Configurable FPGA-Based Outlier Detection for Time Series Data

by

Ghazaleh Vazhbakht, B.Sc.

A thesis submitted to the Faculty of Graduate and Postdoctoral Affairs
in partial fulfillment of the requirements for the degree of

Master of Applied Science

in

Electrical and Computer Engineering

Ottawa-Carleton Institute for Electrical and Computer Engineering

Carleton University
Ottawa, Ontario

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ACRONYMS

ACF	Autocorrelation Function
AD	Anomaly Detection
ADAM	Anomaly Detection and Mitigation
AO	Additive Outlier
AR	Autoregressive
ARIMA	Autoregressive Integrated Moving Average
ARMA	Autoregressive Moving Average
ASIC	Application-Specific Integrated Circuit
BIC	Bayesian Information Criterion
DFT	Discrete Fourier Transform
DSP	Digital Signal Processing
ECG	Electrocardiography
FEM	Feature Extraction Module
FIFO	First In First Out
FN	False Negative
FP	False Positive
FPGA	Field Programmable Gate Array
GPS	Global Positioning System
HDL	Hardware Description Language
IO	Innovative Outlier
KNN	K-Nearest Neighbour
LN	Linear Network
LOF	Local Outlier Factor
LS	Level Shift
MA	Moving Average
MAD	Median Absolute Deviation
ML	Maximum Likelihood
MLP	Multi-Layer Perception
NIC	Network Interface Card
NIDS	Network Intrusion Detection System
PACF	Partial Autocorrelation Function
PCA	Principle Component Analysis
PI	Prediction Interval
RTL	Register Transfer Level
SAX	Symbol Aggregate Approximation
SFDIA	Sensor Failure Detection, Identification and Accommodation
SVR	Support Vector Regression
TC	Temporary Change
TN	True Negative
TP	True Positive
VHDL	Very High Speed Integrated Circuit Hardware Description Language

ACKNOWLEDGMENTS

First and foremost, I would like to express my gratitude to my supervisor Dr. Leonard MacEachern for his patience and support. I appreciate the expertise and time that he has contributed throughout the course of this thesis.

I would like to thank Ms. Anna Lee and Ms. Blazenka Power for their supports in administrative works all these years. Also, I am grateful to Mr. Nagui Mikhail for his technical and lab supports.

My deepest gratitude goes to my family; my parents and my grandmother for their continuous love and confidence in me, my sister Bahareh for her encouragements and moral support throughout my life.

Last but not least, I am grateful to Majid Namaki Shoushtari, who helped me tremendously with all the academic discussions and motivated me with his encouragements.

I would also like to acknowledge my friends and colleagues who have directly or indirectly, knowingly or unknowingly contributed towards this thesis.

ABSTRACT

FPGA-based configurable real-time outlier detection hardware is presented. The FPGA implementation reduces the power consumption by 89% compared to the existing software implementations on a general purpose processor. The implemented outlier detection technique is based on the joint estimation of model parameters and outlier effects in time series. It models the data with an autoregressive-moving-average (ARMA) process and identifies the outliers based on test statistics using unbiased parameters. A configuration of our implementation, that is discussed and evaluated in this thesis, is capable of detecting multiple additive outliers in time series data with a detection accuracy of 99% and has a type I error rate of 1.05%.

We used the MATLAB implementation of the outlier detection technique to compare its accuracy to the existing implementation of the same algorithm in "tsoutliers" package of the statistical programming language R. Using the MATLAB HDL Coder, we generated a configurable hardware implementation in Verilog. The design was simulated to verify its correctness and synthesized on an Altera FPGA device from the Stratix V family. The design is configurable by adjusting the number of iterations for the optimization process, the number of samples in the time series data, and the critical value. A configuration of this design has a total power dissipation of 1.14W, while processing 35 million data points per second, giving an energy usage of 0.032 microjoule per processed data point.

Chapter 1

Introduction

With the advances in networking and the ever-increasing number of devices connected to networks, the amount of gathered data has been increasing rapidly. Data is collected from a variety of sources; from devices which we willingly use such as our mobile phones and wearable devices, to sensors in the environment, such as traffic cameras and GPS satellites. As a result of the rapid inflation in the amount of generated data, the urge to analyze the data has also escalated in different domains including economics, business, engineering, health and environmental science.

A valuable piece of information that can be derived from data is the possibility of any abnormalities in the system. These abnormalities can indicate an unusual behaviour in the system that is caused by an unexpected event; or they could be due to measurement errors in the data generation process that can affect further processing of data. In either case, it is essential to identify these outliers[1]. The approaches to detect these outliers are provided by the algorithms developed in fields including machine learning, computer science, and statistics.

[1] Abnormality, outlier and anomaly are used interchangeably throughout the rest of the thesis.

In several applications, outlier detection algorithm plays a critical role. For instance: detecting unusual activities on credit cards can indicate credit card fraud [1]; Unexpected changes in temperature, humidity or annual rainfall over time show the possibility of drought or flood and climate change [2][3]; Health monitoring systems can identify symptoms of various diseases by detecting outliers [4]; Unreasonable data transactions in a network can indicate unauthorized access to a device or an intrusion in the network system [5].

1.1 Outlier Detection in Time Series

Many data sets appear in the form of time series. Monthly level of pollution in a city, daily price of stock, hourly temperature of a furnace, and number of users connected to a server in a day are examples of time series data. Different data types require their own dedicated outlier detection algorithms [6]. For example, in analyzing time series data, it is important to consider the correlation between consecutive data points.

Statistical algorithms are the common approach for time series analysis in which describing the data, that represents the behaviour of a physical phenomenon with mathematical models is well-established [7]. The procedure of detecting outliers in time series involves fitting a model to the data and using statistical hypothesis tests to determine whether the observed samples belong to the model or not. Algorithms such as estimation of time series parameters in the presence of outliers [8], and joint estimation of model parameters and outlier effects [9] have been proved to be adequate and the latter algorithm is widely used in the industry.

Software packages and tools such as SAS, R, and Python ease the process of analyzing time series data. For instance, the procedure of detecting outliers presented in joint estimation of model parameters and outlier effects is implemented in a package called

"tsoutliers" in R [10]. However, these software implementations that run on general purpose processors, are not suitable choices for application domains that require real-time and low power detection of outliers (e.g. battery-powered devices).Accordingly, a dedicated hardware that can perform this task is desirable.

1.2 Thesis Objective and Contributions

Our goal in this thesis is to investigate and evaluate the feasibility of implementing a time series outlier detection technique in hardware which is more power efficient than a software implementation where a package such as tsoutliers[1] in R runs on a general purpose processor. Additionally, the hardware implementation should have a similar accuracy to the software implementation while performing the outlier detection task in real-time.

In this thesis, we present a configurable FPGA-based outlier detection technique for time series that has a detection accuracy of 99%, and average type I error rate (False positive rate) of 1.05%. Our experiments show that the FPGA implementation has a total power dissipation of 1.14W and reduces the power consumption by 89% compared to a general purpose processor running tsoutliers package.

The implemented outlier detection technique uses an ARMA process to model the data and identify the outliers. In the next chapter, the concepts of outliers and different techniques to detect them are discussed. In addition, Chapter 2 reviews the hardware implementation of outlier detection techniques. Chapter 3 explains the theory behind modeling time series with ARMA models and the details of the outlier detection algorithm that we have selected to implement in this thesis. Also, It discusses the assumptions and modifications required for hardware design. The details of hardware implementation are explained

[1] "tsoutliers" is a package in statistical R language for detecting outliers in time series data. The approach implemented in this package is based on the algorithm proposed in [9].

in Chapter 4. Results and discussions are presented in Chapter 5, and Chapter 6 concludes this thesis.

Chapter 2

Background

2.1 Definition of Outlier

Although it seems that there is no universally accepted definition for outlier, there is a convenient one that has been used in statistics. Outlier as defined by Hawkins [11] is "an observation that deviates so much from other observations as to arouse suspicions that it was generated by a different mechanism". Another definition was proposed by Barnett and Lewis [12] which states "an observation (or subset of observations) which appears to be inconsistent with the remainder of that set of data is outlier". Grubs [13] suggests that, "an outlying observation or an outlier is one that appears to deviate markedly from other members of the sample in which it occurs". Not only the definitions are slightly different; but often authors chose distinct names such as anomalies, deviations, and exceptions based on their application.

Figure 2.1 shows outliers in a two-dimensional data set. As can be seen, data has been classified into two classes of C_1 and C_2. The points O_1, O_2 and O_3 don't belong to any of the classes and are marked as outliers.

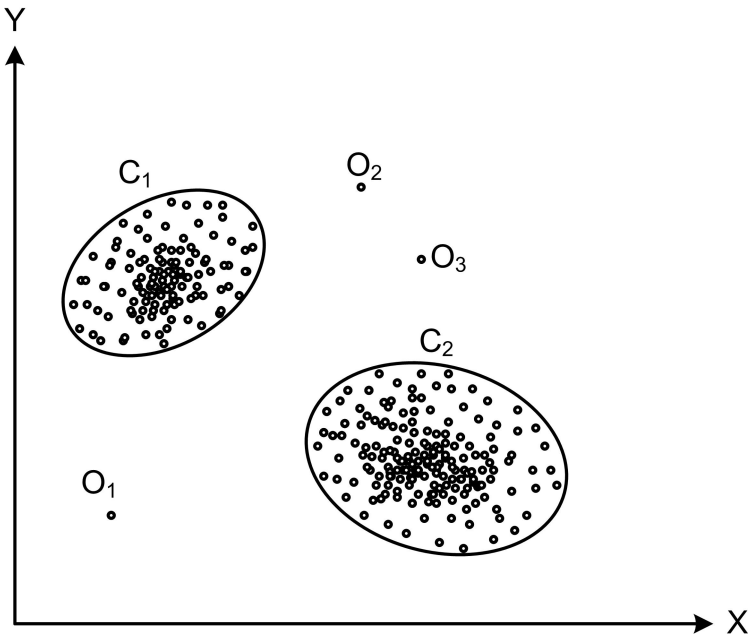


Figure 2.1: Outliers in two-dimensional data set adapted from [14], O_1, O_2 and O_3 are outlier observations that don't belong to any class.

Outliers are the result of an unusual behavior in data; therefore often they indicate abnormal characteristics in the system and sometimes they are due to measurement errors, human errors or instrumentation errors. In all these cases, they can affect the dynamics of data and its analysis. Consequently, outlier detection plays an important role in providing insight about the data. As mentioned before, outlier detection can be used in different applications such as, intrusion detection, credit card fraud detection, sensor networks, system diagnosis, health care systems, and environmental science.

2.2 Outlier Detection

Outlier detection is the process of identifying outliers in data. Different techniques have been proposed for outlier detection during the past couple of decades. Although the concept and base of these algorithms are similar, they often have different names, such as anomaly detection, noise detection, novelty detection and deviation detection. In [15], the

authors have surveyed a wide range of these techniques derived from multiple fields, such as statistics, neural networks and machine learning. They have categorized these techniques into three major groups:

- Unsupervised outlier detection techniques: There is no prior knowledge about the data under analysis.
- Supervised outlier detection techniques: There are samples of both normal and abnormal data.
- Semi-supervised outlier detection techniques: There are only samples of the normal data and no samples of abnormal ones.

A structured and comprehensive overview on a wider range of anomaly detection techniques has been provided in [14]. All the techniques have been categorized in six groups, namely classification based, clustering based, near neighbor based, statistical, information theoretic and spectral. For each of these approaches, a basic method and its derivatives, advantages and disadvantages have been explained. In addition, applications in which these anomaly detection algorithms can be used have been described. Furthermore, aspects such as nature of data and type of anomaly have been considered to determine the complexity of different applications and techniques. Anomalies have been categorized in three groups of point anomalies, contextual anomalies and collective anomalies.

The type of data is a significant aspect that is also used to categorize the outlier detection techniques. Data can be continuous, discrete, categorical, sequenced, spatial or graphed data and specific approaches can work better for different types of data. For example, with regard to temporal data, the continuity of time is the contextual variable and plays an important role in the formulation and the analysis of data [6]. Gupta et al. [6] have categorized the algorithms for temporal data based on the data types after considering

various facets of temporal outlier detection techniques. They have reviewed outlier detection for time series, stream data, and network data.

2.3 Scope of Work

As mentioned above, there exist numerous outlier detection techniques in the literature for different types of inputs, applicable to a wide range of applications. Approaches used in outlier detection techniques for time series data predict future values either by using prediction models, or preserving the normal profile of the system and creating profile based models [6]. The difference between these predicted values and the actual observations defines the outlier score for the points. Then, outliers are identified by comparing the outlier score of the points with a threshold value. A key distinction between these techniques is the specific model that they use. For example, authors in [16] proposed a neural network based technique, to solve the problem of sensor failure detection, identification and accommodation (SFDIA) in flight control systems. Neural networks are used to create analytical redundancy and preserve the normal profile of the system. When the system is not faulty, at the nominal conditions, a gradient descent optimization method is used to minimize the estimated error terms and estimate the parameters of the neural network. Later on, the actual system response is compared to the estimated value from the neural network. If the difference between these two values is greater than a threshold, then the sensor is deemed faulty.

A similar approach is used in [17] in which a data-driven outlier-based monitoring and alerting framework is proposed for detecting unusual patient-management decisions. This framework can either be used as a standalone system or alongside the already existing knowledge-based alerting systems, to help reduce medical errors. The alerting systems are based on a set of rules which are coded representations of clinical knowledge of the experts

in this domain. The proposed approach uses a conditional outlier detection algorithm to identify clinical care outliers based on information of past patient cases. Outliers are defined as unusual actions with respect to the condition of the patient which may lead to patient-management errors. Therefore, the conditional outlier detection algorithm that is implemented with support vector machine tries to identify unusual actions for a given patient based on the information of past patients that had the same or similar conditions. Using segmentation on the stored data, patient states and management actions are defined. An anomaly score is calculated for each action using conditional probability measures. Based on these anomaly scores, an alert score is calculated. Alerts are generated for alert scores that are higher than a predefined threshold. The results show that this approach generates meaningful alerts with the true alert rates ranging from 0.25 for weak alerts to 0.66 for strong alerts.

As opposed to the above techniques, a real-time anomaly detection method is proposed for environmental sensor data in [18] that uses various prediction models. The algorithm is based on an auto-regressive data-driven model. The prediction model is built using four different data-driven methods: nearest cluster, single-layer linear network, naive, and multi-layer perception. For each of these methods, a 10-fold cross-validation technique is used to estimate the model parameters. A prediction interval (PI) is calculated and if the data points fall outside the PI range, they will be marked as outliers. Based on the identified outliers, two strategies are used for processing future data. Either the outlier point is classified as an outlier, or it is replaced by the predicted value and is classified as a normal data. The anomaly detection algorithm has been applied to a wind speed sensor data in which 6% of the data is affected by error. Results show that the MLP detector using the second strategy has the best performance with a false positive rate of 1% and a false negative rate of 2% [18] . Also, in [19] a support vector regression algorithm has been used to create an online novelty detection algorithm for temporal data. The role of the SVR is to build a prediction model for the temporal data.

Instead of using a prediction model, an outlier detection algorithm has been proposed in [20] that uses the correlation between the points that are closer in time to predict the value at each point. Two different variations of the method are presented. The first method uses k data points from the past and k data points from the future, then it takes the median of this window of points as the predicted value. As a result, this method will detect an outlier in point t after k data points delay. In order to have real-time outlier detection algorithm, the second method uses $2k$ points from past data to calculate the median. Both methods will compare the difference between actual data point and the median with an arbitrary threshold.

A category of outlier detection techniques use outlier aware prediction models. These techniques detect outliers based on the predicted values from the prediction model, and use the effect of outliers to estimate the parameters of the prediction model. The concept behind these techniques is the fact that the model parameters can be affected by the existing outliers. The effect of an outlier is related to its type. Prior to reviewing these techniques, we discuss different types of outliers in the next subsection.

2.3.1 Types of Outliers

Early works on outlier detection in statistics assumed that the sample points are independent and normally distributed. In order to have a better understanding of the time series models and detecting outliers, the correlation between samples should be considered as well. In [21], two models have been presented for two types of outliers based on their effects on the neighboring samples. The first type only affects the outlier sample, the second one affects the future sample points as well as the outlier sample. Using these models, the likelihood ratio criteria have been obtained to detect the outliers. Many other works have used the same two models to detect outliers, identify their types or have a better estimate

of the model parameters [22][23][24].

There are four established type of outliers based on their effects. They are Innovational Outliers (IO), Additive Outliers (AO), Level Shifts (LS) and Temporary Changes (TC) [9]. More complex effects can be estimated by a combination of these types. Assuming that the series Y_t is an ARMA process, we have:

$$Y_t = \frac{\theta(B)}{\alpha(B)\phi(B)}a_t, \qquad t = 1, \cdots, n \tag{2.1}$$

where B is the lag operator, a_t is noise term from normal distribution with zero mean and constant variance, n is the number of observations, $\theta(B)$, $\phi(B)$ and $\alpha(B)$ are polynomials of B; all roots of $\theta(B)$ and $\phi(B)$ are outside the unit circle so that the process is both stationary and invertible [7] and all roots of $\alpha(B)$ are on the unit circle which gives the differencing filter [10]. The definition of outliers can be specified as $L(B)I(t_j)$. $I(t_j)$ is an indicator of outlier occurrence. Hence at time $t = j$, $I(t_j)$ is 1 if the observation is an outlier otherwise it is 0. $L(B)$ is a polynomial of B that specifies the effect of the outlier that exist at time $t = j$. $L(B)$ for an IO is:

$$L(B) = \frac{\theta(B)}{\alpha(B)\phi(B)}. \tag{2.2}$$

It can be seen that the effect of an IO is highly dependent to the ARMA model and it should be discussed based on the model. The effect of a TC is defined as:

$$L(B) = \frac{1}{(1 - \delta B)}. \tag{2.3}$$

A TC affects future samples in addition to the sample at time $t = j$, however the effect of this outlier is not permanent. The pace at which the effect will die out in the future observations is specified by δ. Authors in [9] have recommended $\delta = 0.7$ for TC. AOs are special cases of TC where $\delta = 0$, which means that the AOs only affect the sample at time

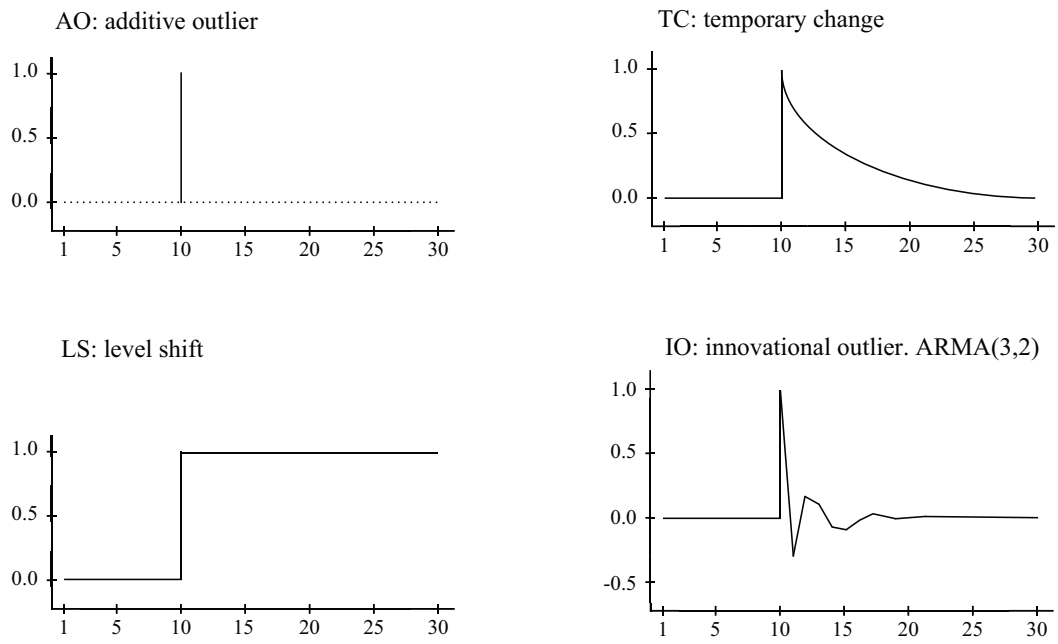


Figure 2.2: A unit impulse at time $t = 10$ as different types of outliers showing their effects, which is adapted from [10]. AO only affects the outlier sample at time $t = 10$, TC affects the outler point and dies out slowly in future samples, LS affects the outlier point and future samples permanently and never dies out, and the effect of IO depends on the model which can also be seen in its formulation.

$t = j$. The effect of an AO is represented as:

$$L(B) = 1. \tag{2.4}$$

LSs are also special cases of TC however they affect future samples permanently; hence $\delta = 1$. LSs have the same effect on the future data that they have on the sample point at $t = j$, and the effect is defined as:

$$L(B) = \frac{1}{(1 - B)}. \tag{2.5}$$

Figure 2.2 shows the effect of different types of outliers when a unit impulse is applied as the outlier. A unit impulse is applied at time $t = 10$ as each type of outlier and its effect is plotted using Equations 2.2 —2.5.

2.3.2 Techniques Based on Outlier Aware Prediction Models

In [8], an outlier aware prediction model is used for estimating time series parameters in the presence of outliers. Using ARIMA modeling to represent the data, a likelihood ratio is defined to find outliers and their types. The impacts of two different types of additive and innovational outliers are considered to create three hypotheses. Either no outliers exists at time t, there is an AO at time t or there is an IO at time t. Testing these hypotheses versus each other creates three different statistics that can be used to detect outliers and their types. If it is decided that the sample point at time t is an outlier, it is classified as IO if the statistics based on the impact of IO is greater than AO, otherwise it is classified as AO. Then the impact of the outlier are removed from the residuals and the model parameters are estimated using the modified residuals. The performance of the algorithm is examined using experiments with one, two and three outliers in the time series data. Additionally, the impact of number of samples in the time series data and the critical value are examined. The detection accuracy using 3.5 as the critical value ranges from 89.6% to 98.8% when one outlier exists in the data, and ranges from 79.2% to 95.2% for the case when two outliers exist in the time series data.

A joint estimation of model parameters and outlier effects is proposed in [9], which uses all outlier types explained in the previous section and models the time series data with an ARMA process. Using this model, an iterative algorithm identifies the location and type of outliers based on the residuals and test statistics for each type. Subsequently, the effect of outliers are removed from data, and model parameters are estimated again for adjusted time

series data. It is important to re-estimate the model parameters since they can be biased due to the existence of outliers. These new model parameters help in finding new outliers in the following iterations or in further analysis of time series data. Their experiments show that for the critical value of 3.0, the detection accuracy for outliers of size 4 and 5 is between 85% and 99%. This algorithm is widely used in the industry, and is implemented in a time series outlier detection package in R language, called"tsoutliers". Therefore, we used a simplified version of this technique for our hardware implementation. The complete algorithm and the reductions to practice are discussed in the next chapter and the effects of these simplification on the performance of the algorithm is discussed in chapter 5.

2.3.3 Managing Detected Outliers

The final part of an outlier detection technique is handling the identified outliers which sometimes is regarded as cleaning the data or outlier mitigation. Managing outliers requires the knowledge about their source or origin. As mentioned before, outliers often indicate an unusual event. In such cases, discovering the specific event that is causing these outliers is the main concern. In situations where outliers are results of errors or noise artifacts, and have influence on other analyses of data, it is necessary to clean the data for further use.

The common approach in techniques that use a prediction model to calculate predicted values is to replace the outlier point with the predicted value. In [18] two different strategies are proposed to detect outliers, one employs both past and future neighboring values while the other just uses past values. In both methods outliers are replaced by the predicted values to clean the data.

In [18], following the detection of outliers, two different strategies are compared for future processing of the data. In the first strategy which is called anomaly detection(AD),

detected outliers are applied to four different data-driven models (nearest cluster, single-layer linear network, naive and multi-layer perception) as outlier inputs. In the second strategy, they are replaced by the predicted value from the data-driven model and the clean data is the input for the models. This is called anomaly detection and mitigation(ADAM). Both strategies are successful in detecting a great number of anomalies but the performance of LN and MLP-based classifiers is enhanced by employing the ADAM strategy [18].

In conclusion, following the detection of outliers, it is important to remove their effect for further analysis of data. Simple methods for cleaning the data involves substituting outliers with more reasonable values such as mean or median of the data, a local mean or median, or the predicted value for that specific observation. Another possibility is to simply remove the anomalous point from the data. In this thesis, we only detect the outlier sample points and leave the decision of cleaning the data to the users.

2.4 FPGA-based Outlier Detection Techniques

Recently, several architectures have been proposed for network intrusion detection systems (NIDS)[5][25]. NIDS is a system which monitors a network for malicious activities, and reports them to the administrator. In [5], an FPGA-based network intrusion detection architecture is presented. The proposed FPGA implementation takes advantage of pipelining and parallelism and consists of two major components; 1)A feature extraction module (FEM), which extracts the network header data, and represents the state of the network based on its behaviour; 2) An Outlier detection component which is based on principal component analysis (PCA). PCA represents normal data as sets of axes in the training phase. Then, it maps the live network data to these axes and calculates the distance from them. Using the Mahalanobis distance as an outlierness metric, it determines attacks by comparing the distance to a predefined threshold. The Mahalanobis distance is a distance measure

between a sample point and the distribution of the data, which takes into account the correlation in the data by using the inverse of covariance matrix [26]. The detection algorithm detects both time based and connection based attacks, and its detection rate is greater than 99% while its false positive rate is 1.95%. Their target FPGA is from Xilinx Pro II family. The throughput for the FEM component is 21.25Gb/s and for the outlier detection component, it is 23.76Gb/s. An FPGA-based programmable embedded intrusion detection is proposed in [25]. The design has a packet summary and normalization unit which extracts and normalizes features including, protocol type, packet size, source and destination port numbers. Then, the normalized packet summary is used in the packet classification unit. The classification algorithm is based on Bayes theorem which is also known as naive Bayesian. The intrusion detection system is designed on a system on a programmable chip using the Altera Cyclone III family of FPGAs. The performance of the algorithm is analyzed using the confusion matrix in the training phase and its accuracy is 78% with the false positive rate of 3.9%. However, the detection accuracy and the false positive rate of the FPGA-based intrusion detection in [5] is by far better.

Mahalanobis distance based outlier detection techniques are also implemented on FPGA-based network interface cards (NIC) [27][28]. Outlier detection is used to perform information filtering in NIC. Information filtering is an approach to reduce the storage demand, in which only outlier samples are stored and the normal samples are discarded. The idea is to move the outlier detection from the application layer to the NIC. In [27], a fully pipelined outlier detection technique, based on mahalanobis distance, is implemented on a Xilinx Virtex VI FPGA family in which the learning phase and the calculation of covariance matrix are implemented in the application layer, and only the outlier detection is moved to the FPGA. Since the application layer doesn't have any information about normal and abnormal samples, it needs to read the buffer from the NIC periodically. The throughput of the design is 14 million samples-per-second compared to the case without outlier detection at NIC that processes 800 thousand samples-per-second, and for samples with 5 features,

the design requires 120 DSP units. The authors of [28], have proposed an FPGA implementation of the same algorithm as [27]. However, they moved all the calculations to the hardware. The design consists of a FIFO, a matrix inversion component and an outlier detection hardware. Two versions of the design have been implemented on Xilinx Virtex VII family FPGAs. One implementation is optimized to reduce the total number of clock cycles which employs 1660 DSP units. The other implementation is optimized for critical path latency and uses 1191 DSP units. The authors claim that the hardware is 37 times faster than the software implementation of the outlier detection algorithm. Although the number of required DSP units increased greatly to implement all the calculations within the hardware, but the throughput is improved by 64.7% and 84.23% for the two implementations compared to the FPGA implementation proposed in [27].

In [29], authors implemented two outlier detection techniques based on local outlier factor (LOF) and k-nearest neighbour using data set cache approach on an FPGA-based NIC. A data set cache on the FPGA retains the portion of the reference data that is accessed frequently. The resources required for hardware implementation are proportional to the size of the cache, for example the DSP units required for different sizes of the cache are between 1024 to 2070. Outlier detection is done in hardware using the data set cache that stores a portion of the data set, and the identified outliers are sent to the NIC for outlier detection since the data set in hardware is incomplete. Since the true negative ratio (i.e., hit ratio) is between 45% to 90% based on the simulation results and only identified outliers are sent to the NIC, the workload of the host is reduced. Also, the throughput is improved by 1.82 to 10 times compared to the NIC without outlier detection.

An FPGA-based spectral anomaly detection system is proposed in [30]. A new online algorithm is proposed for power spectra computation which is based on DFT. The anomaly detection technique uses symbolic aggregate approximation (SAX) to generate the symbolized version of the time series. Then, bitmaps are constructed using the symbolized

Table 2.1: Summary of the performance of the state of the art FPGA implementations of outlier detection techniques.

	Accuracy	**FP**	**TN**	**Throughput**	**Power**	**DSP**
From[5]	99%	1.95%	NA	23.76Gb/s	NA	NA
From[25]	78%	3.9%	NA	NA	NA	NA
From[27]	NA	NA	NA	14Ms/s	NA	120
From[28]	NA	NA	NA	39.7 & 88.8 Ms/s	NA	1191 & 1660
From[29]	NA	NA	45%-90%	1.82x-10x	NA	1024-2070
From[30]	NA	NA	NA	1x	0.3W(5x)	14

time series, and the anomaly score is calculated based on the frequency counts of the sequences reported in bitmaps. The algorithm with only one frequency channel is implemented on a Xilinx Virtex VII FPGA family which requires 14 DSP units. The FPGA implementation has a power consumption of 0.3W which compared to the C implementation of the algorithm on a 1.6GH Intel Core i5 processor is 5 times more power efficient and 4.25 times more energy efficient[30].

There has been a growing interest in hardware implementation of outier detection techniques in the recent decade. Table 2.1 summarizes the performance of all the FPGA implementations explained above in terms of speed, power, accuracy and area for further comparisons. However, most of the existing FPGA implementations are designed for NICs or only for network intrusion detection. Only in [30], an FPGA implementation is proposed that can be used in different applications such as machine prognostic and wearable electronics. This FPGA implementation reduces the power consumption compared to its software implementation counterparts. However, the detection accuracy or false positive rate of the implementation are not evaluated at all. The motivation for this thesis is the need to detect outliers in application domains that require power efficient and real-time detection of outliers such as in wearable devices that are battery powered and server farms that can take advantage of a power efficient accelerator than can run a specific task such as outlier detection.

Therefore, the goal is an FPGA implementation that is more power efficient compared to the software implementations that has a comparable accuracy to these implementations.

Chapter 3

Theory and Reduction to Practice

The outlier detection technique used in this thesis is a simplified version of the algorithm called joint estimation of model parameters and outlier effect, which is proposed in [9]. This algorithm models the time series data with an ARMA process and then calculates the model parameters and the residuals. Using these calculations, it defines test statistics to determine outlier points. In this chapter, the theory of ARMA modeling and the statistical concepts behind the outlier detection algorithm are discussed. Additionally, the simplifications that are required for the hardware implementation and the simplified algorithm are presented.

First, we will explain the three stages required for modeling a time series with an ARMA process along with the simplifications needed to implement this procedure in hardware. Then, we will discuss the test statistics and the critical value. In the last section, we will present the details of the joint estimation of model parameters and outlier effects and the simplified version of this algorithm that is used for hardware implementation in this thesis.

3.1 ARMA Modeling

In many domains, the behaviour of physical phenomena are described by deriving mathematical models based on empirical laws. The models that can predict the exact behaviour of a process are called deterministic. Many phenomena in real world have some unidentified or uncontrolled elements which have direct impact on their behaviour. Thus, it is impossible to describe them with deterministic models. Alternatively, stochastic models can be an appropriate way to describe these phenomenon [7].

A set of observations from a process can demonstrate its reaction to all the unknown elements over time. A model that is derived from these observations can predict the probability of future data falling in a specific range or a prediction interval. These models are also called probability models since they cannot anticipate the exact behaviour of the process.

The autoregressive and moving-average models are widely used stochastic models. Autoregressive (AR) models describe the process based on its own previous values. Moving-average (MA) models indicate that the output has a linear relationship with the current and previous error terms. Autoregressive-moving-average (ARMA) models are composed of both the autoregressive and moving-average models.

A common practice to model a time series with an ARMA process is to use Box-Jenkins method [7]. Box and Jenkins have stated that in analyzing a time series, the time series will be regarded as a realization of a stochastic process [7]. If the joint probability distribution of this process for any number of observations stays the same when it is shifted in time, the process is in a state of statistical equilibrium and therefore called stationary. In other words, a time series which is generated from a process with constant mean and variance over time is stationary.

The stationarity of the time series is necessary in its analysis using ARMA models.

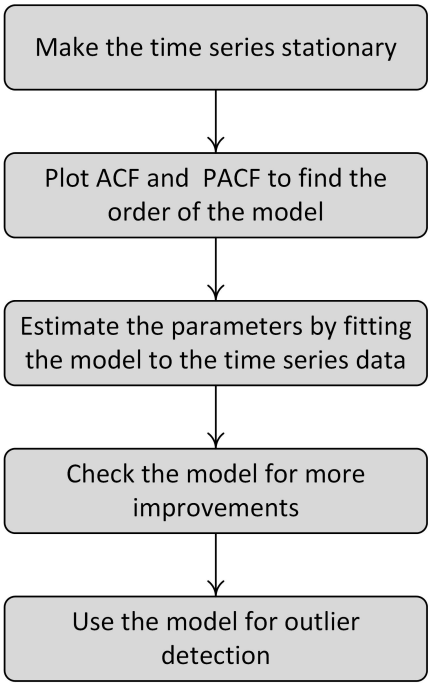


Figure 3.1: Flow of ARMA Modeling for outlier detection task.

It confirms that the properties of the process will remain the same, and implies that future values are predictable with the model. Nevertheless, most of the time series from physical processes exhibit non-stationary behaviour. Although these time series reflect some sort of trend in their nature, it is possible that the difference between two consecutive observations manifests stationarity. In such cases, instead of the original data, we will model the differenced time series. Based on this model, we can predict future difference values and therefore analyze the original time series. In this thesis, we assume that the input time series data is stationary.

The Box and Jenkins method has three stages, namely model selection (i.e., order identification), parameter estimation, and model checking to find the ARMA model that best fits the data.Then, this model can be used for time series data analysis. Figure 3.1 shows the flow of the steps for ARMA modeling. In the next sections, we will explain the details of these stages and the simplifications required for the hardware implementation of each one.

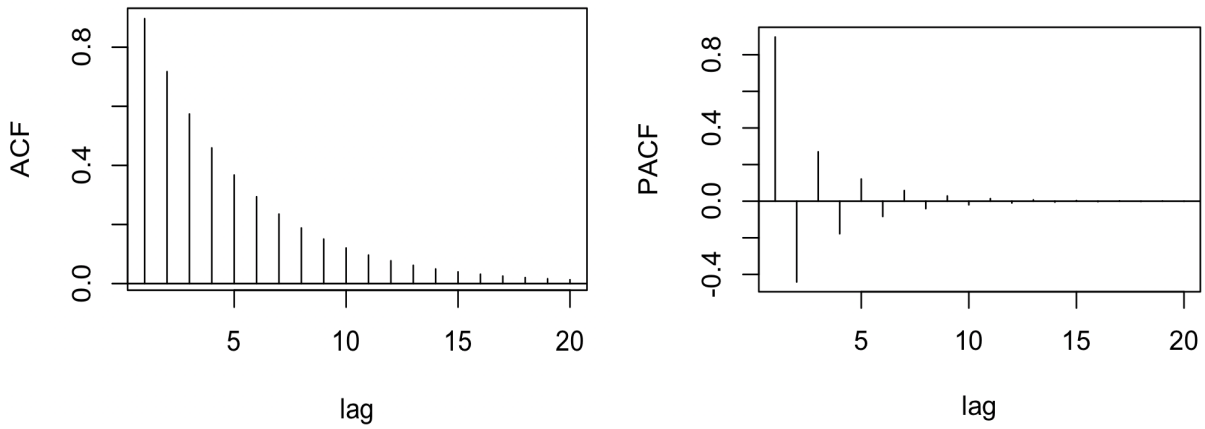
3.1.1 Model Selection

Selecting an ARMA model requires identifying orders of both AR and MA operators and approximating the parameters based on these orders. Finding the desirable model involves an iterative procedure of examining different models and obtaining initial estimates of their coefficients, and evaluating how well the model fits the time series data. The principle of parsimony [31] specifies that the simplest model that explains the data with adequate number of parameters is preferred [32].

Determining the order of AR and MA operators is achievable by inspecting the relationship between observations. Autocorrelation functions (ACF) and partial autocorrelation functions (PACF) depict the correlation of observations that are k lags apart. Hence, the order will be identified by inferring the similarities of observations using ACF and PACF plots. Since a mathematical representation from data does not exist, graphical methods must be used to obtain the model [7].

In the plot of correlation coefficients for different lags, a confidence interval will be defined. If the correlation coefficient at any lag falls outside that interval, it indicates that a term for that lag should exist in the model and the initial parameter is equal to the value of the correlation coefficient. Furthermore, attributes of the ACF and PACF plots are different for each type of ARMA model and can determine the type of the model that needs to be used. For instance, Figure 3.2 shows both ACF and PACF for an ARMA (1,1) process.

The characteristics of ACF and PACF for different ARMA models is summarized in Table 3.1. For an AR model of order p, the autocorrelation function tails off while the partial autocorrelation function cuts off after lag p. Inversely for a MA model of order q, the partial autocorrelation function tails off and autocorrelation function cuts off after lag q. In the case of an ARMA model of order (p, q), the autocorrelation function tails off while containing exponential and damped sine waves after $q-p$ lags. The partial autocorrelation function also



(a) Autocorrelation function (b) Partial autocorrelation function

Figure 3.2: Autocorrelation and partial autocorrelation functions for an ARMA(1,1), where the coefficients are 0.8 and -0.7 for AR and MA respectively (both functions tail off).

Table 3.1: Characteristics of ACF and PACF for different ARMA models adapted from [7] [33].

	$AR(p)$	$MA(q)$	$ARMA(p,q)$
ACF	Infinite(exponential and/or damped sine waves) Tails off	Finite Cuts off after lag q	Infinite(exponential and/or damped sine waves after $q-p$ lags) Tails off
PACF	Finite Cuts off after lag p	Infinite(exponential and/or damped sine waves) Tails off	Infinite(exponential and/or damped sine waves after $p-q$ lags) Tails off

tails off and contains exponential and damped sine waves after $p-q$ lags [7]. As a result, the desirable model can be determined based on the correlations between observations shown in ACF and PACF plots and their shapes. It is evident that this model cannot represent the time series data perfectly. However, it is the best fit for the data.

3.1.1.1 Model Selection Challenges for Hardware Implementation

Since model selection relies more on graphical methods than mathematical ones, it is difficult to find a desirable model with pure mathematics [9]. The existing implementations

of the procedure in software (e.g., auto.arima function in the forecast package of R [34] or the ARMA class in the Statsmodels package of Python [35]) interact with the user through warnings (e.g., letting the user know that assumptions like stationarity or normality don't apply, or asking for more information on how many iterations are needed for fitting the selected model to the data) to compensate for the lack of graphical methods in the implementation. In other words, these packages rely on the user to infer important information from the original data.

Model selection requires evaluating models with different orders to find the acceptable model. Therefore, the implementation of this stage involves exploring many possible models, and selecting the acceptable model after fitting all of them to the data which can be computationally exhaustive. This imposes an additional challenge for the hardware implementation of the model selection stage. From the hardware design point of view, we need to allocate enough resources for the design; therefore, the hardware must be designed for the model with the highest order to cover all the possible models. If the best model is simpler, the required resources will be less. Although this approach is flexible for model selection, the area of the circuit for most of the models is not optimized.

All of these challenges led us to choose a fixed order for the model, and fitting that model to the data for estimating the parameters. The authors in [7, p. 75] indicated that often the principle of parsimony [31] in parameters is achievable by using ARMA models instead of just AR or MA models. It is known that a finite MA model can be represented by an infinite AR model and vice versa [7, p. 75]. However, representing a time series with an infinite model instead of a finite one is not parsimonious. Therefore, it is more reasonable to use an ARMA model as the fixed model rather than AR or MA models. The model chosen for the outlier detection hardware is ARMA(1,1). As it will be discussed in Chapter 5, our experiments show that this model is acceptable and results in a reasonable hardware complexity. Once the model is selected, the parameters of the model should be estimated in

the next stage.

3.1.2 Parameter Estimation

Having identified the best order for the model, it is crucial to properly use the data to estimate the parameters of the model and fit this model to the time series data. The general approach is to use a maximum likelihood estimation or least-squares approximation [7, p. 210].

Suppose that a set of observations generated from an ARMA(p, q) model is used to create a random variable. The probability distribution of the random variable depends on the unknown parameters of the ARMA model. The likelihood function of parameters can be defined for a particular value of the random variable. It determines the probability of different values for the unknown parameters based on a fixed value of the random variable. The likelihood principle indicates that the likelihood function incorporates all the information that the time series data can reveal about the unknown parameters [7, p. 210]. Therefore, it is widely used to estimate the model parameters. The values of the parameters that maximize the function are called maximum likelihood (ML) estimates [7, p. 210].

The ARMA(p, q) can be written as:

$$a_t = z_t - \phi_1 z_{t-1} - \phi_2 z_{t-2} - \ldots - \phi_p z_{t-p} + \theta_1 a_{t-1} + \theta_2 a_{t-2} + \ldots + \theta_q a_{t-q}, \tag{3.1}$$

where z_t's are current and previous sample points, θ's and ϕ's are coefficients that need to be estimated, and a_t's are white noise with a normal distribution with zero mean and constant variance σ^2. The probability density for a_t's is [7, p. 211]:

$$p(a_1, a_2, \ldots, a_n) \propto ({\sigma_a}^2)^{-n/2} \exp[-\sum_{t=1}^{n} \frac{{a_t}^2}{2{\sigma_a}^2}], \tag{3.2}$$

for example

$$p(a_1) = (2\pi {\sigma_a}^2)^{-1/2} \exp[-\frac{{a_1}^2}{2{\sigma_a}^2}]. \tag{3.3}$$

If p values of z_t's and q values of a_t's prior to start of z_t are known, the values of a_t's can be calculated for different values of the model parameters. Assuming z_* and a_* are vectors of the initial values, the log-likelihood of the parameters ϕ, θ and σ_a^2 can be defined as [7, p. 211]:

$$l_*(\phi, \theta, \sigma_a^2) = -\frac{n}{2} \ln(\sigma_a^2) - \frac{S_*(\boldsymbol{\phi}, \boldsymbol{\theta})}{2\sigma_a^2}, \tag{3.4}$$

where

$$S_*(\boldsymbol{\phi}, \boldsymbol{\theta}) = \sum_{t=1}^{n} {a_t}^2(\phi, \theta | z_*, a_*, z). \tag{3.5}$$

It should be noted that $\boldsymbol{\phi}$ and $\boldsymbol{\theta}$ are vectors containing p and q coefficients, respectively. Also, the asterisk subscript designates that both Equation 3.2 and Equation 3.4 are under the assumptions that the initial values are known [7].

The original time series data is only incorporated in the log-likelihood through the sum of squares. Hence, maximizing the log-likelihood can be achieved by minimizing the sum of squares. Therefore, the maximum likelihood estimates have the same behaviour as the least-square estimates [7, p. 211]. Depending on the choice of initial values z_* and a_*, there are two different approaches of conditional and unconditional calculations. The unconditional likelihood is the original technique to determine the parameters in which the initial values must be calculated from the original data. Using a backward-looking process we can calculate the initial values first and then try to measure noise terms. However, when the number of observations in the time series is moderately large[1], initial values can be replaced by appropriate approximations. This approach is called conditional calculations [7, p. 211]. A common practice is to set the initial values equal to their unconditional expectations.

[1]The effect of sample size n on the expected value of the conditional sum of squares for MA(1) model is examined in [36]. Osburn have used sample sizes in the range of 20 to 200. The result show that if the time series data is invertible the effect is negligible. In this thesis we assumed that the time series data is both stationary and invertible.

Thus, the initial noise terms will be equal to zero.

3.1.2.1 Parameter Estimation Simplifications for Hardware Implementation

As discussed in Section 3.1.1.1, in this thesis we assume that the model is ARMA$(1, 1)$; therefore the model can be defined as:

$$a_t = z_t - \phi z_{t-1} + \theta a_{t-1}, \tag{3.6}$$

where ϕ and θ are the parameters that must be approximated using the original data, and a_t and a_{t-1} are current and previous noise terms. Estimating ϕ and θ requires minimizing the sum of squares for the noise terms given in Equation 3.5.

The initial values, a_{t-1} and z_{t-1} are necessary to calculate the values of a_t's. For a conditional least square estimate, we substitute the initial states with their expected values. Therefore, a_{t-1} will be considered zero and z_{t-1} is also equal to the mean of the samples [7, p. 212]. An optimization algorithm determines the values of coefficients while minimizing the sum of squares of noise terms. Therefore, the objective function to minimize is:

$$S(\phi, \theta) = \sum_{t=1}^{n} {a_t}^2(\phi, \theta | z_{t-1}, a_{t-1}, z), \tag{3.7}$$

where n is the number of samples in the time series data.

3.1.2.2 Optimization Algorithms

As mentioned above, it is a common approach to approximate the log-likelihood with a least square estimate. In order to solve the non-linear least square estimation to determine the model parameters, two different types of iterative algorithms can be employed

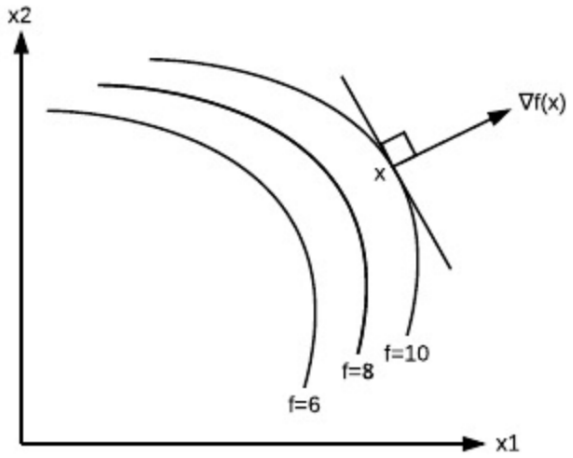


Figure 3.3: A gradient vector is used to find the "improving direction" in a maximization problem for the steepest descent method; adapted from [38].

[37]:

- Derivative based methods which require the objective function and its derivatives to check the optimality conditions. Two main categories of these methods are gradient-based and Newton-based methods. The gradient-based methods, such as steepest descent and conjugate gradient, are iterative algorithms that use the first derivative of the objective function to search for the improving direction. Subsequently, line search techniques determine how far to move in that direction. Figure 3.3 shows how the gradient vector helps in finding the improving direction for a maximization problem. The Newton method itself is not a robust approach to minimization but the variations of it, that use the same concept, are powerful methods [38]. These algorithms require second order derivatives of the objective function or the Hessian matrix in addition to first order derivatives. Gauss-Newton and Levenberg-Marquardt are commonly used methods. Similar to gradient-based methods, these algorithms employ a line search technique, however they converge at a faster rate [38].

- Derivative free methods, which depend only on evaluation of the objective function

itself for variation of parameters to find the optimum. These methods are called direct or zero order methods [38]. Since they don't require the gradient or Hessian information, they are convenient when this information is not accessible. The Hooke and Jeeves method, the Rosenbrock search method, and the simplex method of Nelder and Mead are some of the commonly used direct methods.

The concept behind all of these algorithms is similar. A starting point is received and flagged as the current point. Then, using different criteria, the search for an improving direction begins. When the direction is found, a new point is obtained by moving along the direction. The next iteration uses this new point as the current value. Additionally, all of these algorithms have a stopping strategy, either reaching maximum iterations or reaching the predefined tolerance. The selected algorithm for implementation in this thesis is the direct method. Hooke and Jeeves introduced the direct method in [39] and have indicated: "We use the phrase "direct search" to describe sequential examination of trial solutions involving comparison of each trial solution with the "best" obtained up to that time together with a strategy for determining (as a function of earlier results) what the next trial solution will be. The phrase implies our preference, based on experience, for straightforward search strategies which employ no techniques of classical analysis except where there is a demonstrable advantage in doing so."
which covers the concept behind their method. Their strategy for determining the next probable solution is discussed in Section 3.1.2.3. Hooke and Jeeves have specified a list of reasons that motivated them to examine these methods, such as:

- Direct methods can converge to a solution for some problems that classical methods cannot solve. Additionally, they take less time to converge for problems that both categories can resolve.

- In all stages, the algorithms give an approximate solution while trying to optimize it

even more.

- They consist of simple arithmetic operations that require simple logic; therefore the implementation is straightforward and flexible with electronic computers.

The reasons mentioned above are still valid today, although the classical methods also have been improved and can solve a wider range of problems. The authors in [40] have explored the reasons why direct methods are commonly used after the improvements to classical algorithms. They have specified three reasons regarding the efficiency of direct methods compared to quasi-newton method:1) although many direct search methods use heuristics, they assure global convergence, and they cover a wider range of nonlinear optimization problems; 2) they can be easily implemented due to their simplicity while quasi-newton methods require calculating the derivatives [40]; 3) the direct methods are preferred over gradient-based and newton-based methods due to their robustness and simplicity [40]. The Hooke and Jeeves method is our preferred method for minimizing the sum of squares for the same reasons explained above. Also, it is specified that this method implicitly depends on classical analysis [40]. The fact that it does not require derivatives of an objective function reduces its computational complexity and even makes it a better candidate for hardware implementation.

3.1.2.3 Hooke and Jeeves Pattern Search Algorithm

The Hooke and Jeeves algorithm requires a starting point and an initial perturbation vector $\boldsymbol{P} = (\Delta x_1, \Delta x_2, \ldots, \Delta x_n)$. Also, for a stopping strategy, it needs a predefined perturbation tolerance. Pattern search algorithms also known as direct search consist of two different types of moves, exploratory search and pattern move. Prior to explaining the details of the algorithm, these two types of moves are discussed.

Exploratory Search: For each variable x_j of the current point $\boldsymbol{X}$ in turn, evaluate the objective function with $x_j + \Delta x_j$. If the objective function is improved compared to the old objective function value, then keep $x_j + \Delta x_j$ as the jth element in the new point $\boldsymbol{X\prime}$ and update the old objective function value. Otherwise, change x_j to $x_j - \Delta x_j$ and evaluate the objective function again. If the value of objective function is not improved by either $x_j + \Delta x_j$ or $x_j - \Delta x_j$, keep the x_j and evaluate the objective function by changing the next element x_{j+1}. After examining the changes for all the elements in $\boldsymbol{X}$, if the exploratory search was successful the new point $\boldsymbol{X\prime}$ gives a pattern direction by $\boldsymbol{X\prime} - \boldsymbol{X}$ where both $\boldsymbol{X\prime}$ and $\boldsymbol{X}$ are vectors of the variables. Otherwise, search for the new point $\boldsymbol{X\prime}$ using modified perturbation vector [41] [38].

Pattern Search: Following a successful exploratory search, a move from the initial point is done through the $\boldsymbol{X\prime}$ to a tentative point.

$$\boldsymbol{X_t} = \boldsymbol{X} + a(\boldsymbol{X\prime} - \boldsymbol{X}), \tag{3.8}$$

where $\boldsymbol{X_t}$ is the tentative point and a is an acceleration factor which ranges between 0.5 and 2 [38]. Also, $\boldsymbol{X\prime} - \boldsymbol{X}$ is the pattern direction [38].

Complete Algorithm There are three main steps [38]:

Initialize: Setting all the inputs: perturbation vector $\boldsymbol{P}$, perturbation tolerance $\boldsymbol{T}$, acceleration factor a and starting point $\boldsymbol{X}$.

Start/Restart: Look for a pattern direction by finding the new point $\boldsymbol{X\prime}$ that improves the objective function, using exploratory search around $\boldsymbol{X}$. If there is no new point $\boldsymbol{X\prime}$ that improves the objective function compared to $\boldsymbol{X}$, modify the perturbation vector to half of its current size. If the perturbation vector $\boldsymbol{P}$ is less than the perturbation

tolerance $\boldsymbol{T}$, terminate the process with $\boldsymbol{X}$ as the optimized solution. Otherwise, go to restart. If the exploratory search was successful and a new point $\boldsymbol{X\prime}$ is found, reset the perturbation vector and go to pattern move.

Pattern Move: Calculate the tentative $\boldsymbol{X_t}$ using the pattern direction $\boldsymbol{X\prime - X}$. Conduct an exploratory search around $\boldsymbol{X_t}$. If the new point $\boldsymbol{X_t\prime}$ doesn't improve the objective function, update the current point with the best solution so far $\boldsymbol{X = X\prime}$ and go to restart. Otherwise, the pattern move is successful, update the current point $\boldsymbol{X = X_t}$ and $\boldsymbol{X\prime = X_t\prime}$ and go to pattern move.

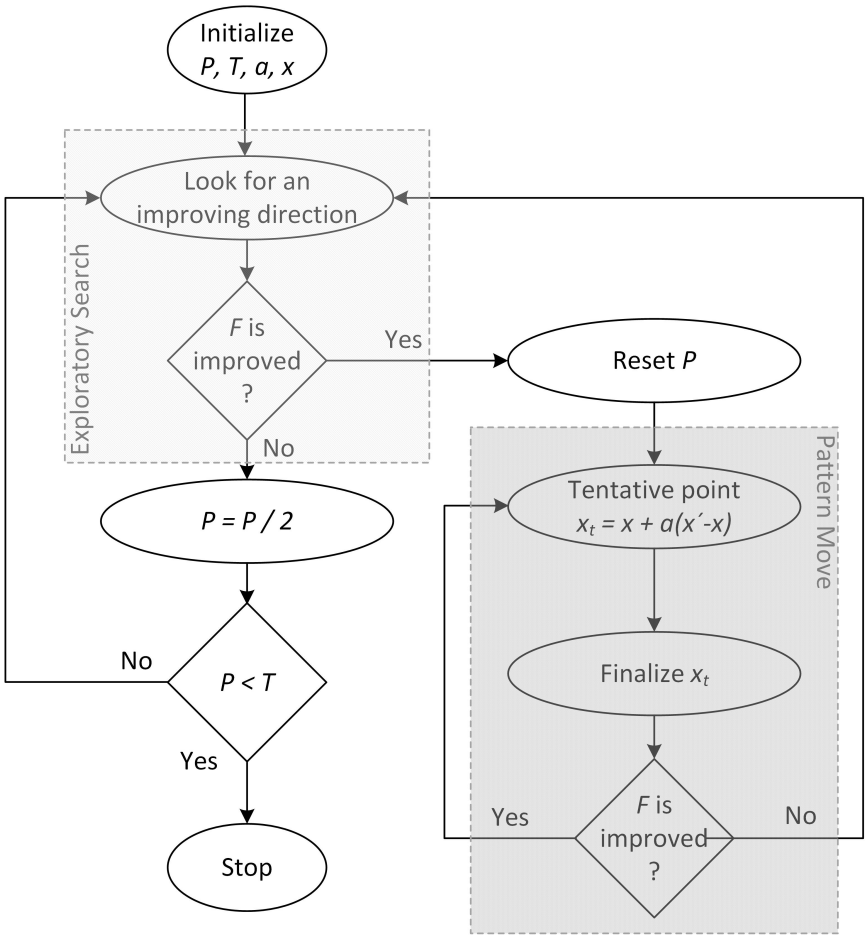


Figure 3.4: The flowchart of the Hooke and Jeeves algorithm which uses exploratory search and pattern move. For each new pattern direction, a pattern move gives the next current point.

Figure 3.4 summarizes the algorithm explained above in a flowchart. As it can be seen, pattern search is an iterative procedure which consists of two tasks, exploratory search and pattern move. In exploratory search, we look for a pattern direction and for each new direction we execute a pattern move. The search for pattern direction and moving along that direction continues until no improving directions are found, in which case we try to find a new direction with modified perturbation vector in the next exploratory search. Finally, the procedure ends when the perturbation vector is smaller than the predefined perturbation tolerance. The direct search algorithm has been implemented in MATLAB [38] and Mathematica [41].

3.1.2.4 Complexity of Direct Search and its Challenges for Hardware Implementation

Although the direct search method is simpler than the other optimization algorithms as discussed in 3.1.2.2, its complexity varies depending on the add/multiplication calculations required for the objective function. Based on the objective function defined in Equation 3.7, the complexity of the algorithm is proportional to the number of samples in the time series data n. Therefore, there is a trade-off between the number of samples and the area of the hardware.

Direct search method is an iterative procedure and the number of iterations varies for each problem depending on the starting point, perturbation vector, and the precision required for the solution. However, for hardware implementation this number must be fixed. The original algorithm stops iterating when the perturbation vector is less than the perturbation tolerance. However, one of the characteristics of this algorithm is that at each iteration the current point is the optimized solution seen so far [39]. Taking advantage of this characteristic, we modified the condition which stops the procedure. Instead of the original stopping condition we used a maximum number of iterations which assures that we

have a solution that is optimized up to this iteration. Using a maximum number of iterations makes it possible to implement this algorithm in hardware. The hardware implementation of the modified algorithm is discussed in the next chapter.

3.1.3 Model Checking

It is evident that the fitted model is just an estimation of the original data and is not a perfect representation. Therefore the fitted model needs to be evaluated for possible improvements. There are tests that can analyze the adequacy of the fitted model. For instance, overfitting is a technique in which a more detailed model will be fitted to the data and is compared to the original one. The detailed model has more parameters and is more complex; hence, it can describe the data more accurately. However, if the analysis confirms that the fitted model is satisfactory without the added complexity, the simpler model is preferred. Otherwise, the fitted model should be modified. However, the problem with overfitting is that it needs knowledge about how to define a more complete model.

Another method for checking the fitted model involves evaluating the residuals. Residuals are the noise terms (a_t's) calculated from the fitted model. The autocorrelation of the residuals can include important information on the adequacy of the fitted model. If the results of these analyses indicate that the model is not adequate, we need to start from the model selection stage and determine a new model. In this thesis, Since we assumed that the ARMA(1,1) is the best model that represents the data, we do not implement this stage.

3.2 Test Statistics

A test statistic is an assessment of a specific attribute of a sample point that can be used to test a statistical hypothesis. The test statistic represents the behavior of data

set with a quantity, which makes the hypothesis testing easier. In general, the test statistic is used to decide whether to accept or reject the null hypothesis in favor of an alternative one. For example, to determine whether the mean of a population is greater than μ_0, the null hypothesis is $H_0 : \mu = \mu_0$. The test statistic to test the null hypothesis is:

$$t = \frac{\bar{x} - \mu_0}{s/\sqrt{n}}, \tag{3.9}$$

where $\bar{x}$ is the mean of a sample of size n and s is the sample standard deviation [42].

In outlier detection, test statistics help to distinguish if a sample point is an outlier or not. The test statistic is compared to a point from a test distribution which is called the critical value. If the absolute value of the test statistic is greater than the critical value, the null hypothesis is rejected due to statistical significance. For instance in [8], test statistics based on the effect of outliers are used to test three hypotheses which results in identifying the type and location of outliers. The same approach is used in the outlier detection technique that is selected for hardware implementation in this thesis.

3.2.1 Critical Value

An important decision in outlier detection is how significantly an observation must deviate from the rest of observations to be considered as an outlier. This decision will lead to designation of an acceptable region by defining a critical value.

The authors in [9] have studied the sampling behavior of the test statistics, and realized that the number of samples in the data has an influence on the test statistics. These test statistics are compared to the critical value to detect outliers; therefore, the critical value should be defined considering the number of observations in the data. Using different sample sizes, the authors in [9] have done simulations, and suggested a guideline for

Table 3.2: Guideline for defining critical value based on the number of samples suggested in [9].

Critical Value	**Number of Samples**
Between 2.5 and 2.9	Less than 100
3.0	Between 100 and 200
Greater than 3.0	Over 200

defining the critical value which is shown in Table 3.2. They have analyzed the influence of different choices of critical value on the performance of their technique and concluded that the power of correct detection of both type and location of an outlier decreases when the critical value increases. The relation between the critical value and the type I errors in the detection procedure which is the average number of misidentified outliers, is also the same. In addition,they observed that the possibility of misidentifying the location of an outlier by one time period is higher when the critical value is too low. Therefore, it is important to define an appropriate critical value.

3.3 Outlier Detection Algorithm

An iterative procedure is proposed for joint estimation of model parameters and outlier effects in [9]. The idea is to use the known location and effect of outliers to adjust the model parameters, and then use the adjusted parameters to detect outliers and estimate their effects. The effectiveness of this method in outlier detection is due to the consideration of the effect of outliers on the model parameters. Accounting for biased parameters in the presence of outliers helps in dealing with masked and spurious outliers.

Let Y be a time series following a general ARMA process,

$$Y_t = \frac{\theta(B)}{\alpha(B)\phi(B)}a_t, \qquad t = 1, \cdots, n \tag{3.10}$$

where n is the number of samples in the time series; $\theta(B)$, $\phi(B)$ and $\alpha(B)$ are polynomials of B; all roots of $\theta(B)$ and $\phi(B)$ are outside the unit circle; and all roots of $\alpha(B)$ are on the unit circle [9]. For example, Figure 3.5 depicts an ARMA(2,1) process with the AR coefficients of 0.6 and -0.3 and the MA coefficient of -0.4.

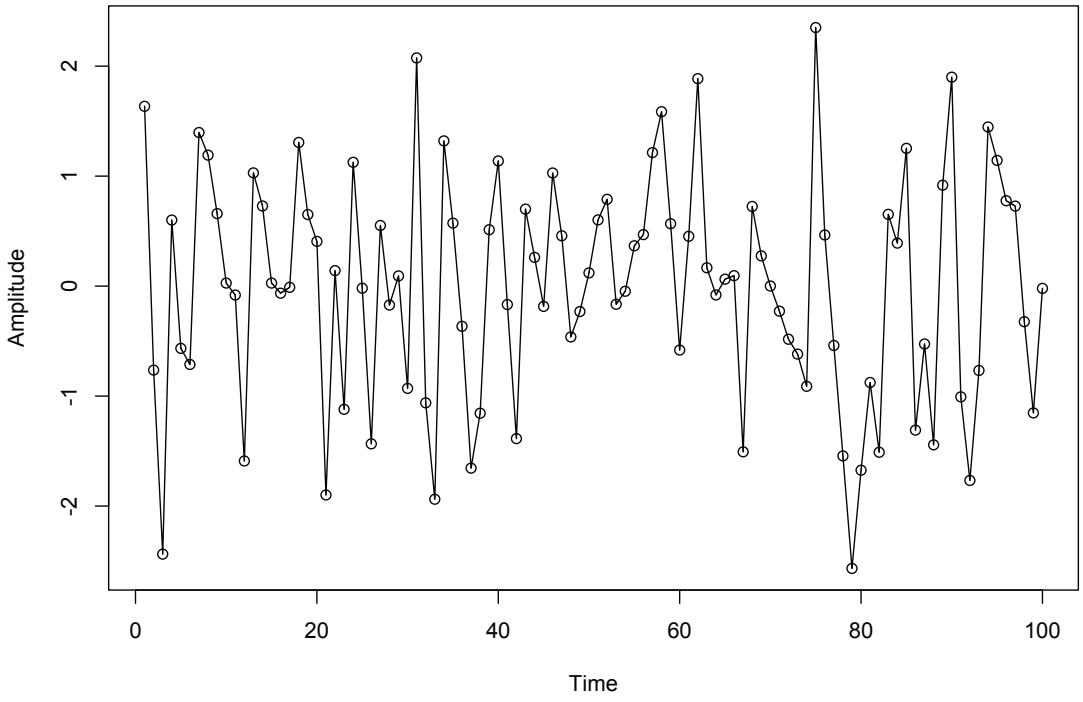


Figure 3.5: Plot of 100 samples from an ARMA(2,1) process where the AR coefficients are 0.6 and -0.3 and the MA coefficient is -0.4.

Equation 3.11 models the general ARMA process described in Equation 3.10, under the impact of an outlier:

$$Y_t^* = Y_t + \omega L(B) I_t(t_1), \tag{3.11}$$

where $I_t(t_1)$ signifies the existence of an outlier at time $t = t_1$, therefore $I_t(t_1) = 1$ if $t = t_1$, and $I_t(t_1) = 0$ otherwise. The magnitude of an outlier effect is indicated by ω and $L(B)$ refers to the dynamic pattern of the outlier [9]. Considering the four types of outliers described in Equation 2.2, the dynamic pattern is equal to the effect of the outlier based on its type.

Figure 3.6 shows the same ARMA(2,1) process depicted in Figure 3.5 with an additive outlier at time $t = 50$. Equation 3.11 can represents this process with $\omega = 5$, $L(B) = 1$ and $I_t(50) = 1$.

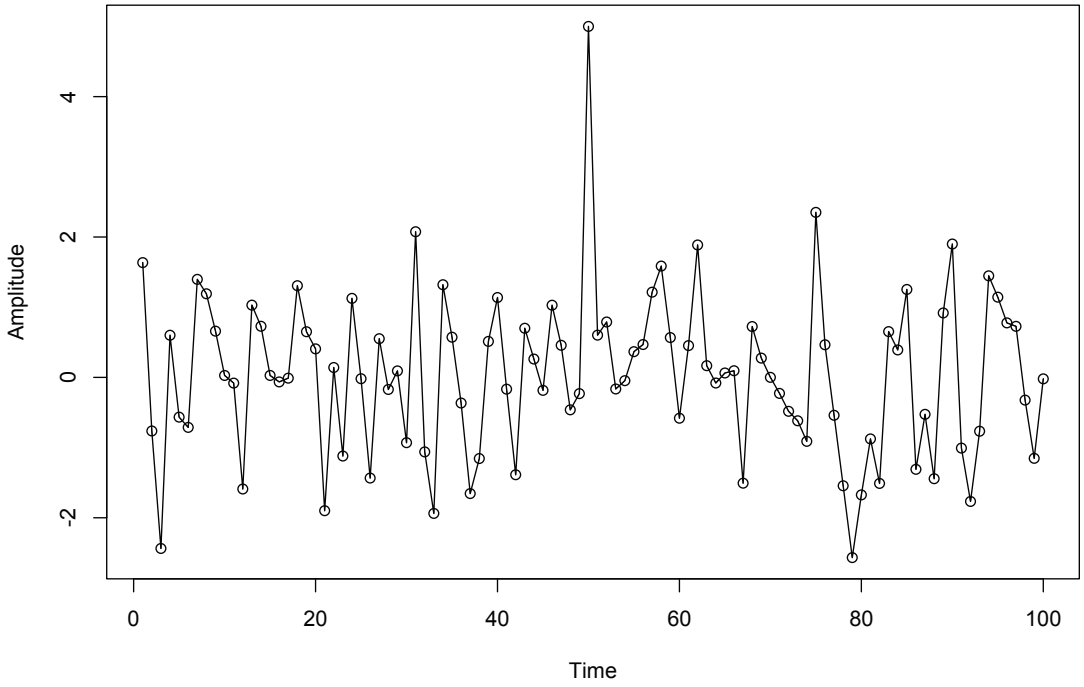


Figure 3.6: Plot of 100 samples from an ARMA(2,1) process where the AR coefficients are 0.6 and -0.3 and the MA coefficient is -0.4.The sample point at $t = 50$ is an additive outlier.

3.3.1 Estimating Outliers Effects

In order to find the location of outliers and adjust the time series, the effect of outliers on residuals must be estimated. It is assumed that time series has been observed from an advanced point in time, so that all initial values are known. For example from $t = -J$ to $t = n$, where J is an integer larger than $p + d + q$, and $1 \leq t_1 \leq n$ where p, d and q are orders of the polynomials $\phi(B)$, $\alpha(B)$ and $\theta(B)$ [9]. The $\pi(B)$ polynomial is defined

as:

$$\pi(B) = \frac{\phi(B)\alpha(B)}{\theta(B)} = 1 - \pi_1 B - \pi_2 B^2 - \ldots, \tag{3.12}$$

where the π_j weights for j beyond a moderately large J become essentially equal to zero, because roots of $\theta(B)$ are all outside the unit circle [9]. Using the formulation for a time series under the influence of outliers represented in Equation 3.11, the estimated residuals $\hat{e}_t$ can be calculated as:

$$\hat{e}_t = \pi(B)Y_t^*, \qquad t = 1, \cdots, n. \tag{3.13}$$

It can be seen that the effect of outliers on the estimated residuals have been considered by using the ARMA process under the impact of an outlier instead of the original model. Substituting dynamic patterns for different types of outliers, the estimated residuals are [9]:

$$IO : \hat{e}_t = \omega I_t(t_1) + a_t, \tag{3.14}$$

$$AO : \hat{e}_t = \omega\pi(B) I_t(t_1) + a_t, \tag{3.15}$$

$$TC : \hat{e}_t = \omega \frac{\pi(B)}{(1-\delta B)} I_t(t_1) + a_t, \tag{3.16}$$

$$LS : \hat{e}_t = \omega \frac{\pi(B)}{(1-B)} I_t(t_1) + a_t. \tag{3.17}$$

Alternatively, the Equations 3.14-3.17 can be expressed in a general form as [9]:

$$\hat{e}_t = \omega x_{it} + a_t, \qquad t = t_1, t_1 + 1, \cdots, n \qquad \text{and} \qquad i = 1, 2, 3, 4 \tag{3.18}$$

where $x_{it} = \pi(B)L_i(B)I_t(t_1)$ and i signifies the type of existing outlier. $x_{it} = 0$ for all i and $t < t_1$, $x_{it_1} = 1$ for all i and $t = t_1$ and for $k \geq 1$, $x_{1(t_1+K)} = 0$, $x_{2(t_1+K)} = -\pi_k$, $x_{3(t_1+K)} = 1 - \sum_{j=1}^{k} \pi_j$, and $x_{4(t_1+K)} = \delta^k - \sum_{j=1}^{k-1} \delta^{k-j}\pi_j - \pi_k$ where k is the number of data samples between the outlier and the current sample [9]. Subsequently, the least squares estimate for the effect of an outlier at time $t = t_1$ can be written using the general form

expressed in Equation 3.18 as [9]:

$$\hat{\omega}_{IO}(t_1) = \hat{e_{t_1}}, \tag{3.19}$$

$$\hat{\omega}_{AO}(t_1) = \sum_{t=t_1}^{n} \hat{e}_t x_{2t} / \sum_{t=t_1}^{n} x_{2t}^2, \tag{3.20}$$

$$\hat{\omega}_{LS}(t_1) = \sum_{t=t_1}^{n} \hat{e}_t x_{3t} / \sum_{t=t_1}^{n} x_{3t}^2, \tag{3.21}$$

$$\hat{\omega}_{TC}(t_1) = \sum_{t=t_1}^{n} \hat{e}_t x_{4t} / \sum_{t=t_1}^{n} x_{4t}^2. \tag{3.22}$$

3.3.2 Locating and Identifying Outliers

The methodology used to locate and identify outliers is based on analyzing the maximum value of the test statistics of the outlier effects. Using the above equations, test statistics expressed in [9] are as follows:

$$\hat{\tau}_{IO}(t_1) = \hat{\omega}_{IO}(t_1)/\hat{\sigma}_a, \tag{3.23}$$

$$\hat{\tau}_{AO}(t_1) = \hat{\omega}_{AO}(t_1)/\hat{\sigma}_a(\sum_{t=t_1}^{n} x_{2t}^2)^{1/2}, \tag{3.24}$$

$$\hat{\tau}_{LS}(t_1) = \hat{\omega}_{LS}(t_1)/\hat{\sigma}_a(\sum_{t=t_1}^{n} x_{3t}^2)^{1/2}, \tag{3.25}$$

$$\hat{\tau}_{TC}(t_1) = \hat{\omega}_{TC}(t_1)/\hat{\sigma}_a(\sum_{t=t_1}^{n} x_{4t}^2)^{1/2}. \tag{3.26}$$

If at time $t = t_1$ the maximum value of these test statistics is greater than a threshold value, then an outlier exists at time $t = t_1$ and the test statistic with the maximum value signifies the type of outlier. Knowing the effect of the outlier and its location, both time series and residuals can be easily adjusted.

The estimated effects are based on the assumption that there is only one outlier

in the time series. In case of multiple outliers, the effect of the outlier at time $t = t_1$ is a consequent of all the present outliers in the time series. Therefore, effects expressed by Equations 3.19—3.22 can be biased. In order to get the unbiased effect, all outliers and their effects must be considered at time $t = t_1$. There are two different approaches to estimate the effects. The first approach estimates the effects jointly by considering the effects of all outliers at the same time. The second approach estimates the effects sequentially. It considers the the effect of one outlier at a time and adjusts the residuals. Then using the adjusted residual, it considers the effect of second outlier and so forth for the rest of outliers.

Since the location and number of outliers are unknown, it is impractical to use the joint estimate effects for the detection process and it is more reasonable to detect outliers one by one even when multiple outliers are present in the time series [9].

3.3.3 Residual Standard Deviation

Since the residuals might be contaminated by outliers, the estimation of residual standard deviation σ_a might be affected as well. In the presence of outliers, σ_a might be biased or overestimated. There are different approaches that can be used for a better estimate which result in more precise values for test statistics given in Equations 3.23—3.26. The median absolute deviation (MAD) method is a measure of statistical dispersion which is more robust to outliers. It is defined as the median of the absolute deviation from the samples' median [43]:

$$\hat{\sigma}_a = 1.483 \times \text{median}(|\hat{e}_t - \tilde{e}|). \tag{3.27}$$

The scale factor is $k = 1.483$ for normally distributed samples, and $\tilde{e}$ is the median of estimated residuals. The $\alpha\%$ trimmed method is an estimation procedure which considers the extreme values as outliers and removes them using a truncation process to calculate the mean. The trimmed estimator is more robust than using the sample standard deviation

since the outlier samples are discarded. In the truncation process, $\alpha\%$ of the lowest and highest values will be excluded and the estimation will be based on the central samples [44]. Another method that excludes one sample is the omit-one method [9]. In this procedure the only sample that is being removed is at time $t = t_1$ for which an outlier test is being examined. The omit-one method is not as efficient as the other two since it does not remove the effect of extreme values and only removes the current residual. However, omit-one is computationally less expensive as it does not require the sorted residuals [9].

3.3.4 The Procedure of Joint Estimation of Model Parameters and Outlier Effects

The algorithm proposed in [9] is an iterative procedure that consists of three stages. Figure 3.7 depicts the flowchart of the algorithm. In the initialize stage, the time series is modeled by an ARMA process and the model parameters are estimated. Then residuals are calculated based on the model parameters. In the first stage, test statistics for different types of outliers are computed for each sample point. Subsequently, the maximum of all test statistics is selected as the dominant type and is compared to the critical value. If the test statistic is greater than the critical value, then the sample point is considered as an outlier. Therefore, its effect must be removed from the residuals. After adjusting both the residuals and the time series, the next sample point is analyzed. At the end, if no outliers are found, then the time series under question is clean and free of outliers; otherwise, the algorithm moves to the next stage.

In the second stage, the effect of all outliers is estimated jointly for each outlier point found in the first stage. Then, test statistics are calculated based on these new effects. If the test statistic for an outlier point is no longer greater than the critical value, the sample point is removed from the list of outliers. This indicates that the point is not an outlier and

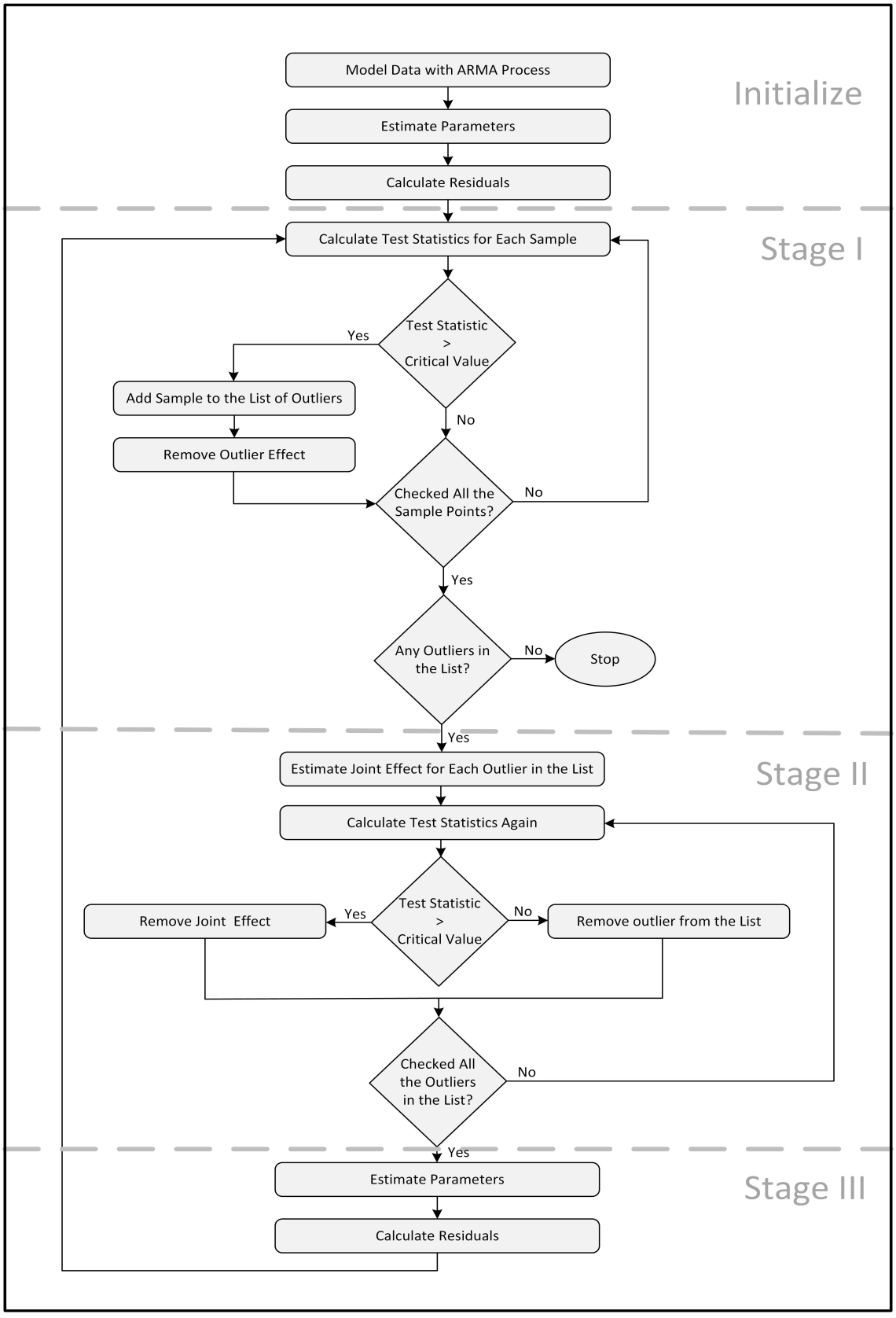


Figure 3.7: The flowchart of the complete outlier detection algorithm proposed in [9] which shows different stages of the algorithm.

it was considered as one due to the effect of other outliers found in the time series. However, if the test statistic is still greater than the critical value, then the point is an outlier and the computed effect must be removed from the residuals and the observations. After analyzing all the outlier points, the model parameters are estimated again.

In the last stage, the residuals are calculated again based on estimations of the model parameters. Then, the algorithm iterates through the first two stages once again to make sure that the outliers found are real outliers and to estimate more precise outlier effects based on the new model parameters.

It can be seen that the algorithm is iterative and estimates the model parameters after each iteration. This algorithm has two goals: to detect the outliers, and to estimate the unbiased model parameters for future analysis. Since our goal in this thesis is to implement outlier detection in hardware, a simplified version of this algorithm is used which is explained next.

3.3.4.1 Simplified Outlier Detection Algorithm for Hardware Implementation

As it can be seen in Figure 3.7, the first stage of the algorithm detects outliers. It calculates the test statistics using the effect of one outlier at a time, and uses the test statistics to detect outliers sequentially. The second stage determines the spurious outliers and removes them from the list of outliers, and the last stage uses the re-estimated model parameters to detect masked outliers. A simplified version of the algorithm that uses only the first stage is selected for hardware implementation. This simplification in the algorithm reduces the computation complexity in terms of various iterations and ARMA modelings. However it increases the probability of false positives due to the spurious outliers and decreases the accuracy of the algorithm due to the masked outliers.

The method in [9] considers four types of outliers, and calculates four test statistics

for each sample point. Since the outliers in the sampled signals from sensors are impulse type errors [45], only additive outliers are considered for hardware implementation. In order to add the other types, we only need to calculate the corresponding test statistics.

The simplified version of algorithm is divided in two steps:

Step 1: Parameter Estimation

(a) The time series data is modeled with an ARMA(1,1) process, and non-linear least square estimates of its parameters are calculated using the modified direct search method explained in Section 3.1.2.2.

(b) Based on the model, the residuals are computed. Assuming that the model fitted to the data is Y_t and the type of outlier is AO for which $L(B) = 1$, the intervention model can be expressed as:

$$Y_t^* = Y_t + \omega I_t(t_1). \tag{3.28}$$

Using the intervention model, the residuals are estimated as:

$$\hat{e_t} = \omega\pi(B)I_t(t_1) + a_t, \tag{3.29}$$

which considers the effect of outliers by using the intervention model instead of the original one.

(c) The standard deviation of residuals is estimated, using a trimmed estimator for robustness. The trimmed estimator discards two extreme sample points from the residuals, the minimum and the maximum, and uses the middle points to approximate an unbiased value.

Step 2: Outlier Identification

(a) The test statistic is calculated for each sample point, considering the effect of an additive outlier at each one. Using residuals standard deviation (σ_a) which is calculated in step 1, the test statistic is defined as:

$$\hat{\tau}(t_1) = \hat{\omega}(t_1)/\hat{\sigma}_a(\sum_{t=t_1}^{n} x_{2t}^2)^{1/2}, \tag{3.30}$$

where $\hat{\omega}(t_1)$ is the effect of an AO at time $t = t_1$ and is expressed as:

$$\hat{\omega}(t_1) = \sum_{t=t_1}^{n} \hat{e}_t x_{2t} / \sum_{t=t_1}^{n} x_{2t}^2. \tag{3.31}$$

The effect of one outlier at a time is considered for calculating the test statistic even when there are multiple outliers in the data, same as the original algorithm in [9].

(b) For all the sample points, the absolute value of the test statistic is compared to a predefined critical value C. If the $|\hat{\tau}(t_1)| > C$, then the sample point at $t = t_1$ is an outlier of the type "additive". If none of the sample points are outliers then the original time series is clean and free of outliers.

The main goal of this simplified algorithm is to identify the location of outliers. Following the detection of these outliers and based on the application, it can be decided whether to take any further actions regarding their presence or just replace them with a more reasonable value such as a local or global mean or median.

Chapter 4

Design and Implementation

In this chapter, the implementation of the simplified outlier detection algorithm based on the joint estimation of model parameters and outlier effects that was presented in Section 3.3.4.1 is discussed. Our target is an FPGA implementation of this algorithm that is more power efficient than its corresponding software implementation that uses a general purpose processor. Moreover, our goal is to achieve a higher throughput than the software-based counterpart while having a comparable detection accuracy. The implementation consists of two phases. In the first phase, the simplified algorithm is implemented in MATLAB to verify its functionality and accuracy. In the second phase, using MATLAB HDL Coder (a MATLAB product), the hardware description of the algorithm is generated in Verilog. The design is simulated and verified using MATLAB HDL Coder, and synthesized on an Altera FPGA using Intel Quartus.

4.1 Profiling the Outlier Detection Algorithm

The algorithm proposed in [9] is an iterative algorithm that models the time series data with an ARMA process in each iteration. Additionally, ARMA modeling is also an iterative procedure that selects a model by iterating over a set of possible models, and estimates the model parameters using an optimization algorithm that is also iterative. Therefore, it is evident that the algorithm is complex by nature. Furthermore, the implementation of the algorithm in tsoutliers package of R only implements the calculation of the test statistics. For ARMA modeling and calculating the residuals under the effect of outliers, it uses functions from other packages in R such as forecast and stats packages. This means that the hardware implementation of the algorithm should include parts of the forecast and the stats packages as well as the tsoutliers package. Therefore, we simplified the outlier detection algorithm for hardware implementation. However, prior to implementing the simplified algorithm, it was important to confirm that the simplifications done to the original algorithm will improve the performance of the hardware implementation in terms of speed as well as the area.

In order to evaluate the performance of the simplified algorithm, the original algorithm, specifically the ARMA modeling procedure was implemented in MATLAB (version 9.1 (R2016b)) based on the approach used in the tsoutliers package. In the tsoutliers package a function called `Auto.Arima` from the forecast package is used for modeling the time series data with an ARMA process [46]. This function evaluates different orders, and fits all of them to the data. Then based on a user defined criteria such as BIC or AIC, it selects the model and estimates the parameters.

We implemented the same procedure in MATLAB using similar functions from Econometrics toolbox [47]. In each iteration, the procedure considers one of the possible model orders and fits the model to the time series data. Then based on the BIC criteria, the model order is selected from the evaluated possible models. The Profiler in MATLAB is a

tool that measures the time spent for each part of the code and is useful for both optimizing and debugging. The execution time for the ARMA modeling procedure was measured using the Profiler. It took 15.09s to select the model for a time series with 1000 sample points between 9 different models and 98% of this time was spent on fitting these models to the time series data. Alternatively, the ARMA modeling procedure was modified to fit a single known model to the time series data. In this case, it took 2.19s to estimate the parameters of the model. It can be inferred that knowing the order and using a fixed order reduces the execution time by 85%. Additionally, ARMA modeling used almost 82% of the execution time of the complete algorithm in both cases; therefore, it is important to implement this task in hardware.

4.2 MATLAB Implementation of the Simplified Algorithm

Figure 4.1 depicts the flowchart of the simplified outlier detection algorithm that is implemented in MATLAB. Assuming that the model is ARMA(1,1), the implementation of ARMA modeling contains the procedure of fitting the model to the time series data using the optimization algorithm. Therefore, the modified pattern search method explained in Section 3.1.2.2 is implemented as the ARMA modeling procedure in a function called `Patternsearch`.

Residuals are calculated in a function called `Residuals` using the estimated parameters from pattern search. However, the estimated residuals require the $\pi(B)$ polynomial based on Equation 3.29. The tsoutliers package uses the function `ARMAtoMA` from the stats package for collecting the π polynomial coefficients. This function obtains the coefficients from the relation of $\pi(B)$ with $\theta(B)$ and $\phi(B)$ polynomials. The `ARMAtoMA` function in our

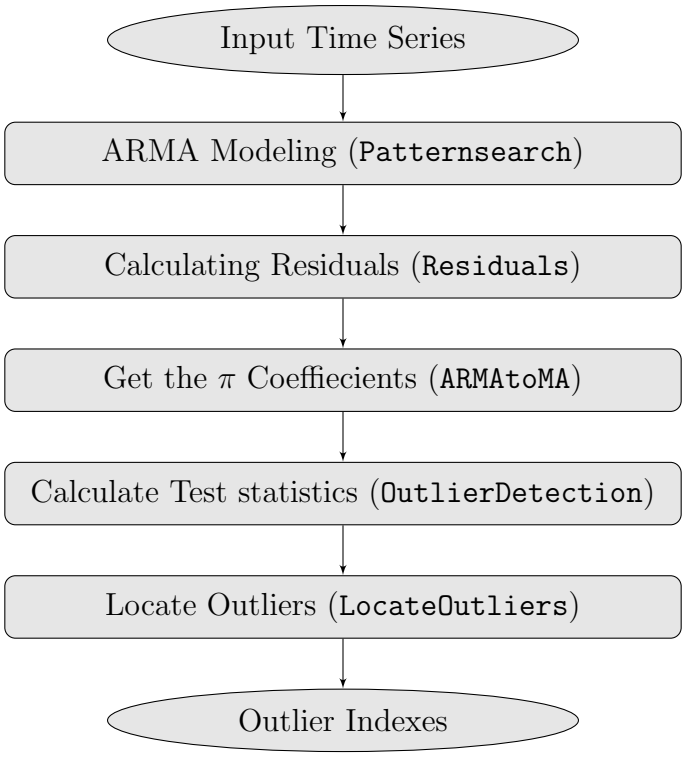


Figure 4.1: Flowchart of the implemented outlier detection algorithm and its different steps and functions.

implementation does the same thing. This function receives the AR and MA coefficients and returns the coefficients for $\pi(B)$ polynomial.

The test statistics are calculated for additive outliers at each sample point based on the Equation 3.30, and using the estimated residuals and their standard deviation. The standard deviation of residuals is approximated using the trimmed method which requires the sorted residuals. The `Sorting` function, sorts the residuals using bubble sort algorithm. The `LocateOutlier` function takes the test statistics and the critical value as its input arguments and compares the absolute value of the test statistics with the critical value and returns the indexes of the sample points that are outliers. Finally, the MATLAB implementation plots the time series data and marks the outliers with an asterisk. This implementation is presented in Appendix A.1.

4.3 Hardware Implementation of the Outlier Detection Algorithm

MATLAB HDL Coder (version 3.9) is used to generate HDL code for the MATLAB implementation of the outlier detection algorithm. MATLAB HDL Coder can generate target-independent, portable and synthesizable Verilog or VHDL files from MATLAB functions and Simulink models [48]. These Verilog or VHDL codes can be used for FPGA programming or ASIC prototyping. Figure 4.2 depicts the design flow for the outlier detection algorithm using HDL Coder. The first step in utilizing HDL Coder is to design the model with MATLAB functions which is explained in Section 4.2.

The second step in the design flow is fixed-point conversion and HDL generation. Hardware designs are composed of fixed resources, and it is important to manage data word lengths to use these resources efficiently. However, converting algorithms developed using double-precision floating-point arithmetic to fixed-point implementations while preserving the required accuracy is a difficult task. One of the advantages of HDL Coder is the floating-point to fixed-point conversion.

The fixed-point conversion starts by analyzing the compatibility of the design with the code generation process [49]. Then, the procedure proposes fixed-point data types based on the test benches that are provided. It is required that the test benches cover the dynamic range of variables and execute all parts of the design. However, the user can also define the types and word lengths manually. The accuracy is verified to be maintained by validating the proposed fixed-point data types. For instance, the fixed-point conversion suggested 9-bit inputs for HDL generation using synthetic time series data inputs in the range of -10.0 to 10.0.

Then, HDL Coder uses the fixed-point implementation to generate the HDL codes.

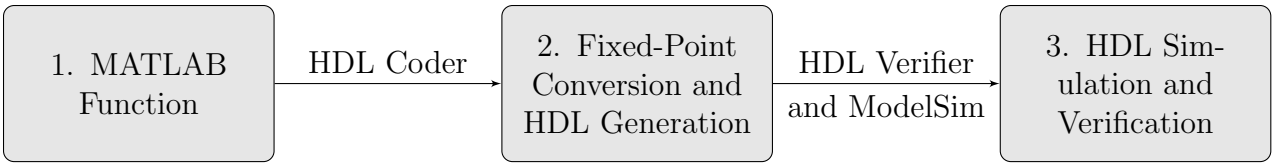


Figure 4.2: Design flow used for outlier detection algorithm in MATLAB HDL Coder. The fixed-point conversion and HDL generation are done using the HDL coder. Simulation is done in Modelsim integrated environment, and verification is done using the HDL verifier.

Workflow advisor of HDL Coder automates the procedure of HDL generation, and grants control to the user over settings such as the coding style, the target language and optimization settings. At the same time, MATLAB test bench can also be converted to Verilog or VHDL for simulation purposes.

The last step in the design flow is the HDL simulation and verification. HDL Coder has the ability to connect to third party simulation and synthesis tools such as Modelsim, Altera Quartus, and Xilinx ISE. It can either connect to these tools on its own or through another tool called HDL Verifier. Employing these tools in an integrated environment, it can run simulations to verify the HDL design and synthesize it. HDL Verifier provides the option to verify the HDL design using the HDL test bench generated in the previous step. Simulators like ModelSim can be integrated in the HDL Verifier to run the simulation. Finally, the result from this simulation is compared to the one from the fixed-point conversion to verify the accuracy and the functionality of the HDL design. For our outlier detection Algorithm the simulation is done using the Modelsim integrated environment and verification is done using HDL Verifier (version 5.1).

4.3.1 Synthesizable MATLAB Implementation

HDL Coder is a powerful tool that receives a function accompanied by few test benches, and generates portable, synthesizable Verilog or VHDL codes. However, it is important to know the capabilities and limitations of HDL Coder. In order to generate synthe-

sizable code, the .m file must be descriptive enough and must follow the rules of hardware description languages. Since our initial MATLAB implementation was not synthesizable, we needed to modify our MATLAB implementation to make the generated code synthesizable. Some tips on how to modify the MATLAB code for HDL generation are listed below:

- All the functions from MATLAB library that are not supported in HDL Coder must be replaced with a user defined one.
- Function calls inside for loops and conditional statements cannot be translated to module instantiations and will be translated inline.
- Loops with an undefined number of iterations are not supported. Specifically `while` loops are only acceptable when the number of iterations are known.
- In order to have variables registered with a clock the keyword `Persist` must be used in their definition.
- Boolean variables are not supported and will be translated to "ufix1" type. The type "ufix1" is an arithmetic type that represent numeric values 0 and 1 and supports arithmetic operations.

The MATLAB implementation of our outlier detection algorithm is modular and the code is broken into multiple functions. We found it easy to modify each function separately according to the rules for a synthesizable code. However, for ARMA Modeling to be synthesizable, the modified pattern search method for hardware implementation must be used. The modified direct search method uses a maximum number of iterations as its stopping strategy as explained in Section 3.1.2.3. Each iteration starts by an exploratory search, and based on the result of the search either goes through a pattern move or not. The RTL block diagram for one iteration can be seen in Figure 4.3.

Each iteration receives the input time series data and the initial perturbation vector from the top-level module, and gets the perturbation vector, the current point or coefficients and the old objective function value from the previous iteration. Additionally, each iteration outputs the perturbation vector, the old objective function which is the optimized value so far and the current point which is the coefficients that optimize the objective function value.

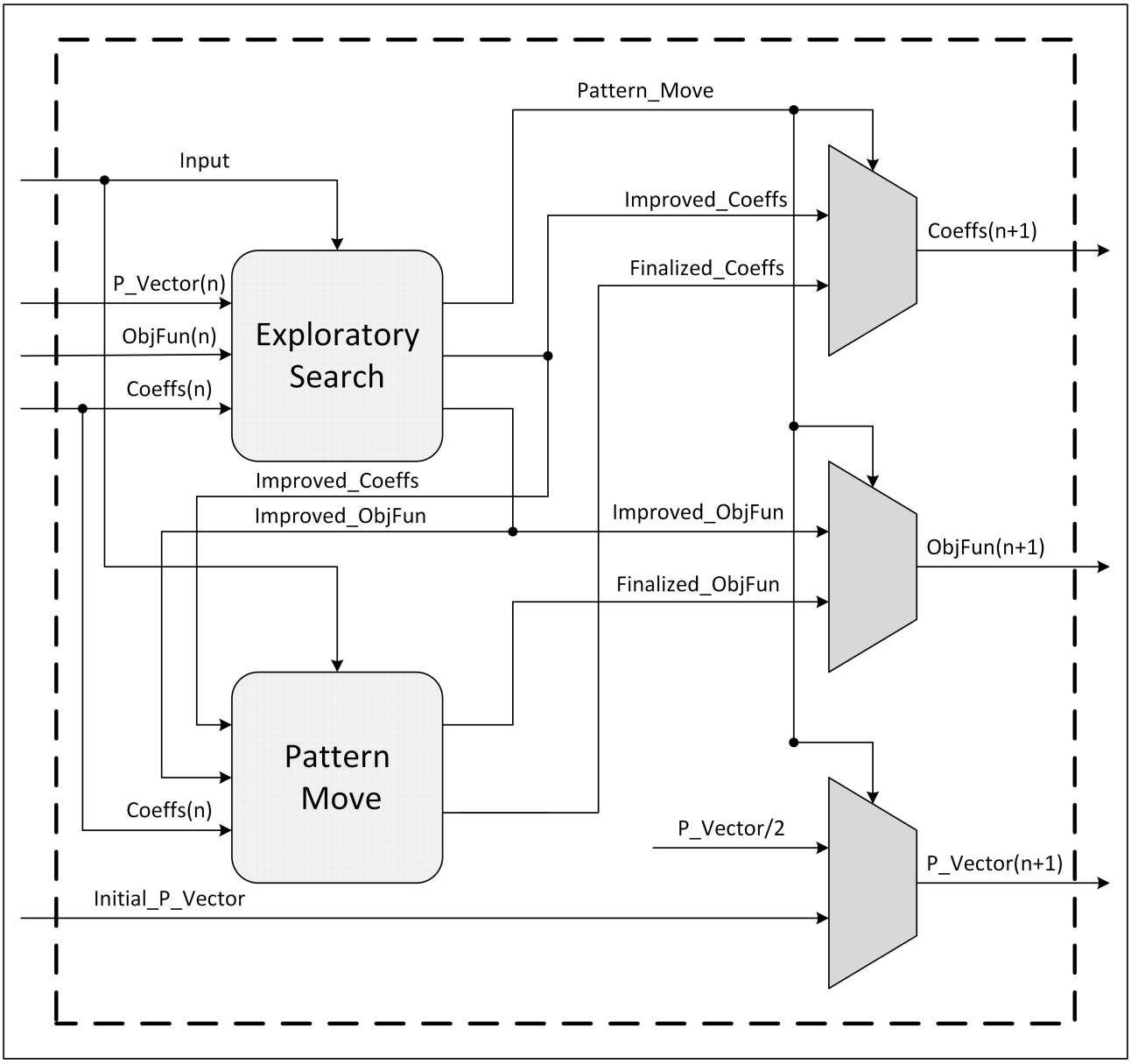


Figure 4.3: The block diagram for n^{th} iteration of the pattern search with inputs coming from the previous iteration and outputs going to the next iteration. The multiplexers select the best results of this iteration using the pattern move flag.

The three multiplexers select and assign the outputs based on the internal "Pattern_Move" flag. Pattern_move flag is true when the exploratory search around the current point is successful and specifies that a pattern move is required. As a consequence if the flag is true then the results from pattern move block are assigned to the outputs, otherwise the results from the exploratory search are selected as the outputs which are the same results from previous iteration with the modified perturbation vector.

It can be seen that the outputs from each block are the inputs to the next, and it is easy to connect these blocks together. Therefore, the complete algorithm is implemented by instantiating the same block for the maximum number of iteration times and connecting them together. Figure 4.4 demonstrates the RTL block diagram of the modified direct search method with n iterations. Subsequently, the modified direct search method is implemented in MATLAB by calling the function for one iteration multiple times which results in multiple instantiations of the same hardware in the HDL code as shown in Figure 4.4.

The hardware implementation of the simplified outlier detection algorithm is configurable and synthesizable. The design parameters are; 1) Number of iterations in the optimization algorithm; 2)Number of samples or the size of time series data; 3) Critical value. The synthesizable MATLAB implementation for the configuration with 20 samples in

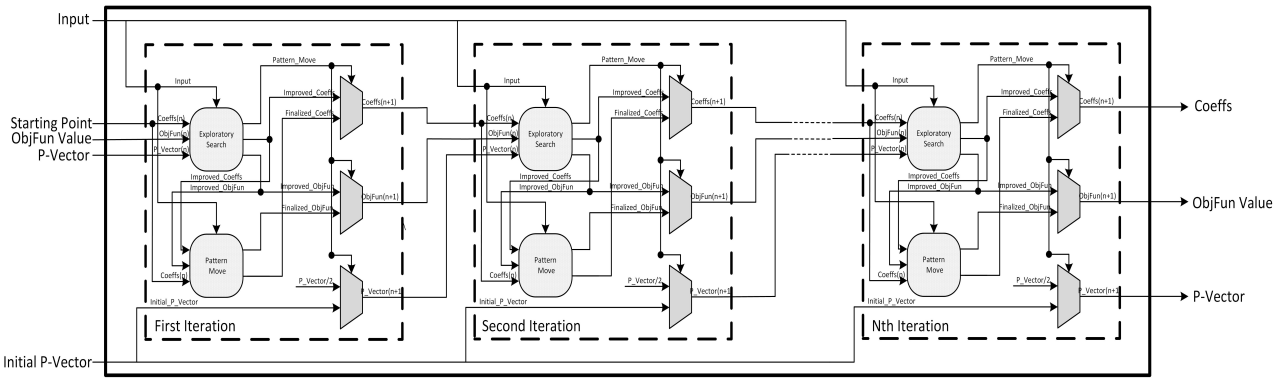


Figure 4.4: The block diagram for n^{th} iteration of the pattern search with inputs from the previous iteration and outputs for the next iteration. The multiplexers select the best results of this iteration using the pattern move flag.

the time series data and 5 iterations of the optimization algorithm is presented in Appendix A.2.

4.4 Synthesizing the Design

Once the HDL code generated by MATLAB HDL Coder was simulated and verified, Intel Quartus Prime version 17.0.0 Standard Edition was used for both synthesis and place and route. The outlier detection algorithm is implemented on an Altera Stratix V family FPGA. The Stratix V family is a 28nm process technology FPGA, which has lowest power consumption for high performance applications compared to the previous Altera generation devices. The Altera Startix V 5SGSED8N3F4514 device has 1963 DSP units and 262400 Logic Elements, and is desirable for the outlier detection algorithm.

A configuration of the outlier detection algorithm for an input time series with 20 sample points and 5 iterations of optimization algorithm is fitted on the aforementioned device. The compilation process required 78GB of RAM and took almost 27 hours to complete.

Chapter 5

Experimental Results

In this chapter, the simulation results are presented for two versions of the implementation of the simplified algorithm. The first version is the MATLAB implementation of the simplified algorithm which was not synthesizable. However, it can process input time series data with any number of samples and uses 10 iterations of the optimization algorithm. This version is presented in Appendix A.1 and is called the Original Implementation in this chapter. The second implementation is a configuration of the modified version of the previous implementation which is synthesizable; therefore it is called Synthesizable Implementation which is presented in Appendix A.2. This Implementation processes only 20 sample points at a time and uses 5 iterations of the optimization algorithm. Using both synthetic and real world data, the results from these two implementations are compared to the result from tsoutliers package in R. Then, the detection accuracy and the false positive rate are evaluated for the Synthesizable Implementation. Also, the results for synthesizing and fitting the Synthesizable Implementation on an FPGA are also presented and the throughput and power of the FPGA is analyzed.

5.1 Experiments with Synthetic Data

Test data can be generated from an ARMA process in MATLAB. Using `arima` function, the model of the process can be defined and using the `simulate` function along with the defined model, sample series can be easily generated.

5.1.1 tsoutliers' Example

The same example that is used in tsoutliers package documentation [10], is used in order to compare the results. A time series of 120 samples is generated from an ARMA(1,1) process where MA = -0.4 and AR = 0.7. A number of outliers are manually injected into the data; two additive outliers at times $t = 15$ and $t = 45$ and a level shift from $t = 80$ to $t = 120$.

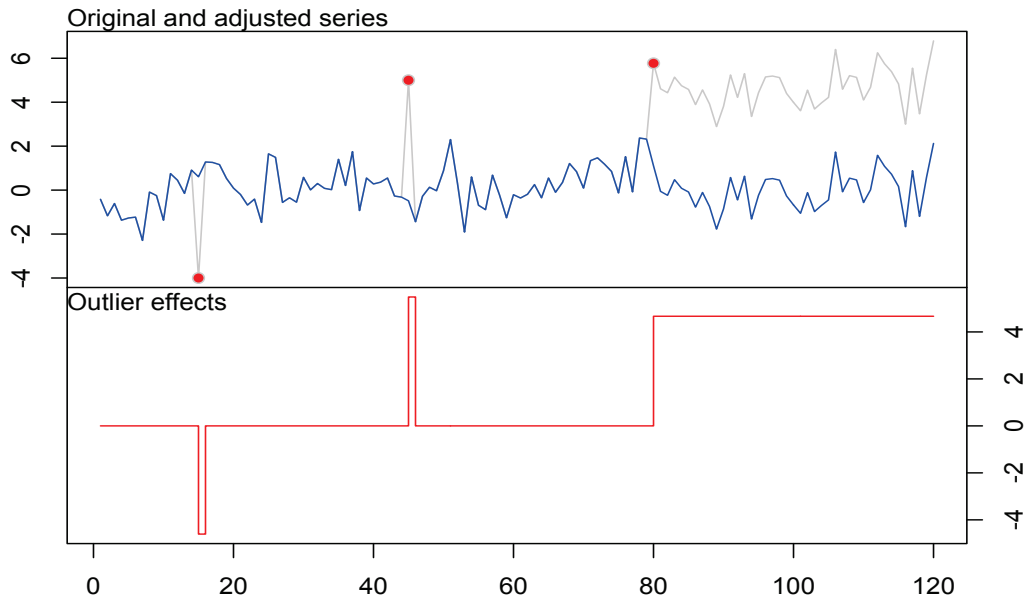


Figure 5.1: The re-generated result of tsoutliers package for the ARMA process data where MA = -0.4 and AR = 0.7 from [10]. In addition to original and adjusted series the outlier effects are also shown.

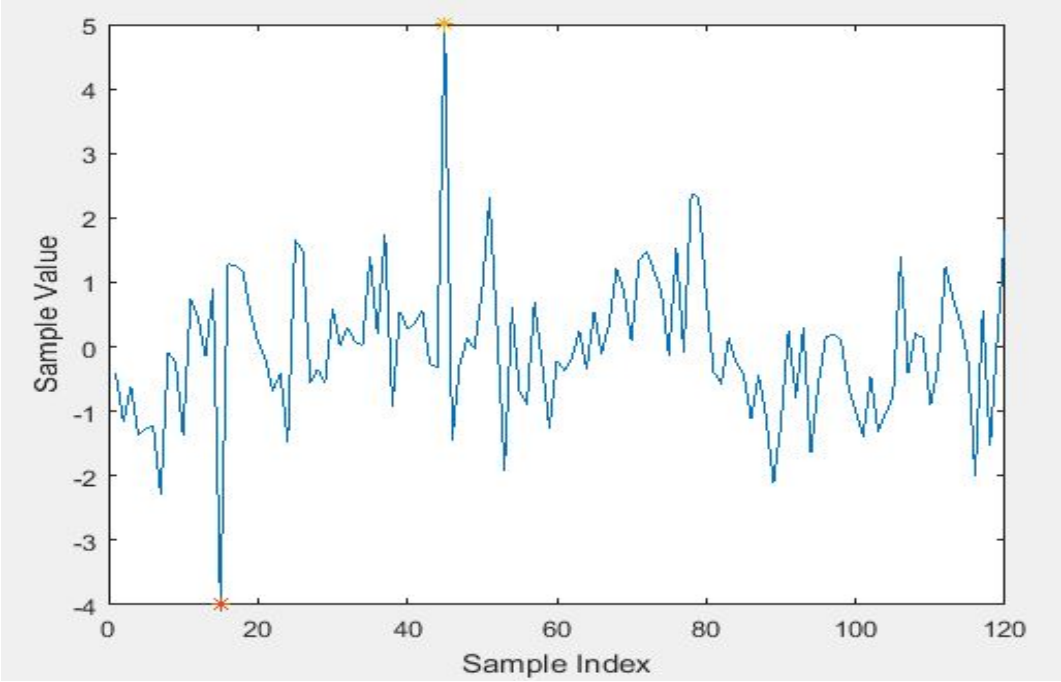
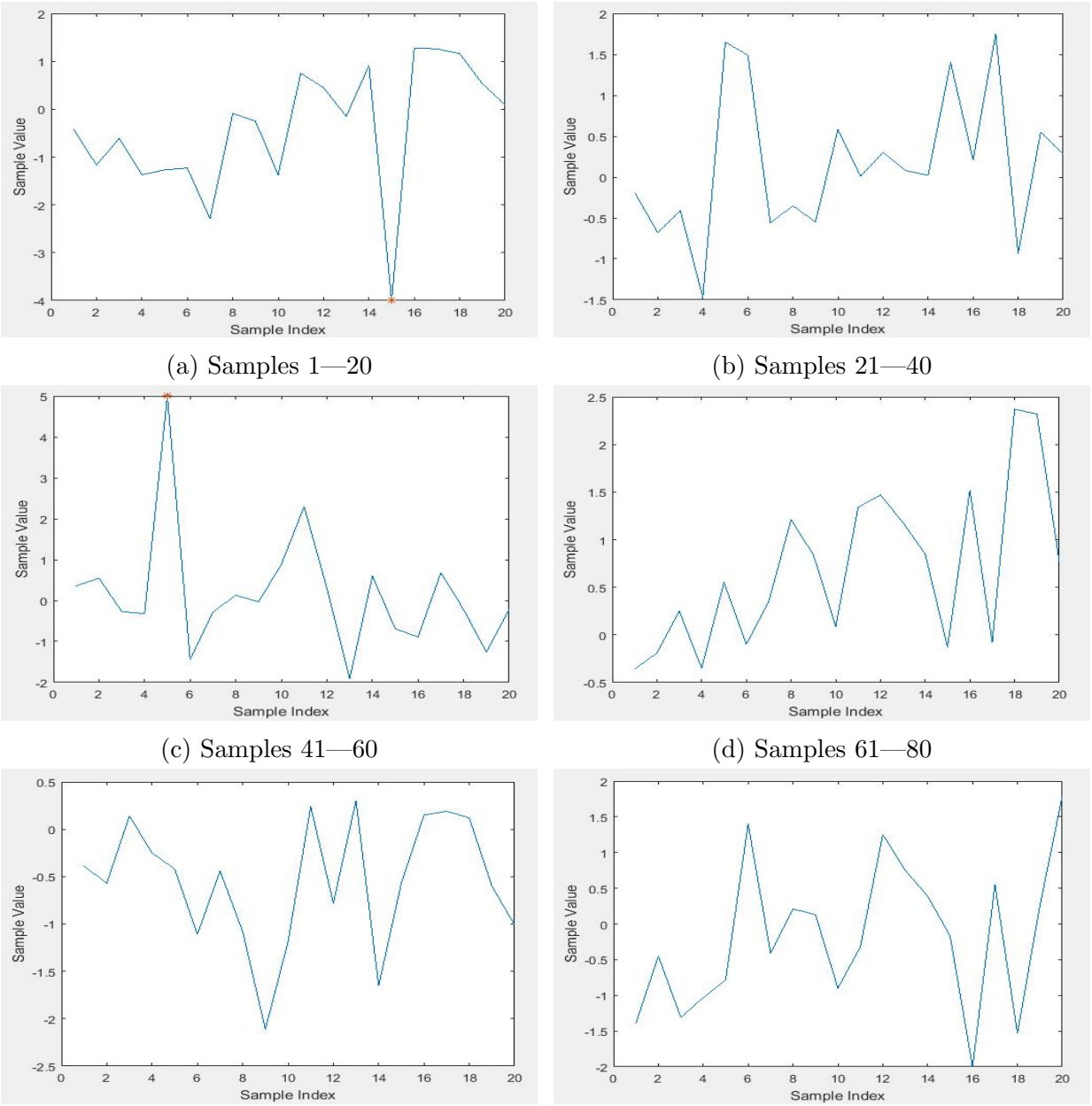


Figure 5.2: The result from Original Implementation for the ARMA process of MA = -0.4 and AR = 0.7 generated data with outlier points $t = 15$ and $t = 45$ detected.

Figure 5.1 shows the result of this experiment for tsoutliers package. The top plot contains both the original and adjusted series. The original series is the transparent one that contains outliers while the adjusted one is bold and free of outliers. The bottom plot shows the location of detected outliers and specifies their type based on their plotted effect. Although the time series data is discrete, the tsoutliers package plots both the input and the adjusted series continuously. In order to easily compare our results to tsoutliers, we plotted the time series data continuously as well.

The same generated data with two injected AOs at times $t = 15$ and $t = 45$ was applied to both the Original Implementation and the Synthesizable Implementation. Figure 5.2 shows the result for the Original Implementation. The asterisks show outliers and their locations. Since only AOs are considered, there is no need to specify the type. The result of the same experiment for the Synthesizable Implementation is shown in Figure 5.3. Since this implementation processes 20 sample points at a time, the input time series data is divided

into 6 segments and each segment is applied to the algorithm separately. It can be seen that the Synthesizable Implementation also detects both outliers injected in the time series data.



(a) Samples 1—20

(b) Samples 21—40

(c) Samples 41—60

(d) Samples 61—80

(e) Samples 81—100

(f) Samples 101—120

Figure 5.3: The result of the synthesized algorithm for the ARMA process of MA = -0.4 and AR = 0.7 generated data with outlier points $t = 15$ and $t = 45$. The time series data is divided into 6 segments of 20 samples and each segment is applied to the algorithm separately.

5.1.2 Random Data

Another time series example is produced based on an ARMA(1,1) process with MA = 0.7 and AR = 0.2, which are selected randomly. Table 5.1 shows all AOs that have been injected in the time series data. It can be seen that outliers have different magnitudes. Figure 5.4 shows the result of this experiment for tsoutliers package. All the outliers are detected, however, the additive outlier at time $t = 105$ is misidentified as a temporary change.

Table 5.1: List of additive outliers injected in the random data generated from ARMA(1,1) with MA = 0.7 and AR =0.2

Index	**Value**
32	3
58	5
75	6
91	-4
105	-6

Figure 5.5 depicts the result of the same experiment for the Original Implementation. The sample point at time $t = 2$ is wrongly marked as an outlier, and the outlier at time $t = 32$ is masked and not detected.

The reason for the undetected and spurious outliers are the simplifications done to the complete algorithm that is implemented in tsoutliers package. The complete algorithm has different stages to deal with spurious and masked outliers. The spurious outliers are removed in the second stage of the algorithm by calculating the joint effect for each detected outlier and re-estimating the model parameters to make sure that they are not spurious outliers. In the third stage the algorithm iterates through the first two stages again to

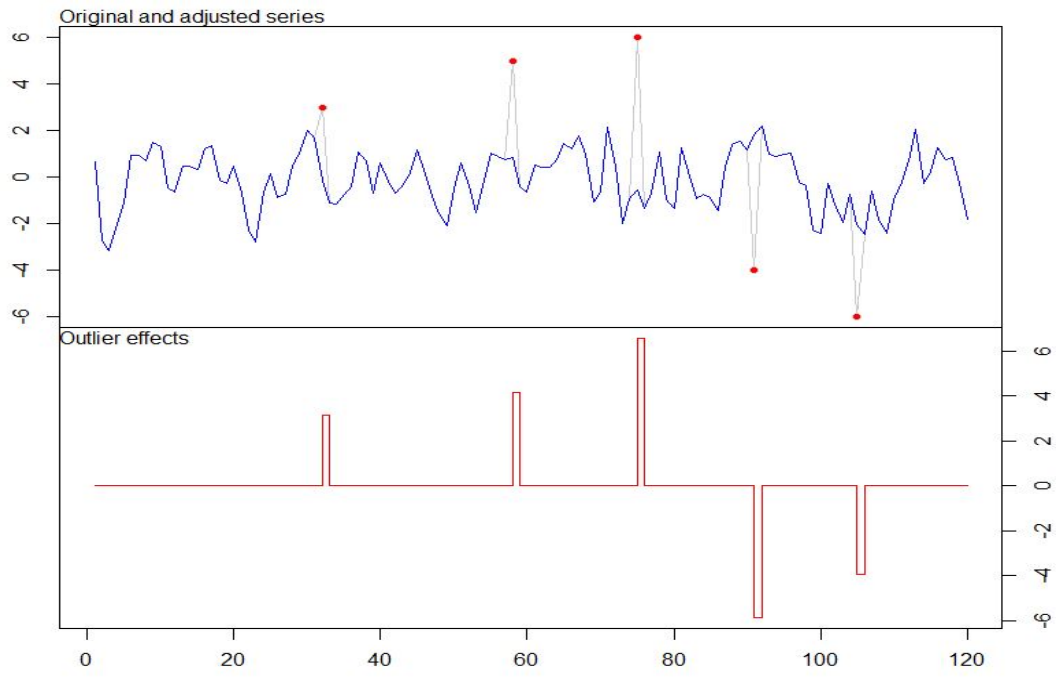


Figure 5.4: The result of the tsoutliers package for the ARMA process generated data where MA = 0.7 and AR = 0.2 with 5 additive outliers. All outliers are detected.

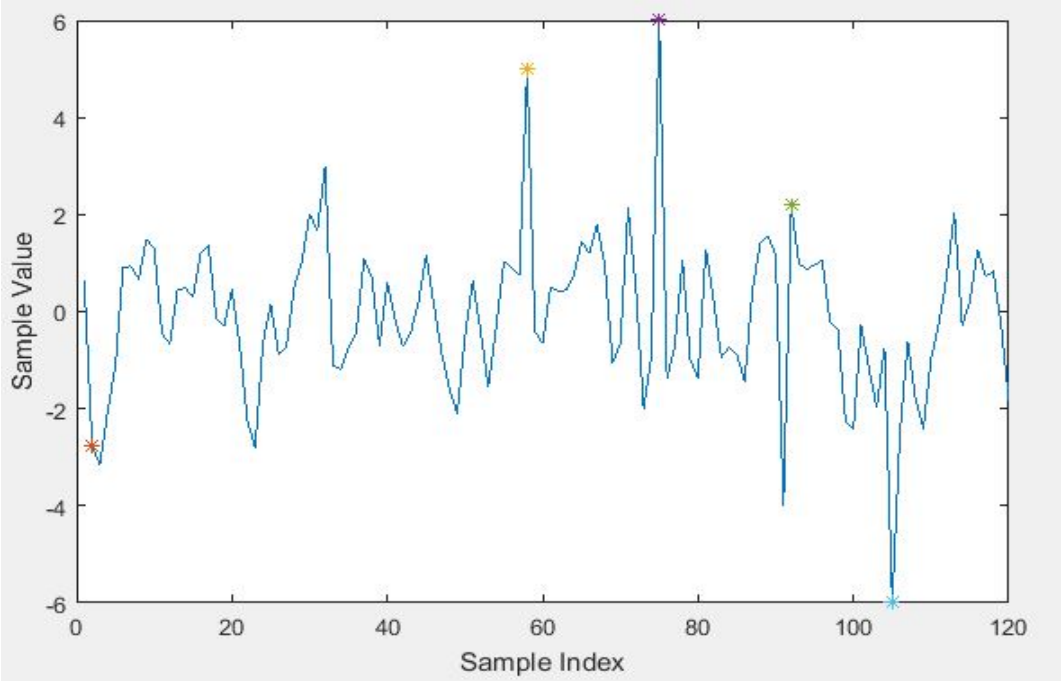
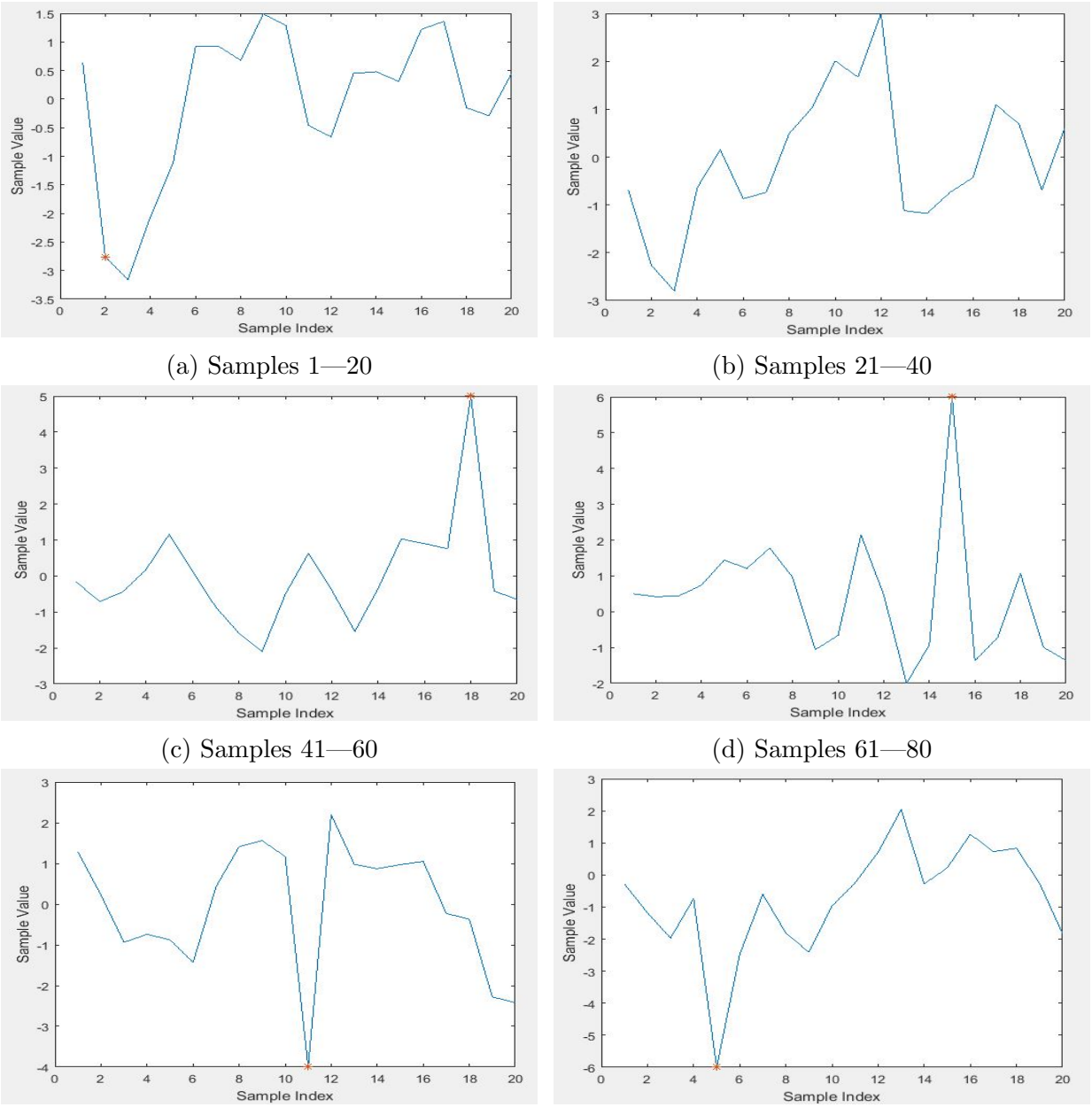


Figure 5.5: The result of the Original Implementation for the ARMA process of MA = 0.7 and AR = 0.2 with 5 additive outliers. The outlier at time $t = 2$ is a spurious outlier.

identify new outliers which were undetectable previously. However, only the first stage of the algorithm is implemented in both of our implementations.



(a) Samples 1—20

(b) Samples 21—40

(c) Samples 41—60

(d) Samples 61—80

(e) Samples 81—100

(f) Samples 101—120

Figure 5.6: The result of the sSynthesizable Implementation for the ARMA process of MA = 0.7 and AR = 0.2 with 5 outliers. The time series data is divided into 6 segments of 20 samples and each segment is applied to the algorithm separately.The outlier at time $t = 2$ is a spurious outlier.

Figure 5.6 shows the result of Synthesizable Implementation for the same experiment. As expected, this implementation also marks the sample point at $t = 2$ as an outlier and is unable to detect the outlier at time $t = 32$. Although AOs only affect the outlier sample point, they can have an impact on the test statistics of the previous sample points. This is due to the fact that for each sample point, only the effect of that exact point as an outlier is considered which could result in spurious outliers. It can be inferred that the spurious outlier at time $t = 2$ is either due to the effect of all the other outliers that are present in the time series data or due to the estimation of the model parameters. Also, the masked outlier at time $t = 32$ has a small magnitude compared to the other outliers which can be the reason that it is not detected.

5.2 ECG Signal Test data

In order to truly examine the capabilities of our implementations in detecting outliers, we used an Electrocardiography (ECG) signal for testing. ECG is the measurement of the heart's electrical activities that are clinically captured through electrodes connected to the body. During each heartbeat, the heart muscle has a specific pattern of depolarizing and repolarizing. These patterns can be recorded from the skin. ECG signals provide important information regarding patient's health.

Recently, the idea of using ECG signals in the field of biometrics has been investigated [50]. The reason that ECG signals are targeted is that they contain subject-dependent information, and are constantly available. The existing clinical procedure of collecting ECG signals is not convenient for applications outside the clinic. New approaches have been proposed that are more reasonable such as using textile electrodes, using a necklace with the pendant, and using lead sensor at fingers and hand palms [45]. However, one of the challenges in these approaches is that the signal may contain noise artifacts. The authors in [45]

have shown that a signal collected from the finger can contain additive noise level identical to the maximum amplitude in the signal due to the motion artifacts. It is expected that our implementations detect these noise artifact.

The ECG signals collected using the aforementioned approaches are not publicly available. Therefore, we used both normal and abnormal ECG signals from the PhysioNet databases [51] and randomly injected additive outliers in them.

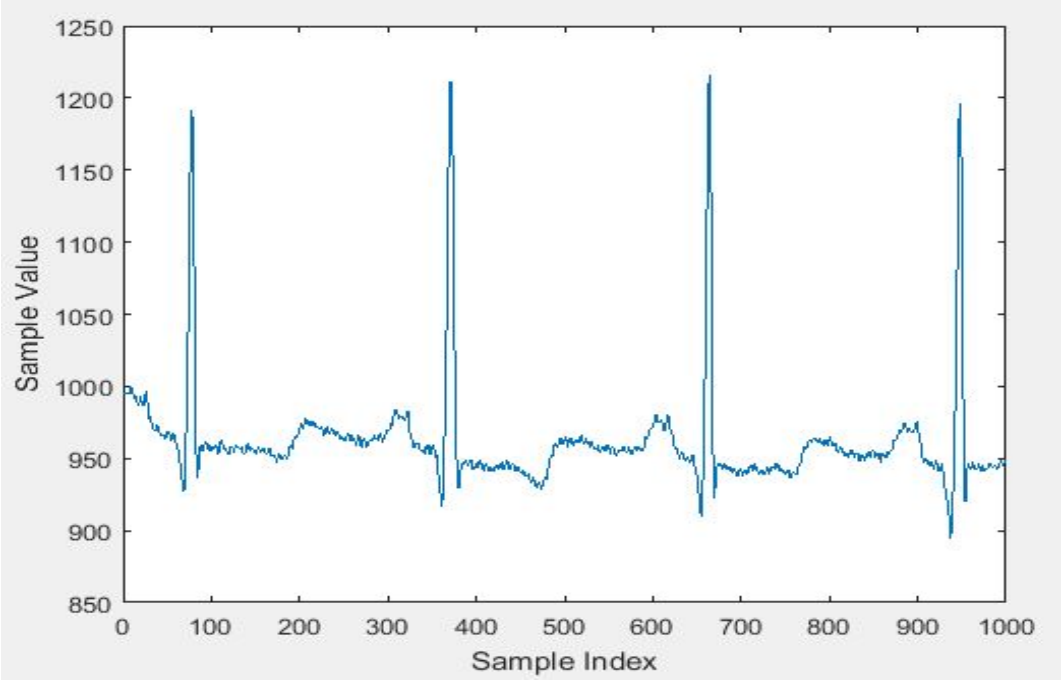


Figure 5.7: A segment of an abnormal ECG signal. The signal is not stationary but it is not easy to realize the trend.

The example shown here is from the MIT-BIH Arrhythmia database [52]. Before analyzing the result, it is important to validate the stationary assumption. Figure 5.7 depicts a small segment of an original ECG signal. It is not easy to see the trend and non-stationarity of the signal. However, Figure 5.8 shows that the best model which can be fitted to the signal is ARIMA(2,1,2). This means that the original signal is not stationary but its first difference is, and the best model that fits the differenced signal is ARMA(2,2).

```
>> estimate(AutoArima(Orig_Sig),Orig_Sig)

    ARIMA(2,1,2) Model:
    --------------------
    Conditional Probability Distribution: Gaussian

                                  Standard          t
     Parameter       Value          Error       Statistic
    -----------   -----------   ------------   -----------
     Constant    -0.00400979     0.0797525     -0.0502779
        AR{1}        1.39017      0.011259        123.473
        AR{2}       -0.663237    0.00880432       -75.3309
        MA{1}       -0.174347     0.0148243       -11.7609
        MA{2}        0.248092     0.0158698         15.633
     Variance          18.031      0.271762        66.3485

ans =

    ARIMA(2,1,2) Model:
    --------------------
    Distribution: Name = 'Gaussian'
               P: 3
               D: 1
               Q: 2
        Constant: -0.00400979
              AR: {1.39017 -0.663237} at Lags [1 2]
             SAR: {}
              MA: {-0.174347 0.248092} at Lags [1 2]
             SMA: {}
        Variance: 18.031
```

Figure 5.8: Best ARMA model that can represent the ECG signal. Notice that the original signal needs to be differenced and the order is ARMA(2,2).

Figure 5.9 shows the differenced ECG signal that is stationary. The differenced ECG signal is used for an experiment and outliers with the same magnitude as the peaks are injected at times $t = 220$, $t = 665$ and $t = 873$. The result of this experiment for the Original Implementation can be seen in Figure 5.10. There are three detected outliers, however the outlier at time $t = 665$ is misidentified by one sample point. It is worth mentioning that time $t = 665$ is a sample point in one of the peaks of the ECG signal. Although the algorithm misidentified the location of this outlier, it is important that it can detect an outlier in the peak.

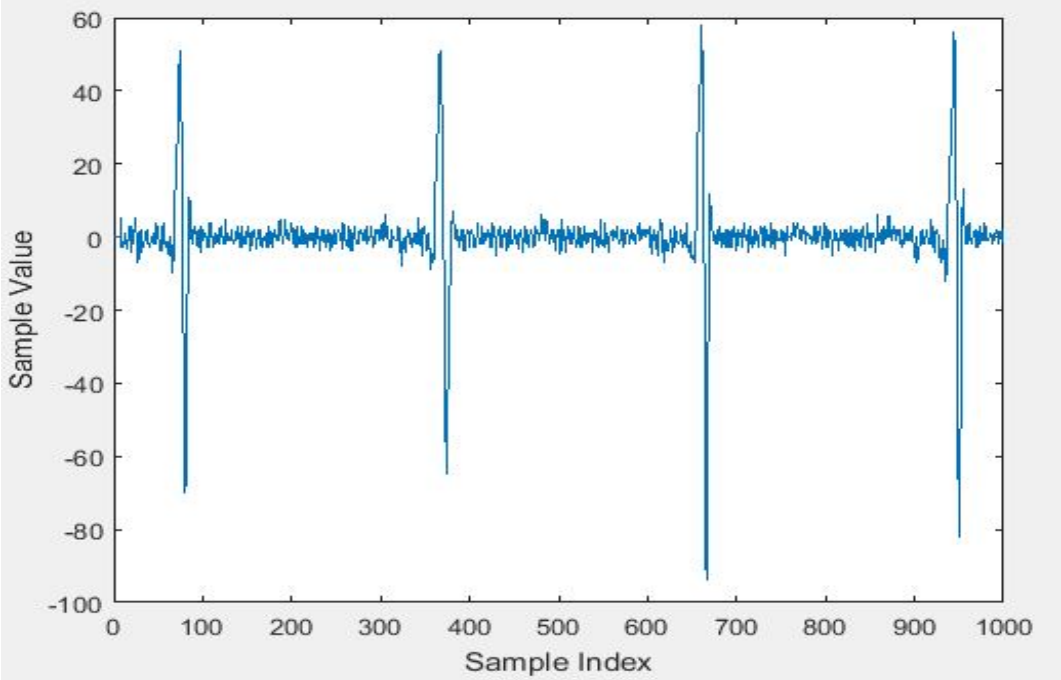


Figure 5.9: The differenced version of the same segment of ECG signal shown in Figure 5.7 which is stationary.

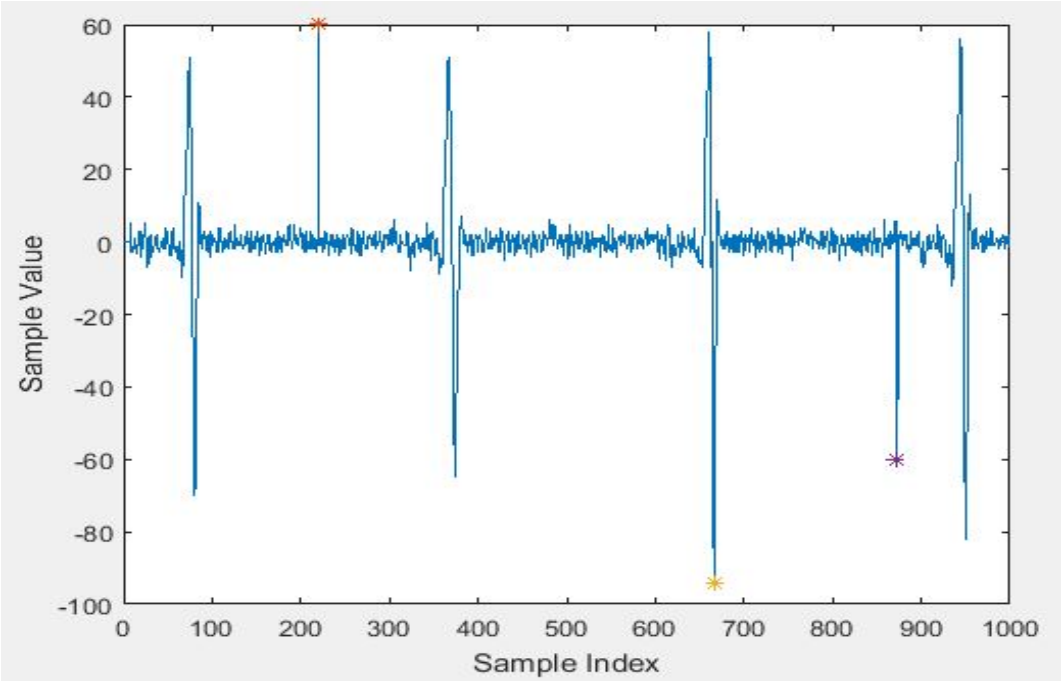
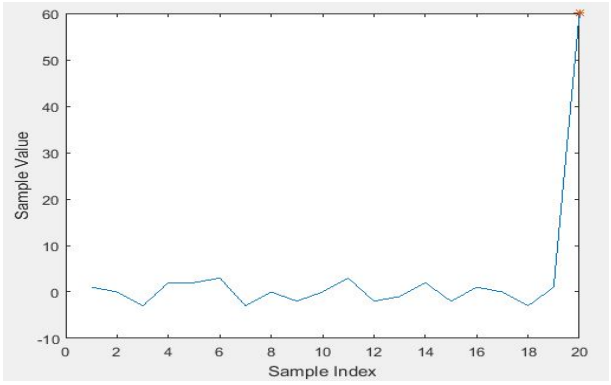
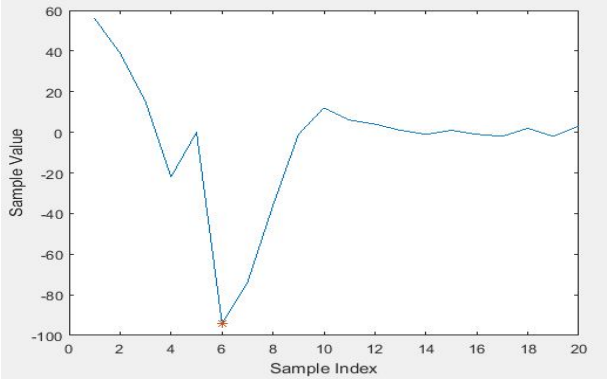


Figure 5.10: The result of the Original Implementation for ECG signal, asterisks are detected outliers. Three outliers are injected at times $t = 220$, $t = 665$ and $t = 873$. It can be seen that three outliers are detected; however one of them is misidentified by one sample.

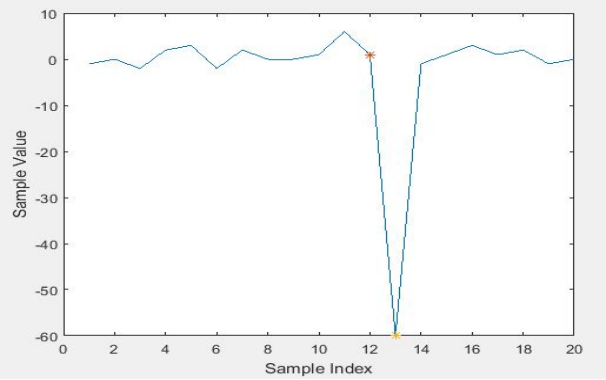
The differenced ECG signal was fed into the Synthesizable Implementation as well and the result can be seen in Figure 5.11. Only the segments with outliers are plotted. The outlier at time $t = 665$ is misidentified by one sample. Also, the sample point at time $t = 872$ is wrongly marked as an outlier. The difference between the results of Original Implementation and Synthesizable Implementation is due to the estimation of the model parameters. The Synthesizable Implementation estimates the parameters using 20 sample points, however the Original Implementation estimates the parameters using all 1000 samples which results in a more accurate approximations. The result from tsoutliers package is also shown in Figure 5.12, and it can be seen that all the outliers are identified correctly.



(a) Samples 201 —220



(b) Samples 661—680



(c) Samples 861—880

Figure 5.11: The result of the Synthesizable Implementation for ECG signal with 3 outliers at times $t = 220$, $t = 665$ and $t = 873$. The time series data is divided into segments of 20 samples and each segment is applied to the algorithm separately. Only the segments with identified outliers are plotted here.

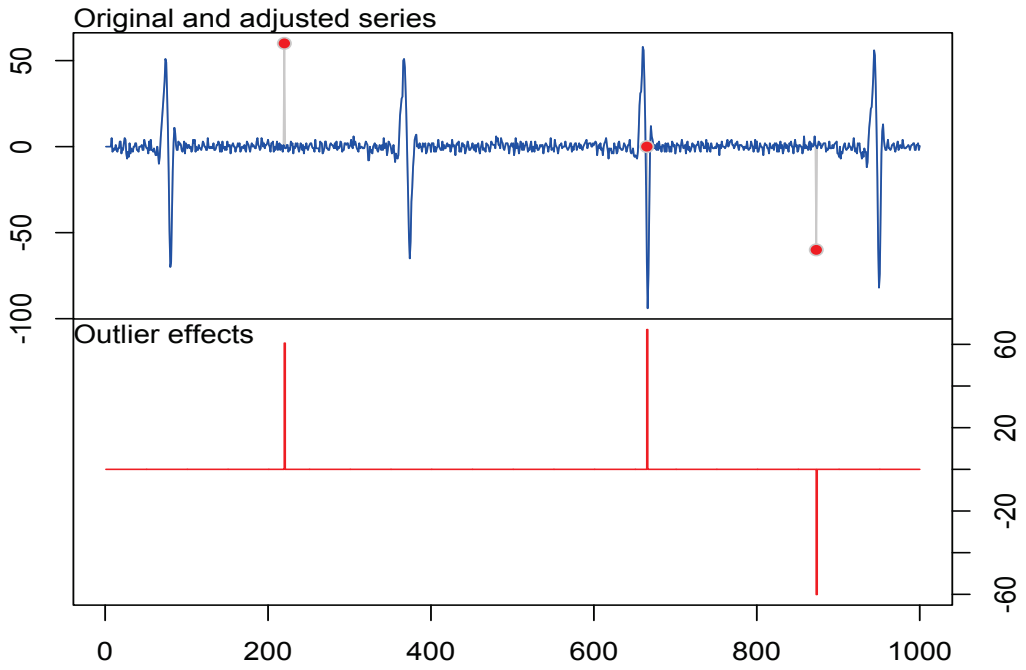


Figure 5.12: The result of tsoutliers package for ECG data with 3 outliers at times $t = 220$, $t = 665$ and $t = 873$. All outliers are detected.

In conclusion, our implementations can detect outliers in ECG signals. However, since they implement a simplified version of the complete algorithm, it is not as accurate as the tsoutliers implementation. Additionally, the Synthesizable Implementation only considers limited number of samples at a time, which affects the estimation of the model parameters and the detection accuracy even more. The detection accuracy of this implementation is presented in the next section.

5.3 Detection Performance

In order to measure the detection accuracy and the false positive rate (i.e., the type I error rate) of the Synthesizable Implementation of the outlier detection algorithm, several experiments on both synthetic data and ECG signals, are conducted. Using the

Table 5.2: Confusion Matrix

	Actual Outlier	**Actual Normal**
Predicted Outlier	True Positive(TP)	False Positive(FP)
Predicted Normal	False Negative(FN)	True Negative(TN)

Confusion Matrix shown in Table 5.2, the accuracy and the false positive rate are calculated for each experiment. Equations 5.1 and 5.2 represent the accuracy and false positive rate, respectively.

$$Accuracy = \frac{TP + TN}{TP + TN + FN + FP}, \tag{5.1}$$

$$FalsePositiveRate = \frac{FP}{TN + FP}. \tag{5.2}$$

Table 5.3: Detection accuracy and false positive rate of the Synthesizable Implementation for various experiments using both ECG signal and Synthetic data

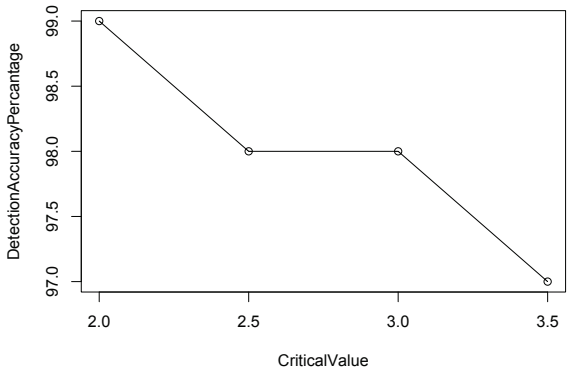
	Percentage of Outliers	**Detection Accuracy**	**False Positive Rate**
ECG Signal	1.4%	100%	0%
ECG Signal	5%	99.8%	0%
ECG Signal	0.3%	99.8%	0.3%
Synthetic Data	4.1%	98.3%	0.8%
Synthetic Data	5.8%	98.3%	0%
Synthetic Data	10%	96.6%	0.9%

Table 5.3 represents the accuracy and false positive rate for several experiments using both ECG and synthetic data. In experiments with the ECG signal, the percentage of outliers are lower than the experiments that use synthetic data and the magnitude of the outliers are close to the peaks of the signal. In the experiments with the synthetic data, outliers with different magnitudes are injected in the time series data. For instance, in the

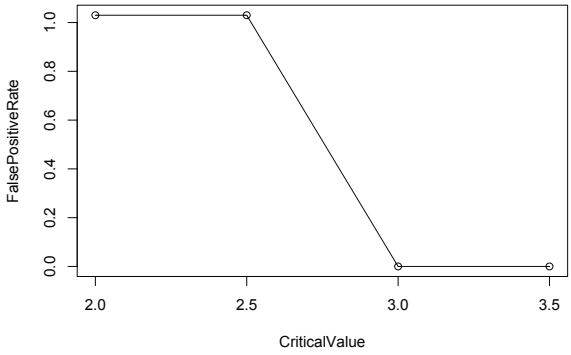
last experiment, 10% of the samples are outliers with different magnitudes. The existance of too many outliers in the time series data affects the model selection step, and results in an ARMA model that fits the data with outliers perfectly, which is very different from the model that represents the outlier-free time series data. Therefore, it is harder to detect outliers. However, it is worth mentioning that the false positive rate is still very low. It can be seen that for ECG signals the detection accuracy is higher and the false positive rate is lower. As a result we can conclude that the magnitude and the number of outliers have direct impact on both the detection accuracy and the false positive rate.

Using the Synthesizable Implementation that considers 20 sample points at a time with 5 iterations of optimization algorithm, the relation between the detection accuracy and false positive rate with factors such as critical value, number of outliers in the time series data and the magnitude of the outliers are analyzed. In order to evaluate the effect of each factor, we used synthetic time series data of 100 sample points and fixed the other factors. It is worth mentioning that the results for the detection accuracy and false positive rate are for this specific experiment. Therefore for a different experiment the values might change, however we expect the impact of the factor to be the same as the reported experiment. Plots depicted in Figure 5.13 show the impact of the critical value on both the detection accuracy and false positive rate. As expected, increasing the critical value decreases both the detection accuracy and the false positive rate.

Figure 5.14 demonstrates the influence of the magnitude of outliers in the time series on the detection accuracy and the false positive rate. For each case, all the outliers had the same magnitude either positive or negative. As the magnitude increases the false positive rate increases as well, which is due to the effect of outliers on the test statistics for normal points. The relation is different for detection accuracy. Outliers with small magnitudes are harder to detect; however, as the magnitude increases past a point, the detection accuracy starts to decrease.

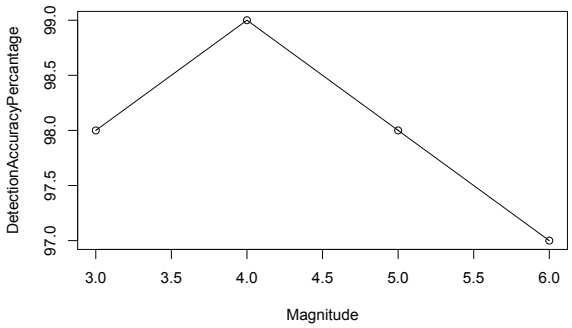


(a) Detection accuracy

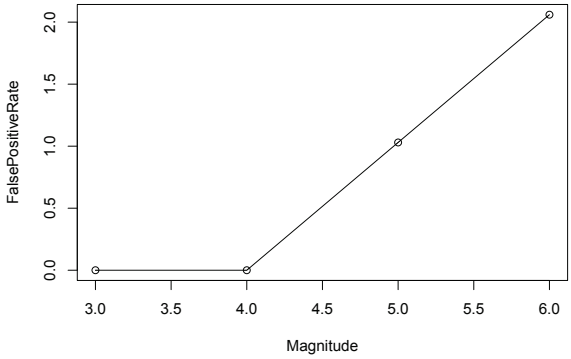


(b) False positive rate

Figure 5.13: The impact of the critical value on both the detection accuracy and false positive rate. As the critical value increases, the chance to mark any point as outlier reduces which decrease both detection accuracy and false positive rate.



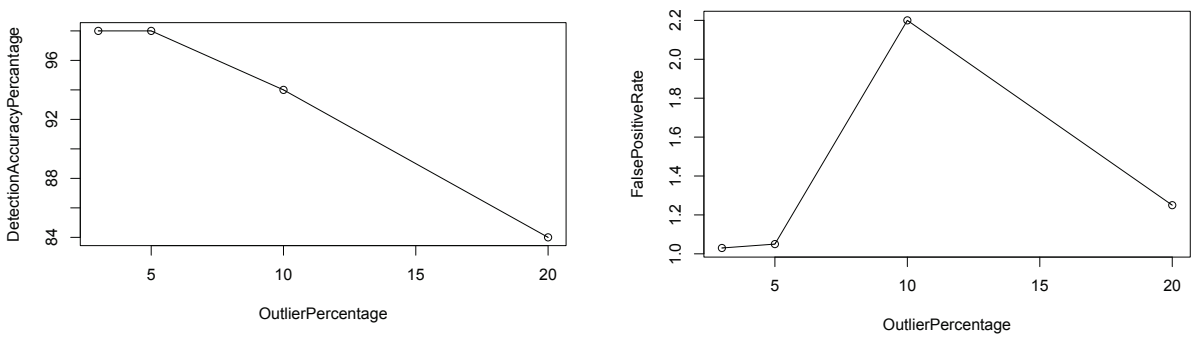
(a) Detection accuracy



(b) False positive rate

Figure 5.14: The impact of the magnitude of the outliers on both the detection accuracy and false positive rate. As the magnitude increases, the detection accuracy decreases and the false positive rate increases.

Figure 5.15 shows the influence of the number of outliers in the time series data on both the detection accuracy and the false positive rate. As the number of outliers increases, the detection accuracy decreases which is due to the effect of outliers on model parameters, and the false positive rate increases due to the effect of outliers on normal points. However, when the percentage of outliers goes from 10% to 20%, there is a huge drop in false positive



(a) Detection accuracy

(b) False positive rate

Figure 5.15: The impact of the number of outliers in the time series data on both the detection accuracy and false positive rate. More outlier points in the time series data results in lower accuracy and higher false positive rate.

rate. In such a case, the model perfectly fits the time series data with outliers which results in lower detection accuracy and false positive rate at the same time.

There is one more factor that can influence the detection accuracy and the false positive rate, which is the number of samples in the time series data. However, since it also affects the area and the resource usage of the hardware implementation, it is discussed in the next section.

5.4 Synthesis Report

We generated synthesizable configurable HDL code for the outlier detection algorithm using HDL Coder. A configuration of the hardware implementation with 5 iterations of optimization algorithm and 20 samples in the time series data was synthesized and fitted on a Stratix V 5SGSED8N3F4514 device FPGA. Quartus Prime PowerPlay power analyzer tool was used for power analysis. The results are summarized in Table 5.4. It can be seen that the number of DSP units needed are 1963, the propagation delay is 571ns and the total

power dissipation is 1.14W. The FPGA implementation processes 35 million data points per second with an energy usage of 0.032 microjoule per processed data point.

Table 5.4: Synthesis results for the outlier detection algorithm.

Device Family	Stratix V
Device	5SGSED8N3F4514
DSP Units	1963
Logic Utilization	11%
Power Dissipation	1.14W
Propagation Delay	571ns
Throughput	35 M data point/sec

Table 5.5 shows the hardware resources required for different configurations along with their detection accuracy and false positive rate for a synthetic data experiment. A time series of 100 samples was used as test in which 5% of the samples were outliers with different magnitudes. It is worth mentioning that the values for detection accuracy and the false positive rate are calculated for this specific test. As a result, the false positive rates that are reported 0% for configurations with 50 samples are not absolutely zero and other experiments may result in non-zero error rates. Similarly the values for detection accuracy might change for different test as well. This is just the comparison of the detection accuracy and false positive rate between different configurations using one experiment. The first number in the configuration shows the number of samples in the time series data, and the second number is the number of iterations used in the optimization algorithm.

Table 5.5: Comparison of hardware requirements, accuracy and false positive rate for different configurations.

Configuration	20x2	20x5	50x2	50x5
DSP Units	1371	2622	4437	4733
Logic Utilization	4%	4.5%	10.9%	16.5%
Detection Accuracy	96%	99%	98%	99%
False Positive Rate	4.21%	1.05%	0%	0%

It can be seen that there is a trade-off between the area of the hardware and the accuracy of the algorithm. Considering more sample points results in higher detection accuracy and lower false positive rate. Also it can be seen that the number of iterations in optimization algorithm has the same impact. We used 20 sample points for the FPGA implementation since it is gives us a reasonable detection accuracy and false positive rate and it has less delay compared to 50 samples.

Table 5.6: Detection performance comparison with the FPGA implementations proposed in [5] and [25].

	Detection Accuracy	**False Positive Rate**
The Synthesizable Implementation	99%	1.05%
Intrusion Detection from[5]	Up to 99%	1.95%
Intrusion Detection from[25]	78%	3.9%

Finally, Table 5.6 shows the comparison of the detection accuracy and the false positive rate between our FPGA implementation and the FPGA-based outlier detection techniques proposed in [5] and [25]. It is worth mentioning that the FPGA implementations represented in [5] and [25] are for intrusion detection and use a different type of technique and

input data. However this comparison shows that the detection accuracy and false positive rate of our implementation is similar or slightly better than the existing outlier detection implementations on FPGA. We used the detection accuracy and false positive rate for the experiment with 5% outliers in time series data where the magnitude of the outliers are different.

5.5 Throughput and Power Analysis

We calculated the power consumption of the software implementation by running tsoutliers package on a 2.20GHz Intel Corei7 processor with 16GB of memory and 4MB cache, running the Ubuntu 16.04 LTS kernel version 4.4.0 operating system. We calculated the static power at ideal state through powerstat[1] tool to be 10.7W. We used a synthetic data experiment with 20 sample points and we only selected AOs for detection, however we ran this experiment in a loop with 100 iterations. After running the algorithm in R, the power consumption reached 21.5W. Therefore the estimated power consumption for the algorithm in R is 10.8W.

We also calculated the runtime for the algorithm in R to be 0.04s using the linux time command. This is the runtime for running the same synthetic data experiment that was used for FPGA implementation just once. The experiment uses synthetic data with 20 samples in the time series data and only AOs are selected for detection process. However using another input time series data with different number of outliers might change the run time. In summary, the FPGA implementation reduces the power consumption by 89% and increases the throughput by 99% compared to the implementation in tsoutliers package.

Table 5.7 shows the comparison between the power consumption and power reduc-

[1]Powerstat is the power consumption calculator for Ubuntu linux. It measures the power consumption of a computer that has a battery power source or supports the RAPL (Running Average Power Limit) interface.

Table 5.7: Power consumption comparison with the FPGA implementations proposed in [30].

	Power Consumption	**Power Reduction**
The Synthesizable Implementation	1.14W	9.5 times
Anomaly Detection from[30]	0.3W	5 times

tion compared to the software implementation of our FPGA implementation compared to the FPGA-based anomaly detection proposed in [30]. Although the FPGA implementation in [30] is more power efficient but they only reduced the power consumption 5 times compared to the corresponding software implementation of their technique. Also, the detection accuracy of their FPGA implementation is not evaluated at all.

Chapter 6

Conclusion

In this thesis, we implemented a configurable time series outlier detection technique in hardware. The number of iterations for the optimization algorithm, number of samples in the time series data and the critical value are the design parameters that can be adjusted. The goal was to present a hardware implementation that is more power efficient compared to the general purpose processors required to run the tsoutliers package which is a software implementation of the same algorithm. A configuration of our hardware implementation was synthesized and fitted on an Altera Stratix V 5SGSED8N3F4514 device which reduces the power consumption by 89% compared to the the tested Intel Corei7 processor running tsoutliers. Additionally, the FPGA implementation has the detection accuracy of 99% and the false positive rate of 1.05%. We examined the impact of design parameters on the performance of the algorithm in Chapter 5. Results show that more iterations in the optimization technique results in higher accuracy for model parameters and at the same time increases the area. Critical value influences the accuracy of detecting outliers, lower values result in a higher false positives rate while higher values decrease the detection accuracy.

The outlier detection technique fits an ARMA model to the data and approximates

the residuals. Then, using the model parameters and residuals, it calculates test statistics to determine the outlier score for each sample. It compares the test statistics and the critical value, and it identifies the outliers. The main drawback of the FPGA implementation is that only ARMA(1,1) models are considered in the ARMA modeling procedure, and the procedure is still computationally complex as it requires 1963 DSP units. Using only ARMA(1,1) decreases the detection accuracy and increases the false positive rate of the algorithm.

Finally, we showed that the FPGA implementation is capable of detecting additive noise artifacts in ECG signals and is capable of processing 35 million data point per second with an energy usage of 0.032 microjoule per processed data point.

6.1 Future Work

Our FPGA implementation only contains the first stage of the original algorithm proposed in [9]. One approach to improve the detection accuracy and false positive rate is to implement the second and third stages of the algorithm which remove the effect of detected outliers and modifies the model parameters. The idea is to remove the effect of outliers, and re-estimate the model parameters using ARMA(1,1). This approach can help in dealing with both fake and masked outliers, and increases the detection accuracy while reduces the false positive rate.

The current version of the hardware implementation is completely combinational and processes a fixed number of samples in one clock cycle. Consequently, it requires multiple instantiations of the same hardware for one iteration of the optimization algorithm. However, a sequential implementation can reuse the hardware for one iteration in consecutive clock cycles. This approach will reduce the resource utilization, specially the number of required DSP units by reusing them in consecutive clock cycles. However, this would result in an

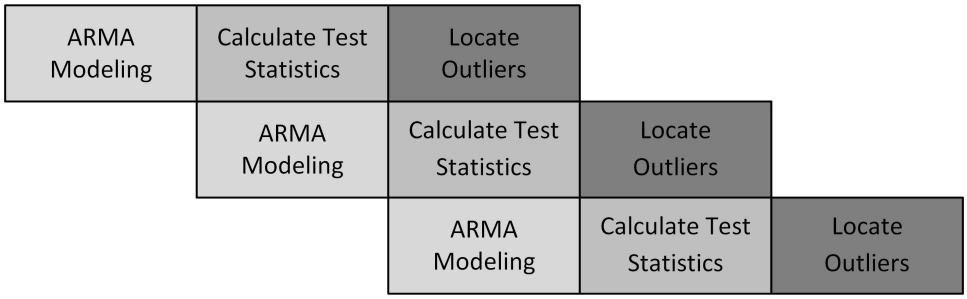


Figure 6.1: Different parts of the design can be pipelined which improves the throughput while the latency is the same.

increase in the latency of detection.

Additionally, we can improve the amount of processed data points per second by using pipelining and parallelisem. This can be achieved by dividing the design in three pipes, ARMA modeling, calculating test statistics and locating outliers. Each pipe execute a portion of the algorithm and sends the results to the next stage. This means that instead of processing n sample points at a time, we can process $3 * n$ samples when the pipeline is full. Figure 6.1 shows the general idea and how to divide the algorithm into three pipes.

Also, using the verilog implementation of the algorithm, we can implement the algorithm in ASICs. The verilog implementation of the algorithm is generated using HDL Coder which can be used to generate layout of the design in tools such as Cadence. The ASIC implementation can reduce the power consumption and improve the propagation delay compared to the FPGA implementation. The ASIC implementation can have application domains different to the FPGA one. For instance the FPGA implementation can be used in server farms while the ASIC implementation suits better in wearable devices.

Finally for a better comparison of the power and throughput, the implementation of the complete algorithm on a microprocessor can be compared to the FPGA implementation. However, implementing the complete algorithm on a microprocessor might not be feasible because the tsoutliers package itself contains thousand lines of code and it is also using couple of functions from other packages.

Bibliography

[1] Emin Aleskerov, Bernd Freisleben, and Bharat Rao. Cardwatch: A Neural Network Based Database Mining System for Credit Card Fraud Detection. In *Computational Intelligence for Financial Engineering (CIFEr), 1997., Proceedings of the IEEE/IAFE 1997*, pages 220–226. IEEE, 1997.

[2] Mahashweta Das and Srinivasan Parthasarathy. Anomaly Detection and Spatio-Temporal Analysis of Global Climate System. In *Proceedings of the third international workshop on knowledge discovery from sensor data*, pages 142–150. ACM, 2009.

[3] Elizabeth Wu, Wei Liu, and Sanjay Chawla. Spatio-Temporal Outlier Detection in Precipitation Data. In *Knowledge discovery from sensor data*, pages 115–133. Springer, 2010.

[4] Clay Spence, Lucas Parra, and Paul Sajda. Detection, Synthesis and Compression in Mammographic Image Analysis with A Hierarchical Image Probability Model. In *Mathematical Methods in Biomedical Image Analysis, 2001. MMBIA 2001. IEEE Workshop on*, pages 3–10. IEEE, 2001.

[5] Abhishek Das, David Nguyen, Joseph Zambreno, Gokhan Memik, and Alok Choudhary. An FPGA-Based Network Intrusion Detection Architecture. *IEEE Transactions on Information Forensics and Security*, 3(1):118–132, 2008.

[6] Manish Gupta, Jing Gao, Charu C Aggarwal, and Jiawei Han. Outlier Detection for Temporal Data: A Survey. *IEEE Transactions on Knowledge and Data Engineering*, 26(9):2250–2267, 2014.

[7] George E P Box, Gwilym M Jenkins, and Gregory C Reinsel. *Time Series Analysis: Forecasting & Control*. 1994.

[8] Ih Chang, George C Tiao, and Chung Chen. Estimation of Time Series Parameters in the Presence of Outliers. *Technometrics*, 30(2):193–204, 1988.

[9] Chung Chen and Lon-Mu Liu. Joint Estimation of Model Parameters and Outlier Effects in Time Series. *Journal of the American Statistical Association*, 88(421):284–297, 1993.

[10] Javier López-de Lacalle. tsoutliers R Package for Detection of Outliers in Time Series. Retrieved from: `https://jalobe.com/doc/tsoutliers.pdf`, 2016.

[11] Douglas M Hawkins. *Identification of Outliers*, volume 11. Springer, 1980.

[12] Vic Barnett and Toby Lewis. *Outliers in Statistical Data.*, volume 3. New York: Wiley, 1994.

[13] Frank E Grubbs. Procedures for Detecting Outlying Observations in Samples. *Technometrics*, 11(1):1–21, 1969.

[14] Varun Chandola, Arindam Banerjee, and Vipin Kumar. Anomaly Detection: A Survey. *ACM Computing Surveys(CSUR)*, 41(3):99–126, 2009.

[15] Victoria J Hodge and Jim Austin. A Survey of Outlier Detection Methodologies. *Artificial intelligence review*, 22(2):85–126, 2004.

[16] G Silvestri, F Bini Veronat, M Innocenti, and M Napolitano. Fault Detection Using Neural Networks. *IEEE World Congress on Computational Intelligence*, 6:3796–3799, 1994.

[17] Milos Hauskrecht, Iyad Batal, Michal Valko, Shyam Visweswaran, Gregory F Cooper, and Gilles Clermont. Outlier Detection for Patient Monitoring and Alerting. *Journal of Biomedical Informatics*, 46(1):47–55, 2013.

[18] David J. Hill and Barbara S. Minsker. Anomaly Detection in Streaming Environmental Sensor Data: A Data-Driven Modeling Approach. *Environmental Modelling and Software*, 25(9):1014–1022, 2010.

[19] Junshui Ma and Simon Perkins. Online Novelty Detection on Temporal Sequences. *Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining - KDD '03*, pages 613–618, 2003.

[20] Sabyasachi Basu and Martin Meckesheimer. Automatic Outlier Detection for Time Series: An Application to Sensor Data. *Knowledge and Information Systems*, 11(2):137–154, 2007.

[21] Anthony J Fox. Outliers in Time Series. *Journal of the Royal Statistical Society . Series B (Methodological)*, pages 350–363, 1972.

[22] Bovas Abraham and George E P Box. Bayesian Analysis of Some Outlier Problems in Time Series. *Biometrika*, 66(2):229–236, 1979.

[23] Lorraine Denby and R Douglas Martin. Robust Estimation of the First-Order Autoregressive Parameter. *Journal of the American Statistical Association*, 74(365):140–146, 1979.

[24] Ruey S Tsay. Time Series Model Specification in the Presence of Outliers. *Journal of the American Statistical Association*, 81(393):132–141, 2016.

[25] Taner Tuncer and Yetkin Tatar. FPGA Based Programmable Embedded Intrusion Detection System. In *Proceedings of the 3rd international conference on Security of information and networks*, pages 245–248. ACM, 2010.

[26] Roy De Maesschalck, Delphine Jouan-Rimbaud, and Désiré L Massart. The Mahalanobis Distance. *Chemometrics and intelligent laboratory systems*, 50(1):1–18, 2000.

[27] Ami Hayashi, Yuta Tokusashi, and Hiroki Matsutani. A Line Rate Outlier Filtering FPGA NIC Using 10GbE Interface. *ACM SIGARCH Computer Architecture News*, 43(4):22–27, 2016.

[28] Yuto Arai, Shin'ichi Wakabayashi, Shinobu Nagayama, and Masato Inagi. An Efficient FPGA Implementation of Mahalanobis Distance-Based Outlier Detection for Streaming Data. In *Field-Programmable Technology (FPT), 2016 International Conference on*, pages 257–260. IEEE, 2016.

[29] Ami Hayashi and Hiroki Matsutani. An FPGA-Based In-NIC Cache Approach for Lazy Learning Outlier Filtering. In *Parallel, Distributed and Network-based Processing (PDP), 2017 25th Euromicro International Conference on*, pages 15–22. IEEE, 2017.

[30] Duncan JM Moss, Zhe Zhang, Nicholas J Fräser, and Philip HW Leong. An FPGA-Based Spectral Anomaly Detection System. In *Field-Programmable Technology (FPT), 2014 International Conference on*, pages 175–182. IEEE, 2014.

[31] Elliott Sober. The Principle of Parsimony. *The British Journal for the Philosophy of Science*, 32(2):145–156, 1981.

[32] John W Tukey. Discussion, Emphasizing the Connection Between Analysis of Variance and Spectrum Analysis. *Technometrics*, 3(2):191–219, 1961.

[33] Søren Bisgaard and Murat Kulahci. *Time Series Analysis and Forecasting by Example*. John Wiley & Sons, 2011.

[34] Rob J Hyndman, Mitchell O'Hara-Wild, Christoph Bergmeir, Slava Razbash, Earo Wang, and Maintainer Rob Hyndman. Package 'forecast'. Retrieved from: `https://cran.r-project.org/web/packages/forecast/forecast.pdf`, 2017.

[35] Skipper Seabold and Josef Perktold. Statsmodels: Econometric and Statistical Modeling with Python. In *9th Python in Science Conference*, 2010.

[36] Denise R Osborn. On The Criteria Functions Used for the Estimation of Moving Average Processes. *Journal of the American Statistical Association*, 77(378):388–392, 1982.

[37] Per Christian Hansen, Víctor Pereyra, and Godela Scherer. *Least Squares Data Fitting with Applications*. JHU Press, 2013.

[38] Ashok D Belegundu and Tirupathi R Chandrupatla. *Optimization Concepts and Applications in Engineering*. Cambridge University Press, 2011.

[39] Robert Hooke and Terry A Jeeves. "Direct Search" Solution of Numerical and Statistical Problems. *Journal of the ACM (JACM)*, 8(2):212–229, 1961.

[40] Robert Michael Lewis, Virginia Torczon, and Michael W Trosset. Direct Search Methods: Then and Now. *Journal of computational and Applied Mathematics*, 124(1):191–207, 2000.

[41] Predrag S Stanimirovic, Milan B Tasic, and Miroslav Ristic. Symbolic Implementation of the Hooke-Jeeves Method·. 1999.

[42] George G Roussas. *An Introduction to Probability and Statistical Inference*. Academic Press, 2014.

[43] Christophe Leys, Christophe Ley, Olivier Klein, Philippe Bernard, and Laurent Licata. Detecting Outliers: Do Not Use Standard Deviation Around the Mean, Use Absolute Deviation Around the Median. *Journal of Experimental Social Psychology*, 49(4):764–766, 2013.

[44] J Bednar and T Watt. Alpha-Trimmed Means and Their Relationship to Median Filters. *IEEE Transactions on acoustics, speech, and signal processing*, 32(1):145–153, 1984.

[45] André Lourenço, Hugo Silva, Carlos Carreiras, et al. Outlier Detection in Non-Intrusive ECG Biometric System. In *International Conference Image Analysis and Recognition*, pages 43–52. Springer, 2013.

[46] Rob J. Hyndman and Yeasmin Khandakar. Automatic Time Series Forecasting: The Forecast Package for R. *Journal Of Statistical Software*, 27(3):C3–C3, 2008.

[47] MathWorks. Econometrics Toolbox. `https://www.mathworks.com/products/econometrics.html`, 2016.

[48] MathWorks. HDL Coder. `https://www.mathworks.com/products/hdl-coder.html`, 2016.

[49] MathWorks. *HDL Coder Users Guide*, 2016. Available at `http://www.mathworks.com/help/pdf_doc/hdlcoder/hdlcoder_ug.pdf`, version 3.11.

[50] Lena Biel, Ola Pettersson, Lennart Philipson, and Peter Wide. ECG Analysis: A New Approach in Human Identification. *IEEE Transactions on Instrumentation and Measurement*, 50(3):808–812, 2001.

[51] Ary L Goldberger, Luis AN Amaral, Leon Glass, Jeffrey M Hausdorff, Plamen Ch Ivanov, Roger G Mark, Joseph E Mietus, George B Moody, Chung-Kang Peng, and H Eugene Stanley. Physiobank, Physiotoolkit, and Physionet. *Circulation*, 101(23):e215–e220, 2000.

[52] George B Moody and Roger G Mark. The Impact of the MIT-BIH Arrhythmia Database. *IEEE Engineering in Medicine and Biology Magazine*, 20(3):45–50, 2001.

Appendix A

MATLAB Codes

Both the MATLAB implementation of the simplified algorithm (i.e., Original Implementation) and the modified MATLAB implementation for the synthesized configuration (i.e., Synthesizable Implementation) are presented.

A.1 Matlab Implemetation of the Simplified Algorithm

The top-level function that receives the design parameters is called OutlierDetection. It also has 4 optional arguments which are the required variables for optimization algorithm. If these arguments are not provided, it will use the default values defined in the first couple of lines. It is also worth mentioning that the user should either provide all of the optional arguments or none of them at all. This function connects all the other functions and outputs the indexes of the outlier points in an array called Outlier_index. Also, It plots the time series data while marking the outliers with an asterisk.

```
function Outlier_index = OutlierDetection(Input,cval, a, P_vector, P_tolerance, Beta)
```

```
% Use default value for the optinal arguments if they are not provided
if nargin < 3
    a=2;
    P_vector = [0.2,0.2];
    P_tolerance = [0.001,0.001];
    Beta  = [0.8, -0.3];
end
[n, r] = size(Input);
% using Pattern search algorithm for parameter estimation
Coeffs = PatternSearch(Input, n, a, P_vector, P_tolerance, Beta);
% calculating residuals based on the model parameters
resid = Residuals(Input, Coeffs(1),Coeffs(2), n);
% Calculates the pi polynomials coefficients
AR = Coeffs(1);
MA = Coeffs(2);
pi_Coeffs = ARMAtoMA(-MA, -AR, n-1);
pi_Coeffs = [1; pi_Coeffs];
pi_Coeffs_Rev = pi_Coeffs(end:-1:1);
resid_sorted = Sorting(resid, n); % Sorting the residuals
resid_revised = resid_sorted(2:n-1); % Truncating the residuals
resid_mean = 0;
% Estimating the residuals standard deviation using the truncated residuals
for k = 1:n-2
    resid_mean = resid_mean + resid_revised(k);
end
resid_mean = resid_mean /(n-2);
sum = 0;
for m=1:n-2
    sum = sum + (resid_revised(m)-resid_mean)^2;
end
sigma_revised = sqrt(sum/(n-3));
% Calculating the denominator and numerator for test statistics
xy = zeros(n, 1);
```

```
    xx = zeros(n, 1);
    for i = 1:n
        numerator = 0;
        denominator = 0;
        for j= i:n
            denominator = pi_Coeffs_Rev(j)^2+ denominator;
            numerator = resid(j)*pi_Coeffs(j-i+1) + numerator;
        end
        xy(i) = numerator;
        xx(i)= denominator;
    end
    tauAO = xy ./(sigma_revised*sqrt(xx));
    Outlier_index = LocateOutliers(tauAO, n,cval); % Locating outliers
    %plotting and Printing
    plot(Input);
    xlabel('Time');
    ylabel('Sample Value')
    hold on;
    for i=1:n
        if Outlier_index(i)~= 0
            plot(Outlier_index(i),Input(Outlier_index(i)),'*');
        end
    end
end
```

the PatternSearch function receives the input from top-level function and fits an ARMA(1,1) to the data and estimates the parameters. It outputs the coefficients of the model.

```
function Coeffs = PatternSearch(Input, n, a, P_vector, P_tolerance, Beta)
I_P_vector = P_vector;
Pre_Beta = Beta;
% Calculating objective function for starting point
```

```
I_Value = ObjFunction(Input, Beta(1), Beta(2),n);
Temp = Beta;
Success = false;
Pattern_Move = false;
Move_Success = false;
Continue = false;
Max_Iter = 10;
for k=1:Max_Iter
    % using both maximum iteration and perturbation tolerance as stopping strategy
    if (P_vector(1) >= P_tolerance(1) && P_vector(2)>=P_tolerance(2))
        for i=1:2 %Exploratory Search
            Temp(i)= Beta(i) + P_vector(i);
            F_Value = ObjFunction(Input, Temp(1), Temp(2), n);
            if F_Value < I_Value
                 Success = true;
            else
                Temp(i)= Beta(i) - P_vector(i);
                F_Value = ObjFunction(Input, Temp(1), Temp(2), n);
                if F_Value < I_Value
                    Success = true;
                else
                    Success = false;
                end
            end
            if (Success == true)
                Pre_Beta(i) = Beta(i);
                Beta(i) = Temp(i);
                I_Value = F_Value;
                Pattern_Move = true;
                Success = false;
            else
                Temp(i) = Beta(i);
                Success = false;
```

```
    end
end
if Pattern_Move == true %Pattern Move
    Pattern_Move = false;
    P_vector = I_P_vector; %reset Perturbation vector
    Continue = true;
    for m=1:Max_Iter
        if Continue == true
            Tentative_Beta = (a*Beta) - Pre_Beta; % Calculate the tentative point
            Continue = false;
            for i=1:2 % Exploratory Search around tentative point
                Temp(i)= Tentative_Beta(i) + P_vector(i);
                F_Value = ObjFunction(Input, Temp(1), Temp(2), n);
                if F_Value < I_Value
                    Move_Success = true;
                else
                    Temp(i)= Tentative_Beta(i) - P_vector(i);
                    F_Value = ObjFunction(Input, Temp(1), Temp(2), n);
                    if F_Value < I_Value
                        Move_Success = true;
                    else
                        Move_Success = false;
                    end
                end
                if (Move_Success == true)
                    Pre_Beta(i) = Beta(i);
                    Beta(i) = Temp(i);
                    I_Value = F_Value;
                    Continue = true;
                    Move_Success = false;
                else
                    Temp(i) = Beta(i);
                    Move_Success = false;
```

```
                    end
                end
            end
        end
    else %No Pattern Move
        P_vector = P_vector / 2; % Modify the perturbation vector to half of its value
    end
  end
end
Coeffs = Beta;
end
```

The ObjFunction is the function that calculates the value of objective function. It receives the coefficients from the PatternSearch function and evaluates the objective function.

```
function FuncValue = ObjFunction (Input, Phi, Theta, n)
  a_noise = zeros(n+1, 1);
  x_data = [0 ; Input];
  Temp = 0;
  for i = 2:n+1 % Estimating the noise terms based on the coefficients provided
      a_noise(i) = x_data(i)- (Phi*x_data(i-1))+ (Theta*a_noise(i-1));
  end
  for j= 2:n+1 % calculating the objective function
      Temp = Temp + (a_noise(j)^2);
  end
  FuncValue = Temp;
end
```

The Residuals function receives the estimated coefficients from top-level function and returns the noise terms using the ARMA(1,1) formulation.

```
function Resid = Residuals(Input, Phi, Theta, n)
   Resid = zeros(n, 1);
   a_noise = zeros(n+1, 1);
   x_data = [0 ; Input];

   for i = 2:n+1 % Calculating the residuals based on the model coefficients
       a_noise(i) = x_data(i)- (Phi*x_data(i-1))+ (Theta*a_noise(i-1));
   end
   Resid(1:n) = a_noise(2:n+1);
end
```

The ARMAtoMA function receives the estimated coefficients and calculates the coefficients of the π polynomial.

```
function pi_Coeffs = ARMAtoMA(ar,ma,lag)
 p = length(ar);
 q = length(ma);
 m = lag;
 pi_Coeffs = zeros(m,1);
 phi = ar;
 theta = ma;
 % Determining the pi coefficients based on the coefficients of phi and theta polynomials
 for i = 1:m
     if (i < q+1)
         tmp = theta(i);
     else
         tmp = 0.0;
     end
     for j = 1:(min(i+1,p+1)-1)
         if ((i-j-1)>=0)
             tmp = tmp + phi(j)*pi_Coeffs(i-j);
         else
```

```
            tmp = tmp + phi(j);
        end
    end
    pi_Coeffs(i) = tmp;
end
```

The Sorting function sorts the residuals using the bubble sort algorithm. It receives the residuals from top-level function and outputs the sorted version of the residuals.

```
function sorted = Sorting(Input, n)
n = size(Input);
my_Vector = Input;
for i = n-1:-1:1 %Sorting residuals based on bubble sort algorithm
    for j = 1:i
        if my_Vector(j)> my_Vector(j+1)
            temp = my_Vector(j);
            my_Vector(j)= my_Vector(j+1);
            my_Vector(j+1)=temp;
        end
    end
end
sorted = my_Vector;
end
```

The LocateOutlier function receives the test statistics that are calculated in the top-level function and the critical value. It outputs the Outlier_index based on the comparisons of the test statistics and the critical value.

```
function indAO = LocateOutliers(tauAO, n,cval)
    j = 1;
    temp = zeros(n,1);
```

```
    % locating outliers based on the comparison of test statistics with critical value
    for i= 1:n
        if abs(tauAO(i))> cval
           temp(j) = i;
           j = j+1;
        end
    end
    indAO = temp;
end
```

A.2 Synthesizable MATLAB Implementation

The Synthesizable Implementation uses a fixed number of samples in the time series data which is 20 in this configuration. Therefore, all the n's are replaced by 20. The OutlierDetection function is the top-level function that receives the input and critical value and outputs a one for the sample indexes that are outlier.

```
function Outlier_index = OutlierDetection(Input, cval)
    % Using Pattern search algorithm for parameter estimation
    Coeffs = Patternsearch(Input);
    % Calculating residuals based on the model parameters
    resid = Residuals(Input, Coeffs(1),Coeffs(2));
    % Calculating pi polynimials coefficients
    AR = Coeffs(1);
    MA = Coeffs(2);
    pi_Coeffs = ARMAtoMA(-MA, -AR);
    pi_Coeffs = [1; pi_Coeffs];
    pi_Coeffs_Rev = pi_Coeffs(end:-1:1);
    resid_sorted = Sorting(resid); % Sorting the residuals
    resid_revised = resid_sorted(2:19); % Truncating the residuals
```

```
    % Extimating the residuals standard deviation using the truncated residuals
    resid_mean = 0;
    for k = 1:18
        resid_mean = resid_mean + resid_revised(k);
    end
    resid_mean = resid_mean /18;
    sum = 0;
    for m=1:18
        sum = sum + (resid_revised(m)-resid_mean)^2;
    end
    sigma_revised = sqrt(sum/7);
    % Calculating the denominator and numerator for test statistics
    xy = zeros(20, 1);
    xx = zeros(20, 1);
    for i = 1:20 %xy and xx calculation
        numerator = 0;
        denominator = 0;
        for j= i:20
            denominator = pi_Coeffs_Rev(j)^2+ denominator;
            numerator = resid(j)*pi_Coeffs(j-i+1) + numerator;
        end
        xy(i) = numerator;
        xx(i)= denominator;
    end
    tauAO = xy ./(sigma_revised*sqrt(xx));
    Outlier_index = LocateOutliers(tauAO, cval); % Locating outliers
end
```

The Patternsearch function calls the one_Block function 5 times, which means that the optimization algorithm has 5 iterations. It is worth mentioning that the function calls are separate and not in a for loop to keep the code compatible with HDL Coder. Also the initial variables of the optimization algorithm are defined in this function.

```
function Coeffs = Patternsearch(Input)
P_Vector = [0.2,0.2]; % An arbitrary perturbation vector
I_P_Vector = P_Vector;
Beta_M(1,:)= [0.8,-0.3]; % An arbitrary starting point
Beta_M = zeros(6,2);
I_Value_M = zeros(6,1);
P_Vector_M = zeros(6,2);
% Calculating objective function for starting point
I_Value_M(1) = ObjFunction(Input, Beta_M(1,1), Beta_M(1,2));
P_Vector_M(1,:) = I_P_Vector;

% Instantiating one iteration for 5 times
[Beta_M(2,:),I_Value_M(2),P_Vector_M(2,:)]= One_Block(Beta_M(1,:), I_Value_M(1),
    P_Vector_M(1,:), Input, I_P_Vector);
[Beta_M(3,:),I_Value_M(3),P_Vector_M(3,:)]= One_Block(Beta_M(2,:), I_Value_M(2),
    P_Vector_M(2,:), Input, I_P_Vector);
[Beta_M(4,:),I_Value_M(4),P_Vector_M(4,:)]= One_Block(Beta_M(3,:), I_Value_M(3),
    P_Vector_M(3,:), Input, I_P_Vector);
[Beta_M(5,:),I_Value_M(5),P_Vector_M(5,:)]= One_Block(Beta_M(4,:), I_Value_M(4),
    P_Vector_M(4,:), Input, I_P_Vector);
[Beta_M(6,:),I_Value_M(6),P_Vector_M(6,:)]= One_Block(Beta_M(5,:), I_Value_M(5),
    P_Vector_M(5,:), Input, I_P_Vector);

Coeffs = Beta_M(6,:);
end
```

The one_Block function receives the outputs of the previous iteration as inputs, and outputs the inputs for the next iteration. This function calls both ExploratorySearch and Pattern-Move functions.

```
function [N_Beta,N_I_Value,N_P_Vectore]=One_Block(Beta,I_Value,P_Vector,Input,I_P_Vector)
```

```
Final_Beta = Beta;
Final_I_Value = I_Value;
Final_P_Vector = P_Vector;
% intsntiating both Exploratory search and pattern move blocks
[Cal_Beta, Cal_I_Value, Pattern_Move]= ExploratorySearch(Beta, P_Vector,I_Value, Input);
[Pcal_Beta, Pcal_I_Value] = PatternMove(Beta, Cal_Beta, Cal_I_Value, Input);
% Multiplexers that assign the outputs
if Pattern_Move == 1
    Final_Beta = Pcal_Beta;
    Final_I_Value = Pcal_I_Value;
    Final_P_Vector = I_P_Vector;
else
    Final_Beta = Cal_Beta;
    Final_I_Value = Cal_I_Value;
    Final_P_Vector = P_Vector/2;
end

N_Beta = Final_Beta;
N_I_Value = Final_I_Value;
N_P_Vectore = Final_P_Vector;
end
```

The ExploratorySearch function receives the current most optimized point and looks for a new point that optimizes the objective function using a exploratory search.

```
function [Cal_Beta, Cal_I_Value, Pattern_Move]= ExploratorySearch(Beta,
    P_vector,I_Value, Input)
Cal_Pattern_Move =0;
Success = 0;
Cal_Beta = zeros(1,2);
Temp = Beta;
Best_Value = I_Value;
```

```
Best_Beta = Beta;

% ExploratorySearch for the first element of the coefficients
Temp(1)= Beta(1) + P_vector(1);
F_Value = ObjFunction(Input, Temp(1), Temp(2));
if F_Value < Best_Value
    Success = 1;
else
    Temp(1)= Beta(1) - P_vector(1);
    F_Value = ObjFunction(Input, Temp(1), Temp(2));
    if F_Value < Best_Value
        Success = 1;
    else
        Success = 0;
    end
end
if (Success == 1)
    Best_Beta(1) = Temp(1);
    Best_Value = F_Value;
    Cal_Pattern_Move = 1;
    Success = 0;
else
    Best_Beta(1) = Beta(1);
    Best_Value = I_Value;
    Success = 0;
    Cal_Pattern_Move = 0;
end

% ExploratorySearch for the second element of the coefficients
Temp(2)= Beta(2) + P_vector(2);
F_Value = ObjFunction(Input, Temp(1), Temp(2));
if F_Value < Best_Value
    Success = 1;
```

```
else
    Temp(2)= Beta(2) - P_vector(2);
    F_Value = ObjFunction(Input, Temp(1), Temp(2));
    if F_Value < Best_Value
        Success = 1;
    else
        Success = 0;
    end
end
if (Success == 1)
    Best_Beta(2) = Temp(2);
    Best_Value = F_Value;
    Cal_Pattern_Move = 1;
    Success = 0;
else
    Best_Beta(2)= Beta(2);
    Best_Value = I_Value;
    Success = 0;
    Cal_Pattern_Move = 0;
end

Pattern_Move = Cal_Pattern_Move;
Cal_I_Value = Best_Value;
Cal_Beta = Best_Beta;
end
```

The PatternMove function moves through the pattern direction found in the exploratory search. It outputs the point that optimizes the objective function after the pattern move.

```
function [Pcal_Beta, Pcal_I_Value] = PatternMove(Beta, Cal_Beta, Cal_I_Value, Input)
a=2; % An arbitrary acceleration factor
Pbest_Value = Cal_I_Value;
```

```
Pbest_Beta = Cal_Beta;

% Calculating the tentative point
Tentative_Beta = (a*Cal_Beta) - Beta;
F_Value = ObjFunction(Input, Tentative_Beta(1), Tentative_Beta(2));
if F_Value < Cal_I_Value
    Pbest_Beta = Tentative_Beta;
    Pbest_Value = F_Value;
else
    Pbest_Beta = Cal_Beta;
    Pbest_Value = Cal_I_Value;
end

Pcal_Beta = Pbest_Beta;
Pcal_I_Value = Pbest_Value;
end
```

The ObjFunction is the function that calculates the value of objective function. It receives the coefficients from the Patternsearch function and evaluates the objective function.

```
function FuncValue = ObjFunction (Input, Phi, Theta)
   a_noise = zeros(21, 1);
   x_data = [0 ; Input];
   Temp = 0;
   for i = 2:21 % estimating the noise terms based on the coefficients provided
       a_noise(i) = x_data(i)- (Phi*x_data(i-1))+ (Theta*a_noise(i-1));
   end
   for j= 2:21 % Calculating the objective function
       Temp = Temp + (a_noise(j)^2);
   end
   FuncValue = Temp;
end
```

The Residuals function receives the estimated coefficients from top-level function and returns the noise terms using the ARMA(1,1) formulation.

```
function Resid = Residuals(Input, Phi, Theta)
   a_noise = zeros(21, 1);
   Resid = zeros(20, 1);
   x_data = [0 ; Input];

   for i = 2:21 % Calculating residuals based on the model coefficients
       a_noise(i) = x_data(i)- (Phi*x_data(i-1))+ (Theta*a_noise(i-1));
   end
   Resid(1:20) = a_noise(2:21);
end
```

The ARMAtoMA function receives the estimated coefficients and calculates the coefficients of the π polynomial.

```
function pi_Coeffs = ARMAtoMA(ar,ma)
 p = length(ar);
 q = length(ma);
 pi_Coeffs = zeros(19,1);
 phi = ar;
 theta = ma;
 % Determining the pi coefficients based in the coefficients og phi and theta polynomials
 for i = 1:19
     if (i < q+1)
         tmp = theta(i);
     else
         tmp = 0.0;
     end
     for j = 1:(min(i+1,p+1)-1)
```

```
        if ((i-j-1)>=0)
            tmp = tmp + phi(j)*pi_Coeffs(i-j);
        else
            tmp = tmp + phi(j);
        end
    end
    pi_Coeffs(i) = tmp;
end
```

The Sorting function sorts the residuals using the bubble sort algorithm. It receives the residuals from top-level function and outputs the sorted version of residuals.

```
function sorted = Sorting(Input)
my_Vector = Input;
for i = 20-1:-1:1 % sorting residuals based in bubble sort algorithm
    for j = 1:i
        if my_Vector(j)> my_Vector(j+1)
            temp = my_Vector(j);
            my_Vector(j)= my_Vector(j+1);
            my_Vector(j+1)=temp;
        end
    end
end
sorted = my_Vector;
end
```

The LocateOutlier function receives the test statistics that are calculated in the top-level function and the critical value. It outputs the Outlier_index based on the comparisons of the test statistics and the critical value.

```
function indAO = LocateOutliers(tauAO, cval)
```

```
    temp = zeros(20,1);
    % Locating outliers based on the comparison of test statistics with critical value
    for i= 1:20
        if abs(tauAO(i))> cval
           temp(i) = 1;
        end
    end
    indAO = temp;
end
```