.MConnect3();
            double result = OnurMConnect.result3;
            System.out.println("Online Data is being received from the system...");
            System.out.println("Agent received the Result from Matlab: " + OnurMConnect.getResult());
        } catch (MatlabConnectionException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        } catch (MatlabInvocationException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
    }
}
public class onlineCheck extends OneShotBehaviour {
    public void action() {
        try {
            OnurMConnect.MConnect4();
            double result = OnurMConnect.result4;
            ACLMessage msg = new ACLMessage(ACLMessage.INFORM);
            msg.setSender(new AID("OnlineCheck", AID.ISLOCALNAME));
            msg.setContent(OnlineDM);

            if (result == 1.0) {
                System.out.println("Existing outliers removed from online data");
                myAgent.send(msg);
                System.out.println("Message sent to Fault Detection Agent.");
            } else if (result == -1.0) {
                System.out.println("No new data added");
            } else {
                System.out.println("No outliers detected in the online data");
                myAgent.send(msg);
                System.out.println("Message sent to Fault Detection Agent.");
            }
        } catch (MatlabConnectionException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        } catch (MatlabInvocationException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
    }
}

public class onlineCheck1 extends OneShotBehaviour {
    public void action() {
        OnurMConnect.MConnect1();
        System.out.println("No outliers detected in the online data");
        ACLMessage msg = new ACLMessage(ACLMessage.INFORM);
        msg.addReceiver(new AID("FaultDetectionAgent", AID.ISLOCALNAME));
        msg.setContent(OnlineDM1);
        myAgent.send(msg);
    }
}
public class onlineCheck2 extends OneShotBehaviour {
    public void action() {
        OnurMConnect.MConnect1();
        System.out.println("No outliers detected in the online data");
        ACLMessage msg = new ACLMessage(ACLMessage.INFORM);
        msg.addReceiver(new AID("FaultDetectionAgent", AID.ISLOCALNAME));
        msg.setContent(OnlineDM2);
        myAgent.send(msg);
    }
}

public class ReceiveStatusOfModules extends CyclicBehaviour {
    public void action() {
        MessageTemplate tpl = MessageTemplate.and(
            MessageTemplate.MatchContent(HistoricalDT),
            MessageTemplate.MatchSender( new AID("OperatorAgent",AID.ISLOCALNAME) ));
        MessageTemplate tp2 = MessageTemplate.and(
            MessageTemplate.MatchContent(OnlineDM),
            MessageTemplate.MatchSender( new AID("OperatorAgent",AID.ISLOCALNAME) ));
        MessageTemplate tp3 = MessageTemplate.and(
            MessageTemplate.MatchContent(OnlineDG),
            MessageTemplate.MatchSender( new AID("OperatorAgent",AID.ISLOCALNAME) ));
        MessageTemplate tp4 = MessageTemplate.and(
            MessageTemplate.MatchContent(HistoricalDT1),
            MessageTemplate.MatchSender( new AID("OperatorAgent",AID.ISLOCALNAME) ));
        MessageTemplate tp5 = MessageTemplate.and(
            MessageTemplate.MatchContent(OnlineDM1),
            MessageTemplate.MatchSender( new AID("OperatorAgent",AID.ISLOCALNAME) ));
        MessageTemplate tp6 = MessageTemplate.and(
            MessageTemplate.MatchContent(HistoricalDT2),
            MessageTemplate.MatchSender( new AID("OperatorAgent",AID.ISLOCALNAME) ));
        MessageTemplate tp7 = MessageTemplate.and(
            MessageTemplate.MatchContent(OnlineDM2),
            MessageTemplate.MatchSender( new AID("OperatorAgent",AID.ISLOCALNAME) ));

        ACLMessage msg1 = myAgent.receive(tpl);
        ACLMessage msg2 = myAgent.receive(tp2);
        ACLMessage msg3 = myAgent.receive(tp3);
        ACLMessage msg4 = myAgent.receive(tp4);
        ACLMessage msg5 = myAgent.receive(tp5);
        ACLMessage msg6 = myAgent.receive(tp6);
        ACLMessage msg7 = myAgent.receive(tp7);
if( msg1 != null ) {
    System.out.println("\n\n" + getLocalName() + "]" + " received a message from [Operator Agent]");
    System.out.println("P.C.A. Data Training is being started up...");
    addBehaviour(new historicalCheck());
} else {block();}

if( msg2 != null ) {
    String msgsender = msg2.getSender().getName();
    System.out.println("\n\n" + getLocalName() + "]" + " received a message from [Operator Agent]");
    System.out.println("Online Data Manipulation is being started up...");
    addBehaviour(new onlineCheck());
} else {block();}

if( msg3 != null ) {
    String msgsender = msg3.getSender().getName();
    System.out.println("\n\n" + getLocalName() + "]" + " received a message from [Operator Agent]");
    System.out.println("Online Data Generation is being started up...");
    addBehaviour(new OnlineDataGenerator());
} else {block();}

if( msg4 != null ) {
    addBehaviour(new historicalCheck1());
} else {block();}

if( msg5 != null ) {
    addBehaviour(new onlineCheck1());
} else {block();}

if( msg6 != null ) {
    addBehaviour(new historicalCheck2());
} else {block();}

if( msg7 != null ) {
    addBehaviour(new onlineCheck2());
} else {block();}

}
public class MessageSender extends OneShotBehaviour {
    public void action() {
        ACLMessage msg = new ACLMessage(ACLMessage.INFORM);
        msg.addReceiver(new AID("WhateverAgent", AID.ISLOCALNAME));
        msg.setContent(NEWDATA);
        myAgent.send(msg);
        System.out.println("Message sent to Whatever Agent.");
    }
}

A.3. FAULT DETECTION AGENT

package OMatlab;
import java.util.Arrays;
import jade.core.Agent;
import jade.core.AID;
import jade.core.behaviours.*;
import jade.lang.acl.ACLMessage;
import jade.lang.acl.MessageTemplate;
import matlabcontrol.MatlabConnectionException;
import matlabcontrol.MatlabInvocationException;
import OMatlab.MatlabConnectVariable;
import OMatlab.PreprocessingAgent.historicalCheck1;
import OMatlab.PreprocessingAgent.onlineCheck1;
@SuppressWarnings({"serial", "unused"})

public class FaultDetectionAgent extends Agent {
    public final static String NEWDATA = "NEWDATA";
    public final static String HistoricalDT = "HistoricalDT";
    public final static String HistoricalDT1 = "HistoricalDT1";
    public final static String HistoricalDT2 = "HistoricalDT2";
    public final static String OnlineDM = "OnlineDM";
    public final static String OnlineDM1 = "OnlineDM1";
    public final static String OnlineDM2 = "OnlineDM2";
    public final static String NewCluster = "NewCluster";
    public final static String FaultDetected = "FaultDetected";
    private MatlabConnectVariable OnurMConnect = new MatlabConnectVariable();
    private double[][] array1;
protected void setup()
{
    System.out.println("LSU PSE Agent Research Project.");
    System.out.println("Agent " + getLocalName() + "]" + " has been initiated...
    
    addBehaviour(new ReceiveStatusOfModules());
}

public class historicalCheck extends OneShotBehaviour {
    public void action() {

        try {
            OnurMConnect.MConnect9();
            OnurMConnect.MConnect2();
            double result = OnurMConnect.result2;

            if (result == 1.0)
                System.out.println("P.C.A. Model has been trained using Historical Data.");
            System.out.println("-----------------------------------------------------
You can start Online Process Monitoring.---------------------------------------------------");
        }

        else {
            System.out.println("PCA Model training has failed");
            
        } catch (MatlabConnectionException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        } catch (MatlabInvocationException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
    }
}

public class historicalCheck1 extends OneShotBehaviour {
    public void action() {

        try {
            OnurMConnect.MConnect11();
            System.out.println("F.D.A. Model has been trained using Historical Data.");
            System.out.println("-----------------------------------------------------
You can start Online Process Monitoring.---------------------------------------------------");
        }

        else {
            System.out.println("F.D.A Model training has failed");
            
        } catch (MatlabConnectionException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        } catch (MatlabInvocationException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
    }
}
e.printStackTrace();}

}

public class historicalCheck2 extends OneShotBehaviour {
    public void action() {
        try {
            OnurMConnect.MConnect13();
            System.out.println("S.O.M. Model has been trained using Historical Data.");
            System.out.println("-----------------------------------------------------You can start Online Process Monitoring.-----------------------------------------------------");
        } catch (MatlabConnectionException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        } catch (MatlabInvocationException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
    }
}

public class onlineCheck extends OneShotBehaviour {
    public void action() {
        try {
            OnurMConnect.MConnect5();
            double result = OnurMConnect.result5;
            OnurMConnect.MConnect7();
            OnurMConnect.MConnect8();

            ACLMessage msg = new ACLMessage(ACLMessage.INFORM);
            msg.addReceiver(new AID("OperatorAgent", AID.ISLOCALNAME));
            msg.addReceiver(new AID("FaultDiagnosisAgent", AID.ISLOCALNAME));
            msg.setContent(FaultDetected);
            msg.setContentType(FaultDetected);

            if (result == 1.0) {
                System.out.println("WARNING: ONLINE FAULT DETECTED!");
                System.out.println("!TURNED OFF! Contribution Plot.");
                OnurMConnect.MConnect6();
                myAgent.send(msg);
            } else {System.out.println("No fault detected in the online data");}
        }
    }
public class onlineCheck1 extends OneShotBehaviour {
    public void action() {
        OnurMConnect.MConnect1();
        ACLMessage msg = new ACLMessage(ACLMessage.INFORM);
        msg.addReceiver(new AID("FaultDiagnosisAgent", AID.ISLOCALNAME));
        msg.setContent(OnlineDM1);
        myAgent.send(msg);
    }
}

public class onlineCheck2 extends OneShotBehaviour {
    public void action() {
        OnurMConnect.MConnect1();
        ACLMessage msg = new ACLMessage(ACLMessage.INFORM);
        msg.addReceiver(new AID("FaultDiagnosisAgent", AID.ISLOCALNAME));
        msg.setContent(OnlineDM2);
        myAgent.send(msg);
    }
}

public class ReceiveStatusOfModules extends CyclicBehaviour {
    public void action() {
        MessageTemplate tpl = MessageTemplate.and(
            MessageTemplate.MatchContent(HistoricalDT),
            MessageTemplate.MatchSender( new AID("PreprocessingAgent", AID.ISLOCALNAME) ));
        MessageTemplate tp2 = MessageTemplate.and(
            MessageTemplate.MatchContent(OnlineDM),
            MessageTemplate.MatchSender( new AID("PreprocessingAgent", AID.ISLOCALNAME) ));
        MessageTemplate tp3 = MessageTemplate.and(
            MessageTemplate.MatchContent(HistoricalDT1),
            MessageTemplate.MatchSender( new AID("PreprocessingAgent", AID.ISLOCALNAME) ));
        MessageTemplate tp4 = MessageTemplate.and(
            MessageTemplate.MatchContent(OnlineDM1),
            MessageTemplate.MatchSender( new AID("PreprocessingAgent", AID.ISLOCALNAME) ));
    }
}
MessageTemplate tp5 = MessageTemplate.and(
    MessageTemplate.MatchContent(HistoricalDT2),
    MessageTemplate.MatchSender( new AID("PreprocessingAgent",AID.ISLOCALNAME) ));
MessageTemplate tp6 = MessageTemplate.and(
    MessageTemplate.MatchContent(OnlineDM2),
    MessageTemplate.MatchSender( new AID("PreprocessingAgent",AID.ISLOCALNAME) ));

ACLMessage msg1 = myAgent.receive(tp1);
ACLMessage msg2 = myAgent.receive(tp2);
ACLMessage msg3 = myAgent.receive(tp3);
ACLMessage msg4 = myAgent.receive(tp4);
ACLMessage msg5 = myAgent.receive(tp5);
ACLMessage msg6 = myAgent.receive(tp6);

if( msg1 != null ) {
    String msgsender = msg1.getSender().getName();
    System.out.println( 
        getLocalName() + getLocalName() + " received a message from " + msgsender);
    System.out.println("PCA Model Training is being started up.");
    addBehaviour(new historicalCheck());
} else {block();}
if( msg2 != null ) {
    String msgsender = msg2.getSender().getName();
    addBehaviour(new onlineCheck());
} else {block();}
if( msg3 != null ) {
    addBehaviour(new historicalCheck1());
} else {block();}
if( msg4 != null ) {
    addBehaviour(new onlineCheck1());
} else {block();}
if( msg5 != null ) {
    addBehaviour(new historicalCheck2());
} else {block();}
if( msg6 != null ) {
    addBehaviour(new onlineCheck2());
A.4. FAULT DIAGNOSIS AGENT

package OMatlab;
import java.util.Arrays;
import jade.core.Agent;
import jade.core.AID;
import jade.lang.acl.ACLMessage;
import jade.lang.acl.MessageTemplate;
import matlabcontrol.MatlabConnectionException;
import matlabcontrol.MatlabInvocationException;
import OMatlab.MatlabConnectVariable;
@SuppressWarnings({ "serial", "unused" })

public class FaultDiagnosisAgent extends Agent {
    public final static String NEWDATA = "NEWDATA";
    public final static String HistoricalDT = "HistoricalDT";
    public final static String OnlineDM = "OnlineDM";
    public final static String OnlineDM1 = "OnlineDM1";
    public final static String OnlineDM2 = "OnlineDM2";
    public final static String NewCluster = "NewCluster";
    public final static String FaultDetected = "FaultDetected";
    private MatlabConnectVariable OnurMConnect = new MatlabConnectVariable();
    private double[][] array1;

    protected void setup()
    {
        System.out.println("LSU PSE Agent Research Project. ");
        System.out.println("Agent " + "[" + getLocalName() + "]" + " has been initiated...
        addBehaviour(new ReceiveStatusOfModules());
    }

    public class onlineCheck extends OneShotBehaviour{
        public void action() {
            // Implementation of the onlineCheck action
        }
    }
}
try {
    OnurMConnect.MConnect10();
    double result = OnurMConnect.result10;
    System.out.println("The classification result is: FAULT # " + result);
}

} catch (MatlabConnectionException e) {
    // TODO Auto-generated catch block
    e.printStackTrace();
} catch (MatlabInvocationException e) {
    // TODO Auto-generated catch block
    e.printStackTrace();
}

public class onlineCheck1 extends OneShotBehaviour {
    public void action() {
        try {
            OnurMConnect.MConnect12();
            double result = OnurMConnect.result12;
            System.out.println("GUESSED CLASS: " + result);
        } catch (MatlabConnectionException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        } catch (MatlabInvocationException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
    }
}

public class onlineCheck2 extends OneShotBehaviour {
    public void action() {
        try {
            OnurMConnect.MConnect14();
            double result = OnurMConnect.result14;
            if (result == 1.0) {
                System.out.println("WARNING: ONLINE FAULT DETECTED!");
                double result1 = OnurMConnect.result141;
                System.out.println("GUESSED CLASS: " + result1);
            } else {
                System.out.println("No fault detected in the online data");
            }
        } catch (MatlabConnectionException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        } catch (MatlabInvocationException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
    }
}
public class ReceiveStatusOfModules extends CyclicBehaviour {
    public void action() {
        MessageTemplate tp1 = MessageTemplate.and(
            MessageTemplate.MatchContent(FaultDetected),
            MessageTemplate.MatchSender(new AID("FaultDetectionAgent", AID.ISLOCALNAME)));
        MessageTemplate tp2 = MessageTemplate.and(
            MessageTemplate.MatchContent(OnlineDM1),
            MessageTemplate.MatchSender(new AID("FaultDetectionAgent", AID.ISLOCALNAME)));
        MessageTemplate tp3 = MessageTemplate.and(
            MessageTemplate.MatchContent(OnlineDM2),
            MessageTemplate.MatchSender(new AID("FaultDetectionAgent", AID.ISLOCALNAME)));

        ACLMessage msg1 = myAgent.receive(tp1);
        ACLMessage msg2 = myAgent.receive(tp2);
        ACLMessage msg3 = myAgent.receive(tp3);

        if( msg1 != null ) {
            String msgsender = msg1.getSender().getName();
            addBehaviour(new onlineCheck());
        } else {block();}

        if( msg2 != null ) {
            String msgsender = msg2.getSender().getName();
            addBehaviour(new onlineCheck1());
        } else {block();}

        if( msg3 != null ) {
            String msgsender = msg3.getSender().getName();
            addBehaviour(new onlineCheck2());
        } else {block();}
    }
}
public class MessageSender extends OneShotBehaviour {
    public void action() {
        ACLMessage msg = new ACLMessage(ACLMessage.INFORM);
        msg.addReceiver(new AID("OutlierAgent", AID.ISLOCALNAME));
        msg.setContent(NEWDATA);
        myAgent.send(msg);
        System.out.println("Message sent to WhicheverAgent.");
    }
}

A.5. OPERATOR AGENT USER INTERFACE

package Onur;

import java.awt.BorderLayout;
import java.awt.EventQueue;
import java.io.IOException;
import java.io.OutputStream;
import java.io.PrintStream;
import javax.swing.DefaultComboBoxModel;
import javax.swing.JComboBox;
import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.JTextArea;
import javax.swing.JScrollPane;
import javax.swing.JTextPane;
import java.awt.SystemColor;
import javax.swing.JButton;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;
import java.awt.event.ItemEvent;
import java.awt.event.ItemListener;
import javax.swing.JMenuBar;
import javax.swing.JMenu;
import javax.swing.JMenuItem;
import Onur.AboutDialog;
import javax.swing.JPopupMenu;
import java.awt.Component;
import java.awt.event.MouseAdapter;
import java.awt.event.MouseEvent;
import javax.swing.JToggleButton;

@SuppressWarnings({"serial", "unused"})
public class OperatorGUI extends JFrame {
    private JTextArea textArea = new JTextArea();
    private JPanel contentPane;
    public String cluster;
    private OperatorAgent myAgent; // Reference to the agent class
    public void setAgent(OperatorAgent a) {
        myAgent = a;
    }

    private void updateTextArea(final String text) {
        SwingUtilities.invokeLater(new Runnable() {
            public void run() {
                textArea.append(text);
            }
        });
    }

    private void redirectSystemStreams() {
        OutputStream out = new OutputStream() {
            @Override
            public void write(int b) throws IOException {
                updateTextArea(String.valueOf((char) b));
            }
            @Override
            public void write(byte[] b, int off, int len) throws IOException {
                updateTextArea(new String(b, off, len));
            }
            @Override
            public void write(byte[] b) throws IOException {
                write(b, 0, b.length);
            }
        };
        System.setOut(new PrintStream(out, true));
        System.setErr(new PrintStream(out, true));
    }

    /** Launch the application. **/
public static void main(String[] args) {
    EventQueue.invokeLater(new Runnable() {
        public void run() {
            try {
                OperatorGUI frame = new OperatorGUI();
                frame.setVisible(true);
            } catch (Exception e) {
                e.printStackTrace();
            }
        }
    });
}

/** Create the frame. */
public OperatorGUI() {
    setTitle("Operator Agent GUI");
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    setBounds(100, 100, 645, 525);
    JMenuBar menuBar = new JMenuBar();
    setJMenuBar(menuBar);
    JMenu mnFile = new JMenu("File");
    menuBar.add(mnFile);
    JMenuItem mntmItem = new JMenuItem("Item 1");
    mnFile.add(mntmItem);
    JMenu mnAbout = new JMenu("About");
    menuBar.add(mnAbout);
    JMenuItem mntmAboutThisGui = new JMenuItem("About this GUI");
    mnAbout.add(mntmAboutThisGui);
    JMenu mmFile = new JMenu("File");
    menuBar.add(mmFile);
    JMenuItem mntmItem = new JMenuItem("Item 1");
    mmFile.add(mntmItem);
    JMenu mmAbout = new JMenu("About");
    menuBar.add(mmAbout);
    JMenuItem mntmAboutThisGui = new JMenuItem("About this GUI");
    mnAboutThisGui.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e) {
            AboutDialog ad = new AboutDialog();
            ad.setVisible(true);
        }
    });
    mmAbout.add(mntmAboutThisGui);
    contentPane = new JPanel();
    contentPane.setBorder(new EmptyBorder(5, 5, 5, 5));
    getContentPane().add(contentPane);
    contentPane.setLayout(null);
    final JPopupMenu popupMenu = new JPopupMenu();
    addPopup(contentPane, popupMenu);
}
JMenuItem mntmAbout = new JMenuItem("About");
mntmAbout.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        AboutDialog ad = new AboutDialog();
        ad.setVisible(true);
    }
});
popupMenu.addMouseListner(new MouseAdapter() {
    @Override
    public void mouseReleased(MouseEvent e) {
        if (e.getButton() == MouseEvent.BUTTON3)
        {
            popupMenu.show(e.getComponent(), e.getX(), e.getY());
        }
    }
});
popupMenu.add(mntmAbout);

JScrollPane scrollPane = new JScrollPane();
scrollPane.setBounds(10, 28, 610, 307);
contentPane.add(scrollPane);
scrollPane.setViewportView(textArea);
textArea.setEditable(false);
JTextPane txtpnConsoleOutput = new JTextPane();
    txtpnConsoleOutput.setEditable(false);
    txtpnConsoleOutput.setBackground(SystemColor.control);
    txtpnConsoleOutput.setText("Console Output: ");
    txtpnConsoleOutput.setBounds(0, 8, 99, 20);
    contentPane.add(txtpnConsoleOutput);

JButton btnHistoricalDataTraining = new JButton("P.C.A. Data Training");
btnHistoricalDataTraining.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent arg0) {
        myAgent.behaSend1();
    }
});
btnHistoricalDataTraining.setBounds(10, 346, 178, 28);
contentPane.add(btnHistoricalDataTraining);
JToggleButton tglbtnNewToggleButton = new JToggleButton("P.C.A. Online Fault Detection");
tglbtnNewToggleButton.addItemListener(new ItemListener() {
    public void itemStateChanged(ItemEvent ev) {
        if(ev.getStateChange()==ItemEvent.SELECTED){
            myAgent.behaSend2();
            System.out.println("'P.C.A. Online Fault Detection' button is selected");
        } else if(ev.getStateChange()==ItemEvent.DESELECTED){
            myAgent.behaSend3();
            System.out.println("'P.C.A. Online Fault Detection' button is not selected");
        }
    }
});
tglbtnNewToggleButton.setBounds(397, 346, 223, 28);
contentPane.add(tglbtnNewToggleButton);

JToggleButton tglbtnGenerateDataOnline = new JToggleButton("Generate Data Online");
tglbtnGenerateDataOnline.addItemListener(new ItemListener() {
    public void itemStateChanged(ItemEvent ev) {
        if(ev.getStateChange()==ItemEvent.SELECTED){
            myAgent.behaSend4();
            System.out.println("'Generate Data Online' button is selected");
        } else if(ev.getStateChange()==ItemEvent.DESELECTED){
            myAgent.behaSend5();
            System.out.println("'Generate Data Online' button is not selected");
        }
    }
});
tglbtnGenerateDataOnline.setBounds(198, 372, 189, 58);
contentPane.add(tglbtnGenerateDataOnline);

JButton btnFdaDataTraining = new JButton("F.D.A. Data Training");
btnFdaDataTraining.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent arg0) {
        myAgent.behaSend6();
    }
});
btnFdaDataTraining.setBounds(10, 385, 178, 28);
contentPane.add(btnFdaDataTraining);

JToggleButton tglbtnFdaOnlineFault = new JToggleButton("F.D.A. Online Fault Detection");
tglbtnFdaOnlineFault.addItemListener(new ItemListener() {
    public void itemStateChanged(ItemEvent ev) {
        if(ev.getStateChange()==ItemEvent.SELECTED){
            myAgent.behaSend7();
            System.out.println("'F.D.A. Online Fault Detection' button is selected");
        } else if(ev.getStateChange()==ItemEvent.DSELETED){
            myAgent.behaSend8();
            System.out.println("'F.D.A. Online Fault Detection' button is not selected");
        }
    }
});
tglbtnFdaOnlineFault.setBounds(397, 385, 223, 28);
contentPane.add(tglbtnFdaOnlineFault);

JButton btnSomDataTraining = new JButton("S.O.M. Data Training");
btnSomDataTraining.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent arg0) {
        myAgent.behaSend9();
    }
});
btnSomDataTraining.setBounds(10, 424, 178, 28);
contentPane.add(btnSomDataTraining);

JToggleButton tglbtnSomOnlineFault = new JToggleButton("S.O.M. Online Fault Detection");
tglbtnSomOnlineFault.addItemListener(new ItemListener() {
    public void itemStateChanged(ItemEvent ev) {
        if(ev.getStateChange()==ItemEvent.SELECTED){
            myAgent.behaSend10();
            System.out.println("'S.O.M. Online Fault Detection' button is selected");
        } else if(ev.getStateChange()==ItemEvent.DESELECTED){
            myAgent.behaSend11();
            System.out.println("'S.O.M. Online Fault Detection' button is not selected");
        }
    }
});
tglbtnSomOnlineFault.setBounds(397, 424, 223, 28);
contentPane.add(tglbtnSomOnlineFault);
redirectSystemStreams();
}

private static void addPopup(Component component, final JPopupMenu popup) {
    component.addMouseListener(new MouseAdapter() {
        public void mousePressed(MouseEvent e) {
            if (e.isPopupTrigger()) {
                showMenu(e);
            }
        }
        public void mouseReleased(MouseEvent e) {
            if (e.isPopupTrigger()) {
                showMenu(e);
            }
        }
        private void showMenu(MouseEvent e) {
            popup.show(e.getComponent(), e.getX(), e.getY());
        }
    });
}

public void popup() {
    OnurComboBox onurcombobox = null;
    try {
        onurcombobox = new OnurComboBox();
    } catch (IOException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
    }
    onurcombobox.setVisible(true);
    // cluster = onurcombobox.combo;  HAVE TO GET THIS AFTER THE WINDOW IS DISPOSED; SO PROBABLY IN A DIFFERENT METHOD AFTWEARDS.
    // System.out.println("Cluster :::" + cluster);
}
package OMatlab;
import matlabcontrol.*;
import matlabcontrol.extensions.*;
import java.util.Arrays;
@SuppressWarnings("unused")
public class MatlabConnectVariable extends MatlabProxyFactory{
    private String MConnectStatus;
    public double result1;
    public double result2;
    public double result3;
    public double result4;
    public double result5;
    public double result6;
    public double result10;
    public double result12;
    public double result14;
    public double result141;
    public double[][] array1;
    public double[][] array2;
    public int i;
    public boolean conStat;

    public String getMConnectStatus() {
        return MConnectStatus;
    }

    MatlabConnectVariable() {
        if (conStat = true) {
            MConnectStatus="The MatlabConnect class instance has been initiated";
        } else {MConnectStatus="WARNING: The MatlabConnect class instance could not been initiated";}
    }

    public void MConnect1() throws MatlabConnectionException, MatlabInvocationException {
        MatlabProxyFactoryOptions options = new MatlabProxyFactoryOptions.Builder()
            .setUsePreviouslyControlledSession(true)
MatlabProxyFactory factory = new MatlabProxyFactory(options);
MatlabProxy proxy = factory.getProxy();
conStat = proxy.isConnected();
proxy.feval("data_cleaner_script");
result1 = ((double[]) proxy.getVariable("outliers_removed"))[0];
proxy.disconnect();

public void MConnect2() throws MatlabConnectionException, MatlabInvocationException
{
MatlabProxyFactoryOptions options = new MatlabProxyFactoryOptions.Builder()
    .setUsePreviouslyControlledSession(true)
    .setMatlabLocation(null).build();
MatlabProxyFactory factory = new MatlabProxyFactory(options);
MatlabProxy proxy = factory.getProxy();
conStat = proxy.isConnected();
proxy.feval("pca_model_script");
result2 = ((double[]) proxy.getVariable("trained"))[0];
proxy.disconnect();
}

public void MConnect3() throws MatlabConnectionException, MatlabInvocationException
{
MatlabProxyFactoryOptions options = new MatlabProxyFactoryOptions.Builder()
    .setUsePreviouslyControlledSession(true)
    .setMatlabLocation(null).build();
MatlabProxyFactory factory = new MatlabProxyFactory(options);
MatlabProxy proxy = factory.getProxy();
conStat = proxy.isConnected();
proxy.feval("Online_Data_Generator");
proxy.disconnect();
}

public void MConnect4() throws MatlabConnectionException, MatlabInvocationException
{
MatlabProxyFactoryOptions options = new MatlabProxyFactoryOptions.Builder()
    .setUsePreviouslyControlledSession(true)
    .setMatlabLocation(null).build();
MatlabProxyFactory factory = new MatlabProxyFactory(options);
MatlabProxy proxy = factory.getProxy();
conStat = proxy.isConnected();
proxy.feval("Online_Data_Generator");
proxy.disconnect();
}
MatlabProxyFactory factory = new MatlabProxyFactory(options);
MatlabProxy proxy = factory.getProxy();
conStat = proxy.isConnected();
proxy.feval("online_data_cleaner_script");
result4 = ((double[]) proxy.getVariable("is_outlier"))[0];
proxy.disconnect();
}

public void MConnect5() throws MatlabConnectionException, MatlabInvocationException
{
MatlabProxyFactoryOptions options = new MatlabProxyFactoryOptions.Builder()
    .setUsePreviouslyControlledSession(true)
    .setMatlabLocation(null).build();
MatlabProxyFactory factory = new MatlabProxyFactory(options);
MatlabProxy proxy = factory.getProxy();
conStat = proxy.isConnected();
proxy.feval("Tsq_Q_detection_script");
result5 = ((double[]) proxy.getVariable("fault_detected"))[0];
proxy.disconnect();
}

public void MConnect6() throws MatlabConnectionException, MatlabInvocationException
{
MatlabProxyFactoryOptions options = new MatlabProxyFactoryOptions.Builder()
    .setUsePreviouslyControlledSession(true)
    .setMatlabLocation(null).build();
MatlabProxyFactory factory = new MatlabProxyFactory(options);
MatlabProxy proxy = factory.getProxy();
conStat = proxy.isConnected();
proxy.feval("contribution_plot_script");
proxy.disconnect();
}

public void MConnect7() throws MatlabConnectionException, MatlabInvocationException
{
MatlabProxyFactoryOptions options = new MatlabProxyFactoryOptions.Builder()
    .setUsePreviouslyControlledSession(true)
    .setMatlabLocation(null).build();
MatlabProxyFactory factory = new MatlabProxyFactory(options);
MatlabProxy proxy = factory.getProxy();
conStat = proxy.isConnected();
proxy.feval("dynamic_plot_Tsq");
proxy.disconnect();

public void MConnect8() throws MatlabConnectionException, MatlabInvocationException
{
    MatlabProxyFactoryOptions options = new MatlabProxyFactoryOptions.Builder()
        .setUsePreviouslyControlledSession(true)
        .setMatlabLocation(null).build();
    MatlabProxyFactory factory = new MatlabProxyFactory(options);
    MatlabProxy proxy = factory.getProxy();
    conStat = proxy.isConnected();
    proxy.feval("dynamic_plot_Q");
    proxy.disconnect();
}

public void MConnect9() throws MatlabConnectionException, MatlabInvocationException
{
    MatlabProxyFactoryOptions options = new MatlabProxyFactoryOptions.Builder()
        .setUsePreviouslyControlledSession(true)
        .setMatlabLocation(null).build();
    MatlabProxyFactory factory = new MatlabProxyFactory(options);
    MatlabProxy proxy = factory.getProxy();
    conStat = proxy.isConnected();
    proxy.feval("pca_model1_script");
    proxy.feval("pca_model2_script");
    proxy.feval("pca_model4_script");
    proxy.feval("pca_model8_script");
    proxy.feval("pca_model11_script");
    proxy.feval("pca_model13_script");
    proxy.feval("pca_model14_script");
    proxy.disconnect();
}

public void MConnect10() throws MatlabConnectionException, MatlabInvocationException
{
    MatlabProxyFactoryOptions options = new MatlabProxyFactoryOptions.Builder()
        .setUsePreviouslyControlledSession(true)
        .setMatlabLocation(null).build();
    MatlabProxyFactory factory = new MatlabProxyFactory(options);
    MatlabProxy proxy = factory.getProxy();
    conStat = proxy.isConnected();
    proxy.feval("pca_model1_script");
    proxy.feval("pca_model2_script");
    proxy.feval("pca_model4_script");
    proxy.feval("pca_model8_script");
    proxy.feval("pca_model11_script");
    proxy.feval("pca_model13_script");
    proxy.feval("pca_model14_script");
    proxy.disconnect();
}
.setMatlabLocation(null).build();
MatlabProxyFactory factory = new MatlabProxyFactory(options);
MatlabProxy proxy = factory.getProxy();
conStat = proxy.isConnected();
proxy.feval("Fault_Diagnosis_script");
result10 = ((double[]) proxy.getVariable("classification_result"))[0];
proxy.disconnect();
}

public void MConnect11() throws MatlabConnectionException, MatlabInvocationException
{
MatlabProxyFactoryOptions options = new MatlabProxyFactoryOptions.Builder()
.setUsePreviouslyControlledSession(true)
.setMatlabLocation(null).build();
MatlabProxyFactory factory = new MatlabProxyFactory(options);
MatlabProxy proxy = factory.getProxy();
conStat = proxy.isConnected();
proxy.feval("FDA_Historical_Training");
proxy.getVariable("classification_result")[0];
proxy.disconnect();
}

public void MConnect12() throws MatlabConnectionException, MatlabInvocationException
{
MatlabProxyFactoryOptions options = new MatlabProxyFactoryOptions.Builder()
.setUsePreviouslyControlledSession(true)
.setMatlabLocation(null).build();
MatlabProxyFactory factory = new MatlabProxyFactory(options);
MatlabProxy proxy = factory.getProxy();
conStat = proxy.isConnected();
proxy.feval("FDA_Historical_Training");
result12 = ((double[]) proxy.getVariable("guessed_class"))[0];
proxy.disconnect();
}

public void MConnect13() throws MatlabConnectionException, MatlabInvocationException
{
MatlabProxyFactoryOptions options = new MatlabProxyFactoryOptions.Builder()
.setUsePreviouslyControlledSession(true)
.setMatlabLocation(null).build();
MatlabProxyFactory factory = new MatlabProxyFactory(options);
MatlabProxy proxy = factory.getProxy();
conStat = proxy.isConnected();
proxy.feval("SOM_Historical_Training");
}
proxy.disconnect();

public void MConnect14() throws MatlabConnectionException, MatlabInvocationException {
    MatlabProxyFactoryOptions options = new MatlabProxyFactoryOptions.Builder()
        .setUsePreviouslyControlledSession(true)
        .setMatlabLocation(null).build();
    MatlabProxyFactory factory = new MatlabProxyFactory(options);
    MatlabProxy proxy = factory.getProxy();
    conStat = proxy.isConnected();
    proxy.feval("SOM_Online_Detection");
    result14 = ((double[]) proxy.getVariable("fault_detected"))[0];
    result141 = ((double[]) proxy.getVariable("guessed_class"))[0];
    proxy.disconnect();
}
}
THE VITA

Onur Dogu was born in Ankara, Turkey. After completing his work at TED Ankara College Foundation High School, Ankara, in 2007, he entered Middle East Technical University (METU) in Ankara. He received the degree of Bachelor of Science in Chemical Engineering from METU in June 2011. Thereafter, he proceeded to Louisiana State University, Louisiana, United States of America, for graduate studies and is currently a candidate for the degree of Master of Science in Chemical Engineering, to be awarded in May 2014.