A REPRESENTATION METHOD FOR LARGE AND COMPLEX ENGINEERING DESIGN DATASETS WITH SEQUENTIAL OUTPUTS

A Thesis
Presented to
The Academic Faculty

by

Curtis K. Iwata

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
School of Aerospace Engineering

Georgia Institute of Technology
December 2013

A REPRESENTATION METHOD FOR LARGE AND COMPLEX ENGINEERING DESIGN DATASETS WITH SEQUENTIAL OUTPUTS

Approved by:

Dimitri Mavris, Advisor
School of Aerospace Engineering
Georgia Institute of Technology

Brian German
School of Aerospace Engineering
Georgia Institute of Technology

Daniel Schrage
School of Aerospace Engineering
Georgia Institute of Technology

Santiago Balestrini
Georgia Tech Research Institute

Vitali Volovoi
School of Aerospace Engineering
Georgia Institute of Technology

Date Approved: 21 August 2013

To my parents, my sister, and their cats.

ACKNOWLEDGEMENTS

First and foremost, I would like to thank my advisor Dr. Mavris for guiding me patiently through this long but fulfilling process of ideating, researching, getting discouraged, researching some more, hypothesizing, testing, getting discouraged again, and occasionally "Aha"-ing. I would also like to thank my committee members, Dr. Schrage, Dr. Volovoi, Dr. German, and Dr. Balestrini, for their wisdom, time, and encouragement throughout this journey.

Much of the inspiration and motivation came from working with Phil Fahringer, Heather Miller, and Anne Flannigan from the Lockheed Martin Corporation, and their knowledge and expertise on vehicle maintenance processes and logistics systems have been invaluable in learning about the field. I also want to thank my team members at ASDL, John Salmon and Elizabeth Saltmarsh, for being supportive and at times critical of my work. I enjoyed working with them, and I wish them good luck on the research.

I would not have been able to complete this document without the literary acumen of my peer reviewers, James Arruda, Dane Freeman, Joseph Iacobucci, Olivia Pinon, John Salmon, Elizabeth Saltmarsh, and Elizabeth Tang. Thanks guys! There were also numerous individuals who have supported me through this endeavor, and I thank them all for their contribution. A special thanks goes to Elizabeth Tang for helping me collect signatures and submit documents while I was finishing this thesis from

another state. My roommates and friends were an integral part of my graduate student life, and I thank them for keeping it fun and exciting.

And finally, I thank my family for supporting me all these years. I did it!

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

List of Abbreviations

ABM	Agent-Based Modeling
ALS	Autonomic Logistics System
Ao	Operational Availability
ARIMA	Autoregressive Integrated Moving Average
ASDL	Aerospace Systems Design Laboratory
BP	Breakpoints
CART	Classification and Regression Trees
DBSCAN	Density Based Clustering of Applications with Noise
DBSCANmod	Density Based Clustering of Applications with Noise modified
DES	Discrete Event Simulation
DFT	Discrete Fourier Transform
DoE	Design of Experiments
DWT	Discrete Wavelet Tranform
FAMOS	Fighter Aircraft Maintenance and Operations Simulation
FIFO	First In First Out
FPM	Failures Per Mission
GAM	Generalized Additive Models
GP	Gaussian Process
IQR	Interquartile Range
JSF	Joint Strike Fighter

KNN	K Nearest Neighbor
LCOM	Logistics Composite Model
LLE	Locally-Linear Embedding
LMC	Lockheed Martin Company
LogSAM	Logistics Simulation and Analysis Model
M&S	Modeling and Simulation
MARS	Multivariate Adaptive Regression Splines
MDS	Multidimensional Scaling
METRIC	Multi-Echelon Technique for Recoverable Item Control
MFE	Model Fit Error
MFHBF	Mean Flight Hours Between Failure
MMHF	Maintenance Man Hours per Flight Hour
MRE	Model Representation Error
MRFAMOS	Multirole Fighter Aircraft Maintenance and Operations Simulation
NN	Neural Network
O&S	Operations and Sustainment (or Support)
PA	Parallel Analysis
PAA	Piecewise Aggregate Approximation
PC	Principal Component
PCA	Principal Component Analysis
PHM	Prognostic and Health Management
PLA	Piecewise Linear Approximation

PLR	Piecewise Linear Regression
PLRDG	Piecewise Linear Regression Data Group
PR	Piecewise Regression
RQ	Research Question
SACT	Strategic Airlift Comparison Tools
SIM-FORCE	Scalable Integration Model for Objective Resource Capability Evaluations
SCOPE	Supply Chain Operational Performance Evaluator
SMARTS	Surrogate Modeling And Regression for Time Sequences
SVD	Singular Value Decomposition
SVM	Support Vector Machines
SVR	Support Vector Regression
TD3S	Top-Down Design Decision Support
TS	Time Sequence/Sequential
USAF	United States Air Force
VE-OPS	Virtual Environment for O&S Process Simulation
VM	Virtual Machine

SUMMARY

This thesis presents a methodology to create surrogate models of large time sequential (TS) datasets using piecewise linear regressions (PLRs) as it pertains to the engineering design of systems that perform operations and sustainment (O&S) of vehicles. The challenges tackled by the engineering design process continue to grow more complex, and one example of such challenges is the design and implementation of the O&S system for the F-35 Joint Strike Fighter. The vehicle needs thousands of parts to operate, and it will be supported by a global supply chain while operating under a cost-constrained budget. Establishing a sustainable O&S strategy is crucial for the success of the program, and this has to be designed and implemented before the fleet is fully deployed. Modeling and simulation (M&S) allows the testing of different approaches before the final implementation, and advancements in computing enable the exploration of many potential solutions. Surrogate modeling is used to summarize the data and to interpolate between existing data points, and advanced visualizations help decision makers understand the key tradeoffs.

O&S systems are complex, and M&S is a powerful approach to gain insight into them. An O&S system can be composed of various sub-systems that are organized in a complex network, and its dynamic behavior can be tracked as a time history. The trends indicate the performance and health of the system, and they are used to identify any anomalies. TS data of key metrics can be generated using simulation

models, and parameters of the simulation can be varied to study the interactions between inputs and outputs. To fully study a complex system, many combinations of parameter values need to be evaluated, but running the simulation for all potential solutions may take too long. Moreover, finding the important trends in the resulting dataset may prove to be difficult because of its size and high dimensionality.

Surrogate models can be used to estimate new data points and enable advanced visualization techniques, but care must be taken when fitting the models. For any given project, there is a limit to computational resources and time to perform simulation runs, and surrogate models can be used to fill the gaps by interpolating the data. They can also enhance the user experience of visual analytics tools such as interactive views and connected graphs by making them more portable and continuously parameterized, which is made possible by replacing the data with mathematical equations. One drawback is that regression methods have practical limits to fitting large TS datasets. The time to fit a surrogate model is directly related to the size of the dataset as well as the type of the method. The accuracy of the model is also influenced by the method, and creating a surrogate model for nonlinear data requires an equally sophisticated regression method, which takes longer to run. A tradeoff must be made between fit time and the accuracy of the model, but current methods do not provide a favorable compromise for design applications.

To create a surrogate model that can accommodate large TS datasets, the proposed methodology was developed using insights gained from analyzing the structure of O&S simulation models and its output data. The TS shapes that were generated

by the simulation models were organized into a reduced set, and a natural progression from one shape to the next was found. This observation uncovered an underlying mechanism of the simulation model, where the inputs were transformed into internal variables that ultimately determined the output shape. The Surrogate Model and Regression for Time Sequences (SMARTS) methodology was developed by imitating this two-step structure by using two regression models. One of the regressions captures the full range of TS shapes in the data, while the other regression creates a mapping between the input variables and its corresponding shape. An intermediate domain was introduced to serve as the bridge between the input and output domain, and principal component analysis (PCA) was determined to be the most appropriate method to calculate this intermediate domain.

TS shape function was created by extending PLR so that it can capture a wide range of TS shapes. The TS can be described as a shape composed of line segments. For example, a triangular wave can be represented as a sequence of rising and falling lines. In order to accommodate slight differences in the shapes, each segment of the PLR was parametrized, and separate PLRs were fit to distinctly different TS shapes and features. To find these groupings automatically, clustering methods and feature selection methods were explored. The most effective set of features was demonstrated to be the PCA of the TS dataset and the PCA of the breakpoint locations of each of the TS. This set with a model-based clustering method created groups of points that resulted in PLRs with good fit characteristics.

The SMARTS methodology was tested against linear regression and NN using data from the Multirole Fighter Aircraft Maintenance and Operations Simulation

(MRFAMOS) model. The simulation was based on a maintenance scenario for a fighter aircraft fleet, and it captures the sortie generation, maintenance, and spare parts supply chain processes. The SMARTS methodology provided a balanced solution for fitting TS data with a fast fit time and good fit characteristics. In terms of visual representation of the TS data, it was one of the best among the regression methods. The experiments demonstrated that the SMARTS methodology is conditionally better than other regression methods in representing TS data for engineering design applications.

CHAPTER I

INTRODUCTION

1.1 Engineering Design

Engineering design is a decision-making process to select the best solution to solve a problem or satisfy a need. It involves understanding the relationships between variables such as how increased performance affects the cost, and the design space is explored systematically to enumerate the potential solutions. Modeling and simulation (M&S) is used to provide insights into the complex interactions between systems, and surrogate models are often used to summarize the data, to capture its trends, and to analyze the impact of design decisions. Surrogate models are essential in the management and navigation of large design problems, and new techniques are needed as these engineering challenges continue to grow in dimension and complexity.

The process of problem solving as described by the Top-Down Design Decision Support (TD3S) process are composed of the following steps [104]:

1. Establish the Need
2. Define the Problem
3. Establish Value
4. Generate Feasible Alternatives
5. Evaluate Alternative

6. Make Decision

The first step in solving a problem is to identify that indeed a problem exists, whether it is improving an existing system or designing something new. The need should have sufficient consequence; otherwise, higher priority issues should be addressed first. Next is to formulate and structure the problem so that specific solutions can be designed to solve the challenge. A baseline and set of metrics also need to be defined so that the solution candidates can be evaluated. Then, the design space is populated with alternatives and compared. Finally, the decision is made, and the impact is tracked to check if it satisfies the original need. The TD3S process is generic and can be applied recursively to sub-problems that arise through the process.

Within this context, this thesis focuses on improving the surrogate modeling of multi-dimensional and sequential data, namely time sequential (TS) data. Surrogate modeling is involved in the last three steps of the TD3S process. One way to generate feasible alternatives is to use M&S, and because not all options can be evaluated due to resource constraints, some can be interpolated using regression. Evaluation of the alternatives and decision making are aided by using visualization, and surrogate models can be used to represent the dataset rather than querying a large table of values. The goal is to extend the capabilities of surrogate model to incorporate TS data into this process.

M&S is a low-cost alternative to physical experimentation in order to explore the design space and find feasible alternatives. As a problem becomes more complex, additional dimensions are typically required to capture the complexity, and this also

introduces the dilemma of handling large datasets. This is known as the "curse of dimensionality" [5]. Improvements in microprocessors and computing have made operating on datasets on the order of 10 to 100 MB a simple task today, but this was much slower only a few years ago. Large datasets with high number of dimensions also make it difficult to identify the important trends and key tradeoffs. The scope and complexity of engineering designs have also grown over the years [55], ensuring the longevity of this "curse".

Surrogate modeling has many applications to design and aids in coping with large datasets. It can be used to estimate design points that have not been evaluated with M&S. It can also be used to gain "increased insight in the problems" [29] by finding relationships that are obscured by the multiple dimensions and large volume of data. Surrogate models are embedded into decision making tools to approximate complex simulation models, and they are enablers for this style of knowledge discovery. The screenshot shown in Figure 1 is an example of a decision making tool that employs surrogate modeling to capture important interactions. This particular tool allows the user to compare the benefits of upgrading the aging fleet of aircraft using the data from a simulation model. Because the data for the entire design space is too large to be portable, a set of surrogate models is used as a substitute, and it enables interpolation to estimate values for design cases that have not been simulated. This interface is described in more detail in Section 1.4.

The ability to present TS data onto visual interfaces is important to decision making. TS data provides insight into the behavior of a system over time, and it is a key characteristic about dynamic systems. Time variant data can be reduced to

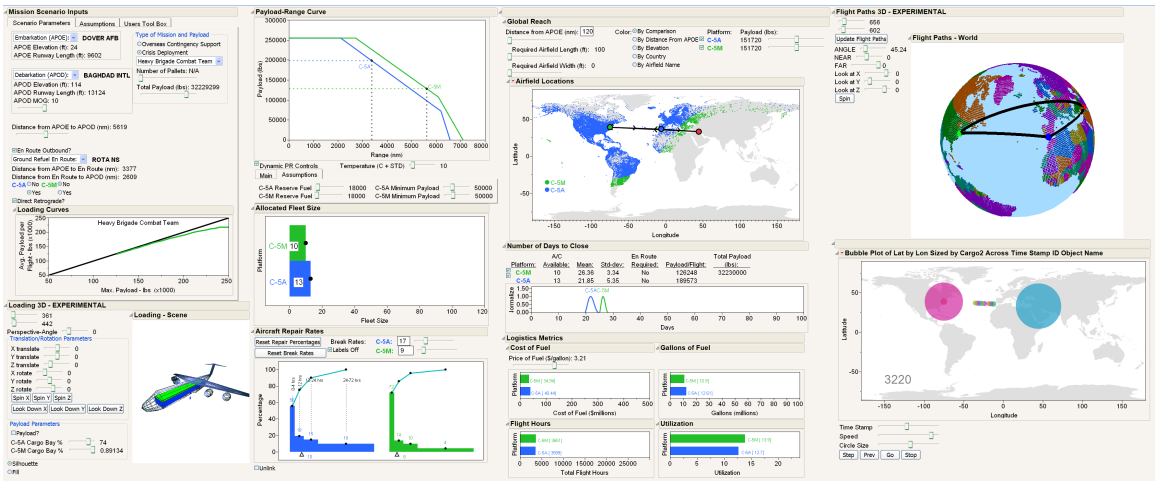


Figure 1: An example of a decision support tool [98]

statistics such as mean and standard deviation, but this can hide trends. For example, two systems can have the same mean, but one can be trending upward while the other downward. For certain systems such as for operations and sustainment, viewing the TS can offer insights into the behavior of the system as it adapts to changing conditions.

1.2 Operations and Sustainment

The motivation for a new regression method originates from studying operations and sustainment or support (O&S) systems. The O&S system encompasses the operations and maintenance of a vehicle as well as the support structure to sustain its activities. The cost for O&S represents a majority of the total ownership cost of a vehicle system. For military systems, the O&S cost is responsible for over 60% [121], as shown in Figure 2. One example is the F-35 Joint Strike Fighter program. Its acquisition is estimated to cost nearly $400 billion [115] while its O&S is projected to cost over $1 trillion across a period of 55 years [12, 105].

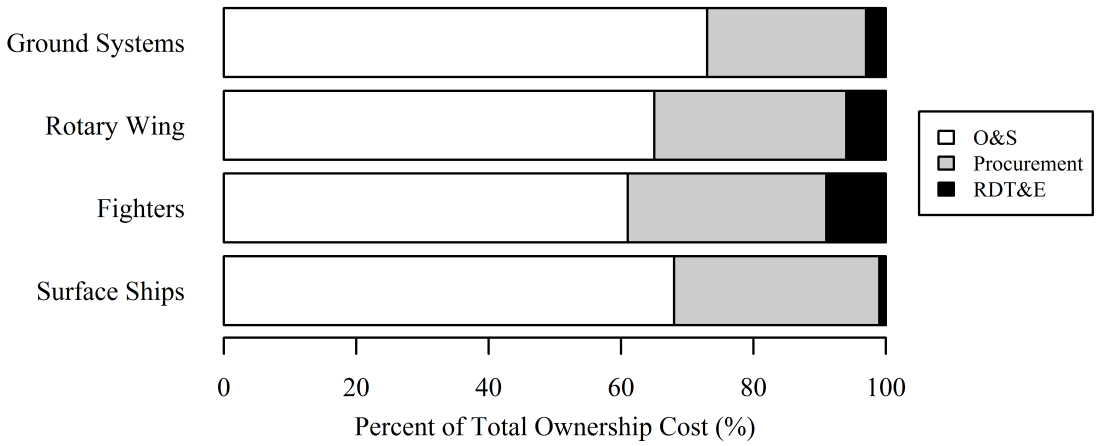


Figure 2: Operations and sustainment cost as percent of total ownership cost [121]

There is significant potential for cost reduction for operators and profits for service providers. In 2011, the United States Air Force (USAF) spent fifty billion dollars on operations and maintenance programs [122], which includes facility upkeep and upgrades as well as training and administrative costs. Looking at the direct operational and maintenance costs of its primary combat forces, the USAF spent over six billion dollars on its operations and another two billion on its depot maintenance [122], and this represents about 18% of the USAFs operations and maintenance budget. On the commercial aviation side, the market for maintenance, repairs, and overhauls of aircraft (not including aircraft weighing less than 19,000 lbs. spares, and inventory) was estimated to be $36.1 billion in 2003. In the current environment of shrinking budgets and commercial competition, effort is needed to help reduce costs, and by tackling a large piece like O&S, a small percentage reduction is still large in total dollars.

The challenges faced by an O&S system can vary from strategic to operational in scope. Strategic questions include policies on how to control inventory, investment

decisions on infrastructure, structuring supply chain assets for higher profitability, and impact analysis of new regulations and economic changes. Operational questions include how to use the available assets to meet changing demands and how efficiently the current logistics network performs its tasks. The O&S system has many components, and the complex connections between the subsystems make it difficult to assess the impact of any changes made to it.

Before beginning to find a way to answer the questions about the O&S system, a basic understanding of its processes and purpose is needed. The three main processes of O&S that is of interest for this research are sortie generation, maintenance, and spare parts repair and logistics. Sortie generation is the process in which aircraft are used to complete specific tasks, and the requirements for sortie generation typically drive the overall requirements for the rest of the O&S system because it is directly related to accomplishing the goals of the organization. Maintenance includes both preventive maintenance, such as inspection and scheduled servicing, as well as corrective or unscheduled maintenance to replace the broken parts. The logistics encompasses the management and transportation of spare parts and equipment that is needed for the maintenance of the vehicles. The relationship between the three processes are illustrated in Figure 3.

In the sortie generation process, aircraft that are capable of flying are matched with missions. Missions can have specific hardware requirements; for example, a reconnaissance mission may require a certain suite of sensors. These mission-specific hardware can be unavailable or broken, but that does not prevent an aircraft from flying other missions. Once the aircraft is assigned a mission, there can also be

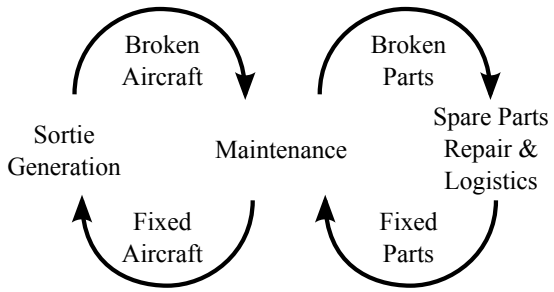


Figure 3: The relationship between sortie generation, maintenance, and spare parts repair and logistics

pre-flight routines such as a mission briefing and equipment loading.

The maintenance process begins after the vehicle returns from its mission [24]. It first undergoes routine inspections and post-mission activities. Then, it performs any corrective or unscheduled maintenance to fix parts that have broken during the mission, any predictive maintenance for vehicles with prognostic capabilities like prognostic health management (PHM), and any scheduled maintenance that is due. The scheduled maintenance is typically based on a measurable metric such as flight time, distance traveled or number of operational cycles. Once all the maintenance tasks for the vehicle are complete, turnaround activities are performed such as refueling before the vehicle is ready for more missions.

Alongside the maintenance process are the logistics to transport, to repair, and to restock broken and spare parts. When a vehicle returns with a broken part, the part is removed and replaced with a new one from inventory. Depending on the part, it can be fixed locally or sent to a repair depot. After it is repaired, it is transported back to the warehouse. Depending on the size and complexity of the distribution network, there can be a larger centralized warehouse that receives the fixed parts and replenishes the smaller local inventory.

The steps of the maintenance and logistics processes are simple: inspect and identify broken parts, replace aging and broken parts, perform turnaround tasks, send out broken parts, and stock new spare parts. The difficulty arises when considering the details. Vehicles require many unique parts to function, and these parts are delivered through various supply chain networks. There are different management strategies to govern the O&S system. For example, a broken part on a vehicle can be replaced as soon as it breaks or an alternative strategy is to defer the maintenance until a major failure causes the vehicle to be inoperable. Furthermore, these activities face human and geographical constraints. For a vehicle with many critical and highly technical parts such as an aircraft, the O&S system is vast, and making the right decisions that balance the "competing needs of improving performance, reducing costs, and enhancing safety" is difficult [87]. In these situations, M&S offers a cost effective method to understand the consequences of proposed changes.

1.3 Modeling and Simulation of O&S Systems

Most O&S systems are complex, and they involve multiple transport vehicles, different destinations, scheduling issues, and many constraints. It is also an area of great interest because a significant portion of the total cost of a product or service is in the O&S. Cost reduction can be achieved through different options such as through investing in new technologies, optimizing scheduling and asset mix, and changing management strategies to a lean approach. For example, Subramanian et al. developed a better fleet scheduling algorithm for Delta Airlines, and they reported a daily savings of $220,000 from their initial tests and projected to save $100 million annually

[114]. Subramanian's fleet scheduling technology is one of many solutions to improve the O&S system, and the ability to investigate and prioritize the options can help maximize the cost savings.

Modeling and simulation (M&S) is useful for these type of problems. M&S is an effective alternative to physical experimentation which is often difficult to conduct and sometimes impossible for large complex systems such as those for logistics and supply chain [14]. Other benefits include cost and time savings, ability to run replications, and safety from loss of profit and resources [84]. M&S enables the efficient exploration of different alternatives in a controlled environment.

There are different M&S tools available to analyze logistics problems [14]. Spreadsheet models are fast to run and simple to use, and they are suited for evaluating many scenarios. However, they are usually deterministic. There are analytical models that are fast to execute, and they are suited for scenario comparison and analysis of long-term behavior. Finally, simulation models offer the most flexibility and capability because they can model a problem to a high degree of fidelity. However, they usually take longer to run, and the results may be difficult to interpret and require statistical analysis.

According to Sadoun, there are mainly three types of simulations [96]. The Monte Carlo simulation uses random numbers for probabilistic problems that do not vary with time. The trace-driven simulation uses time-dependent data for input, such as weather prediction models. In a discrete event simulation (DES) model, a system is modeled as a series of events. The system state changes at a discrete time within the simulation as opposed to continuous systems where the states change continuously

over time.

The structure of DES is well-suited for a logistics problem, and consequently, it is the most common when such system is being modeled [46, 92, 116]. For example in an aircraft maintenance scenario, an aircraft's arrival at an airbase is an event. The amount of detail included in the simulation can vary, such as modeling the time it takes for each airplane to land and taxi or in finer detail by accounting for individual cargo inside the aircraft. This fidelity depends on each problem, the desired results from the model, and on the amount of resources available to pursue the answers.

TS data can be generated from DES by calculating and tracking metrics within the simulation. DES makes calculations at each event so that computation is not wasted on time steps with no events. This allows DES to simulate long time horizons efficiently. Whenever there is a change in one of the metrics, the values can be recorded with a time stamp so that this data can be plotted and analyzed after the simulation. Because the simulation is event driven, there is no guarantee that the metrics will vary in a smooth manner.

Two simulation models were developed to gain a better understanding of the interactions between these processes. The first model, the Fighter Aircraft Maintenance and Operations Simulation (FAMOS) model, was developed to simulate a fleet of aircraft while tracking a single generic part that can break and be replaced. The successor to the FAMOS model is the Multirole Fighter Aircraft Maintenance and Operations Simulation (MRFAMOS) model, and it is a higher fidelity model. It can simulate multiple missions types as well as multiple types of parts. Figure 4 shows a simplified depiction of these two simulation models. To support the modeling and

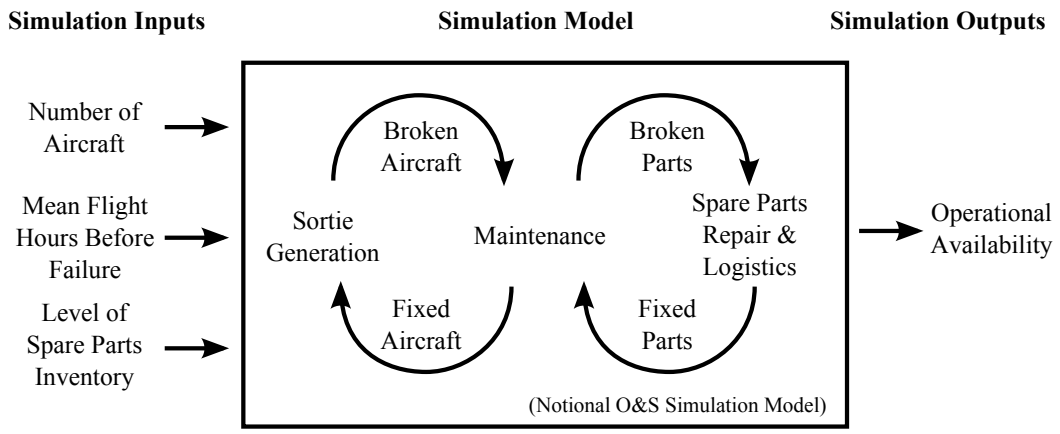


Figure 4: Simplified depiction of the simulation model

simulation activity for operations and sustainment, the Virtual Environment for O&S Process Simulation (VE-OPS) was also developed [50], and VE-OPS provides a set of tools built on on the Simulation in Python (SimPy) package [69] in the Python programming language [86]. More details on the O&S modeling and simulation, the FAMOS and MRFAMOS models, and the VE-OPS environment are discussed in Chapter 2.

Once the simulation model is created, many different design points can be run to explore the design space, but with a complex system such as with O&S, there can be many variables and interactions between them. Understanding the data can become a very challenging task, and visual analytics are proposed as a useful means for a user or decision maker to understand the relationships between the variables.

1.4 Visualization

Visual analytics are popular and powerful tools to help sift through large amounts of data, such as simulation data, and extract meaningful insights about the O&S system. Because simulation models can be executed repeatedly and relatively quickly, a lot of

data can be generated. Although, humans are good at detecting trends in the data, this task becomes cognitively difficult with large number of dimensions and with large amount of data; in other words, people get overwhelmed and overloaded with data. In order to assist in this task, visual tools such as line graphs and bar charts have been created, but these have their practical limits.

To help people cope with more dimensions and larger datasets, visual analytics use advanced visualization techniques and tools such as interactive charts and connected views. Figure 1 shows the Strategic Airlift Comparison Tool (SACT) [98], and it is an example of a visual analytics interface. It was designed to compare strategic airlift options such as C-5A and C-5M under different scenario conditions. The tool is interactive, and the results are updated automatically on the right as the inputs are specified on the left. Figure 5 shows a close-up view of the input panel. Many different combinations of input parameters can be specified using the interface so that different scenarios can be considered. Starting with the top-left section of the input panel, the overall scenario is specified by selecting two airports and the cargo. The first airport is where the cargo originates, and the second is the destination. Thousands of airports are stored in the tool to enable any number of airport combinations, and even an unspecified location can be used for the scenario by inputting the distance from the starting location. The flight route can also include an en route refueling location. The type of cargo is specified next, and depending on the type, the total payload amount and packing factor have already been preprogrammed into the interface. The bottom-left section shows a three-dimensional representation of the aircraft and cargo. The chart on the top-right displays the payload-range curve, and the slider

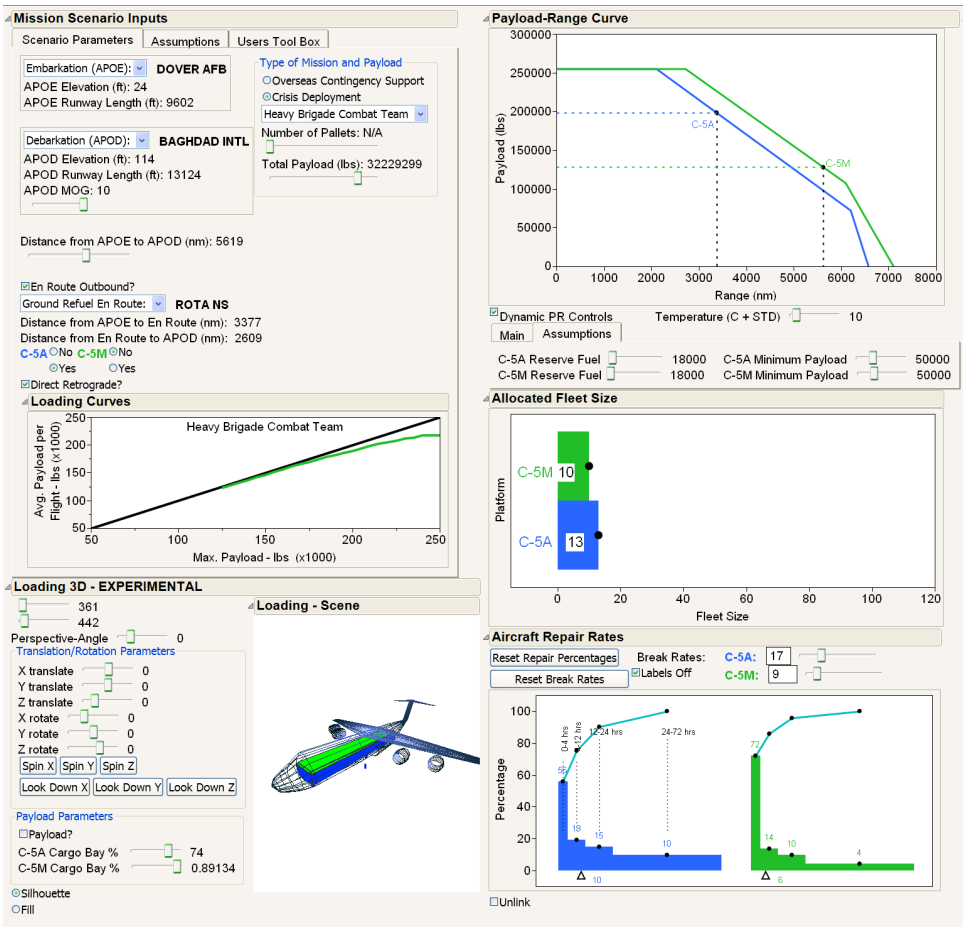


Figure 5: The inputs panel of the decision support tool shown in Figure 1 [98]

bars underneath it allows the user to change the temperature and other assumptions. The bar charts on the right and the bottom-right are the settings for fleet size and aircraft repair rates, respectively, and their values can be changed by dragging the handles like a slider bar.

Figure 6 shows the output panel of the SACT. The two views on the top show the flight routes projected onto two- and three-dimensional maps. The flat map is convenient to look at, but the routes are not accurate because they do not use the great circle routes. The three-dimensional globe provides a better perspective when evaluating the true flight paths. The charts on the bottom-left show the output

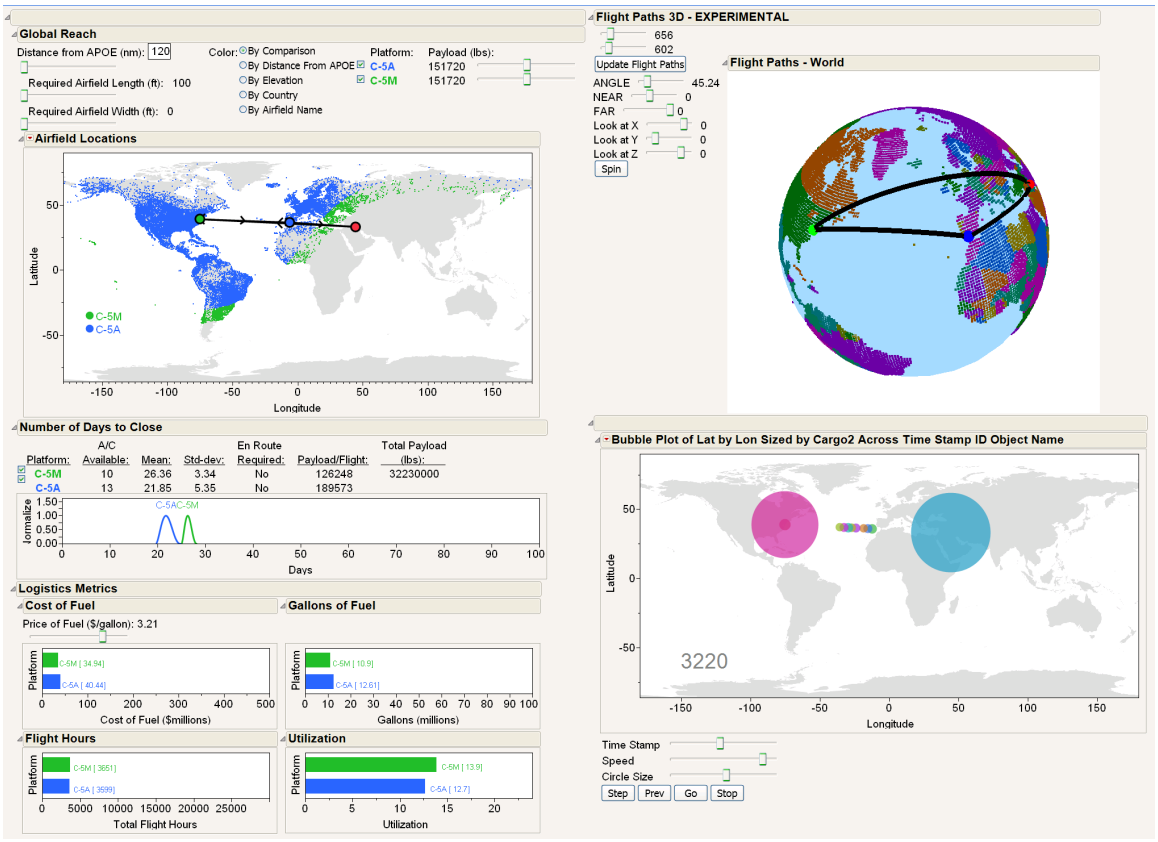


Figure 6: The outputs panel of the decision support tool shown in Figure 1 [98]

metrics including the time duration to complete the mission, total cost of fuel for the mission, and fleet utilization. The bottom-right chart is an animated bubble chart that illustrates how the aircraft in the simulation transport the cargo between the destinations.

The SACT is a feature-rich tool that allows the user to explore various combinations and understand the relationships between the inputs and outputs. Any changes made to the slider bars and various options on the input panel are instantaneously updated in the outputs, and it makes the data exploration interactive. As slider bars are moved, the user can test his or her assumptions of how it impacts the outputs by

visually inspecting the behavior of the outputs. Traditionally, any new input combination would require a new simulation run, and generating and preparing the results may take a some time. With this interface, the results and trends are at the user's fingertips because they have already been generated by running a large number of simulation runs and by fitting surrogate models to the data.

The use of these visualizations are important for decision making because it helps identify the important tradeoffs in the problem. Even with simulation models for large-scale systems such as O&S systems, there are other considerations that are difficult to capture, such as political objectives and hidden motives of each decision maker. Visual analytics are useful in conveying the technical tradeoffs, and they provide the technical context upon which the negotiations and decision making can take place.

1.5 Surrogate Modeling

Surrogate modeling is a common technique in aerospace design and also an enabler of visual analytics techniques. The goal is to estimate and represent the true behavior of a model or system using a limited amount of data collected from it [87]. A representative set of points of the design space is selected, generally with a design of experiments (DoE), and simulations are run on that sample. If the model is stochastic, several repetitions are run to capture the average behavior. The points are then used to fit regression equations. Surrogate models are suited to grasp a global view of the problem. Because they run quickly across many variables and over wide ranges, an analyst can perform sensitivity studies and experiment with different design variable

combinations. This can reveal hidden trends and provide insight into the problem [29]. It is also useful in optimization where it serves as the predictor of the optimum. Because surrogate modeling is the central topic to this thesis, a working definition is provided below. The topic is also further discussed in Chapter 4.

Definition 1: Surrogate Modeling
A set of techniques and methods employed for the creation of mathematical representations of simulation models based on structural combinations of input and output data obtained from these models

There have been numerous advances in surrogate modeling, but engineering design tasks continue to outgrow the capabilities of these methods. The problems are more complex, and the resulting design space is a combination of being large, nonlinear, noisy, discontinuous and multimodal. The established methods have difficulty generating an accurate representation of this kind of output with a limited set of resources of time, computational power and data storage (e.g. Figures 100 - 104 in Section 8.2.5).

There are many types of surrogate models, and each have their strengths and weaknesses. Some of the popular methods include polynomial response surfaces, neural networks (NN), Gaussian process (GP) models, radial basis functions, support vector machines, autoregressive models, and regression trees [75, 111, 106]. Among these, the first three are perhaps the most widely used in engineering design.

Polynomial response surface models are one of the fastest to fit, but they do not perform well when fitting complex surfaces associated with nonlinear and multimodal

spaces. NN can fit nonlinear spaces well in a relatively short amount of time, but it has a tendency to overfit or smooth out sharp corners. GP models fits nonlinear spaces very well, but it takes a long time to fit and are inconvenient to store and transfer.

The complexity or flexibility of a model is a measure of the range of shapes it can take. This is controlled by the number of parameters included in the equation and how they are combined [76]. For example, a cubic polynomial equation with four parameters, $f(\beta_0, \beta_1, \beta_2, \beta_3) = \beta_0 + \beta_1 x + \beta_2 x^2 + \beta_3 x^3$, has lower complexity than a sine function with four parameters, $f(\beta_0, \beta_1, \beta_2, \beta_3) = \beta_0 + \beta_1 sin(\beta_2 x + \beta_3)$, because the sine function can also handle cyclical data, but that is not to say that the sine function is always the better choice. It is typically better to choose the less complex model to avoid overfitting the data.

Despite the advances in surrogate modeling techniques, there are still datasets that it has trouble fitting properly. Large TS datasets that are generated in design space exploration activities are one of these because it is large and nonlinear.

1.6 Challenges of Working with Time Sequential Data

Time sequential (TS) data are a set of time-ordered values and are a practical output for the analysis and design of simulation models and dynamic systems. This type of data can be found in logistics and supply chain problems, and simulation models are created to analyze these systems. The time history provides context to the metrics of interest, and this additional information can aid decision making by providing insight into how the system evolves. For example, there is a significant operational difference

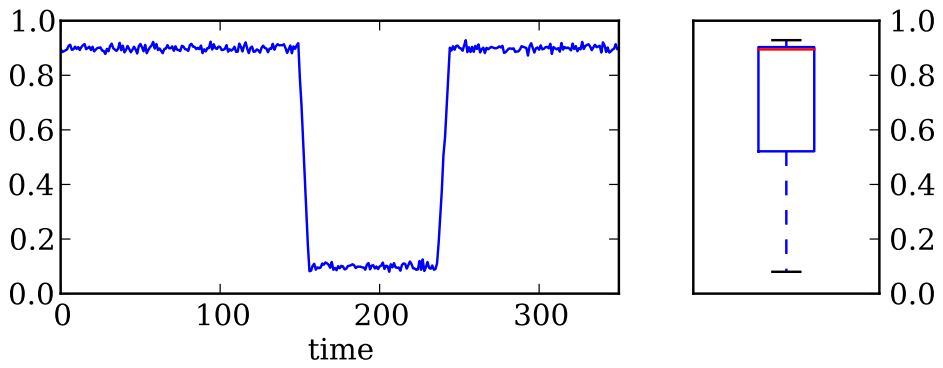


Figure 7: Comparison of TS data plotted against time and a box plot representation

between systems if the system utilization for one was running consistently at the target value of 90% compared to one that averages to 90% by cycling between 100% and 0%. This behavior is difficult to convey with an aggregated metric such as an average, but it is apparent when the TS is plotted as demonstrated in Figure 7.

The challenges of working with TS data can be organized into three main issues. The first is the large volume of data that is generated (Section 1.6.1). By recording the time history of a variable, the data grows with the length of the time sequence, which can get large for long simulations. The second is the nonlinearity of TS data (Section 1.6.2). The nonlinear shape is difficult to fit with surrogate models especially across a wide design space. The third is the difficulty in aggregating replications of TS data (Section 1.6.3). These three aspects are discussed in further detail in the subsections below.

More commonly, time sequential data are called time series data, but the term "time series" has subtle differences in connotation depending on the field of study. For example, in finance and economics, time series typically refers to stock prices and various indexes, and the goal of fitting the data is to predict the direction of

the price. In control theory, the system response is tracked over time to control its future behavior, and properties like zero mean and stationarity is of great interest. The goal of this thesis is to preserve the general shape of the time sequence at each design point. Furthermore, mathematically speaking, series is the sum of a sequence of terms [77]. For the purposes of this thesis, this type of data will be referred to as time sequential data to avoid confusion of mixing the similar literature and different motivations. The only exception is when referring to a specific field with "time series" in the name such as time series data mining in respect of the body of literature.

Definition 2: Time Sequential Data

Any data that is ordered by time, where given any output variable y_i, there is an associated time value t_i

To demonstrate the problems of working with TS data, a sample dataset from an aircraft sustainment supply chain model is presented below.

Example 1.1: TS data of aircraft operational availability

Figure 8 plots the output of operational availability (A_O) of the training fleet of aircraft, and this metric is measured daily. The data is generated using the Fighter Aircraft Maintenance Operations Simulation (FAMOS) model, which is covered in detail in Section 2.3. A_O is defined as the number of aircraft available to fly missions divided by the total number of aircraft in the fleet, and it ranges from 0% to 100%. (More detail on A_O is provided in Section 2.1.1). The simulation has a warm-up period which is not shown. There are two operations that require

a subset of the training fleet, and each lasts 180 days. The first operation begins on Day 0, and the second operation begins on Day 15. The vertical lines in the graph represent the beginning of the first operation and the end of the last operation. Each graph shows five repetitions with the same input values, and between each graph, all variables are held constant except the ratio of the total inventory of spare parts to the baseline value, which is varied from 90% to 110%.

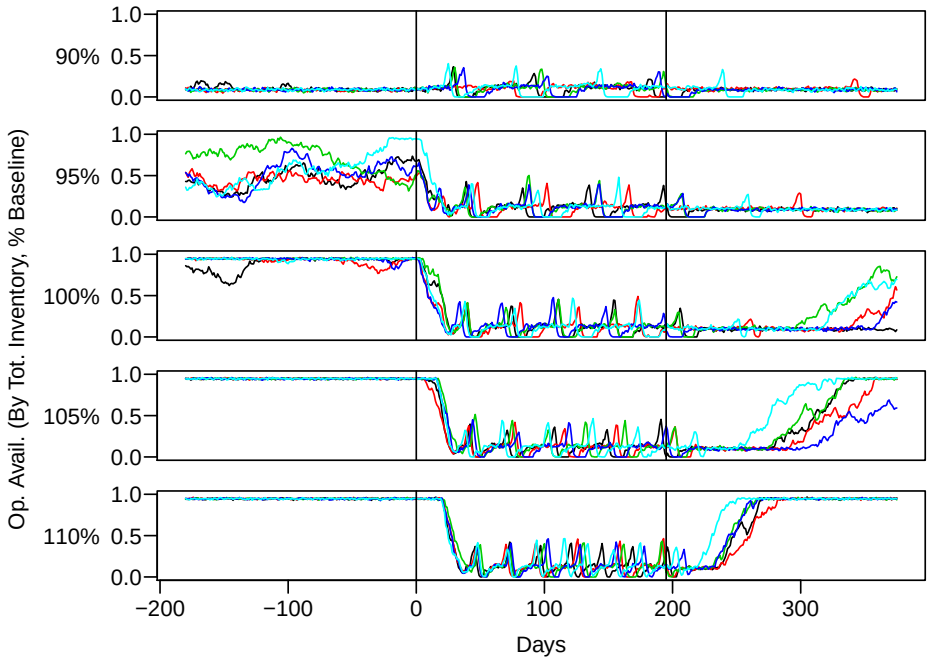


Figure 8: Simulation data from aircraft sustainment model. Total inventory is varied from 90% to 110% of baseline in 5% increments. Each graph shows five replications. Vertical lines indicate the beginning and end of operations.

For this example, a total of 25 runs have been simulated, and it produced approximately 0.1 MB of data. Considering 10 variables at 2 settings with 30 repetitions resulted in approximately 120 MB of data for a single metric. This

value can quickly grow large with more variables, more settings, and more output metrics. This would be inconvenient to manipulate and analyze, and the surrogate modeling process will take a long time.

The shape of the curves vary from a flat line at 90% inventory to a bathtub shape at 110%, and although not shown, the line becomes a flat line at about 95% A_O at higher inventory values. The behavior is nonlinear, and there is large uncertainty in some parts of the design space such as seen at the beginning of 95% inventory and at end of 100% and 105% inventory.

Finally, the stochastic nature of the simulation causes a shifting of TS features. The features in the example include the steady state region before the beginning of the first operation, the subsequent fall in A_O, the small peaks, and the eventual recovery after the end of the last operation. Within each graph, each line exhibits a common set of features with slight variations. For example, the recovery in the fifth graph with 110% total inventory occurs at different timings and with varying slopes, which shows that this feature can shift and warp horizontally.

The three main issues that were raised previously and demonstrated in the example will now be explored in further depth.

1.6.1 Large Dataset

The main problem with a large TS dataset is that it is inconvenient to work with; for example, analysis of the data will take longer because there are more data points to sort and evaluate. Regressions take longer to fit because there are more points, and the fit time is not linear to the number of points. For example, Gaussian process

models scale on the order of cubic time ($O(n^3)$) or number of points to the third power [40], and a regression of data that takes 10 seconds to fit will take 80 seconds if the data size is doubled. When sufficient computational resources are not available, it simply means more time is required to accomplish the tasks, but time may also be limited.

1.6.2 Nonlinear TS data

Fitting complex TS data can be done, but the task becomes difficult when it also has to fit across a large design space. Most surrogate models have to balance between being specific enough to capture certain features but general enough to explain global trends. A method may be capable of capturing the shape of a single run well, but when it is required to account for multiple dimensions of the design space at the same time, it may have to compromise its fits of each TS run to capture the trends in the design space.

The balance between fitting the local and global features and trends corresponds to the balance that exists between overfitting and smoothing. In an extreme case of overfitting, the regression will fit each point by fitting a wavy function that passes through all the points, but the result may not be representative of the underlying function. This is not useful for predicting new points or interpolating between the points. On the other hand, surrogate modeling can be used to smoothen the local variability to approximate the underlying function, and its extreme case is the average of the data, which is also not useful as a representation. Surrogate modeling algorithms balance between these two extremes. If the underlying function is not

known, the "correct" fit comes down to a subjective decision.

Smooth functions have a difficult time fitting nonlinear outputs. For example, a simple function $X^{-X^2-Y^2}$ is plotted in Figure 9. The values have been defaulted to 0 under a threshold of 0.001. This is done to mimic the behavior in discrete event simulations where events occur at some time and not at infinite time. When a neural net is fitted to this peak and valley function, it fits the peak and valley of this space well, but it overfits the flat regions, causing it to be wavy.

$$Z = \begin{cases} X^{-X^2-Y^2} & \text{if} X^{-X^2-Y^2} > 0.001 \\ 0 & \text{if} X^{-X^2-Y^2} < 0.001 \end{cases} \tag{1}$$

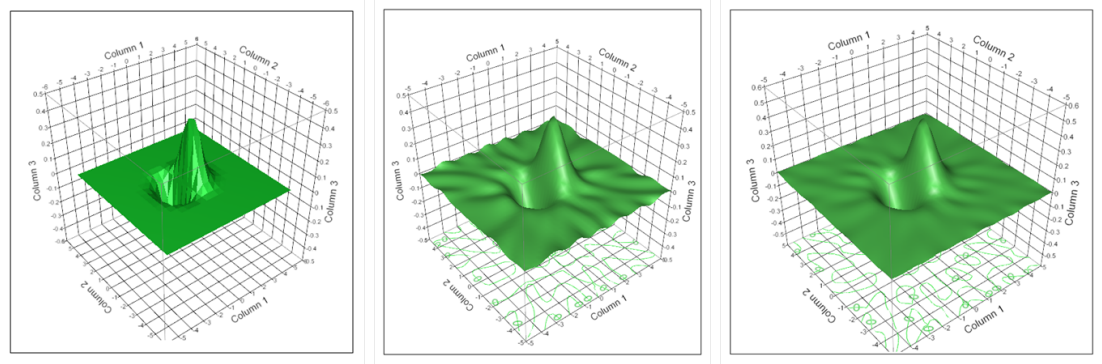


Figure 9: Peak Valley Function. Left: Points generated from -5 to 5 with 0.5 spacing. Center: NN, 1 hidden layer, 20 nodes, Overfit Penalty = 0.001, R^2=0.98142, Right: NN, 1 hidden layer, 30 nodes, Overfit Penalty = 0.001, R^2=0.99767, No validation points

A real world extension to this kind of output data could be found when studying the inventory policy of an aircraft repair facility. For example, a notional part on an aircraft takes an average of six months to repair while it takes an average of three months to break. Because the part breaks faster than it can be fixed, extra parts need to be stocked so that it can be replaced on the aircraft when it comes in for repairs.

The rate at which the parts are breaking needs to be balanced with the rate at which the parts are being fixed. This is accomplished by adjusting the level of inventory so there are enough parts in the repair queue. If there are not enough parts, there will be a backlog of aircraft, and if there is an excess, then it is extra inventory, which is not cost effective.

Finding the optimum level of inventory in this simple scenario is an easy task because it reaches a steady state after a while. In real world applications, there are occasions when aircraft are flown more often such as during a holiday season, and this will cause the parts to fail correspondingly. If the inventory does not account for this new rate of failure, it will soon run out of spare parts.

If the backlog over time is tracked for this example, there will be several distinct features. For those situations with an excess of inventory, the backlog will always be low. For those with just enough inventory for normal operations, the backlog will increase when the peak season arrives, and it will return to normal after the peak season ends. For those situations with insufficient inventory, the backlog will increase until all the aircraft are in backlog. As in the peak valley example, there are flat regions and "mountainous" regions with distinct boundaries. Smooth equations like NN tend to overfit flat regions and output negative backlog, which is impossible.

1.6.3 Aggregating Repetitions of TS data

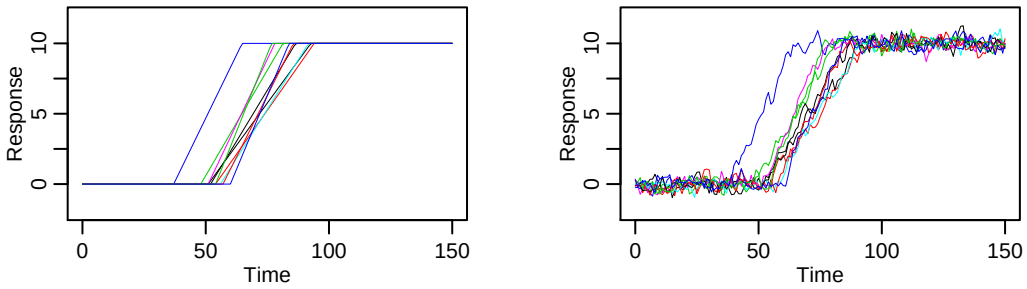
The overall behavior of a stochastic simulation is usually similar for the same inputs. There are certain characteristics and features that are unique to each simulation model, and those can be used to compare different TS data. Distinguishing features

include flat segments, general upwards or downwards trends, peaks and valleys, and noise. When two TS segments have the same initial inputs, the two should look similar with a few minor differences, unless there is a high level of uncertainty or stochasticity in the simulation model.

The difference between two time sequences from a stochastic simulation mainly stems from the randomness programmed into the model. Randomness manifests itself in a few different ways. Uncertainty can be incorporated by assigning random variables to model parameters. The variables can have shape functions such as uniform, triangular, and normal distributions, and they get aggregated through the simulation to provide the distribution of an output parameter. For example, the level of inventory depends on how the simulation has been consuming the inventory up to that point. In such case, the consumption rate can be a random variable. The level of inventory at any given time will have a standard deviation which can be interpreted as the uncertainty or variability.

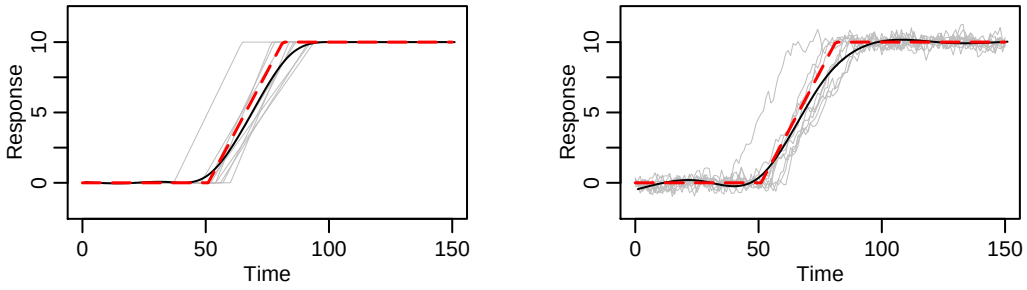
TS data can have a special error called autocorrelation error because the current value typically depends on the previous value, and so, it is correlated with itself. When creating a surrogate model for this data, it violates the assumption that the errors are independent. If care is not taken to account for autocorrelation, the calculated error when fitting the model will be much larger than it is supposed to be.

One of the problems with fitting multiple TS data is that the features may be shifted and warped in each replication. For example, Figure 10a is a notional TS dataset of ten runs. There is an underlying process that moves the response from 0 to 10 at a random time and rate. The randomness causes the horizontal shift and



(a) Overlay of 10 TS

(b) Overlay of 10 TS with noise



(c) Intuition vs. Spline Regression

(d) Intuition vs. Spline Regression

Figure 10: Overlays of time sequential data with shifting and variable slope coefficients. Red line is a piecewise continuous linear line based on intuition. The black line is a 9th order spline.

warping.

Generating a representative or aggregate curve of the lines in Figure 10a is a tricky task. The data provides insight into the underlying process, and the curve needs to capture this process. The TS replications in the figure have sharp corners so the representative line should also have sharp corners like the red line in Figure 10c. If a smooth regression function is used to fit the data, it would produce something like the black line, which has rounded corners, and this artificial smoothness could be misleading. Figures 10b and 10d have the same underlying TS as the plots to its left, but they have been corrupted with autoregressive noise to represent a more realistic scenario. These figures illustrate how noise in the data makes the task of determining the representative curve more difficult because the corners are obscured.

Even if the features and the differences are identified, quantifying the differences is a difficult task. In design space exploration, multiple repetitions are run to quantify the uncertainty due to the stochasticity of the model. In a non-TS dataset, the output values may be different, but they are not dependent on time so there is no problem of shifting and warping. A surrogate model is fit using some form of best fit criteria such as an ordinary least squares approach while and another surrogate model could be fit on the data's standard deviation. Collapsing this dataset into a set of statistics is not an issue. The same cannot be said for TS data because the behavior and the shape of the sequence is equally important as the data. Hence, to capture the behavior, the model parameters such as the intercept and slope need distributions. Otherwise, if the standard procedure is taken to regress a model to time series data, there will be large deviations in the region where the outputs have shifted. In regressing this type of data, it is important to realize that the problem is not only that the values become highly uncertain during the transition but also that other parameters, such as the timing of the transition, are uncertain.

1.7 Previous Research on TS data and Engineering Design

Most of the literature that is related to TS data regression focuses on trend extrapolation, such as for stock market forecasting. Geospatial regression is also relevant because the relative location of the data with respect to other points is important, but the Kriging method that is popular for this application is known have long fit times.

There is little research being done with respect to TS regression to aid in engineering design and simulation. Moon modeled advanced ship concepts that have a self-reconfigurable fluid system [72]; these ships are designed to reroute its fluid system in the event that certain areas of the ship are damaged. In order to simulate the ship-level reaction to damages, the component-level systems need to be simulated as well, and this can take a long time due to the complexity. In order to reduce the simulation time, Moon created surrogate models for the components, which were then used in place of the actual component simulation blocks. Because the responses are nonlinear and transient, he modified recurrent neural networks to model the components. The fit times of each of these surrogate models varied from several minutes to several hours depending on the complexity of the data. Moreover, some of the fits did not accurately match the original data, but it was sufficient for the target investigation.

Phan incorporated time-transient regimes into aircraft design space exploration, such as fluctuations of voltages due to high electric loads [83]. The transient outputs from various simulation runs were plotted onto a single plot, and the visualization allowed the user to filter out the responses that violated a specific constraint in an interactive manner. To create the surrogate model, Phan used wavelet neural networks or wavenets. The fit times of these wavenets took 2 hours per response for a set of 250 cases of runs.

In both Moon's and Phan's research, a variant of the NN was used because of its ability to fit highly nonlinear data. However, the fit times of these advanced NN are long, and its use is limited to low number of dimensions and small datasets.

1.8 Research Goal and Method Overview

The main research objective that drives the investigation is posed as follows:

Research Objective
Create a rapid and efficient regression method for the visualization and analysis of sequential data to support engineering design decision making

In accomplishing this objective, there are also several requirements. The resulting surrogate model needs to have good model representation, both statistically and visually. The model execution needs to be quick because it will be used to interpolate points in visual interfaces. The model should also have fast fit times because no matter how sophisticated the method is, it still requires iteration to create a good model. Finally, it needs to handle large and nonlinear datasets.

A preview of the proposed methodology is presented here to serve as a guide through the literature review in the succeeding chapters. To overcome the challenges listed above, the proposed methodology splits the regression into two steps, which have different roles. The first step is responsible for capturing the variability caused by changing the input values. The second step is responsible for recreating the full range of possible TS shapes.

This is accomplished by introducing a third domain (Z) to connect the input (X) to the output (Y). The Z domain is calculated from the outputs using principal component analysis (PCA). PCA is a dimensionality reduction technique and will be covered more in depth in Chapter 3.

PCA reduces the TS data from n dimensions to one dimension, where n is the length of the TS. Then a regression is fit from the X domain to the Z domain using NN. A separate regression is fit from the Z domain to the Y using piecewise linear regression (PLR). The $Z \rightarrow Y$ regression involves several steps such as data reduction and clustering, which will be discussed in Chapters 4 and 5.

The resulting method is called the **S**urrogate **M**odeling **A**nd **R**egression of **T**ime **S**equences (SMARTS), and the main steps are summarized below.

1. Create an intermediate set of axes (Z domain) from the output data (Y domain)
2. Fit a regression model from the input space (X domain) to the Z domain
3. Fit a piecewise linear regression model from the Z domain to the output TS space (Y domain)

A detailed development of the SMARTS methodology is provided in Section 6.3. The methodology combines methods from various fields, and Figure 11 organizes the content and steps with respect to the chapter numbers in which they are described.

1.9 Summary

This introductory chapter presented the goal of this thesis, which is to create a surrogate modeling methodology for large datasets with sequential data. The motivation is derived from the need to approximate a simulation model for O&S design problems. The surrogate model enables visually-driven data exploration and supports decision making, which will hopefully help reduce O&S costs. The main challenges in creating such a mathematical approximation lies in the size of the design space, nonlinearity

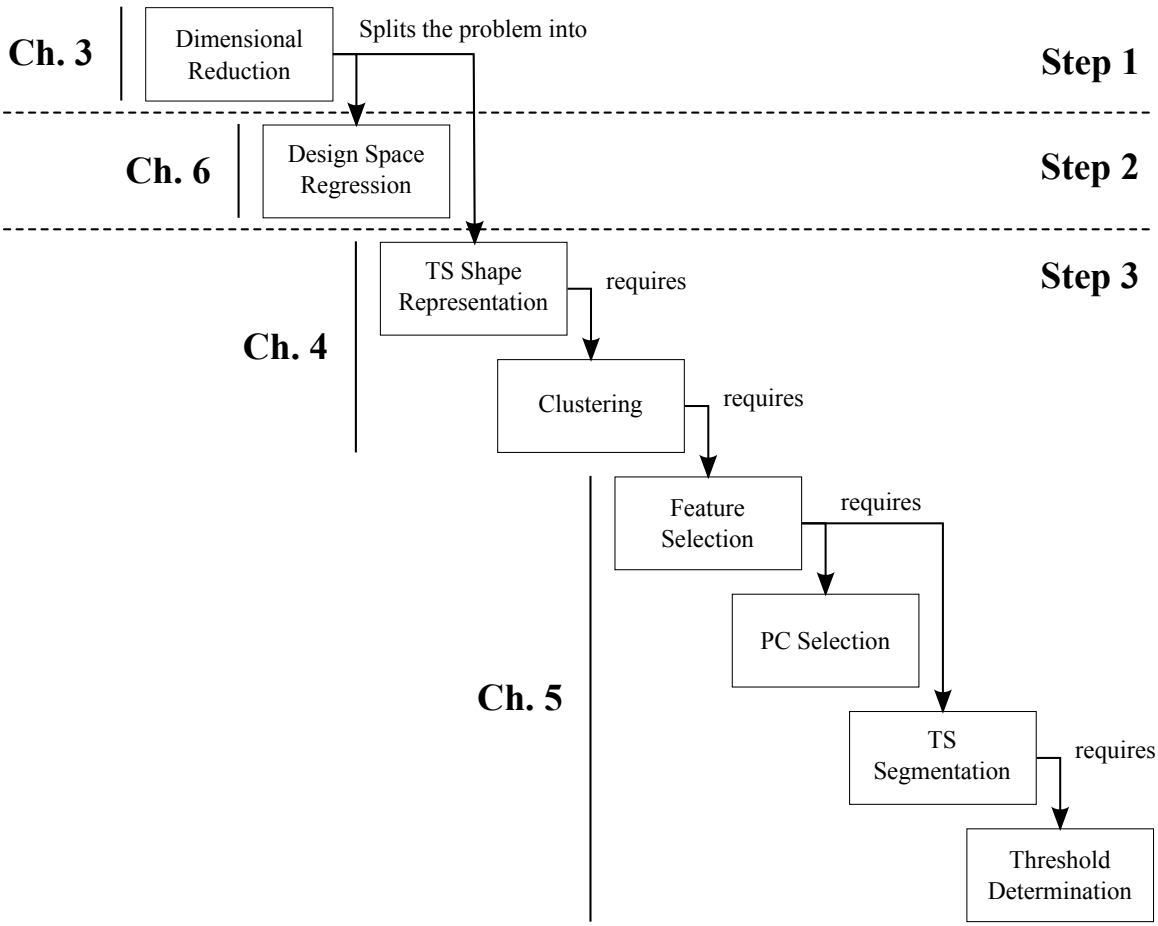


Figure 11: Content organized by chapters and SMARTS methodology steps

of the TS data, and the stochasticity of the model. A new method called SMARTS is proposed to answer these challenges.

The succeeding chapters will describe and explore the concepts that compose the proposed method. It will also address the problems highlighted in this chapter. Chapter 2 covers the M&S background for O&S applications and the VE-OPS tool set, which is used to create O&S simulation models. It also examines a sample dataset which provides insights and important observations to guide the research. Chapter 3 focuses on dimensionality reduction techniques and discusses the benefits of using an intermediate dimension for regression of high-dimensional outputs. Chapter 4 reviews surrogate modeling methods and lead to the decision to use piecewise linear regressions. Chapter 5 discusses TS segmentation and extracting features from the data for clustering applications.

Chapter 6 covers the proposed method in depth and introduces a morphological matrix containing alternative approaches and methods to each of the steps listed above. Chapter 7 provides the claims, hypothesis, and experiments. Chapter 8 discusses the details of each experiments, presents the results, and discusses its implications. Finally, Chapter 9 will wrap up the research with conclusions, list of contributions, and potential extensions.

CHAPTER II

OPERATIONS AND SUSTAINMENT SIMULATION MODELING

In the previous chapter, a brief background to modeling and simulation (M&S) and its utility to analysis of operations and sustainment (O&S) systems was introduced. In this chapter, a more thorough background on O&S simulation modeling is provided (Section 2.1). Then, the **V**irtual **E**nvironment for **O**&S **P**rocess **S**imulation (VE-OPS) is quickly reviewed to give insight into how O&S simulation models are structured (Section 2.2).

2.1 Maintenance and Logistics Simulation Background

Over the course of the past five decades, different aspects of the maintenance and logistics process have been studied. Simulation models are being used to study requirements for inventory levels, logistics, workforce size, and workforce skill sets. With the development of the Joint Strike Fighter (JSF), numerous studies and theses have been produced to understand the impact of autonomic logistics systems (ALS) and prognostic and health management (PHM) systems on the maintenance and logistics process. There are also studies that provide insight into better planning and scheduling strategies.

One of the main applications for simulation modeling is to determine the inventory and logistics requirements, and there are many tools to support this activity.

The Logistics Composite Model (LCOM) is a Monte Carlo simulation model that was originally developed in the 1960s by the RAND Corporation and the Air Force Logistics Command to advise base-level logistics requirements [8]. The Multi-Echelon Technique for Recoverable Item Control (METRIC) and its variants, MOD-METRIC and VARI-METRIC, are analytical models first developed in the late 1960s to evaluate the operational availability (A_O) of an aircraft based on the multilevel supply stock levels [109, 73, 110, 108]. The military uses a variety of tools for their logistics and supply chain analysis and requirements building including Logistics Simulation and Analysis Model (LogSAM), Supply Chain Operational Performance Evaluator (SCOPE), Scalable Integration Model for Objective Resource Capability Evaluations (SIM-FORCE), and SIMLOX [38, 94].

Another important use of maintenance and logistics modeling is to analyze the size and composition of the workforce. LCOM has been continually improved over the years [19], and one of its main applications is to determine the manpower requirements for maintenance [8]. Gallasch et al. study personnel requirements using a Colored Petri Net model and explore different ways of representing personnel within the simulation [35]. Bazargan and Jiang develop a detailed Arena model of AirTran's maintenance operation to improve the company's manpower planning capability [13]. MacKenzie uses an agent-based model to study the impact of different skill levels of the maintenance workers on the sortie generation rate of a single fighter aircraft unit [66, 67].

Some of the recent research efforts by the military focus on the ALS and the PHM system that are being implemented on the Joint Strike Fighter (JSF) program. One

of the early simulation projects is documented in Rebulanan's thesis in 2000 where he creates a simulation model called ALSim [91] in JAVA using the Silk simulation package [44]. This initial model is extended to study the benefits of PHM system for the JSF [46, 68]. Around the same time, Schaefer and Haas investigate the impact of health and usage monitoring system (HUMS) on helicopter operations and maintenance [103]. Yager models the ALS using a queuing network model to study its impact on sortie generation rates [124]. Tsoutis studies the impact of ALS to failures per mission (FPM) and maintenance man hours per flight hour (MMHF) by applying ALS to the repair process of the Navy's F/A-18E/F jet engines using a DES package Simkit [120]. One of the major concerns for a PHM system is the impact of false alarms to its performance, and several studies have highlighted the importance of the detection accuracy of PHM systems [103, 25, 71].

There have been other research efforts to investigate the development, deployment and execution strategies of maintenance and logistics systems. Cannibalization and different cannibalization policies for aircraft fleets have been studied using analytical models, inventory models and simulation models [28, 27, 97]. Rodrigues et al. explore inventory consolidation and part transfer time of critical components with long repair cycle times as a way to maintain high readiness levels of A-4 fleets for Argentinian and Brazilian military [93]. Sarma and Ramchand study the interaction between air fleet and maintenance system from a control theory perspective to provide options for fleet performance improvement [100, 99]. Hill et al. examine different deployment strategies of equipment needed for specific repairs on vehicles [46]. Iakovidis evaluates the impact of different maintenance scheduling philosophies on the operational

performance of an F-16 fighter fleet in the Arena DES package [49]. Pendley et al. demonstrate through a simulation that different maintenance priorities, such as first in first out (FIFO), can impact the metrics measured at higher levels of an organization and conclude that lower level metrics need to align with the top level goals [80].

Simulation models have been created and used to understand the impact of various O&S factors to the outcomes of interests. The number of studies and their variety have increased over time, and this can be attributed to a number of factors including the progress of computation capabilities and ubiquity, the improvements to M&S tools, and the acceptance of M&S as a tool to study these problems. Furthermore, as systems become more complex and as constraints such as costs get tighter, the need for more sophisticated analysis techniques and accurate results will drive new improvements to M&S tools and methods.

2.1.1 Metrics for O&S Simulation

Metrics are measurements that provide insight into the health of a system [88]. They can be quantitative or qualitative, and they are recorded over time so that a baseline value can be established. The baseline serves as a point of comparison for future measurements of performance, and the time history provides a way to track improvement. There are a variety of metrics in O&S, and some of the high level metrics for O&S include sortie generation rate (SGR), operational availability (A_O), mission capable (MC) and not mission capable (NMC).

Sortie Generation Rate

SGR is the sustainable number of missions an aircraft unit can generate over a specific time period [21]. It is an overall metric of an aircraft unit's performance, and it encompasses maintenance and supply chain aspects as well as the reliability of the aircraft and parts. It is calculated as follows:

$$SGR = \frac{\text{aircraft launches}}{\text{time}}$$

Operational Availability

A_O is an overall metric that measures the percentage of the time a system is ready for use or action [13]. It is expressed as a value between 0 and 1 or as a percentage, and the equation is written as follows:

$$A_O = \frac{\text{uptime}}{\text{uptime} + \text{downtime}}$$

where uptime is the time the system is available for use and downtime is when the system is not available due to maintenance or other reasons. The instantaneous fleet-wide A_O can be calculated using MC and NMC:

$$A_O = \frac{\text{MC}}{\text{MC} + \text{NMC}}$$

Mission Capable

Mission capable is the number of aircraft that are ready to fly missions [88]. MC can also be expressed in hours or rates, at which point MC becomes A_O. Depending on the aircraft, there can also be fully mission capable (FMC), where the vehicle is fully repaired, or partially mission capable (PMC), where some parts are broken or missing but the aircraft can still fly some missions.

Not Mission Capable

Not mission capable is the complement to MC, and it can be further classified into NMC due to supply (NMCS) and due to maintenance (NMCM) [88]. NMCS is caused when a part is not available, and it is a measure of the supply chain effectiveness. NMCM is caused by a lack of personnel to perform the maintenance, and it can be an indicator of lack of maintenance staff, poor scheduling, and insufficient equipment and facilities.

2.2 VE-OPS

A set of modeling tools tailored to generate O&S simulation models is being developed at Gerogia Tech's Aerospace Systems Design Laboratory (ASDL) as a part of an ongoing collaboration with Lockheed Martin Company (LMC). The toolset is called the **V**irtual **E**nvironment for **O**&S **P**rocess **S**imulation (VE-OPS), and it is a framework with associated programming functions and structured variables to simplify the creation of simulation models of medium complexity [50]. The goal of VE-OPS is to reduce the need to *reinvent the wheel* by standardizing model components that are common across different O&S simulation projects.

There are several factors that influenced the design and implementation of VE-OPS. Some of the criteria that were sought at the beginning of the project include the following:

- Fast execution time for Monte Carlo analysis
- Ease of sharing and executing the model on different computers

- Ease of modifying the behavior and model structure
- Ability to handle large numbers of simulation objects
- Ability to create a "server" version of the model which would execute simulations on-demand when given the appropriate inputs

Fast execution time is desirable because thousands and potentially millions of simulation runs will be performed to explore large design spaces. Simulation models may be probabilistic so multiple runs will be needed to capture the uncertainty. The need for easy sharing was inspired by the overhead involved with working with sharing tools developed on engineering software platforms. Commercial programs usually require licenses, and some can be expensive. A similar problem can occur when executing the program on multiple machines because each machine would require its own license. Once the model is developed, it would save time and effort if it were easy to modify for different types of experiments and adaptable to other projects. Some simulation programs have difficulty handling large numbers of objects, and this is a problem when studying large O&S systems. Lastly, being able to connect the model to a visual interface allows new simulation runs to be performed while exploring parts of the design space that have not been sampled before, and this is why a "server" version that can form a live connection with other tools is part of the VE-OPS development criteria.

2.2.1 Discrete Event Simulation and Software Platform Selection

A brief introduction to discrete event simulation (DES) was provided in Chapter 1. As mentioned previously, DES is well-suited to model systems that are process-driven, meaning there are well-defined steps in accomplishing a task. This works well for O&S systems which usually have set processes. Many of the simulation models mentioned in Section 2.1 are DES models. One interesting exception is MacKenzie's agent-based model (ABM) that investigates the impact of human resources to the mission readiness [66, 67], and although this could have been accomplished in DES, the unique need to define the behaviors of individual workers is better captured in an ABM.

There are many DES software packages that are available commercially, and some of the more popular ones include Arena, Extend, Flexsim, Process Simulator, SimEvents, and Simio. While most of these are standalone platforms, Process Simulator works with Microsoft Visio, and SimEvents is an additional package to MATLAB's Simulink environment. The Simio model is visualized in a three-dimensional virtual environment, and this is useful when studying problems with spatial components. There are also open-source projects such as OMNeT++, PowerDEVS, Simkit, SimPy, and SystemC.

Based on the criteria stated above, VE-OPS was developed on the Simulation in Python (SimPy) package [69] in the Python programming language. Python is an open source, object-oriented programming language that is platform independent [69]. SimPy is a process-based DES package for Python, and it is an open source project

as well [74]. SimPy has been used for various applications including teaching, disease epidemics, air space surveillance and logistics [74, 4, 98]. Although this platform is not as user-friendly as commercial software, it is light-weight and flexible, and it avoids the licensing challenges, making it a useful platform for research. An example using SimPy is provided below to show how a simulation is developed and executed.

Example 2.1: DES example using SimPy

A simple coding example is presented here to illustrate how DES models are created in SimPy. This example models the loading operations of an aircraft at an airport. The actual loading operation will simulated as a simple wait time. For the simulation, an `airport`, `aircraft`, `queue1`, and `queue2` objects are created. The cargo is represented as float (or real) values. The entire simulation will consist of the `airport` process grabbing an `aircraft` object from `queue1`, loading the cargo, and putting the aircraft into `queue2`.

The model parameters are summarized in Table 1.

Table 1: Mission parameters for the SimPy example

Task	Values	Units
Load Time	1.5	hr
Fleet Size	5	aircraft
Carrying Capacity	20	tonnes
Amount of Cargo	90	tonnes

First, the necessary Python packages are imported, and the variables in Table 1 are initialized.

```
# Package Imports
from SimPy.Simulation import *
import numpy

# Variable Initialization
load_time = 1.5
fleet_size = 5
carrying_capacity = 20.
total_cargo = 90.
```

Figure 12: Package imports and variable initializations

Next, the `aircraft` class is defined, as seen in Figure 13. The `aircraft` class is initialized with a name parameter and given a carrying capacity of how much cargo it can hold. At the beginning, the aircraft holds no cargo.

```
class aircraft(object):
    ''' Defines the aircraft class object '''
    def __init__(self, name, carrying_capacity):
        self.name = name
        self.carrying_capacity = carrying_capacity
        self.cargo = 0.0
```

Figure 13: Class definition for the `aircraft` class

Then the `airport` class is defined, as seen in Figure 14. It inherits from the `Process` class, which is a SimPy class that enables event scheduling within the simulation. DES is built up by scheduling these events and processing them sequentially. When an instance of the `airport` class is created, it is initialized with the reference to `queue1` and `queue2`, and it is given a cargo amount sitting at the airport. The `sim` variable is necessary to associate this object to the simulation being created. A `print` statement is also included to indicate

when the loading operation is completed for one aircraft. The `load_aircraft` method is the main routine for this class, and it defines its behavior during the simulation. Finally, the `get_cargo` method determines if there is enough cargo at the airport, and returns the lesser of the requested amount and amount left.

```
class airport(Process):
    ''' Takes an aircraft object from get_ac_q1, loads cargo, and
        puts the aircraft in put_ac_q2 '''
    def __init__(self, sim, get_ac_q1, put_ac_q2, init_cargo=0):
        super(airport, self).__init__(sim=sim)
        self.get_ac = get_ac_q1
        self.put_ac = put_ac_q2
        self.cargo = init_cargo
    def load_aircraft(self):
        while True:
            # Get an aircraft from queue1
            yield get, self, self.get_ac, 1
            ac = self.got[0]

            # Load cargo onto the aircraft
            capacity = ac.carrying_capacity
            cargo = self.get_cargo(amount_requested=capacity)
            yield hold, self, 1.5
            ac.cargo = cargo
            print ('At time ' + str(self.sim.now()) +
                   ', finished loading ' + str(cargo)
                   + ' tonnes of cargo onto ' + ac.name)

            # Put the aircraft in queue2
            yield put, self, self.put_ac, [ac]
    def get_cargo(self, amount_requested):
        ''' Determines the lesser of requested amount and amount
            of cargo left at the airport '''
        cargo = numpy.min((amount_requested, self.cargo))
        self.cargo -= cargo
        return cargo
```

Figure 14: Class definition for the `airport` class

The simulation is constructed using the components defined thus far. The simulation is first initialized. Then the aircraft, queues, and airport objects are instantiated. The queues use a special SimPy class called `Store` that handles

the storage and retrieval of objects, which in this case are the `aircraft` objects. Because the `airport` is a SimPy `Process` class, it needs to be activated before it can schedule events. Finally, the simulation is run.

```
# Initialized the Simulation
sim = Simulation()
sim.initialize()

# Instantiate the aircraft
queue_of_aircraft = []
for i in range(fleet_size):
    ac_name = '_'.join(('aircraft', str(i+1)))
    ac = aircraft(name=ac_name,
                  carrying_capacity=carrying_capacity)
    queue_of_aircraft.append(ac)

# Instantiate the queues
queue1 = Store(name='Aircraft Queue 1',
               initialBuffered=queue_of_aircraft, sim=sim)
queue2 = Store(name='Aircraft Queue 2', sim=sim)

# Instantiate the airport
myAirport = airport(sim=sim,
                    get_ac_q1=queue1,
                    put_ac_q2=queue2,
                    init_cargo=total_cargo)

# Activate Process components and Execute
sim.activate(myAirport, myAirport.load_aircraft())
sim.simulate(100);
```

Figure 15: Main code block

Finally, the simulation output is given as print statements, which was programmed in the `aircraft` class, and the output text is shown in Figure 16.

```
At time 1.5, finished loading 20.0 tonnes of cargo onto ac_1
At time 3.0, finished loading 20.0 tonnes of cargo onto ac_2
At time 4.5, finished loading 20.0 tonnes of cargo onto ac_3
At time 6.0, finished loading 20.0 tonnes of cargo onto ac_4
At time 7.5, finished loading 10.0 tonnes of cargo onto ac_5
```

Figure 16: Outputs from the example simulation

In this way, new process blocks can be created to perform different tasks and chained together to form a more complex simulation. New objects such as vehicle parts and workers can also be defined depending on whether the simulation project needs to track these objects.

2.2.2 Simulation Flow

VE-OPS captures the sortie generation, maintenance, and spart parts logistics processes, which are the three main processes involved in operating and maintaining a vehicle. The simulation flow of VE-OPS is depicted in Figure 17. The arrows represent the flow of the simulation objects and the boxes are the processes. There are five types of simulation objects that are tracked, and these are vehicles, parts, missions, workers, and workstations. Although these objects have specific names, they are simply a container for attributes, such as the vehicle name, part life, and number of hours in operation. The processes perform different tasks and change the objects' attributes. For example, in the Mission Executor block, the vehicle will perform its mission, and the block will modify the number of hours the vehicle was in use and record if the mission was a success or a failure. Once the process block completes its list of tasks, the object is passed to the next process like on an assembly line.

All objects in the simulation are derived from a generic simulation object class that has an identification and name properties. The vehicle class has properties associated with the usage of the vehicle as well as parts and missions assigned to it. The vehicle has the capacity to carry multiple parts, and depending on the configuration, it can be partially mission capable. The vehicle object also tracks the amount of time it is

in operation as well as the number of missions it has executed. The vehicle class also has methods associated with updating its status, adding flight hours, and removing and installing parts.

The part class also has similar properties as the vehicle class such as flight hours and number of flights flown. The life of each part is pre-calculated when it is created or fixed using a distribution associated with it. As these parts are used by the vehicle, it accumulates flight hours, and when the flight hours exceed the part life, it is declared broken. When a part object is created, it is also assigned a distribution for time to repair it, time to replace a broken part from the vehicle, and time it takes to transfer between the supply depot and the repair facilities, and a new value is calculated each time one of these operations are performed.

The mission tracks the length, mission type, and success or failure. The mission length can be set separately using a distribution. The worker and workstation classes can be used to introduce operational constraints, such as the total number of workers and available workstations. Furthermore, workers can have specific working hours, skill sets, and locations they are supposed to work at. Workstations can have types of fixes it can perform as well as location information.

The process blocks can be categorized into three groups, which are the sortie generation, maintenance, and parts repair processes. The sortie generation process is responsible for using the vehicles and breaking the parts. It consists of the Mission Manager, Mission Executor, Mission Generator and Mission Assessment blocks. The Mission Manager is responsible for assigning vehicles to missions, and some vehicles may not be able to perform certain missions so there is a matching that occurs.

Currently, the matching occurs by picking the mission at the top of the stack of missions and picking the first vehicle that can execute it, and there is room for improving this matching to optimize the number of missions flown. The Mission Executor is responsible for executing the missions and determining if a vehicle part fails due to usage. When a part breaks while the mission is being executed, the mission is aborted. The Mission Generator creates missions based on scenario inputs, and it has the ability to create a mission profile with a varying number of missions per day. Additionally, different mission types can be generated such as air-to-air and reconnaissance missions, and the mission type is responsible for defining its length and distribution. As missions are generated by the Mission Generator, its type is chosen randomly from a list of possible missions, and the probability of picking a specific mission type can be varied as well. The Mission Assessment process tracks the mission success, the aborts, and the incomplete missions that could not be flown that day if the missions are not allowed to rollover to the next day.

The maintenance process determines if a vehicle needs a repair and fixes it if necessary. It consists of the Vehicle Assessment, Maintenance Manager, and Repair Vehicle blocks. The Vehicle Assessment process determines if the vehicle needs to go to repair or into the working vehicles queue. Depending on the repair policy, it may assign only some of the broken parts to be replaced to bring the vehicle to partial mission capable status. A vehicle can operate while having broken parts that are not critical to its missions. The Maintenance Manager process coordinates the available resources to repair the fleet of broken aircraft. It pulls the necessary parts, workers, and workstation objects that are needed to perform the maintenance activity. The

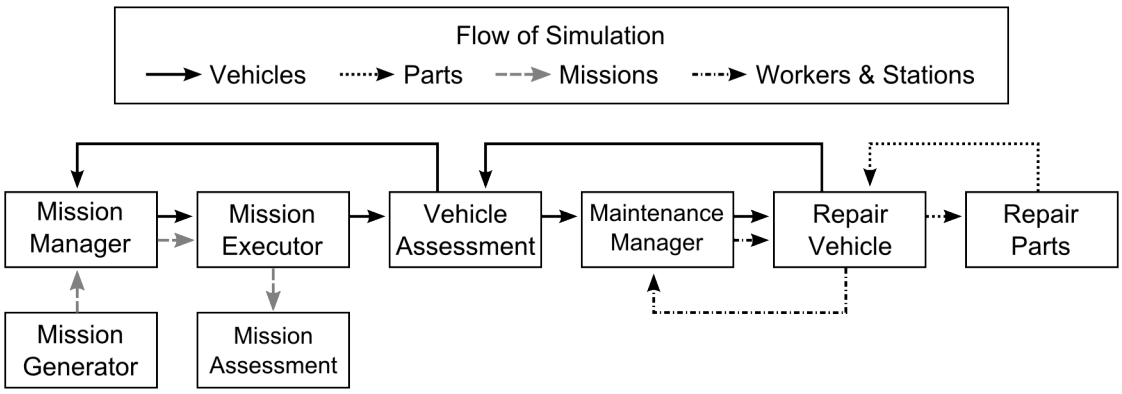


Figure 17: Simulation Block Flow Diagram

vehicle is repaired in the Repair Vehicle process, and the repair is simulated by a wait time that corresponds to the time it takes to remove the broken part and install a spare part. The vehicle is then sent back to the Vehicle Assessment process to determine if further repairs are necessary. The broken part is sent to the part repair loop, and the worker and workstation objects are placed back into their respective pools of resources.

The parts repair process consists of the Repair Parts process. The broken parts are delivered to the process block, the part is held for a certain amount of time to simulate the part being repaired, and the part is returned to the spare parts inventory. This repair loop is simplistic, but it can be replaced with other supply chain and logistics programs or new blocks can be created to give more fidelity to this process. Alternatively, a parts maintenance loop can be developed like the vehicle maintenance loop.

This simulation block flow diagram is similar to Faas' simulation diagram, which was created for the sortie generation process, and his diagram included scheduled and unscheduled maintenance [24]. The blocks in VE-OPS do not include as much detail

as Faas' model, but it can be added easily. Furthermore, the VE-OPS can create models with a loop for spare parts supply chain and logistics.

2.3 Characterizing the O&S Simulation Data

Time sequential (TS) data from an O&S model is evaluated to learn more about the data and the behavior of the simulation model. The Fighter Aircraft Maintenance and Operations Simulation (FAMOS) model was created in partnership with LMC as an initial phase to provide insight into O&S modeling for aircraft fleets. It was designed to capture the overall trends of aircraft fleet performance such as A_O and mission effectiveness. The FAMOS effort was the precursor to the development of VE-OPS, and it follows a similar structure as described previously where there is a sortie generation, maintenance and spares logistics cycle.

The FAMOS model simulates fleet sizes of over 1000 aircraft to assess the aggregate impact on the spare parts logistics and fleet-level performance metrics. At the beginning of the simulation, all aircraft fly training missions to satisfy a monthly flight hour requirement. Around halfway through the simulation, two contingencies occur that require two subsets of aircraft to be deployed at a remote airbase, and these aircraft fly longer missions and fly more frequently. These contingencies are meant to burden the O&S system. Each vehicle also has one generic part that can be removed and replaced after it breaks, and in addition to the part failures, vehicles have a failure mechanism that is based on random chance. The overall failure rate of the aircraft is calibrated by adjusting the part life and the frequency of the non-part failure. The simulation is in turn calibrated by adjusting the spares inventory level

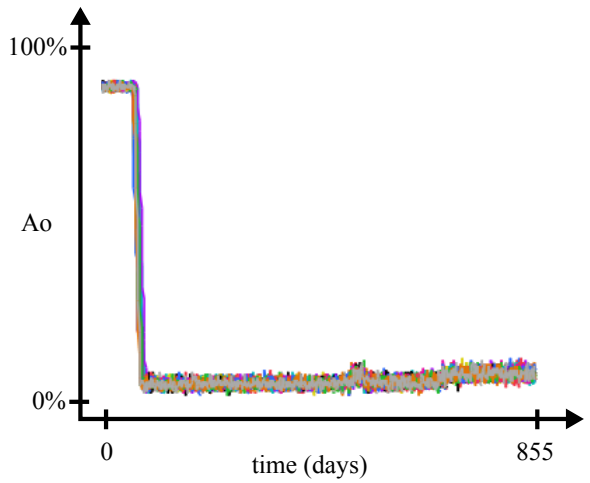


Figure 18: Example output from the FAMOS model with an overlay of 25 repetitions

such that the A_O of the fleet is at 90%.

A set of repetitions of a TS output for A_O is plotted in Figure 18. The chart shows the time history of the A_O of the training fleet, and it spans the entire simulation of 855 days. 25 repetitions were run for a set of input parameters, and the plot shows the overlay of these repetitions. In this scenario, the A_O quickly drops to zero, and it does not recover for the entire simulation. Either because of lack of parts or high break rates, the majority of the vehicles in the simulation remain broken except for the rare occasion when a vehicle manages to get repaired, but the part on the vehicle breaks again very quickly.

To illustrate how inputs vary the output behavior of A_O with simulation inputs, two input variable were varied, and the result is presented in Figure 19 as a table of charts. The plots vary horizontally by mean flight hours between failure (MFHBF). MFHBF is the average flight time between one part failure to the next, and it is an indicator of part reliability. The panels vary vertically with the global inventory multiplier, and this multiplier is applied to the calibrated inventory value of the

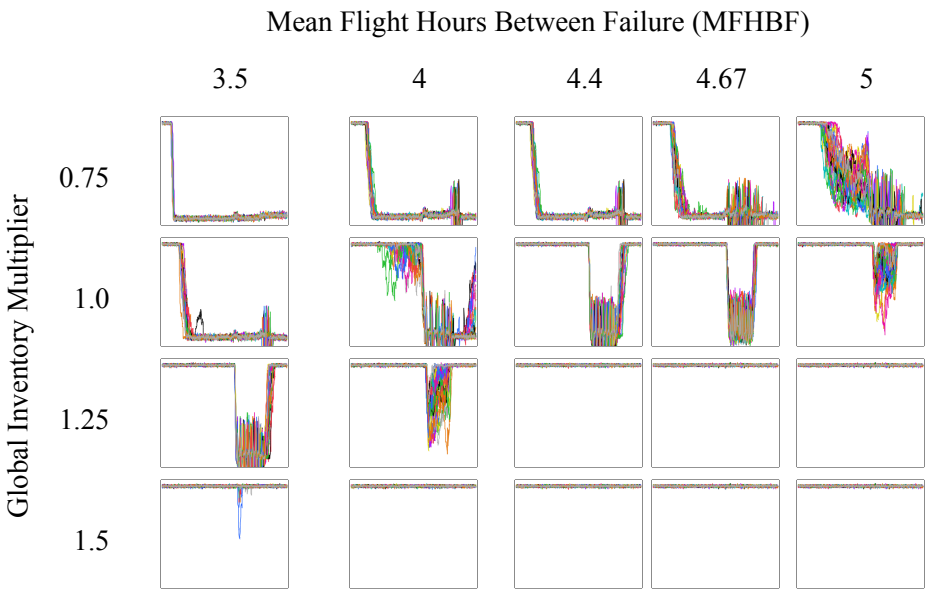


Figure 19: TS plots from FAMOS model. Each panel shows the operational availability (A_O) of training fleet vs time. (Horizontal spacing is purposefully non-uniform)

simulation.

The preferred state is the bottom right corner where the A_O is high throughout the simulation. As the MFHBF and global inventory are lowered, the A_O degrades. The top left panel shows how the A_O drops down close to zero as soon as the simulation begins. Around the midpoint of the simulation, the contingency operations begin. Because contingency missions have higher priority than training, parts are preferentially sent to the contingency bases at the expense of the inventory of the training. That is why the A_O for training suffers in the second half of the simulation in some of the panels.

The cause of low A_O produced from these simulations is mainly due to a lack of spare parts to repair the vehicle, and this shortage of parts can be caused by different mechanisms. The parts can be used faster than the rate that they can be replenished

into the inventory. The part consumption rate is influenced by how long and often the vehicles are flown and the life of the parts. For example, as seen in Figure 19, MFHBF is directly related to part life, and as MFHBF is lowered, the part consumption rate goes up. The rate of spares replenishment is affected by how quickly the parts can be repaired. If a broken part can be repaired immediately, then there would be no need for spares at all. There are different aspects of the system that can be varied, but ultimately, the goal is to balance the consumption and supply while optimizing metrics such as performance and cost.

As mentioned previously, the benefit of using a simulation model is the ability to adjust the parameters and observe the outcome quickly, independent of the actual O&S system. However, there is a limit to the number of solutions that can be evaluated due to the curse of dimensionality. Another aspect of the curse is the amount of data that is produced, and recording time sequential data as seen in Figure 19 results in large amounts of data. For the plots shown in the figure, there are 20 input settings and 25 repetitions, which is 500 simulation runs, and for each run, 855 time steps are kept, which results in 427,500 values. Another way to think about TS data is to compare it with how many input variables it corresponds to. For example, 100 time steps is on the same magnitude as seven variables at 2 settings each ($2^7 = 128$).

Observation 1

Time sequential data exacerbates the curse of dimensionality. For the same data volume, the length of the TS data must be traded off with the number of variables and their levels.

Another observation can be made based on the panels of line graphs in Figure 19. There is a progression of shapes that can be inferred, and if the plots are re-organized into a single row, it can be shown as Figure 20. Even though the input space is two-dimensional (MFHBF and global inventory multiplier), the behavior is one-dimensional. This result is consistent with what is known about how the A_O is tied to the movement of spare parts in the simulation.

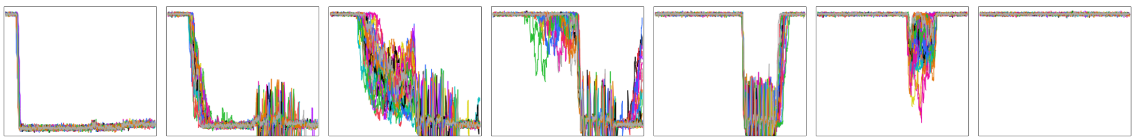


Figure 20: Panels from Figure 19 arranged in one dimension

The TS output from the FAMOS model can be considered as having 855 dimensions, which is one dimension per time step of the output. Based on the observation from Figure 20, this can be reduced to a handful of dimensions, and in the case of the FAMOS model outputs, 855 can be reduced to 1.

Observation 2
The time sequential output data generated by simulation models has the potential to be organized in a dimensional space that is lower than the original input dimensions.

In terms of regressions, the typical mapping from inputs to outputs ($X \rightarrow Y$) is accomplished through a third dimension ($X \rightarrow Z \rightarrow Y$). The intermediate space can be derived from the outputs that simplify the regressions ($Z \rightarrow Y$), and the inputs can be regressed to this intermediate dimension using normal regression methods.

The benefits of this intermediate dimensional space are further discussed in Chapter 3.

One last observation that can be made from the FAMOS model outputs is that the line graphs in Figure 19 can be approximated as piecewise linear segments. Although there are portions of the TS that are noisy, there are general trends that can be discerned. The average or median can be taken to reduce the noise due to variations between repetitions.

Observation 3
Time sequential data from long-term O&S simulations that have large transitions, such as A_O from the FAMOS model, can be approximated as piecewise linear segments.

2.4 Multirole Fighter Sustainment Scenario Description

The Multirole Fighter Aircraft Maintenance and Operations Simulation (MRFAMOS) model is the follow-on project to the FAMOS model, and it was developed using the VE-OPS toolset. The logic within the simulation was developed based on publicly available literature on O&S processes, such as from Faas [24], and it was verified with the help of researchers from LMC. The data and the results presented in this document do not include any proprietary information from the project; however, the output data is representative of what is normally expected from this simulation model. The MRFAMOS model follows the same simulation flow described in Section 2.2.2 and Figure 17. The goal of the scenario is to sustain a high A_O for a fleet

of aircraft, and the sustainment operations encompass the unscheduled maintenance and a simple spare parts supply chain.

The MRFAMOS model improves on the FAMOS model in several aspects. First, the simulation is structured to be more modular so that various model decision processes such as the Mission Manager and the Vehicle Assessment can be swapped out with different variants. This allows the user to adjust certain aspects of the simulation without touching the rest. More than one mission type can be defined, and although missions are just a length of time that the aircraft is flying, each mission type can have its own distribution. Furthermore, the aircraft can carry multiple parts, and each part can be labeled as being significant to specific missions. There are also constraints that can be placed on the number of maintenance workers and workstations. Consequently, because there are more objects and "moving parts" in the simulation, the runtime of the MRFAMOS model is longer than the FAMOS model for the same amount of aircraft, but it has more parameters that can be varied.

The MRFAMOS model is used to test the scalability of the SMARTS methodology. The description of this dataset can be found in Section 8.1.3.

2.5 Gray Box Modeling

Insights from the simulation can be used to help with the surrogate modeling process. In the field of system identification, this is called gray box modeling [64, 65]. When no information about the system is used to create a surrogate model, this is called black box modeling because the system is treated as a black box as depicted on the top in Figure 21. This leads to the following research question:

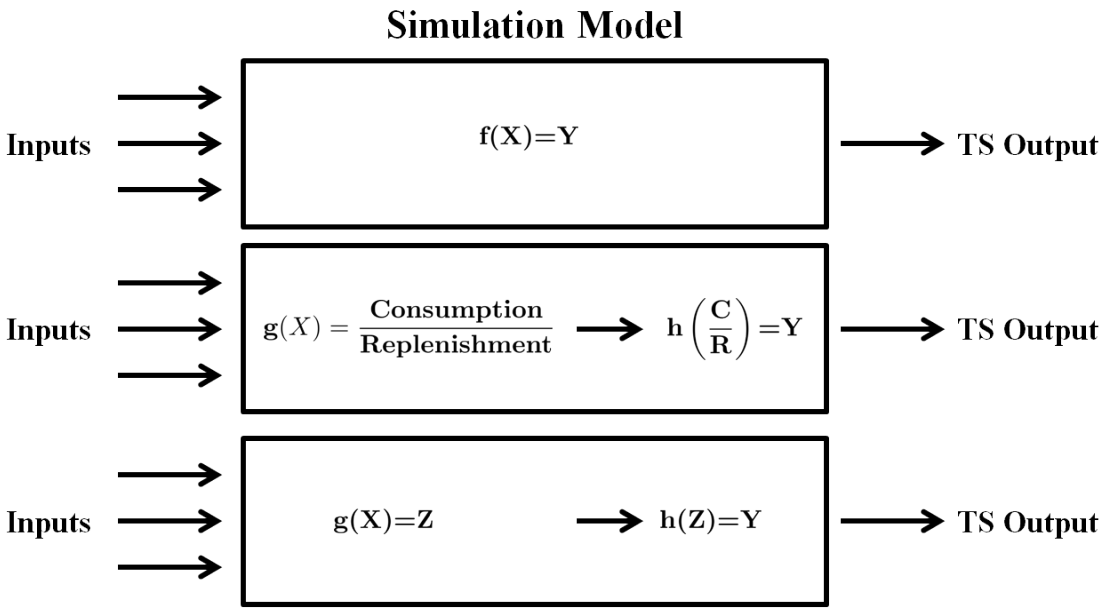


Figure 21: Depiction of black box and gray box surrogate modeling

Research Question 1
What aspects of O&S simulation models and its TS outputs can be exploited to help create surrogate models?

The FAMOS model can be described as the interaction of two processes, which are the consumption of parts and replenishment of parts. As mentioned before, the simulation model for O&S systems combines the sortie generation, maintenance, and spare parts repair and logistics processes. The sortie generation is responsible for breaking the parts on the vehicle, and maintenance replaces these broken parts. The net outcome of these two processes is to break parts. On the other hand, the maintenance process swaps a good part from inventory with a broken part, and the repair and logistics takes broken parts and replenishes the inventory with good parts. The net outcome is to fix broken parts. This interaction of consumption and replenishment

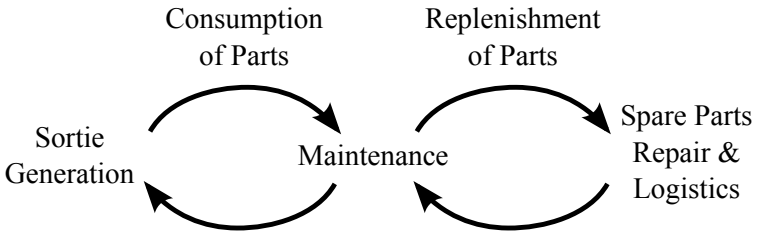


Figure 22: Generalizing the simulation processes with parts consumption and replenishment cycles

of parts is depicted in Figure 22.

To a large degree, the behavior of the simulation is dictated by the ratio of the rates at which the parts are consumed and replenished. If the parts are consumed faster than they are replenished, then spare parts in the inventory will run out and lead to lower A_O, and vice versa. Then, a wide range of TS shapes produced by the simulation can be explained by this ratio, and the input parameters that go into the simulation are responsible for changing the two rates. This also provides an explanation to Observation 2, which stated that the TS output shapes that were generated by the FAMOS model could be ordered as a linear sequence even though the input was multivariate.

This insight into the O&S simulation model can be generalized. When the input space is multi-dimensional and the simulation TS outputs can be organized in lower dimensions (as shown in Figure 20), then there is a set of hidden or untracked variables in the simulation that controls the outputs. In the case of the FAMOS model, this variable was the ratio between consumption and replenishment rates. Identifying these hidden factors are important in understanding the underlying mechanics of the simulation, but depending on the simulation, these factors may not be as intuitive or

physically meaningful. For the purposes of creating a surrogate model, the meaning of these variables is not important. However, the insight that the variables can be transformed into a smaller set of variables or dimensions is useful.

The simulation model, which was created using DES for FAMOS and MRFAMOS, can be described using a set of variables and functions. Using the FAMOS model as an example, the simulation model takes MFHBF and global inventory multiplier (x) and transforms them into an intermediate variable z which is ratio of consumption and replenishment ($z \cong \frac{consumption}{replenishment}$). Then using the value from z, the TS output A_O (y) is generated ($A_O = h(z)$). The functions $g()$ and $h()$ are responsible for transforming the variables between the three domains. The FAMOS model performs these transformations using simulation.

The surrogate model for the TS data can follow a similar two-step structure by separating the responsibilities of capturing the design space variability and creating the TS regression. The first function $g()$ is responsible for capturing how the shape changes as the input variables X are varied. Certain combinations of input parameters will result in the TS taking certain shapes. Using the FAMOS model outputs as shown in Figure 19 as an example, an MFHBF of 3.5 hours and global inventory multiplier of 0.75 results in a cliff-shaped TS line, while MFHBF of 4.4 hours and a multiplier of 1.25 results in a flat TS line. A new scale can be created to identify these shapes; for example, the cliff-shaped line can be 0 in this new scale while the flat line can be a 1. If the other TS lines are organized as in Figure 20, then they can be assigned values that fall between 0 and 1. The $g()$ function will be responsible for calculating the values on the new scale based on the input variables.

The second function $h()$ is then responsible for creating all the possible TS shapes from the simulation. Given a set of TS shapes, such as from Figure 20, $h()$ should be able to recreate those shapes with some level of accuracy and do so parametrically. Because $g()$ maps the X to Z, then $h()$ needs to be able to take the values in Z to recreate TS shapes in Y. So, when $g()$ returns a 0, then the $h()$ function needs to create a cliff-shaped line. This leads to a research question regarding the new structure of the regression that is being proposed.

Research Question 2
Will creating regressions that use an intermediate set of axes lead to a better regression when fitting TS data?

In order to answer this question, the regression methodology needs to be fully developed and tested. Creating a surrogate model using this divide-and-conquer approach takes three main steps. The first is to create the intermediate set of dimensions, which will be called Z. Then, the TS shape function $h()$, which outputs a TS line given a set of values from Z, is generated. Finally, the $g()$ function, which transforms X to Z, is created. The order of the last two steps is interchangeable, but for the purposes of the narrative, the development of the TS shape function is described first.

2.6 Summary

Modeling and simulation has supported O&S for over four decades, and its use is becoming more common due to a variety of factors such as improving user-friendliness and computational power. More models are being created, and the problems they

tackle are targeted to more specific problems. The FAMOS and MRFAMOS models were created to understand the complex nature of O&S systems. A sample dataset from the FAMOS model was examined more closely, and several observations regarding the characteristics of the data were made, which are the following:

Observation 1 Time sequential data exacerbates the curse of dimensionality.

Observation 2 The time sequential data generated by simulation models has the potential to be organized in a dimensional space.

Observation 3 Time sequential data from logistics simulation can be approximated as piecewise linear segments.

Insights were drawn from the simulation model in order to assist in the surrogate modeling process. With O&S models like the FAMOS and MRFAMOS models, the internal mechanism can be described as the interaction between the consumption and replenishment of vehicle parts. This led to the realization that there are hidden factors or variables (Z) inside of simulation models which drive its behavior and that the simulation can be conceptualized as a pair of transformation functions where the first transforms the inputs (X) into the intermediate domain (Z), and the second transforms the Z into the output (Y). A similar structure can be applied to the regression model where the first function identifies what the output should be based on the inputs, and the second function generates the actual data. To create this surrogate model, three steps are necessary, which are summarized in Figure 23, and these three steps constitute the Surrogate Modeling And Regression for Time Sequences (SMARTS) methodology.

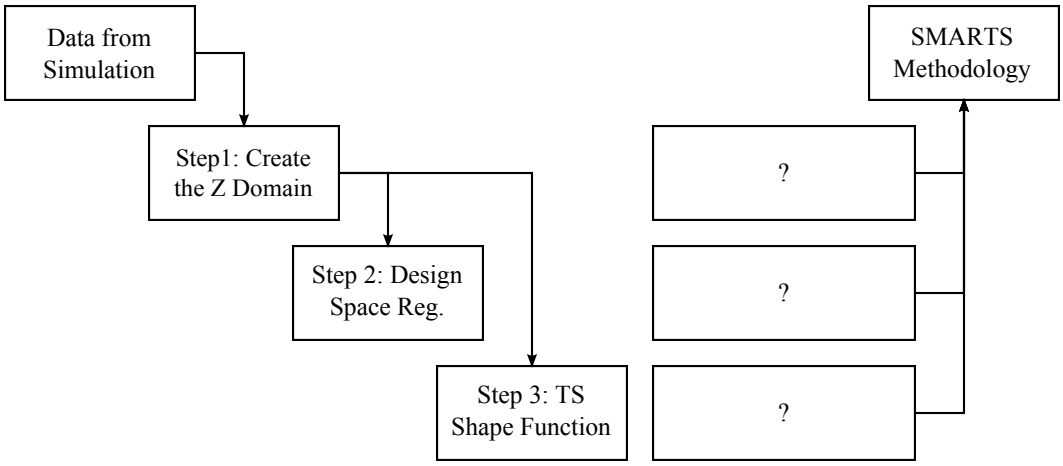


Figure 23: Three main steps for the SMARTS methodology

In the following chapters, the specific methods for each of the empty boxes on the right of Figure 23 will be defined. Next chapter focuses on Step 1 and Observation 2, and it explores dimensionality reduction techniques in order to reduce the data volume and find a way to handle nonlinear TS datasets.

CHAPTER III

DIMENSIONALITY REDUCTION

The previous chapter discussed the need to create an intermediate Z domain to separate the responsibility of capturing the design space variability and replicating the different TS output shapes. This leads to the following research question:

Research Question 3
What method should be used to derive a Z domain so that it can be used for TS regression?

To answer this question, this chapter expands on Observation 2, which stated that the dimensionality of time sequential (TS) data from operations and sustainment (O&S) simulations can be reduced. Dimensionality and data reduction techniques for high-dimensional data are covered, and in particular, principal component analysis (PCA) is described in detail. Then benefits of splitting the regression into two steps are discussed.

3.1 Dimensionality and Data Reduction Techniques

Data generation has been made easier thanks to the proliferation of computers, improvement in data storage devices and reduction in cost of size of sensors. Moreover,

there is a cultural shift towards taking, maintaining and analyzing data. Consequently, overabundance of data has become a problem, and there are various techniques to sift through this data to find the relevant and useful trends.

Dimensionality reduction is a category of techniques that looks for a relational organization of the data using fewer dimensions. The main assumption is that a dataset with hundreds of attributes (i.e. columns in the dataset) only has a handful of attributes that have a large impact to the trend of interest. This is an application of the Pareto principle and is the same concept as screening tests in design. Dimensionality reduction techniques are useful in determining these factors. There are many dimensional techniques and variants in literature, and some of the most popular ones are presented below. These are principal component analysis (PCA), multidimensional scaling (MDS), Isomap and locally linear embedding (LLE).

Similar to dimensionality reduction, data reduction aims to reduce the amount of volume while retaining the main features of the data. Data reduction methods can be used on simulation data, but the reduced data may be expressed in variables that do not exist in the original data. Two popular data reduction methods, discrete Fourier transform (DFT) and discrete wavelet transform (DWT), are presented as well.

To illustrate how the different dimensionality reduction techniques transforms the data, a sample dataset composed of black and white images is used. The image is of a black box, and the position of the box within the image is varied horizontally and vertically. The entire set of images are shown in Figure 24. Because the box moves in two dimensions, only two axes are necessary to differentiate between the images. Dimensionality reduction techniques will be applied to this set of images as

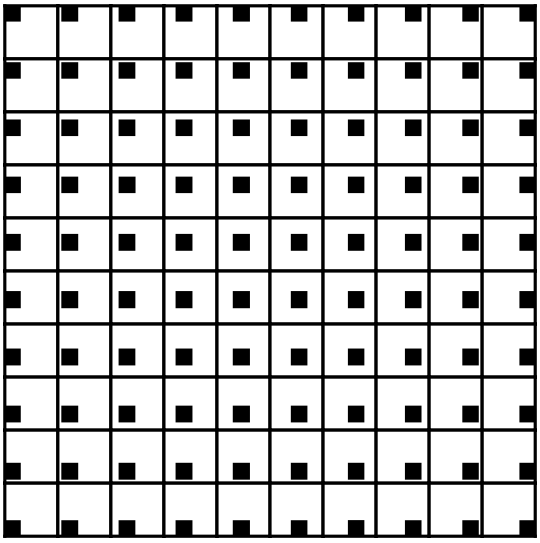


Figure 24: Set of images of a black box with a white background

an example of how each technique transforms the data.

3.1.1 Principal Component Analysis

PCA is a data transformation technique that aligns the data to a different set of orthogonal axes. The first axis or principal component (PC) is chosen so that it has the largest possible variance, and the successive PCs has the next largest possible variance while maintaining orthogonality. There are various ways PCA can be calculated such as by taking the eigenvalues and eigenvectors of the covariance matrix of the data [53]; however, many support the computation using singular value decomposition (SVD) as the best approach [36, 53]. The algorithm for PCA using the eigenvalues and eigenvectors of the covariance matrix is summarized in Algorithm 1.

An example using this algorithm is provided below to demonstrate how PCA works.

Algorithm 1 Principal Component Analysis

1: $X' = X - \bar{X}$ {Subtract the Mean}
2: $C = cov(X')$ {Calculate Covariance Matrix}
3: $det(C - \lambda I)$ {Calculate Eigenvalues}
4: $Cv - \lambda v = 0$ {Calculate Eigenvectors}
5: {Order the eigenvectors}
6: {Calculate the new axis values}

Example 3.1: Derivation of PCA using covariance matrix

The derivation of PCA using the covariance matrix is presented. This approach provides an intuitive understanding of what the method is doing compared to some of the other, more efficient algorithms. First set of values is plotted in Figure 25 and also presented in Table 2.

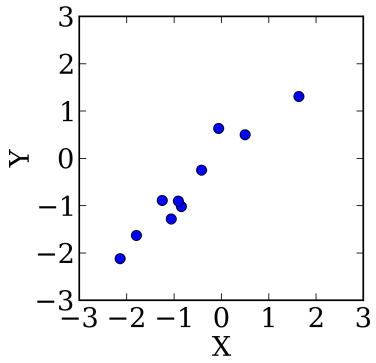


Figure 25: Plot of the data in Table 2

Table 2: Sample data for PCA example

X	Y
-0.417	-0.251
-0.056	0.631
-2.136	-2.124
1.640	1.305
-1.793	-1.632
-0.842	-1.021
0.503	0.497
-1.245	-0.893
-1.058	-1.282
-0.909	-0.906

Step 1: Subtract the Mean

The data is adjusted so that each column of data is subtracted by its mean. The resulting data is shown in Table 3.

Table 3: Mean adjusted data

X'	Y'
0.2143	0.3166
0.5753	1.1986
-1.5047	-1.5564
2.2713	1.8726
-1.1617	-1.0644
-0.2107	-0.4534
1.1343	1.0646
-0.6137	-0.3254
-0.4267	-0.7144
-0.2777	-0.3384

Step 2: Calculate the covariance matrix

The covariance of Table 3 is calculated. The equation for the covariance matrix is the following:

$$cov(X,Y) = \frac{\sum_{i=1}^{n}(X_i - \bar{X})(Y_i - \bar{Y})}{(n-1)}$$

The resulting covariance matrix is the following:

$$C = \begin{bmatrix} 1.235 & 1.166 \\ 1.166 & 1.185 \end{bmatrix}$$

Step 3: Calculate the Eigenvalues and Eigenvectors

Next step is to calculate the eigenvalues (λ) and eigenvectors (v) of the covariance matrix. The eigenvalues can be calculated by solving for the roots of the characteristic polynomial of C.

$$det(C - \lambda I)$$

The eigenvectors are vectors that satisfy the eigenvalue equation for C.

$$Cv - \lambda v = 0$$

The eigenvalues and eigenvectors of C are the following:

$$eigenvalues = \lambda = \begin{bmatrix} 2.376 \\ 0.0444 \end{bmatrix}$$

$$eigenvectors = V = \begin{bmatrix} 0.715 & -0.700 \\ 0.700 & 0.715 \end{bmatrix}$$

The eigenvectors form the new axis in which the data can be transformed, and the eigenvalues show the relative importance of each axis. The first eigenvalue has a very large value, and the eigenvector that is associated with it aligns with the data. The eigenvectors are plotted with the data in Figure 26.

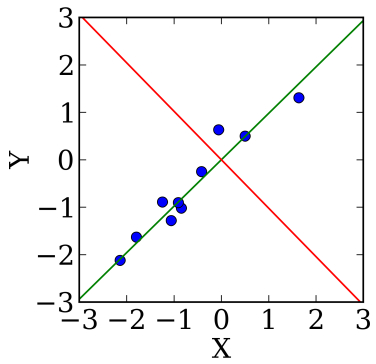


Figure 26: Plot of the data with the eigenvectors

Step 4: Order the eigenvectors

The next step is to order the eigenvectors based on decreasing value of eigenvalues. In the example case, the eigenvectors are already in the right order.

Step 5: Calculate the new axis values

The last step is to calculate the values of the data in the new coordinate system. This is accomplished by multiplying the transpose of the eigenvectors (V) and the mean adjusted data (M_{adj}).

$$\text{New Data} = V^T M_{adj}$$

The new values are shown in Table 4 and plotted in Figure 27.

Table 4: Data transformed into new axes

X_{new}	Y_{new}
0.375	0.076
1.250	0.454
-2.164	-0.060
2.933	-0.251
-1.575	0.052
-0.468	-0.177
1.555	-0.033
-0.666	0.197
-0.805	-0.212
-0.435	-0.048

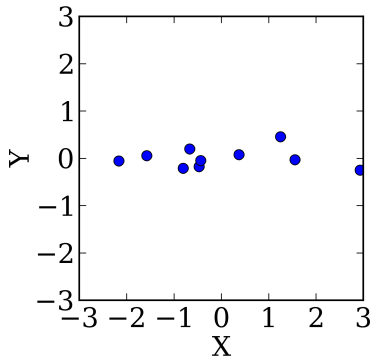
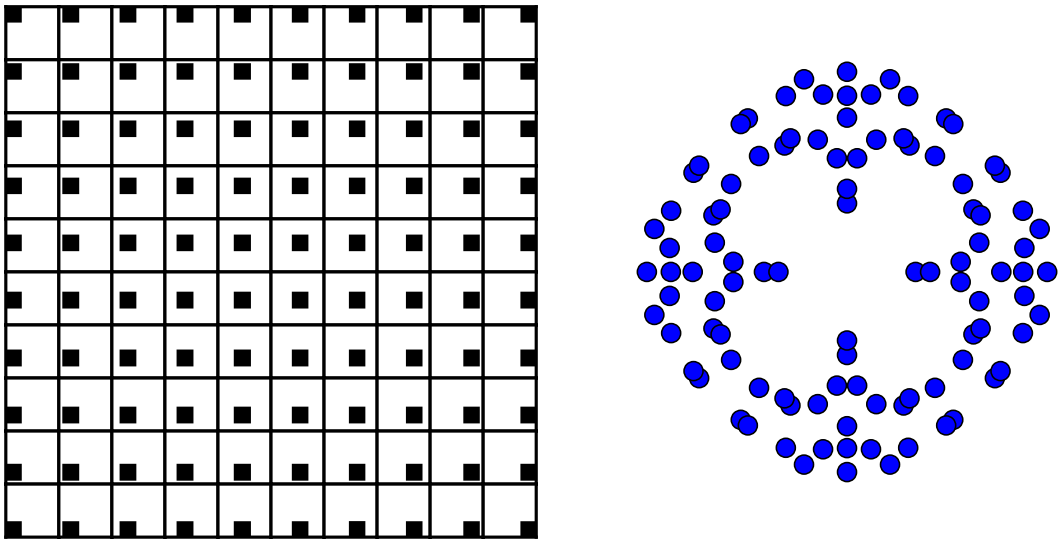


Figure 27: Data transformed into new axes (Table 4)

PCA is applied to the set of images with the black boxes, and the result is shown

in Figure 28. The figure on the right plots the set of images as shown on the left using the first and second PC. Each dot from the right figure represents an images from the left figure. Although PCA finds some form of structure between the images, it does not appear that the method found the expected uniform grid-like structure. This is because PCA is a linear dimensionality reduction technique while images typically require nonlinear techniques. For TS data from O&S applications, it will be shown later that PCA is one of the best methods.

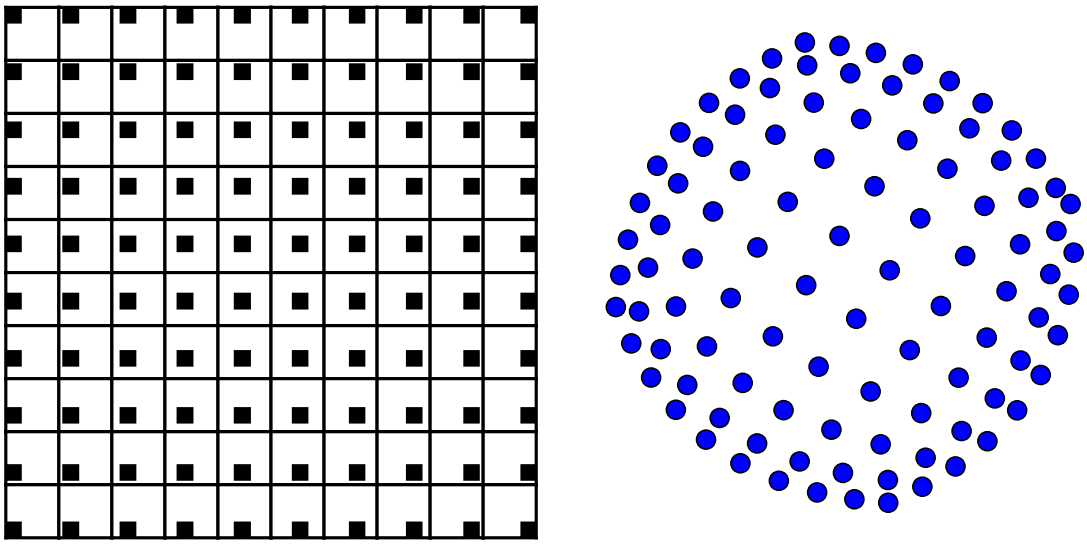


(a) Original Images (b) PCA

Figure 28: PCA applied to the set of images with the moving black box

3.1.2 Multidimensional Scaling

Multidimensional scaling (MDS), similar to PCA, is a linear dimensionality reduction technique. Classical MDS calculates a reduced set of dimensions that preserves the interpoint distances, and it is equivalent to PCA when Euclidean distances are used [118]. Given a matrix D of squared distances between all the points in the dataset,



(a) Original Images (b) MDS

Figure 29: MDS applied to the set of images with the moving black box

MDS calculates the eigenvalues and eigenvectors of the matrix $-\frac{1}{2}HDH$, where $H = I_{n\times n} - \frac{1}{n}\mathbf{1}_{n\times n}$ and n is the number of data points. Then, in a similar fashion to PCA, the original data can be projected onto the new coordinate system created by MDS. The MDS algorithm is summarized in Algorithm 2.

Algorithm 2 Multidimensional Scaling

1: M_{dist} {Calculate distance matrix}
2: $D_{ij} = M_{ij}^2$ {Square distance matrix}
3: $-\frac{1}{2}HDH - \lambda I$ {Calculate Eigenvalues}
4: $H = I_{n\times n} - \frac{1}{n}\mathbf{1}_{n\times n}$
5: $-\frac{1}{2}HDHv - \lambda v = 0$ {Calculate Eigenvectors}
6: {Calculate New Values}

Figure 29 shows the results of MDS applied to the set of black box images. Although MDS is a linear method, it performs significantly better than PCA in organizing the images. The right figure shows how the images have a grid-like relationship; however, the arrangement of the points are not straight, making it look "bloated".

3.1.3 Manifold Learning Techniques

Manifold learning techniques use a different approach to find a lower dimensional relationship between the data compared to PCA. Manifold learning is a general class of methods for nonlinear dimensionality reduction, and two of the most popular ones are presented - Isomap and locally-linear embedding (LLE).

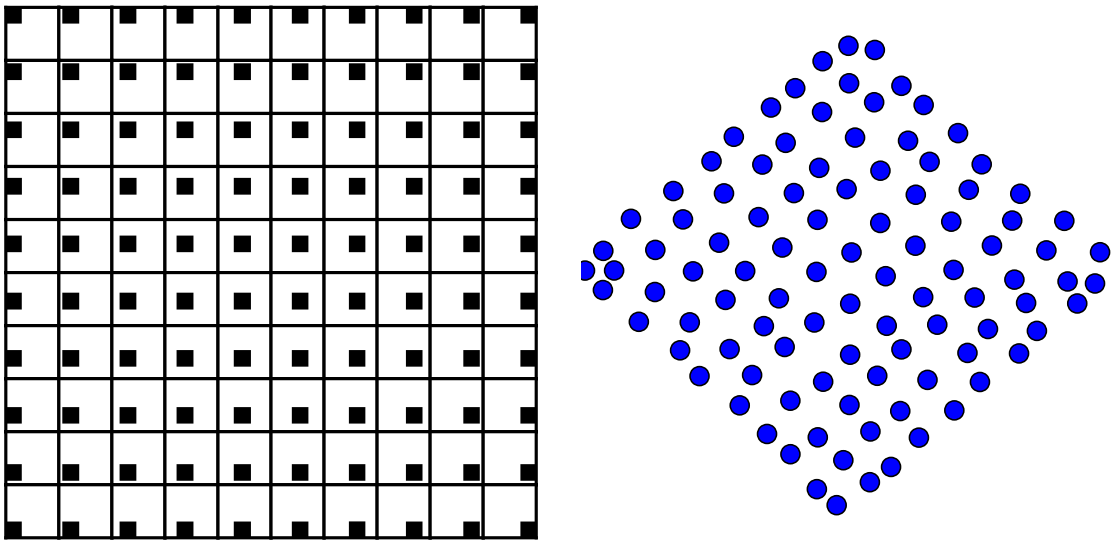
Isomap

The isometric feature mapping (Isomap) tries to preserve the "intrinsic metric structure" or the distance along an inherent manifold [117]. Isomap is very similar to MDS, and in fact, it uses a portion of the MDS algorithm as its last step. The key difference with Isomap is that it calculates the distance along geodesic paths instead of a Euclidean distance. The algorithm has three main steps [118]. The first step is to construct the neighborhood matrix. Neighbors of each point is determined by either defining a fixed radius ϵ or K nearest neighbor. The next step is to calculate the shortest distance between all pairs of points using the neighborhood matrix. The final step is to apply MDS to the matrix of distances calculated in the second step. The Isomap algorithm is summarized Algorithm 3.

Algorithm 3 Isomap

1: Construct neighborhood matrix
2: Calculate shortest distance between all pairs of points
3: Apply MDS to matrix of distances

Figure 30 shows the results of applying Isomap to the black box image set. It is better able to capture the global relationship as shown by the "square" configuration of the points, but the local spacing is not uniform.



(a) Original Images (b) Isomap

Figure 30: Isomap applied to the set of images with the moving black box

Locally Linear Embedding

In contrast to Isomap which tries to preserve global distances, locally linear embedding (LLE) is focused on preserving local distance structures [95]. The algorithm begins by finding neighbors of each point by using methods such as K nearest neighbors. Then, each point is estimated using a weighted sum of the neighbors determined in the first step with a constraint that the weights must sum to 1. Finally, a new set of lower dimensional coordinates Y_i is calculated by minimizing embedding cost function:

$$\min_{Y} \Phi(Y) = \sum_{i} \left| \overrightarrow{Y_i} - \sum_{j} W_{ij} \overrightarrow{Y_j} \right|^2$$

which can be rewritten as

$$\Phi(Y) = \sum_{ij} M_{ij} (\overrightarrow{(} Y_i) \cdot \overrightarrow{Y_j})$$

where

$$M_{ij} = \delta_{ij} - W_{ij} - W_{ji} + \sum_{k} W_{ki} W_{kj}$$

$$\delta_{ij} = \begin{cases} 1, & i = j \\ 0, & i \neq j \end{cases}$$

The optimal embedding is then found by calculating the bottom $d+1$ eigenvectors of the M matrix, where d is the number of desired dimensions, and the last eigenvector is discarded because it is the unit vector with all components. The LLE algorithm is summarized in Algorithm 4.

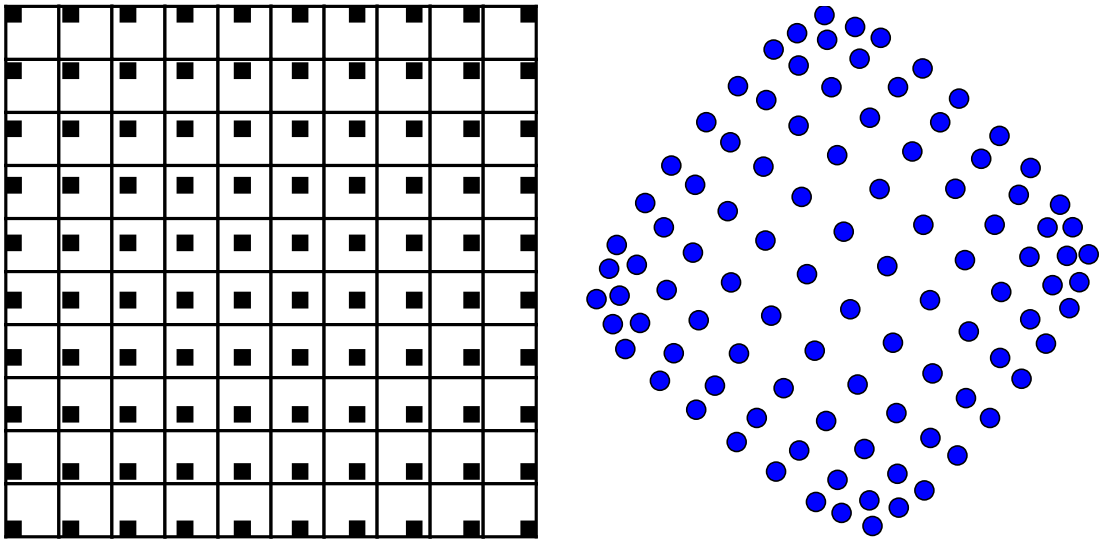
Algorithm 4 Locally Linear Embedding

1: Find neighbors of all points
2: Reconstruct each point as a weighted sum of its neighbors
3: $\min \Phi(Y) = \sum_i \left| \overrightarrow{Y_i} - \sum_j W_{ij} \overrightarrow{Y_j} \right|^2$

Figure 31 shows the results of applying LLE to the black box image set. LLE tries to preserve the local relationships by representing each point using its neighbors, and this is apparent in the right figure. The points in the middle are evenly spaced, but the edge and corners are more crowded. The global structure (the "square" shape) is not preserved like with Isomap.

3.1.4 Discrete Fourier Transform

Discrete Fourier transform (DFT) is a signal processing technique that converts the TS data into the frequency domain [43]. It represents the TS data into a linear combination of sinusoidal waves, and each wave is represented with a Fourier coefficient, which is a complex number [107]. When the TS is converted into the frequency domain, most of the Fourier coefficients have a small amplitude and thus contribute little to the overall signal. These would be considered as noise in the signal. The data can be truncated to remove these noise components and to reduce the data size



(a) Original Images (b) LLE

Figure 31: LLE applied to the set of images with the moving black box

while retaining its important features. To reduce the data, DFT is applied to each TS in the dataset and set number of components are kept.

3.1.5 Discrete Wavelet Transform

Discrete wavelet transform (DWT) represents the data using a linear combination of a base function called a mother wavelet [57]. A wavelet is a function with a feature that is localized in time; for example, the Haar wavelet is like a square wave with one period (Figure 32). These mother wavelets are scaled and transformed to create a representation of the data. The DWT is similar to the DFT, but the wave components in DFT are defined globally while the wavelets in DWT are localized. This allows DWT to create wavelets that are specific to a subregion of the TS data, and this is useful for multiresolution analysis. The overall trends of the data can be coarsely defined with a few wavelets, and certain regions can be given more fidelity

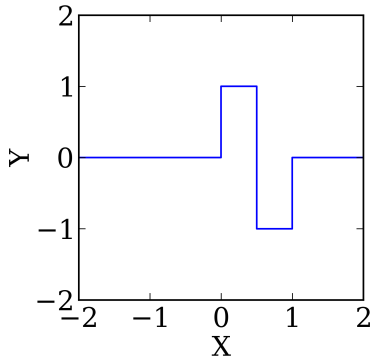


Figure 32: Haar wavelet

with additional wavelets [57]. Similar to DFT, wavelets that contribute little to the overall signal can be truncated [43]. To reduce the data with DWT, the transform is applied to each TS in the dataset and a certain number of components are kept.

3.2 Selecting a Dimensionality Reduction Technique for TS Data

The dimensionality reduction techniques were introduced and described in the previous section, and each technique was applied to a set of images to show that there are differences in the results due to the intent and the algorithms. This leads to the next research question.

Research Question 4
Which dimensionality reduction technique is the most suitable for TS of O&S simulation?

To answer this question, it is necessary to apply dimensionality reduction techniques to TS data. Because the goal of the dimensionality reduction is to find trends within the data, a sample dataset is generated with intentional relationships between

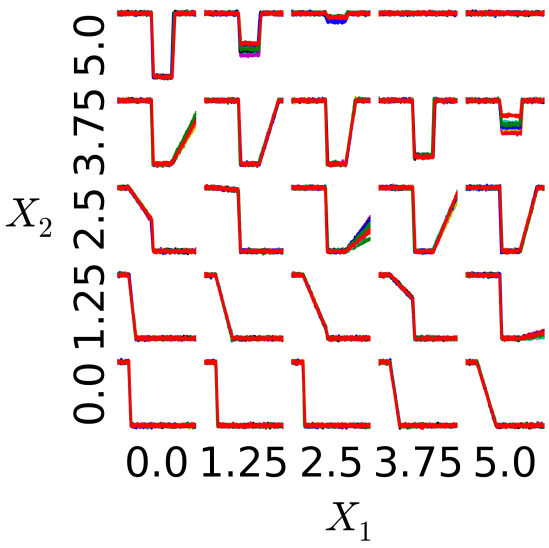
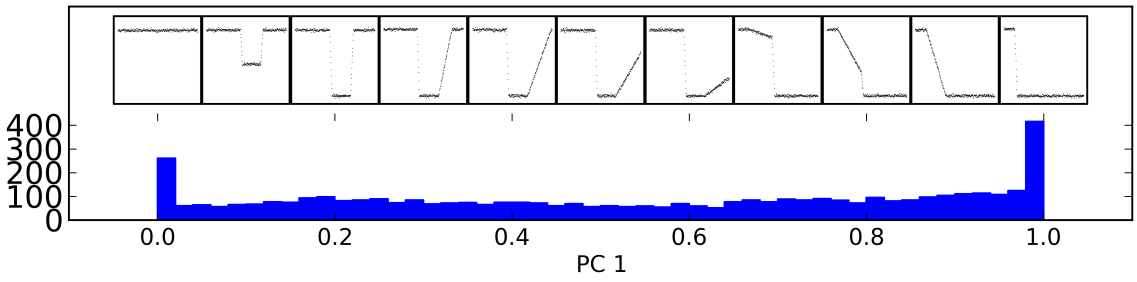


Figure 33: A matrix plot of TS outputs from a TS function

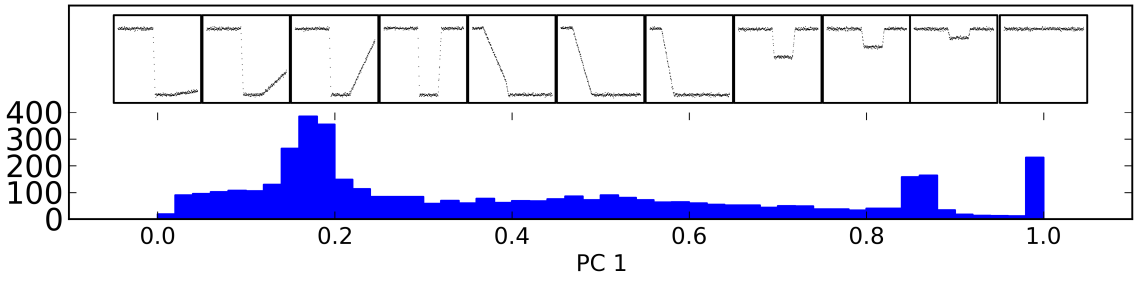
the outputs. A TS function $Y = f(X_1, X_2)$ is created such that it takes two inputs to create the TS outputs as shown in Figure 33.

PCA, MDS, Isomap, and LLE are applied to the TS outputs, and the results are plotted in Figure 34. In each of the plots, the bottom histogram represents where the TS plots from Figure 33 fall along the first axis, and the top row of TS plots shows the TS taken from along that axis.

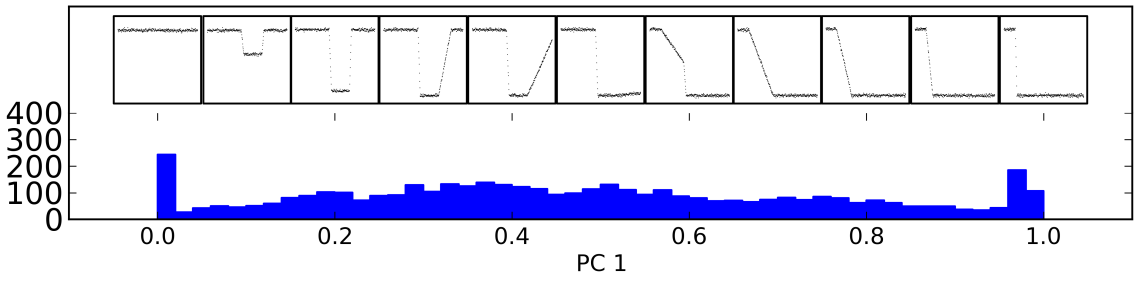
Among the four methods, PCA and Isomap have the correct order of shapes from the flat line to the cliff-shape line. The order of shapes produced by the MDS seems to jump around where the bucket-shape line is next to the cliff-shape line around PC1 = 0.35 and vice versa around PC1 = 0.65. The LLE method also misplaces the TS shapes such as the cliff-shape line at PC1 = 0.5, which should be at the end at PC1 = 1.0. Based on this investigation, PCA will be used to derive the Z domain.



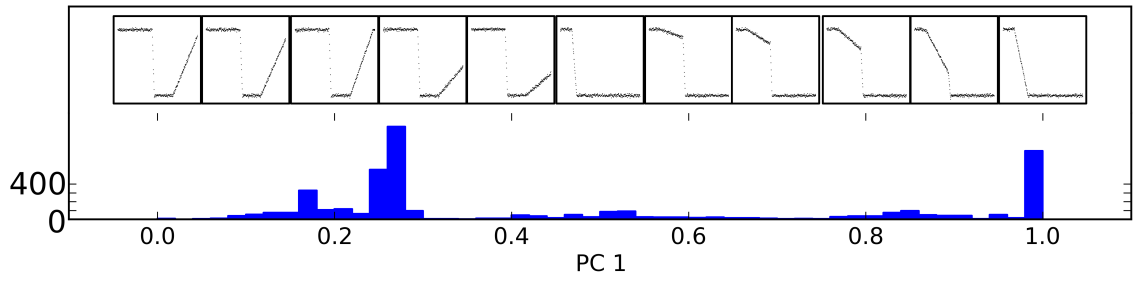
(a) PCA



(b) MDS



(c) Isomap



(d) LLE

Figure 34: Ordering TS dataset using dimensionality reduction techniques

3.3 Discretization of the Z domain

When a design space is sampled to create a surrogate model, the inputs points are spaced, typically using a design of experiments (DoE), to maximize the information gained from the simulation model or function of interest, and the distribution of the points is close to uniform. However, there is no guarantee that the distribution of TS outputs projected into the Z domain is uniformly distributed, and most likely it is not, as demonstrated in Figure 34. This could potentially be a problem when fitting a regression to the shapes in Step 3 of the Surrogate Modeling and Regression for Time Sequences (SMARTS) methodology because the shapes with a higher occurrence would essentially be weighted more than the others.

There is also a possibility that the Z domain is oversampled for the purposes of creating the $Z \rightarrow Y$ mapping. The Z domain can have fewer dimensions than the X domain, and this is the case with the FAMOS model. When this is true, the density of points in the Z domain is higher (assuming that the axes in X and Z are normalized). For example, 1000 points sampled from a design space with 10 variables has a lower sampling density or points per variable compared to 1000 points from 5 variables.

This increased density of points can be exploited in several ways. It is beneficial for the regression for the $Y \rightarrow Z$ mapping because there are more points that can be used for this regression. If there is little noise between similar points, a subsample of the data can be taken for the regression, and this helps reduce the total volume of data needed to create $y = g(z)$. In order to make the distribution of TS shapes uniform, the data can be binned or discretized according to a uniform grid in the Z

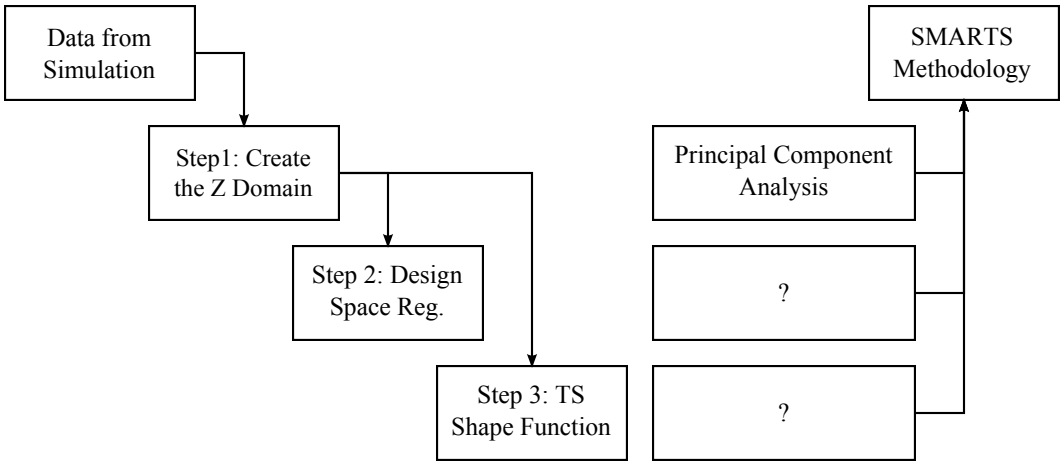


Figure 35: Three main steps of the SMARTS methodology with the Step 1 filled in

domain. If there is uncertainty in the model outputs, a representative TS shape can be derived from each of the bins using an average or median in a similar way to how repetitions are used.

3.4 Summary

This chapter presented an overview of dimensional and data reduction techniques based on Observation 2, which was on the lower dimensional nature of the TS output data from an exploratory modeling exercise. PCA, MDS, Isomap, and LLE were applied to a sample TS dataset, and PCA was chosen to be the most appropriate method to derive the Z domain from TS datasets. Figure 35 shows the three steps of the SMARTS methodology updated with the PCA as the first step.

It was also found that the number of TS needed to create the TS shape function can be greatly reduced. The data can be discretized and downsampled once the outputs are transformed into the Z domain because there are redundant shapes.

In the next chapter, Step 3 of the SMARTS methodology will be developed by exploring various surrogate modeling methods to find a technique that can help tackle

the problem of creating regressions for large TS datasets, which was a problem stated in Observation 1.

CHAPTER IV

SURROGATE MODELING AND PIECEWISE LINEAR REGRESSIONS

The previous chapter explored dimensionality reduction techniques to derive the Z domain that will be used to connect the inputs in the X domain and the time sequences (TS) in the Y domain, and it determined that principal component analysis (PCA) to be the most appropriate technique for Step 1 of the Surrogate Modeling and Regression for Time Sequences (SMARTS) methodology. This chapter and the next focus on finding techniques to create the TS shape function, which is Step 3 of the SMARTS methodology. This function is responsible for generating the corresponding TS data given a set of values from the Z domain.

The TS shapes can vary significantly within the design space, and this motivates the main research question for this chapter.

Research Question 5
What regression method is capable of representing the entire range of TS shapes in a TS dataset?

This chapter begins with a review on surrogate modeling literature in order to identify strategies to regress with large TS datasets. Among the regression methods that are introduced, attention is given to piecewise linear regression (PLR), which

divides data into smaller portions, and this approach helps reduce the complexity of the data and the regression fit time. This aligns with the Observation 3, which stated that TS data can be represented with linear segments, and it also helps alleviate the problem of trying to fit large and complex TS datasets (Observation 1). The rest of the chapter focuses on extending the PLR from representing a single TS to apply to the entire design space.

4.1 Surrogate Modeling

In Chapter 1, the difficulties of fitting a surrogate model to TS data for some design space problems were shown. In search of a solution, the surrogate modeling literature is first reviewed to gain an understanding of how different techniques perform their regressions. The literature spans many application domains including data representation, data compression, classification, clustering, design space exploration, optimization, and system identification. Some of the most popular methods include linear regression, neural networks (NN), and Gaussian Process (GP) models. There are additional techniques such as hybrids, ensembles, and piecewise regressions. A brief explanation of these various techniques are provided below.

4.1.1 Surrogate Modeling Methods

Linear Regression

In linear regression, the data is assumed to be in the form of the following equation [75]:

$$y = \beta_0 + \beta_1 x_1 + ... + \beta_n x^n + \epsilon = X\beta + \epsilon$$

where β are the linear model parameters and ϵ is the error, which is assumed to normally distributed with a mean of 0 and standard deviation of 1 $N(0,1)$. The model parameters are fit by minimizing the least squared error. It is one of the simplest and fastest surrogate modeling techniques, but it does not handle nonlinear data well. Response surface equations (RSE) used in engineering design is a linear regression.

There are a variety of extensions to linear regression to make it more flexible. One approach is to transform the data, assuming the underlying data can be fit with a linear model. For example, logistic regression assumes a logistic function, and this works well with binary data.

$$y = \frac{1}{1 + e^{-x}}$$

To overcome some of the problems associated with calculating the model parameters, people have proposed adding regularization parameters to control overfitting, and examples include ridge and lasso regression.

Piecewise Regression

Piecewise regressions are composed of multiple functions, and each function is used to represent a portion of the entire space. They typically have logical operators or activation functions to turn on or off functions depending on the input values. Continuity between functions is not guaranteed, but the boundaries can be made continuous and smooth using link functions such as sigmoid functions. There are several popular piecewise functions such as the regression trees [10], multi-attribute adaptive regression splines (MARS) [33], B-splines and P-splines [78, 22]. Piecewise

regression is discussed in further detail below in Section 4.2.

Support Vector Regression

The support vector machine (SVM) algorithm finds the optimal hyperplane that separates groups of points. SVM is used in classification tasks, and the relevant points are the ones that define the group boundaries. Support vector regression (SVR) flips the SVM problem formulation by calculating a "hyper-tube" that encompasses most of the points. SVR uses a regularization parameter that smooths the regression to control the "flatness" [113]. If needed, the dataset is transformed so that a linear separation can be achieved. Similar to NN, SVR produces a smooth function and can have smoothing problems.

Neural Network

The neural network (NN) regression model borrows its structure from the biological organization of neurons [75]. It is organized in layers, and the simplest is the 1-hidden layer model where there is a layer for inputs, a layer for outputs, and a layer of activation functions, which is the hidden layer. The hidden layer has multiple independent functions called nodes, and these are typically functions that are asymptotic between two values so that it operates like a step function. These nodes are then summed at the output layer. These coefficients or weightings are determined through a fitting or training process.

Different functions can be used for the activation function of the NN. The most common is the hyperbolic tangent. If radial basis functions are used, then the NN becomes a radial basis function network.

NNs use simple functions for the nodes of the hidden layer, but it can be highly

flexible due to its network structure. By adding additional nodes and hidden layers, it can fit highly nonlinear functions. Recursion can also be added to capture cyclic effects in the data.

Gaussian Process Regression

The basic idea of the GP regression, also known as Kriging, is that the data is sampled from a multivariate Gaussian distribution from a high-dimensional space, and this distribution is allowed to have a complex correlation structure [85]. New points are estimated from this underlying distribution. Because the model is based on a distribution, the output is the expected value at the new location.

Ensemble Methods

The reasoning behind ensemble methods is that a weighted sum of multiple regressors, which are derived from the same data, will yield better results rather than using a single regression [34, 37]. This aggregate or network approach is similar to NN. One downside to ensemble methods is that the model becomes less interpretable because it is now a collection. There are different approaches to creating ensembles such as the generalized additive model (GAM) and boosting methods. GAM takes a set of generalized linear regression models and uses a backfitting algorithm to fit and weight the models. Boosting on the other hand iteratively changes the weighting on the data based on how well successive models fit certain regions of the data. In this way, the next iteration tries to overcome the weaknesses of the previous model.

***K* Nearest Neighbor Regression**

K nearest neighbor (KNN) regression takes k number of nearest neighbors and estimates the value at an unknown location [20]. It retains the original dataset, and for

large design spaces, an equivalently large dataset is necessary for good results.

4.1.2 Methods Related to Time Sequences

ARIMA Model

TS data are usually correlated in time where each point is influenced by previous points. ARIMA models try to account for this correlation. ARIMA stands for autoregressive integrated moving average (ARIMA), and it is a combination of the three types of models. The model is usually written as ARIMA(p,d,q), where the parameters correspond to each of the autoregressive, integrated, and moving average models, and the functional form is given by the following equation [11]:

$$\left(1 - \sum_{i=1}^{p} \phi_i L^i\right) (1 - L)^d X_t = \left(1 + \sum_{i=1}^{q} \theta_i L^i\right) \epsilon_t \tag{2}$$

where ϕ_i are the parameters for the autoregressive part, L is the lag operator where $(LX_t = X_{t-1})$, X_t is the TS data indexed by t, θ_i are the parameters for the moving average part, and ϵ_t are the error terms $(\epsilon_t = N(\mu = 0, \sigma^2))$.

Kalman Filtering

Kalman filtering finds the optimal estimate of the system state from timed measurements with noise and other inaccuracies [54]. The system state is a property of a system such as position and velocity. Kalman filtering is useful for applications where the system state cannot be measured directly or without some uncertainty. The algorithm works in two steps, the prediction and update steps. It first predicts the next values of the system states and its uncertainties using a model that was created from previous measurements. Each time a new measurement is made, the model is

updated using weighted averages, giving preference to the estimates which have less uncertainty. Kalman filtering is used extensively in guidance, navigation and control because it can synthesize inputs from multiple sensors to provide higher accuracy as a whole compared to the accuracy of the individual sensors.

4.1.3 Discussion of regression methods

Generally, there are two metrics that are of interest when performing a regression, which are regression time and goodness of fit, and the goal is to reduce time and increase goodness of fit. The regression time is dependent on the flexibility or complexity of the method and the volume of data, and an increase in flexibility or volume corresponds to an increase in time. Goodness of fit is also dependent on the flexibility of the regression method and the complexity or nonlinearity of the data, and an increase in method flexibility increases goodness of fit while increase in data complexity usually decreases goodness of fit.

$$\text{Regression Time} \approx f(\text{Flexibility of Method, Volume of Data})$$

$$\text{Goodness of Fit} \approx f(\text{Flexibility of Method, Complexity of Data})$$

Regression assumes that there is some underlying function f that explains the relationship between the inputs x and the outputs y with some error ϵ, $y = f(x) + \epsilon$. The regression methods that were reviewed above use a combination of strategies to improve its ability to conform to nonlinear data. Some of these include:

- Varying the functional form
- Transforming the data

- Aggregating a set of functions
- Constructing a unique structure of functions
- Interpolating between input points
- Decomposing the data into smaller subsets

Data of different shapes can be fit by varying the functional form such as different order polynomials and trigonometric functions. Transformations can also be considered as part of the overall function, and it can help reveal a hidden structure within the data. When one function is insufficient, multiple functions can be combined and aggregated to absorb the complexity of the data using the structure, and ensemble and network methods are representative of this. Some methods rely on estimating data using its relationship to existing points. Finally, data decomposition manages complexity with the divide-and-conquer approach.

With respect to the two metrics for regressions, most of the regression strategies are relevant in terms of the flexibility of the method, and only the data decomposition is relevant to the data. Ideally, the regression method should be the simplest possible for the purposes of regression time and parsimony of the equation. If the data can be smartly decomposed so that the subsets have lower complexity, then the regression of each subset can be performed with a less flexible method.

The only class of regression techniques that uses subsets of data is the piecewise regression. One of the drawbacks to using piecewise regression is that a new problem of grouping the data is incurred. However, there maybe a way to extend piecewise regressions to large time sequential datasets.

Observation 4

Piecewise regressions divide the data into smaller groups, which can result in faster fit times. The partitioning of the data also aids in reducing the complexity of nonlinear data. This makes piecewise models an ideal candidate for fitting large nonlinear datasets.

4.2 Piecewise Regressions

The process of creating a piecewise regression can be split into two steps, the partitioning step and the regression step. The partitioning step creates the groups of points, and there are two main approaches to generate these groupings. The first is the recursive partitioning approach where groups are iteratively broken up into smaller groups until a fit criteria is met. The recursive approach in fact iterates between the partitioning and the regression steps, and regression trees and MARS are representative of this. The other approach is to predefine the number boundaries of the groups as done with B-splines and P-splines.

4.2.1 Regression Trees

The regression tree algorithm recursively partitions the data to create a hierarchical set of rules. It is also known as *recursive partitioning* and *decision trees*. At each step of the algorithm, the tree is grown by picking a design variable at random and picking a value at random, which becomes a new rule. Everything to the left of the value is one subset, and everything to the right is another. The resulting subset in tree modeling terminology is called a leaf or terminal node. A fit statistic is calculated

and compared to the tree prior to the new rule, and if it is an improvement, it is kept. This process is repeated for each of the leaves or subspaces until some threshold is satisfied. The algorithm can be improved by including different processes in addition to the growing step such as pruning less effective branches, changing the partition rule value, and swapping the rule order.

Regression trees were popularized through Breiman, Friedman, Olshen and Stone's classification and regression trees (CART) [10]. A Bayesian formulation was developed by Chipman, George, and McCulloch [16, 17, 18]. The original regression trees fit a mean to the terminal node or subset, and this is applicable in classification but not necessarily for regression because the output varies over the range. Different regression functions have been used at the terminal nodes starting with linear regressions [9, 32] to GP models [40, 39, 41].

One of the problems with regression trees is that it performs poorly when coupling effects are present. The method proposes cuts along a single input variable at a time so the subsets are always rectangular. There are two ways to overcome this shortcoming and increase the flexibility of regression trees. The first is to change the partitioning rules of the method from a univariate rule to a more flexible rule such as support vector machines (SVM) [15, 26]. The other approach is to use a flexible model at the terminal node such as GP models [40]. This increases the complexity and execution time of the tree regression, but this also results in a smaller tree.

4.2.2 Splines

There are various spline techniques that employ a chain of simple regressions connected by link functions or by sharing common points to enforce smoothness. These include MARS, B-splines and P-splines.

The MARS method is a nonparametric regression technique that automatically generates a piecewise linear regression [33]. It is composed of weighted basis functions, which can be a constant, a hinge function or a product of more than one hinge functions. The algorithm takes a forward pass to add the basis functions and a backward pass to prune the least effective ones, which is a function of the model complexity and goodness of fit. MARS borrows elements from regression trees, but unlike the latter, the models are continuous and can incorporate interactions.

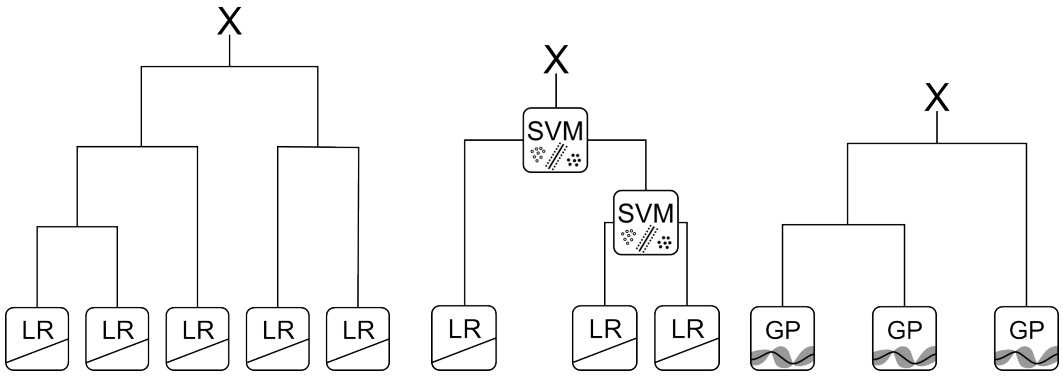
B-splines are piecewise functions that are composed of overlapping polynomials connected at specific locations called knots. They are an attractive option for regression because B-splines are flexible, continuous and differentiable up to a certain degree. However, it is difficult to optimize the number of knots and their locations automatically [22]. To make B-splines into a more robust surrogate model, a penalty function was added to the regression objective function [78, 22], and this new function is called P-splines. The model is given a large number knots which would overfit the data, but the penalty function helps control the smoothness of the model to produce a more ideal surrogate model.

4.2.3 Discussion of Piecewise Regressions

There is a balance that must be found between the number of subsets and the flexibility of the surrogate model [22] just as there is a balance between overfitting and underfitting in regression. Flexible models are more computationally expensive and do not scale linearly with the number of rows of data points. For example, Gaussian process models scale on the order of $O(N^3)$ time, where N is the number of data points [40]. If the regression is split into two GP models, it would take roughly a quarter of the time to fit, $O(2 \times (\frac{1}{2}N)^3) = O(\frac{1}{4}N^3)$. There is a natural synergy in splitting up the modeling into smaller pieces. Furthermore, there are times when some of the partitions are flat like a plane so a flexible model would only add unnecessary computational burden.

There are several strategies to handle complexity using piecewise regressions. Figure 36 shows three different types of regression trees. A traditional regression tree method such as CART uses linear regression at its leaf node to fit the data, and in order to fit nonlinear data, it has to increase the size of the tree to absorb the complexity. The resulting tree will have many small subspaces that are fit using linear regression models. In this approach, the method "absorbs" or compensates for the complexity by expanding the structure of the tree and making the tree proportionally as complex as the data.

Another approach is to use more flexible methods at the nodes and leaf. At the nodes, the CART method uses a simple rule to split the data. The data is split along a coordinate axis, and anything less than a certain value falls into one grouping and



(a) Simple Regression Tree (b) Regression Tree with SVM nodes (c) Regression Tree with GP leaves

Figure 36: Illustration of different ways of handling complexity (LR is linear regression, SVM is support regression machine, and GP is Gaussian process)

anything greater in the other. This rule can be made more complex. For example, SVMs can be used to divide the data, and this is beneficial if the partitions are nonlinear or if there is any interaction between the input parameters. This tree structure is depicted in Figure 36b. By using a more flexible node, the resulting tree will have less nodes and result in a smaller tree, and the complexity of the model has shifted from the structure of the tree to its nodes. Similarly, the leaf of the regression tree can be made complex instead by changing the linear regression into a GP, for example. Even if the partitioning is insufficient, a flexible regression method at the leaf can make up for the difference. Hence, more of the complexity is shared by the leaf, and the size of the resulting tree will be smaller. In these ways, tradeoffs can be made with how to absorb the complexity of the data using piecewise regressions.

There are different levels of integration when using partitioning and regression together. They can be performed separately or together. When they are performed separately, the subsets and their boundaries are determined first, and then each part

is fit individually. The benefit to this strategy is that different algorithms for partitioning and for surrogate modeling can be used according to the problem. One of the drawbacks with this approach is that there is no guarantee that the partitioned parts are simple enough to be fit by the surrogate model. Also, the continuity at the boundaries are not ensured unless boundary conditions are imposed or a link function is used.

Many of the methods that perform piecewise modeling perform the two functions together because the two can work in synergy. The regression provides the goodness-of-fit metrics that the partitioning algorithm can use to accept or reject the new partition boundaries. Because the number of partitions and the optimal boundary locations are unknown, some methods iterate between creating partitions and fitting the models until some convergence is met, and this iteration can take a while.

4.3 Extending Piecewise Linear Regression

In the time series data mining literature, piecewise linear regression can also be called piecewise linear approximation or representation, and it is found to be an acceptable way to represent time sequential data for various tasks such as compression and querying [58]. Depending on the level of coarseness, this is also visually acceptable. However, these representations are only created for each TS at a time.

Research Question 6
How can piecewise linear regression be extended to fit time TS data for design space exploration applications?

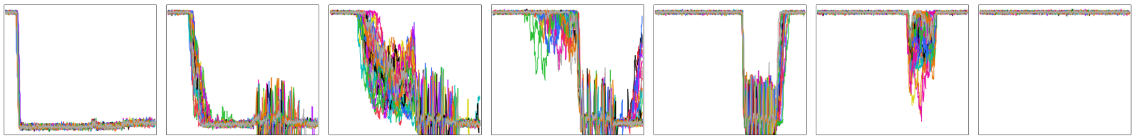


Figure 37: Panels from Figure 19 arranged in one dimension

In piecewise regressions for TS, functions are generally defined for specific time segments (t_i, t_{i+1}). This would need to be extended so that it can also accept parameters that change the shape of each of the functions, and the final form of the regression would need to assume something like this:

$$y(t) = f(t,z) = \begin{cases} g_1(t,z) & a_1 < t \leq t_2 \\ g_2(t,z) & a_2 < t \leq t_3 \\ \vdots & \\ g_k(t,z) & a_k < t \leq t_{k+1} \end{cases}$$

where z is the vector of parameters (in this case the coordinates from the Z domain), t is for time, g_i $(i = 1 \ldots k)$ is the piecewise function, k is the number of piecewise segments, and t_j where $j = 1 \ldots k+1$ is the breakpoint that defines the bounds of the segments. This assumes that the number of segments k and the breakpoint locations t_j are constant. However, these assumptions are too restrictive. As seen in Figure 37, some parts of the data can be represented with just one segment, while others require much more.

One option is to calculate the maximum number of segments that are necessary and apply that across the entire design space. The drawback to this approach is that it still does not fully address the problem with the breakpoint locations. The breakpoints can be assumed to be static, but this is not optimal. If the breakpoint

locations are allowed to vary across the design space, then breakpoint locations from all sequences can be collected, and the unique set of breakpoints can be applied to all sequences. In the limiting case, every time step in the sequence will be a breakpoint location, which does not seem efficient.

So far, only the segmentation of the individual TS have been considered. What if the data is also partitioned into smaller groups? Then perhaps the groups can be created with unique properties that lead to the creation of piecewise regressions with good fits.

Research Question 7
What unique properties should each group exhibit to create piecewise regressions with good fits?

In order to make groups that will result in a good fit for PLR, there are three main considerations.

Membership Are the points in the group the right ones?

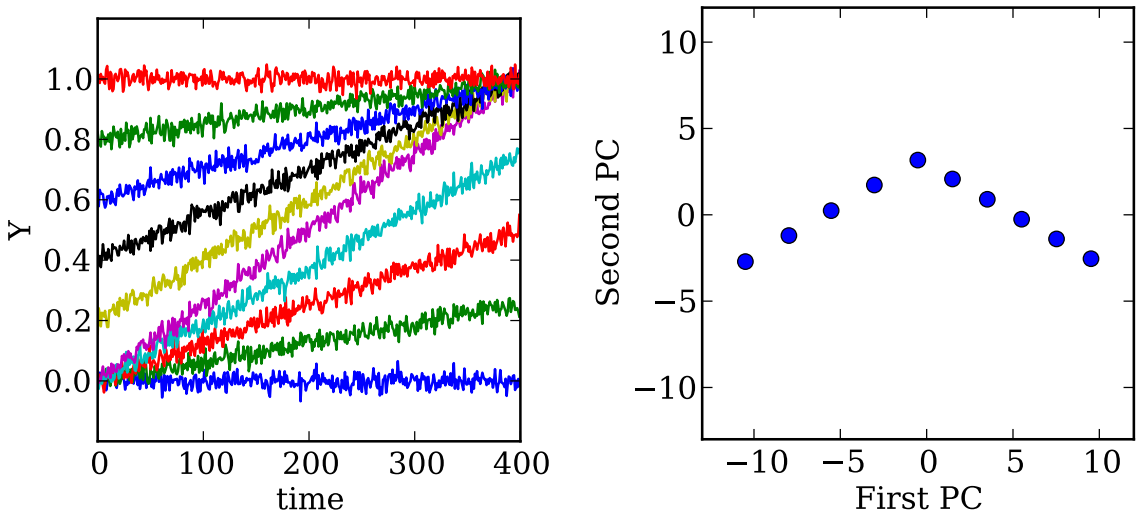
Alignment Are the set of breakpoints in the group similar?

Volume Are there enough points to make a regression?

If a these aspects are satisfied, the group of points will yield good fit characteristics when a PLR is created. These groups of points that satisfy the requirements will be called piecewise linear regression data groups (PLRDG) for convenience. Next, each requirement is discussed in further detail.

Membership

There are within-group and across-group conditions for membership. Within-group membership is concerned with shape of each TS point in the group and the number of transitions. First of all, each group needs to have a set of similar TS shapes and the number of segments need to be the same. This is the *similarity rule*. Furthermore, Second, these "similar" TS shapes should have either have the same general shape or be smoothly transforming from one shape to the next. This *transformation rule* is necessary to ensure smooth fits of piecewise regressions when the input parameters are varied. Figure 38a shows a set of lines with some normally distributed noise.



(a) Actual TS data

(b) First and Second PC

Figure 38: TS data with transformation and the data projected onto the first and second principal components

Because the regressions are going to be first-order linear equations, the transformation in each group should also be linear. As will be shown later, this *complexity rule* is solved by using a data-derived coordinate system that linearizes TS transformations.

The membership can be checked by performing PCA. If the shapes are similar with no transition, then the points should tightly cluster together; otherwise, there will be more than one cluster. If there is one transformation, then the result should be a linear line, and if there are two, then the line will be bent. The set of lines in Figure 38a has two transformations. The first transformation is the rise of the end point from 0 to 1 while the start point is held constant at 0. The second transformation is the rise of the start point from 0 to 1 while the end point is held constant at 1. When the data is dimensionally reduced using PCA, the plot of the first two PCs is shown in Figure 38b. The linear trends indicate the transformation and the bend shows that there are two transformations occurring in the dataset.

Across-group membership requires that there is no overlap between the groups within the parameter space in which the regression will occur. This constraint can be relaxed if the boundaries between groups are assumed to be fuzzy, but this leads to the question of transition rules between groups. To limit the scope of the current research, the groups are considered to be hard clusters.

Alignment

Alignment is strictly concerned with the accuracy of the breakpoint locations. Because the breakpoints are found using segmentation algorithms, there is a possibility that the breakpoints are incorrect despite having the same general shape. If the breakpoints are not correctly found, the regression of the segments will be biased because it would be similar to creating a regression with outliers. Furthermore, the breakpoint locations are also estimated using a regression, and it will end up providing biased starting and ending locations when estimating the segments.

Even though the number of segments may be the same, there is the possibility that the shapes are different, and this can be captured by checking alignment of the breakpoint locations in each TS. Because the breakpoints are allowed to vary, the *alignment rule* is necessary to ensure good fits when creating regressions of the breakpoint locations.

Volume

The linear regressions are hyperplanes that span the time domain and unknown number of design space variables so there needs to be a minimum number of points to which a regression is fit (*sparsity rule*).

Having decided on the structure of the regression, the next question is how groups are created from the data. Because the number of segments k per group is constant based on the *similarity rule*, one way to group the data is to cluster using k. However, each group should also result in having good fit characteristics when the regression is performed. Most piecewise methods use an iterative method to find the optimal groupings, but it is preferable if there are properties that can guarantee a good fit and are simpler to assess than actually fitting a regression.

This grouping problem falls under the field of data clustering literature, which focuses on finding clusters or groups of points that are associated with each other in some way.

4.4 Data Clustering

Breaking up a problem into smaller steps is a fundamental principle in problem solving [3]. The body of work that studies the meaningful decomposition of data is cluster

analysis or clustering, and various algorithms exist that accomplish this task. Subsets created by clustering can be called a block, cluster, part, segment, subsection, subsequence, subset and subspace depending on the field of study. The objects or functions that define the bounds of separation are called boundaries, divisions, partitioning rules, and rules. These terms will be used interchangeably to fit the context of the discussion.

Partitioning rules can be created using different approaches. They are typically generated according to a similarity metric such as the Euclidean distance, and the partitioning process uses information gathered from the data to find an optimal location for each boundary. Rules can be decided independent of the data, such as an equally spaced grid, but this does not take advantage of the natural groupings that occur in the data. Prior knowledge about the data structure can be used to create rules as well. Koch splits the surrogate modeling of a turbine design problem based on the physical and functional hierarchy of the design variables [59]. He notes, however, that natural groupings may not exist or the decomposition may be difficult for some problems.

Clustering is helpful when analyzing large data. It reduces the complexity of the data by grouping similar points. The analysis is made simpler because the data can be studied in parts, and new comparisons can be made across the subsets. This is also useful in regression because a simpler surrogate model can be used, but it does add an extra step to the overall regression process.

4.4.1 Clustering Methods

The field of cluster analysis has a comprehensive collection of black-box approaches to divide the data. Its goal is to gain new insight into the dataset by looking for similarities and differences of groupings of points. In literature, clustering techniques are generally classified into hierarchical and partitioning[1] clustering techniques [6]. With both hierarchical and partitioning algorithms, the number of clusters (n) is usually predefined. There are other techniques that do not quite fall into either categories such as density-based models.

Hierarchical clustering algorithms build up to n clusters. Partitioning clustering algorithms begin with n clusters and adjust the groupings until some convergence criteria is met. Hierarchical clustering can be further categorized based on whether it is a bottom-up or top-down approach, which in the literature is called agglomerative and divisive clustering, respectively. An agglomerative algorithm will first determine which points are closest to each other to form a cluster, and the distance that determines the cluster boundary is increased until there is one cluster. Divisive algorithms start with one cluster and create smaller clusters until n number of clusters are created or until each point is in its own cluster. In this way, hierarchical algorithms create a hierarchy of clusters which can be represented as a tree or dendrogram. Classification tree and methods based on singular value decomposition fall under the divisive category. There are different ways of measuring the distance or length of linkages

[1]It is important to note that the term *partitioning* in the cluster analysis literature refers to a set of particular techniques, while in this document, it is used broadly to refer to any method that purposefully splits the data.

between points such as a simple Euclidean distance. Because the distance between each point must be calculated, it does not scale well for large number of points.

Partitioning clustering algorithms can be further classified based on the approach of how clusters are created [6]. In the first kind, the clusters are defined by optimizing an objective function such as the distance between points to the center of the cluster and the distances between cluster centers. K-means and k-medoids fall into this category. These algorithms begin with n reference locations within the data space from which a cluster-defining boundary or an association rule is calculated, and the locations are updated according to the points around it. The other is to take a "conceptual" view of a cluster, and the algorithm determines the underlying parameters that define the cluster [6]. Model-based or probabilistic models fall into this category. Model-based models assumes that the data is a mixture or combination of an unknown number of multivariate normal distributions [30]. Based on the data, it infers the size, shape and orientation of the distributions. It also identifies groups automatically and is robust to noise. It performs well if the clusters are ellipsoidal.

Density-based clustering algorithms such as DBSCAN (Density Based Clustering of Applications with Noise) create clusters based on the proximity of points [23]. These methods apply the concept of density by measuring the distance between neighborhood points. If the points are within some threshold of each other, they are grouped as being part of the same group, regardless of the shape of the cluster, while the points which are far away from these dense clusters are categorized as outliers. It is also one of the few methods that can identify outliers in the data.

Other techniques exist that do not fit in this classification scheme for clustering

algorithms [6]. There are grid-based methods that divide the data into subspaces. Another variation is to apply the methods that are mentioned above on the statistics calculated from the data instead of working with the data directly. Other methods use the attributes, which are in the columns, instead of the rows of the data.

Having reviewed some of the clustering methods available, this leads to another research question:

Research Question 8
What is the appropriate clustering technique to create groups that will yield piecewise linear regressions with good fits?

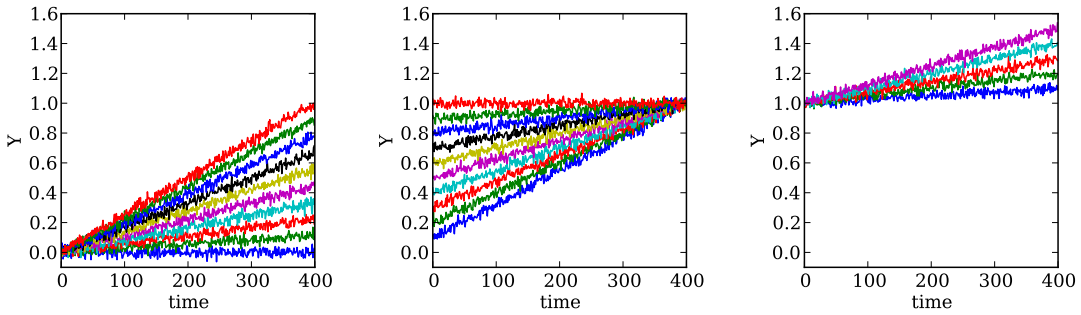
In order to reduce the number of candidates, the methods will be examined in the next section.

4.4.2 Discussion of Clustering Methods

Based on initial evaluation of the available clustering methods, the hierarchical clustering methods and partitioning methods that rely on an objective function, such as K-means, do not seem appropriate because they require the user to specify the number of clusters to generate. This can be overcome by sweeping through the number of clusters and calculating a goodness metric that penalizes solutions that creates unnecessarily large numbers of clusters. However, after the data is processed with methods like PCA, it is expected that the points will be organized in a line and not in hemispherical clusters. Hierarchical and partitioning clusters are better suited for *round* clusters and not for the data organized in lines, which is the case with data examined in this thesis. This point is highlighted using Example 4.1.

Example 4.1: Hierarchical and Partitioning Clustering on TS data with 3 transformations

To illustrate the point that hierarchical and partitioning algorithms are not well suited for TS data that varies smoothly across a design space, a sample dataset is depicted in Figure 39. The dataset is composed of lines, each with 100 time steps and small amount of noise. Furthermore, it is contains 3 transformations: (1) end point rises from 0 to 1, (2) start point rises from 0 to 1, and (3) end point rises from 1 to 1.5. In total, there are 25 rows of time sequences. The last transformation is shorter than the first two to give some asymmetry to the dataset.

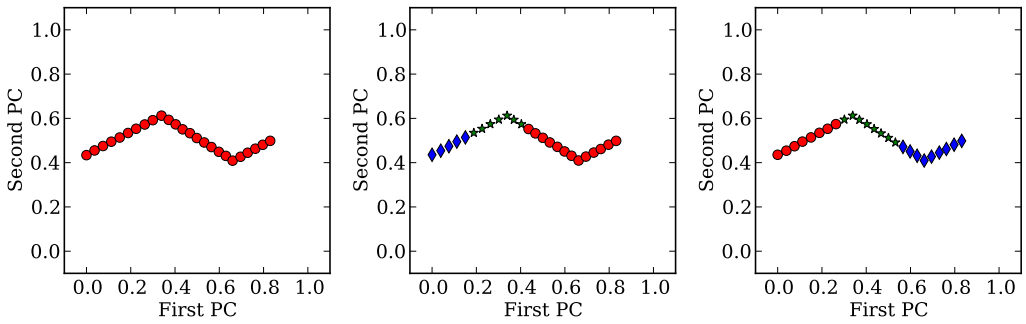


(a) Transformation 1 (b) Transformation 2 (c) Transformation 3

Figure 39: Sample dataset for clustering example

PCA is applied to this dataset, and Figure 40a shows the data plotted against the first and second principal components, and the points are colored by their correct clusters. For the hierarchical clustering, a method based on the Ward algorithm from Python's scikit-learn package is used, and for the partitioning,

the K-means algorithm from scikit-learn is used [79]. As there are three transformations, the algorithms were run to find three clusters, and the results are shown in Figure 40b and 40c, respectively.



(a) Correct Clustering (b) Hierarchical (Ward) (c) Partitioning (K-means)

Figure 40: Data plotted along the first and second principal components

Hierarchical and partitioning algorithms fail to make a distinction when the trend changes in the data, and the algorithms make arbitrary boundaries.

Density-based clustering can find clusters of arbitrary shape as long as the points are in proximity of each other. This characteristic is useful for the TS data that is under investigation. However, it also groups points from different transformations if the clusters are connected; for example, the points in Figure 40a will be grouped as a single cluster by the DBSCAN algorithm. But this problem can be resolved as long as clusters form straight lines using algorithms that specifically look for these attributes such as a segmentation algorithm, the detail of which is discussed later in Section 5.2.

DBSCAN is coupled with a multi-dimensional line segmentation algorithm to overcome this problem to create DBSCANmod. The segmentation algorithm is executed

on each of the clusters identified by DBSCAN.

Model-based clustering and Gaussian mixture models at first glance would seem ill-suited because the underlying assumption is that the data is normally distributed in some hyper-spherical shape. The partitioning clustering algorithm, namely K-means, suffered from the same problem. However, the shape of the spheroid can be stretched to fit a line or plane of points. To demonstrate this aspect, the same dataset is clustered using model-based clustering in Example 4.2.

Example 4.2: Model-based clustering on sample TS datasets

The `mclust` package from R is used for the model-based clustering [30, 31]. The same dataset from Example 4.1 was clustered using the `mclust` package, and the results are shown in Figure 41.

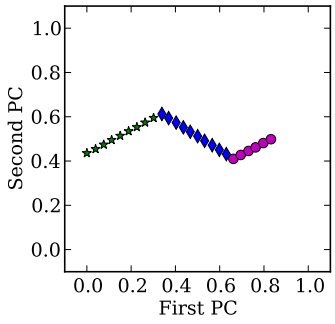


Figure 41: Model-based clustering on sample dataset

4.4.3 Data Features

The use of appropriate data for clustering is as important to choosing the appropriate clustering algorithm. In this case, the data can be the actual output data or attributes derived from the data. There are several issues when dealing with high-dimensional

data like TS datasets [60]. For example, as the number of dimensions grow, the combinations of axes that define the subspaces in which the data can be clustered grows. Also, two points become more difficult to discriminate using concepts like proximity distance in higher dimensions.

Kriegel et al. phrases the problems of high-dimensional data clustering concisely:

> In fact, when clustering high-dimensional data, we face two separate problems. The first problem is the search for the relevant subspaces and the second problem focuses on the detection of the final clusters. [60]

Recalling Chapter 3, PCA performed well in organizing the data, but there are other dimensions reduction techniques and other attributes of the data that can be extracted. In particular, because the TS are being segmented in the regression, the number of segments and the breakpoint locations can also be used as attributes for clustering. With several options available, the selection of features needs further investigation, and this will be addressed later in Chapter 5.

Research Question 9
What are the appropriate features to use for clustering TS datasets to create piecewise linear regressions with good fits?

4.5 Summary

Presented with the research question how to fit large TS datasets with nonlinear behavior (Research Question 5), various surrogate modeling methods were surveyed, and piecewise linear regression was identified as a potential solution. Breaking up

the data can lead to a better fit of the surrogate models because each model can be tailored to the smaller subspace [59, 40]. PLR can be applied to individual TS, and this was expanded to accommodate the rest of the design space by creating PLR Data Groups, which are groupings of TS that are similar enough to yield PLR with good fits. This approach solves Research Question 6. However, this requires an additional step of data clustering to the overall surrogate modeling process, and while decreasing the computational burden of the regression, it has its own computations which must be taken into account. Various clustering techniques are available in literature, and DBSCAN, DBSCANmod, and model-based clustering were chosen as candidates. Because the choice of data features affects the performance of the clustering algorithms, different combinations of clustering algorithms and data features will be examined later using experiments. This leaves Research Questions 8 and 9 unanswered, which are the ones that pertain to clustering and features, respectively. In the next chapter, TS segmentation and feature selection are explored to develop features to cluster the TS dataset.

CHAPTER V

TIME SEQUENCE SEGMENTATION AND DATA FEATURES

This chapter continues to develop Step 3 of the SMARTS methodology, which is responsible for creating a function that can represent TS data. The previous chapter introduced general clustering and piecewise regression methods. Furthermore, it was determined that clustering techniques are needed to create groups to which piecewise linear regressions (PLR) are fitted. In order to perform clustering on high-dimensional data, relevant features must be extracted from the time sequential (TS) data, and that is the focus of this chapter. More attention is given to TS segmentation methods and feature extraction methods to aid with the clustering and regression.

The chapter begins with an introduction to time sequential (TS) data with an overview of TS data mining literature. Then, TS segmentation methods are examined to find a suitable algorithm to break up the TS data. To answer Research Question 9, which is about determining the correct features for clustering, various feature extraction methods are introduced.

5.1 Time Sequence Data

Time sequence or time sequential data, or commonly known as time series, is an "ordered collection of elements", which has a time stamp and value [45]. TS data and graphs are ubiquitous because they are easy to make and understand. They

can be found in finances, economics, agriculture, meteorology, physics, medicine, and transportation to name a few fields of study. If measurements can be made consistently over time, then TS data can be collected. Examples of TS graphs are given in Figure 42. TS are studied to find hidden trends and patterns, and this insight can be used to make future predictions of how the series will continue.

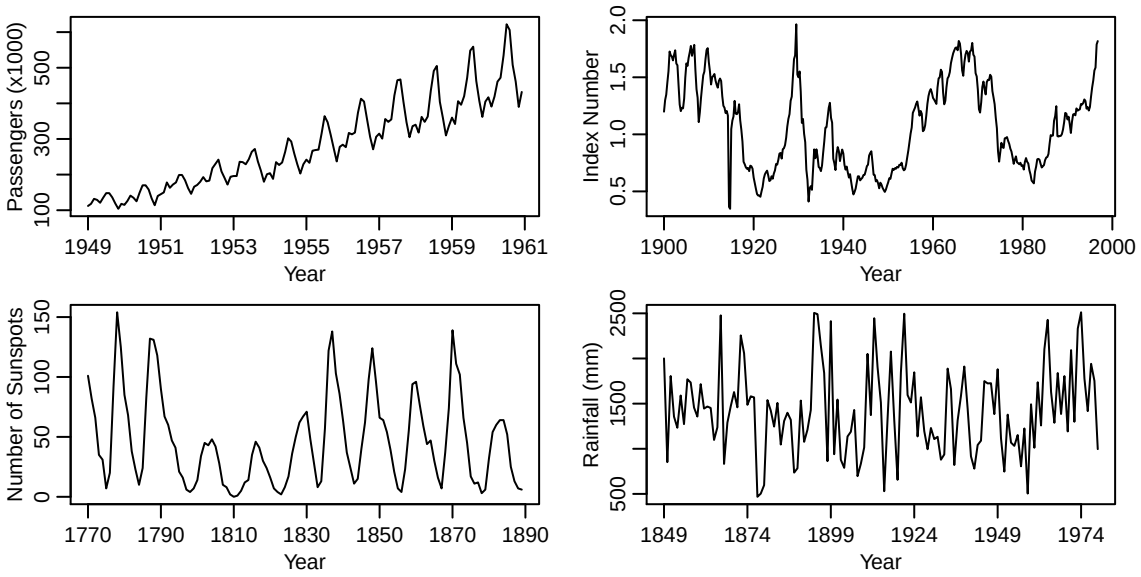


Figure 42: Time sequence data examples [48]. Top-left: Monthly totals of international airline passengers, Jan. 1949 - Dec. 1960. Top-right: Quarterly S&P Index, 1900 - 1996. Bottom-left: Woefler annual sunspot numbers, 1770 - 1889. Bottom-right: Annual rainfall at Fortaleza, Brazil, 1849 - 1979

TS data are also found in engineering and design. Manufacturing looks at the process flow and seeks to optimize the system by tracking performance metrics like utilization over time. Control theory deals with TS outputs using system identification, which is defined as the "art and science of building mathematical models of dynamic systems" [65]. Many simulation models used in engineering are time based such as discrete event simulation and agent-based simulation.

The purpose of studying TS data in engineering design is different from the mainstream of TS analysis and data mining. Under disciplines like finance and meteorology, empirical data cannot be recreated under similar conditions, and as much as there is a desire to understand the underlying process, the purpose is to predict the future and act upon it. In engineering design, the main goal is to understand the underlying process so that it can be modified and improved.

Figure 43 displays the TS data of a simulation run on a user interface for decision making. The input parameters can be controlled using slide bars on the left, and the corresponding time sequence is displayed on the right. The data for these types of applications are generated through computer simulations, and the dashboards are effective tools to navigate high-dimensional problems [98]. Generating all the relevant data points can take a long time, and once they are generated, storage becomes an issue. Typically, these types of dashboards use surrogate models, which compromises accuracy for speed and portability. Unfortunately, it is difficult to fit surrogate models which retain the sharp features such as transitions from one state to another because surrogate models are also smoothing techniques.

There are several qualities that are sought when creating TS surrogate models for decision making tool applications. The model needs good fit characteristics. The representation should have limited overfitting and smoothing, especially in regions with sharp transitions. If the regression is piecewise, there should not be an excessive number of partitions because this is also a case of overfitting. If there are upper and lower bounds such as a metric measured in percentages, the modeling should not exceed these limits.

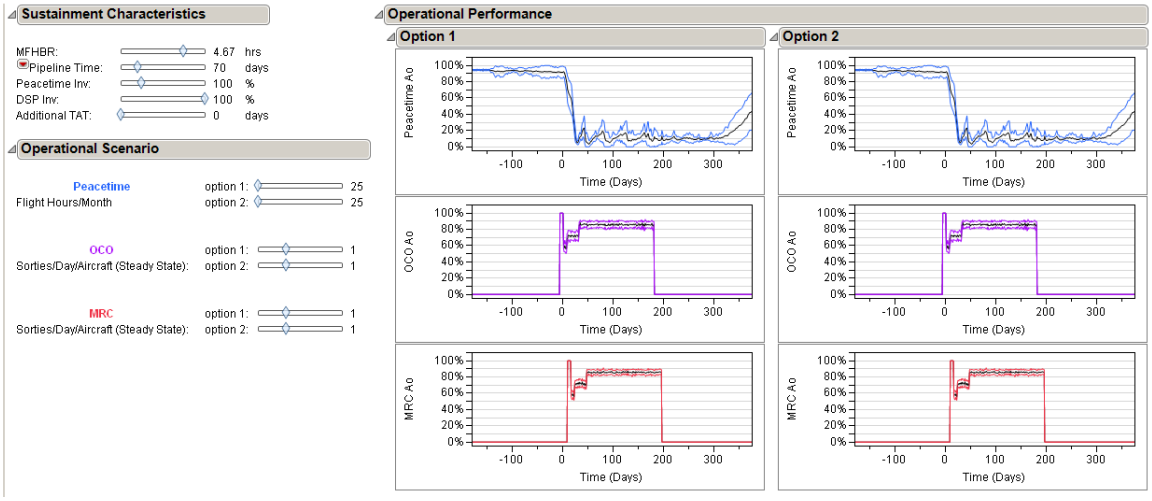


Figure 43: Decision support tool with TS output

In order to create PLR models, it is important to split time sequences into segments that are straight, and the breakpoint locations should be consistent across similar TS shapes. One field that focuses on TS segmentation is the time series data mining community. A review of this field is given below in Section 5.1.1 and its relevant methods in Section 5.2.

5.1.1 Time Series Data Mining

The research in time series data mining emerged to address the problems of managing, analyzing and extracting insights from the rapidly growing volume of TS data. There are seven major tasks that are of interest to the time series data mining community: indexing, clustering, classification, prediction, summarization, anomaly detection and segmentation [89]. Many of these tasks use some form of similarity measures to compare the time series and generalize the data. Because data mining typically involves a large amount of data, there has also been research into reducing the volume of data and its dimensionality so that the tasks can be accomplished within a reasonable

amount of time and resources and without significant loss in accuracy. The concepts from time series data mining are summarized to find applicable methods for partitioning.

Major Tasks of Time Series Data Mining

A quick description of the tasks in time series data mining will be given in order to highlight relevant tasks to surrogate modeling.

Indexing or Query by Content is a matching task to find the most similar time series in the database to one that is supplied.

Clustering looks for natural groupings within the database based off of a similarity measure.

Classification is an assignment problem where a time series not included in the database is assigned to the closest predefined clusters.

Prediction or Forecasting involves looking for precursors of an event of interest (such as natural disasters) based on past time series data.

Summarization provides an overall view of the data most often through text or graphical visualization.

Anomaly Detection looks for the "odd man out" or abnormal behavior in the time series, such as an irregular heartbeat.

Segmentation approximates the time series with a linear combination of simple functions to reduce the amount of data required to represent it.

One of the steps to create the surrogate models is to split the design space into smaller groups, which is a clustering task. The surrogate models will have better fits if the shapes are similar. Also, to make the fits better, the surrogate models can fit TS segments, which requires a segmentation step. Regression allows the estimation of regions between the data points, which is similar to the prediction task but the purpose in this case is not the future. Finally, the goal of creating the surrogate model is inspired by the desire to communicate data to decision makers through visualizations, which is similar to the summarization task.

Similarity Measures

Similarity measures are metrics that can be used to compare the similarity of two time sequences. The simplest and most popular method is to measure the Euclidean distance between the points of each TS. The sum of the distances can be used as a metric to compare its similarity with other pairs. Figure 44 illustrates this concept. It is possible that the two TS are vertically offset or scaled differently, and in these situations, the TS are normalized before the comparison.

There are other measures that try to overcome certain issues such as time shifting. Dynamic time warping deals with time series data that have similar shapes but do not line up along the x-axis. The method applies a non-linear transform or accelerates some of the time segments while slows down others so that the two TS line up better. Another similarity measure looks for the longest common subsequence. There are also measures based on probabilistic models.

In searching for a way to partition the design space, grouping similar TS using metrics that compare every data point can become time prohibitive for large datasets.

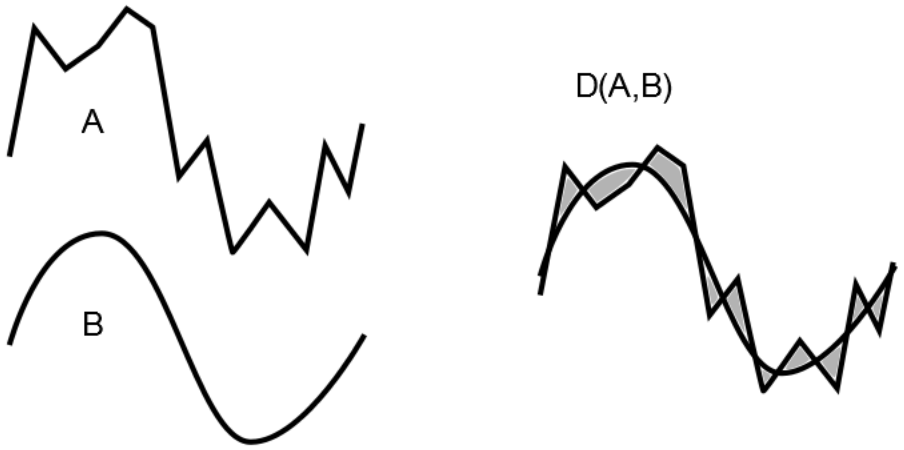


Figure 44: Illustration of Euclidean distance metric for time series comparison (Adapted from [89])

There are some tricks to speed up the process, but these are task specific. For example, there is the early abandoning technique that is used in the indexing task where the comparison is stopped if the cumulative deviation exceeds a certain threshold. In the next section, TS representation methods are evaluated in search of ways to reduce the data volume.

5.1.2 Time Series Representation

Data mining of TS databases typically involves large volumes of data, and the bulk of the computational effort is consumed by the data transactions between the hard disk and main working memory. In order to reduce the access to the hard disk, one of the standard procedures is to create a reduced set of the data from which a candidate solution is found, and this point is validated with the real data. This approach follows the same principles as surrogate-based optimization, which is a common technique in engineering design. A surrogate model is used to find a candidate solution of a design problem because of the inconveniences of working with the actual model.

The class of methods of reducing the dataset is called TS approximation or representation. Some of the more popular methods include discrete Fourier transformation (DFT), discrete wavelet transformation (DWT), singular value decomposition (SVD), piecewise linear approximation (PLA), and piecewise aggregate approximation (PAA). Some of these methods were covered in Section 3.1. Fundamentally, all of these methods are linear combinations or piecewise combinations (which is a linear combination if the irrelevant portions are reduced to zero) of different basis functions of sinusoidal waves, square wavelets, localized waves, lines and square waves, respectively. The main goal of data reduction in time series data mining is to improve the time efficiency of tasks such as pattern matching and querying so the visual aspects are not as important. When the data needs to be shown, the actual data can be plotted.

In representing TS data for design space exploration and visual applications, PLA is an appropriate choice. Methods that use waves to approximate the data, such as DFT and SVD, run the risk of overfitting or smoothing out the corners because the basis functions are smooth waves. Methods that use square waves and wavelets, such as DFT with Haar wavelets and PAA, are poor at visually representing slopes because they approximate it as a step function. PLA can capture sharp transitions as well as sloping lines, and curves can be represented using several segments, making it an appealing representation method for visualization.

Keogh et al. [58] categorizes PLR algorithms into the following three:

Sliding Window Segments are created sequentially by approximating a line to a

subsequence of data points. Starting from one end, the subsequence is expanded until the fit of the line exceeds a threshold. Then a new line is created starting from the next point, and this is repeated until finished.

Top-down The TS is split recursively until some stopping criteria, such as total number of segments, is satisfied.

Bottom-up The TS is built up from the simplest representation (such as a line segment of two adjacent points). The segments are merged into larger segments until some stopping criteria is satisfied.

Among these three types of algorithms, the bottom-up and top-down approaches outperform the sliding window when compared in batch or offline mode, as opposed to online where the TS data is streaming [56]. The sliding window is the only one that can be used online. Bottom-up is also better in most segmentation tasks compared to the top-down approach, and in cases where top-down is better, the difference is small.

When using PLR methods, the individual lines or segments can be approximated using the following two methods:

Linear interpolation A line is drawn from the starting to the end point.

Linear regression A line is fit to the subset of points.

Linear interpolation may not be the best fit line to each segment, but it preserves continuity of the line and is simple to implement and faster to run. When using linear regression, there is no guarantee that each segment will connect to form a continuous

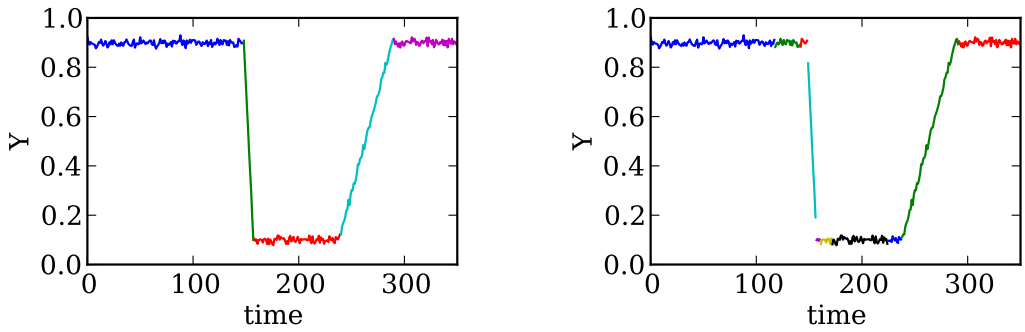
line. The resulting model can be used as the surrogate model for the deterministic case. The PLR can also be modified to use different regressions for the segments, but determining how to segment the TS becomes a trickier task. Different TS regression techniques are covered in the next section.

5.2 TS Segmentation

Several TS representation methods were introduced in the previous section, and PLR was determined to be the suitable choice for the intended application of this research, which is visual representation for engineering decision making. To create PLR, the bottom-up and top-down algorithms were found in literature to be effective in segmenting TS data. Each of these algorithms are described in further detail in this section and demonstrated using simple examples in order to answer the following research question:

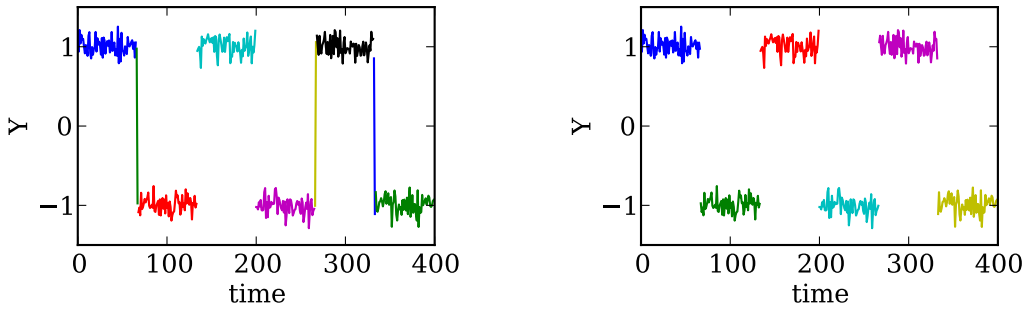
Research Question 10
What TS segmentation algorithm is suitable for splitting TS data to help create PLR Data Groups?

The PLR Data Groups (PLRDG) are groups of TS that will lead to PLR with good fit characteristics. Because each algorithm has certain weakness that are drawbacks for TS segmentation to create PLRDG, a hybrid algorithm which uses both algorithms in series is proposed. The bottom-up and top-down algorithms were coded in Python based on pseudocode by Keogh et al. [58]. Stopping conditions for these algorithms are also addressed.



(a) Bottom-Up (b) Top-Down

Figure 45: TS segmentation using a bottom-up and top-down algorithms



(a) Bottom-Up (b) Top-Down

Figure 46: Segmentation algorithm dealing with discontinuity in the TS data

5.2.1 Bottom-Up Algorithm

A bottom-up algorithm is used to determine the number of segments. In Figure 45a, a sample TS data set is segmented using a bottom-up algorithm. Bottom-up algorithm creates pairs of neighboring points (creating $n/2$ segments) as the first step of the algorithm, and this can cause a problem when there are discontinuities in the data. Depending on where the discontinuity occurs, this discontinuity can be captured as a near-vertical segment as shown in Figure 46a.

Algorithm 5 Bottom-Up TS Segmentation Algorithm (Adapted from [58])

```
Given TS data Y and stopping condition maxError or minSegments
for i = 1:length(Y)/2 {Pair Up Points}
    Segments[i] = Y[2i : 2i + 1]
end
for i = 1:length(Segments)−1 {Find cost of merging segments}
    cost[i] = fnCalculateCost(Segments[i], Segments[i+1])
end
while stoppingCondition == false
    index = argmin(cost) {Find lowest cost merge}
    Segments[index] = fnMerge(Segments[index], Segments[index+1]) {Merge Pair}
    fnDelete(Segments[index+1]) {Update records}
    cost[index] = fnCalculateCost(Segments[i], Segments[i+1])
    cost[index-1] = fnCalculateCost(Segments[i-1], Segments[i])
    if use(maxError) {Check Stopping Condition}
        stoppingCondition = (min(cost) < maxError)
    else
        stoppingCondition = (length(Segments) < minSegments)
    end
end
```

5.2.2 Top-Down Algorithm

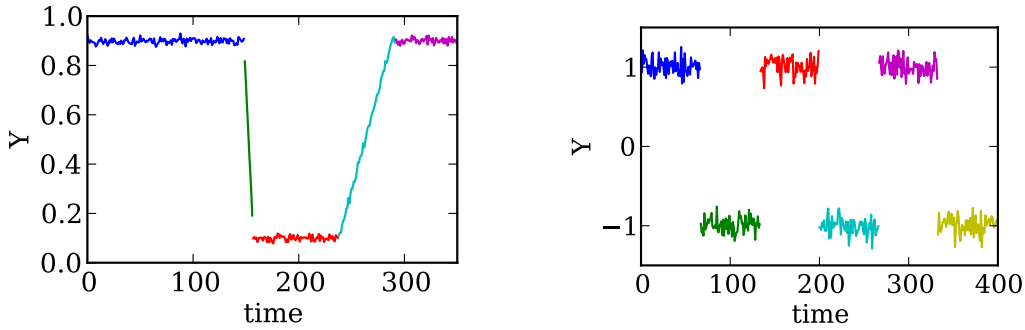
The top-down algorithm evaluates the benefit of splitting a TS data at every location, selects the best one, splits the data into two sets, and continues performing this operation on each of the new sets until some stopping condition is met. Figure 45b uses the same data as Figure 45a but uses a top-down algorithm to segment the data. As seen in the figure, the top-down algorithm does not necessarily capture the correct breakpoints. However, it does not have the pairing problem like in the bottom-up algorithm as show in Figure 46b.

5.2.3 Hybrid Algorithm

In order to overcome the drawbacks of each algorithm, both of the algorithms can be used in succession. The top-down algorithm is used first to create more segments than

Algorithm 6 Top-Down TS Segmentation Algorithm (Adapted from [58])

```
Given TS data Y and stopping condition maxError
for i = 2:length(Y)-2 {Find best place to split}
    improvementToSplit = fnImprovementSplittingHere(Y,i)
    improvementToSplit < bestSoFar
        breakpoint = i
        bestSoFar = improvementToSplit
    end
end
{Recursively Split Left and Right Segment if necessary}
if fnCalculateError(Y[1:breakpoint]) > maxError
    Segments = fnTopDownAlgorithm(Y[1:breakpoing])
end
if fnCalculateError(Y[breakpoint+1:length(Y)])
    Segments = fnTopDownAlgorithm(Y[breakpoint+1:length(Y)], maxError)
end
```



(a) Sample TS data

(b) Square Waves with Discontinuity

Figure 47: TS segmentation using the hybrid algorithm

desired, then the bottom-up algorithm is used to reconnect some of the segments to meet the threshold or desired numbers of segments. Figure 47 show how the hybrid algorithm performs on the two sample datasets. It avoids oversegmenting the valley-shaped TS, and it also does not have the pairing problem like with the bottom-up algorithm.

Algorithm 7 Hybrid TS Segmentation Algorithm

```
1: Given TS data Y and stopping condition maxError or maxSegments
2: if use(maxError)
3:     Segments = fnTopDownAlgorithm(Y, maxError/2)
4:     Segments = fnBottomUpAlgorithm(Segments, maxError)
5: else
6:     sigma = fnStandardDeviation(Y)
7:     while length(Segments) < maxSegments ×2
8:         Segments = fnTopDownAlgorithm(Y, sigma)
9:         sigma = sigma/2
10:    end
11:    Segments = fnBottomUpAlgorithm(Segments, maxSegments)
12: end
```

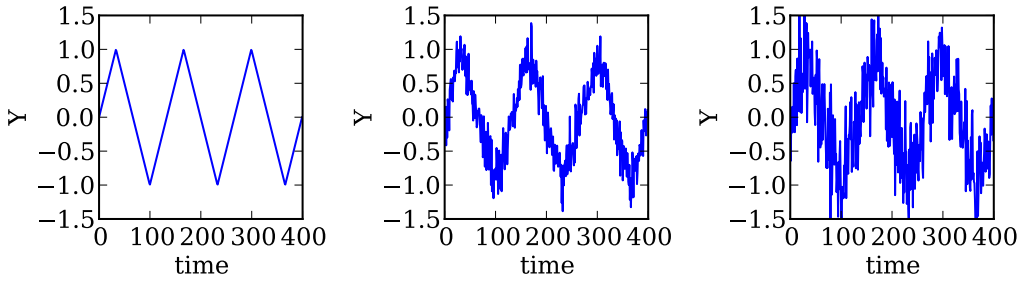
5.2.4 Stopping conditions

In order to create PLRs, finding the right number of segments in a consistent manner is important. The TS segmentation algorithms use a threshold variable to determine when to stop. When applying these algorithms to a TS dataset with varied characteristics, such as varying amplitude and noise, a single setting for the threshold can cause underfitting or overfitting.

There are two main ways to define the stopping conditions for TS segmentation algorithms. One is to predefine how many segments are to be created, and the other is to set a threshold for the goodness of fit, such as standard deviation. For the purposes of defining groups based on the segmentation the TS, the first approach is not compatible. The second approach is the logical choice; however, it also poses problems because the appropriate threshold setting could vary throughout the design space. The problem of using a static threshold setting is illustrated in Example 5.1.

Example 5.1: Static Threshold Setting for Segmentation

To illustrate why using a static threshold setting for TS segmentation algorithms in design space exploration applications, an example using a triangle wave is presented. The hybrid segmentation algorithm was applied to a triangle wave with three periods, amplitude of one, and varying levels of noise (σ) that is normally distributed. For a triangle wave of three periods, the correct number of segments is seven. The data is shown in Figure 48.



(a) $\sigma = 0.0$ (b) $\sigma = 0.2$ (c) $\sigma = 0.4$

Figure 48: Triangle waves with varying levels of noise

The hybrid segmentation algorithm was applied to the triangle wave dataset at three threshold values (0.2, 0.4, 0.6). The threshold values in the algorithm represents the maximum level of standard deviation that each segment can have. The results are plotted in Figure 49.

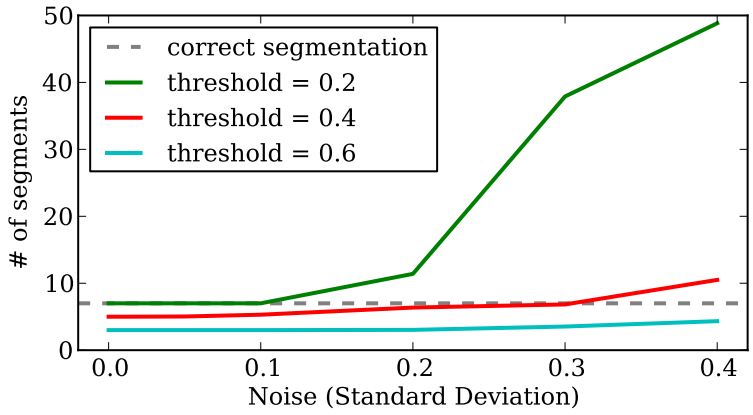


Figure 49: Number of segments vs. the artificial noise in the data

For a low threshold value, the segmentation is correct at low noise levels, but the algorithm oversegments the data with more noise in the data. With higher threshold values, the segmentation algorithm will undersegment the data. Based on the results, it can be seen that the correct threshold standard deviation varies with the standard deviation of noise in the data, and a static threshold setting for segmentation algorithms is not a robust choice.

If the number of segments cannot be used and a static threshold is not sufficient, then the threshold value needs to be calculated based on the data.

Research Question 11
How can a robust threshold setting be derived from the data for TS with noise?

5.2.5 Estimating Noise in a Time Sequential Data

As was shown in Example 5.1, an automatic method to find the threshold value for TS segmentation algorithms is needed. In order to determine the setting, it is first necessary to estimate the noise in the data. One way to do this is to take the difference

between the original TS and one that has been smoothed by using techniques like moving average. The noise can then be estimated by taking the standard deviation of the residuals.

There are a variety of techniques to smooth or denoise the data. One approach is to use a set of repeated simulation runs; by executing the simulation model with the same initial conditions, the effects of the stochasticity in the model can be quantified. Another set of methods use a "moving window", which is a technique to select a subset of consecutive points, and a function, such as a mean, is used to calculate a representative value for that window location. Regression techniques can also be used as a smoothing function, and there are also algorithms that are specifically designed for data smoothing like the Kalman smoother and Savitsky-Golay algorithm. These smoothing methods will now be presented in further detail.

Smoothing Using Repetitions

If the simulation model is stochastic like the FAMOS model, then repetitions are useful to characterize the distribution of outputs that the model generates. If the data is noisy, then the repetitions can be used as a smoothing technique. The average can be taken at each time step to create a mean TS for a set of repetitions. With this approach, the standard deviation of the noise can be reduced at most by $\sqrt{\frac{1}{n}}$, where n is the number of repetitions, and this is under the assumption that the noise is normally distributed. This is because the mean of n random variables (x), which are normally distributed, results in a new random variable α, which has a mean of $\frac{1}{n}\sum x$ and a variance of $\sigma_\alpha^2 = \frac{1}{n}\sum \sigma_x^2$. If 30 repetitions are used, this will result in a maximum of over 80% reduction in noise. In a similar fashion, the median of the

values at each time step can be taken. The benefit of the median is that it is not influenced by outliers as much as the mean.

Smoothing Using Moving Windows

If there are no repetitions or the number of repetitions is low, taking the mean or median at each time step may not be effective. A **moving windows** approach attempts to solve this by incorporating adjacent values to calculate the mean or median. The number of points to use to calculate the mean or median depends on the window size. For example, if the window size is three, then points before and after time step t_i is used to calculate the value at t_i, then the window is advanced to t_{i+1} to calculate the next time step. Because this window moves across the length of the data, it is called the moving windows. The moving average uses an uniform weighting to calculate the value, but different weighting schemes can be used such as a bell curve or Gaussian shaped weighting and triangular weighting schemes.

Smoothing Using Regression

Regression techniques can also be used as a mechanism to smooth the data. They are simplified representation of the data, and they are suited to capture the overall trend of the data while suppressing the noise. Many of the methods have already been introduced in Section 4.1.1.

Other Smoothing Techniques

There are other smoothing techniques that are popular in literature. Kalman smoothing techniques use Kalman filtering in the forward pass and use a backward pass to estimate the smoothed values. Another popular one is the Savitsky-Golay smoothing filter, which performs a polynomial regression in a moving window fashion to calculate

the smoothed value.

Research Question 12
Which smoothing technique is appropriate for smoothing TS data to estimate the noise in the data?

The descriptions of the method do not provide any guidance towards favoring one algorithm over another. In order to determine the appropriate smoothing technique for TS data, an experiment is conducted using different smoothing techniques and by varying different parameters such as TS shape, number of repetitions and amount of noise. The details of this experiment are presented in Section 8.2.2.

5.3 Feature Extraction and Selection

Feature extraction and selection is an important step in mining high-dimensional data [42]. In the case of TS data where each column is not a unique variable, extraction of features from the data becomes important. Feature extraction is a subset of dimensionality reduction, and it is also known as feature transform, generation or construction [119]. There are various possible features to use for the clustering of TS data, and the depending on the application the optimal set of features can change [63]. An experiment is conducted to determine which features were the best to use for O&S simulation data (answers Research Question 9).

In looking for appropriate features to cluster the data, several popular methods from feature extraction techniques and several features based on the TS data were selected as candidates. Based on the literature surveyed, the following features were selected to be tested in the Clustering Experiment (Section 8.2.3).

Mean average of output values of each TS.

Standard Deviation standard deviation of each TS.

Principal Component Analysis (PCA) creates orthogonal axes where subsequent axes are aligned to maximize the remaining variance in the data.

Sorted PCA the order number after the data is sorted instead of the principal component (PC) values. The sorting is based on the first PC.

Isometric Feature Mapping (Isomap) a manifold learning technique that incorporates the geodesic distances between data points.

Locally Linear Embedding (LLE) a manifold learning technique that preserves local distance structures.

Discrete Fourier Transform (DFT) a discrete transform technique that represents the time sequence in the frequency domain.

Number of Segments number of segments needed to represent the TS using piecewise linear segments.

PCA of breakpoints The first PC of the breakpoint locations of each TS is used.

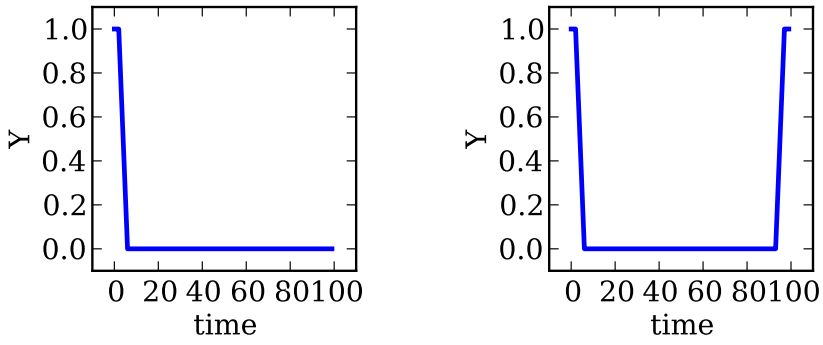
Base2 of breakpoints By representing the breakpoint locations as 1s on a vector of 0s that is as long as the time sequence, a unique value of the breakpoint locations can be created in binary. This binary value is averaged with a binary that is in the reverse order to create this feature.

In order to incorporate the breakpoint locations of the TS, two methods were created to project the breakpoint locations onto a one-dimensional axis. The first method uses PCA on the breakpoint locations. The breakpoint locations of each TS is represented as a vector of indexes. Because the number of segments can vary, shorter vectors are padded with zeros to match the length of the longest one. Finally, PCA is used on this matrix, and the first principal component is retained as the feature.

The second method represents breakpoint locations in binary. A binary string of 0s that is as long as the time sequence is created, and the breakpoints are marked on this string as a 1. The string is then converted into a decimal value, and the logarithm to the base 2 is taken. If the TS has many time steps, then breakpoints on the smaller digits are dwarfed by the larger digits so the binary string is read in both directions, and the two values are averaged to make the feature. An example is presented below to illustrate how this feature is calculated.

Example 5.2: Binary String Representation of Breakpoints

To demonstrate the binary string representation, two different TS shown in Figure 50 are used. Each TS is 100 time steps long. The first TS (`ts1`) has breakpoints at $t = \{3, 6\}$, and the second TS (`ts2`) has breakpoints at $t = \{3, 6, 95, 98\}$. In binary string form, the breakpoints (BP) would be represented as:



(a) `ts1` (b) `ts2`

Figure 50: Example time sequences

$$BP_{ts1} = 00100100...0000000$$

$$BP_{ts2} = 00100100...0100100$$

This binary string is read from left to right and converted to decimal notation. Because the value is extremely large ($x = 1.78^{29}$), the logarithm to base 2 is taken. The values then are $x_{ts1} \approx x_{ts2} = 97.2$, and the difference between the two are minuscule. If the string is read in the other direction, the values are $x_{ts1} = 5.17$ and $x_{ts2} = 97.17$. To calculate the feature value, the two values are normalized, and the weighted sum is returned.

5.3.1 Choosing a Subset of Principal Components

In applying PCA to a dataset, there are three main questions that arise in its applications [123]. Is there any value in using PCA (applicability)? How many should be kept (selection)? What do the selected principal components mean (interpretability)?

For the purposes of this research, the first two questions are relevant to creating regression models, and the need for PCA has already been expressed. While the third is interesting in understanding the simulation data, this is omitted as it is beyond the scope. Interested readers should refer to the following references [52, 53, 81].

Research Question 13
How many principal components should be kept after principal component analysis (PCA) is applied to a dataset?

In choosing the right number of principal components to represent the data, literature is divided on how to choose the correct number of PCs. Basilevsky dismisses rules of thumbs as statistically flawed after a presenting various proper ways of selecting PCs [2]. On the other hand, Jolliffe argues that these statistically sound approaches "offer little advantage over simpler rules" [53]. Among the methods that have been proposed, parallel analysis (PA) is one of the better performing method [82], and it has also been found to be effective with TS data [47, 61] so it will be investigated in further detail.

Parallel Analysis

The parallel analysis (PA) method compares the eigenvalues of the data and eigenvalues calculated from a set of random datasets [82]. Each random dataset is the same size and shape as the output data, and the set is comprised of 1000 of these random datasets. For each of these, the eigenvalues of its correlation matrix are calculated and tabulated, and confidence intervals are derived from the set of eigenvalues.

1. For an output matrix $(n \times p)$, create a matrix with random values $N(0, 1)$ of

the same size ($n \times p$)

2. Calculate the eigenvalues of the correlation matrix ($p \times p$) and record into a temporary matrix

3. Repeat 1 and 2 for 1000 times

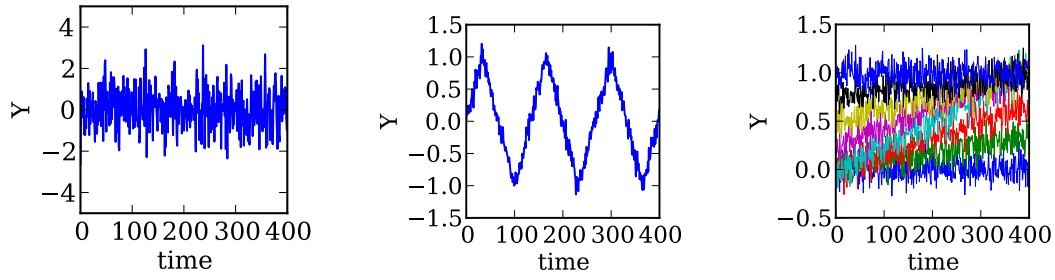
4. Calculate the test statistic for each axis and assess the significance of each eigenvalue (95% $\alpha = 0.05$ one-sided test)

Although Peres et al. recommends the use of Bartlett's test to determine whether or not PCA is warranted, Bartlett's test cannot be performed for datasets that have less samples n than features p. The test uses the determinant of the correlation matrix, which is zero for matrices that are rank deficient. When describing p features with n vectors, only n features can be expressed uniquely, and the rest will be linear combinations of the first n. The PA can be used without the Bartlett's test, and if a set of TS has the same shape, it can return that no PC are needed.

Example 5.3: Parallel Analysis

The parallel analysis (PA) method is used to show how it can determine the number of principal components (PC) to keep from a set of time series data. Three datasets are used to show that it can determine whether PC are needed, and if it is, how many should be kept. Each dataset has $n = 400$ repetitions, and each TS is 400 time steps (p) long so the dataset is 400×400. The first set is a set of flat lines with normally distributed noise with standard deviation of $N(0, 1)$, and this will be referred to as **FLAT**. The second is a set of triangle waves

(**TRIANGLE**) with noise of $N(0, 0.1)$. The third is a set of lines (**LINES**) is a similar set of data as shown in Figure 38a where the start point and end point of the line are varied, but this time a larger set of TS is used. The **LINES** dataset also has a noise of $N(0, 0.1)$. The datasets are illustrated in Figure 51.



(a) **FLAT** (b) **TRIANGLE** (c) **LINES**

Figure 51: Parallel Analysis Example Charts

Step 1: Each dataset is standardized so that it has a mean of ($\mu = 0$) and standard deviation of ($\sigma = 1$). The eigenvalues of the datasets are also calculated.

Step 2: A $n \times p$ matrix with random values is generated. Each value is normally distributed with standard deviation of 1, $N(0, 1)$.

Step 3: A $p \times p$ correlation matrix is calculated.

Step 4: The eigenvalues of the correlation matrix is calculated, and the values are recorded.

Step 5: Steps 2 through 4 are repeated 1000 times.

Step 6: The mean and standard deviation is calculated for each eigenvalue component.

Step 7: A one-sided 95% significance test is performed to see if the eigenvalues calculated from the dataset are significantly higher than the ones derived from the random matrix.

This approach assumes that the eigenvalues of the correlation matrix is normally distributed, and this can be verified by using a Shapiro-Wilk test for normality [90]. The results of the first five eigenvalues are presented in Table 5 and seen in Figure 52.

Table 5: Comparison of eigenvalues from the parallel analysis example

	λ_1	λ_2	λ_3	λ_4	λ_5
FLAT	3.8771	3.8455	3.7924	3.7456	3.6692
TRIANGLE	3.8694	3.8284	3.8071	3.6686	3.5998
LINES	333.6751	26.9739	0.3905	0.3802	0.3712
PA baseline μ	3.9284	3.8341	3.7632	3.7019	3.6483
PA baseline σ	0.0596	0.0463	0.0394	0.0353	0.0333

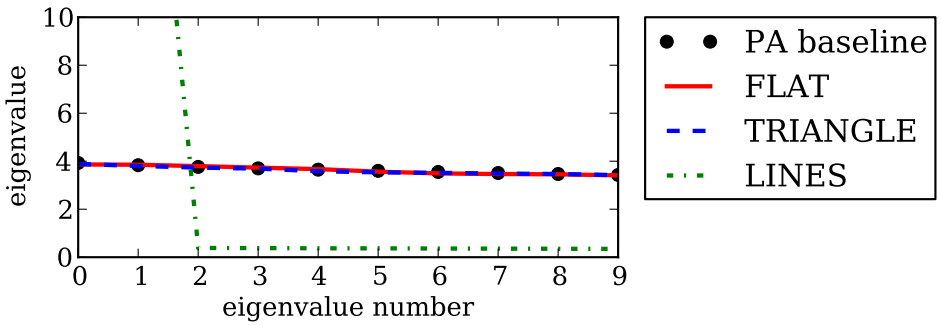


Figure 52: Comparison of eigenvalues

Figure 52 shows that the first and second PC of the **LINES** dataset are significant because both of them are clearly greater than the baseline while the rest are below. The eigenvalues for the **FLAT** and **TRIANGLE** datasets are very close to the baseline, and they are difficult to distinguish in the figure. Based on values shown in Table 5, the eigenvalues are within one standard deviation of the baseline mean, and thus, the two datasets do not have significant PCs. This statistically shows that each row of TS in the datasets are essentially the same aside from the noise.

The example demonstrated the ability of the PA method to determine the number of significant PC for a limited set of TS data. In order to test the broader effectiveness of the PA method, a more extensive experiment is conducted, and the details are presented in Section 8.2.1.

5.4 Summary

In this chapter, TS segmentation and feature extraction methods were presented. The bottom-up and top-down segmentation algorithms were selected for further investigation, and in the end, the hybrid algorithm, which uses both bottom-up and top-down, was selected, thus answering Research Question 10.

In order to segment TS data, the segmentation algorithms require a threshold setting, but it was shown that a static threshold is not robust to different degrees of noise in the data. This problem is captured by Research Question 11, and one approach to calculate such a threshold setting is to take the standard deviation of the difference between the original data and the smoothed version of the data. There are various smoothing techniques that were presented, and to determine which one to choose (Research Question 12), an experiment is conducted. The details of this experiment are presented in Section 8.2.2.

To cluster the data for piecewise regressions, features from the data need to be extracted, and so different features were examined as potential candidates. Because no definitive guidance was found in literature, another experiment is conducted to select the appropriate set of features (Research Question 9). The details are presented in Section 8.2.3.

One of the methods for feature extraction is PCA, and this method will return as many features or principal components (PC) as there are number of variables or number of time steps. However, most of these PC are irrelevant. In order to determine the appropriate number of components to keep, the parallel analysis (PA) method

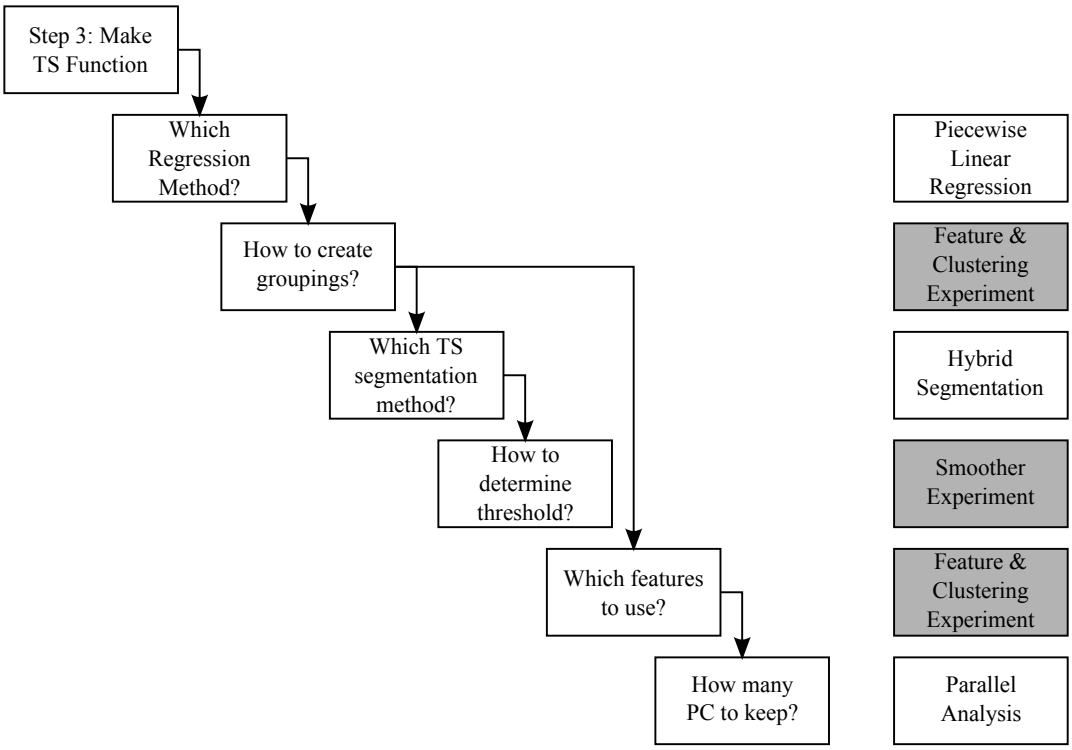


Figure 53: Summary of Step 3 and Chapters 4 and 5

was chosen, and an experiment is conducted to demonstrate its effectiveness on TS data. The details are further discussed in Section 8.2.1.

Figure 53 summarizes the research questions and the solutions found in Chapters 4 and 5. The boxes on the left show the questions, and the boxes on the right are the algorithms and methods selected. The specific methods are not yet determined, and these are colored in gray. The conducted experiments will be discussed in Chapter 8 to choose the appropriate methods. In the next chapter, Step 2 of the SMARTS methodology will be discussed, and the entire methodology will be reviewed in whole.

CHAPTER VI

SMARTS METHODOLOGY

This chapter will summarize what has been presented and develop the Surrogate Modeling and Regression for Time Sequences (SMARTS) methodology. The previous chapters have covered many different methods and algorithms to address Step 1 and 3 of the SMARTS methodology, and these will be reviewed. Then, Step 2 will be discussed before the entire SMARTS methodology is described as a whole.

6.1 Method Morphological Matrix

Various fields and subfields of study are explored throughout this dissertation including data mining, surrogate modeling, dimensional reduction, time series data mining, time series segmentation, and clustering. Table 6 organizes the various methods that were discussed in a concise format. The grayed out options will not be explored further because they are not applicable to the problem or they were determined to be inferior to the other methods for the intended application.

6.1.1 Sampling the Design Space

For most applications, the offline sampling using a design of experiments (DoE) is simpler to implement compared to online or adaptive sampling. It can be performed separately from the surrogate modeling process. The simulation runs can be executed in parallel to speed up the process. This approach will most likely require more points

Table 6: Morphological matrix of different methods and algorithms.

Sampling the Design Space			
Sampling Options	Design of Experiments	Adaptive Sampling	
Dimensionality and Data Reduction Methods			
Dim. Reduction	PCA	Multidimensional Scaling	Isomap
	Locally Lin. Embedding		
Data Reduction	Discrete Fourier Trans.	Discrete Wavelet Trans.	
Clustering Methods			
Class of Method	Hierarchical	Partitioning	Others
Hierarchical	Divisive	Agglomerative	
Partitioning	K-means	K-medoids	Model-based
Density based	Density-based		
TS Methods			
TS Segmentation	Sliding Window	Top-Down	Bottom-Up
	Hybrid Segmentation		
TS Representation	Discrete Fourier Trans.	Discrete Wavelet Trans.	Singular Value Decomp.
	Piecewise Linear Approx.	Piecewise Aggregate Approx.	
Smoothing methods	Average	Median	Moving Average
	Moving Median	Moving Triangle	Moving Gaussian
	MARS	Support Vector Reg.	Kalman Smoothing
	Savitsky-Golay		
Surrogate Modeling			
Models	Linear Regression	Neural Network	MARS
	Gaussian Process	Regression Trees	Support Vector Regression
	Hidden Markov Model	Spline Functions	Forest and Additive Regressions

than adaptive sampling because it will sample portions of the design space that could have been estimated through interpolation. Offline sampling is a better choice when the simulation runs are fast, and large amounts of data can be generated quickly.

Adaptive sampling can have synergistic benefits to the piecewise modeling approach, but it is more complex to implement and computationally more expensive. The sampling algorithm can be written to focus more points on the boundaries of the partition subspaces so that they are better defined. In this way, adaptive sampling will provide more useful information about the design space than a DoE for the same number of points. To implement adaptive sampling with the partitioning and surrogate modeling, these steps have to be integrated. The modeling will drive the partitioning by informing how well the models can fit the subsets. The partitioning will provide the information about the partition boundaries to drive the adaptive sampling. This option becomes favorable if the simulation runs take much longer than the partitioning and surrogate modeling.

For this dissertation, the adaptive sampling option will not be explored because the focus is on the regression method and not adaptive sampling. Also, the simulation runs using FAMOS and MRFAMOS are relatively fast.

6.1.2 Discussion of Methods

Dimensionality reduction methods were covered in Chapter 3. Principal component analysis (PCA) and two manifold learning techniques, Isomap and locally linear embedding (LLE), are selected for further investigation.

Clustering methods were covered in Section 4.4.1. The two major categories are

hierarchical and partitioning methods. Hierarchical methods and objective-based partitioning methods were found to be inappropriate for TS data for design space exploration. Density-based and model-based clustering were determined to be better options instead because the data points are arranged in lines.

As discussed in Section 5.2, there are three main strategies to determine the segments in a TS: sliding window, top-down, and bottom-up. The top-down and bottom-up can be used serially to form a hybrid method. For offline data, sliding windows approach is consistently lower performing than the others [56]. The hybrid method was chosen as the best option because it overcomes the shortcomings of the top-down and bottom-up by combining the two together.

There are five popular methods for TS representation, which are discrete Fourier transformation, discrete wavelet transformation, singular value decomposition, piecewise linear approximation and piecewise aggregate approximation. Based on the observations made about the TS data from simulations, piecewise linear approximations was chosen to be the best choice.

There are many different surrogate modeling methods that are available, and linear regression and neural networks (NN) will be used as benchmark methods.

6.2 Step 2 of the SMARTS methodology

The second step of the SMARTS methodology involves mapping the input X domain to the intermediate Z domain. This step is simplest of the three steps because solutions readily exist in the literature, and thus, it was left until now to discuss the details. The Z domain is nonlinear with respect to X, and the dimensionality is much

lower that the output Y domain. The mapping is a nonlinear multivariate regression, and the research question that results from this is the following:

Research Question 14
What regression method is appropriate for capturing nonlinear variations in the design space?

Some of the features that are sought from the regression method for Step 2 include the ability to handle nonlinear data, the availability of tools to implement the regression, and the robustness and simple implementation of the method. Various regression methods that are listed in Table 6 have been used for design space regression for aerospace design applications [7, 55, 70]. Among the available methods, NN is one of the most versatile and commonly used regression methods because of its ability to handle nonlinear data, and NN is chosen to fulfill Step 2 of the SMARTS methodology.

6.3 Development of the SMARTS methodology

The Surrogate Modeling and Regression for Time Sequences (SMARTS) methodology is developed in this section. The demonstration of the PA method and the selection of the smoothing technique, clustering method, and set of features for clustering have yet to be performed, which will conducted in the later chapters.

The SMARTS methodology is composed of three major steps:

1. Create an intermediate set of axes (Z domain) from the output data (Y domain)
2. Fit a regression model from the input space (X domain) to the Z domain

3. Fit a piecewise linear regression model from the Z domain to the output TS space (Y domain)

Step 1

The first step is to create a set of intermediate axes (Z domain) based on the output data (Y domain). The reason why the Z domain is derived from the Y domain is because there is no point in reducing the dimensions of the inputs (X domain). The set of inputs are typically chosen using a DoE to reveal the relationship between the output behavior and the inputs, and there should be no hidden structure in the X domain if the sampling is done correctly.

Dimensionality reduction methods were discussed in Chapter 3, and these are used to reduce the Y domain into significantly lower dimensions. The Y domain has n dimensions and n is the number of time steps each TS, and this is reduced to less than 10 for most cases. For the SMARTS method, PCA is applied to the TS outputs, and the first principal component (PC) is kept as the Z domain.

Step 2

The next step is to create a mapping from the X domain to the Z domain. The dimensionality of the TS output data has been reduced significantly by applying PCA, and normal regression methods for nonlinear data can now be used. For the SMARTS method, NN is used, and the best of ten fits is kept as the regression. As default parameters for using NN, it uses one hidden layer with five nodes. Training is capped at 500 iterations, and the absolute and relative tolerances are 1×10^{-10} and 1×10^{-20}, respectively.

Step 3

The final step is to create a mapping from the Z domain to the Y domain. This involves the following main steps:

1. Reduce the size of the output data
2. Determine the breakpoint location of each TS
3. Cluster the output data based on extracted features
4. Fit linear regressions to the segments and breakpoints

Data reduction is important for regression fit time and to reduce the noise in the TS. The Z domain organizes the TS shapes so that similar shapes are closer together, and the order is such that the TS shape transforms smoothly from one to another. However, the density of points along the axes of the Z domain is not uniform because the TS data was originally created using the sampling in the X domain. A grid or bins can be created in the Z domain, and a representative TS can be selected or calculated from each bin in order to create a uniform density of points in the Z domain. This also helps reduce the total size of the data used for regressions. If there is noise in the TS data, the average or median can be calculated for each bin to create the representative TS. The SMARTS method takes the median from each bin because the average can be more susceptible to outliers. The default for the number of bins is 200, but any bins that have fewer than 20 TS is merged with its neighbors so that the median TS is based on a sufficient amount of data. As a result, the final number of bins can be less than 200. The number of bins and the number of TS per bin can be

adjusted based on the dataset. As a quick example, if the dataset was created using a full factorial of ten input variables at three settings with ten repetitions, then this will result in roughly 600,000 cases, and this can be reduced to 200 rows of TS data, where each row represents a unique TS shape.

The threshold value which will be used by the TS segmentation algorithm in the next sub-step is also calculated with the data reduction step. To calculate the threshold, the difference between the TS and the smoothed version of the TS needs to be taken. As shown later in the Robust Threshold and TS Smoothing Experiment in Section 8.2.2, there are two options for TS smoothing, which are the median and the moving median. When there are less than five repetitions, the moving median algorithm is the better choice. Because the default minimum bin size is 20, the median TS that was already calculated can be used as the smoothed TS. For each bin, the median TS is subtracted from each TS in the bin, and the standard deviation of the differences is calculated as the threshold value for that median TS.

Once the data has been reduced, the breakpoint locations are estimated using a segmentation algorithm. The SMARTS method uses a hybrid segmentation method. A top-down segmentation method first creates more segments than necessary; then a bottom-up segmentation method recombines the segments. Hybrid segmentation is applied to the median TS from each bin to calculate the breakpoints and the number of segments.

Piecewise Linear Regression Data Groups (PLRDG) are formed using clustering. Although clustering algorithms can be applied to raw data, this is ill-advised for

high-dimensional outputs such as TS. In the Feature Selection and Clustering Experiment in Section 8.2.3, the combination of set of features and clustering algorithm are determined. The set of features consists of the first few PCs of the data and the first PC of the breakpoint locations as described in Section 5.3. The number of PCs is determined by using the Parallel Analysis (PA) method, but it is capped at ten because that is sufficient for most applications. The PCA of the breakpoint locations helps with keeping the number of segments of the TS within each group equal, and it also helps to ensure that the breakpoints in each group are aligned. For the clustering algorithm, model-based clustering using Gaussian mixture models are used. It can cluster a line of points, and bent lines are clustered as two separate clusters. Furthermore, model-based clustering is robust to noise and outliers.

Once the clusters are created, several checks are performed. First, each group is checked to make sure that the number of segments are the same for each TS. If they are not, the most common number of segments is applied to the ones that have different number of TS. The breakpoint locations of the neighboring TS are used as the initial location, and these locations are optimized using a downhill simplex algorithm. The standard deviation of the difference between the TS and the piecewise linear regression is minimized. Any overlap between groups in the Z domain is corrected.

Piecewise linear regression models are created to map the Z domain to the Y domain. A set of linear regression models are created to represent each segment of a group. Another set of linear regression models are created for the breakpoints, which define the beginning and end of each segment. Also, a mapping function between the Z domain and the groups is also created.

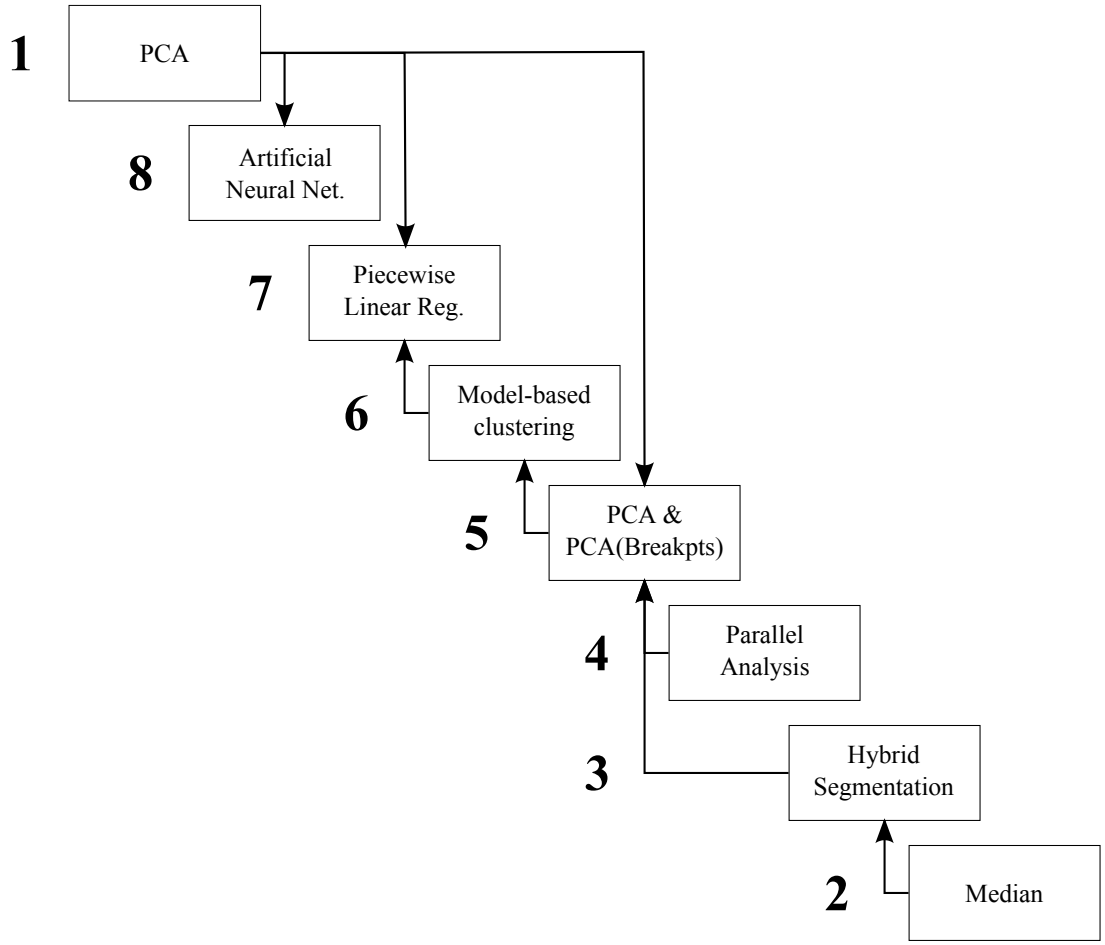


Figure 54: Block flow diagram of the SMARTS methodology. The numbers are the order in which the blocks are executed.

Now that the pieces of the SMARTS methodology have been selected, they can be used to replace the block flow diagram that drove the research of the SMARTS methodology to create the diagram in Figure 54. Each of the blocks have now been replaced with the actual methods that fulfill that role, and the arrows show the direction in which information is passed. The numbers in the chart show the order in which the blocks are executed.

The SMARTS methodology is implemented in the Python programming language, and the source code is provided in Appendix A. There are six main Python files that

are needed to create PLRs.

ApplyClustering.py contains the clustering methods that are used for the experiments in Chapter 8.

FeatureExtract.py contains the methods to extract features from the TS output data such as DFT and Isomap. It also contains the implementation for the PA method.

multivariate_piecewise_linear_regression.py is the main file to create PLRs. It contains the `multivariatePiecewiseRegresesion` and the `PrepOutputData` classes. The `PrepOutputData` class imports the TS outputs which is saved as a comma separated value (CSV) file, applies PCA to the data (Step 1), and reduces the data volume (part of Step 3). The `multivariatePiecewiseRegression` class creates the $X \rightarrow Z$ regression using NN, and it also creates the $Z \rightarrow Y$ regression using PLR. Once the regressions have been created, it has methods to interpolate new TS, and it has several methods for plotting as well.

my_functions.py is a collection of generic helper functions such as normalizing a set of values in an array.

smoothers.py contains all the smoothing functions that were introduced in Section 5.2.5, such as the moving median and Savitsky-Golay smoother.

ts_segments.py contains the methods and classes for creating the canonical TS

datasets such as square and sine waves. The file also contains the TS segmentation methods.

A step by step execution of the methodology using test data is presented in Section 8.2.4 in the SMARTS Feasibility Experiment.

6.4 Upper and Lower TS Bounds for Stochastic Simulation

Given the same input parameters, stochastic simulation models such as those for operations and sustainment produce TS outputs with variations between repetitions. This uncertainty in behavior is a key piece of information when making decisions, and they can be represented using with upper and lower bounding lines. The focus of the SMARTS methodology so far has been on the development of a surrogate model of the median or mean of the TS data. The methodology can be adapted to create the surrogate models of the bounds.

There are two approaches in which the bounds of the TS data can be captured. The first is to create a different TS shape function for each bounding line. This approach results in three different surrogate models in Step 3 of the SMARTS to capture the upper, lower, and middle lines. The other approach is to capture the bounds using different mappings between X to Z, so there will be three NN in Step 2. Using these two approaches, surrogate models of the bounds can be created.

The first approach creates a separate TS shape function for the upper and lower bounds. The regression can be fit to different percentiles of the TS data including the minimum and maximum. The set of repetitions that were initially run to quantify the uncertainty of the simulation is an obvious grouping of the data to calculate the

statistics. Another option is to use the bins from Step 3. Using different statistics results in a set of shapes that are different from those for the median, and the regression fitted to the data will also be different. The benefit of this approach is that it can capture the true bounds of the data.

One drawback to the first approach is that the data of the bounds can be noisy and highly irregular, and fitting a PLR model to this kind of data is difficult. Taking the median of the set of TS helps smooth the data, and the regressions can be fitted more easily. Taking the extremes such as the minimum or maximum, on the other hand, do not have the same smoothing effect and can instead accentuate the irregularities. One way to achieve a smooth upper and lower bound is to use a large number of repetitions or a large bin size, but this requires more simulation runs.

The second approach captures the TS uncertainty by using different regressions in Step 2 while using the same TS shape function in Step 3. Each simulation run corresponds to a value in the Z domain, and because the repetitions are slightly different from each other, this corresponds to a distribution of points on Z. In the SMARTS methodology described above, all the repetitions are used to fit the NN from X to Z. In the second approach, this is modified to capture the upper and lower bounds of this distribution on the Z domain. The variations between repetitions are assumed to be normally distributed, and the mean and standard deviation of the first PC values (or z values) of that set of repetitions are calculated. Then, new first PC values which represent the upper or lower bound are calculated using three standard deviations from the mean. This approach operates under the assumption that the TS shape function can represent all possible TS shapes from the simulation model. The

upper and lower bounds represent the range in which the median can vary. There may be a few outliers that exceed the bounds, but the majority of the lines will be contained within them. The benefit of this approach is that only the TS function for the median is needed, and so it is more robust to noisy data.

6.5 Summary

This chapter summarized the methods and techniques presented in the previous chapters. The steps of the SMARTS methodology was also presented in detail. The second step of SMARTS was also discussed in this chapter, and neural network was chosen as the suitable regression method to map the X domain to the Z domain, thus answering Research Question 14. In the next chapter, research objectives are reviewed and formalized by restating the research questions, claims and hypotheses.

CHAPTER VII

FORMULATION OF HYPOTHESES

The previous chapters provided the theoretical background to formulate a new surrogate modeling approach to represent time sequential (TS) data for a design space exploration application. This chapter summarizes the observations and research questions that motivate the method and formulates the hypotheses that structure the experiments in Chapter 8.

7.1 Review of Observations and Research Questions

Chapter 1 outlined the need for a regression method that can handle TS data for design space exploration. Several observations were made through a closer examination of sample data from the FAMOS model and through examining surrogate modeling literature. The observations are summarized below:

Observation 1 TS data exacerbates the curse of dimensionality.

Observation 2 TS data generated by simulation models have the potential to be organized in a lower dimensional space.

Observation 3 TS data from logistics simulation can be approximated as piecewise linear segments.

Observation 4 Piecewise regressions are ideal candidates for fitting large nonlinear datasets.

Observation 2 provided a hint to develop the SMARTS methodology into a three step method. A closer examination of operations and sustainment (O&S) simulation models also uncovered a similar mechanism by which the inputs are transformed into internal simulation variables, which are then used to run the simulation and generate TS outputs. In the case of the FAMOS model, the inputs affect the consumption and replenishment rates of spare vehicle parts. This $X \rightarrow Z \rightarrow Y$ process was adopted for the SMARTS methodology. By separating the task of capturing the variability across the design space ($X \rightarrow Z$) and representing the range of shapes that are present in the design space ($Y \rightarrow Z$), each regression is made into a simpler problem. This answers Research Question 1, which sought for aspects of the simulation and TS outputs that can be exploited to improve the surrogate modeling process. An additional benefit of this approach is that the output data can be discretized and aggregated in the Z domain in a way that was not possible before, leading to significant data reduction. This leads to Research Question 2, which questions the efficacy of such approach.

To create the Z domain, dimensionality reduction techniques were found to be effective, thus answering Research Question 3. This was naturally followed by Research Question 4 which sought for the appropriate method to reduce TS data, and principal component analysis (PCA) was chosen.

After reviewing the literature on various surrogate modeling techniques, piecewise linear regression (PLR) was identified to possess the potential to have lower fit times and good fit characteristics because the method decomposes the data into smaller groups. This choice also aligned with Observation 3. The PLR still needed to be extended for design space exploration, and Research Question 6 was formed to capture

this requirement.

Because the TS shapes varied across the design space, a part of the regression process was formulated as a clustering problem. A PLR model was created from each of these clusters. Before this can happen, the definition of what makes a *good* cluster was defined (Research Question 7). In order to cluster the TS data, the literature recommended using features extracted from the data instead of using the entire dataset, and Research Question 9 addressed the selection of the appropriate features. A robust method to segment the TS was investigated, and an accurate estimate of the inherent noise was found to be an important parameter because it can be used as the threshold setting for segmentation algorithms (Research Question 11 and 12).

For convenience, the research questions are consolidated below:

RQ 1 What aspects of O&S simulation models and its TS outputs can be exploited to help create surrogate models?

RQ 2 Will creating regressions that use an intermediate set of axes lead to a better regression when fitting TS data?

RQ 3 What method should be used to derive a Z domain so that it can be used for TS regression?

RQ 4 Which dimensionality reduction technique is the most suitable for TS of O&S simulation?

RQ 5 What regression method is capable of representing the entire range of TS

shapes in a TS dataset?

RQ 6 How can piecewise linear regression be extended to fit time TS data for design space exploration applications?

RQ 7 What unique properties should each group exhibit to create piecewise regressions with good fits?

RQ 8 What is the appropriate clustering technique to create groups that will yield piecewise linear regressions with good fits?

RQ 9 What are the appropriate features to use for clustering TS datasets to create piecewise linear regressions with good fits?

RQ 10 What TS segmentation algorithm is suitable for splitting TS data to help create PLR Data Groups (PLRDG)?

RQ 11 How can a robust threshold setting be derived from the data for TS with noise?

RQ 12 Which smoothing technique is appropriate for smoothing TS data to estimate the noise in the data?

RQ 13 How many principal components (PCs) should be kept after principal component analysis (PCA) is applied to a dataset?

7.2 Development of Hypotheses

In this section, several hypotheses are developed with a discussion on the reasoning and likely outcome.

The main hypothesis of this thesis tests the SMARTS methodology.

Hypothesis 1
In the context of design space exploration using simulation models, if the output data consists of high-dimensional and sequential outputs such as a time sequence, then the SMARTS methodology will yield regression models that are equal or better in performance (fit time, model fit, and representation errors) compared to other regression methods.

For TS datasets generated in a design exploration exercise, the inputs are varied in small increments, and the corresponding outputs are TS with shapes that also vary correspondingly in small amounts. There is an assumption that the shape of TS outputs from the simulation model will vary in a smooth manner if the inputs are varied smoothly as well. It is also assumed that the simulation model does not have discontinuities. If this is the case, then separating the regression into two steps like the SMARTS methodology will lead to better fits because it compartmentalizes the regression complexity. The first regression captures the variability in the design space while the second captures how the TS shapes change. In contrast, a single regression model will need to capture these two aspects at the same time.

Hypothesis 2

If principal components (PCs) of the time sequential dataset and the first PC of its breakpoints are used as the features for clustering, then the model-based clustering that uses Gaussian mixture models will yield clusters that are suited for piecewise linear regressions.

Both the features derived from the data and the breakpoints are necessary to satisfy the conditions of a good cluster to make PLRs. The PCs of the TS data are necessary to ensure good membership so that the shapes in each cluster are similar and that the transformation of the TS is one-dimensional. The breakpoints feature is necessary to ensure that the breakpoints within the cluster are aligned.

The TS data should be arranged in line segments when they are projected onto these two feature sets, and density-based and model-based methods are appropriate to cluster these lines. However, if two lines with different angles are touching at the ends, then density-based methods will label these two lines of points as one cluster when they should indeed be two clusters. On the other hand, model-based clustering should be able to identify this as two clusters because it fits an ellipsoidal Gaussian distribution to each line of points as long as the they are distinct enough. Furthermore, if there is some scattering or noise when the data points are projected using the features, density-based clustering may not be able to find the appropriate clusters while model-based clustering can vary the width of the distribution for each cluster.

Claim 1

A smoothing technique can be applied to a time sequence to estimate the noise in the data to a sufficient accuracy so that it can be used as the threshold setting for TS segmentation methods.

It was shown on simulated TS data that a segmentation algorithm can accurately determine the number of segments that compose a TS if the algorithm is applied to the average of the set of repetitions and if the inherent noise in the TS is used as the threshold setting for the algorithm. One of the ways to estimate the noise is to take the difference between the original data and one that has been smoothed by a smoothing algorithm. There are various smoothing algorithms, and they each have different strengths. The appropriate method is needed for TS smoothing.

Claim 2

The parallel analysis method is an appropriate method to determine the number of components to keep when applying principal component analysis to a time sequential dataset that is generated by an engineering simulation model for design space exploration.

The experiment on the parallel analysis (PA) method is a demonstration and a test of its robustness, and this is why the statement is made as a *claim* instead of a *hypothesis*. The literature demonstrates PA as a reliable method, and it is reasonable to believe that it is also applicable for high dimensional TS data.

7.3 Development of Experiments

The experiments are presented in reverse order compared to the research questions and hypotheses. This is because the SMARTS methodology builds and relies on the results of the lower level components before it can proceed to the higher level objectives.

Parallel Analysis Experiment

In order to determine how many PCs should be kept after PCA is applied to a dataset, the PA method was found to be one of the best performing techniques. Section 5.3.1 discussed the details of the PA method and presented an example. To determine how the PA method performs with TS data, an experiment is designed with a wide range of TS shapes and sizes, and it is presented in Section 8.2.1. The purpose of the test is to determine the robustness and accuracy of the PA method to TS data. The PA method has been tested with a wide range of data [51, 82], but it has not been specifically tested for long TS data. The results of this experiment will answer Research Question 13 and demonstrate Claim 2.

Robust Threshold and TS Smoothing Experiment

Various smoothing techniques were presented in Section 5.2.5, and for the SMARTS methodology, they are used to determine the noise in the TS data. To determine the best smoothing technique for SMARTS, the experiment is designed to test all the smoothing functions using a wide range of simulated sequential data, and the details of the experiment are presented in Section 8.2.2. The results of this experiment will answer Research Questions 11 and 12 and demonstrate Claim 1.

Feature Selection and Clustering Experiment

Feature selection and clustering are complementary steps because there are certain types of features that work well with certain clustering algorithms, and these two need to be tested together to find the best combination. A candidate set of features was presented in Section 5.3. These will be tested with a density-based clustering, a modified version of a density-based clustering, and a model-based clustering algorithm. For the actual implementation, DBSCAN from the Python package scikit-learn [79] and R package `mclust` [31] will be used, respectively. A canonical set of TS datasets will be used to test the performance of the clustering algorithms and the feature set. This process requires the breakpoint locations of each TS and the number of components for methods like PCA. To keep the experiments independent, the breakpoint locations will be provided as a known quantity, and the number of components will be used as a parameter of the experiment. The experiment will test Hypothesis 2, and its results will answer Research Questions 8 and 9.

SMARTS Methodology Feasibility and Scalability Experiments

The final set of experiments will test the performance of the SMARTS methodology. The first experiment will test the SMARTS methodology against a simulated TS dataset. This is presented in Section 8.2.4. This experiment will verify that the SMARTS methodology will work on a canonical dataset. Some of the implementation details were provided in Section 6.3, and this experiment will provide a step-by-step account of how the methodology is executed. More details of the implementation will also provided with notes to the code and functions used to create the PLR.

The next experiment will test if the SMARTS methodology can be applied to

a more realistic set of data using simulation data generated using the MRFAMOS model. The experiment is presented in Section 8.2.5. The SMARTS methodology will be compared against several other regression techniques for fit time, model fit error (MFE), and model representation error (MRE). The visual appeal will also be examined because one of the main application areas is for visualization in decision making interfaces.

The final experiment will demonstrate that the regression output from the SMARTS model is applicable for visualization, and it is presented in Section 8.2.6. The data from the previous experiment will be used to develop upper and lower bounds for the TS data. Because the regression is not perfect, the upper and lower bounds can intersect, and visualizations that have constraints to avoid the intersecting lines will be presented.

7.4 Summary

This chapter summarized the observations and research questions that were made throughout this thesis and provided a short narrative of its development. Then hypotheses were formulated based on the research questions, and a short discussion was provided to explain the rationale behind each hypothesis and claim. The experiments were created to answer the hypotheses, and the next chapter describes the experiments in detail and presents the results.

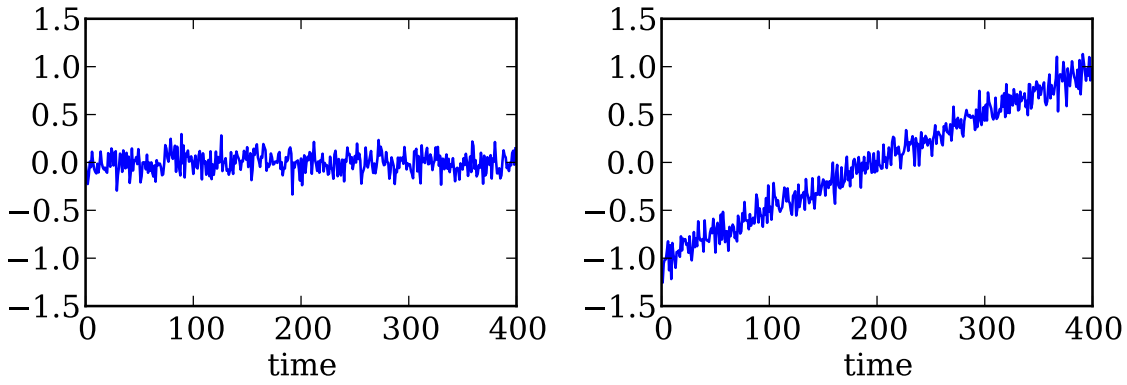
CHAPTER VIII

EXPERIMENTS

The previous chapter reviewed the research questions, formulated the hypotheses and claims, and introduced the necessary experiments. This chapter presents the details of each experiment and its results. The experiments that are performed in this chapter are the following:

1. Parallel Analysis (PA) Experiment
2. Robust Threshold and Time Sequence (TS) Smoothing Experiment
3. Feature Selection and Clustering Experiment
4. SMARTS Methodology Feasibility Experiment
5. SMARTS Methodology Scalability Experiment
6. SMARTS Methodology Visualization Experiment

First, a description of the canonical data is presented, which is used in the experiments. Then, each of the experiments will be explained in detail, and the results will be presented and discussed.



(a) Flat Line (b) Sloped Line

Figure 55: Illustration of the flat and sloped lines

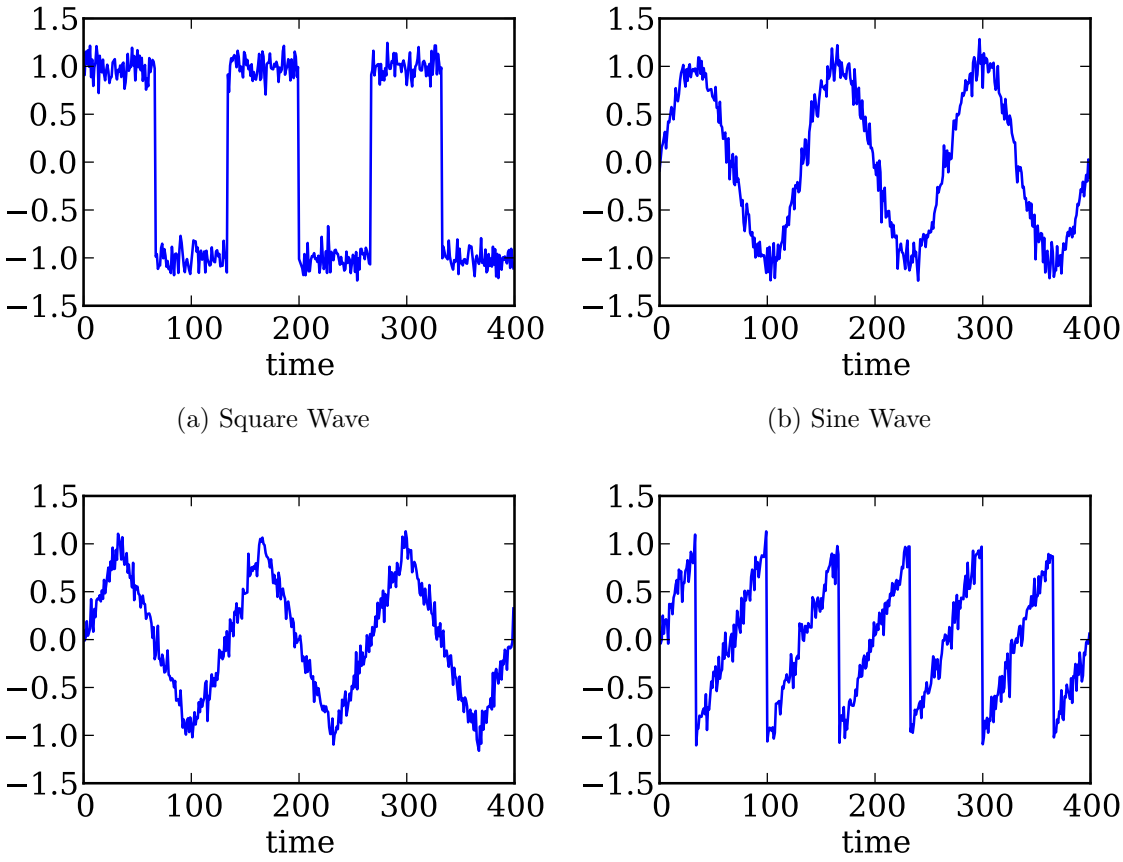
8.1 Data Used in the Experiments

8.1.1 Canonical Datasets

The canonical datasets are composed of different types of TS waves. Some of the waves are flat lines with noise and the sloped lines. Others include the square, triangle, sawtooth and sine waves. These are shown in Figures 55 and 56. Depending on the experiments, these lines and waves are manipulated to create different sequential datasets that change in noise level, amplitudes, phases, number of periods, mean and slope.

Using the different types of TS shapes, four categories of datasets are created, and these are listed in Table 7. Among the four categories, there are seven sets of datasets, which will be called test sets, and each of these test sets are described in further detail in the sections below.

These categories are designed to test different requirements of a piecewise linear regression data group (PLRDG) (see Section 4.3 for PLRDG). For example, the Transformation category contains datasets that test the *transformation rule* as well



(a) Square Wave

(b) Sine Wave

(c) Triangle Wave

(d) Sawtooth Wave

Figure 56: Illustration of the square, sine, triangle and sawtooth waves

Table 7: Categories of datasets

Category	Description	Test Set
Null	Each dataset contains a single TS shape with no variation in its parameters aside from noise	`null`
Transformation	Parameters such as amplitude and slope are varied smoothly	`line`, `waves`
Shape	Parameters are varied discretely to create distinct clusters	`shapes`, `discrete`, `outliers`
TS	Imitates simulation outputs using a specially created TS function	`ts`

Table 8: Description of the `null` test set

Dataset Name	Description	Expected # of PC	Correct # of Clusters
`null-flat`	Set of noisy flat lines		
`null-sawtooth`	Set of sawtooth waves with 3 periods	1	1
`null-sine`	Set of sine waves with 3 periods		
`null-square`	Set of square waves with 3 periods		
`null-triangle`	Set of triangle waves with 3 periods		

as the *alignment rule*, and the Shape category checks the *similarity rule*. This categorization and design of the datasets are important for the clustering experiment (Section 8.2.3), and they will also be used in the parallel analysis experiment (Section 8.2.1).

`null` test set

The `null` test set consists of five different datasets as listed in Table 8. The goal of this test set is to determine the behavior of algorithms when there are no differences between the TS aside from the random noise. Therefore, there is only one PLRDG for each dataset, and only one principal component (PC) is needed to discriminate the rows of TS.

`line` test set

Each dataset in the `line` test is composed of linear lines, and the slope and intercept is varied by moving the start and end points. There are three datasets in the `line` test set, `line-1`, `line-2`, and `line-3`. A summary of this provided in Table 9.

The `line-1` dataset consists of one transformation. The starting point is held

Table 9: Description of the `line` test set

Dataset Name	Description	Expected # of PC	Correct # of Clusters
`line-1`	Line with starting point at 0 and end point varied from 0 to 1	1	1
`line-2`	All of `line-1` plus starting point varied from 0 to 1 while keeping end point at 1	2	2
`line-3`	All of `line-2` plus the end point varied from 1 to 2 while keeping starting point at 1	2	3

constant at 0 while the ending point is varied smoothly from 0 to 1, and so each TS has a different ending point. The ending points are evenly discretized with a spacing of $\frac{1}{n-1}$, where n is the number of TS in the data. Because there is only one transformation, it has one PLRDG, and only one PC should be needed to differentiate the TS.

The `line-2` dataset consists of two transformations. The first transformation is the same as `line-1`. The second transformation holds the ending point constant at 1, and the starting point is smoothly raised from 0 to 1. Because there are two transformations, there are two PLRDGs in this dataset, and at most, two PCs are needed to capture the variation in the data.

The `line-3` dataset consists of three transformations. The first two transformations are the same as `line-2`. The third transformation holds the starting point constant at 1, and the ending point is again raised from 1 to 2. Because there are

Table 10: Description of the `waves` test set

Dataset Name	Description	Expected # of PC	Correct # of Clusters

`amp-smooth-sawtooth`
`amp-smooth-sine`
`amp-smooth-square`
`amp-smooth-triangle`
	(Peak) amplitude is varied smoothly from $A = (1, 2)$	1	1

`phase-sawtooth`
`phase-sine`
`phase-square`
`phase-triangle`
	Phase is varied smoothly $\phi = (-\frac{\pi}{10}, \frac{\pi}{10})$	1	1

`ph-amp-sawtooth`
`ph-amp-sine`
`ph-amp-square`
`ph-amp-triangle`
	Phase is varied smoothly $\phi = (-\frac{\pi}{8}, \frac{\pi}{8})$, then the amplitude is varied smoothly $A = (1, 2)$	2	2

three transformations, there are three PLRDGs in this dataset. The transformations are simple, and the first and third transformation are similar, so two PCs should be sufficient to capture the variation between the TS.

`waves` test set

The `waves` test set varies parameters of wave functions, and the set consists of the `amp-smooth`, `phase`, and `ph-amp` datasets. Each of these are composed of four datasets for each of the different wave functions, which are the sawtooth, sine, square, and triangle waves. Table 10 provides a short description of each of these datasets.

The `amp-smooth` datasets smoothly increase the peak amplitude (A) from 1 to 2 for four wave functions, which are sawtooth, sine, square, and triangle waves. Each wave consists of three periods. Because there is only one transformation, each dataset contains one PLRDG, and one PC should be sufficient to capture the variation in the

data.

The `phase` datasets smoothly vary the phase (ϕ) of the wave functions from $\phi = -\frac{\pi}{10}$ to $\phi = \frac{\pi}{10}$. Each wave consists of three periods. Because there is only one transformation, each dataset contains one PLRDG, and one PC should be sufficient to capture the variation in the data.

The `ph-amp` datasets are a combination of the `amp-smooth` and `phase` datasets. First, the phase is varied from $\phi = -\frac{\pi}{10}$ to $\phi = \frac{\pi}{10}$. Then, while holding ϕ constant, the amplitude is increased from 1 to 2. Each wave consists of three periods. Because there are two transformation, each dataset in `ph-amp` contains two PLRDGs, and two PC are needed to capture differences between TS.

`shapes` test set

The `shapes` datasets contain different TS shapes, and it is similar to combining datasets from the `null` test set into one dataset. The purpose of this test set and the following `discrete` test set is to check if clustering methods can satisfy the *similarity rule* when creating PLRDGs. By creating distinct groups of points, the methods can be checked if they misplace points. This dataset checks to see if different shapes get mislabeled by clustering methods. The `shapes` test set consists of the `shapes-2` dataset and the `shapes-4` dataset, and a summary is provided in Table 11.

The `shapes-2` dataset consists of two TS shapes, which are flat linear lines and square waves. The square waves consist of 3 periods and has an amplitude of 1. Because there are two different shapes, there are two PLRDGs, and only one PC is necessary to differentiate between the flat line and square waves.

Table 11: Description of the shapes test set

Dataset Name	Description	Expected # of PC	Correct # of Clusters
shapes-2	Set of 2 different TS shapes: flat and square	1	2
shapes-4	Set of 4 different TS shapes: sawtooth, sine, square, triangle	3	4

The shapes-4 dataset consists of four TS shapes, which are the four wave types. Each wave consists of 3 periods and has an amplitude of 1. Because there are four different shapes, there are four PLRDGs, and up to three PCs are needed to differentiate between the different wave types.

discrete test set

The discrete test set is created by varying parameters discretely. This creates groups of TS in each dataset that are similar but distinctly different from each other. The discrete test set consists of the amp-discrete datasets and the periods datasets, and these are summarized in Table 12.

The amp-discrete datasets are similar to the amp-smooth datasets, but they vary the amplitude (A) discretely. The amplitude settings are $A = 1, 2, 3$. Because there are three different settings of the amplitude and there is no transformation between them, there are three PLRDGs, but only one PC is needed because only the amplitude is varied.

The periods datasets vary the frequency of the wave functions so that the number of periods (N) in the TS are different. The number of periods in the datasets

Table 12: Description of the `discrete` test set

Dataset Name	Description	Expected # of PC	Correct # of Clusters

`amp-discrete-sawtooth`
`amp-discrete-sine`
`amp-discrete-square`
`amp-discrete-triangle`
	(Peak) amplitude varied discretely $A = \{1, 2, 3\}$	1	3

`periods-sawtooth`
`periods-sine`
`periods-square`
`periods-triangle`
	Number of periods varied discretely $N = \{1, 2, 3, 4\}$	1	4

are $N = 1, 2, 3, 4$, and the amplitude is 1. Because there are four different settings of the period and there is no transformation between them, there are four PLRDGs, but only one PC should be needed because only the period is varied. However, different numbers of periods may appear nonlinear, more PCs may be identified as significant by PA.

`outlier` test set

The `outlier` test set creates data in the same way as the `shapes` and `discrete` test sets, but it replaces some of the rows of the TS dataset with outliers. This test set consists of the `outlier-flat`, `outlier-waves`, `outlier-amps`, and the `outlier-periods` datasets. The last two datasets represent four datasets each, one for each wave type. The `outlier` test set is used to see if clustering methods can identify outliers. The `outlier` test set is summarized in Table 13.

The `outlier-flat` consists of flat lines, and as outliers, square waves with 3 periods and an amplitude of 1 are inserted. This is the same as `null-flat` dataset

Table 13: Description of the `outlier` test set

Dataset Name	Description	Expected # of PC	Correct # of Clusters
`outlier-flat`	Set of flat lines with a square wave as an outlier shape	1	1
`outlier-waves`	Similar to `shapes-4` set with flat line as an outlier shape	3	4
`outlier-amps`	Similar to `amp-discrete` datasets with outlier TS with amplitude of $A = 1.5$ or 2.5 with random uniform probability	1	3
`outlier-periods`	Similar to `periods` datasets with outlier TS with N number of periods, where $N = 1.5, 2.5$ or 3.5 with random uniform probability	1	4

with outliers. There is only one PLRDG, and one PC is needed to group the flat lines and separate out the outlier.

The `outlier-waves` is the same as `shapes-4`, and some of the TS are replaced by flat lines as outliers. There are four PLRDGs, and up to 3 PCs are needed.

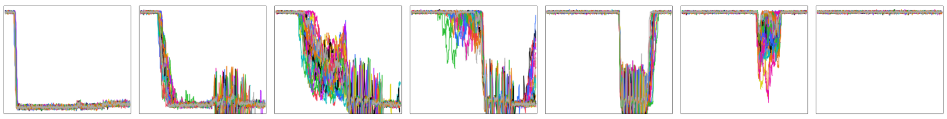
The `outlier-amps` is the same as `amp-discrete`, and some of the TS are given different amplitudes. With a random uniform probability, the amplitude of $A = 1.5$ or $A = 2.5$ is given, and these values were chosen to be between the amplitude of the actual clusters, which is $A = 1, 2, 3$. There are three PLRDGs, and because only the amplitude is varied, only one PC is needed.

The `outlier-periods` is the same as the `periods` dataset, and some of the TS are given different number of periods. With a random uniform probability, the number of periods of $N = 1.5, 2.5$ or 3.5 is given, and these values were chosen so that the outliers are between the main settings, which is $N = 1, 2, 3, 4$. There are four PLRDGs, and because only the period is varied, one PC is needed.

`ts` test set

Based on the data from the FAMOS model that was presented in Section 2.3, a piecewise linear test function was created. The original data and the test function are shown in Figure 57. These shapes are created by defining a TS function that outputs these shapes at certain values. The function receives a single value to create a TS output that is 350 time steps long.

The `ts` test set only consists of the `ts` dataset, and Table 14 provides a quick summary. There are six PLRDGs based on the transformations and the number of breakpoints. Only one PC is needed to organize the data as in Figure 57, but the PA



(a) Output from the FAMOS model

(b) Time sequence output from the test function

Figure 57: Output from the FAMOS model and the TS test function

Table 14: Description of the `ts` test set

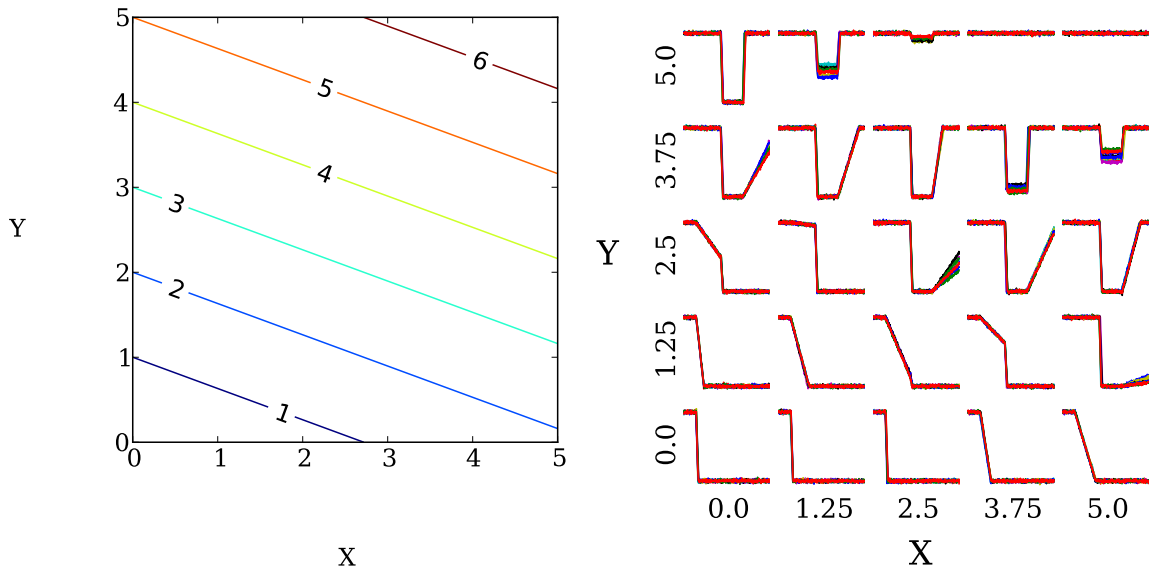
Dataset Name	Description	Expected # of PC	Correct # of Clusters
`ts`	Dataset created by the TS test function	5	6

method may identify more significant PCs because the transformations are complex and because dataset contains a variety of shapes. So, for the estimated number of PCs, a number one less than the number of PLRDGs, 5, is chosen as the estimated number of PCs.

8.1.2 Sample Simulation Dataset

A demonstrative TS dataset (`demo-TS`) is developed using the TS test function so that it can be used to test the SMARTS methodology before applying it to actual simulation outputs. A simulation model takes a number of input values and returns a TS output. In order to imitate this multivariate input, the `demo-TS` is created in two steps.

The TS test function was described in the previous section. As an input to this



(a) Contour plot of $Z = \frac{1}{e}X + Y$

(b) Matrix of TS Outputs

Figure 58: Contour plot of the input function and matrix of TS Outputs of `demo-TS` dataset

TS test function, a separate multivariate input function is defined that takes multiple variables as an input and outputs a single value. The input function has two inputs, X and Y, and the ranges on each are $(0, 5)$. The range is sampled evenly at 21 locations, for a total of 441 input locations. The function has the form $Z = \frac{1}{e}X + Y$, and a contour plot of this function is plotted in Figure 58a. The values from the input function are then fed to the TS test function. Each input value is repeated 10 times. The resulting data can be viewed as a matrix or table of TS plots as shown in Figure 58b.

8.1.3 Simulation Dataset

The simulation dataset is created using the MRFAMOS model, which is a high-level simulation of a fleet of fighter aircraft. It combines three main processes of operations

and sustainment (O&S), which are sortie generation, unscheduled maintenance process, and spare parts logistics. The details of the MRFAMOS model was presented in Section 2.4.

Several scenario-level and vehicle-level parameters are chosen to capture the impact to the fleet-level operational availability (A_O). The parameters to the model are varied according to Table 15. Some of the parameters are varied as a group. In total, there are seven variable groupings. A full-factorial design of experiments at three settings per variable is used to generate the test data. Furthermore, each case is repeated 10 times. This results in 21,870 simulation runs. To generate the validation dataset, an additional 10,000 simulation runs are made with randomized values between the minimum and maximum values in Table 15.

Some of the sample TS from this dataset are shown in Figure 59. Each plot shows all 10 repetitions from a set of input settings, and the dark black line is the median of the 10 runs. The output data is significantly noisier than the data from the FAMOS model, but the median shows a distinct trend through the band of uncertainty.

8.2 Experiments

8.2.1 Parallel Analysis Experiment

The parallel analysis (PA) method is used to determine the number of PCs to be kept from a principal component analysis (PCA). There are alternative methods to accomplish this task, and the justification for PA is made in Section 5.3.1. To quickly summarize the PA method, the eigenvalues from the data and the baseline are compared, and the number of the first few eigenvalues that are greater than the

Table 15: Input parameters and values for MRFAMOS model

Variable Name	Type	Grouping	Values	Units
Fleetsize	integer	–	90, 100, 110	aircraft
Training flight hours per month	float	–	20.0, 22.0, 24.0	hours
C1* fleetsize	integer	1	10, 12, 14	aircraft
C2* fleetsize	integer	1	10, 12, 14	aircraft
C1* sorties per day	integer	1	10, 12, 14	missions
C2* sorties per day	integer	1	10, 12, 14	missions
Mission A length	float	2	1.0, 1.5, 2.0	hours
Mission B length	float	2	2.0, 3.0, 4.0	hours
Mission C length	float	2	2.0, 3.0, 4.0	hours
Part Life**	Exp(lambda)	–	0.054, 0.06, 0.066	1/hours
Part Repair Time**	Tri(mode, min, max)	–	(1296, 972, 2592) (1440, 1080, 2880) (1584, 1188, 3168)	hours
Global Inventory Level**	integer	–	400, 450, 500	parts

* C1, C2 is Contingency Operation 1 and 2

** applies to all parts in the simulation

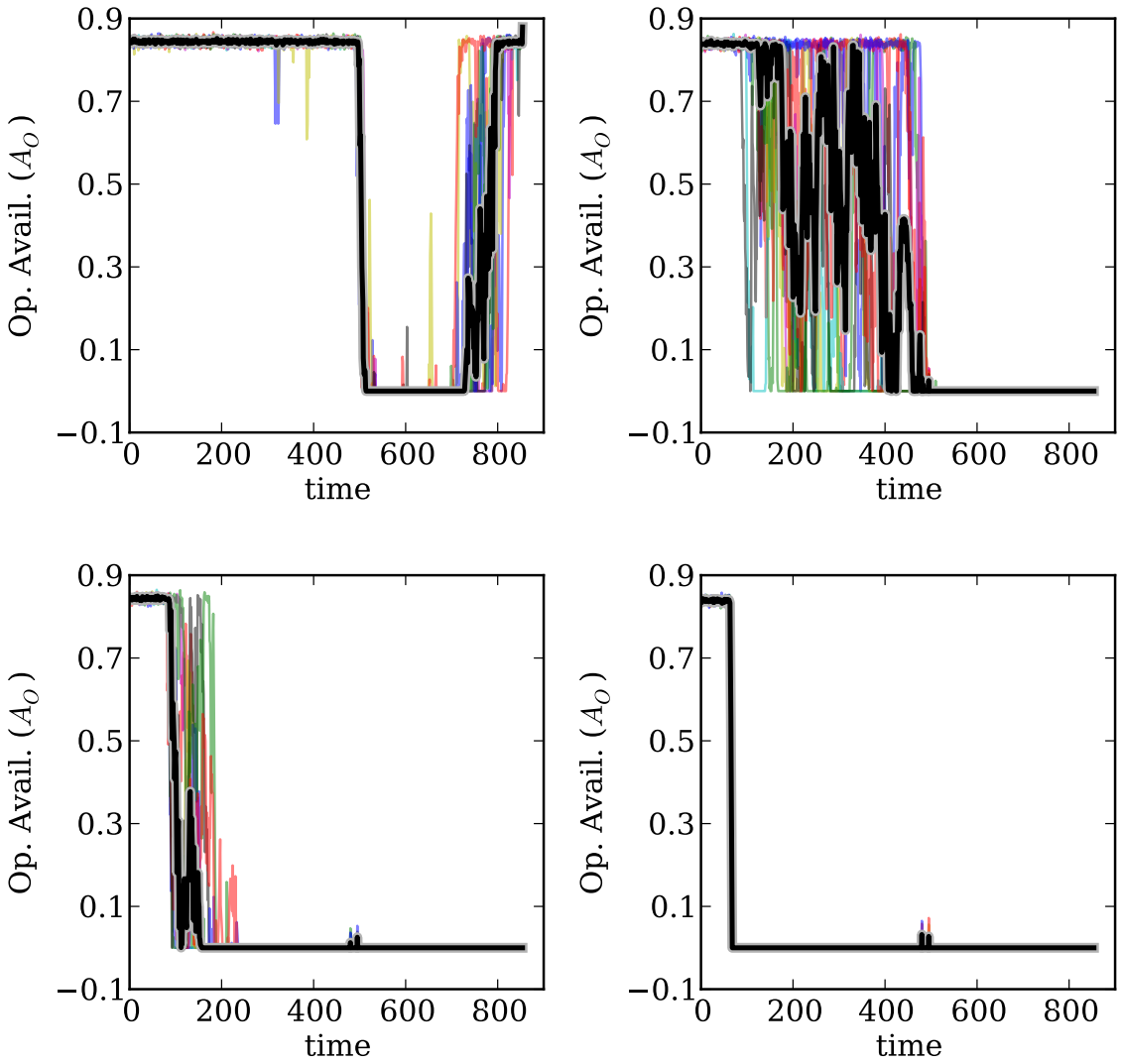


Figure 59: Sample runs from the MRFAMOS model. Operational availability A_O plotted over simulation time. Each plot shows 10 repetitions of actual simulation results, and the black line is the median of the 10 runs.

Table 16: Design space for varying the size of the baseline dataset

Number of Rows	10, 30, 50, 100, 150, 200, 300, 400, 500, 1000, 2000
Number of Columns	100, 200, 300, 400, 500

baseline is the number of PCs to keep. The eigenvalues are calculated from the correlation matrix of the TS dataset, and the baseline is calculated from a set of randomly generated matrices, which has the same dimensions as the original data. Example 5.3 explains the PA method step-by-step.

8.2.1.1 Baseline Investigation

The trends of the baseline are examined first to understand the behavior of the baseline. The size of the baseline dataset is varied according to Table 16. The results are plotted in Figure 60. To generate these eigenvalues, the average of the first 10 eigenvalues from 1,000 datasets is taken. The plots show the first 10 values of the eigenvalues calculated from the baseline datasets. As the number of rows increases, the values decrease, and as the number of columns or variables increases, the values of the first 10 eigenvalues also increase. The eigenvalues indicate the amount of variability that is captured by that axis, so with more rows, the ability to explain the variability in the data is more evenly distributed. Therefore, the value of the first 10 eigenvalues decrease. Because the random dataset is not truly random, it is possible to collect more of the variability of the data into the first 10 eigenvalues. If the number of columns or variables is increased, then this increases the dimensionality of the data as well as the total amount of variability that can be explained by the first few components.

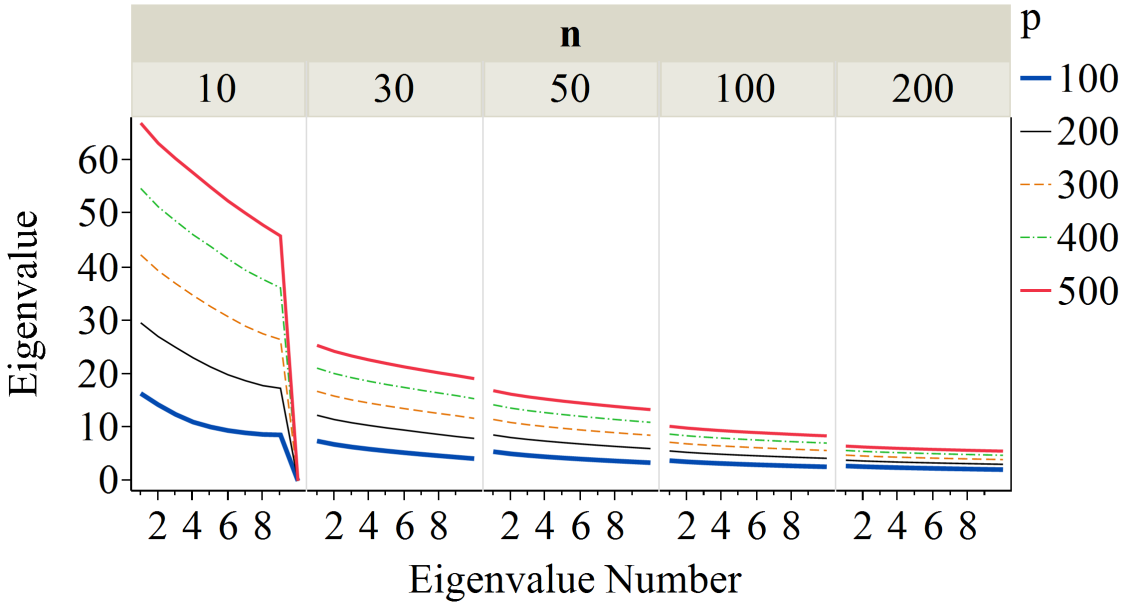


Figure 60: Values of the first 10 eigenvalues of the baseline varied by the number of rows (n) and columns (p)

8.2.1.2 Focused investigation

Next, the behavior of eigenvalues of specific canonical datasets are examined. The datasets that are used include the following: `line-1`, `line-2`, `line-3`, `shapes-4`, `periods-square`, `amp-smooth-square`, and `phase-square`. These are selected because each dataset varies a specific aspect of the TS. The results are plotted in Figures 61 and 62, and the respective baselines are overlaid in black.

The `shapes-4` dataset uses four different waves shapes so at most three different axes are needed to distinguish the shapes, and Figure 62 shows that the first three are above the baseline, meaning three PCs have meaningful information. The `amp-smooth-square` dataset only needs one PC, and this is confirmed by the figure as well. Similarly, the `periods-square`, `line-1`, `line-2`, and `line-3` have

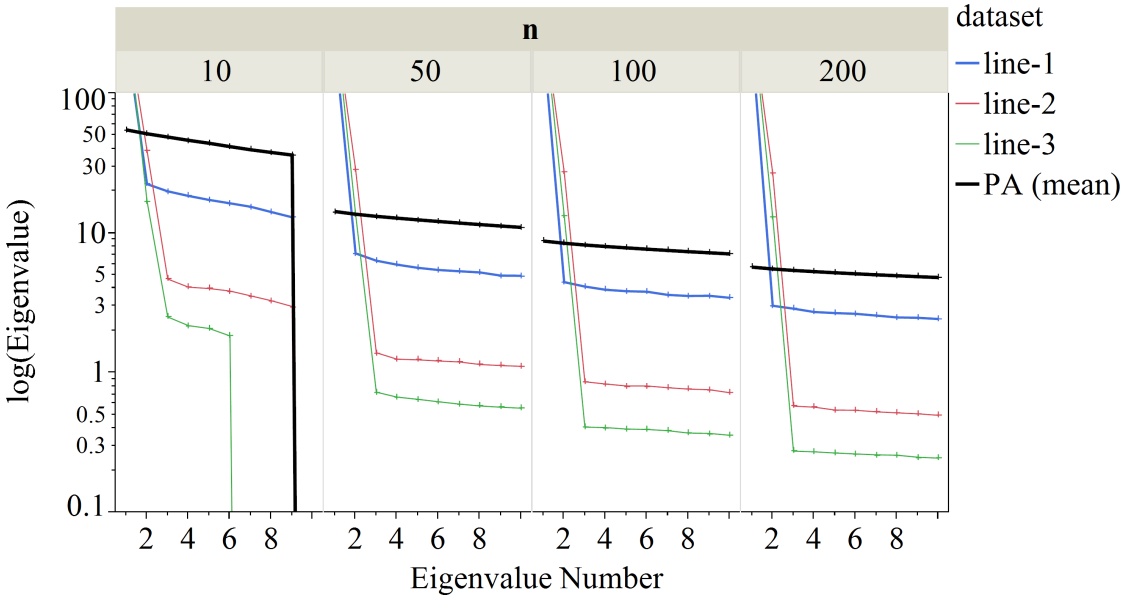


Figure 61: Plots of the values of the first 10 eigenvalues of the correlation matrix of the `lines` dataset against the PA baseline mean

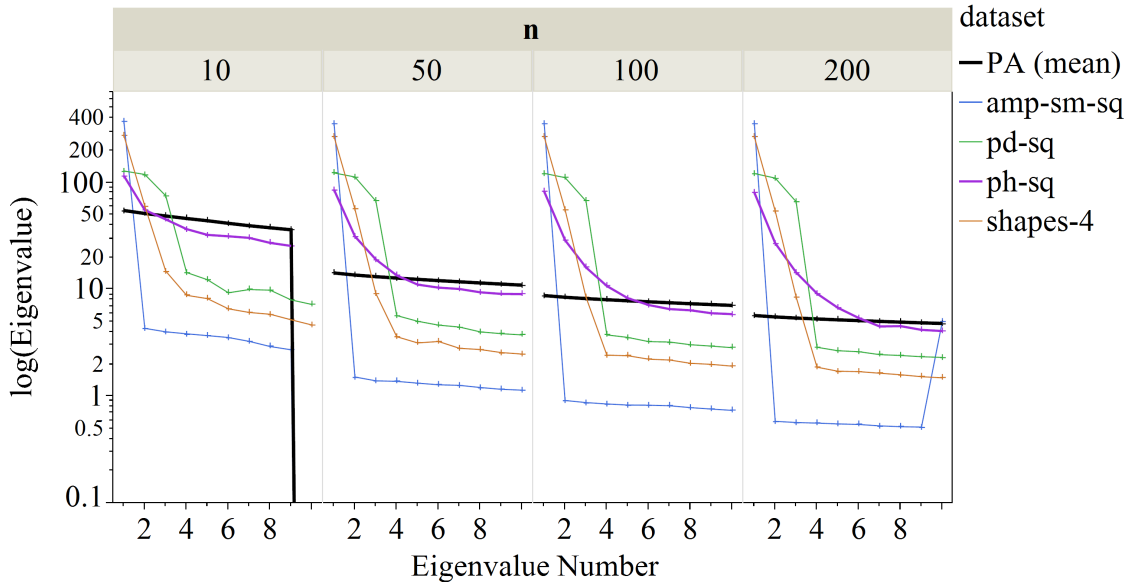


Figure 62: Plots of the values of the first 10 eigenvalues of the correlation matrix of `amp-smooth-square`, `periods-square`, `phase-square`, and `shapes-4` against the PA baseline mean

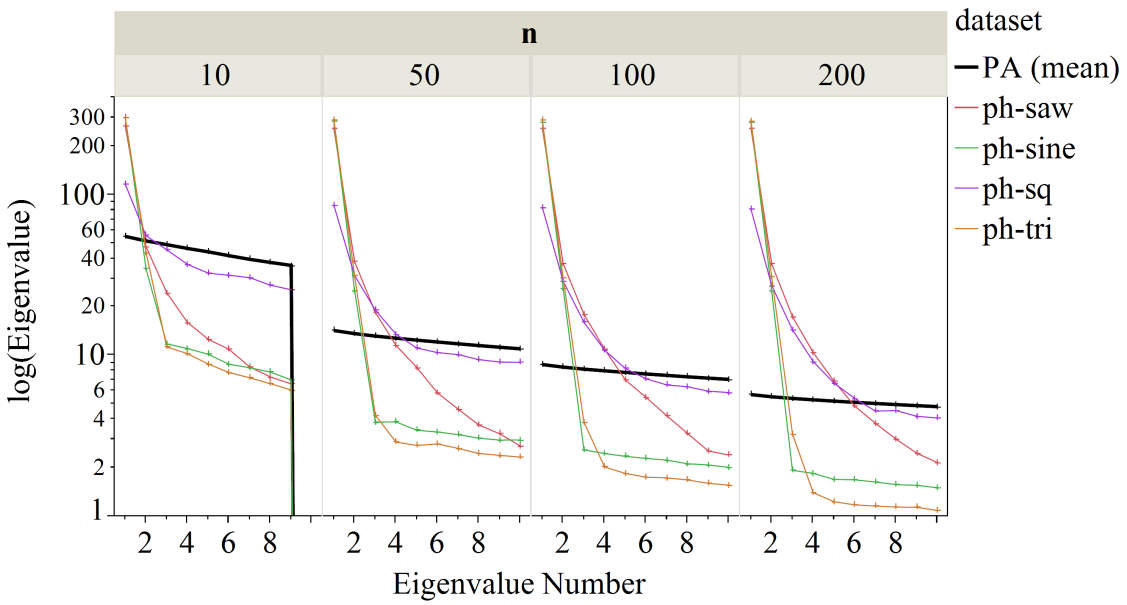


Figure 63: Comparison of the first 10 eigenvalues of the `phase` dataset against the baseline

results that are consistent with expectation. Most of the results show the eigenvalues sharply declining after the first few. The only exception is the `phase-square` dataset, which unlike the others has a more gently sloping line, and this results in more PCs being retained than necessary when only one is needed. This is perhaps due to the discontinuity in the square waves causing a nonlinear correlation between the time steps.

To investigate the smooth decay of eigenvalue values in `phase-square` dataset, the `phase-sawtooth`, `phase-sine`, and `phase-triangle` are also evaluated. The results are compared in Figure 63. The data with discontinuities, `phase-square` and `phase-sawtooth`, have a smoothly decaying eigenvalues while the ones for `phase-sine` and `phase-triangle` drop off sharply.

Table 17: Parameters that are varied in each of the datasets

Type	Values
Number of repetitions	10
Number of time steps (columns)*	400
Size of data in rows	10, 50, 100, 200
Standard deviation of noise ($N(0, \sigma)$)**	0.1, 0.2, 0.3, 0.4

*`ts` dataset has length of 350 time steps

**`ts` dataset only has one noise setting, $\sigma = 0.1$

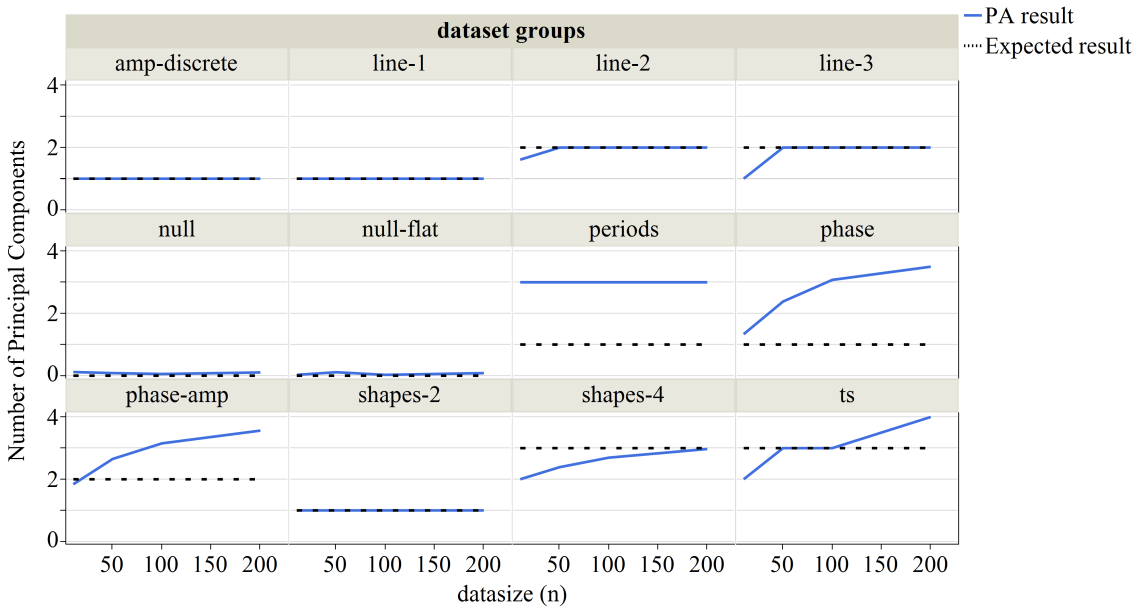


Figure 64: Overall results of the full investigation for PA, broken down by individual datasets

8.2.1.3 Full Investigation

For the full investigation, the PA method is tested against a wide range of canonical datasets, which were described in Section 8.1.1. The ranges of parameters that are varied are summarized in Table 17. The results from the full investigation are shown in Figure 64. The line graphs are the average values of 10 repetitions.

The PA method correctly identified zero significant PC for the `null` test set, but

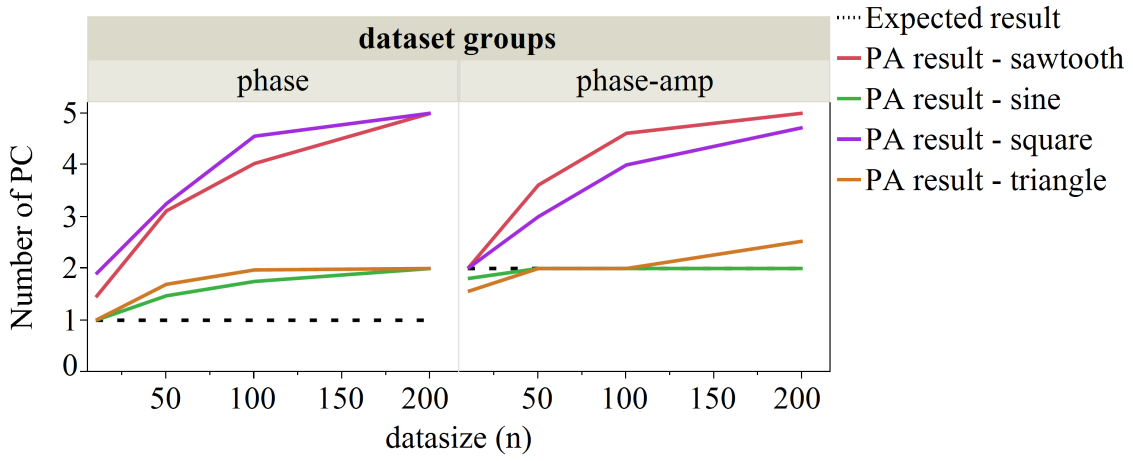


Figure 65: Number of significant PCs as determined by PA for `phase` and `phase-amp` datasets

the identification is not perfect. For the `line` test set, the PA method seems to have trouble when the dataset is small ($N_{rows} = 10$). With the `shapes` test set, the PA method does not have trouble with the `shapes-2` dataset, but it estimates a lower number of PCs with the `shapes-4` dataset. Depending on the size of data and the level noise, the result varies from 2 to 3, while the expected number is 3. For the `amp` dataset, the PA method result is the same as the expected result. For the `periods` dataset, the PA method overestimates the number of significant PC. The expected value is 1 PC because only the number of periods is varied, but the PA result is 3 significant PCs. For the `phase` datasets, the PA method in general identifies more than the expected value, and this is consistent with the findings from the previous focused investigation. For the `phase-amp` datasets, the PA method also overestimates the number of PCs.

Figure 65 shows the results of `phase` and `phase-amp`, and each of the wave function shapes are broken out into their own lines. The PA method identifies more

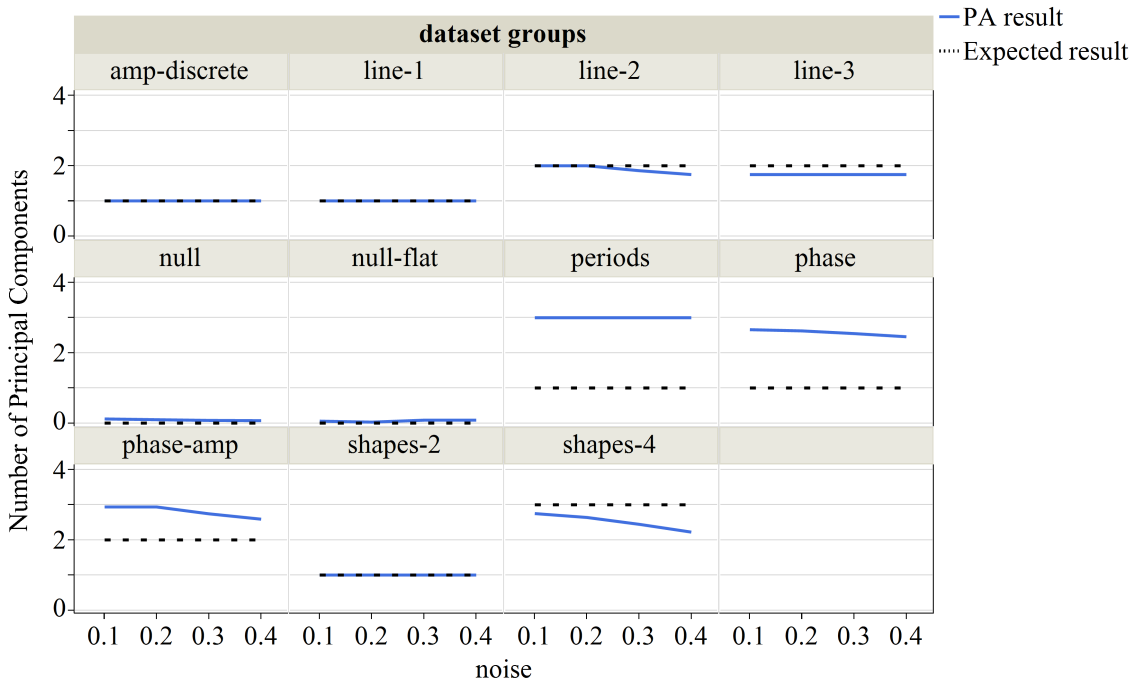


Figure 66: The impact of noise to the number of significant PCs as determined by PA

significant PCs for sawtooth and square waves, which is consistent to what was observed in the previous section. The results from sine and triangle shapes are closer to expectation.

For the `ts` dataset, there is no "correct" number of significant PCs. The data can be organized with one PC, and there appears to be a meaningful structure in the fourth PC. It is interesting to note that the number of significant PCs identified by the PA method varies with the size of the dataset. One explanation is that a larger dataset can capture the complex transformations of the TS in better detail.

Figure 66 shows the results from PA method and the expected results with respect to the noise level. The noise level does not make a difference for most datasets. There is a trend to identify less PCs with increase in noise for the `phase`, `phase-amp`,

and `shapes-4` datasets.

8.2.1.4 Conclusion of the Parallel Analysis Experiment

The PA Experiment was designed to determine if the PA method could determine the significant PCs when PCA is applied to a dataset. Overall, the PA method demonstrated that it can select the same number of PCs as expected.

8.2.2 Robust Threshold and TS Smoothing Experiment

To estimate the noise in the TS, various smoothing techniques are applied to the test datasets. Smoothing functions were introduced in Section 5.2.5, and the algorithms that will be tested are the following:

case mean taking the mean of the repetitions at each time step

case median taking the median of the repetitions at each time step

moving average taking the mean of points across n time steps, where n is the window size, at each time step. The window is centered at the current time step

moving median taking the median of points within the window size at each time step

moving triangle taking a weighted moving average, where the weights are distributed in an equilateral triangle form

moving Gaussian taking a weighted moving average, where the weights are distributed according to a Gaussian distribution

Table 18: Morphological matrix of test functions parameters

Shape	sawtooth	sine	square	triangle			
Noise $N(0, \sigma)$	0.05	0.1	0.15	0.2	0.25	0.3	0.35
Length of TS	100	200	300	400			
Number of periods	1	2	3				
Amplitude	1	2	3				
Repetitions	1	5	10	15	20	25	30

MARS multivariate adaptive regression spline in one-dimension

SVR support vector regression in one-dimension

Kalman one-dimensional Kalman smoother

Savitsky-Golay this smoothing filter fits a polynomial regression within the moving window at each time step, and two variants are tested

- **S-G simple** uses a linear regression with a window size of 3
- **S-G auto** automatically determines the best window size and power of the polynomial

The parameters of the test datasets that are varied in the experiment are listed in Table 18. The repetitions parameter is the number of TS that is used to determine the noise in the data, and each of the TS have the same parameter settings. Some methods can use the repetitions to determine the noise, such as the **case mean**. The **moving** smoothers and the regressions (**MARS** and **SVR**) use the repetitions as well. For **Kalman** and **Savitsky-Golay**, the algorithms are applied to each repetition separately, and so they do not use the repetitions to its advantage.

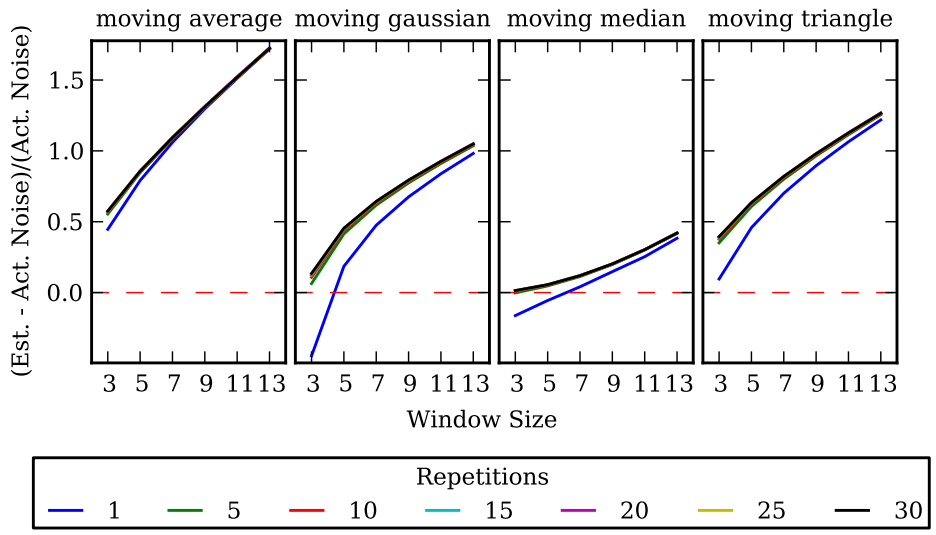


Figure 67: Normalized difference between the estimated and actual noise plotted against the moving window size

The first set of experiments determines the appropriate window size for the "moving" smoothers. An experiment is performed varying the window size from 3 to 13. Figure 67 shows the average difference between the estimated noise using the moving smoothers compared to the actual noise that was present in the data. The actual noise is calcuated by taking the difference between the noisy data and the original signal that was used to create the data. In the plot, the x-axis is the window size, and each line represents the different number of repetitions in the dataset. The effects of the other parameters such as the shape of the wave function and number of periods are collapsed into the mean.

As the window size is increased, the moving smoothers overestimate the noise. The repetitions do not seem to have much impact beyond five repetitions. For all four moving averages, a window size of 3 is the best choice when estimating the error. Because **moving median** outperforms the other three, only the **moving median**

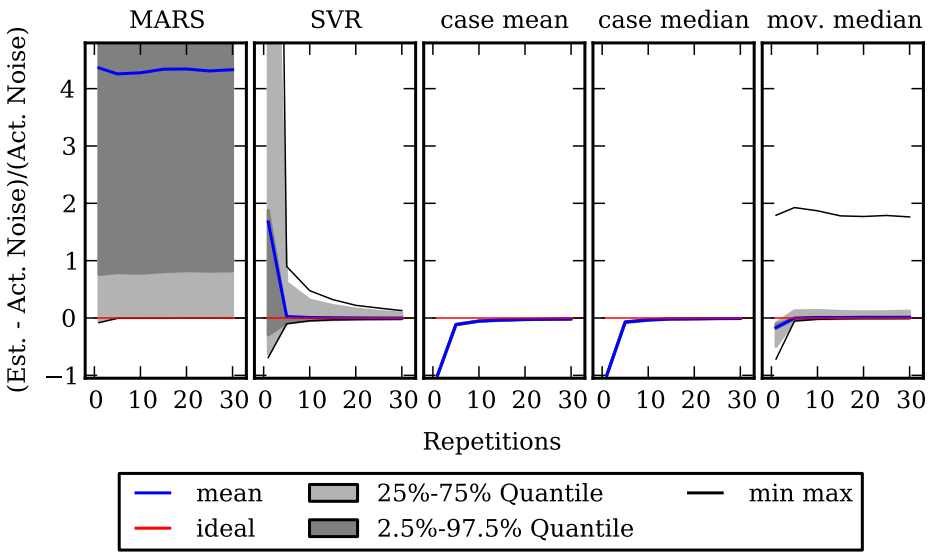


Figure 68: Normalized difference between the estimated and actual noise plotted against the number of repetitions

will be evaluated against the rest of the options in the next experiment.

The rest of the smoothers are evaluated next. Figure 68 shows the results of **MARS**, **SVR**, **case mean**, **case median**, and **moving median**. The plot shows the normalized difference between the estimated noise that the smoothers calculated and the actual noise in the data, and the x-axis is now the number of repetitions. Overall, the **moving median** performs the best across wide ranges of repetitions. For higher number of repetitions, the **case mean** and **case median** perform better than the **moving median**, but they perform poorly when there are no repetitions because they fail to smooth the data.

Figure 69 shows the results of **Kalman** and **Savitsky-Golay** against the **moving median**. The x-axis this time is the length of the TS. The repetitions do not make an impact to the performance of the **Kalman** and **Savitsky-Golay** smoothers because the algorithms are applied to each TS even if a set of repetitions are provided to the

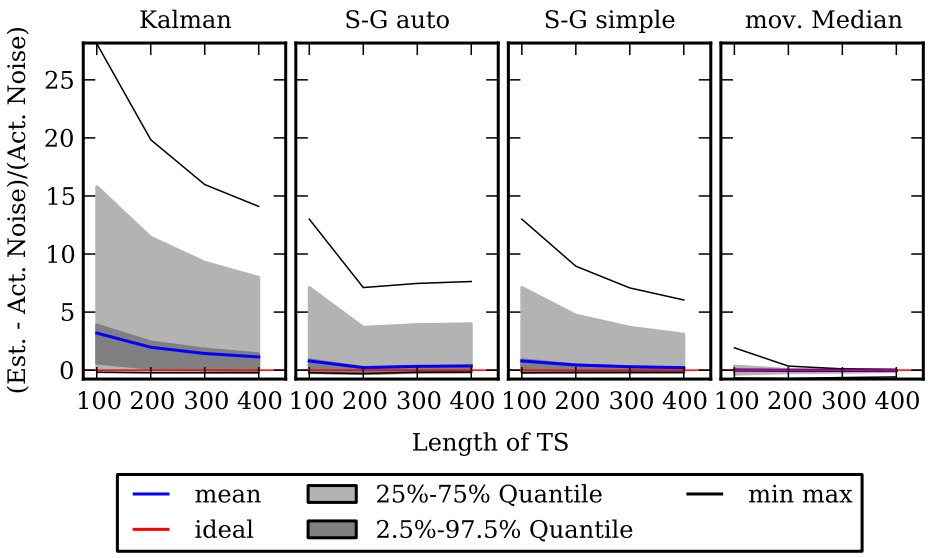


Figure 69: Normalized difference between the estimated and actual noise plotted against the length of the TS. **S-G** is short for Savitsky-Golay

algorithms. The noise estimates improve with the length of the TS, and the **moving median** is the best choice among these smoothers.

8.2.2.1 Conclusion of the Smoothing Experiments

Based on the experiments, the **moving median** smoother with a window size of 3 was determined to be the best overall choice to estimate the noise in TS data. However, when there are repetitions that can be used to smooth the data, then **case mean** and **case median** are better choices. The median has an additional benefit of being more robust to outliers. So, if there are repetitions, then **case median** is the best smoother, and if there are no repetitions, then **moving median** should be used.

Table 19: List of parameters used for the canonical datasets

Parameters	Values			
Shape	sawtooth	sine	square	triangle
Noise $N(0, \sigma)$	0.1	0.2	0.3	
Length of TS	400			
Number of rows	10	50	100	200
Number of outliers	1	2	3	
Number of principal components	1	2	3	4
Repetitions	10			

8.2.3 Feature Selection and Clustering Experiment

The Feature Selection and Clustering Experiment is conducted to determine the best combination of set of data features and a clustering algorithm to group TS data. The description of the datasets have already been given in Tables 8 to 14. Table 19 lists the parameters that are varied, and not all datasets use all the parameters. For example, the `line` test set does not use the *shape* parameter.

The set of features are listed in Table 20. For some of the feature extraction methods like PCA, there are multiple features (i.e. multiple PCs) that can be kept, and the number of features that are kept from these algorithms is varied as a parameter.

Features can also be used together, but this is a combinatorial problem. Even considering pairs of features results in 56 combinations, and this excludes the additional dimensions of multiple PCs from some of the methods. Furthermore, these sets of features need to be tested with clustering algorithms. In the interest of time, several promising feature combinations are handpicked. PCA is chosen as one of the main features because it performs relatively well on TS datasets. Breakpoint locations are

used in some of the combinations because they are important for fitting PLR. The sorted PCA has interesting characteristics where it spaces out data points that are close together. The mean and standard deviation are included to see if it enhances the capabilities of the PCA.

Three clustering algorithms will be used together with the set of features, and the specific implementations of the algorithms are DBSCAN, DBSCANmod, and `mclust`. DBSCAN is a density-based clustering algorithm. DBSCANmod runs a multivariate line segmentation to the results of DBSCAN. The R package `mclust` is a density-based clustering algorithm. The details of these methods are discussed in Section 4.4.1 and 7.3.

There are a total of (14 feature sets $\times$ 3 clustering algorithms = 42) combinations of feature sets and clustering algorithms. If the number of components, such as PCs from PCA, are considered separately, then there are an additional 90 combinations. These combinations are tested against 44 datasets, and each of the datasets are created using the parameters in Table 19.

8.2.3.1 Metrics of Interest

To evaluate the performance of the clustering algorithms, the metrics of interest are related to the characteristics that make a good cluster for PLR, which was discussed in Section 4.3. The two important criteria are membership of the clusters and the alignment of the breakpoint locations, and these can be checked by evaluating two metrics. The first is to check the number of clusters that are calculated by the clustering algorithm, and this can be easily performed by counting the number of

Table 20: Data feature combinations

Categories	Feature Names	ID#
Dimension reduction	Principal Component Analysis (PCA)*	1
	Isomap*	2
	Locally Linear Embedding (LLE)*	3
General statistics	Mean	4
	Standard deviation	5
Other	Discrete Fourier Transform*	6
	PCA(Breakpoints)**	7
	Base2(Breakpoints)	8
Combinations	PCA* + PCA(Breakpoints)**	9
	PCA* + Base2(Breakpoints)	10
	PCA* + sorted PCA** + PCA(Breakpoints)**	11
	PCA* + sorted PCA** + Base2(Breakpoints)	12
	PCA* + mean	13
	PCA* + standard deviation	14

* Number of components kept was varied from 1 to 4

** Only the first principal component kept

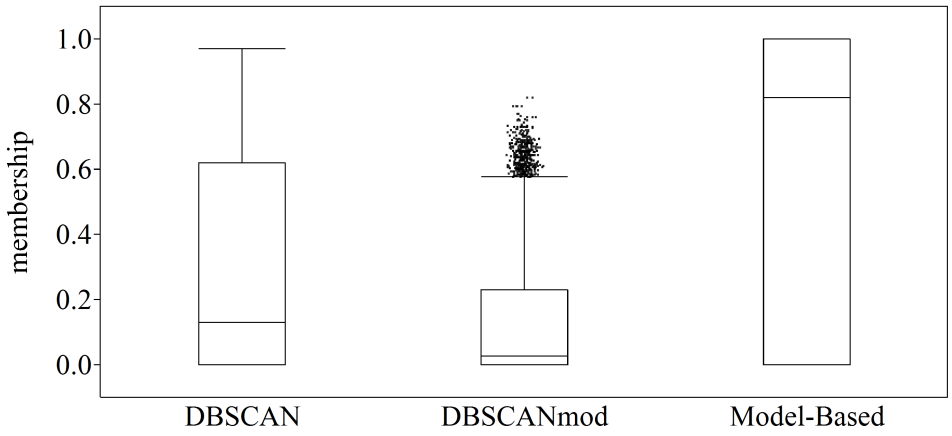
unique labels that are assigned to the data points. This will be referred as the $N_{cluster}$ metric. The other metric is to verify that the points in each cluster are indeed in the correct cluster, and this can be accomplished by comparing the assigned cluster label and the correct cluster label, which is determined beforehand. This will be referred as the *membership* metric.

Sometimes the clustering algorithm fails to find any clusters due to an error during the calculation or in the case of density-based clustering, the points are too far apart to develop any clusters. Checking the number of failed clustering attempts provides insight into the clustering process.

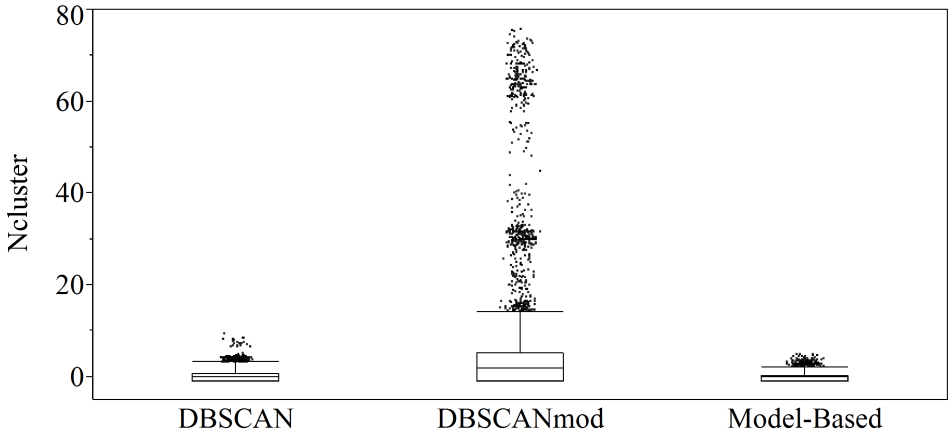
Most of the results will be displayed as outlier box plots. Each data point for the plot is the result of running the clustering algorithm to a set of features extracted from a dataset that was generated using a certain combination of parameters. For example, the DBSCAN is run on the first two PCs of the `null-flat` dataset with noise of $N(0, 0.1)$, 50 rows of TS, and no outliers, and the resulting *membership* is 0.5. Figure 70 is an example of a box plot. The box portion spans from the first to the third quartile, and this range covers 50% of the data and is called the interquartile range (IQR) [102]. The middle line is the median. The whiskers extend out to the furthest point within 1.5×IQR. Points beyond the whiskers are displayed as dots.

8.2.3.2 Evaluation of the `null` test set

The `null` test set is a repetition of the same basic shape so there is only one cluster. This experiment checks to see what the clustering algorithm does in the presence of no sub-clusters, and if there are more than one cluster, then the algorithm is overfitting



(a) *membership*



(b) $N_{cluster}$

Figure 70: Aggregated results of the clustering experiment for the `null` test set

the space. First the aggregated results are presented in Figure 70. It is important to note that the aggregated results can be biased from the selection of features that were chosen. Furthermore, it includes failed points, so Figure 70 and similar plots should only be used to provide an initial insight into the results.

From the aggregated results, it can be seen that DBSCANmod does not perform well compared to DBSCAN and `mclust`. When inspecting the $N_{cluster}$, it is clear that DBSCANmod is creating more clusters than necessary. On the other hand, the

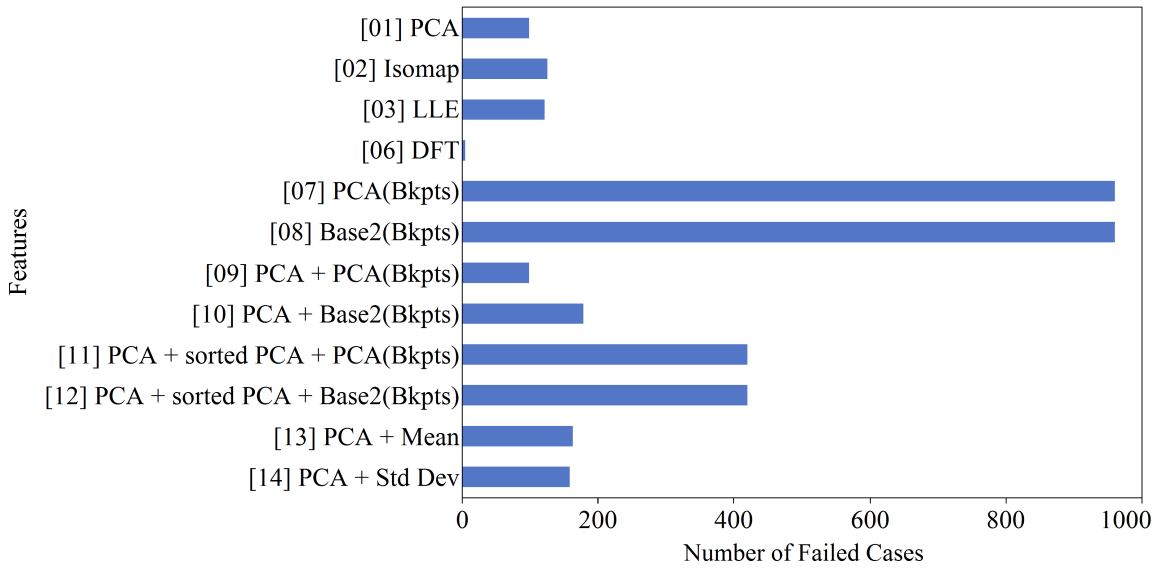


Figure 71: Failed cases categorized by the feature set

model-based clustering is performing better than the others, and its median on the *membership* metric is around 80%, which is very promising considering that this is an aggregated view.

Evaluation of the Failed Cases from the `null` Dataset Experiment

Next, the failed cases are evaluated. There were a total of 3,707 failed runs out of 14,400, which is about 25%. First the failed cases are evaluated by categorizing them using the features as shown in Figure 71.

Failed Cases vs. Features

Each feature set was run with 960 runs each. There was no variation in the break-point locations in the `null` dataset, so the values for `PCA(breakpoints)` and `Base2(Breakpoints)` are the same value. This would result in a singular value when the PCA is applied. Also, normalization is applied to the features as a pre-processing step to the clustering algorithms, and if the vector of feature values is the

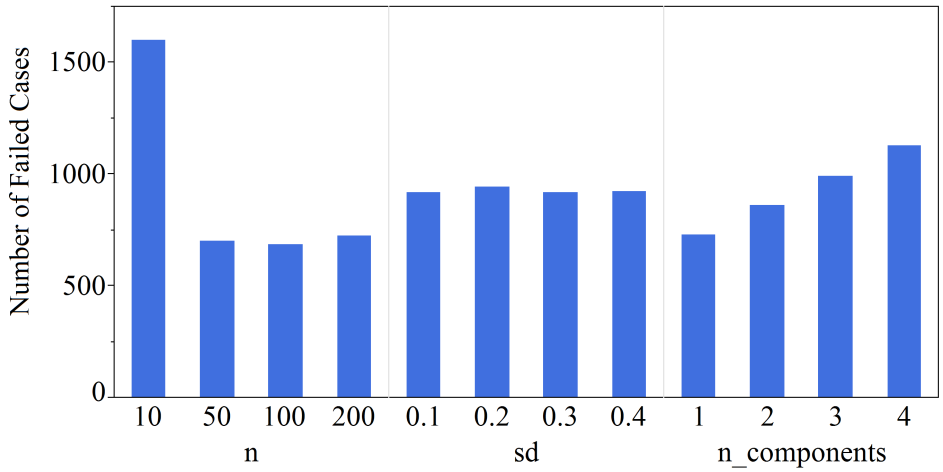


Figure 72: Failed cases categorized by the size of the data n, noise in the data sd, and the number of components kept by PCA, DFT, Isomap and LLE

same number, this would result in a division by zero. This would cause the clustering to fail, and this is the case with `Base2(Breakpoints)`.

The sets of features that use the `sorted PCA` are the third and fourth highest in the number of failed cases. These also use the breakpoints, but as another preprocessing step, the columns with non-real numbers (such as `NaN` or "not a number") are eliminated so the breakpoints features should not have an impact on the results. Thus, it appears that the combination of `PCA + sorted PCA` yields more failed cases than just using `PCA` on its own. `DFT` has the lowest number of failed cases.

Next, the failed cases are categorized using other parameters of the experimental design, and these are presented in Figures 72 and 73.

Failed Cases vs. Data Size

When the failed cases are categorized according to the size of the data (Figure 72), the small data size of 10 points claims roughly half of all the failed cases, which is a little over 10% of the entire `null` test set. Even if the points are grouped closely together,

the normalization can make the distances between the points to appear greater if the sample size is small. Density-based clustering would especially have a difficult time because it uses a fixed distance ϵ to determine its neighborhood of points. So if the points are farther apart, then each point will not be able find neighbors without increasing ϵ. A dataset with 10 points is very small, and there is no point in applying a clustering algorithm. However, it has been useful in highlighting the importance of scaling of the feature values.

Failed Cases vs. Noise

The middle bar chart of Figure 72 shows the failed cases as a function of noise, and at least in this view, noise does not have a noticeable impact to failed cases. The only difference between the rows of TS in each dataset is the random noise, and because the data is normalized before clustered, the change in noise has no impact.

Failed Cases vs. Number of Components

The histogram on the right of Figure 72 shows the failed cases as a function of the number of components used for PCA, DFT, Isomap, and LLE. An increase in the number of components corresponds to a rise in failed cases. This result is consistent with the reasoning presented when evaluating the impact of the size of the data. Adding components increases the dimension of the feature domain, and this causes the points to be farther apart when the distance is measured using Euclidean distance. Once again, this is due to the fact that the features are normalized. Furthermore, the `null` test set should only require one component, and additional components only add extraneous features that do not help differentiate the data points.

Failed Cases vs. Clustering Method

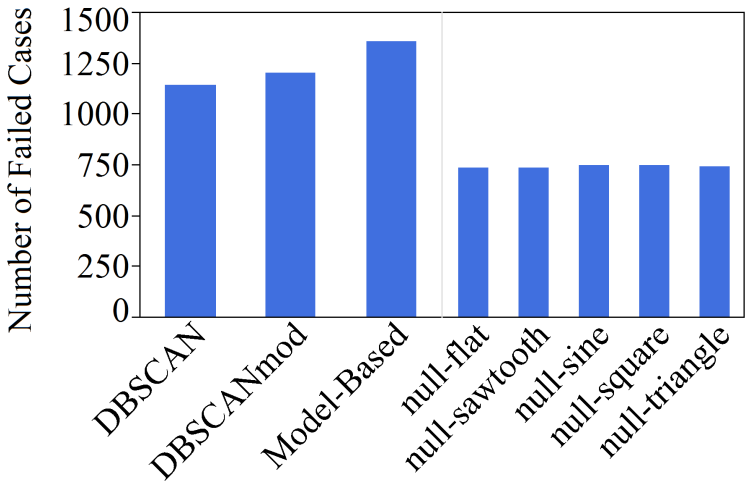


Figure 73: Failed cases categorized by the clustering algorithms and the datasets

The left histogram of Figure 73 shows the number of failed cases as categorized by the clustering algorithms. Although there is no large difference between the methods, the model-based method has a higher number of failed cases compared to the other two.

Failed Cases vs. Dataset

The right histogram of Figure 73 shows the number of failed cases as categorized by each datasets in the `null` test set. The distribution of failed cases is uniform across the datasets. There are no major differences besides the shape as expected. The failed cases are driven more by the other factors such as the type of features or the size of the data.

Evaluation of the Good Cases from the `null` Test Set

The failed cases are removed from the results of the `null` test set experiment to evaluate the good case runs. The *membership* and $N_{cluster}$ metrics are evaluated against the datasize (n) and the number of components. The final assessment is made

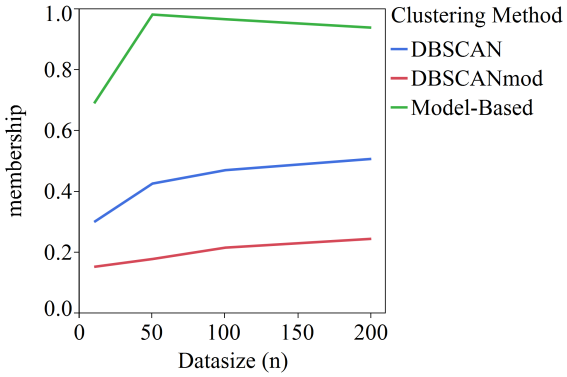


Figure 74: Plot of *membership* accuracy vs. datasize (n) and clustering methods

by considering the *membership* against the feature set and clustering algorithm. The *membership* and $N_{cluster}$ metrics are plotted against the data size (n) in Figures 74 and 75.

membership vs. Data Size

In the analysis of failed cases, the data size was one of the largest contributors. The sparsity of data points make the clustering task difficult. This result is also reflected in Figure 74. The *membership* accuracy is low for a small data size. There is only one valid grouping for all of the `null` datasets, and the failed cases have been removed. Thus, a low *membership* implies that the clustering algorithms identifies more than one cluster or classified some of the points as outliers.

The model-based method performs the best among the three methods and with good margin. One unexpected behavior is that the *membership* accuracy for the model-based method peaks at $n = 50$ and slightly declines thereafter. This could be due to the fact that the higher number of points could make it easier for more than one cluster to be identified.

$N_{cluster}$ vs. Data Size

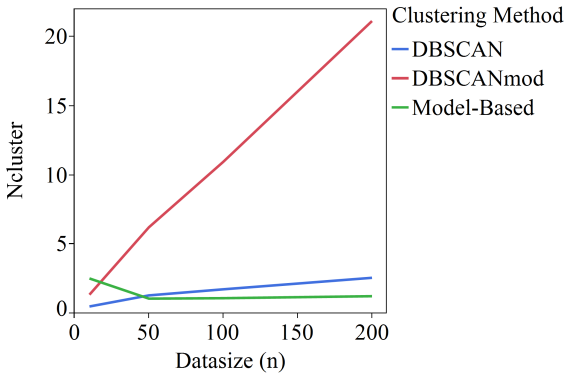


Figure 75: Plot of number of clusters vs. data size (n) and clustering methods

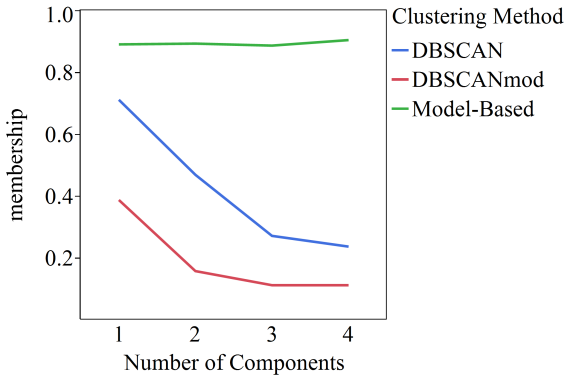


Figure 76: Plot of *membership* accuracy vs. number of components and clustering methods

Next the relationship between $N_{cluster}$ and data size is evaluated in Figure 75. The number of clusters identified by the DBSCANmod method grows linearly with the data size, and the large number of clusters suggests that it is over partitioning the data. While the DBSCAN method does grow as well, it does not grow as quickly. The DBSCANmod uses the DBSCAN as the starting point, so this suggests that the modifications made to improve DBSCAN is actually detrimental. The model-based method identified more clusters with smaller data, and this is consistent with the observation made when evaluating the *membership* results.

membership vs. Number of Components

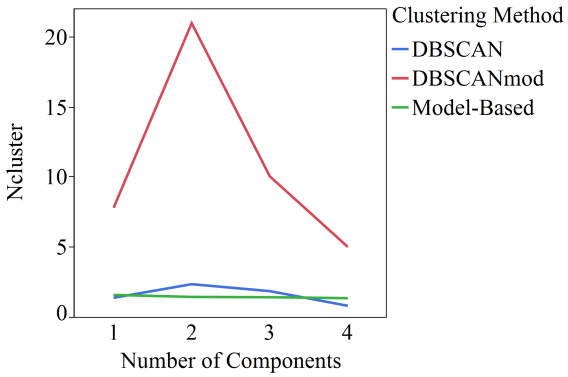


Figure 77: Plot of number of clusters vs. number of components and clustering methods

Based on the results from the failed cases, an increase in the number of components should correspond to worse performance. Figure 76 plots the *membership* against the number of components. DBSCAN and DBSCANmod behave according to expectations. Additional dimensions increase the distance between points when the axes are normalized and when Euclidean distance is used. The model-based method, on the other hand, is not affected by the number of components. This robustness to additional components can be useful in case the number of components cannot be predicted accurately.

$N_{cluster}$ vs. Number of Components

In Figure 77, $N_{cluster}$ is plotted against the number of components. The one striking aspect is that the number of clusters identified by DBSCANmod peaks at 2 and declines. With a closer look at the plot, the method is amplifying the number of clusters identified by DBSCAN. As more clusters are identified by DBSCAN, the modified version is splitting up the clusters into smaller sub-clusters.

Membership vs. Feature Sets

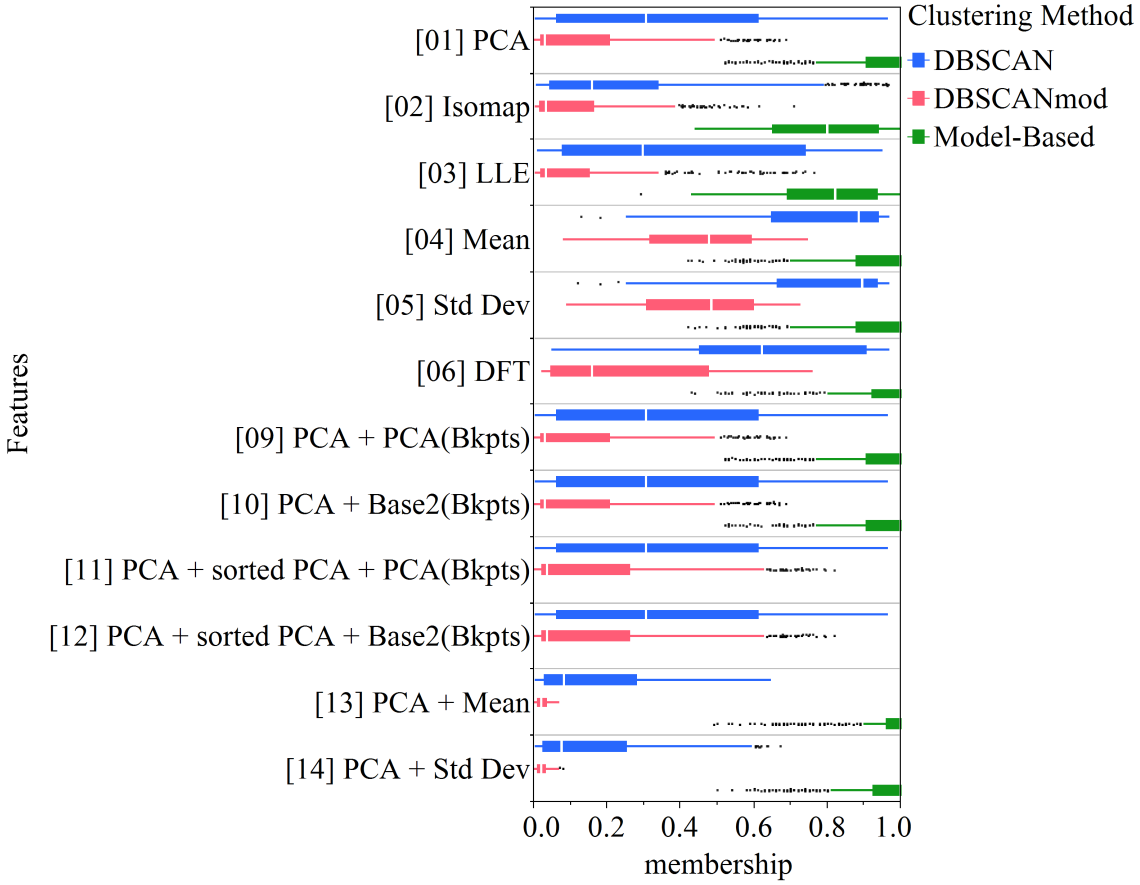


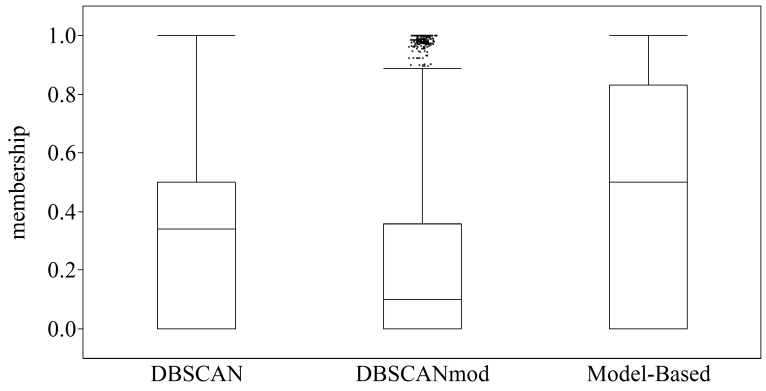
Figure 78: Box plot of *membership* accuracy vs. feature sets and clustering methods for the `null` test set

Finally, the clustering methods and the feature sets are evaluated together using the *membership* accuracy metric in Figure 78. In order to fit the labels, the outlier box plot is transposed, and the *membership* metric is along the bottom axis while the features are stacked on the vertical axis. Each set of features has three box plots, one for each clustering method. The breakpoints features (numbers 7 and 8) form singular clusters and are not meaningful to compare using this test set so there were omitted from the figure.

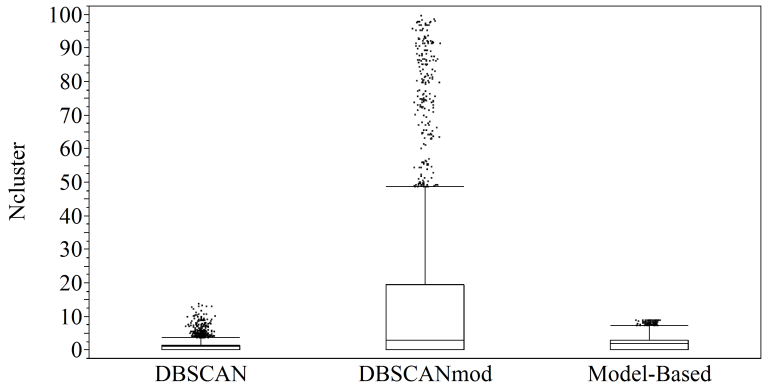
Overall, the model-based method outperforms the other two methods with the `null` test set. The `mclust` algorithm seems to be incompatible with the `sorted PCA` feature (numbers 11 and 12) as there are no box plots for it in the respective rows. This also explains the high failure count in Figure 71 for that feature. The feature sets with breakpoints (numbers 9 and 10) did not improve the results compared to the plain PCA, and this is expected because the breakpoint locations did not change and provided no additional information. The best performing feature sets with the model-based clustering include those with `PCA` (numbers 1, 2, 9, 10, 13, and 14) and DFT. The `mean` and `standard deviation` also performed well for all clustering methods, and this is understandable considering there is only one cluster in the data.

Conclusion of the `null` test set experiment

The `null` test set provided important insights into how the clustering methods react to small data sizes and how the number of components affects the inter-point distance after the data is normalized. The model-based clustering was the clear winner in this first experiment, but there is no clear choice for the feature set to couple with the clustering algorithm yet.



(a) *membership*



(b) $N_{cluster}$

Figure 79: Aggregated results of the clustering experiment for the `line` test set

8.2.3.3 Evaluation of the `line` test set

The `line` test set is composed of a linear line segment with noise that has moving starting and ending points. Because the TS is a line, the breakpoint metrics are not meaningful as it was shown previously, so the features `PCA(breakpoints)` and `Base2(breakpoints)` (numbers 7 and 8) are not considered in this evaluation. The aggregated results are presented in Figure 79.

Membership vs. Feature Sets

Figure 80 shows the *membership* against the features and clustering algorithms. The

model-based clustering method performs the best overall, and the combinations with the PCA feature yield the best results. Although the Mean feature does not yield good clustering results on its own, the combination with the PCA gives the best results. DBSCAN and DBSCANmod have difficulty finding the right clusters. Even though the breakpoints do not contribute meaningful features in this test set, the PCA + Base2(Breakpoints) yields better results than PCA alone. This is probably caused by the removal of the poor performing points, which improved the overall results for PCA + Base2(Breakpoints) feature.

Conclusion of line test set experiment

The combination of PCA + Mean feature set and model-based clustering give the best results for clustering the line test set. The PCA feature set with the model-based clustering follows closely behind in performance.

8.2.3.4 Evaluation of the waves test set

The waves test set contains datasets with TS transformations such as amplitude and phase variation. Similar to the previous set, the goal is to determine if TS transformations can be clustered properly, but these datasets use wave functions instead of linear lines. The aggregated results are presented in Figure 81, and they are slightly better than compared to the ones from the line test set. The overall results of DBSCAN is on par with the model-based clustering algorithm.

Membership vs. Feature Sets

Overall, the model-based clustering using Standard Deviation feature had the best results, followed by the Mean feature. The other combinations of clustering

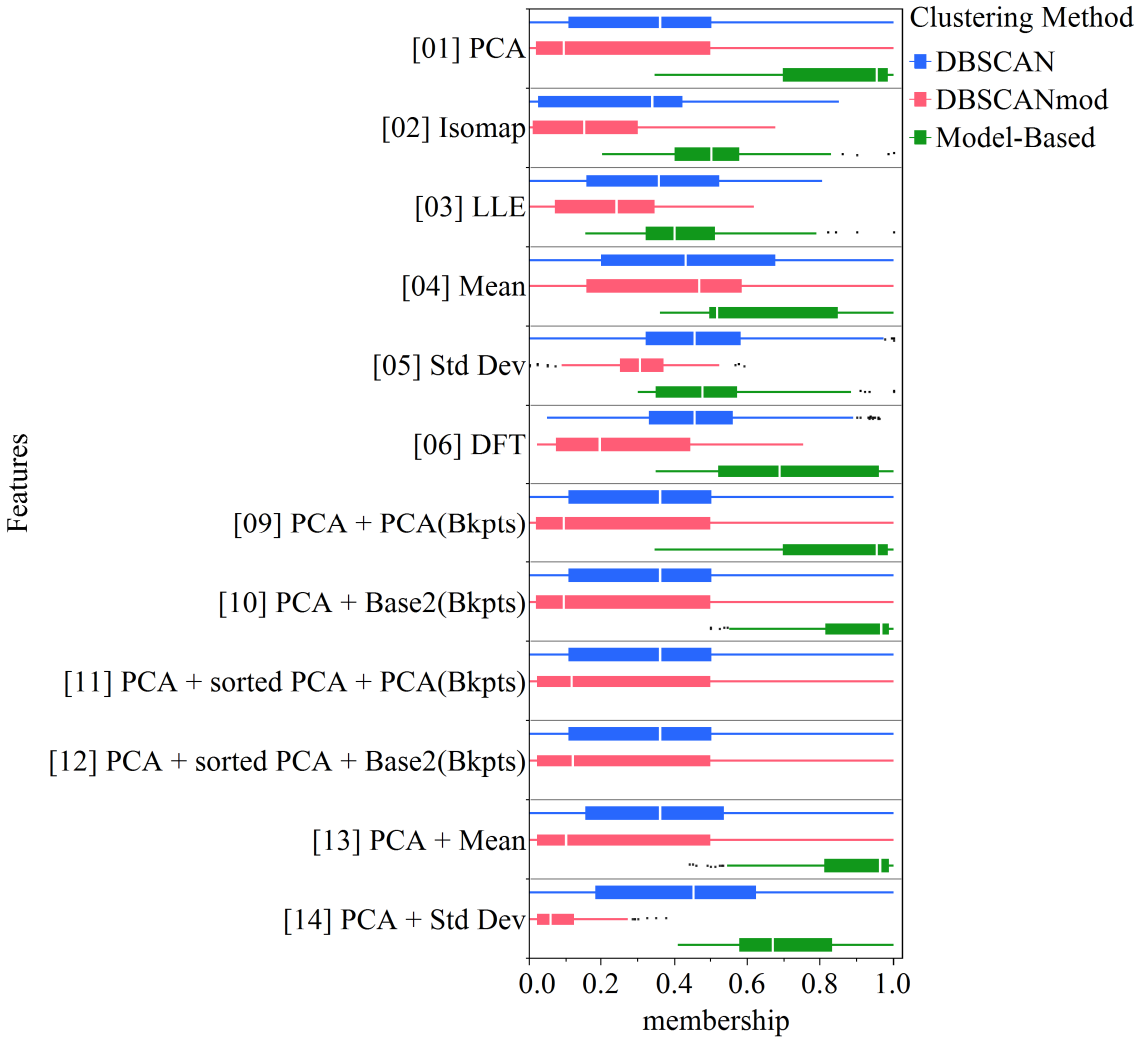
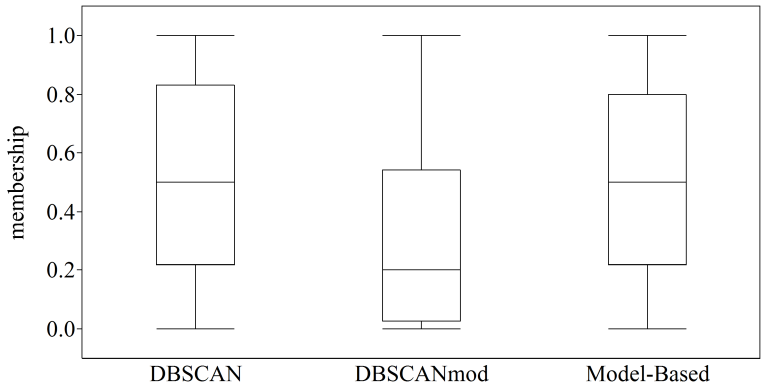
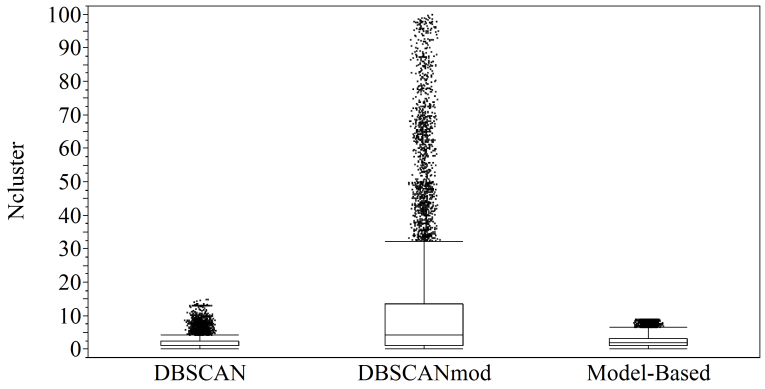


Figure 80: Plot of *membership* accuracy vs. feature sets and clustering algorithms for the `line` datasets



(a) *membership*



(b) $N_{cluster}$

Figure 81: Aggregated results of the clustering experiment for the `waves` test set

method and feature sets produce mediocre results. The increase in amplitude in the amp-smooth essentially improves the signal-to-noise ratio because the data is normalized before clustering so the effective noise in the data shrinks linearly with amplitude and the mean does not change. This will appear as a single cluster in those two feature spaces. And the others should not have that much difficulty with this one as well. In the phase dataset, the mean and standard deviation do not change so the points will appear as one cluster in the two feature spaces. However, in the other feature spaces, this could be more challenging, especially if there are discontinuities like the square and sawtooth waves as was seen in the PA Experiment. Similar conclusions can be made about the phase-amp dataset.

The breakpoints features performed better than Isomap and LLE features, but they still performed poorly. It provided a slight improvement to the PCA when combined together.

Conclusion of the waves test set

Overall, the Standard Deviation and Mean features with the model-based clustering algorithm produces the best results for the waves test set.

8.2.3.5 Evaluation of the shapes test set

The shapes test set is composed of datasets with distinctly different shapes. The goal is to determine if the clustering methods with the features can discriminate between these shapes. The aggregated results are presented in Figure 83.

Based on Figure 83, DBSCAN is the overall best method for clustering TS data in the shapes test set. Because the data is created using discrete settings, the

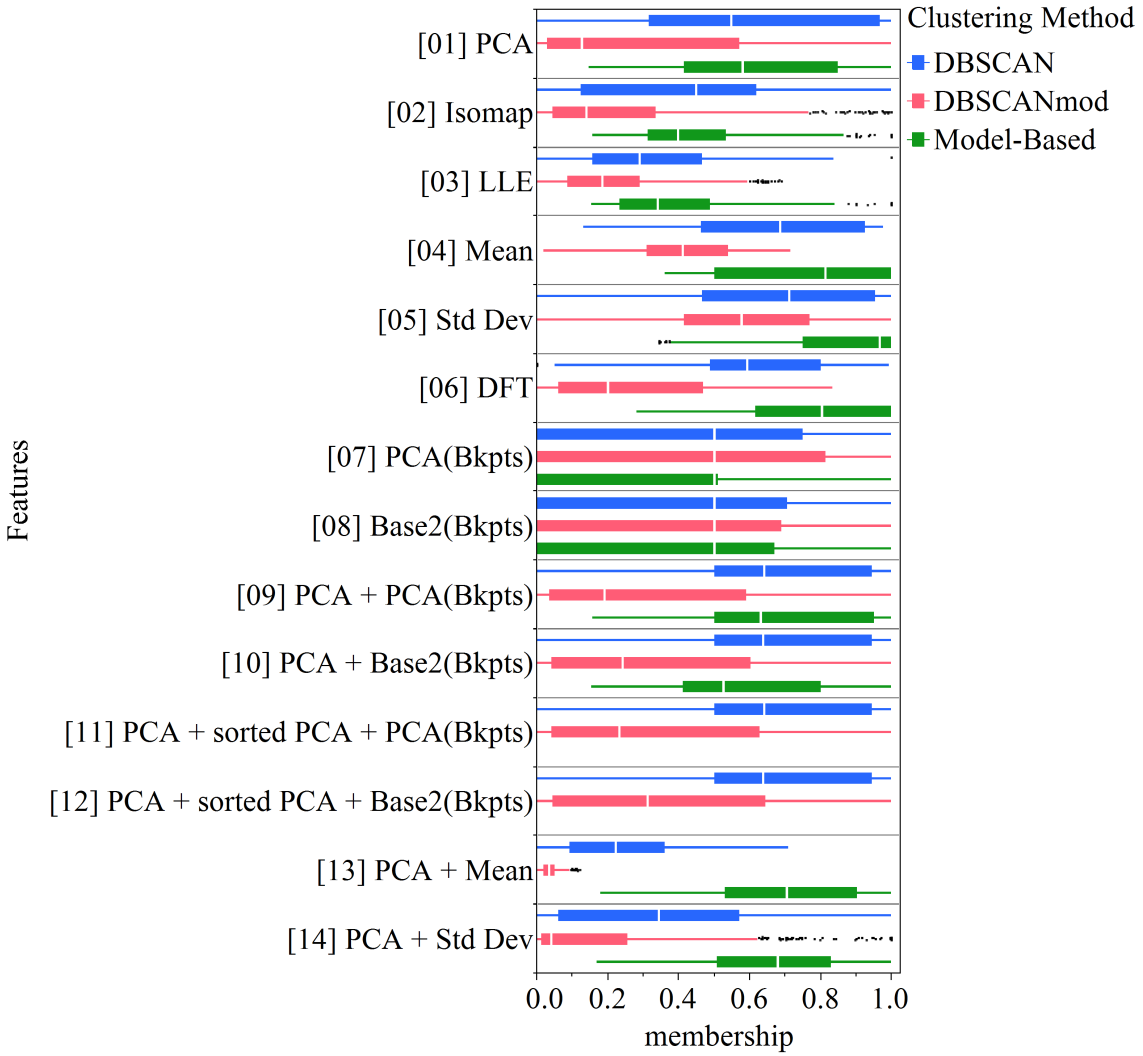
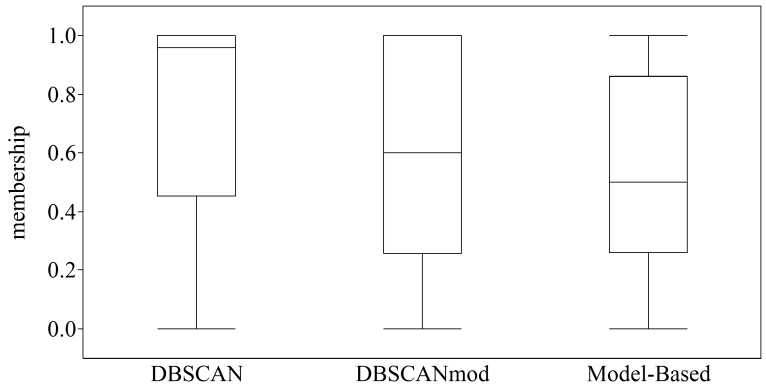
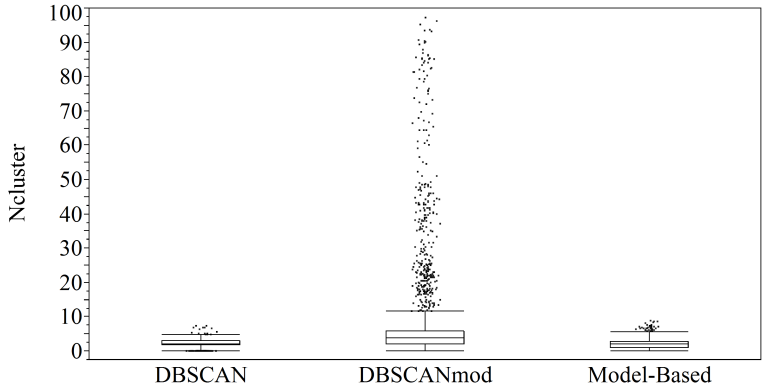


Figure 82: Plot of *membership* accuracy vs. feature sets and clustering algorithms for the `waves` test set



(a) *membership*



(b) $N_{cluster}$

Figure 83: Aggregated results of the clustering experiment for the `shapes` test set

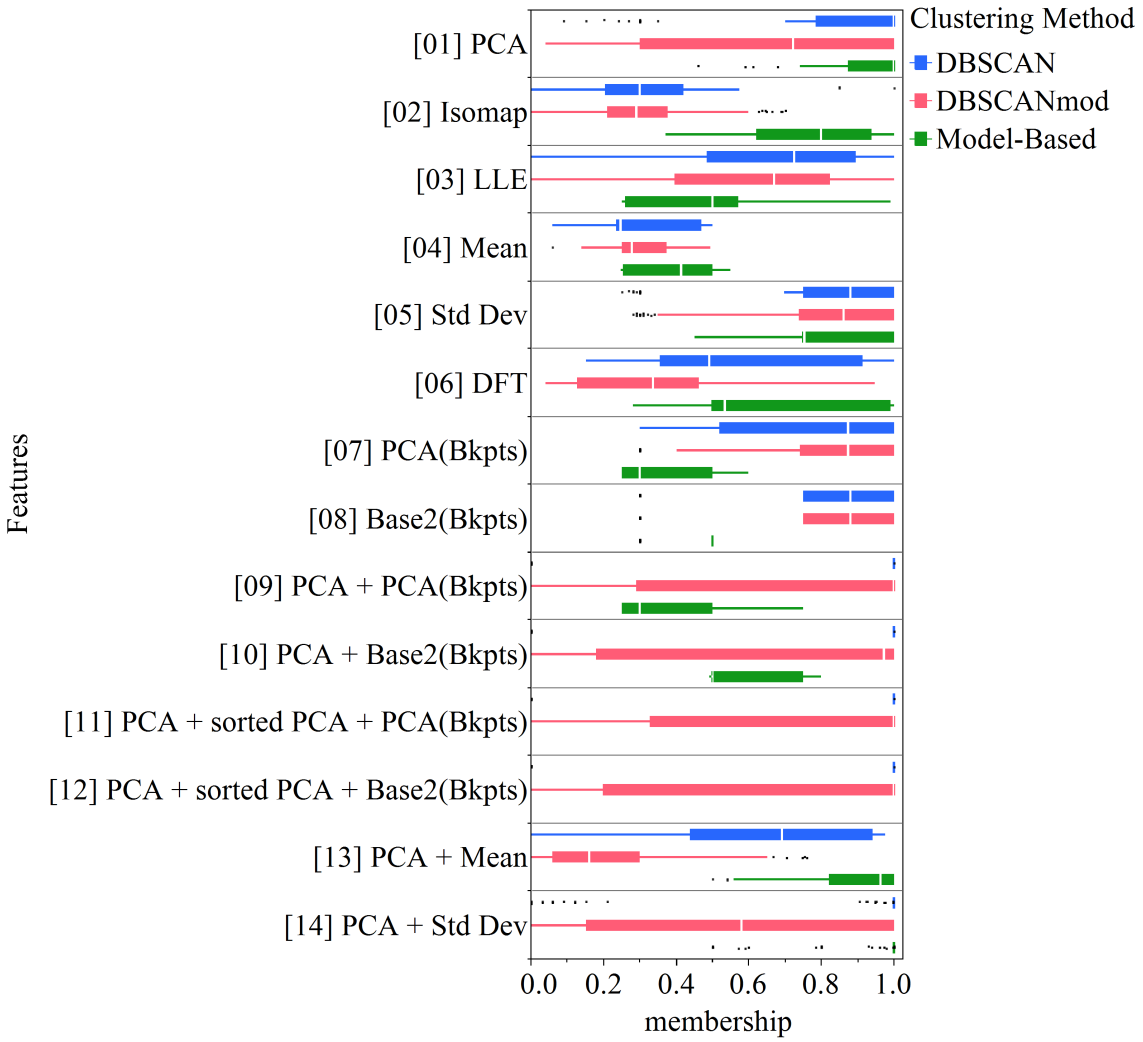


Figure 84: Plot of *membership* accuracy vs. feature sets and clustering algorithms for the `shapes` test set

points should cluster tightly around each other for the same settings. This situation is ideal for density-based clustering because the correct points are already in the same neighborhood. On the other hand, the model-based method seems to have difficult dealing with these tight clusters and performs poorly.

Membership vs. Feature Sets

When the *membership* metric is broken down to show the feature sets as shown in Figure 84, the performance of DBSCAN becomes even clearer. DBSCAN and

the model-based method with the PCA feature provide good results, but when DBSCAN is paired with PCA and a breakpoints feature (PCA(breakpoints) and Base2(breakpoints)), the accuracy is close to perfect. On the other hand, the model-based clustering with the same set of features actually yields a worse result than PCA alone. The PCA + Standard Deviation feature set with both DBSCAN and model-based clustering also produced good results.

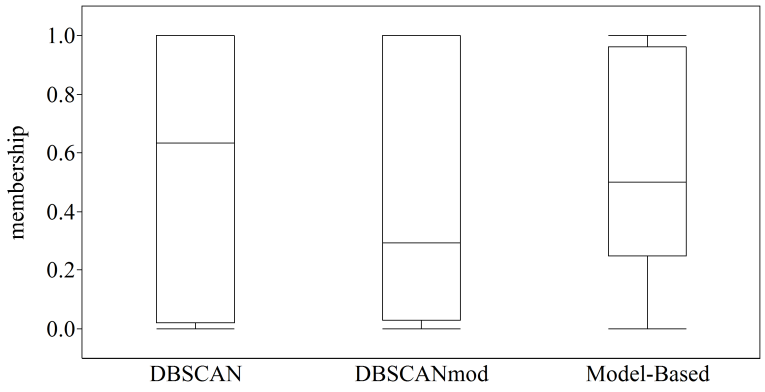
Conclusion of shapes test set

The experiments using the shapes test set illustrated the importance of choosing the correct clustering algorithm with the correct set of features. An addition of a feature can improve or worsen the clustering results, and care is needed in the selection process. The breakpoints features are not useful on their own, but it can be very useful when used together with other features. Also, the model-based clustering showed good results when the data was noisy and spread out, such as in the null test set experiments, but it has difficulty clustering highly localized set of points.

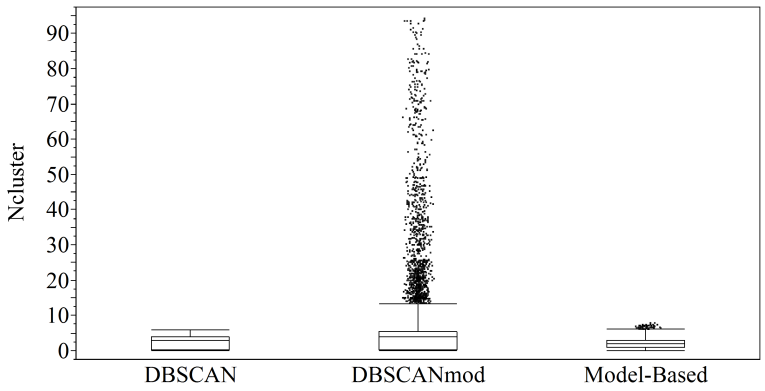
8.2.3.6 Evaluation of the discrete test set

The datasets in the discrete test set are made with discrete settings for the various wave functions. Same as shapes test set, the goal is to see if the clustering methods correctly categorize the TS with different attribute settings. The overall results are provided as box charts in Figure 85.

The overall results for *membership* and $N_{cluster}$ puts DBSCAN better than the rest, but the median results do not look promising and the variability of the results is large.



(a) *membership*



(b) $N_{cluster}$

Figure 85: Aggregated results of the clustering experiment for the `discrete` test set

Membership vs. Feature Sets

As shown in Figure 86, the `PCA` feature with model-based clustering has the best result, and `PCA + Mean` and `PCA + Standard Deviation` are close seconds. The box plot in Figure 85 gives the impression that the overall results of the model-based clustering has a large spread, but when the results are broken down into features, the wide range is due to a mix of good results and poor results.

The DBSCAN and DBSCANmod methods on the other hand have wide ranges in both figures. The TS shapes should appear as tight groups just like in the `shapes` test set because the parameters are varied discretely, and the density-based methods should have performed better than the model-based clustering.

Conclusion of `discrete` test set

Overall, the model-based clustering method with `PCA` and combinations with `PCA` have the best results. The density-based methods have a difficult time with the `discrete` test set.

8.2.3.7 Evaluation of the `outlier` test set

The `outlier` test set contains a handful of TS that clearly do not belong to a cluster and thus is an outlier. The aim of this test set is to evaluate how the different clustering algorithms handle outliers, which is a concern when creating groups of points for piecewise linear regressions. The overall results are shown in Figure 87.

One of the key features of density-based clustering methods is that it can identify outliers that are isolated. At first glance this would appear to be the reason why DBSCAN is performing the best. However, it is important to note that the test data

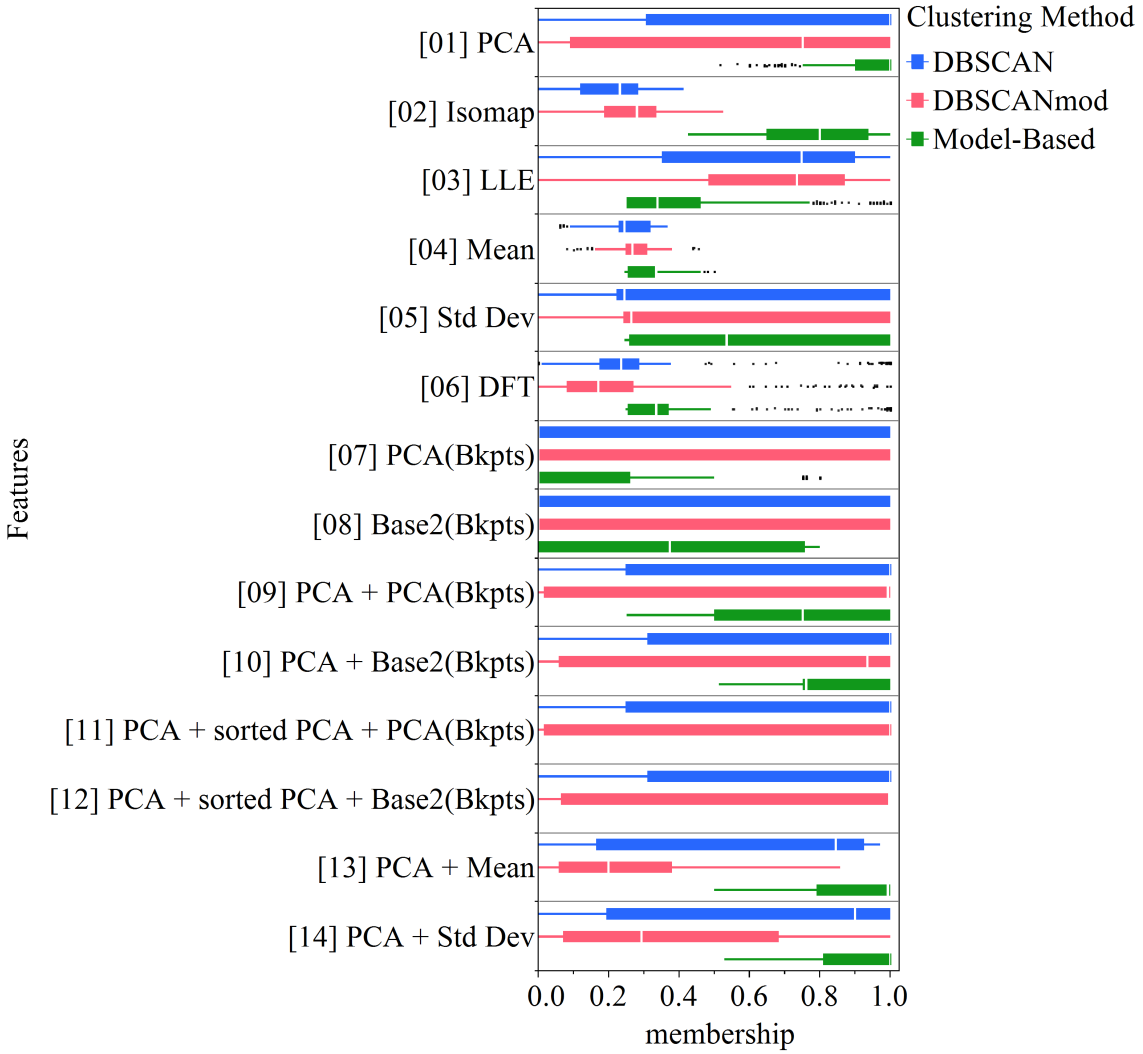
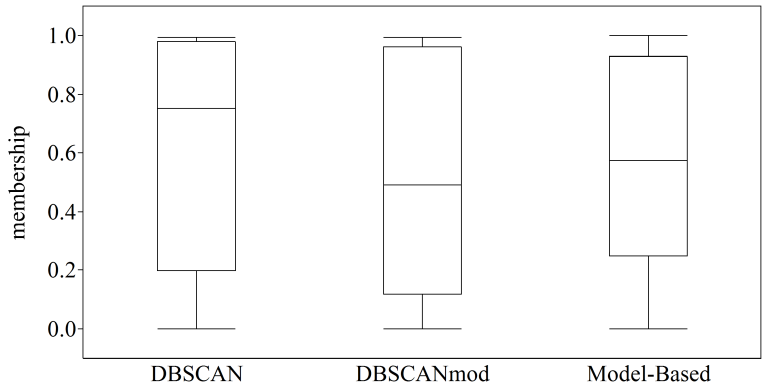
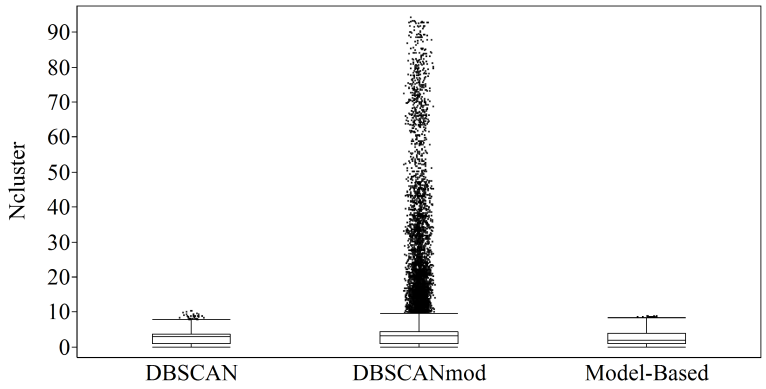


Figure 86: Plot of *membership* accuracy vs. feature sets and clustering algorithms for the `discrete` test set



(a) *membership*



(b) $N_{cluster}$

Figure 87: Aggregated results of the clustering experiment for the `outlier` test set

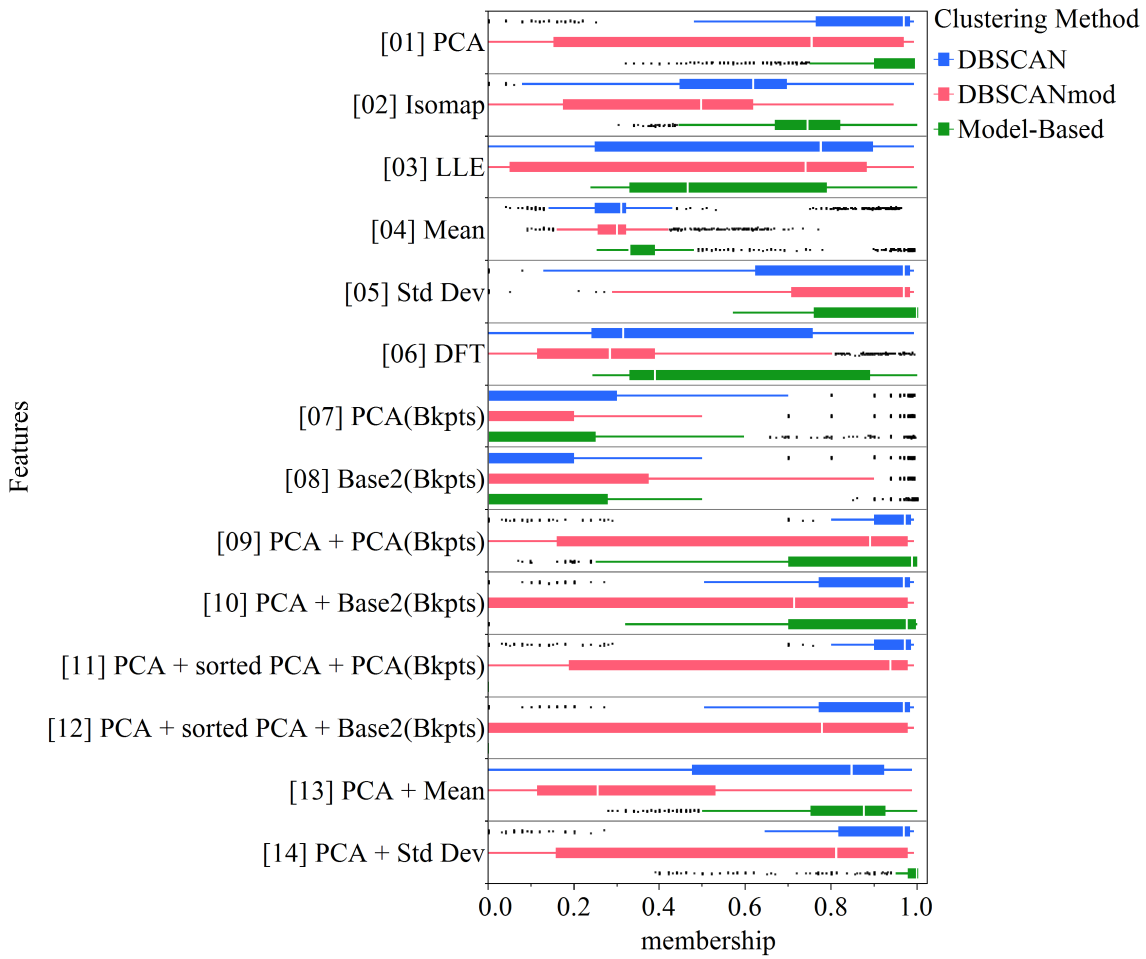
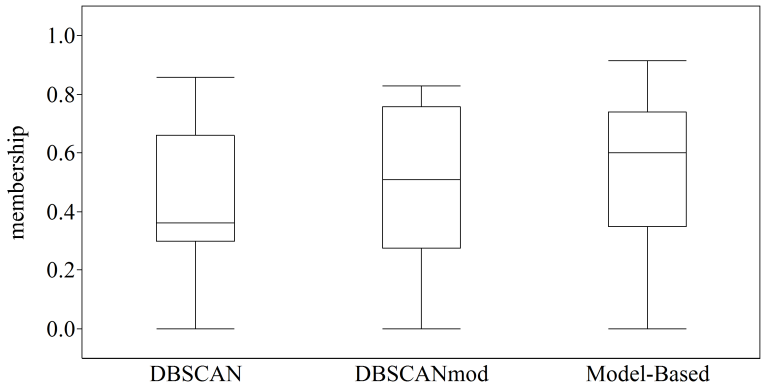


Figure 88: Plot of *membership* accuracy vs. feature sets and clustering algorithms for the `outlier` test set

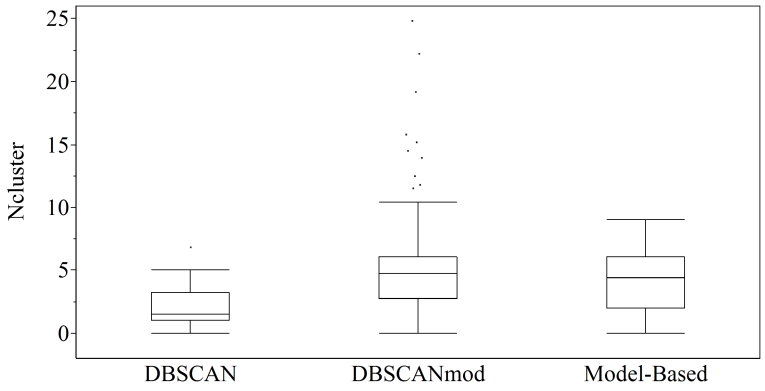
is practically the same as the `shapes` test set, and in that experiment, DBSCAN was already the best of the three methods.

Membership vs. Feature Sets

The overall trend with DBSCAN across the different feature sets is the same as compared to the results from the `shapes` test set. However, the performance of DBSCAN actually degrades compared in the `outlier` test set, which suggests that it was not successful in identifying the outliers, at least in some situations.



(a) *membership*



(b) $N_{cluster}$

Figure 89: Aggregated results of the clustering experiment for the `ts` test set

8.2.3.8 Evaluation of the `ts` test set

The `ts` test set emulates the FAMOS model data using a function that varies the TS shape. This test set is the most complex of the canonical test sets as it has different shapes of TS that vary smoothly, and the breakpoints are different across the TS. The number of segments also varies depending on the shape.

The aggregated results in Figure 89 show the model-based clustering having a slightly higher performance, but the median performance for *membership* is around 60%. Looking at the $N_{cluster}$ chart, the correct number of clusters is 6, but DBSCAN

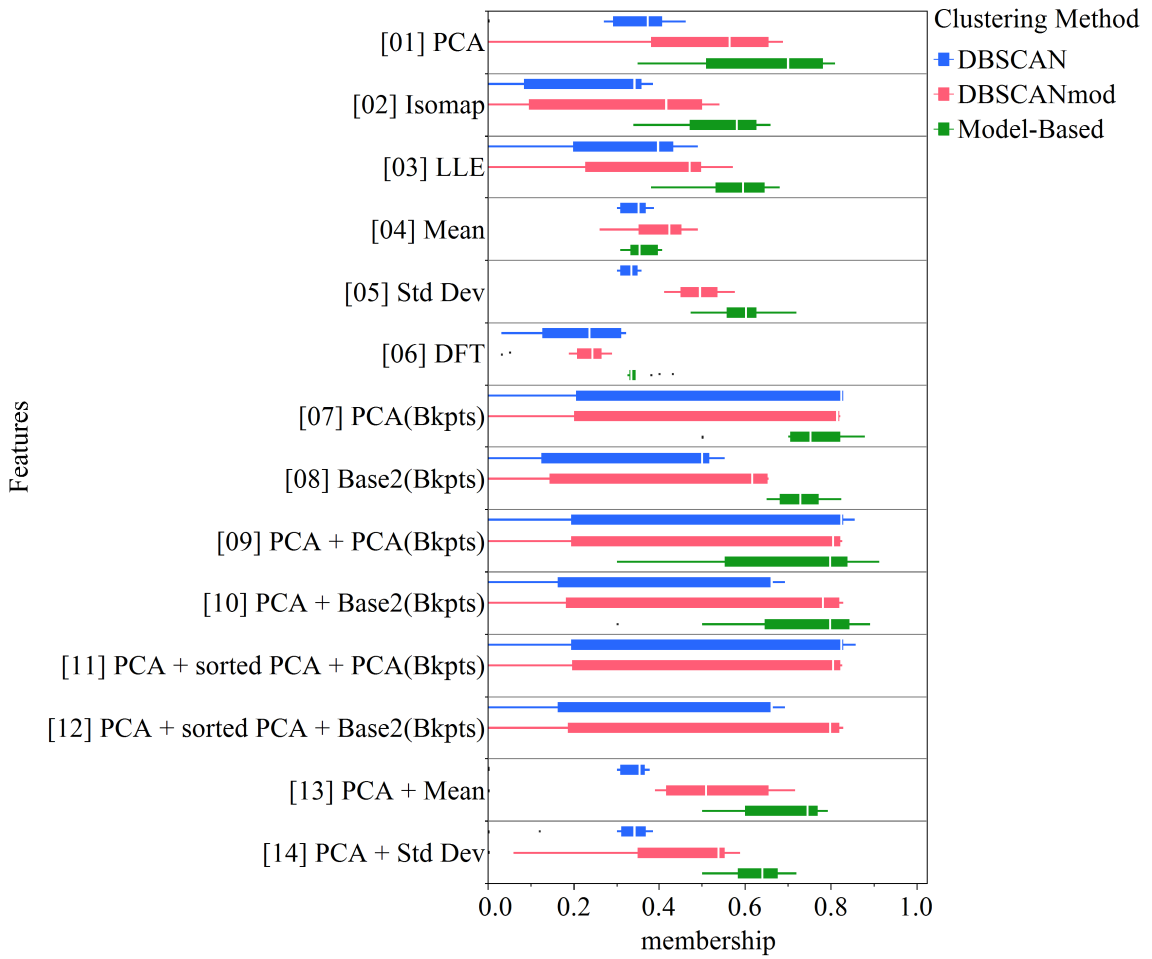


Figure 90: Plot of *membership* accuracy vs. feature sets and clustering algorithms for the `ts` test set

identifies around 3 in the median. This is because the clusters are organized as lines and different clusters are intersecting in the feature space. The median value from DBSCANmod, which was designed to solve this problem, is closer to the correct number of clusters, although it still has a problem of creating too many clusters.

Membership vs. Feature Sets

Overall, the model-based clustering is the best performing of the methods if the feature sets including `sorted PCA` are ignored as shown in Figure 90. The *membership* accuracy seems to hit a ceiling around 0.8 for all features and methods, and this may

be a difference in interpretation of what the correct answer should be.

The DBSCAN and DBSCANmod show good median values when using feature sets that have the breakpoints features. However, the IQR is very wide, and the whiskers of the box plot extend out to 0. This suggests that there is a bimodal distribution with a fair amount of points clustered at 0.8 and 0. With most feature sets in Figure 90, DBSCANmod outperforms DBSCAN, and this is most likely because some of the clusters are touching, and DBSCAN groups them together (as it was discussed in Section 4.4.1) while DBSCANmod separates those clusters.

Next, individual feature sets are evaluated. The simple metrics of `Mean` and `Standard Deviation` perform poorly as there are complex TS transformations that are occurring that cannot be captured with a simple statistic. `DFT` also performs poorly, and in fact it is the worst. This is somewhat unexpected considering how well it performed in the `null` test set and decently in the other test sets. This maybe caused by the number of components that are used, and for the DFT to properly capture the variations in the TS shapes, more components may be necessary. The manifold learning methods (`Isomap` and `LLE`) do not perform that well, and they are on par with the `Standard Deviation` feature. `PCA` with model-based clustering performs well, and the breakpoints features, particularly `PCA(breakpoints)`, perform really well as standalone features.

Figure 90 shows that combining other features with `PCA` does improve the results over just using only `PCA` but with mixed effects. With DBSCAN and DBSCANmod with `PCA` and breakpoint locations (feature sets 9, 10, 11, and 12), the median is improved but the IQR becomes wider, and with `PCA + Mean` and `PCA +`

`Standard Deviation`, the results do not improve. Model-based clustering with `PCA + PCA(Breakpoints)` and `PCA + Base2(Breakpoints)` improves the median compared to when the features are used separately, but the ranges on the box plots also widen.

Conclusion of `ts` test set experiment

The `ts` test set experiment showed that the model-based method to be the best performing clustering method. It also highlighted the descriptive power of using breakpoints as a feature.

8.2.3.9 Conclusion of the Clustering Experiment

The combination of fourteen sets of features and three clustering algorithms were tested using canonical datasets. The experiments were designed to determine if any of the combinations of methods and features would yield PLRDGs. The test sets identified different weaknesses in each of the methods and the features.

The model-based clustering performed the best overall. It is robust to some level of noise, and it finds clusters in straight lines. However, it fails to clusters that are tightly grouped together in the feature space. Because the method tries to fit a Gaussian distribution to the clusters, some noise or variation between points are needed. On the other hand, DBSCAN performed well where the model-based clustering failed, but because it grows clusters by adding nearby points, it groups clusters that are touching as a single cluster. DBSCANmod tries to fix this problem, but it is prone to creating too many clusters when there is noise.

Among the dimensional reduction methods, `PCA` consistently performs better

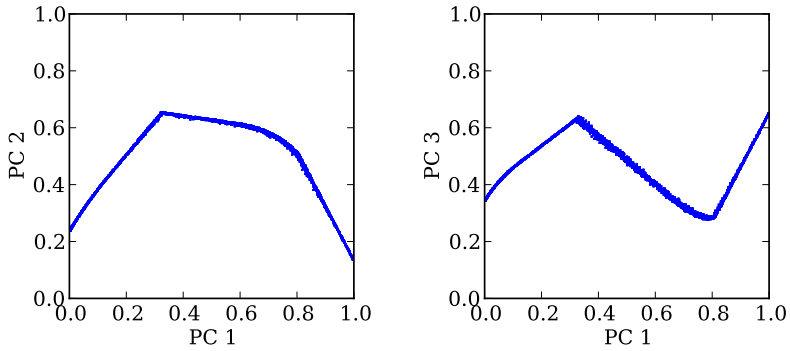
than Isomap and LLE. Perhaps the data was too simple to warrant the more powerful nonlinear dimensional reduction techniques. The statistics of the TS, Mean and Standard Deviation, performed well with the null test set, but they had mediocre results otherwise. However, when they are combined with the PCA feature, they generally improved the *membership* accuracy. The DFT feature performed well with the null test set, but otherwise was one of the weaker features for clustering, especially in the case of the ts test set.

The PCA(Breakpoints) and Base2(Breakpoints) features performed well when it mattered. Most of the datasets were ill-suited for the breakpoints features because they either did not have any breakpoints or the breakpoint locations did not move, but when breakpoints did exist and shift, like in the ts test set, they performed really well. In fact, it was one of the best performing features to create clusters. The difference in performance between PCA(Breakpoints) and Base2(Breakpoints) was minor. To maintain the simplicity of the method, using PCA(Breakpoints) is preferred so that the same PCA method is applied to the data and the breakpoints.

Model-based clustering with PCA + PCA(Breakpoints) is the best combination to cluster TS datasets to create PLRDGs.

8.2.4 SMARTS Feasibility Experiment

The feasibility of the SMARTS methodology is tested using the demo-ts dataset. It consists of 4410 rows of TS, which are 350 time steps long. The methodology will be demonstrated step by step along with notes on how it is implemented in



(a) First vs. Second PCs (b) First vs. Third PCs

Figure 91: Plot of the first three principal components of the `demo-ts` data

the Python source code. Because most of the methods are implemented in the `multivariate_piecewise_linear_regression_7.py` file, references to class variables and methods will be from this file unless explicitly called out.

Step 1: Create Z Domain

The first step is to create the intermediate Z domain using the dimensionality reduction technique PCA. The first three PCs are plotted in Figure 91. Although the data has meaningful structures in the 2nd and third dimensions instead of random scatter, the TS shapes can be organized using just the first PC as shown in Figure 92, so only the first PC will be used to create the Z domain.

This step is implemented in the `perform_pca()` method, which is a part of the `PrepOutputData` class. Because the results from the PCA are also used later in the clustering step, the other PCs are kept as well. To determine how many PCs are kept, the PA method is used, and this results in 5 PCs. Although only the first PC is used in Step 1, the remaining four PCs are carried over until Step 3.

Step 2: $X \rightarrow Z$ Regression

The input domain X is mapped to the intermediate Z domain using NN. The inputs

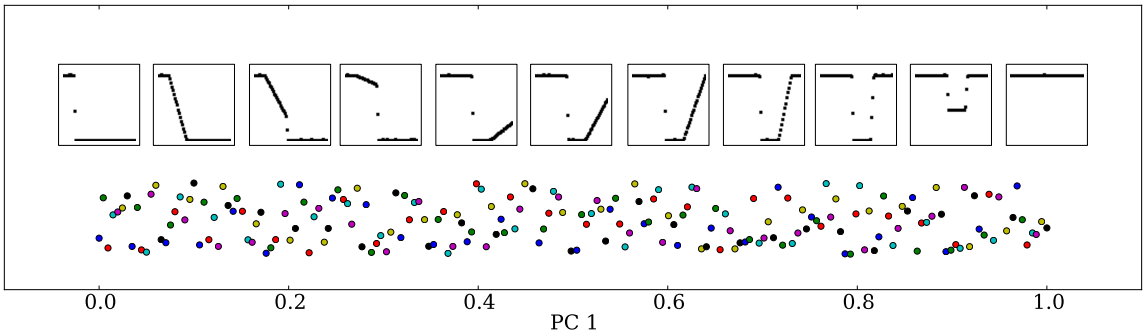


Figure 92: `demo-ts` output data organized along the first PC. The points are jittered to avoid overlaps, and the plots of TS are associated with location along the PC.

are fit to the Z domain values (the first PC) calculated in Step 1. The NN settings are 1 hidden layer with 5 nodes. The best of 10 training sessions is used, and each NN is trained for a maximum of 500 epochs.

This step is implemented by the `_x_to_z2()` method, which is a part of the `multivariatePiecewiseRegression` class. This is also where the user can specify if the regression is to capture the median, lower or upper line. If the median is chosen, then the regression will use all the points in the Z domain to create the NN regression. If the lower or upper option is chosen, then the function calculates the three sigma value using the z values of the repetitions for each set of inputs, and uses this new z value for the regression.

Step 3a: Reduce the Data

The next step is to reduce the data by combining similar shapes. The TS are binned in the Z domain, which in this case is one-dimensional along the first PC. Figure 93 shows the distribution of points after the data has been organized. There is a large amount of TS at the two extremes, which are the "cliff" shape and the flat line. Because the goal of the $Z \rightarrow Y$ regression is to capture the shape and its

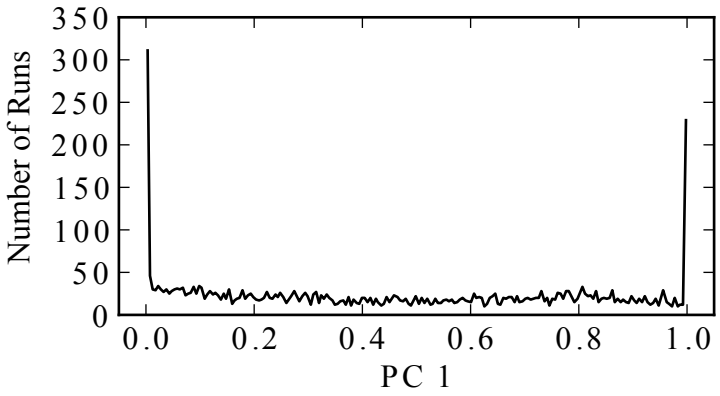


Figure 93: Distribution of points along the first PC

variations and not to fit the data, there is no need to retain all the TS because they are redundant shapes.

The data is binned along the first PC, and initially, 200 bins are targeted. A minimum threshold of 10 data points per bin is used to ensure that each bin has a sufficient number of points. The default is 20 points, but because the dataset is small, a lower bin size is used here. Then the median of the TS in each bin is taken as the representative TS shape. The median is used because it is less influenced by outliers, but depending on what is sought, different statistics can be used such as the mean. Instead of using the statistics, a random TS can be selected as well. It was discussed in Section 6.4 of how the upper and lower bounds can be captures in two ways, and one of them was implemented in Step 2. The other approach is to implement it here by choosing a different statistics such as the minimum, maximum or some quantile.

Several other values are prepared here for the later steps. The PC values that were calculated in Step 1 are averaged across the bin to create a representative value. There were five that were saved from Step 1, and they will be used for the clustering step. The first PC value is also used for creating the PLR. The noise in the TS is also

estimated using the bins. Because there are more than 20 TS per bin, the median of each time step (instead of the moving median) is used to estimate a smooth version of the TS per bin. Then the difference is taken from each TS with the smoothed TS. The standard deviation of the residuals is recorded as the threshold setting which will be used by the TS segmentation step.

After this process, the 44,100 rows of TS in the original `demo-ts` are reduced to 196. Because some bins contain less than the threshold of 10 points, they are merged with its neighbors.

This step is implemented in the `reduce_data()` method, which is a part of the `PrepOutputData` class. The class can receive the data as an input or import the data using the file locations of the output data. The number of bins and bin size can be specified as well.

Step 3b: Determine the Breakpoints of the TS

The estimated noise from the previous step is used as the threshold to segment the TS. A hybrid segmentation algorithm is applied to each TS, and the breakpoint locations and the number of segments are recorded. Figure 94 shows examples of the TS segmentation.

The TS segmentation is performed by the `_initial_segmentation()` method, which is a part of the `multivariatePiecewiseRegression` class, and this method loops through each TS in the reduced TS dataset to apply hybrid segmentation algorithm, which is implemented by the `hybrid_aggr()` method in the `ts_segments.py` file.

Step 3c: Cluster the Data

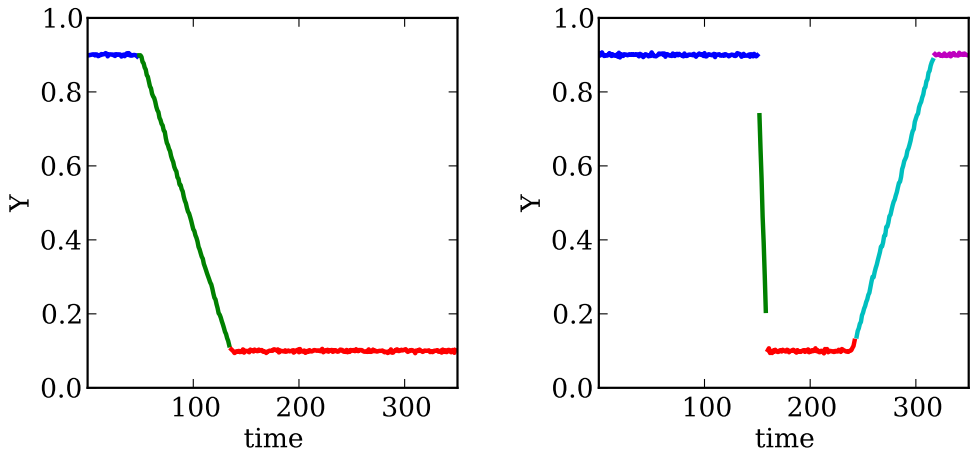


Figure 94: Examples of segmented TS

The next step is to cluster the data into PLRDGs. The features that will be used are the PCs from Step 3a and the breakpoints from Step 3b. The breakpoints are transformed using PCA so they can be used as a feature. Because the TS have different number of breakpoints, those with fewer breakpoints are padded with zeros. A matrix of breakpoints is created where the number of rows is the same as the number of TS and the number of columns is the maximum number of segments, so in this case, the matrix is 196rows $\times$ 5columns. The first PC is kept from applying PCA to the matrix of breakpoints.

Based on the results from the Feature Selection and Clustering Experiment, the `PCA + PCA(Breakpoints)` feature set will be used. Five PCs were kept from Step 1 and reduced in Step 3a, and one PC from the breakpoints was just calculated. This creates a matrix with six features. Model-based clustering is applied to this matrix. The initial result of the clustering is presented in Figure 95, which plots the data with respect to the first PC as calculated in Step 3a and the first PC of the breakpoints. The numbers indicate the group label with which the point is associated.

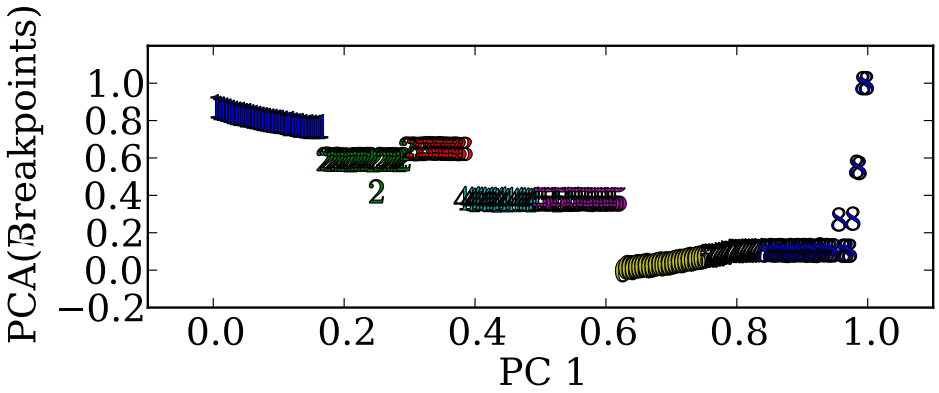


Figure 95: First PC of the data vs. the first PC of the breakpoints

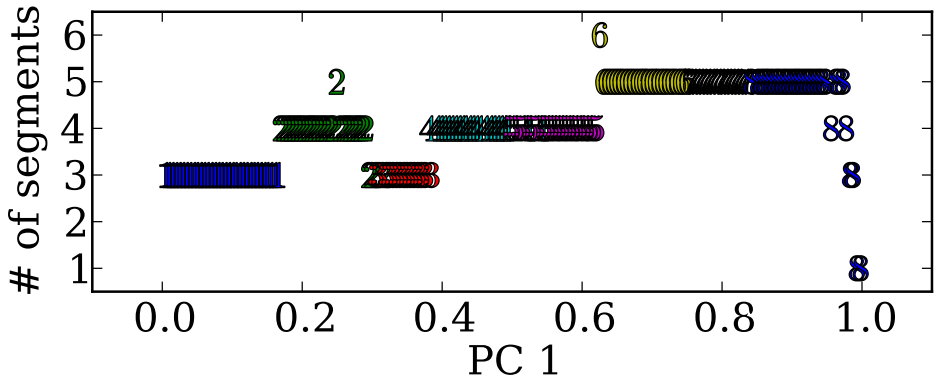


Figure 96: First PC of the data vs. the number of segments

Figure 96 shows a similar view but with the y-axis being the number of segments. Points of a single group should lie on the same segment number, but group 8 is split across different number of segments at the right hand side. The clustering process is not perfect, and the groups need refinement.

A simple routine is implemented to check and fix the number of segments in each group. The majority rule is implemented if there are different numbers of segments in a group. Overlaps are also resolved, and the overlapped region is incorporated into the group with a higher number of segments. Figure 97 shows the data organized in first PC and number of segments after the refinement procedure.

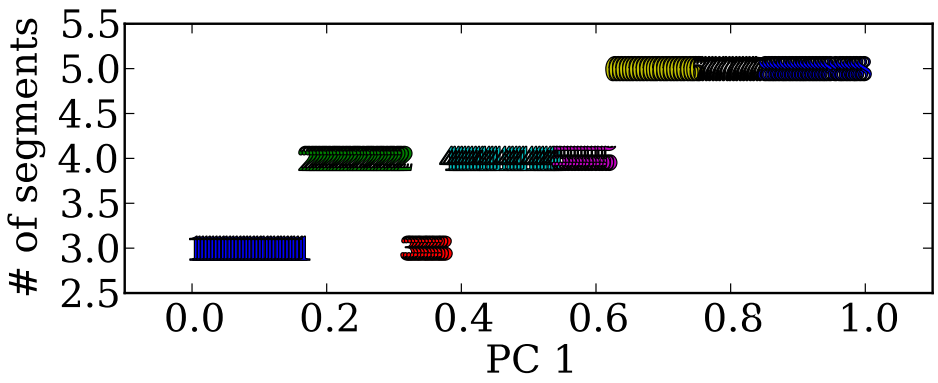


Figure 97: First PC of the data vs. the number of segments after refinement

One thing to note about model-based clustering is that it has a tendency to breakup a line of points if there is little noise. For example, in Figure 97, groups 4 and 5 (between 0.4 and 0.6 on the x-axis) should technically be a single group. This does not pose a problem for the regression.

The `PCA(Breakpoints)` feature is calculated by the `_extract_PCA_Breakpoints()` method. It creates a matrix of breakpoints as described above, applies PCA, and saves the normalized first PC. The clustering is performed by the `_group_data()` method, which in turn calls the `_model_based_clustering()` method in the `ApplyClustering.py` file. This method is a wrapper for the R package `mclust`. The `_clean_up_groups_2()` and `_check_group_nseg()` methods are responsible for resolving the overlaps and ensuring that TS within each group has the same number of segments.

Step 3d: $Z \rightarrow Y$ Regression

Now that the points are grouped into PLRDGs, a PLR model can be fit to each of them. For each group, the segments are fit separately with a first order linear regression, and the breakpoint locations are also fit in the same way. So, for a group

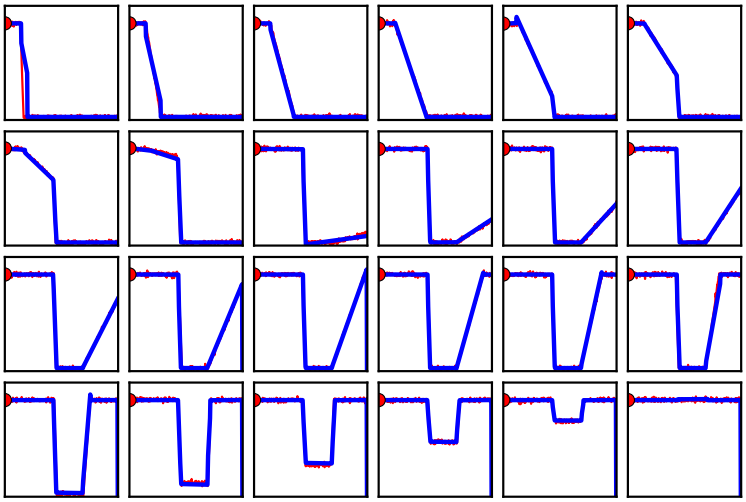


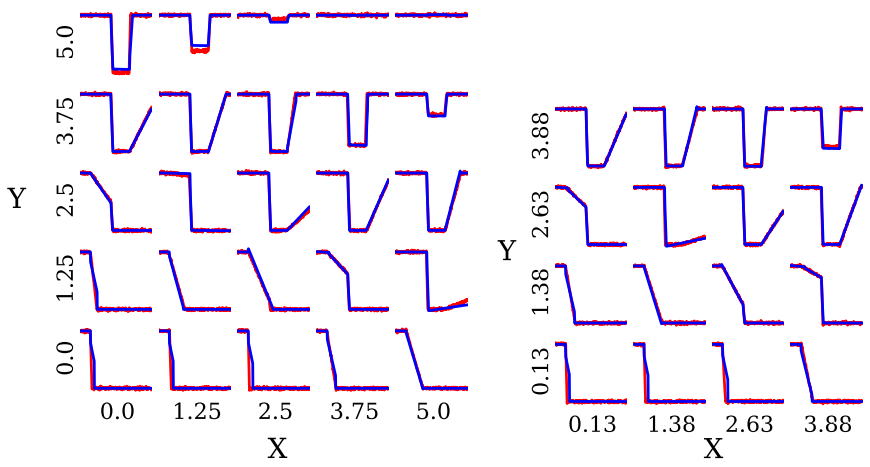
Figure 98: $Z \rightarrow Y$ regression plotted with the TS shape from the original data with the closest Z domain value. Starting from the top-left, the plots are ordered from left to right and continue on successive rows.

with three segments, a total of five linear regressions are fit, and the start and end points are also recorded for bookkeeping purposes. In order connect the Z domain to the group labels, a mapping function is created. Figure 98 plots the shapes of the $Z \rightarrow Y$ regression in blue as it sweeps across the Z domain. The TS from the original data with the closest Z domain value is also plotted in red, but it is difficult to distinguish between the two because of the close fit.

This regression step is performed by the `_create_regressions()` method. It calls the `_create_regressions_segments()` method to fit the segments of each group, and it calls the `_create_regressions_breakpoints()` method for the breakpoints. The mapping is created with Python's `lambda` function and the `_give_me_vals()` method.

Using the PLR

Once the SMARTS methodology is completed, the regression can be used to recreate



(a) Model Fit Check (b) Model Representation Check

Figure 99: SMARTS model output plotted in blue and actual data in red

and interpolate TS data. Figure 99a shows the prediction by the SMARTS model in blue with the original data in red. For completeness, Figure 99b shows the estimates by the SMARTS model for points that were not included in the original `demo-ts` dataset.

To use the PLR, the `predict()` method is used. The input variables and the time steps are passed to the method, and the corresponding TS data is returned.

Conclusion of the Feasibility Experiment

The SMARTS methodology was applied to the `demo-ts` dataset, which uses a TS function that was derived from simulation data. The data has little noise or variability between TS with the same inputs, and the TS have very prominent features. Details on the implementation and notes on the programming code were also provided. Through this experiment, it was shown that the SMARTS model was able to capture the corners of the TS and matched the original data very closely.

Table 21: List of regression methods used for the scalability experiment

Method	Model Name	Settings	Notes
Mean	`mean`		Average of the output data
Linear Regression	`linreg2`		2nd order linear regression with interaction terms
Neural Networks	`nn20`	1 hidden layer, 20 nodes	On entire dataset
	`nn20-ts`	1 hidden layer, 20 nodes	On each time step
	`nn20-ts (smooth)`	1 hidden layer, 20 nodes	`nn20-ts` smoothed with moving median
SMARTS	`SMARTS`	1000 bins	

8.2.5 SMARTS Scalability Experiment

To test the scalability of the SMARTS methodology, regressions using competitive methods were performed on data generated by the MRFAMOS model. The data was previously presented in Section 8.1.3. The regression methods and their settings are listed in Table 21. The mean is listed as a baseline to compare the other results, and the regression methods need to have better results than the mean. The linear regression is a second order linear regression with interactions, and this is basically the response surface method. Several variants of the neural network (NN) is used. The `nn20` fits a NN to the entire dataset, while the `nn20-ts` fits a NN to each time step. For the MRFAMOS data, this amounts to 850 regressions. The `nn20-ts (smooth)` applies a moving median smoother on the output of `nn20-ts` because the output is very noisy. Finally, the SMARTS methodology is used with 1000 bins.

8.2.5.1 *Metrics of Interest*

In comparing the different regression methods, there are four metrics of interest. The first is the time it takes to fit the regression models. Fitting a good regression model takes several iterations of trial and error despite the sophistication of modern statistical software tools, and additional simulation runs may be necessary if the fits are poor. Faster fit times allow the user to try different settings and conduct more iterations in the same amount of time. Therefore, faster fit times are better.

The next are the model fit error (MFE) and the model representation error (MRE). The MFE is a measure of how well the regression model represents the training data that was used to fit the model. The MRE is a measure of how well the regression model represents the simulation model in regions between the points in the test data, and this is also known as the validation set [1, 7]. These will be measured using the root mean square error (RMSE), which is expressed using the following equation [112]:

$$\text{RMSE} = \sqrt{\frac{\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}{n}}$$

where n is the number of points in the dataset. The MFE will use the training data, and the MRE will use a separate validation dataset.

The fourth metric that will be used is the visual representation of the TS by the regression models. This is a subjective metric, but it is important nonetheless. One of the intended applications for the regression method is to present the data in a decision making interface, and the visual appearance of the model is also important.

Table 22: Regression results (values in parentheses are the ratio to the `mean` model or `SMARTS` model)

Model	Runs	Fit Time [sec]		MFE		MRE		platform
`mean`	–	–		0.384	(2.97)	0.365	(2.49)	JMP
`linreg2`	20	35	(0.16)	0.240	(1.85)	0.227	(1.55)	JMP
`nn20`	1	85,700	(379.27)	0.119	(0.916)	0.314	(2.15)	JMP
`nn20-ts`	3	21,800	(97.36)	0.113	(0.874)	0.239	(1.64)	JMP
`nn20-ts (smooth)`	3	21,800	(97.36)	0.113	(0.873)	0.122	(0.84)	JMP
`SMARTS`	10	224	(1.00)	0.129	(1.00)	0.146	(1.00)	Python Script on Linux VM

8.2.5.2 Fit time, MFE, and MRE of the Regression Fits

Each of the regression methods were fit several times to get an average performance. Because of the size of the training data (over 18 million data points), some of the methods could only be executed a couple times. The regression time, MFE and MRE are summarized in Table 22.

The machine that was used to fit these regressions is an Intel Core i7-2600 running at 3.40 GHz with 8 GB of RAM and 64-bit Windows 7 operating system. Most of the models were created using the JMP version 10 software [101], and the program utilizes multithreading to take advantage of the multi-core processors. The `SMARTS` model on the other hand was created using Python scripts that used various Python and R libraries, and the script was executed on a virtual machine (VM) on the same computer running Linux Mint operating system [62], which is an Ubuntu distribution. The script did not use multi-threading. Thus, in terms of computational resources,

it is not an equivalent comparison, but in terms of user experience, it is a fair comparison.

The SMARTS model performed the best overall compared to the other regression methods. In terms of regression time, it was second to the 2nd order linear regression model linreg2, and compared to the neural networks, it is two orders of magnitude faster. Looking at the model fit error (MFE), the nn20 and nn20-ts model have better MFE than the SMARTS model. However, when the model representation error (MRE) is examined, SMARTS outperforms the nn20-ts. The MRE for nn20-ts is in fact higher than the MFE despite the training data being over double the size of the testing data. This indicates that the nn20-ts model is overfitting the training data. The nn20-ts (smooth), on the other hand, does not fall victim to the overfitting due to the smoothing function, and in both the MFE and MRE, nn20-ts (smooth) outperforms the SMARTS model by 13% and 16%, respectively. However, it still suffers from the long fit times.

8.2.5.3 Visual Inspection of the Regressions

Because the original motivation was to create a simplified visual representation of TS data, it is important to examine the plots of the time sequential data. Figures 100 through 104 show plots of each regression methods. Ten sets of inputs from the testing dataset are shown. The actual input parameters are given in Tables 23 through 27.

Each figure shows the simulation outputs at two different input settings, and they are juxtaposed because the output TS shapes are similar. The black lines in

each plot are overlays of 30 simulation runs, and the thicker colored lines are the respective regressions, which are `linreg2`, `nn20`, `nn20-ts`, `nn20-ts (smooth)`, and `SMARTS`.

Overall, the `linreg2` model captures the general trend, but it is far from capturing the nonlinear behavior of the TS shapes. The `nn20` model is better at capturing the data, but there are segments along the TS that do not match the data at all. For instance in Run 2, the regression line falls off the margins of the plot after $t = 600$. The `nn20-ts` captures the general shape well, but it is also extremely noisy. Because there are no constraints to be similar to neighboring time steps, the manner in which it overfits the data causes this noisy output. By applying a smoothing filter, this noise is reduced in `nn20-ts (smooth)`, and it produces a good regression result. The `SMARTS` model captures the shape really well, especially the distinct features such as the "cliffs" in Figure 100. But it still has difficulty with segments of the TS that are very noisy, such as the first half of the TS in Figure 102.

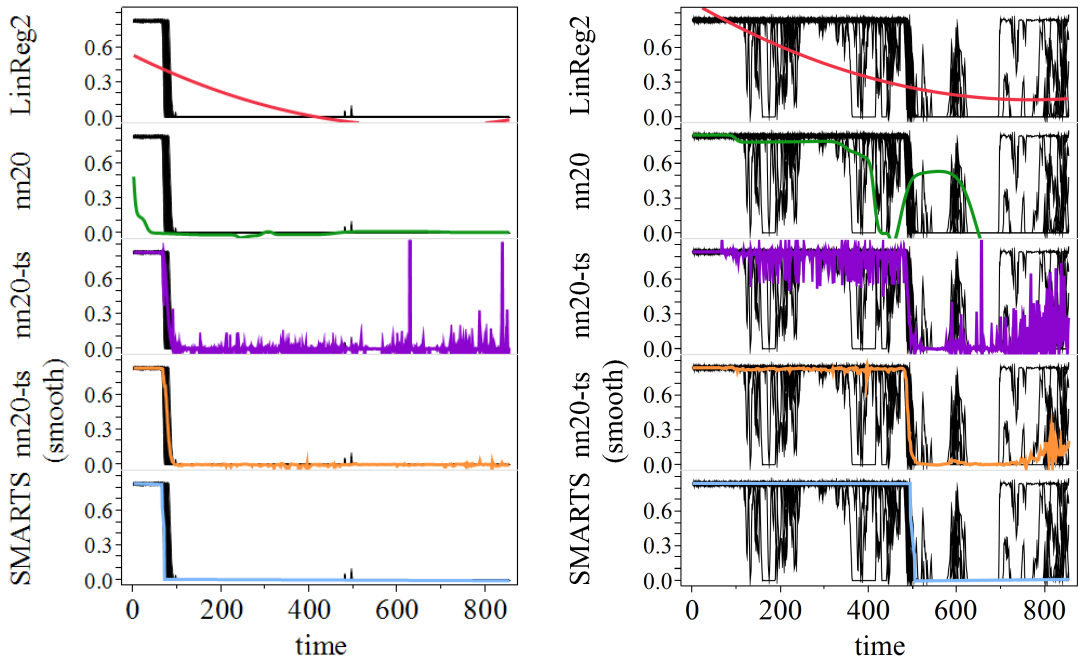
Now inspecting the fits one figure at a time, Figure 100 shows two different inputs that result in the "cliff" occurring at different locations. In the context of the simulation, the sudden drop in Run 1 occurs because the simulation has not "warmed up" yet. Once the parts in the inventory are consumed, the A_O drops because the consumption rate is much faster than the replenishment rate. The sudden drop in Run 2 occurs because of the beginning of the contingency missions. The contingencies have priority over the training and have a higher consumption rate. The parts are diverted from being used for aircraft that are flying training missions, and the A_O suffers due to a lack of parts. The input settings are such that the A_O does not recover even after

the end of contingency missions. For these two inputs, the SMARTS model predicted the shapes very accurately. The nn20-ts model appears to do well aside from the noise, but it is somewhat deceiving because the degree of underfitting is not apparent due to the setting of the axes.

Figure 101 shows two cases that form the "bucket" shape. The ann20 model tries to capture the bucket shape, but it does a poor job. The nn20-ts model does a much better job, but it appears to be underfitting in Run 4 at around $t = 700$. The SMARTS model does a good job fitting the left half of the TS. In Run 3, it appears to be overfitting in the second half of the TS, and this may be due to the use of the median when aggregating the TS. In Run 4, the SMARTS model misses the transition timing.

Figure 102 shows plots where there is significant uncertainty between $t = 100$ to $t = 500$. All the regression models to a decent job of going through the noisy section, and with the regression models used here, there is probably no good way of capturing the uncertainty in behavior of the TS.

Figures 103 and 104 show the variability in the MRFAMOS output based on how the regression models fail to conform to the data. Figure 103 shows two plots that also show similar uncertain behavior as in Figure 102. Run 7 has a drop at $t = 300$ with an attempt to recover the A_O at $t = 500$, whereas Run 8 has a permanent drop at $t = 500$ with some random random dips beforehand. The regression models fit these two with basically the same output shapes. Similarly, in Run 10, the SMARTS model traces the recovery of the A_O after $t = 700$, but the actual data remains flat.



(a) Run 1

(b) Run 2

Figure 100: Comparison of regression results. Runs 1 and 2.

Table 23: Input parameters for runs 1 and 2

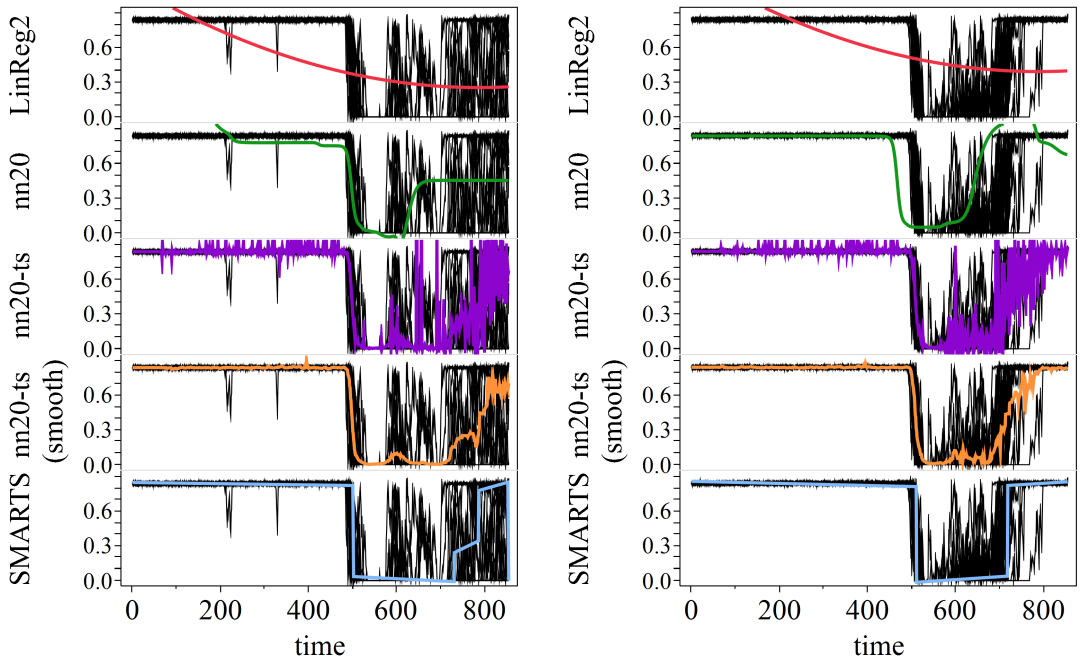
Run #	Fleetsize [# a/c]	Training FHPM[1] [hrs]	Contingency[2]	Mission A Len. [hrs]	Mission B & C Len. [hrs]	Part Life [1/hrs]	Part Repair TAT[3] [hrs]	Inv.[4]
Run 1	103	23.2	11	1.65	3.31	0.0550	1604	479
Run 2	92	21.9	13	1.86	3.71	0.0630	1700	420

[1] FHPM - Flight Hours Per Month

[2] Represents the number of aircraft sent to Contingency 1 and 2 as well as the number of missions flown per day during contingency operations

[3] TAT - Turnaround Time

[4] Inv. - Inventory Level



(a) Run 3

(b) Run 4

Figure 101: Comparison of regression results. Runs 3 and 4.

Table 24: Input parameters for runs 3 and 4

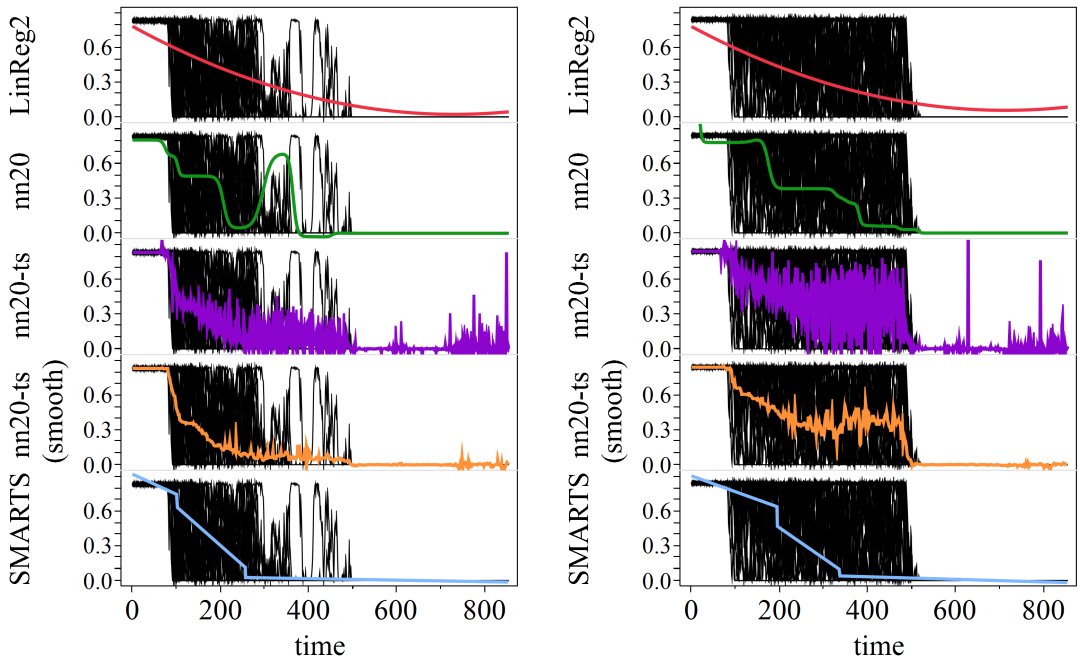
Run #	Fleetsize [# a/c]	Training FHPM[1] [hrs]	Contingency[2]	Mission A Length [hrs]	Mission B & C Lengths [hrs]	Part Life [1/hrs]	Part Repair TAT[3] [hrs]	Inv.[4]
Run 3	90	21.5	13	1.59	3.18	0.0574	1268	483
Run 4	91	20.2	12	1.01	2.02	0.0613	1299	414

[1] FHPM - Flight Hours Per Month

[2] Represents the number of aircraft sent to Contingency 1 and 2 as well as the number of missions flown per day during contingency operations

[3] TAT - Turnaround Time

[4] Inv. - Inventory Level



(a) Run 5

(b) Run 6

Figure 102: Comparison of regression results. Runs 5 and 6.

Table 25: Input parameters for runs 5 and 6

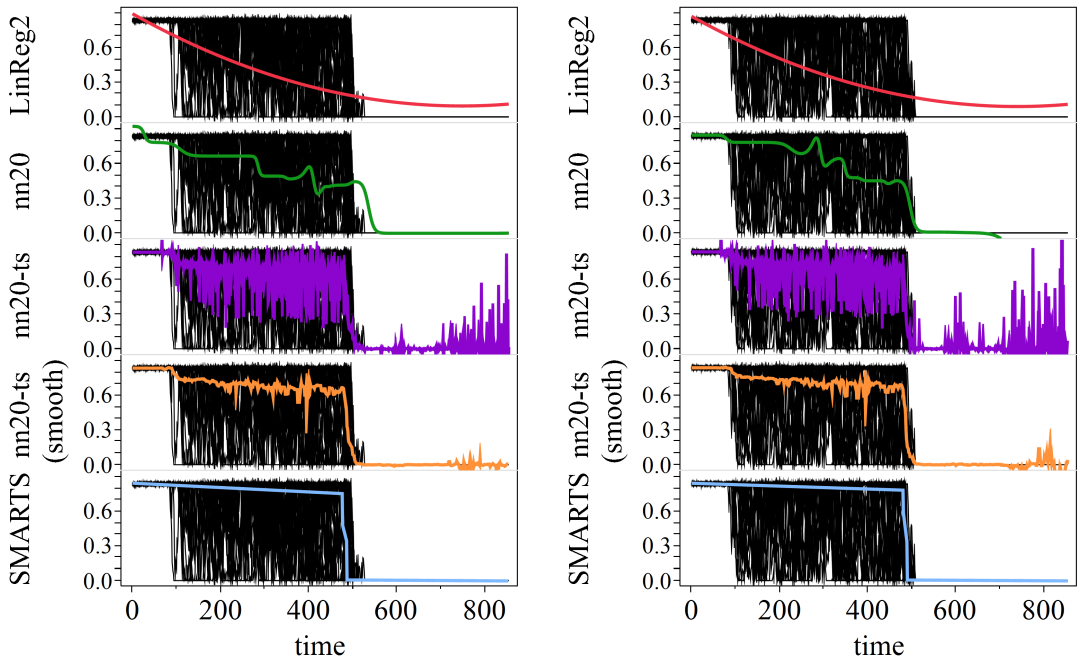
Run #	Fleetsize [# a/c]	Training FHPM[1] [hrs]	Contingency[2]	Mission A Length [hrs]	Mission B & C Lengths [hrs]	Part Life [1/hrs]	Part Repair TAT[3] [hrs]	Inv.[4]
Run 5	93	23.8	10	1.35	2.69	0.0613	1635	476
Run 6	103	20.0	10	1.99	3.98	0.0606	1710	429

[1] FHPM - Flight Hours Per Month

[2] Represents the number of aircraft sent to Contingency 1 and 2 as well as the number of missions flown per day during contingency operations

[3] TAT - Turnaround Time

[4] Inv. - Inventory Level



(a) Run 7

(b) Run 8

Figure 103: Comparison of regression results. Runs 7 and 8.

Table 26: Input parameters for runs 7 and 8

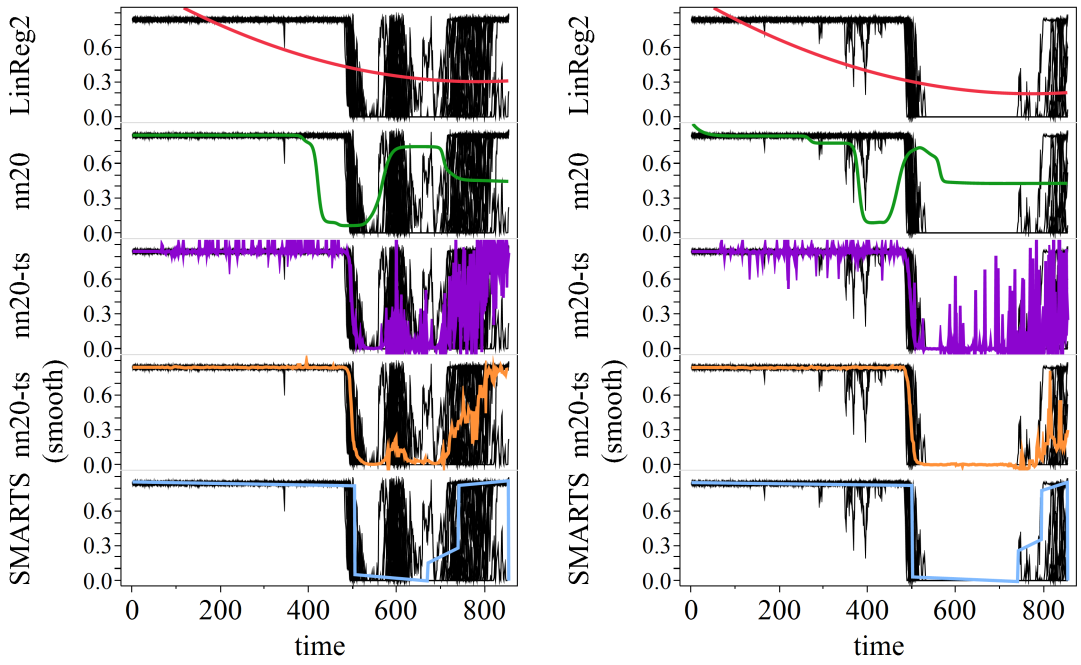
Run #	Fleetsize [# a/c]	Training FHPM[1] [hrs]	Contingency[2]	Mission A Length [hrs]	Mission B & C Lengths [hrs]	Part Life [1/hrs]	Part Repair TAT[3] [hrs]	Inv.[4]
Run 7	92	22.9	10	1.18	2.36	0.0628	1202	455
Run 8	95	21.9	12	1.54	3.08	0.0608	1715	477

[1] FHPM - Flight Hours Per Month

[2] Represents the number of aircraft sent to Contingency 1 and 2 as well as the number of missions flown per day during contingency operations

[3] TAT - Turnaround Time

[4] Inv. - Inventory Level



(a) Run 9

(b) Run 10

Figure 104: Comparison of regression results. Runs 9 and 10.

Table 27: Input parameters for runs 9 and 10

Run #	Fleetsize [# a/c]	Training FHPM[1] [hrs]	Contingency[2]	Mission A Length [hrs]	Mission B & C Lengths [hrs]	Part Life [1/hrs]	Part Repair TAT[3] [hrs]	Inv.[4]
Run 9	92	20.4	14	1.72	3.44	0.0635	1493	479
Run 10	93	21.1	11	1.55	3.11	0.0582	1535	438

[1] FHPM - Flight Hours Per Month

[2] Represents the number of aircraft sent to Contingency 1 and 2 as well as the number of missions flown per day during contingency operations

[3] TAT - Turnaround Time

[4] Inv. - Inventory Level

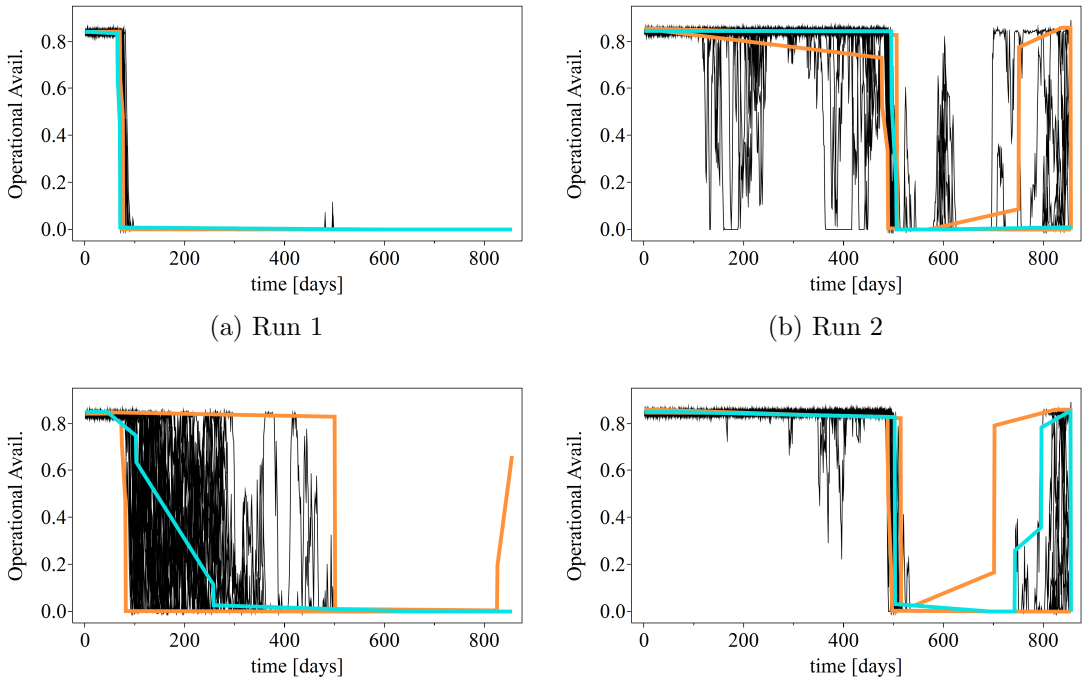
8.2.5.4 *Discussion of the Scalability Experiment*

The SMARTS methodology was used on a set of simulation data from the MR-FAMOS model, and the results were compared against several regression methods. The SMARTS methodology is not a winner in any one category, but when the overall aspects are considered, it is a very competitive method. The MRE and MFE are second to `nn20-ts (smooth)`, and the fit times are on the order of several minutes compared to hours using NN.

In the execution of the SMARTS code, the PA is used to determine the number of PCs to keep. It returned the maximum number of 10 PCs, and it appears that the PA is not robust to significant noise in the TS data. However, the regressions results from the `SMARTS` model looked acceptable, and it demonstrates the robustness of the model-based clustering to noise.

8.2.6 SMARTS Visualization Experiment

The SMARTS Visualization Experiment is designed to demonstrate that the regression outputs from the SMARTS model can be used to create plots that can be incorporated into visualizations. Some of the same data and plots that were used in the previous experiment will be used to show the bounds. The upper and lower bounds do not represent the minimum and maximum. Instead, they show the three sigma bounds of what the median value can take. Figure 105 shows examples of the upper and lower bounds created using SMARTS. The median line is plotted in blue while the bounds are in orange. The black lines represent 30 repetitions of actual simulation runs.



(a) Run 1

(b) Run 2

(c) Run 5

(d) Run 10

Figure 105: Showing the upper and lower bounds created using SMARTS. The light blue lines are the median, and the orange lines are the upper and lower bounds.

Because regressions are not perfect, they can violate implicit rules such as the upper and lower bounds should not intersect. When visualizations are created, these rules are enforced to reflect how the data should look like. Several constraints are applied to the PLRs and plotted in Figure 105. The constraints include the following: making sure the upper bound, lower bound, and the median line do not intersect, the lines are greater than or equal to zero because the plot charts operational availability which is defined from zero to one, and the lines should not exceed 0.86 because that is the observed maximum from the dataset.

The simulation runs from Run 1 all follow the same "cliff" shape pattern, and the bounds reflect this. The upper and lower lines tightly bound the median line. The small bumps at around time step 500 appear to be outliers, and the bounds also

ignore them. The simulation data in Run 2 appears extremely noisy, but the plot shows a narrow range. The lower bound seems to suggest that the noisy activity before time step 400 is an outlier and that most of the TS hover above 0.8. Similar conclusion can be drawn about the lines in Run 10. The bounds in Run 5 neatly capture the upper and lower bounds of the data. The upper bound towards the end of the simulation time shows a small rise, suggesting that for some repetitions, the A_O would begin to recover.

Summary of SMARTS Visualization Experiment This experiment demonstrated that the upper and lower bounds can be generated using the SMARTS method and that the bounds can sufficiently encompass the data. Because the bounds are created using three standard deviations from the average PC in the Z domain, it does not capture the minimum and maximum. It does vary the range of possible shapes depending on the situation. It also ignores TS segments that appear to be outliers.

8.3 Summary

The experiments were introduced in Chapter 7, and the details and the results were presented in this chapter. Each experiment was designed to test specific aspects of the SMARTS methodology. The Parallel Analysis Experiment demonstrated that the PA method is capable of determining the correct number of PCs. The Robust Threshold and TS Smoothing Experiment found that **case mean** and **case median** are the best method for smoothing data when repetitions can be used. Otherwise, the **case median** with a window size of three outperforms other smoothing methods. The Feature Selection and Clustering Experiment conducted an extensive evaluation

of various combinations of clustering methods and feature sets, and model-based clustering using PCA of the data and TS breakpoints as the features proved to be the best combination to cluster TS data and create PLRDGs. The SMARTS Feasibility Experiment provided a step-by-step example of implementing the methodology, and the experiment proved the feasibility on a sample dataset. The SMARTS Scalability Experiment tested the methodology on data from the MRFAMOS model, and showed that it performs really well as a whole. The SMARTS Visualization Experiment showed that upper and lower bounds can also be generated using the methodology. In the next chapter, the results from the experiments are reviewed, and the hypotheses and research questions are revisited.

CHAPTER IX

DISCUSSION AND CONCLUSIONS

The previous chapter performed the experiments that were introduced in Chapter 7 and presented the initial findings. In this chapter, the results are summarized and discussed, and the claims and hypotheses are revisited in light of the new information from the experiments. The chapter concludes with a summary of contributions and suggestions for future extensions of the research.

The Surrogate Model and Regression for Time Sequences (SMARTS) methodology has been developed to create surrogate models tailored for TS data. The main application for the regression is for visualization purposes so that the information can be represented and interpolated without the original data. The regression of large TS datasets is difficult because they are nonlinear, and the size of the datasets causes the regression to be slow. These types of datasets can be found with simulation models for operations and sustainment (O&S) research where the simulation is run varying multiple variables at the same time. To help with the modeling, initial insights were drawn from studying the Fighter Aircraft Maintenance and Operations Simulation (FAMOS) model. It was observed that TS data can be represented using piecewise linear regression (PLR), and the entire dataset can be captured using two separate regressions, where is for the design space variability while the other for the TS shapes. This led to the development of the SMARTS methodology into a three step process as

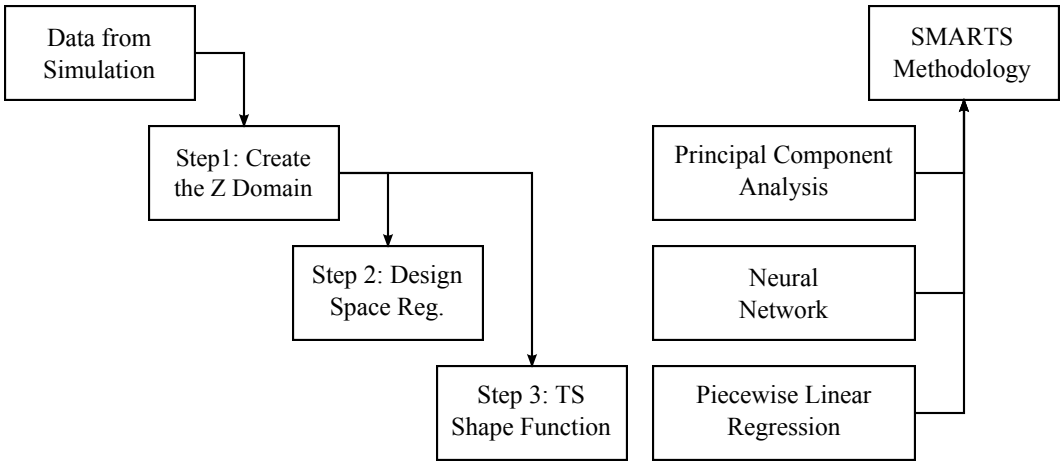


Figure 106: Three steps of the SMARTS methodology and the corresponding methods that satisfy the steps

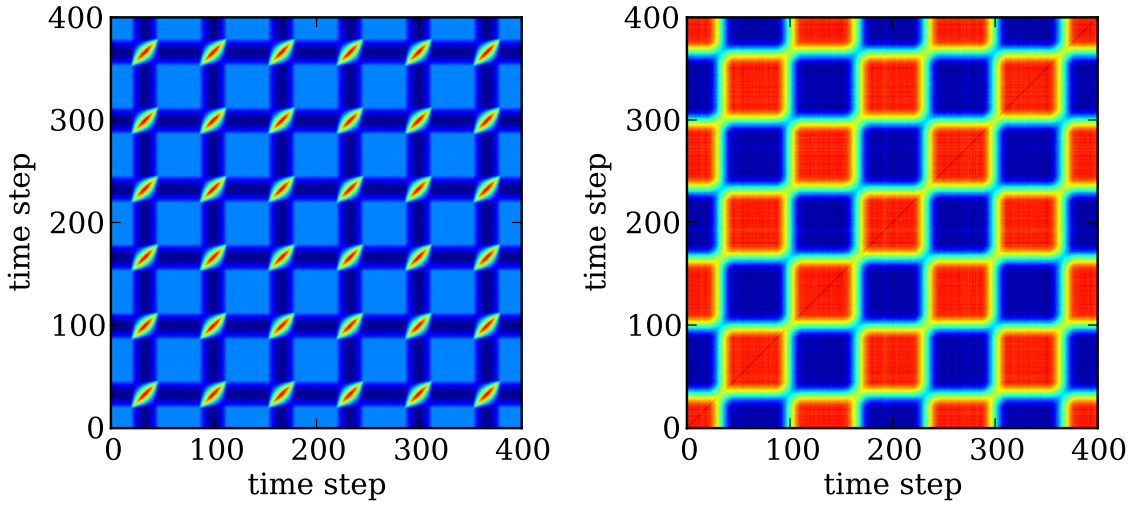
depicted in Figure 106. The literature was surveyed to determine how to accomplish those steps, and experiments were designed for those with inconclusive answers, and the following section discusses the results of these experiments.

9.1 Discussion of Results and Revisiting the Hypotheses

9.1.1 Parallel Analysis Experiment

The parallel analysis (PA) experiment was designed to test Claim 2, which stated the following, "The parallel analysis method is an appropriate method to determine the number of components to keep when applying principal component analysis to a time sequential dataset that is generated by an engineering simulation model for design space exploration." The PA method was tested against a large set of test data, and it was indeed successful in identifying the correct number of PCs in most cases. However, the PA method had trouble with sets of discontinuous TS that were phase shifting.

Principal component analysis (PCA) used the covariance matrix to capture the



(a) Sawtooth Wave

(b) Triangle Wave

Figure 107: The covariance matrix of `phase-sawtooth` and `phase-triangle` datasets with data size of $n = 400$ and noise of $\sigma = 0.1$. The red color indicates high covariance, and blue indicates low covariance.

relationship between the variables, and it did not presume that the data is sequential. The sequential nature of TS was captured by calculating the correlation of a point moving in consort with its neighbors. Therefore, a discontinuity that "moves through" the TS through phase shifting disrupts the correlation with its neighboring variables. This caused the variance to be distributed into more dimensions or PCs.

The differences can be seen by comparing the heat map of the covariance matrix as shown in Figure 107. The `phase-sawtooth` and `phase-triangle` datasets with data size of $n = 400$ and noise of $\sigma = 0.1$ were used to create the plots. In the regions of the discontinuity with `phase-sawtooth` (e.g. time step = 100), there is a high covariance, but the shape is also highly elliptic. The relationship between the time steps is distinctly different when compared with the covariance matrix calculated from `phase-triangle` dataset.

When the PA method was used in the Scalability Experiment (Section 8.2.5), the method consistently returned the maximum limit of 10 PCs, which is a preset upper limit for the SMARTS methodology. This indicated that the PA method is not robust a to large amount of noise. The MRFAMOS data also showed regions that looked like square waves where the operational availability (A_O) jumped between 0 and 0.9 due to the stochasticity in the simulation model, and perhaps, a phenomenon similar to the phase shifting problem was occurring here as well.

Discontinuous data is a possibility in simulation models, and phase shifting and feature shifting are also potential problems. However, these are beyond the scope of this thesis. At least for the data that was used and presented in this thesis, shifting was minimal, and the noise did not pose a problem to the results of the experiments.

Responding to the Claim 2, the experiment demonstrated the effectiveness of PA in identifying the correct number of PCs.

Claim 2 is demonstrated.

This claim ties back to Research Question 13, which stated "How many principal components should be kept after principal component analysis (PCA) is applied to a dataset?" By using the PA method, the correct number of PCs can be determined.

9.1.2 Robust Threshold and TS Smoothing Experiment

The robust threshold experiment was designed to test Claim 1, which stated the following, "A smoothing technique can be applied to a time sequence to estimate the noise in the data to a sufficient accuracy so that it can be used as the threshold setting for TS segmentation methods." Various smoothing functions were tested against a set

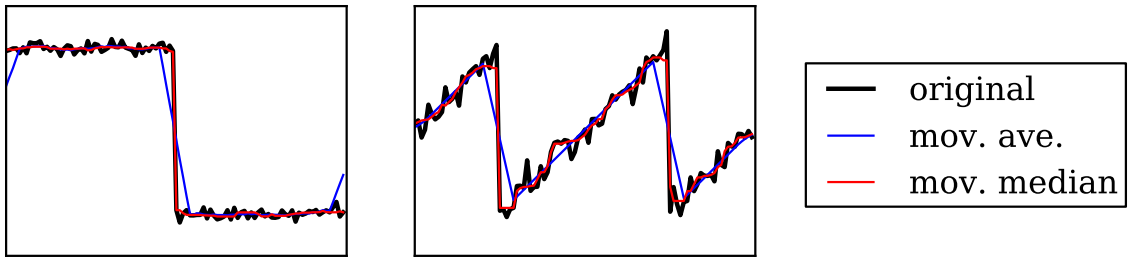


Figure 108: Comparison of the **moving average** and **moving median** smoothers with window size of 9 on a square and sawtooth wave

of sequential data, and the **case median** and the **moving median** with a window size of 3 determined to be the best options. The **case median** was superior when there are repetitions to use for smoothing, while the **moving median** was the better option when there are fewer than five repetitions to use. For the SMARTS methodology, the **case median** was used because it has repetitions for smoothing.

The higher performance of **moving median** smoother among the other moving window smoothers was partially due to the TS datasets that were used to evaluate the smoothers. The **moving average** and its variants (**moving triangle** and **moving Gaussian**) tended to deemphasize sharp corners and features as illustrated in Figure 108. The comparison is slightly exaggerated in the figure by using a larger window size of 9 for both smoothers. As mentioned previously, discontinuous data is a possibility, and so, the use of the square and sawtooth waves for comparing the smoothing functions was fair.

With respect to the other smoothers, the **case mean** and **case median** are better performing methods when repetitions are guaranteed. The **Kalman** smoother and **Savitsky-Golay** smoothing filter could also be modified to incorporate repetitions by applying the algorithms on an averaged TS (or the result of **case mean**), but

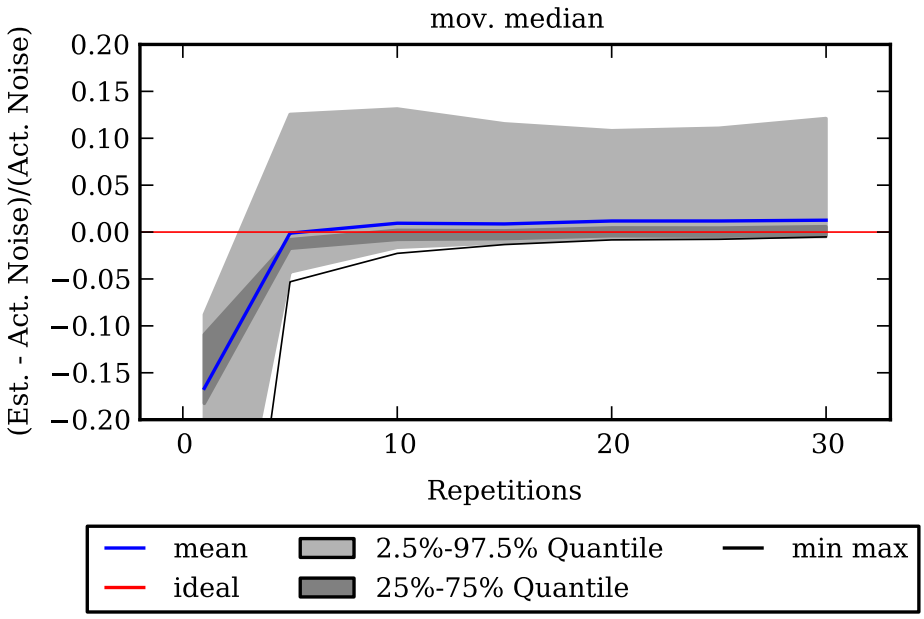


Figure 109: Estimation of noise in TS using a **moving median** smoother varied over the number of repetitions used for the estimate. (This is a closeup view from Figure 68)

this mixes different algorithms and opens doors to a large number of combinations. These combined approaches were not explored. The **Savitsky-Golay** method fits a regression to a subset of points, and if the simplest linear regression is used, it would suffer similar problems to the **moving average**. There were options to specify a higher order polynomial which could avoid this problem. Figure 109 shows a closeup view of the **moving median** from Figure 68. For most of the data that was tested, the **moving median** estimates had less than 5% error.

In answering Claim 1, certain smoothing algorithms were indeed capable of approximating the noise in the TS, and the **case median** and the **moving median** with a window size of 3 were the best among them for the tested conditions.

Claim 1 is demonstrated.

The result of this experiment answers Research Question 11, which stated "How can a robust threshold setting be derived from the data for TS with noise?" and Research Question 12, which stated "Which smoothing technique is appropriate for smoothing TS data to estimate the noise in the data?" The **case median** and **moving median** can be used as a smoother to determine the threshold setting for the TS segmentation algorithms.

9.1.3 Clustering and Feature Selection Experiment

The clustering and feature selection experiment was designed to test Hypothesis 2. The hypothesis stated the following, "If principal components (PC) of the time sequential dataset and the first PC of its breakpoints are used as the features for clustering, then the model-based clustering that uses Gaussian mixture models will yield clusters that are suited for piecewise linear regressions." The experiment concluded that indeed model-based clustering using PCA on the output data and breakpoints as features was the best combination overall, but it did identify a few weaknesses. For example, model-based clustering did not perform well when a cluster is singular in any dimension.

In the context of clustering the data from simulation models, singular data is unlikely. Instead, it is more likely to have noisy data, and in the case where the PA method overestimates the number of PCs, there could be extraneous feature dimensions. This situation was observed in the scalability experiment where the PA method identified ten or potentially more significant PCs, and despite this, the model-based clustering performed well.

The density-based clustering DBSCAN performed well where the model-based method faltered. Density-based methods worked well when the points were clustered together as it was demonstrated with the `shapes` test set where there were distinct clusters of points. However, it did have trouble with the `discrete` test set, which was also designed to have easily identifiable clusters. The distance between the clusters may have been too close and caused DBSCAN to group them together. Another issue identified with DBSCAN was that the inter-point distances can be distorted after the data is normalized, and this caused points in a cluster to be too far apart to be grouped together. This was especially true when there are few points in the dataset.

Because DBSCAN grouped intersecting clusters together, the modified version DBSCANmod applied a line segmentation algorithm to the results of DBSCAN. The clusters for TS data formed lines in n-dimensional space with each cluster pointing in different directions, and the ends of the clusters could be touching as the data transitions from one type of transformation to another. DBSCAN grouped these two together because they were touching, so a segmentation algorithm was needed to detect the change in direction and segment the cluster. DBSCANmod adopted the shortfalls of DBSCAN, and it had its own share of problems. Because the threshold for this method was difficult to set, it oftentimes ended up oversegmenting the data, especially if the data is noisy. It succeeded in performing better than DBSCAN as intended when it was applied to the `ts` test set, but it was still inferior to the model-based clustering method.

The `PCA` feature was one of the best performing features overall. `Isomap` and

LLE did not do well, and perhaps the TS data is not suited for these methods. The Mean and Standard Deviation features worked pretty well for being a simple feature, and this could be because most of the testing datasets were also simple in nature. The two features did not perform well with the ts test set, which has more complex transformations. The DFT feature performed really well with the null test set, but it was inferior to other features in other test sets. The breakpoints features were mainly useful with the ts test set as expected.

Combinations of features have resulted in mixed results. Sometimes, the clustering accuracy was between that of the individual features. In other circumstances, the combined performance was better than any of its constituent features alone. More information about the data was not necessarily better, and it suggested that the features should be chosen and used carefully. For the purposes of creating PLRDGs, PCA + PCA(Breakpoints) feature set was the best option because it captured aspects of the data as well as its breakpoints.

Hypothesis 2 is supported.

The results of this experiment answered Research Question 8, which asked, "What is the appropriate clustering technique to create groups that will yield piecewise linear regressions with good fits?" and Research Question 9, which asked, "What are the appropriate features to use for clustering TS datasets to create piecewise linear regressions with good fits?"

Also, with the conclusions from the Robust Threshold and TS Smoothing Experiment and the Clustering and Feature Selection Experiment, the sub-steps of Step 3

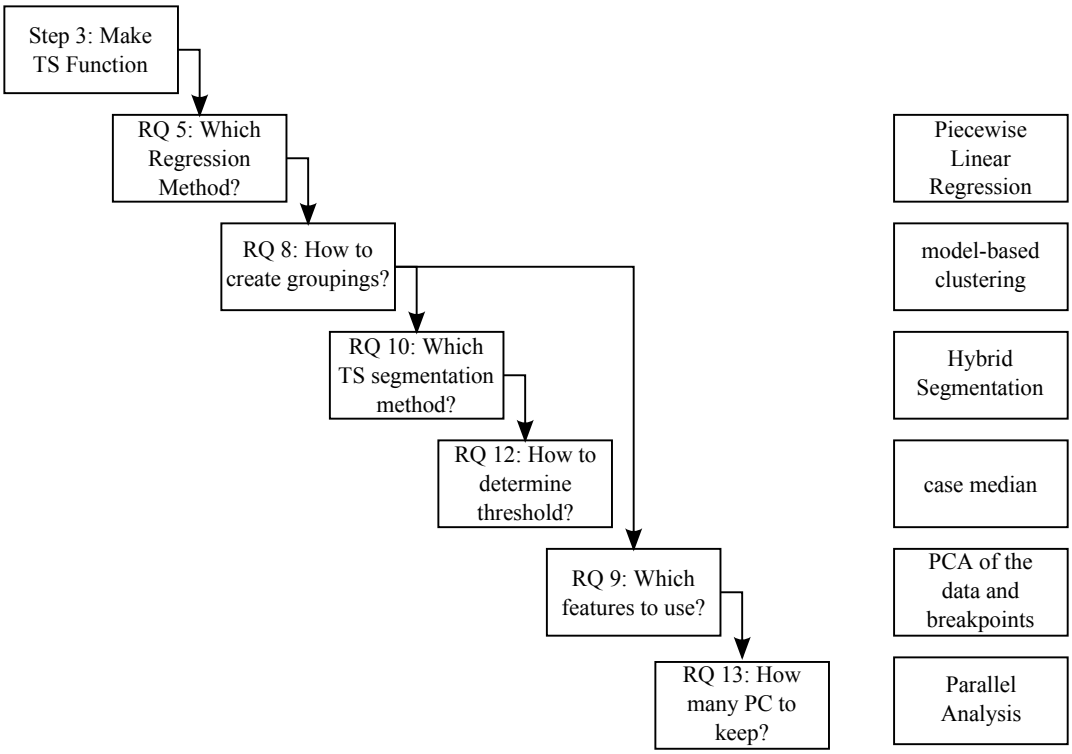


Figure 110: Steps and methods of Step 3 of the SMARTS methodology

were completed. Figure 110 shows the complete list of methods that satisfy Step 3.

9.1.4 SMARTS Experiments

Two experiments were conducted to test Hypothesis 1, which stated, "In the context of design space exploration using simulation models, if the output data consists of high-dimensional and sequential outputs such as a time sequence, then the SMARTS methodology will yield regression models that are equal or better in performance (fit time, model fit, and representation errors) compared to other regression methods." The SMARTS Feasibility Experiment tested if the SMARTS methodology was effective on a set of test data, and the results were promising. Then the method was

applied to data generated by the Multirole Fighter Aircraft Maintenance and Operations Simulation (MRFAMOS) model, and the SMARTS method was overall a highly competitive method but was not superior in any particular metric.

However, the regression model generated using SMARTS had the best overall performance. In terms of the fit time, it was second to the second-order linear regression, and it was two orders of magnitude faster than the neural network (NN) regression models. Two NN models were initially created. One fit a NN to the entire dataset while the other fit NNs to each time step. Both of these NN models had better model fit errors (MFEs) than the SMARTS model, but it was clear that it was overfitting the data when the model representation errors (MREs) were evaluated. When the models were compared visually, the NN models generated TS shapes that were either noisy or poor representations. Then, the NN model that was fit to each time step was modified by applying a moving median smoother. This modified NN had better MFE and MRE than the SMARTS model by about 15%, and the visual representation was also good. The SMARTS model appeared somewhat simplistic when plotted with the noisy MRFAMOS data, but it captured the major features very well.

The SMARTS Visualization Experiment was performed to show that the SMARTS methodology could be used to capture the upper and lower bounds for a specific input. The SMARTS method did not capture the absolute minimum or maximum but rather showed the ± 3 standard deviations range in which the median could vary. It also did not capture outlier segments that strayed from the rest of the TS segments, which were caused by the stochasticity in the model.

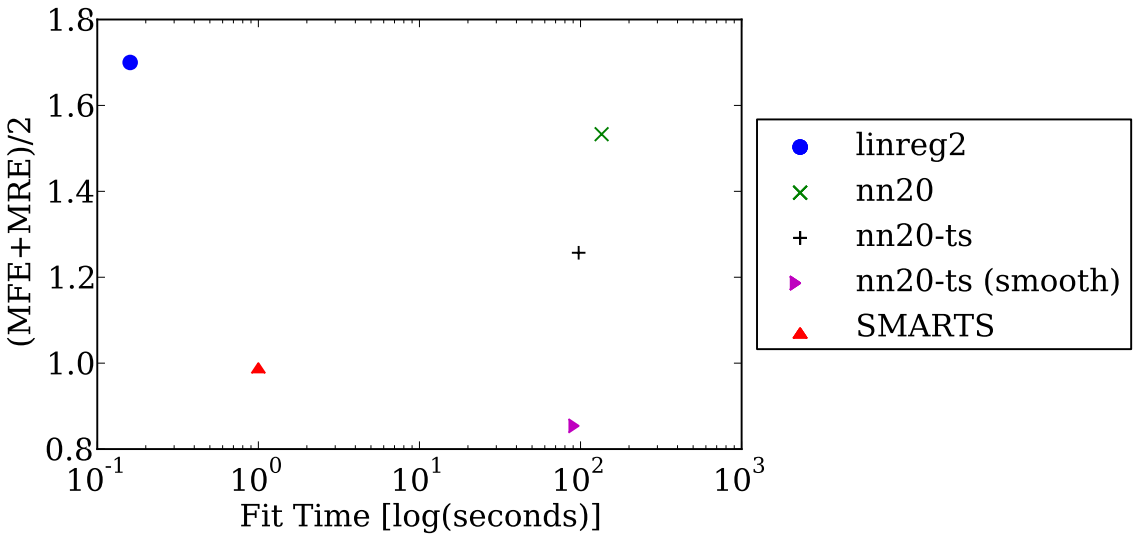


Figure 111: Fit times vs. the average of MFE and MRE for several regression methods. Values are scaled to the SMARTS regression model.

Hypothesis 1 stated that the SMARTS methodology would be superior in performance to other regression methods. The results showed that the methodology was not superior in any one particular metric of fit time, fit error, and visual representation. However, it was conditionally better compared to the others when its performance was considered as a whole. Figure 111 shows the results of the SMARTS Scalability Experiment. The x-axis plots the normalized fit times in log scale, and the y-axis is the average of the normalized MFE and MRE. The linear regression, the smoothed NN for every time step, and the SMARTS methodology lie on the Pareto frontier of fit time and performance, and tradeoffs can be made depending on what is desired when choosing a regression method for TS data.

Hypothesis 1 is conditionally supported.

This experiment ultimately satisfied the initial goals of this research which were

discussed in the first chapter, which was to create a regression methodology to represent large datasets with sequential data.

9.2 Summary of Contributions

The main contributions are the following:

- The use of an intermediate domain to split the regression into two steps, separating the complexity into two separate regressions - one to capture the variability due to design space variables and one to capture the output TS shapes
- The method to create piecewise linear regressions (PLR) for TS data generated by simulation models for operations and sustainment
- The method to create PLRs using model-based clustering as the method for partitioning the data
- The evaluation of the parallel analysis method for time sequential data over a wide range of parameter settings
- The evaluation of a comprehensive set of smoothing techniques on time sequential data
- The criteria which a group of points should satisfy in order to result in good PLR fits

The most significant contribution was the splitting of the regression into two steps to manage the complexity and to reduce the amount of data that is required for the

regression. This approach bridged the field of image processing with the field of engineering design. Currently, there is significant research being done in image processing to organize a set of unlabeled photographs in relation to each other, and manifold learning techniques are heavily employed to reduce the dimensions of the images to capture the underlying actions. In image processing, the process usually stops with finding this manifold and using it for pattern matching or image regression. This is because the images are the starting and ending point, whereas in design, the starting point is the set of inputs into a system or simulation model. Purposeful changes are made using the design of experiments, and the end results are the TS shapes. This was how the SMARTS methodology bridged design and image processing.

The next contribution built on the first to enable regressions of TS data to be used in the engineering design process. Regression models are useful in design tasks such as decision making. The surrogate models can be used in decision making interfaces to allow quick tradeoff studies, and design points can be interpolated easily. It also improves the portability of these tools because the large datasets are represented using equations and functions and do not need to accompany the tools. The SMARTS methodology expands the tools available to engineers so regression models can be fit to TS data.

The SMARTS methodology approached piecewise regression (PR) differently from the conventional PR methods and demonstrated a novel combination of techniques tailored towards TS data. PR methods such as regression trees use a recursive approach to partition the data until a threshold condition is satisfied. On the other

hand, B-splines and P-splines start with more than enough breakpoints and use optimization routines to fit the model. The SMARTS methodology took a more serial approach where the regression did not inform the creation of the *pieces* of the piecewise regression. Instead, the methodology tried to guarantee the goodness fit across these pieces by using a specific combination of clustering and data feature sets.

The PA method had been tested against a large selection of test data [51, 82], and a limited study had been performed on TS data [47, 61]. However, there has not yet been a study published on the PA method applied to TS data to this detail. One of the more interesting findings was the impact of discontinuities or large jumps in phase shifting sequential data to the results of the PA method.

Another contribution was the comprehensive study performed on one-dimensional smoothing techniques on test TS data. The investigation also included repetitions of data, which is usually not done because data with the same initial conditions are difficult to collect in real world applications.

The piecewise linear regression data groups (PLRDGs) provided requirements that a group of points must exhibit, and if satisfied, a PLR model generated from that group of points had some level of guarantee of having good fit characteristics.

9.3 Potential Extensions

The SMARTS methodology can be considered as a framework for sequential data regression, and it was tailored for TS data from O&S simulation results. There is room for improvement because the investigations into each components were not exhaustive. Furthermore, there are potentially new applications based on some findings from the

research.

The choice of PLR was driven by the observations and the need for data reduction, but this can be relaxed if the visual representation does not need to be piecewise and linear or if the data constraints are not as severe. Once the Z domain is created, the task of capturing the shape could be performed with other regression methods besides PLR.

The Z domain in this thesis was limited to one dimension to limit the scope of the work but also because the simulation model only behaved in such a way. A Z domain with more than one dimension is theoretically possible, but it was not demonstrated in this thesis. Some of the challenges associated with this would be the resolution of overlapping groups after the clustering. Also, as more dimensions are added, the number of individual regressions can increase very rapidly.

The creation of the Z domain introduces a new way to sample the design space for adaptive sampling. Because the TS is fit using the Z domain, it is most effective to add points to where it is the most sparse in the intermediate domain. However, if the number of dimensions in Z is less than X, it is likely that there is a non-unique mapping from $Z \rightarrow X$. At the same time, there is a need to sample sparse areas in X to make a better mapping of $X \rightarrow Z$. There is an opportunity to optimize the points that are sampled in X which will maximize the information gained in both mappings.

9.4 Summary

This final chapter summarized the research through the performed experiments, highlighted the contributions, and discussed avenues of extending the research. The experiments and development of the SMARTS methodology was successful, but it did not satisfy the audacious goal of becoming a regression method that is better compared to existing methods in fit time, accuracy, and visual appeal for TS data. However, the methodology presented a good and competitive compromise of these aspects.

Many options were explored through an extensive literature survey and experimentation process to create the final form of the SMARTS methodology, but many more improvements can be made to the approach and implementation. Creating surrogate models for large TS datasets is challenging, and the SMARTS methodology has successfully overcome this trial with an ingenuous combination of existing methods and prior knowledge, and this was only made possible by standing on the shoulders of giants.

APPENDIX A

SMARTS SOFTWARE CODE

A.1 multivariate_piecewise_linear_regression.py

```
import ts_segments_2 as ts_segments
import TimeSeriesDB
import my_functions
import smoothers
import MyScripts.FeatureExtract as FeatureExtract
import MyScripts.ApplyClustering as ApplyClustering

import numpy as np
from sklearn.decomposition import PCA
from sklearn.cluster import DBSCAN
from scipy.spatial import distance
from itertools import cycle
import rpy2 as rp
import rpy2.robjects as robjects
import pandas as pd
import pandas.rpy.common
from rpy2.robjects.packages import importr
from matplotlib import gridspec, offsetbox
import matplotlib.pyplot as plt
import copy
from collections import Counter
from itertools import izip

from pybrain.datasets import SupervisedDataSet
from pybrain.tools.shortcuts     import buildNetwork
from pybrain.supervised.trainers import BackpropTrainer
from rpy2.robjects.packages import importr
import gc
from setuptools.dist import Feature
from scipy.interpolate import interp1d

def lazyprop(fn):
    attr_name = '_lazy_' + fn.__name__
    @property
    def _lazyprop(self):
        if not hasattr(self, attr_name):
            setattr(self, attr_name, fn(self))
        return getattr(self, attr_name)
    return _lazyprop

class multivariatePiecewiseRegression(object):
    def __init__(self, df, name='MPLR', test_data=None,
```

```
                df_inputs=None):
        super(multivariatePiecewiseRegression, self).__init__()
        self.name = name
        assert isinstance(df, pd.DataFrame)
        self.df = copy.copy(df)
        self.test_data = test_data
        if df_inputs is None:
            self.df_inputs = self.df.xs('inputs', axis=1)
            self._assign_pca()
            self.df_pca = self.df.xs('attr',
                                     axis=1).xs('pca',axis=1)
        else:
            self.df_inputs = df_inputs.xs('inputs', axis=1)
            self.df_pca = df_inputs.xs('attr',
                                       axis=1).xs('pca',axis=1)
            self.df_run_numbers = df_inputs.xs('run_numbers',
                                               axis=1)
        self._x_to_z_NN = None
        self._z_to_y_breakpoints = None
        self._z_to_y_regressions = None
        self._z_to_y_pca2groups = None
        self._df_pca_pred = None

    @classmethod
    def from_mplr(cls, mplr):
        df_pca = pd.DataFrame(mplr.df_pca, columns=['pca'])
        df_inputs = pd.concat({'inputs':mplr.df_inputs,
                               'attr':df_pca,
                               'run_numbers':mplr.df_run_numbers},
                              names=['L1','L2'], axis=1)

        reg = cls(df=mplr.df, name=mplr.name,
                  test_data=mplr.test_data,
                   df_inputs=df_inputs)
        reg._x_to_z_NN = mplr._x_to_z_NN
        reg._z_to_y_breakpoints = mplr._z_to_y_breakpoints
        reg._z_to_y_regressions = mplr._z_to_y_regressions
        reg._z_to_y_pca2groups =  mplr._z_to_y_pca2groups
        reg._df_pca_pred = mplr._df_pca_pred
        return reg

    @staticmethod
    def update_mplr(mplr):
        return multivariatePiecewiseRegression.from_mplr(mplr)

    @lazyprop
    def _x_var_names(self):
        nms = list(self.df_inputs)
        if 'mc' in nms:
            nms.remove('mc')
        return np.sort(nms)

    def make_regressions(self):
        self._z_to_y()
```

```
97          self._x_to_z()
98
99      def _z_to_y(self):
100         self._initial_segmentation()
101         self._extract_features()
102         self._group_data()
103         self._segmentation_by_group()
104         self._clean_up_groups_2()
105         self._check_group_nseg()
106         self._extract_features()
107         self._create_regressions()
108
109     def _x_to_z2(self, how='median'):
110         if how == 'median':
111             self._x_to_z()
112         elif how == 'min':
113             self._x_to_z_min()
114         elif how == 'max':
115             self._x_to_z_max()
116         else:
117             raise RuntimeError("mplr._x_to_z2 --> invalid 'how'")
118
119     def _x_to_z_min(self):
120         # For each input case, calculate the average and
121         # standard deviation
122         # of the first PC. The "min" value is 3 deviations away.
123         run_numbers = set(self.df_run_numbers.values.ravel())
124         df_pca = self.df_pca
125         df_pca_min = pd.DataFrame([])
126         for rn in run_numbers:
127             df_pca_subset = df_pca[self.df_run_numbers.xs( \
128                       'run_number', axis=1)==rn]
129             pca_values = df_pca_subset.values
130             pca_mean = np.mean(pca_values)
131             pca_std = np.std(pca_values)
132             # add because pca flips the axes
133             pca_val = pca_mean + pca_std * 3
134             if pca_val < 0: pca_val = 0
135             df_pca_min = pd.concat((df_pca_min,
136                             pd.DataFrame([pca_val],
137                             columns=['pca'],
138                             index=[df_pca_subset.index[0]])))
139
140         nms = list(self.df_inputs.columns)
141         nms1 = np.array(nms)
142         if 'mc' in nms1:
143             nms.pop(np.argwhere(nms1 == 'mc'))
144         df_inputs = self.df_inputs.xs(np.sort(nms), axis=1)
145         df_inputs = df_inputs.ix[df_pca_min.index]
146
147         # Merge Inputs and PCA so that run_id matches
148         df_all = df_inputs.join(df_pca_min)
149
150         # Split pandas DataFrame into Inputs and PCA
```

```
151         nms = list(df_all.columns)
152         nms1 = np.array(nms)
153         nms.pop(np.argwhere(nms1 == 'pca'))
154         pca = df_all.xs(['pca'], axis=1)
155 
156         df_inputs = df_all.xs(nms, axis=1)
157 
158         # R
159         R = robjects.r
160         importr('nnet')
161 
162         R_df_x = pandas.rpy.common.convert_to_r_dataframe(df_inputs)
163         R_df_y = pandas.rpy.common.convert_to_r_dataframe(pca)
164         fit_nn = None
165         fit_value = 1e10
166         for i in range(10):
167             nn = R.nnet(x=R_df_x, y=R_df_y, size=5, maxit=500,
168                         decay=1e-4, abstol = 1.0e-10,
169                         reltol = 1.0e-20, trace=False)
170             if fit_value > nn.rx2('value')[0]:
171                 fit_value = nn.rx2('value')[0]
172                 fit_nn = nn
173         self.fit_nn = fit_nn
174         self._x_to_z_NN = fit_nn
175         self.garbage_collect()
176 
177     def _x_to_z_max(self):
178         # For each input case, calculate the average and
179         # standard deviation
180         # of the first PC. The "min" value is 3 deviations away.
181         run_numbers = set(self.df_run_numbers.values.ravel())
182         df_pca = self.df_pca
183         df_pca_max = pd.DataFrame([])
184         for rn in run_numbers:
185             df_pca_subset = df_pca[self.df_run_numbers.xs(
186                 'run_number', axis=1)==rn]
187             pca_values = df_pca_subset.values
188             pca_mean = np.mean(pca_values)
189             pca_std = np.std(pca_values)
190             # subtract 3*pca_std because pca flips the axes
191             pca_val = pca_mean - pca_std * 3
192             if pca_val > 1: pca_val = 1
193             df_pca_max = pd.concat((df_pca_max,
194                                     pd.DataFrame([pca_val],
195                                     columns=['pca'],
196                                     index=[df_pca_subset.index[0]])))
197 
198         nms = list(self.df_inputs.columns)
199         nms1 = np.array(nms)
200         if 'mc' in nms1:
201             nms.pop(np.argwhere(nms1 == 'mc'))
202         df_inputs = self.df_inputs.xs(np.sort(nms), axis=1)
203         df_inputs = df_inputs.ix[df_pca_max.index]
204 
```

```
205         # Merge Inputs and PCA so that run_id matches
206         df_all = df_inputs.join(df_pca_max)
207 
208         # Split pandas DataFrame into Inputs and PCA
209         nms = list(df_all.columns)
210         nms1 = np.array(nms)
211         nms.pop(np.argwhere(nms1 == 'pca'))
212         pca = df_all.xs(['pca'], axis=1)
213 
214         df_inputs = df_all.xs(nms, axis=1)
215 
216         # R
217         R = robjects.r
218         importr('nnet')
219 
220         R_df_x = pandas.rpy.common.convert_to_r_dataframe(df_inputs)
221         R_df_y = pandas.rpy.common.convert_to_r_dataframe(pca)
222 #         fit_nn = R.nnet(x=R_df_x, y=R_df_y, size=10,
223                         decay=0.0001, maxit=500)
224         fit_nn = None
225         fit_value = 1e10
226         for i in range(10):
227             nn = R.nnet(x=R_df_x, y=R_df_y, size=5,
228                         maxit=500, decay=1e-4,
229                         abstol = 1.0e-10, reltol = 1.0e-20,
230                         trace=False)
231             if fit_value > nn.rx2('value')[0]:
232                 fit_value = nn.rx2('value')[0]
233                 fit_nn = nn
234         self.fit_nn = fit_nn
235         self._x_to_z_NN = fit_nn
236         self.garbage_collect()
237 
238     def _x_to_z(self):
239         # Arranges the Inputs to NN
240         nms = list(self.df_inputs.columns)
241         nms1 = np.array(nms)
242         if 'mc' in nms1:
243             nms.pop(np.argwhere(nms1 == 'mc'))
244         df_inputs = self.df_inputs.xs(np.sort(nms), axis=1)
245         df_pca = self.df_pca
246 
247         # Merge Inputs and PCA so that run_id matches
248         df_all = df_inputs.join(df_pca)
249 
250         # Split pandas DataFrame into Inputs and PCA
251         nms = list(df_all.columns)
252         nms1 = np.array(nms)
253         nms.pop(np.argwhere(nms1 == 'pca'))
254         pca = df_all.xs(['pca'], axis=1)
255 
256         df_inputs = df_all.xs(nms, axis=1)
257 
258         # R
```

```
259         R = robjects.r
260         importr('nnet')
261 
262         R_df_x = pandas.rpy.common.convert_to_r_dataframe(df_inputs)
263         R_df_y = pandas.rpy.common.convert_to_r_dataframe(pca)
264         fit_nn = None
265         fit_value = 1e10
266         for i in range(10):
267             nn = R.nnet(x=R_df_x, y=R_df_y, size=5, maxit=500,
268                         decay=1e-4, abstol = 1.0e-10,
269                         reltol = 1.0e-20, trace=False)
270             if fit_value > nn.rx2('value')[0]:
271                 fit_value = nn.rx2('value')[0]
272                 fit_nn = nn
273         self.fit_nn = fit_nn
274         self._x_to_z_NN = fit_nn
275         self.garbage_collect()
276 
277     def fn_X2Z(self, x_df):
278         assert isinstance(x_df, pd.DataFrame)
279         if self._x_to_z_NN is None:
280             myMsg = 'Neural Net for X to Z does not exist'
281             raise RuntimeError(myMsg)
282         # Check column names
283         col_nms = list(x_df.columns)
284         assert len(col_nms) == len(self._x_var_names)
285         if np.any(self._x_var_names != np.sort(col_nms)):
286             myMsg = 'Input var names do not match to existing ones'
287             raise RuntimeError(myMsg)
288         # R
289         R = robjects.r
290         df_inputs = x_df.xs(self._x_var_names, axis=1)
291         R_df_x = pandas.rpy.common.convert_to_r_dataframe(df_inputs)
292         pca_pred = R.predict(self._x_to_z_NN, newdata=R_df_x)
293         return pca_pred
294 
295     def predict(self, inputs, t):
296         pca = self.fn_X2Z(inputs)
297         myOut = []
298         for p in pca:
299             if p > 1:
300                 p = 1
301             elif p < 0:
302                 p = 0
303             new_y = self.fn_Z2Y(p, t)
304             myOut.append(new_y)
305         return myOut
306 
307     def _extract_features(self, n_components=None):
308         df_bp_pca = self._extract_PCA_Breakpoints()
309         var = df_bp_pca.columns.tolist()[0]
310         if var in self.df.columns.tolist():
311             self.df.update(df_bp_pca)
312         else:
```

```
313             self.df = pd.merge(self.df, df_bp_pca,
314                                left_index=True, right_index=True)
315
316     def _extract_PCA(self, n_components=2):
317         # Extract PCA
318         y = self.df.xs('outputs', axis=1).values
319         fit_pca = PCA(n_components=n_components)
320         X_pca = fit_pca.fit_transform(y)
321         X_pca = my_functions.normalize(X_pca)
322         var = range(n_components)
323         col_names = [''.join(('pca',str(i))) for i in var]
324         col_idx = pd.MultiIndex(levels=[['ApplyPCA'], col_names],
325                                 labels=[[0]*n_components,
326                                 range(n_components)],
327                                 names=['L1','L2'])
328         df = pd.DataFrame(X_pca, index=self.df.index,
329                           columns=col_idx)
330         return df
331
332     def _extract_PCA_Breakpoints(self):
333         # Extract Breakpoints PCA
334         segs = self.df.xs('attr', axis=1).xs('segs', axis=1).values
335         segs = [eval(row) for row in segs]
336         bp = FeatureExtract.breakpoint_formatting(segs,
337                                                   style='short')
338         fit_pca = PCA(n_components=1)
339         X_bp_pca = fit_pca.fit_transform(bp)
340         X_bp_pca = my_functions.normalize(X_bp_pca)
341
342         col_names =['pca0']
343         col_idx = pd.MultiIndex(levels=[['ApplyPCA_Breakpoints'],
344                                 col_names],
345                                 labels=[[0],[0]],
346                                 names=['L1','L2'])
347         df = pd.DataFrame(X_bp_pca, index=self.df.index,
348                           columns=col_idx)
349         return df
350
351     def _assign_pca(self):
352         ''' Perform PCA on data. '''
353         y = self.df.xs('outputs', axis=1).values
354         fit_pca = PCA(n_components=1)
355         X_pca = fit_pca.fit_transform(y)
356         X_pca = my_functions.normalize(X_pca)
357         df_pca = pd.DataFrame(X_pca, index=self.df.index,
358                               columns=[('attr','pca')])
359         if df_pca.columns[0] in self.df.columns:
360             self.df.update(df_pca)
361         else:
362             self.df = pd.merge(self.df, df_pca,
363                                left_index=True, right_index=True)
364
365     def _initial_segmentation(self):
366         ''' Segments the runs. '''
```

```
        # Grab Data
        mat = self.df.xs('outputs', axis=1).values
        sigmas = self.df.xs('attr', axis=1).xs('sigma',
                                               axis=1).values
        # Prep
        nr, nc = mat.shape
        t = range(nc)
        segs = [''] * nr
        nseg = [0] * nr
        # Segmentation
        for i, (y, sd) in enumerate(izip(mat, sigmas)):
            ts, ys = ts_segments.hybrid_aggr(t, y, sd)
            seg_index = ts_segments.return_break_index(t, ts)

            ref_seg = [x1[0] for x1 in seg_index]
            ref_seg = ref_seg[1:]

            ts, ys = ts_segments.optimize_segmentation(t, y,
                                                       ref_seg)
            seg_index = ts_segments.return_break_index(t, ts)

            segs[i] = str(seg_index)
            nseg[i] = len(seg_index)
        d = {('attr','segs'): segs, ('attr','nseg'): nseg}
        df_segs = pd.DataFrame.from_dict(d)
        df_segs.index = self.df.index
        # Save segmentation
        self.df = pd.merge(self.df, df_segs, left_index=True,
                           right_index=True)

    def _segmentation_by_group(self):
        ''' Segments the runs by groups. '''
        # get groupings
        grps = list(set(self.df.ix[:, ('attr','group')].values))

        for g in grps:
            # Get index of groups
            idx = self.df.index[self.df.xs(('attr','group'),
                                axis=1)==g]
            # Grab Data
            df_subset = self.df.ix[idx]
            mat = df_subset.xs('outputs', axis=1).values
            sigmas = df_subset.xs(('attr','sigma'), axis=1).values
            # Prep
            nr, nc = mat.shape
            t = range(nc)
            segs = [''] * nr
            nseg = [0] * nr
            # Segmentation
            for i, (y, sd) in enumerate(izip(mat, sigmas)):
                ts, ys = ts_segments.hybrid_aggr(t, y, sd)
                seg_index = ts_segments.return_break_index(t, ts)

                ref_seg = [x1[0] for x1 in seg_index]
```

```
421                 ref_seg = ref_seg[1:]
422
423                 ts, ys = ts_segments.optimize_segmentation(t, y,
424                                                            ref_seg)
425                 seg_index = ts_segments.return_break_index(t, ts)
426
427                 segs[i] = str(seg_index)
428                 nseg[i] = len(seg_index)
429             d = {('attr','segs'): segs, ('attr','nseg'): nseg,
430                  ('attr','group'):g}
431             df_segs = pd.DataFrame.from_dict(d)
432             df_segs.index = idx
433             # Save segmentation
434             self.df.update(df_segs, 'left')
435
436     def _group_data(self):
437         ''' Cluster data using DBScan and line_regression on
438             PCA and PCA(Breakpoints). '''
439         df1 = self.df.xs('ApplyPCA', axis=1)
440         df2 = self.df.xs('ApplyPCA_Breakpoints', axis=1)
441         df = pd.concat({'ApplyPCA': df1,
442                         'ApplyPCA_Breakpoints': df2},
443                        names=['L1','L2'], axis=1)
444         print '((( features for clustering: ', df.columns, ')))'
445         lbls = ApplyClustering.model_based_clustering(df)
446         lbls = np.array(lbls, dtype=int)
447         df_grp = pd.DataFrame(lbls, index=df.index,
448                               columns=[('attr','group')])
449         if ('attr','group') in self.df.columns.tolist():
450             self.df.update(df_grp)
451         else:
452             self.df = pd.merge(self.df, df_grp, left_index=True,
453                                right_index=True)
454
455     def _check_group_nseg(self):
456         ''' This checks to see if the number of segments is the
457             same across a group '''
458         groups = list(set(self.df.ix[:, ('attr','group')]))
459         # loop over groups
460         for g in groups:
461             if g == -1: continue
462             m = self.df[self.df.ix[:,('attr','group')]==g]
463             nsegs = m.ix[:,('attr','nseg')].values
464             # If just one nseg, skip
465             if np.all(nsegs == nsegs[0]):
466                 continue
467             else:
468                 # Find the most common nseg --> dominating
469                 nseg_dom = -1
470                 nseg_dom_ind = np.array([])
471                 nseg_options = list(set(nsegs))
472                 for ns in nseg_options:
473                     x = np.argwhere(nsegs == ns).ravel()
474                     if len(x) > len(nseg_dom_ind):
```

```
                        nseg_dom_ind = x
                        nseg_dom = ns
                # If  most common nseg is 1 segment,
                # make all else nseg=1
                if nseg_dom == 1:
                    idx = list(m.index)
                    segs = m.ix[:,('attr','segs')].values
                    new_seg = eval(segs[nseg_dom_ind[0]])
                    for i in range(len(m)):
                        if i not in nseg_dom_ind:
                            self.df.ix[idx[i], ('attr','segs')] = \
                                    str(new_seg)
                            self.df.ix[idx[i], ('attr','nseg')] = 1
                # Otherwise, enforce nseg on the other ones using
                # optimized segmentation
                else:
                    nseg_dom_ind2 = m.index[nseg_dom_ind]
                    pcas = m.ix[:, ('attr',
                                    'pca')].values[nseg_dom_ind]
                    m_ = m.xs('attr', axis=1).xs(['pca', 'nseg'],
                                                 axis=1)
                    for i, v in m.iterrows():
                        if i not in nseg_dom_ind2:
                            arg = my_functions.arg_closest(
                                      v[('attr', 'pca')], pcas)
                            arg = nseg_dom_ind[arg]
                            ref_seg = eval(m.ix[m.index[arg],
                                           ('attr','segs')])
                            ref_seg = [x1[0] for x1 in ref_seg]
                            ref_seg = ref_seg[1:]
                            y = m.xs('outputs',
                                     axis=1).ix[i,:].values
                            t = range(len(y))
                            ts, ys = \
                            ts_segments.optimize_segmentation(t, y,
                                                              ref_seg)
                            seg_index = \
                            ts_segments.return_break_index(t, ts)
                            self.df.ix[i, ('attr','segs')] = \
                            str(seg_index)
                            self.df.ix[i, ('attr','nseg')] = \
                            nseg_dom

    def _find_good_neighbors(self, x, neighs, grps,
                             n=5, outlier= -1):
        ind = np.argwhere(x == neighs)
        upper = []
        lower = []
        i9, j9 = 0, 0
        while len(upper) < n:
            i9 += 1
            ind1 = ind + i9 + j9
            try:
                up1 = grps[ind1]
```

```
                if up1 != outlier:
                    upper.append(np.asscalar(ind1))
                else:
                    j9 += 1
            except:
                break
        i9, j9 = 0, 0
        while len(lower) < n:
            i9 += 1
            ind1 = ind - i9 - j9
            try:
                lo1 = grps[ind1]
                if lo1 != outlier:
                    lower.append(np.asscalar(ind1))
                else:
                    j9 += 1
            except:
                break
        return lower, upper
    def _new_label(self, ind, pca, seg, that_mat, outlier= -1):
        ''' This function finds a group with the same number
            of segments
        '''
        mat10 = that_mat.copy()
        # Sort based on closest PCA
        mat10[:, 2] = np.abs(mat10[:, 2] - pca)
        mat10_sort = mat10[mat10[:, 2].argsort()]
        seg10 = seg
        new_group = None
        i = 0
        for i in range(1, len(mat10_sort)):
            if mat10_sort[i, 3] != outlier:
                if mat10_sort[i, 1] == seg10:
                    new_group = mat10_sort[i, 3]
                    return new_group
        # If no group is found, make a new one
        groups = self.tsDB.groups
        new_group = max(groups) + 1
        return new_group

    def _clean_up_groups_2(self):
        ''' This module takes care of overlaps between groups '''
        ## Prep Data
        attr = self.df.xs('attr',axis=1)
        nseg_pca_grp = attr.xs(['nseg','pca','group'], axis=1)
        runid = pd.DataFrame(self.df.index, columns=['run_id'])
        Runid_nseg_pca_grp = runid.join(nseg_pca_grp)

        my_order =  Runid_nseg_pca_grp.values
        my_order2 = Runid_nseg_pca_grp.values

        groups = list(set(self.df.ix[:, ('attr','group')].values))

        # Take care of overlaps
```

```
583         gs11 = groups
584         min_max = []
585         nseg11 = []
586         # Get the upper and lower pca values for each group
587         for gr in gs11:
588             if gr == -1:
589                continue
590             temp = Runid_nseg_pca_grp[ \
591                    Runid_nseg_pca_grp.xs('group', axis=1) == gr]
592             mi = np.min(temp.xs('pca', axis=1).values)
593             ma = np.max(temp.xs('pca', axis=1).values)
594             ns = temp.xs('nseg', axis=1).values[0]
595             min_max.append([gr, mi, ma])
596             nseg11.append(ns)
597         min_max = np.array(min_max)
598         nseg11 = np.array(nseg11)
599         new_order = np.argsort(min_max[:, 1], axis=0).ravel()
600         min_max12 = min_max[new_order, :]
601         gs12 = np.array(min_max12[:,0], dtype=int)
602         min_max12 = min_max12[:,[1,2]]
603         nseg12 = nseg11[new_order]
604
605         # Determine which groups are overlapping
606         overlap = []
607         for i in range(len(min_max12) - 1):
608             if min_max12[i, 1] > min_max12[i + 1, 0]:
609                 overlap.append(i)
610         if len(overlap) > 0:
611             for i in overlap:
612                 # check if range of [i+1] is within the range of [i]
613                 if min_max12[i, 1] > min_max12[i + 1, 1]:
614                     # determine which has a larger nseg
615                     if nseg12[i] < nseg12[i + 1]:
616                         # range of [i+1] is within
617                         # the range of [i] and nseg of [i+1]
618                         # is larger so raise nseg of [i] that
619                         # overlaps with [i+1]
620                         ind12_0 = np.argwhere(my_order2[:, 2] >= \
621                         min_max12[i + 1, 0]).ravel()
622                         ind12_1 = np.argwhere(my_order2[:, 2] <= \
623                         min_max12[i + 1, 1]).ravel()
624                         ind12_2 = np.argwhere(my_order2[:, 3] == \
625                         gs12[i]).ravel()
626                         change_index = np.intersect1d(ind12_0,
627                                                       ind12_1)
628                         change_index = np.intersect1d(change_index,
629                                                       ind12_2)
630                         for i12 in change_index:
631                             my_order2[i12, 1] = nseg12[i + 1]
632                             my_order2[i12, 3] = gs12[i + 1]
633                         # split [i] into 2 groups
634                         ind12_3 = np.argwhere(my_order2[:, 2] > \
635                         min_max12[i + 1, 1]).ravel()
636                         ind12_4 = np.argwhere(my_order2[:, 3] == \
```

```
637                         gs12[i]).ravel()
638                         change_index = np.intersect1d(ind12_3,
639                                                       ind12_4)
640                         new_g12 = int(np.max(gs12) + 1)
641                         gs12 = np.hstack((gs12, new_g12))
642                         for i12 in change_index:
643                             my_order2[i12, 3] = new_g12
644                     else:  # nseg12[i] > nseg12[i+1]:
645                         # This means range of [i+1] is within
646                         # range of [i] and nseg is lower
647                         # so [i+1] will get absorbed by [i]
648                         ind12_ = np.argwhere(my_order2[:, 3] == \
649                         gs12[i + 1])
650                         for i12 in ind12_:
651                             my_order2[i12, 1] = nseg12[i]
652                             my_order2[i12, 3] = gs12[i]
653                 else:  # min_max12[i,1] <= min_max12[i+1,1]:
654                     if nseg12[i] < nseg12[i + 1]:
655                         ind12_0 = np.argwhere(my_order2[:, 2] >= \
656                         min_max12[i + 1, 0]).ravel()
657                         ind12_1 = np.argwhere(my_order2[:, 2] <= \
658                         min_max12[i + 1, 1]).ravel()
659                         ind12_2 = np.argwhere(my_order2[:, 3] == \
660                         gs12[i]).ravel()
661                         change_index = np.intersect1d(ind12_0,
662                                                       ind12_1)
663                         change_index = np.intersect1d(change_index,
664                                                       ind12_2)
665                         for i12 in change_index:
666                             my_order2[i12, 1] = nseg12[i + 1]
667                             my_order2[i12, 3] = gs12[i + 1]
668
669                         gr1 = Runid_nseg_pca_grp[ \
670                             Runid_nseg_pca_grp.xs('group',
671                             axis=1).values==gs12[i]]
672                         gr2 = Runid_nseg_pca_grp[ \
673                             Runid_nseg_pca_grp.xs('group',
674                             axis=1).values==gs12[i+1]]
675                         idx1, idx2 = my_functions.find_overlap( \
676                             gr1.xs('pca',axis=1).values,
677                             gr2.xs('pca',axis=1).values)
678                         pca1 = gr1.xs('pca',axis=1).values
679                         pca2 = gr2.xs('pca',axis=1).values
680                     else:  # nseg12[i] >= nseg12[i+1]:
681                         ind12_0 = np.argwhere(my_order2[:, 2] >= \
682                         min_max12[i, 0]).ravel()
683                         ind12_1 = np.argwhere(my_order2[:, 2] <= \
684                         min_max12[i, 1]).ravel()
685                         ind12_2 = np.argwhere(my_order2[:, 3] == \
686                         gs12[i + 1]).ravel()
687                         change_index = np.intersect1d(ind12_0,
688                                                       ind12_1)
689                         change_index = np.intersect1d(change_index,
690                                                       ind12_2)
```

```
691                         for i12 in change_index:
692                             my_order2[i12, 1] = nseg12[i]
693                             my_order2[i12, 3] = gs12[i]
694         if len(overlap) > 0:
695             # Apply new segmentation
696             # find where my_order differs from my_order2
697             new_seg13 = np.argwhere(my_order2[:,3]!=my_order[:,3])
698             new_seg13 = new_seg13.ravel()
699 #            print 'len(new_seg13):',len(new_seg13)
700 #            print 'new_seg13:', new_seg13
701             for i in new_seg13:
702                 run_id, new_nseg, pca, grp = my_order2[i, :]
703 #                print type(new_nseg)
704                 y = self.df.xs('outputs',axis=1).ix[run_id,
705                                                     :].values
706                 t = range(len(y))
707 #                ts, ys = ts_segments.bottom_up_aggr(t, y,
708                                                      nseg=new_nseg)
709                 ts,ys = ts_segments.hybrid_aggr(t, y,
710                                                 nseg=int(new_nseg))
711                 seg_index = ts_segments.return_break_index(t, ts)
712                 self.df.ix[run_id, ('attr','segs')] = str(seg_index)
713                 self.df.ix[run_id, ('attr','nseg')] = len(seg_index)
714                 self.df.ix[run_id, ('attr','group')] = grp
715     def _clean_up_groups(self):
716         # Prep Data
717         cases = self.tsDB.get_case_numbers()
718         nseg = self.tsDB.get_val_each_case('db_attr', 'nseg')
719         nseg = [row[0] for row in nseg.values()]
720         pcas = self.tsDB.get_val_each_case('db_attr', 'pca')
721         pcas = [row[0] for row in pcas.values()]
722         grps = self.tsDB.get_val_each_case('db_attr', 'group')
723         grps = [row[0] for row in grps.values()]
724         # Create a matrix of [case#, nseg, pca, grp]
725         my_order = np.array((cases, nseg, pcas, grps))
726         my_order = my_order.T
727
728         # Take Care of Outliers
729         # sort by pca
730         my_order_pca = my_order[my_order[:, 2].argsort()]
731         # find where group == -1, outlier
732         k_outlier = np.arange(len(my_order))[my_order_pca[:,
733                                                           3] == -1]
734
735
736         if len(k_outlier) > 0:
737             new_seg_9 = []
738             for i3 in k_outlier:
739                 good_neighs = self._find_good_neighbors( \
740                         my_order_pca[i3, 2],
741                         my_order_pca[:, 2],
742                         my_order_pca[:, 3])
743                 if len(good_neighs[0]) > 0:
744                     l9 = my_order_pca[np.array(good_neighs[0]), 1]
```

```
745                 else:
746                     l9 = [0]
747                 if len(good_neighs[1]) > 0:
748                     r9 = my_order_pca[np.array(good_neighs[1]), 1]
749                 else:
750                     r9 = [0]
751                 new_9 = int(np.max((np.median(l9), np.median(r9))))
752                 new_seg_9.append(list(my_order_pca[i3,
753                                       :]) + [i3, new_9])
754             new_seg_9 = np.array(new_seg_9)
755             new_gs = []
756             for row in new_seg_9:
757                 new_g = self._new_label(*row[[4, 2, 5]],
758                                         that_mat=my_order_pca)
759                 new_gs.append(new_g)
760
761 #             print 'k_outlier',k_outlier
762 #             print 'new_seg_9',new_seg_9
763
764             # update the nseg and grouping of the
765             # temporary matrix my_order
766             new_gs_ = np.array(new_gs).reshape((-1, 1))
767             new_seg_10 = np.hstack((new_seg_9, new_gs_))
768             my_order2 = my_order.copy()
769             for row in new_seg_10:
770                 ind11 = int(row[0])
771                 my_order2[ind11, 1] = row[5]
772                 my_order2[ind11, 3] = row[6]
773         else:
774             my_order2 = my_order.copy()
775
776         # Take care of overlaps
777         gs11 = list(set(my_order2[:, 3]))  # sort by group number
778         gs11.sort()
779         min_max = []
780         nseg11 = []
781         # Get the upper and lower pca values for each group
782         for i in gs11:
783             temp = my_order2[np.argwhere(my_order2[:,
784                                  3] == i).flatten()]
785             mi = np.min(temp[:, 2])
786             ma = np.max(temp[:, 2])
787             ns = temp[0, 1]
788             min_max.append([mi, ma])
789             nseg11.append(ns)
790         min_max = np.array(min_max)
791         nseg11 = np.array(nseg11)
792         new_order = np.argsort(min_max[:, 0], axis=0)
793         min_max12 = min_max[new_order]
794         nseg12 = nseg11[new_order]
795         gs12 = np.array(gs11)[new_order]
796
797         # Determine which groups are overlapping
798         overlap = []
```

```
        for i in range(len(min_max12) - 1):
            if min_max12[i, 1] > min_max12[i + 1, 0]:
                overlap.append(i)
        if len(overlap) > 0:
            for i in overlap:
                # check if range of [i+1] is within the range of [i]
                if min_max12[i, 1] > min_max12[i + 1, 1]:
                    # determine which has a larger nseg
                    if nseg12[i] < nseg12[i + 1]:
                        # range of [i+1] is within the
                        # range of [i] and nseg of [i+1]
                        # is larger so raise nseg of
                        # [i] that overlaps with [i+1]
                        ind12_0 = np.argwhere(my_order2[:, 2] >= \
                        min_max12[i + 1, 0]).ravel()
                        ind12_1 = np.argwhere(my_order2[:, 2] <= \
                        min_max12[i + 1, 1]).ravel()
                        ind12_2 = np.argwhere(my_order2[:, 3] == \
                        gs12[i]).ravel()
                        change_index = np.intersect1d(ind12_0,
                                                      ind12_1)
                        change_index = np.intersect1d(change_index,
                                                      ind12_2)
                        for i12 in change_index:
                            my_order2[i12, 1] = nseg12[i + 1]
                            my_order2[i12, 3] = gs12[i + 1]
                        # split [i] into 2 groups
                        ind12_3 = np.argwhere(my_order2[:, 2] > \
                        min_max12[i + 1, 1]).ravel()
                        ind12_4 = np.argwhere(my_order2[:, 3] == \
                        gs12[i]).ravel()
                        change_index = np.intersect1d(ind12_3,
                                                      ind12_4)
                        new_g12 = int(np.max(gs12) + 1)
                        gs12 = np.hstack((gs12, new_g12))
                        for i12 in change_index:
                            my_order2[i12, 3] = new_g12
                    else:  # nseg12[i] > nseg12[i+1]:
                        # This means range of [i+1] is
                        # within range of [i] and nseg is lower
                        # so [i+1] will get absorbed by [i]
                        ind12_ = np.argwhere(my_order2[:,
                                             3] == gs12[i + 1])
                        for i12 in ind12_:
                            my_order2[i12, 1] = nseg12[i]
                            my_order2[i12, 3] = gs12[i]
                else:  # min_max12[i,1] <= min_max12[i+1,1]:
                    if nseg12[i] < nseg12[i + 1]:
                        ind12_0 = np.argwhere(my_order2[:, 2] >= \
                        min_max12[i + 1, 0]).ravel()
                        ind12_1 = np.argwhere(my_order2[:, 2] <= \
                        min_max12[i + 1, 1]).ravel()
                        ind12_2 = np.argwhere(my_order2[:, 3] == \
                        gs12[i]).ravel()
```

```
                        change_index = np.intersect1d(ind12_0,
                                                      ind12_1)
                        change_index = np.intersect1d(change_index,
                                                      ind12_2)
                        for i12 in change_index:
                            my_order2[i12, 1] = nseg12[i + 1]
                            my_order2[i12, 3] = gs12[i + 1]
                    else:  # nseg12[i] >= nseg12[i+1]:
                        ind12_0 = np.argwhere(my_order2[:, 2] >= \
                        min_max12[i, 0]).ravel()
                        ind12_1 = np.argwhere(my_order2[:, 2] <= \
                        min_max12[i, 1]).ravel()
                        ind12_2 = np.argwhere(my_order2[:, 3] == \
                        gs12[i + 1]).ravel()
                        change_index = np.intersect1d(ind12_0,
                                                      ind12_1)
                        change_index = np.intersect1d(change_index,
                                                      ind12_2)
                        for i12 in change_index:
                            my_order2[i12, 1] = nseg12[i]
                            my_order2[i12, 3] = gs12[i]
        if (len(k_outlier) > 0) or (len(overlap) > 0):
            # Apply new segmentation
            # find where my_order differs from my_order2
            new_seg13 = np.argwhere(my_order2[:,1]!=my_order[:,1])
            for i in new_seg13:
                ddd = self.tsDB.get_by_case(int(i))
                for d in ddd.values():
                    t = d['t']
                    y = d['y']
                    ts, ys = ts_segments.bottom_up_aggr(t, y,
                                        nseg=my_order2[i, 1])
                    seg_index = ts_segments.return_break_index(t,ts)
                    self.tsDB.change_segs(segs=seg_index,
                                          run_id=d['run_id'])
            # Change Groups
            case_run_id = self.tsDB.get_val_each_case('db_attr',
                                                      'run_id')
            for row in my_order2:
                for i in case_run_id[row[0]]:
                    self.tsDB.change_group(grp=row[3], run_id=i)
    def _check_and_fix_group_membership(self):
        ''' sqlite3 can only handle 999 sql variables per retrieval
            so this is tocheck  to make keep the number of runs per
            group to an acceptable level
        '''
        pca = self.tsDB.get_val_all('db_attr', 'pca', 'data frame')
        groups = self.tsDB.get_val_all('db_attr', 'group',
                                       'data frame')
        cases = self.tsDB.get_val_all('db_attr', 'case',
                                      'data frame')
        groups_pca = pandas.merge(cases, groups, on='run_id')
        groups_pca = pandas.merge(groups_pca, pca, on='run_id')
        groups_pca = groups_pca.values
```

```
907         group_counts = Counter(groups.values[:, 1])
908 #         print 'group_counts:', group_counts
909 
910         for k, v in group_counts.items():
911             if k == -1: continue
912             # Check if the number of items is greater than the limit
913             if v > 900:
914                 # Split the group into n groups
915                 n_split = int(np.ceil(len(groups_pca) / 900))
916                 gr_pca = groups_pca[np.argwhere(groups_pca[:,
917                                                 2] == k).ravel()]
918                 gr_pca0 = gr_pca[gr_pca[:, 3].argsort()]
919                 n_gr_pca = np.array_split(gr_pca0, n_split)
920 #                 print 'n_split:', n_split
921 #                 print 'n_gr_pca:', n_gr_pca
922                 groups = self.tsDB.groups
923                 for i, S in enumerate(n_gr_pca[1:]):
924                     group_number = groups[-1] + i + 1
925                     for run_id in S[:,0]:
926                         self.tsDB.change_group(grp=group_number,
927                                                run_id=run_id)
928     def _give_me_vals(self, target, ranges, vals):
929         assert len(ranges) == len(vals) - 1
930         for i, x in enumerate(ranges):
931             if target < x:
932                 return vals[i]
933         return vals[-1]
934 
935     def _create_regressions_segments(self, grps):
936         ''' Creates a piecewise function that can
937             be regressed by PCA values
938         '''
939         # Create Regressions by Groups by Segments
940         fits = dict([(x, {}) for x in grps])
941         R = robjects.r
942         mfe = 0.
943         for g in grps:
944             df_g = self.df[self.df.ix[:, ('attr','group')]==g]
945             nr, nc = df_g.shape
946             segs = df_g.ix[:, ('attr','segs')].values
947             segs = [eval(row) for row in segs]
948             nseg = list(set(df_g.ix[:, ('attr','nseg')].values))
949             if len(nseg) > 1:
950                 myMsg1 = 'Making Regressions: nseg of '
951                 myMsg2 = 'group {} is not uniform'.format(g)
952                 raise RuntimeError(myMsg1 + myMsg2)
953             nseg = int(nseg[0])
954             ys = df_g.xs('outputs',axis=1).values
955             tt = range(len(ys[0]))
956             pcas = df_g.ix[:, ('attr','pca')].values
957 
958             for i_seg in range(nseg):
959                 d = {'y': [], 't': [], 'pca': []}
960                 m = pd.DataFrame.from_dict(d)
```

```
                for i_row, row in enumerate(ys):
                    st, en = segs[i_row][i_seg]
                    if st == en:
                        continue
                    y = row[st:en]
                    t = range(st,en)
                    p = pcas[i_row]
                    d = {'y': y, 't': t, 'pca': p}
                    df = pd.DataFrame.from_dict(d)
                    m = pd.concat((m,df), axis=0, ignore_index=True)
#                print m.head(20)

                R_df = pandas.rpy.common.convert_to_r_dataframe(m)
                fit_lm = R.lm('y~pca+t+pca*t', data=R_df)
                # fit_lm = R.lm('y~pca+t+I(t^2)+pca*t',data=R_df)
                # fit_lm = R.lm('y~pca+I(pca^2)+t+pca*t',data=R_df)
                fits[g][i_seg] = fit_lm
                # print 'g,seg: ', g, seg
        return fits

    def _create_regressions_breakpoints(self, grps):
        # Create Regressions for intersections of segments
        R = robjects.r

        brks = dict([(x, {}) for x in grps])
        for g in grps:
            # Get data by group
            df_g = self.df[self.df.ix[:, ('attr','group')]==g]
            nr, nc = df_g.shape
            # get breakpoint locations
            segs = df_g.ix[:, ('attr','segs')].values
            segs = [eval(row) for row in segs]
            nseg = list(set(df_g.ix[:, ('attr','nseg')].values))
            if len(nseg) > 1:
                myMsg1 = 'Making Regressions: nseg of '
                myMsg2 = 'group {} is not uniform'.format(g)
                raise RuntimeError(myMsg1 + myMsg2)
            nseg = int(nseg[0])
            pcas = df_g.ix[:, ('attr','pca')].values

            # Collect Data
            for i_seg in range(nseg):
                pca_beg = np.zeros((nr,2))
                pca_end = np.zeros((nr,2))
                for i_row, row in enumerate(segs):
                    pca_beg[i_row, 1] = row[i_seg][0]
                    pca_beg[i_row, 0] = pcas[i_row]
                    pca_end[i_row, 1] = row[i_seg][1]
                    pca_end[i_row, 0] = pcas[i_row]
                # Create regressions for the segment beginnings
                df_br = pd.DataFrame(pca_beg, columns=['pca','br'])
                cmn = pandas.rpy.common
                R_df_br = cmn.convert_to_r_dataframe(df_br)
                fit = R.lm('br~pca', data=R_df_br)
```

```
1015                brks[g][i_seg] = fit
1016            # Create one regression for the end
1017            df_br = pd.DataFrame(pca_end, columns=['pca','br'])
1018            cmn = pandas.rpy.common
1019            R_df_br = cmn.convert_to_r_dataframe(df_br)
1020            fit = R.lm('br~pca', data=R_df_br)
1021            brks[g][i_seg+1] = fit
1022        return brks
1023
1024    def _create_regressions(self, grps=None):
1025        ''' Creates a piecewise function that
1026            can be regressed by PCA values
1027        '''
1028        if grps is None:
1029            grps = list(set(self.df.ix[:, ('attr','group')].values))
1030        if -1 in grps:
1031            grps.remove(-1)
1032        fits = self._create_regressions_segments(grps)
1033        brks = self._create_regressions_breakpoints(grps)
1034
1035        # Create the PCA to group relationship
1036        ranges14 = []
1037        vals14 = []
1038        for g in grps:
1039            df_g = self.df[self.df.ix[:, ('attr','group')]==g]
1040            ranges14.append(df_g.ix[:, ('attr','pca')].max())
1041            vals14.append(g)
1042        ranges15 = np.array(ranges14)
1043        vals15 = np.array(vals14)
1044        ranges16 = ranges15[ranges15.argsort()]
1045        vals16 = vals15[ranges15.argsort()]
1046        ranges16 = ranges16[:-1]
1047
1048        print ranges16
1049        print vals16
1050
1051        self._z_to_y_breakpoints = brks
1052        self._z_to_y_regressions = fits
1053        self._z_to_y_pca2groups = lambda \
1054                    pca: self._give_me_vals(pca, ranges16,
1055                                            vals16)
1056 #        # Create Regression Function
1057
1058    def fn_Z2Y(self, pca, t):
1059        ''' Returns the predicted values '''
1060        # Stuff in the Beginning
1061        pca = float(pca)
1062        assert pca >= 0
1063        assert pca <= 1
1064        R = robjects.r
1065        t = np.array(t, dtype=np.float)
1066        g = self._z_to_y_pca2groups(pca)
1067        g_fits = self._z_to_y_regressions[g]
1068        g_brks = self._z_to_y_breakpoints[g]
```

```
1069          # Calculate the Breaks
1070          my_breaks = []
1071          for i in range(len(g_brks)):
1072              X_1 = pandas.DataFrame([pca], columns=['pca'])
1073              R_X_1 = pandas.rpy.common.convert_to_r_dataframe(X_1)
1074              bk = R.predict(g_brks[i], R_X_1)[0]
1075              my_breaks.append(bk)
1076          # if the breakpoints are not
1077          # monotonically increasing, don't use it
1078          use_these = [i for i in range(1,
1079                  len(my_breaks)) if my_breaks[i] > my_breaks[i - 1]]
1080          use_these = [0] + use_these
1081          # Predict
1082   #       pw = np.array([])
1083          pw = np.zeros(len(t))
1084          for i0, i1 in zip(use_these[0:-1], use_these[1:]):
1085              st = my_breaks[i0]
1086              en = my_breaks[i1]
1087              use_fit = g_fits[i1 - 1]
1088              locs = np.all((t >= st, t < en), axis=0)
1089              t_args = np.argwhere(locs)
1090              t_seg = t[t_args]
1091              dat1 = {'t':t_seg.flatten(), 'pca':pca}
1092              X_1 = pandas.DataFrame(dat1)
1093              R_X_1 = pandas.rpy.common.convert_to_r_dataframe(X_1)
1094              y_out = R.predict(use_fit, R_X_1)
1095  #             pw = np.hstack((pw,np.array(y_out)))
1096              pw[locs] = np.array(y_out)
1097          return pw
1098
1099      def _fit_error(self, test_data=None):
1100          if test_data is None:
1101              if self.test_data is None:
1102                  raise RuntimeError('No Test Data')
1103              else:
1104                  test_data = self.test_data
1105          fit_err = 0.
1106          n = 0.
1107          inputs = test_data.xs('inputs', axis=1)
1108          col_names = list(inputs.columns)
1109  #         print col_names
1110  #         col_names.remove('run_id')
1111          if 'mc' in col_names:
1112              col_names.remove('mc')
1113          inputs = inputs.xs(col_names, axis=1)
1114          for i in range(len(inputs)):
1115  #             d = test_data.get_run(i)
1116              y_test = test_data.xs('outputs', axis=1).ix[i,:].values
1117              t = range(len(y_test))
1118              x = inputs.take([i])
1119  #             print x
1120              y_pred = self.predict(x, t)
1121  #             print len(np.array(d['y'])),np.array(d['y'])
1122  #             print len(y_est[0]),y_est[0]
```

```
1123            err = y_test - y_pred[0]
1124            fit_err += sum(err ** 2)
1125            n += len(err)
1126        return fit_err, n
1127
1128    def MSE(self, test_data=None):
1129        fit_err, n = self._fit_error(test_data)
1130        return fit_err / float(n)
1131
1132    def MFE(self):
1133        mfe = 0.
1134        for fit in self._z_to_y_regressions.values():
1135            for seg in fit.values():
1136                mfe += np.sum(np.array(seg.rx2('residuals'))**2)
1137        return mfe
1138
1139    def MRE(self, test_data=None):
1140        if test_data is None:
1141            test_data = self.test_data
1142        fit_err, n = self._fit_error(test_data)
1143        return fit_err
1144
1145    def mplot_regressions_vs_actual(self, nr=4, nc=5, fig=None,
1146                                    font_settings=None,
1147                                    show_text=True,
1148                                    df_pca=None, df_outs=None):
1149        nr15 = nr
1150        nc15 = nc
1151        sweep = np.linspace(0, 1, nr15 * nc15)
1152        if fig is None:
1153            fig = plt.figure(figsize=(2 * nc15, 2 * nr15))
1154        if font_settings is None:
1155            font_settings = {'fontsize':14, 'fontname':'sans'}
1156
1157        if show_text:
1158            spacing = {'hspace':.3}
1159        else:
1160            spacing = {'hspace':.1, 'wspace':.1}
1161        gs = gridspec.GridSpec(nr15, nc15, **spacing)
1162
1163#        id_pca = self.tsDB.get_val_all('db_attr', 'pca')
1164        if df_pca is None:
1165            pcas = self.df.ix[:, ('attr','pca')].values
1166            df_outs = self.df.xs('outputs',axis=1)
1167        else:
1168            pcas = df_pca.values
1169
1170        y_min = self.df.xs('outputs', axis=1).min().min()
1171        y_max = self.df.xs('outputs', axis=1).max().max()
1172        y_min -= np.abs(y_min)*.15
1173        y_max += np.abs(y_max)*.15
1174        for i, v in enumerate(sweep):
1175            ax = plt.subplot(gs[i])
1176
```

```
1177             # plot original
1178             run_id = my_functions.arg_closest(v, pcas)
1179             closest_pca = pcas[run_id]
1180             y = df_outs.ix[run_id, :].values
1181             t = range(len(y))
1182 #            ax.plot(t, y, lw=1, c='r')#c='0.8')
1183             ax.plot(t, y, '.', c='k')#c='0.8')
1184 
1185             # plot fit
1186             y = self.fn_Z2Y(v, t)
1187             ax.plot(t, y, c='r',lw=1)
1188             ax.plot(t[0],y[0],'o')
1189             if show_text:
1190                 xpos = (t[0] + t[-1]) / 2.
1191                 ax.text(xpos, y_max*1.25,
1192                         'Act.= '+str(round(closest_pca, 3)),
1193                         ha='center', **font_settings)
1194                 ax.text(xpos, y_max * 1.05,
1195                         'PCA = ' + str(round(v, 3)),
1196                         ha='center', **font_settings)
1197 
1198             plt.xticks([])
1199             plt.yticks([])
1200             plt.ylim([y_min, y_max])
1201 #        plt.tight_layout()
1202         return fig
1203     def plot_pca_1D(self, ts_resample='auto', dot_size=0, fig=None,
1204                     font_settings=None):
1205         ''' plots the image of the time series along
1206             the first PCA axis with equal spacing
1207         '''
1208         yvals = self.df.xs('outputs', axis=1).values
1209         minimum = np.min(np.min(yvals))
1210         minimum -= np.abs(minimum)*.1
1211         maximum = np.max(np.max(yvals))
1212         if ts_resample == 'auto':
1213             L = len(yvals[0, :])
1214             ts_resample = int(L / 65)
1215 #            print L,ts_resample
1216 #        pca = self.tsDB.get_val_all('db_attr', 'pca')
1217         pca = self.df.xs('attr',axis=1).xs('pca',axis=1).values
1218         X = pca
1219 
1220         x_min, x_max = np.min(X, 0), np.max(X, 0)
1221         X = (X - x_min) / (x_max - x_min)
1222 
1223         sweep = np.linspace(0, 1, 11)
1224         if fig is None:
1225             fig = plt.figure(figsize=(20, 5))
1226         if font_settings is None:
1227             font_settings = {'fontname': 'sans', 'fontsize': 12}
1228         ax = plt.subplot(111)
1229 #        yloc = np.sin(np.arange(len(X)) * 2) / 4
1230         yloc = np.random.uniform(-.25, .25, len(X))
```

```
1231            for i in range(len(X)):
1232                plt.plot(X[i], yloc[i], 'o')
1233
1234            if hasattr(offsetbox, 'AnnotationBbox'):
1235                # only print thumbnails with matplotlib > 1.0
1236                for sw in sweep:
1237                    i = my_functions.arg_closest(sw, X)
1238    #                d = self.tsDB.get_run(i)
1239                    y = self.df.xs('outputs',axis=1).ix[i,:].values
1240                    img = my_functions.make_ts_img(y, ts_resample,
1241                                                   dot_size,
1242                                                   minimum, maximum)
1243                    imagebox = offsetbox.AnnotationBbox(
1244                        offsetbox.OffsetImage(img,
1245                        cmap=plt.cm.gray_r, zoom=1),
1246                        (X[i], .8))
1247                    ax.add_artist(imagebox)
1248            plt.xlim([-.1, 1.1])
1249            plt.ylim((-.5, 1.5))
1250            # plt.xticks([])
1251            plt.yticks([])
1252            plt.setp(ax.get_xticklabels(),**font_settings)
1253            plt.xlabel('PC 1', **font_settings)
1254            return fig
1255
1256        def plot_pca_1D_hist(self, ts_resample='auto', dot_size=0,
1257                             fig=None, font_settings=None):
1258            ''' plots the image of the time series along the
1259                first PCA axis with equal spacing '''
1260            yvals = self.df.xs('outputs', axis=1).values
1261            minimum = np.min(np.min(yvals))
1262            minimum -= np.abs(minimum)*.1
1263            maximum = np.max(np.max(yvals))
1264            if ts_resample == 'auto':
1265                L = len(yvals[0, :])
1266                ts_resample = int(L / 65)
1267    #             print L,ts_resample
1268    #         pca = self.tsDB.get_val_all('db_attr', 'pca')
1269            pca = self.df.xs('attr',axis=1).xs('pca',axis=1).values
1270            X = pca
1271
1272            x_min, x_max = np.min(X, 0), np.max(X, 0)
1273            X = (X - x_min) / (x_max - x_min)
1274
1275            sweep = np.linspace(0, 1, 11)
1276            if fig is None:
1277                fig = plt.figure(figsize=(20, 5))
1278            if font_settings is None:
1279                font_settings = {'fontname': 'sans', 'fontsize': 12}
1280            ax2 = plt.subplot(212)
1281    #         yloc = np.sin(np.arange(len(X)) * 2) / 4
1282    #         yloc = np.random.uniform(-.25, .25, len(X))
1283    #         for i in range(len(X)):
1284    #             plt.plot(X[i], yloc[i], 'o')
```

```
1285
1286            ax2.hist(X, bins=50, normed=False, color='b', edgecolor='b')
1287            plt.xlim([-.1, 1.1])
1288    #        ax2.set_yticklabels([])
1289    #        ax2.set_yticks([])
1290            ax2.yaxis.labelpad = -10
1291            plt.xlabel('PC 1', **font_settings)
1292
1293            ax1 = plt.subplot(211)
1294            if hasattr(offsetbox, 'AnnotationBbox'):
1295                # only print thumbnails with matplotlib > 1.0
1296                for sw in sweep:
1297                    i = my_functions.arg_closest(sw, X)
1298    #                d = self.tsDB.get_run(i)
1299                    y = self.df.xs('outputs',axis=1).ix[i,:].values
1300                    img = my_functions.make_ts_img(y, ts_resample,
1301                    dot_size, minimum, maximum)
1302                    imagebox = offsetbox.AnnotationBbox(
1303                        offsetbox.OffsetImage(img,
1304                        cmap=plt.cm.gray_r, zoom=1),
1305                        (X[i], 0))
1306                    ax1.add_artist(imagebox)
1307            plt.ylim([-1,1])
1308            plt.xlim([-.1,1.1])
1309            plt.xticks([])
1310            plt.yticks([])
1311            ax2.set_xticks(np.arange(0,1.1,.2))
1312            plt.setp(ax2.get_xticklabels(),**font_settings)
1313
1314            ax2.spines['top'].set_visible(False)
1315            ax1.spines['bottom'].set_visible(False)
1316            fig.subplots_adjust(hspace=0)
1317            return fig
1318
1319        def plot_pca_vs_nseg(self, fig=None, font_settings=None):
1320            x = self.df.ix[:, ('attr','pca')].values
1321            y = self.df.ix[:, ('attr','nseg')].values
1322            g = self.df.ix[:, ('attr','group')].values
1323            groups = list(set(g))
1324    #        x = self.tsDB.get_val_all('db_attr', 'pca')
1325    #        y = self.tsDB.get_nsegs()
1326    #        g = self.tsDB.get_val_all('db_attr', 'group')
1327    #        x = x[:, 1]
1328    #        y = y.xs('nseg', axis=1).values
1329    #        g = g[:, 1]
1330    #        groups = self.tsDB.groups
1331            if fig is None:
1332                fig = plt.figure()
1333            if font_settings is None:
1334                font_settings = {'fontsize':14, 'fontname':'sans'}
1335            ax = plt.subplot(111)
1336            for gr in groups:
1337                ind = np.argwhere(g == gr)
1338                if gr == -1:
```

```
1339                 ax.plot(x[ind], y[ind], 'ok')
1340             else:
1341                 marker = '$' + str(int(gr)) + '$'
1342                 ax.plot(x[ind], y[ind], '.', marker=marker, ms=10.)
1343         plt.xlim([-.1, 1.1])
1344 #        plt.ylim([np.min(groups)-.5,np.max(groups)+.5])
1345         plt.ylim([0, np.max(y) + .5])
1346         plt.xlabel('PC 1', **font_settings)
1347         plt.ylabel('# of segments', **font_settings)
1348         plt.setp(ax.get_xticklabels(),**font_settings)
1349         plt.setp(ax.get_yticklabels(),**font_settings)
1350 
1351         return fig
1352 
1353     def plot_pca_vs_brkpca(self, fig=None, font_settings=None):
1354 
1355         groups = list(set(self.df.ix[:, ('attr', 'group')].values))
1356 
1357         # Plot Figure
1358         if fig is None:
1359             fig = plt.figure()
1360         if font_settings is None:
1361             font_settings = {'fontsize':14, 'fontname':'sans'}
1362         ax = plt.subplot(111)
1363         for gr in groups:
1364             df_subset = self.df[self.df.ix[:, ('attr',
1365                                                'group')] == gr]
1366             pca = df_subset.ix[:, ('attr', 'pca')].values
1367             brk_pca = df_subset.ix[:,
1368                     ('ApplyPCA_Breakpoints','pca0')].values
1369             if gr == -1:
1370                 ax.plot(pca, brk_pca, 'ok')
1371             else:
1372                 marker = '$' + str(int(gr)) + '$'
1373                 ax.plot(pca, brk_pca, '.', marker=marker, ms=10.)
1374         plt.xlim([-.1, 1.1])
1375 #        plt.ylim([np.min(groups)-.5,np.max(groups)+.5])
1376         y_min = self.df.ix[:, ('ApplyPCA_Breakpoints',
1377                                'pca0')].min() - .2
1378         y_max = self.df.ix[:,
1379                     ('ApplyPCA_Breakpoints','pca0')].max() + .2
1380         plt.ylim([y_min, y_max])
1381         plt.xlabel('PC 1', **font_settings)
1382         plt.ylabel('PCA(Breakpoints)', **font_settings)
1383         plt.setp(ax.get_xticklabels(),**font_settings)
1384         plt.setp(ax.get_yticklabels(),**font_settings)
1385 
1386         return fig
1387 
1388     def plot_y_by_group(self, fig=None):
1389         ''' This plot function plots the runs in the
1390             database and each plot is grouped by the grouping
1391         '''
1392         if fig is None:
```

```
1393            fig = plt.figure()
1394
1395        # Set up Data
1396        ys = self.tsDB.get_y()
1397        grps = self.tsDB.get_val_all('db_attr', 'group',
1398                                     out_style='data frame')
1399        grps.columns = pd.MultiIndex(levels=[['run_id','group'],
1400                                             ['run_id','group']],
1401                                     labels=[[0, 1],[0,1]])
1402        df = pd.merge(ys, grps)
1403        groups = self.tsDB.groups
1404
1405        gs = gridspec.GridSpec(nrows=len(groups), ncols=1)
1406
1407        for i_row, gr in enurate(groups):
1408            df_test = df[ df.xs(('group','group'),axis=1) == gr]
1409            ys_test = df_test.xs('y',axis=1)
1410            segs_test = df_test.xs('segs',axis=1)
1411            ax = plt.subplot(gs[i_row])
1412            for y in ys_test.iterrows():
1413                ax.plot(y[1].values)
1414        return fig
1415
1416    def plot_y_by_group_and_seg(self, fig=None):
1417        ''' Plots the y in the database.
1418            Produces a n x m matrix of plots, where n is the
1419            number of groups and m is the maximum number of
1420            segments + 1.
1421            Plots the whole y in the first column.
1422            Plots each of the segments in the rest of the columns.
1423        '''
1424        if fig is None:
1425            fig = plt.figure(figsize=(20,20))
1426
1427        # Prepare the data
1428        # Grab the Y
1429        ys = self.tsDB.get_y()
1430        # Grab Groups
1431        grps = self.tsDB.get_val_all('db_attr', 'group',
1432                                     out_style='data frame')
1433        grps.columns = pandas.MultiIndex(levels=[['run_id','group'],
1434                                             ['run_id','group']],
1435                                             labels=[[0, 1],[0,1]])
1436        # Grab Segmentation
1437        segs = self.tsDB.get_val_all('db_attr', 'segs',
1438                                     out_style='list')
1439        segs_ = [ [k, str(v)] for k, v in segs ]
1440        segs_ = np.array(segs_)
1441        run_ids_ = np.array(segs_[:,0],dtype=int)
1442        segs_ = segs_[:,1]
1443        d_segs = {('run_id','run_id'):run_ids_,
1444                  ('segs','segs'):segs_}
1445        df_segs = pandas.DataFrame.from_dict(d_segs)
1446        # Merge Data
```

```
1447          df = pandas.merge(ys, grps)
1448          df = pandas.merge(df, df_segs)
1449          # Determine max number of segments
1450          df_nseg = self.tsDB.get_val_all('db_attr',
1451                      'nseg',out_style='data frame')
1452          max_nseg = np.max(df_nseg.ix[:,1].values)
1453          groups = self.tsDB.groups
1454          # Option to remove the outlier points
1455          #if -1 in groups: groups.remove(-1)
1456          # Specify the number of rows and columns in graph
1457          nr = len(groups)
1458          nc = max_nseg + 1
1459          # Define gridspec
1460          gs = gridspec.GridSpec(nrows=nr, ncols=nc)
1461
1462          my_iter=0
1463
1464          # Iterate over the groups
1465          for i_row, gr in enumerate(groups):
1466              # Grab subset of data for each group
1467              df_test = df[ df.xs(('group','group'),axis=1) == gr]
1468              ys_test = df_test.xs('y',axis=1)
1469              segs_test = df_test.xs('segs',axis=1)
1470              seg_ = eval(segs_test.get_value(segs_test.index[0],
1471                          'segs'))
1472              ax = plt.subplot(gs[i_row,0])
1473              axs = [ plt.subplot(gs[i_row,
1474                                     c+1]) for c in range(len(seg_)) ]
1475              # Iterate over each row in group
1476              for y, seg in izip(ys_test.iterrows(),
1477                                 segs_test.iterrows()):
1478                  # First Column of plots is all the segments
1479                  ax.plot(y[1].values)
1480                  # Iterate over the segments
1481                  seg_ = eval(seg[1].values[0])
1482                  t = np.arange(len(y[1].values))
1483                  for i_col, (st,en) in enumerate(seg_):
1484                      ax_ = axs[i_col]
1485                      ax_.plot(t[st:en],y[1][st:en])
1486  #                if my_iter > 3:
1487  #                    break
1488  #                else:
1489  #                    my_iter += 1
1490  #            break
1491
1492          # update axes
1493          all_axes = fig.axes
1494          y_min = np.min(df.xs('y',axis=1).values)
1495          y_max = np.max(df.xs('y',axis=1).values)
1496          y_range = y_max - y_min
1497          y_min -= y_range*.1
1498          y_max += y_range*.1
1499          x_min = 0
1500          x_max = len(y[1].values)+1
```

```
1501
1502            print df
1503            print 'y_min, y_max:', y_min, y_max
1504            print 'x_min, x_max:', x_min, x_max
1505
1506            for ax in all_axes:
1507                ax.set_ylim(y_min, y_max)
1508                ax.set_xlim(x_min, x_max)
1509            plt.tight_layout()
1510            return fig
1511
1512        def garbage_collect(self):
1513            gc.collect()
1514            R = robjects.r
1515            R.gc()
1516    #        R('gc()')
1517            gc.collect()
1518
1519    class PrepOutputData(object):
1520        def __init__(self, df, n_bins=500,
1521                     bin_threshold=20, resample=None):
1522            self.df = df
1523            self.n_bins = n_bins
1524            self.bin_threshold = bin_threshold
1525            if isinstance(resample, int):
1526                self.df = df.ix[:,::resample]
1527
1528        @classmethod
1529        def from_file_paths(cls, output_files,
1530                            n_bins=1000, resample=None):
1531            '''
1532            Create instance using csv file paths
1533
1534            :param output_files: list of output file locations
1535            '''
1536            # instantiating as a dataframe is faster than sqlite3
1537            df_out = pd.DataFrame([])
1538            for file_path in output_files:
1539                df_part = pd.DataFrame.from_csv(file_path)
1540                df_part = df_part.ix[np.argsort(df_part.index), :]
1541                df_out = pd.concat((df_out, df_part), axis=0)
1542            if isinstance(resample, int):
1543                df_out = df_out.ix[:,::resample]
1544            return cls(df=df_out, n_bins=n_bins)
1545
1546
1547        def perform_pca(self, n_components=5):
1548            '''
1549            Perform pca on data (df).
1550
1551            :param n_components: number of PCs to return
1552            :returns: data frame with principal components
1553            '''
1554            if n_components == 'auto':
```

```
            print '[[[ Calculating n_components ]]]'
            temp_fn = FeatureExtract.determine_n_components
            n_components = temp_fn(dat=self.df.values,
                                   reps=1000)
            print '[[[ n_components = ', n_components, ']]]'
        return self._perform_pca(self.df, n_components)

    def _perform_pca(self, df, n_components=5):

        fit_pca = PCA(n_components=n_components)
        X_pca = fit_pca.fit_transform(df.values)
        X_pca = my_functions.normalize(X_pca)

        nr, nc = X_pca.shape
        pca_col_names = [''.join(('pca',str(i))) for i in range(nc)]
        df_pca = pd.DataFrame(X_pca, index=df.index,
                              columns=pca_col_names)
        return df_pca, n_components

    def reduce_data(self, n_components=5, how='median'):
        df_pca, n_components = self.perform_pca(n_components)

        df_pca_r = df_pca.apply(np.round, decimals=3)
        df_pca_rs = df_pca_r.sort(list(df_pca.columns))

        idx_pca_rs_split = self.make_bin_index2(df_pca, df_pca_r)

        nr, nc = self.df.shape
        n_bins = len(idx_pca_rs_split)
        mat_collapsed = np.zeros((n_bins, nc))
        mat_ = np.zeros((1, nc))
        sigmas = np.zeros(n_bins)
        pcas = np.zeros((n_bins, n_components))
        for i, idxs in enumerate(idx_pca_rs_split):
            df_temp = self.df.ix[idxs, :]
            if how == 'median':
                mat_collapsed[i,:] = df_temp.median(axis=0)
                mat_[0,:] = df_temp.mean(axis=0)
                sigmas[i] = np.std(df_temp.values - mat_[0,:])
            elif how == 'mean':
                mat_collapsed[i,:] = df_temp.mean(axis=0)
                mat_[0,:] = df_temp.mean(axis=0)
                sigmas[i] = np.std(df_temp.values - mat_[0,:])
            elif how == 'min':
                idx = copy.copy(idxs)
                if i > 1:
                    idx.extend(idx_pca_rs_split[i-1])
                if i < len(idx_pca_rs_split)-1:
                    idx.extend(idx_pca_rs_split[i+1])
                if i > 2:
                    idx.extend(idx_pca_rs_split[i-2])
                if i < len(idx_pca_rs_split)-2:
                    idx.extend(idx_pca_rs_split[i+2])
                df_temp = self.df.ix[idx,:]
```

```
                temp_min = df_temp.min(axis=0)
                temp_t = np.arange(len(temp_min))
                min_upper = self.find_upper_envelope_v2(temp_t,
                                                        temp_min, 20)
                min_lower = self.find_lower_envelope_v2(temp_t,
                                                        temp_min, 20)
                min_middle = (min_upper + min_lower)/2
                mat_collapsed[i,:] = min_middle
                sigmas[i] = np.std(temp_min - min_middle)
            elif how == 'max':
                idx = copy.copy(idxs)
                if i > 1:
                    idx.extend(idx_pca_rs_split[i-1])
                if i < len(idx_pca_rs_split)-1:
                    idx.extend(idx_pca_rs_split[i+1])
                if i > 2:
                    idx.extend(idx_pca_rs_split[i-2])
                if i < len(idx_pca_rs_split)-2:
                    idx.extend(idx_pca_rs_split[i+2])
                df_temp = self.df.ix[idx,:]

                temp_max = df_temp.max(axis=0)
                temp_t = np.arange(len(temp_max))
                max_upper = self.find_upper_envelope_v2(temp_t,
                                                        temp_max, 20)
                max_lower = self.find_lower_envelope_v2(temp_t,
                                                        temp_max, 20)
                max_middle = (max_upper + max_lower)/2
                mat_collapsed[i,:] = max_middle
                sigmas[i] = np.std(temp_max - max_middle)
            pcas[i,:] = df_pca.ix[idxs,:].mean(axis=0)

        mat_collapsed = np.vstack((np.repeat(mat_collapsed[:1,:],
                                   10, axis=0), mat_collapsed))
        mat_collapsed = np.vstack((mat_collapsed,
                                   np.repeat(mat_collapsed[-1:,:],
                                             10,axis=0)))
        pcas = np.vstack((np.repeat(pcas[:1,:], 10, axis=0), pcas))
        pcas = np.vstack((pcas, np.repeat(pcas[-1:,:], 10, axis=0)))
        pcas[:10,0] += np.arange(-.1, 0, .01)
        pcas[-10:,0] += np.arange(.01, 0.11, .01)
        sigmas = np.hstack((np.repeat(sigmas[0],10),
                            sigmas, np.repeat(sigmas[-1], 10)))

        headers = [''.join(('pca',
                            str(i) )) for i in range(n_components)]
        df_pca_reduced = pd.DataFrame(pcas, columns=headers)
        d_attr = {'pca':pcas[:,0], 'sigma':sigmas}

        self.df_pcas_all = df_pca
        self.df_pca_reduced = df_pca_reduced
        self.df_outs = pd.DataFrame(mat_collapsed,
                                    columns=self.df.columns)
```

```
1663            self.df_attr = pd.DataFrame.from_dict(d_attr)
1664
1665        def make_bin_index1(self, df_pca_rs):
1666            idx_pca_rs = df_pca_rs.index
1667            idx_pca_rs_split = np.array_split(idx_pca_rs, self.n_bins)
1668
1669        def make_bin_index2(self, df_pca, df_pca_r):
1670            pca0_range = df_pca_r.min()[0], df_pca_r.max()[0]
1671            pca0_bins = np.linspace(pca0_range[0],
1672                                    pca0_range[1], self.n_bins)
1673            idx_pca_rs_split = []
1674            pca0_vals = df_pca.xs('pca0', axis=1).values
1675            i = 0
1676            while i < len(pca0_bins)-1:
1677                idx = []
1678                i2 = i
1679                while len(idx) < self.bin_threshold:
1680                    i2 += 1
1681                    if i2 > len(pca0_bins):
1682                        break
1683                    mi = pca0_bins[i]
1684                    ma = pca0_bins[i2]
1685                    idx_gt = np.argwhere(pca0_vals >= mi).ravel()
1686                    idx_lt = np.argwhere(pca0_vals < ma).ravel()
1687                    idx = list(set(idx_gt).intersection(idx_lt))
1688                idx_pca_rs_split.append(idx)
1689                i = i2
1690            return idx_pca_rs_split
1691
1692        def find_upper_envelope_v2(self, t, y, n_split=10):
1693            t_split = np.array_split(t, n_split)
1694            y_split = np.array_split(y, n_split)
1695            t_new = [0]
1696            y_new = [y[0]]
1697            for i1, i2 in zip(t_split,y_split):
1698                i3 = np.argmax(i2)
1699                t_new.append(i1[i3])
1700                y_new.append(i2[i3])
1701            t_new.append(t[-1])
1702            y_new.append(y[-1])
1703            f_upper = interp1d(t_new, y_new)
1704            y_new2 = f_upper(t)
1705            idx = np.isnan(y_new2)
1706            y_new2[idx] = y[idx]
1707            return y_new2
1708
1709        def find_lower_envelope_v2(self, t, y, n_split=10):
1710            t_split = np.array_split(t, n_split)
1711            y_split = np.array_split(y, n_split)
1712            t_new = [0]
1713            y_new = [y[0]]
1714            for i1, i2 in zip(t_split,y_split):
1715                i3 = np.argmin(i2)
1716                t_new.append(i1[i3])
```

```
            y_new.append(i2[i3])
        t_new.append(t[-1])
        y_new.append(y[-1])
        f_lower = interp1d(t_new, y_new)
        y_new2 = f_lower(t)
        idx = np.isnan(y_new2)
        y_new2[idx] = y[idx]
        return y_new2


if __name__=='__main__':
    print('hello')
```

A.2 ts_segments.py

```
import numpy as np
import matplotlib.pyplot as plt
import TimeSeriesDB
import pandas
import scipy.optimize

import smoothers
from scipy.odr.odrpack import _conv

class tsBase(object):
    name = 'base'
    def __init__(self,t,mu,sd):
        self.t = np.array(t)
        self.size = np.size(t)
        self.y = np.zeros(len(t))
        self.mu = mu
        self.sd = sd
    def noise(self):
        if self.sd == 0:
            return np.zeros(self.size)
        else:
            return np.random.normal(0,self.sd,self.size)
    def plot(self,fig=None):
        if fig is None:
            fig=plt.figure()
        ax = plt.subplot(111)
        ax.plot(self.t,self.y)
    def _find_bkpts(self,y):
        ''' returns segment start and end as pairs given a
            list of disjoint groupings of integers '''
        breakpoints = []
        v1 = y[0]
        b0 = 0
        for i,x in enumerate(y):
            if x != v1:
                b1 = i-1
                if b0 == b1:
                    continue
                breakpoints.append([b0,b1])
                b0 = i
```

```
                v1 = x
        breakpoints.append([b0,i])
        self.breakpoints = breakpoints

class tsFlatLine(tsBase):
    name = 'flat'
    def __init__(self,t,mu=0,sd=0):
        super(tsFlatLine,self).__init__(t,mu,sd)
        self.y = self._make_flat_y()
        self.make_y = self._make_flat_y
        self.type = "flat"
        self.breakpoints = [[0,len(self.t)-1]]
    def _make_flat_y(self):
        if self.sd == 0:
            y = np.repeat(self.mu,self.size)
        else:
            y = np.random.normal(self.mu,self.sd,self.size)
        return y

class tsSineWave(tsBase):
    name = 'sine'
    def __init__(self,t,mu=0.,sd=0.,
                 amplitude=1.,n_waves=1.,phase=0.):
        super(tsSineWave,self).__init__(t,mu,sd)
        self.A=amplitude
        self.f = float(n_waves)/(t[-1]-t[0])
        self.phi = phase
        self.y = self._make_wavy_y()
        self.type = 'sine'
    def _make_wavy_y(self):
        noise = self.noise()
        s = 2*np.pi*(self.f*self.t + self.phi)
        y = self.A * np.sin(s) + noise + self.mu
        s2 = np.zeros((len(s)))
        ss = np.abs(np.sin(s))
        sc = np.cos(s)
        for x in [30,60]:
            s2 += (ss>=np.sin(np.pi*x/180.))*np.sign(sc)
        self._find_bkpts(s2)
        return y

class tsSquareWave(tsBase):
    ''' Creates a Square Wave. Ratio controls the length
        of the high and low and ranges from -1 to 1.
        ratio of -1 would mean all low and 1
        would be all high. It's based on sine wave so the
        ratio is nonlinear.
    '''
    name = 'square'
    def __init__(self,t,mu=0.,sd=0.,amplitude=1.,
                 n_waves=1.,phase=0.,ratio=0.):
        super(tsSquareWave,self).__init__(t,mu,sd)
        self.A = amplitude
        self.f = float(n_waves)/(t[-1]-t[0])
```

```
95          self.r = -ratio
96          self.phi = phase
97          self.y = self._make_square_y()
98          self.type = 'square'
99      def _make_square_y(self):
100         noise = self.noise()
101         s = np.sin(2*np.pi*(self.f*self.t + self.phi))
102         y = ((s>=self.r) * 1 + (s<self.r)*-1)
103         self._find_bkpts(y)
104         y = self.A * y + noise + self.mu
105         return y
106
107 class tsTriangleWave(tsBase):
108     name = 'triangle'
109     def __init__(self, t, mu=0., sd=0.,
110                  amplitude=1., n_waves=1., phase=0.):
111         super(tsTriangleWave,self).__init__(t,mu,sd)
112         self.A = amplitude
113         self.a = 0.5*(t[-1]-t[0])/float(n_waves)
114         self.phi = phase
115         self.y = self._make_triangle_y()
116         self.type = 'triangle'
117     def _make_triangle_y(self):
118         noise = self.noise()
119         a = self.a
120         t = self.t + (self.phi+.5)*self.a*2
121         y = self.A*(2./a)*(t-a*np.floor(t/a+0.5))
122         y *= (-1)**np.floor(t/a-0.5)
123         y += noise + self.mu
124         self._find_bkpts((-1)**np.floor(t/a-0.5))
125         return y
126
127 class tsSawtooth(tsBase):
128     name = 'sawtooth'
129     def __init__(self, t, mu=0., sd=0.,
130                  amplitude=1., n_waves=1., phase=0.):
131         super(tsSawtooth,self).__init__(t,mu,sd)
132         self.A = amplitude
133         self.a = 0.5*(t[-1]-t[0])/float(n_waves)
134         self.phi = phase
135         self.y = self._make_triangle_y()
136         self.type = 'triangle'
137     def _make_triangle_y(self):
138         noise = self.noise()
139         a = self.a
140         t = self.t + (self.phi+.5)*self.a*2
141         y = self.A*2*(t/a-np.floor(t/a+0.5)) + noise + self.mu
142         self._find_bkpts(np.floor(t/a+0.5))
143         return y
144
145 class tsStep(tsBase):
146     name = 'step'
147     def __init__(self,t,mu=0.,sd=0.,separation=1.,loc=0.5):
148         super(tsStep,self).__init__(t,mu,sd)
```

```
149         self.sep = separation
150         self.loc = int(len(t)*loc)
151         self.y = self._make_step_y()
152         self.breakpoints = [[ 0, self.loc ],
153                             [ self.loc+1, len(self.t)-1 ]]
154         self.type = 'step'
155     def _make_step_y(self):
156         noise = self.noise()
157         ind = np.arange(self.loc,len(self.t))
158         y = np.zeros(len(self.t)) + noise
159         y[ind] = y[ind]+self.sep + self.mu
160         return y
161
162 class tsRandomWalk(tsBase):
163     ''' Generates a random walk of length t. The default is
164         Unif(-0.1,0.1)
165     '''
166     name = 'random walk'
167     def __init__(self, t, rand_fn=np.random.uniform,
168                  rand_args=[-.1,.1]):
169         super(tsRandomWalk,self).__init__(t,0,0)
170         self.rand_vec = rand_fn(*rand_args,size=len(t))
171         self.y = self._make_random_walk()
172         self.breakpoints = [ 0, self.t[-1] ]
173         self.type = 'random walk'
174     def _make_random_walk(self):
175         y = []
176         c = 0.
177         for i in self.rand_vec:
178             c += i
179             y.append(c)
180         return y
181
182 class tsLinearLine(tsFlatLine):
183     name = 'sloped'
184     def __init__(self, t, mu=np.nan, sd=0,
185                  slope=np.nan, ystart=np.nan, yend=np.nan):
186         if np.isnan(mu):
187             mu = 0
188             mu_was_nan = True
189         else:
190             mu_was_nan = False
191         super(tsLinearLine,self).__init__(t,mu,sd)
192         self.ystart = np.float(ystart)
193         self.yend = np.float(yend)
194         if np.all(np.isnan([slope,ystart,yend])):
195             ystart = 0
196             yend = 1
197         if np.isnan(slope):
198             self._calc_slope()
199             if self.slope == 0.:
200                 if mu_was_nan:
201                     self.mu = ystart
202         else:
```

```
203             self.slope = slope
204             if np.isnan(ystart):
205                 self.ystart = mu
206                 self.yend = self.slope*self.t[-1]+self.ystart
207         self.y = self._make_sloped_y()
208         self.make_y = self._make_sloped_y
209         self.breakpoints = [[ 0, len(self.t)-1 ]]
210         self.type = "sloped"
211     def _make_sloped_y(self):
212         if self.slope == 0:
213             y = super(tsLinearLine, self)._make_flat_y()
214         else:
215             y = self.slope*self.t + self.ystart
216             y = y + super(tsLinearLine,self)._make_flat_y()
217         return y
218     def _calc_slope(self):
219         if np.isnan(self.ystart):
220             self.slope = 0
221             self.ystart = self.mu
222             self.yend = self.mu
223         else:
224             self.slope = (self.yend - self.ystart)/(self.t[-1] - \
225                           self.t[0])
226
227 class tsLine( tsLinearLine ):
228     def __init__( self, t, mu=0, sd=0, slope=np.nan,
229                   ystart=np.nan, yend=np.nan ):
230         if np.isnan(slope) and np.isnan(ystart) and np.isnan(yend):
231             # This is a flat line so initialize using parent of
232             # tsLinearLine which is tsFlatLine
233             super(tsLinearLine,self).__init__(t=t,mu=mu,sd=sd)
234         else:
235             # This is a sloped line so initialize with parent of
236             # tsLine which is tsLinearLine
237             super(tsLine,self).__init__(t=t,mu=mu,sd=sd,slope=slope,
238                                         ystart=ystart,yend=yend)
239     def add_right(self,seg):
240         new_t = seg.t + self.t[-1] + 1
241         self.t = np.append( self.t, new_t )
242         self.y = np.append( self.y, seg.y )
243         self.breakpoints = np.append( self.breakpoints,
244                                       seg.breakpoints+new_t[0] )
245     def merge_with_linear_connector( self, seg,
246                                      transition_x=np.arange(5),
247                                      sd=None ):
248         if sd is None:
249             sd = self.sd
250         if self.type=="flat":
251             start_point = self.y[-1]
252         elif self.type=="sloped":
253             start_point = self.yend
254         if seg.type=="flat":
255             end_point = seg.y[0]
256         elif seg.type=="sloped":
```

```
257             end_point = seg.ystart
258         seg_connect = tsLinearLine(t=transition_x,mu=0,sd=sd,
259                                    ystart=start_point,
260                                    yend=end_point)
261         self.add_right(seg_connect)
262         self.add_right(seg)
263
264 class make_ts(object):
265     def __init__(self,y_val,mc=1):
266         self.mc = mc
267         self.x_len = 350
268         self.segs = []
269         self.brkpts = []
270         levels = [float(x) for x in range(1,8)]
271         if y_val < levels[0]:
272             fn = self.type1
273         elif y_val < levels[1]:
274             self.val = (y_val-levels[0])/(levels[1]-levels[0])
275             fn=self.type2
276         elif y_val < levels[2]:
277             self.val = (y_val-levels[1])/(levels[2]-levels[1])
278             fn = self.type3
279         elif y_val < levels[3]:
280             self.val = (y_val-levels[2])/(levels[3]-levels[2])
281             fn = self.type4
282         elif y_val < levels[4]:
283             self.val = (y_val-levels[3])/(levels[4]-levels[3])
284             fn = self.type5
285         elif y_val < levels[5]:
286             self.val = (y_val-levels[4])/(levels[5]-levels[4])
287             fn = self.type6
288         else:
289             fn =self.type7
290         self.run_mc(fn)
291     def plot(self):
292         for seg in self.segs:
293             plt.plot(seg.t,seg.y)
294         plt.ylim((0,1))
295         plt.show()
296     def run_mc(self,fn):
297         for i in range(self.mc):
298             seg,brkpts = fn()
299             self.segs.append(seg)
300             self.brkpts.append(brkpts)
301     def type1(self):
302         val1 = round(np.random.normal(50.0,1.0))
303         val3 = round(np.random.normal(self.x_len-60,1))
304         x1 = np.arange(val1)
305         x2 = np.arange(self.x_len-val1-val3)
306         x3 = np.arange(val3)
307         seg1 = tsLine(t=x1,mu=.9,sd=.01)
308         seg3 = tsLine(t=x3,mu=.1,sd=.01)
309         seg1.merge_with_linear_connector(seg3, transition_x=x2,
310                                          sd=.01)
```

```
311         self.seg = seg1
312         self.brkpt = [[0,len(x1)-1],[len(x1),len(x1)+len(x2)-1],
313                       [self.x_len-len(x3),self.x_len-1]]
314         self.group = 2
315         return seg1, self.brkpt
316     def type2(self):
317         x_target = 90*self.val + 60
318         val1 = round(np.random.normal(50,1))
319         val3 = round(np.random.normal(self.x_len-x_target,1))
320         x1 = np.arange(val1)
321         x2 = np.arange(self.x_len-val1-val3)
322         x3 = np.arange(val3)
323         seg1 = tsLine(t=x1,mu=.9,sd=.01)
324         seg3 = tsLine(t=x3,mu=.1,sd=.01)
325         seg1.merge_with_linear_connector(seg3, transition_x=x2,
326                                          sd=.01)
327         self.seg=seg1
328         self.brkpt = [[0,len(x1)-1],[len(x1),len(x1)+len(x2)-1],
329                       [self.x_len-len(x3),self.x_len-1]]
330         self.group = 2
331         return seg1, self.brkpt
332     def type3(self):
333         val1 = round(np.random.normal(50,1))
334         val2 = round(np.random.normal(100,1))
335         val4 = round(np.random.normal(self.x_len-160,1))
336         x1 = np.arange(val1)
337         x2 = np.arange(val2)
338         x3 = np.arange(self.x_len-val1-val2-val4)
339         x4 = np.arange(val4)
340         y_target=(0.9-0.1)*self.val+0.1
341         seg1 = tsLine(t=x1,mu=.9,sd=.01)
342         seg2 = tsLine(t=x2,sd=.01,ystart=0.9,yend=y_target)
343         seg4 = tsLine(t=x4,mu=.1,sd=0.01)
344         seg1.add_right(seg2)
345         seg1.merge_with_linear_connector(seg4,
346                                          transition_x=x3,
347                                          sd=.01)
348         self.seg=seg1
349         self.brkpt = [[0,len(x1)-1],[len(x1),len(x1)+len(x2)-1],
350                       [len(x1)+len(x2),len(x1)+len(x2)+len(x3)-1],
351                       [self.x_len-len(x4),self.x_len-1]]
352         self.group = 3
353         return seg1, self.brkpt
354     def type4(self):
355         val1 = round(np.random.normal(150,1))
356         val2 = round(np.random.normal(10,1))
357         val3 = round(np.random.normal(80,1))
358         x1 = np.arange(val1)
359         x2 = np.arange(val2)
360         x3 = np.arange(val3)
361         x4 = np.arange(self.x_len-val1-val2-val3)
362         y_target = (0.9-0.1)*self.val
363         y_target += 0.1
364         y_target += np.random.normal(0,.3)*(1-self.val)*(self.val)
```

```
365         if y_target > .9:
366             y_target = .9
367         if y_target < .1:
368             y_target = .1
369         y_target -= .1
370         seg1 = tsLine(t=x1,mu=.9,sd=.01)
371         seg3 = tsLine(t=x3,mu=.1,sd=.01)
372         seg4 = tsLine(t=x4,mu=0.1,sd=.01,ystart=0,yend=y_target)
373         seg1.merge_with_linear_connector(seg3,
374                                          transition_x=x2,
375                                          sd=.01)
376         seg1.add_right(seg4)
377         self.seg=seg1
378         self.brkpt = [[0,len(x1)-1],[len(x1),len(x1)+len(x2)-1],
379                       [len(x1)+len(x2),len(x1)+len(x2)+len(x3)-1],
380                       [self.x_len-len(x4),self.x_len-1]]
381         self.group = 4
382         return seg1, self.brkpt
383     def type5(self):
384         x_target = 100*self.val
385         val1 = round(np.random.normal(150,1))
386         val2 = round(np.random.normal(10,1))
387         val3 = round(np.random.normal(80,1))
388         val5 = round(np.random.normal(x_target,2))
389         if val5 < 1:
390             val5 = 1
391         x1 = np.arange(val1)
392         x2 = np.arange(val2)
393         x3 = np.arange(val3)
394         x4 = np.arange(self.x_len-val1-val2-val3-val5)
395         x5 = np.arange(val5)
396         seg1 = tsLine(t=x1,mu=.9,sd=.01)
397         seg3 = tsLine(t=x3,mu=.1,sd=.01)
398         seg5 = tsLine(t=x5,mu=.9,sd=.01)
399         seg1.merge_with_linear_connector(seg3, transition_x=x2,
400                                          sd=.01)
401         seg1.merge_with_linear_connector(seg5, transition_x=x4,
402                                          sd=.01)
403         self.seg=seg1
404         self.brkpt = [[0,len(x1)-1],[len(x1),len(x1)+len(x2)-1],
405                       [len(x1)+len(x2),len(x1)+len(x2)+len(x3)-1],
406                       [len(x1)+len(x2)+len(x3),
407                        len(x1)+len(x2)+len(x3)+len(x4)-1],
408                       [self.x_len-len(x5),self.x_len-1]]
409         self.group = 5
410         return seg1, self.brkpt
411     def type6(self):
412         val1 = round(np.random.normal(150,1))
413         val2 = round(np.random.normal(10,1))
414         val3 = round(np.random.normal(80,1))
415         val4 = round(np.random.normal(10,1))
416         x1 = np.arange(val1)
417         x2 = np.arange(val2)
418         x3 = np.arange(val3)
```

```
419         x4 = np.arange(val4)
420         x5 = np.arange(self.x_len-val1-val2-val3-val4)
421         y_target = (0.9-.1)*self.val
422         y_target += .1
423         y_target += np.random.normal(0,.2)*(1-self.val)*(self.val)
424         if y_target > 0.9:
425             y_target=.9
426         if y_target < .1:
427             y_target = .1
428         seg1 = tsLine(t=x1,mu=.9,sd=.01)
429         seg3 = tsLine(t=x3,mu=y_target,sd=.01)
430         seg5 = tsLine(t=x5,mu=.9,sd=.01)
431         seg1.merge_with_linear_connector(seg3, transition_x=x2,
432                                          sd=.01)
433         seg1.merge_with_linear_connector(seg5, transition_x=x4,
434                                          sd=.01)
435         self.seg=seg1
436         self.brkpt = [[0,len(x1)-1],[len(x1),len(x1)+len(x2)-1],
437                       [len(x1)+len(x2),len(x1)+len(x2)+len(x3)-1],
438                       [len(x1)+len(x2)+len(x3),
439                        len(x1)+len(x2)+len(x3)+len(x4)-1],
440                       [self.x_len-len(x5),self.x_len-1]]
441         self.group = 6
442         return seg1, self.brkpt
443     def type7(self):
444         x1 = np.arange(self.x_len)
445         seg1 = tsLine(t=x1,mu=.9,sd=.01)
446         self.seg=seg1
447         self.brkpt = [[0,self.x_len-1]]
448         self.group = 7
449         return seg1,self.brkpt
450
451 #def bottom_up_aggr(t,y,sigma=1.0):
452 def bottom_up_aggr(t,y,*args,**kwargs):
453
454     t = np.asarray(t)
455     y = np.asarray(y)
456
457     if len(args) == 1:
458         myCutoff_sd = args[0]
459         myOption = 'sigma'
460     else:
461         if 'sigma' in kwargs:
462                 myCutoff_sd = kwargs['sigma']
463                 myOption = 'sigma'
464         elif 'nseg' in kwargs:
465                 myNumSegs = kwargs['nseg']
466                 myCutoff = int(np.ceil(len(y)/2.) - myNumSegs)
467                 myOption = 'nseg'
468                 if myNumSegs == 1:
469                     return [t],[y]
470
471     def calc_stddev_of_resid(T,Y):
472         T = np.array(T)
```

```
473         Y = np.array(Y)
474         A = np.vstack((T,np.ones(len(T))))
475         fit = np.linalg.lstsq(A.T,Y)
476         resid = np.dot(fit[0],A)-Y
477         return np.std(resid)
478 
479     if myOption == 'sigma':
480         y3sd = calc_stddev_of_resid(t, y)
481         if y3sd < myCutoff_sd:
482             return [t],[y]
483 
484     t2 = list()
485     y2 = list()
486 
487     L1 = len(y)
488     L2 = int(np.floor(L1/2))
489 
490     ### Pair Up Points ###
491     for i in range(L2):
492         t2.append(t[2*i:2*i+2])
493         y2.append(y[2*i:2*i+2])
494     # Check to see if sequence is odd
495     if np.mod(L1,2) == 1:
496         t2.append([t[-1]])
497         y2.append([y[-1]])
498 
499     ### Calculate Linear Regression of adj. sets
500     ### Calculate std. dev.
501     y3sd = list()
502     for i in range(len(y2)-1):
503         t3 = []
504         t3.extend(t2[i])
505         t3.extend(t2[i+1])
506         y3 = []
507         y3.extend(y2[i])
508         y3.extend(y2[i+1])
509         y3sd.append(calc_stddev_of_resid(t3, y3))
510 #    print y3sd
511     myMins = [0]
512     myDelta = []
513     myX = []
514 
515     if myOption == 'sigma':
516         i2 = -1
517         while min(y3sd) < myCutoff_sd:
518             i2 += 1
519             #print i2
520             myMins.append(np.min(y3sd))
521             myDelta.append(myMins[i2+1]-myMins[i2])
522             myX.append(myMins[i2+1]/np.mean(myMins[:(i2+2)]))
523             myIndex = y3sd.index(myMins[-1])
524 
525 #             print i2,myIndex,y2
526             y2[myIndex] = np.concatenate((y2[myIndex],
```

```
                                            y2[myIndex+1]))
            y2.pop(myIndex+1)
            t2[myIndex] = np.concatenate((t2[myIndex],
                                            t2[myIndex+1]))
            t2.pop(myIndex+1)
            y3sd.pop(myIndex)

            if myIndex == len(y2)-1:
                i3 = [-1]
            elif myIndex == 0:
                i3 = [0]
            else:
                i3 = [-1,0]
            #print myIndex, i3, len(y2)
            for i in i3:
                t3 = []
                t3.extend(t2[myIndex+i])
                t3.extend(t2[myIndex+i+1])
                y3 = []
                y3.extend(y2[myIndex+i])
                y3.extend(y2[myIndex+i+1])
                y3sd[myIndex+i] = calc_stddev_of_resid(t3, y3)
            if len(y2) < 3:
                break
    elif myOption == 'nseg':
        for i2 in range(myCutoff):
            #print i2
            myMins.append(np.min(y3sd))
            myDelta.append(myMins[i2+1]-myMins[i2])
            myX.append(myMins[i2+1]/np.mean(myMins[:(i2+2)]))
            myIndex = y3sd.index(myMins[-1])

            y2[myIndex] = np.concatenate((y2[myIndex],
                                            y2[myIndex+1]))
            y2.pop(myIndex+1)
            t2[myIndex] = np.concatenate((t2[myIndex],
                                            t2[myIndex+1]))
            t2.pop(myIndex+1)
            y3sd.pop(myIndex)

            if myIndex == len(y2)-1:
                i3 = [-1]
            elif myIndex == 0:
                i3 = [0]
            else:
                i3 = [-1,0]
            #print myIndex, i3, len(y2)
            for i in i3:
                t3 = []
                t3.extend(t2[myIndex+i])
                t3.extend(t2[myIndex+i+1])
                y3 = []
                y3.extend(y2[myIndex+i])
                y3.extend(y2[myIndex+i+1])
```

```
581                 y3sd[myIndex+i] = calc_stddev_of_resid(t3, y3)
582             if len(y2) < 3:
583                 break
584 #    print y3sd
585     myOut = np.array(t2),np.array(y2)
586     return myOut
587 
588 def bottom_up_aggr_std(t,y,sigma=1.,**kwargs):
589     ''' This version standardizes the y '''
590     t = np.asarray(t)
591     y = np.asarray(y)
592     y = (y-np.mean(y))/np.std(y)
593 
594     if 'nseg' in kwargs:
595         myNumSegs = kwargs['nseg']
596         myCutoff = len(y) - myNumSegs
597         myOption = 'nseg'
598         if myNumSegs == 1:
599             return t,y
600     else:
601         myCutoff_sd = sigma
602         myOption = 'sigma'
603         myCutoff = len(y) - 3
604 
605     def calc_stddev_of_resid(T,Y):
606         T = np.array(T)
607         Y = np.array(Y)
608         A = np.vstack((T,np.ones(len(T))))
609         fit = np.linalg.lstsq(A.T,Y)
610         resid = np.dot(fit[0],A)-Y
611         return np.std(resid)
612 
613     if myOption == 'sigma':
614         y3sd = calc_stddev_of_resid(t, y)
615         if y3sd < myCutoff_sd:
616             return [t],[y]
617 
618     t2 = list()
619     y2 = list()
620 
621     L1 = len(y)
622     L2 = int(np.floor(L1/2))
623 
624     ### Pair Up Points ###
625     for i in range(L2):
626         t2.append(t[2*i:2*i+2])
627         y2.append(y[2*i:2*i+2])
628     # Check to see if sequence is odd
629     if np.mod(L1,2) == 1:
630         t2.append([t[-1]])
631         y2.append([y[-1]])
632 
633     ### Calculate Linear Regression of adj. sets
634     ### Calculate std. dev.
```

```
635     y3sd = list()
636     for i in range(len(y2)-1):
637         t3 = []
638         t3.extend(t2[i])
639         t3.extend(t2[i+1])
640         y3 = []
641         y3.extend(y2[i])
642         y3.extend(y2[i+1])
643         y3sd.append(calc_stddev_of_resid(t3, y3))
644 #    print y3sd
645     myMins = [0]
646     myDelta = []
647     myX = []
648 
649     for i2 in range(myCutoff):
650         myMins.append(np.min(y3sd))
651         myDelta.append(myMins[i2+1]-myMins[i2])
652         myX.append(myMins[i2+1]/np.mean(myMins[:(i2+2)]))
653         myIndex = y3sd.index(myMins[-1])
654 
655         #print i2,myIndex,y
656         y2[myIndex] = np.concatenate((y2[myIndex], y2[myIndex+1]))
657         y2 = np.delete(y2,myIndex+1)
658         t2[myIndex] = np.concatenate((t2[myIndex], t2[myIndex+1]))
659         t2 = np.delete(t2,myIndex+1)
660         y3sd.pop(myIndex)
661 
662 #        print len(t),y3sd
663         if myIndex == len(y2)-1:
664             i3 = [-1]
665         elif myIndex == 0:
666             i3 = [0]
667         else:
668             i3 = [-1,0]
669 #        print myIndex, i3, len(y2)
670         for i in i3:
671             t3 = []
672             t3.extend(t2[myIndex+i])
673             t3.extend(t2[myIndex+i+1])
674             y3 = []
675             y3.extend(y2[myIndex+i])
676             y3.extend(y2[myIndex+i+1])
677             y3sd[myIndex+i] = calc_stddev_of_resid(t3, y3)
678         if myOption == 'sigma':
679             if min(y3sd) >= myCutoff_sd:
680 #                print 'y3sd:',y3sd
681                 break
682 #        print 'y3sd:',y3sd
683     myOut = t2,y2
684     return myOut
685 
686 def bottom_up_aggr_2(t,y,*args,**kwargs):
687     t2,y2 = bottom_up_aggr(t,y,*args,**kwargs)
688     iter_range = range(len(t2))[1:-1]
```

```
    for i in iter_range:
        if len(t2[i]) == 2:
            l1 = len(t2[i-1])
            l2 = len(t2[i+1])
            if np.all((l1>2,l2>2),axis=0):
                t_ = t2[i]
                t2[i-1] = np.concatenate((t2[i-1],[t_[0]]))
                t2[i+1] = np.concatenate(([t_[1]],t2[i+1]))
                y_ = y2[i]
                y2[i-1] = np.concatenate((y2[i-1],[y_[0]]))
                y2[i+1] = np.concatenate(([y_[1]],y2[i+1]))
                t2 = np.delete(t2,i,0)
                y2 = np.delete(y2,i,0)
                iter_range.pop(-1)
                i -= 1
    return t2,y2

def weighted_residual_squared(coef,t1,y1,w1):
    ''' This calculates weighted residual squared '''
    new_x = np.vstack((t1,np.ones(len(t1))))
    new_y = np.dot(coef,new_x)
    resid = y1 - new_y
    r2sum = sum(resid*w1*resid)
    return r2sum

def bottom_up_aggr_weighted(t,y,w,*args,**kwargs):
    t = np.asarray(t)
    y = np.asarray(y)
    w = np.asarray(w)

    # Determine which mode it is in
    if len(args) == 1:
        myCutoff_sd = args[0]
        myOption = 'sigma'
    else:
        if 'sigma' in kwargs:
                myCutoff_sd = kwargs['sigma']
                myOption = 'sigma'
        elif 'nseg' in kwargs:
                myNumSegs = kwargs['nseg']
                myCutoff = int(np.ceil(len(y)/2.) - myNumSegs)
                myOption = 'nseg'
                if myNumSegs == 1:
                    return [t],[y]

    # Check to see if segmentation is necessary
    if myOption == 'sigma':
        A = np.vstack((t,np.ones(len(t))))
        result = np.linalg.lstsq(A.T,y)
        coef = result[0]

        wsd = weighted_residual_squared(coef,t,y,w)
        print 'wsd: ',wsd
        if wsd < sigma:
```

```
743             return [t],[y]
744
745     t2,y2,w2 = [],[],[]
746
747     L1 = len(y)
748     L2 = int(np.floor(L1/2))
749
750     ### Pair Up Points ###
751     for i in range(L2):
752         t2.append(t[2*i:2*i+2])
753         y2.append(y[2*i:2*i+2])
754         w2.append(w[2*i:2*i+2])
755     # Check to see if sequence is odd
756     if np.mod(L1,2) == 1:
757         t2.append([t[-1]])
758         y2.append([y[-1]])
759         w2.append([w[-1]])
760
761
762     ### Calculate Linear Regression of adj. sets
763     ### Calculate std. dev.
764     y3sd = list()
765     for i in range(len(y2)-1):
766         t3 = []
767         t3.extend(t2[i])
768         t3.extend(t2[i+1])
769         y3 = []
770         y3.extend(y2[i])
771         y3.extend(y2[i+1])
772         w3 = []
773         w3.extend(w2[i])
774         w3.extend(w2[i+1])
775 #         print w2,w3,i
776         A = np.vstack((t3,np.ones(len(t3))))
777
778         result = np.linalg.lstsq(A.T,y3)
779         coef = result[0]
780 #         wsd = np.sqrt(weighted_residual_squared(coef,t3,y3,w3))
781 #         print coef,t3,y3,w3
782         wsd = weighted_residual_squared(coef,t3,y3,w3)
783         y3sd.append(wsd)
784
785     #print y3sd
786     myMins = [0]
787     myDelta = []
788     myX = []
789 #     myNumSegs = 4
790 #     myCutoff = len(y2) - myNumSegs
791 #     myCutoff_sd = sigma
792
793     if myOption == 'sigma':
794         i2 = -1
795         while min(y3sd) < myCutoff_sd:
796             i2 += 1
```

```
797             #print i2
798             myMins.append(np.min(y3sd))
799             myDelta.append(myMins[i2+1]-myMins[i2])
800             myX.append(myMins[i2+1]/np.mean(myMins[:(i2+2)]))
801             myIndex = y3sd.index(myMins[-1])
802 
803             y2[myIndex] = np.concatenate((y2[myIndex],
804                                           y2[myIndex+1]))
805             y2.pop(myIndex+1)
806             t2[myIndex] = np.concatenate((t2[myIndex],
807                                           t2[myIndex+1]))
808             t2.pop(myIndex+1)
809             w2[myIndex] = np.concatenate((w2[myIndex],
810                                           w2[myIndex+1]))
811             w2.pop(myIndex+1)
812             y3sd.pop(myIndex)
813 
814             if myIndex == len(y2)-1:
815                 i3 = [-1]
816             elif myIndex == 0:
817                 i3 = [0]
818             else:
819                 i3 = [-1,0]
820             for i in i3:
821                 t3 = []
822                 t3.extend(t2[myIndex+i])
823                 t3.extend(t2[myIndex+i+1])
824                 y3 = []
825                 y3.extend(y2[myIndex+i])
826                 y3.extend(y2[myIndex+i+1])
827                 w3 = []
828                 w3.extend(w2[myIndex+i])
829                 w3.extend(w2[myIndex+i+1])
830                 A = np.vstack((t3,np.ones(len(t3))))
831 
832                 result = np.linalg.lstsq(A.T,y3)
833                 coef = result[0]
834                 wsd = weighted_residual_squared(coef,t3,y3,w3)
835                 y3sd[myIndex+i] = wsd
836             if len(y2) < 3:
837                 break
838     else:
839         for i2 in range(myCutoff):
840             myMins.append(np.min(y3sd))
841             myDelta.append(myMins[i2+1]-myMins[i2])
842             myX.append(myMins[i2+1]/np.mean(myMins[:(i2+2)]))
843             myIndex = y3sd.index(myMins[-1])
844 
845             y2[myIndex] = np.concatenate((y2[myIndex],
846                                           y2[myIndex+1]))
847             y2.pop(myIndex+1)
848             t2[myIndex] = np.concatenate((t2[myIndex],
849                                           t2[myIndex+1]))
850             t2.pop(myIndex+1)
```

```
851             w2[myIndex] = np.concatenate((w2[myIndex],
852                                           w2[myIndex+1]))
853             w2.pop(myIndex+1)
854             y3sd.pop(myIndex)
855 
856             if myIndex == len(y2)-1:
857                 i3 = [-1]
858             elif myIndex == 0:
859                 i3 = [0]
860             else:
861                 i3 = [-1,0]
862             for i in i3:
863                 t3 = []
864                 t3.extend(t2[myIndex+i])
865                 t3.extend(t2[myIndex+i+1])
866                 y3 = []
867                 y3.extend(y2[myIndex+i])
868                 y3.extend(y2[myIndex+i+1])
869                 w3 = []
870                 w3.extend(w2[myIndex+i])
871                 w3.extend(w2[myIndex+i+1])
872                 A = np.vstack((t3,np.ones(len(t3))))
873 
874                 result = np.linalg.lstsq(A.T,y3)
875                 coef = result[0]
876                 wsd = weighted_residual_squared(coef,t3,y3,w3)
877                 y3sd[myIndex+i] = wsd
878             if len(y2) < 3:
879                 break
880     myOut = t2,y2
881     return myOut
882 
883 def _bottom_up_aggr(t,y,*args,**kwargs):
884 
885 #     t = np.asarray(t)
886 #     y = np.asarray(y)
887     t = list(t)
888     y = list(y)
889     assert len(t) == len(y)
890     try:
891         t[0][0]
892     except:
893         print 'The input is not structured correctly'
894         return t,y
895     if len(t) < 2:
896         return t,y
897 
898     if len(args) == 1:
899         myCutoff_sd = args[0]
900         myOption = 'sigma'
901     else:
902         if 'sigma' in kwargs:
903                 myCutoff_sd = kwargs['sigma']
904                 myOption = 'sigma'
```

```
905                 myCutoff = len(y) - 3
906         elif 'nseg' in kwargs:
907                 myNumSegs = kwargs['nseg']
908                 myCutoff = len(y) - myNumSegs
909                 myOption = 'nseg'
910                 if myNumSegs == 1:
911                     return t,y
912         else:
913             raise RuntimeError
914 
915     def calc_stddev_of_resid(T,Y):
916         T = np.array(T)
917         Y = np.array(Y)
918         A = np.vstack((T,np.ones(len(T))))
919         fit = np.linalg.lstsq(A.T,Y)
920         resid = np.dot(fit[0],A)-Y
921         return np.std(resid)
922 
923     #TODO: This is a temporary fix
924     def _calc_stddev_of_resid(T,Y):
925         T = np.array(T)
926         Y = np.array(Y)
927         Y_ = np.interp(T, T[[0,-1]], Y[[0,-1]])
928         return np.std(Y-Y_)
929 
930     ### Calculate Linear Regression of adj. sets
931     ### Calculate std. dev.
932     y3sd = list()
933     for i in range(len(y)-1):
934         t3 = []
935         t3.extend(t[i])
936         t3.extend(t[i+1])
937         y3 = []
938         y3.extend(y[i])
939         y3.extend(y[i+1])
940         y3sd.append(calc_stddev_of_resid(t3, y3))
941 
942     if myOption == 'sigma':
943 #         y3sd_tf =  np.less_equal(y3sd,myCutoff_sd)
944 ##         print 'y3sd:',y3sd
945 #         if np.all(y3sd_tf):
946         if min(y3sd) >= myCutoff_sd:
947             return t,y
948 
949 #    print 'y3sd:',y3sd
950 #    print 'myCutoff_sd:',myCutoff_sd
951     myMins = [0]
952     myDelta = []
953     myX = []
954 
955     for i2 in range(myCutoff):
956         myMins.append(np.min(y3sd))
957         myDelta.append(myMins[i2+1]-myMins[i2])
958         myX.append(myMins[i2+1]/np.mean(myMins[:(i2+2)]))
```

```
959             myIndex = y3sd.index(myMins[-1])
960
961     #        print i2,myIndex,y
962             y[myIndex] = np.concatenate((y[myIndex], y[myIndex+1]))
963             y = np.delete(y,myIndex+1)
964             t[myIndex] = np.concatenate((t[myIndex], t[myIndex+1]))
965             t = np.delete(t,myIndex+1)
966             y3sd.pop(myIndex)
967
968     #        print len(t),y3sd
969             if myIndex == len(y)-1:
970                 i3 = [-1]
971             elif myIndex == 0:
972                 i3 = [0]
973             else:
974                 i3 = [-1,0]
975             #print myIndex, i3, len(y)
976             for i in i3:
977                 t3 = []
978                 t3.extend(t[myIndex+i])
979                 t3.extend(t[myIndex+i+1])
980                 y3 = []
981                 y3.extend(y[myIndex+i])
982                 y3.extend(y[myIndex+i+1])
983                 #print y3sd,i,myIndex
984                 y3sd[myIndex+i] = calc_stddev_of_resid(t3, y3)
985             if myOption == 'sigma':
986                 if min(y3sd) >= myCutoff_sd:
987     #                print 'y3sd:',y3sd
988                     break
989     #        print 'y3sd:',y3sd
990         myOut = t,y
991         return myOut
992
993     def return_break_index(original,segmented):
994         myOut = []
995         original = np.array(original)
996         for seg in segmented:
997             p1 = int(np.where(original == seg[0])[0])
998             p2 = int(np.where(original == seg[-1])[0])
999             myOut.append([p1,p2])
1000        return myOut
1001
1002    def top_down_aggr(t,y,sigma=1.0,depth=1):
1003        ''' Coded based on psuedocode from 'Chapter 1: Segmenting
1004            Time Series: A Survey and Novel Approach' by Keogh, Chu,
1005            Hart and Pazzani in 'Data Mining in Time
1006            Series Databases' 2004
1007        '''
1008        assert len(t) == len(y)
1009        t = np.asarray(t)
1010        y = np.asarray(y)
1011
1012        if len(t) < 7:
```

```
1013            return [t],[y]
1014        #TODO: this is a temporary fix
1015        #def lin_interp_resid(T,Y):
1016        def linreg(T,Y):
1017            T = np.array(T)
1018            Y = np.array(Y)
1019            Y_ = np.interp(T, T[[0,-1]], Y[[0,-1]])
1020            return Y-Y_
1021    #    def _linreg(T,Y):
1022    #        T = np.array(T)
1023    #        Y = np.array(Y)
1024    #        A = np.vstack((T,np.ones(len(T))))
1025    #        fit = np.linalg.lstsq(A.T,Y)
1026    #        resid = np.dot(fit[0],A)-Y
1027    #        return resid
1028        if np.std(linreg(t,y)) < sigma:
1029            return [t],[y]
1030
1031        bestsofar = np.inf
1032        bestsofar_l = 0
1033        bestsofar_r = 0
1034        best_t = [t]
1035        best_y = [y]
1036        for i in range(3,len(t)-3):
1037            tl, tr = t[:i],t[(i):]
1038            yl, yr = y[:i],y[(i):]
1039            sdl = linreg(tl,yl)
1040            sdr = linreg(tr,yr)
1041            #sd_temp = (sdl+sdr)/len(t)
1042            # Assumption that normal is wrong!!
1043            # sd_temp = (sdl**2+sdr**2)**.5
1044            sd_temp = np.std(np.hstack((sdl,sdr)))
1045            if sd_temp < bestsofar:
1046                bestsofar = sd_temp
1047                bestsofar_l = np.std(sdl)
1048                bestsofar_r = np.std(sdr)
1049                best_t = [tl,tr]
1050                best_y = [yl,yr]
1051
1052        if bestsofar_l > sigma:
1053            #print 'left'
1054            tl, yl = top_down_aggr(best_t[0], best_y[0],
1055                                   sigma, depth=depth+1)
1056            best_t.pop(0)
1057            best_t = tl+best_t
1058            best_y.pop(0)
1059            best_y = yl+best_y
1060        if bestsofar_r > sigma:
1061            #print 'right'
1062            tr, yr = top_down_aggr(best_t[-1], best_y[-1],
1063                                   sigma, depth=depth+1)
1064            best_t.pop(-1)
1065            best_t = best_t+tr
1066            best_y.pop(-1)
```

```
        best_y = best_y+yr
    return best_t, best_y

def top_down_aggr_std(t,y,sigma=1.0,depth=1):
    ''' Coded based on psuedocode from 'Chapter 1: Segmenting
        Time Series: A Survey and Novel Approach' by Keogh, Chu,
        Hart and Pazzani in 'Data Mining in Time Series
        Databases' 2004
        This version standardizes the y beforehand
    '''
    assert len(t) == len(y)
    t = np.asarray(t)
    y = np.asarray(y)
    y = (y-np.mean(y))/np.std(y)

    if len(t) < 7:
        return [t],[y]
    def linreg(T,Y):
        T = np.array(T)
        Y = np.array(Y)
        A = np.vstack((T,np.ones(len(T))))
        fit = np.linalg.lstsq(A.T,Y)
        resid = np.dot(fit[0],A)-Y
        return resid
    if np.std(linreg(t,y)) < sigma:
        return [t],[y]

    bestsofar = np.inf
    for i in range(3,len(t)-3):
        tl, tr = t[:i],t[(i):]
        yl, yr = y[:i],y[(i):]
        sdl = linreg(tl,yl)
        sdr = linreg(tr,yr)
        #sd_temp = (sdl+sdr)/len(t)
        # Assumption that normal is wrong!!
        #sd_temp = (sdl**2+sdr**2)**.5
        sd_temp = np.std(np.hstack((sdl,sdr)))
        if sd_temp < bestsofar:
            bestsofar = sd_temp
            bestsofar_l = np.std(sdl)
            bestsofar_r = np.std(sdr)
            best_t = [tl,tr]
            best_y = [yl,yr]

    if bestsofar_l > sigma:
        #print 'left'
        tl, yl = top_down_aggr(best_t[0], best_y[0],
                               sigma, depth=depth+1)
        best_t.pop(0)
        best_t = tl+best_t
        best_y.pop(0)
        best_y = yl+best_y
    if bestsofar_r > sigma:
        #print 'right'
```

```
1121         tr, yr = top_down_aggr(best_t[-1], best_y[-1],
1122                                sigma, depth=depth+1)
1123         best_t.pop(-1)
1124         best_t = best_t+tr
1125         best_y.pop(-1)
1126         best_y = best_y+yr
1127     return best_t, best_y
1128
1129 def hybrid_aggr(t,y,sigma=None,nseg=None):
1130     assert len(t) == len(y)
1131     t = np.asarray(t)
1132     y = np.asarray(y)
1133
1134     if np.all((sigma is None, nseg is None)):
1135         print 'sigma or nseg must be specified'
1136         return [t],[y]
1137     if np.all((sigma is not None, nseg is not None)):
1138         print 'please specify only sigma or nseg'
1139         return [t],[y]
1140
1141     if sigma is not None:
1142         if np.std(y) < sigma:
1143             return [t],[y]
1144         sigma2 = sigma/2
1145         t1,y1 = top_down_aggr(t,y,sigma=sigma2)
1146 #        for t_,y_ in zip(t1,y1):
1147 #            plt.plot(t_,y_)
1148 #        plt.show()
1149 #        print 'sigma,sigma2:',sigma,sigma2
1150         t1,y1 = _bottom_up_aggr(t1,y1,sigma=sigma)
1151         return t1,y1
1152     else:
1153         y1 = [[0]]
1154         sigma2 = np.std(y)/nseg
1155         while len(y1) < nseg*2:
1156             t1,y1 = top_down_aggr(t,y,sigma=sigma2)
1157             sigma2 *=.5
1158 #        print np.std(y),sigma2
1159 #        for t_,y_ in zip(t1,y1):
1160 #            plot(t_,y_)
1161         t1,y1 = _bottom_up_aggr(t1,y1,nseg=nseg)
1162     return t1,y1
1163
1164 def hybrid_aggr_std(t,y,sigma=1.0,nseg=None):
1165     assert len(t) == len(y)
1166     t = np.asarray(t)
1167     y = np.asarray(y)
1168     y = (y-np.mean(y))/np.std(y)
1169
1170     if nseg is not None:
1171         y1 = [[0]]
1172         sigma2 = np.std(y)/nseg
1173         while len(y1) < nseg*2:
1174             t1,y1 = top_down_aggr(t,y,sigma=sigma2)
```

```
1175             sigma2 *=.5
1176 #         print np.std(y),sigma2
1177 #         for t_,y_ in zip(t1,y1):
1178 #             plot(t_,y_)
1179         t1,y1 = _bottom_up_aggr(t1,y1,nseg=nseg)
1180     else:
1181         sigma2 = sigma/2
1182         t1,y1 = top_down_aggr(t,y,sigma=sigma2)
1183 #         for t_,y_ in zip(t1,y1):
1184 #             plt.plot(t_,y_)
1185 #         plt.show()
1186         t1,y1 = _bottom_up_aggr(t1,y1,sigma=sigma)
1187         return t1,y1
1188
1189     return t1,y1
1190
1191 def optimize_segmentation(t,y,initial_pos,bounds=None):
1192     ''' This finds the segmentation using fmin, which uses a
1193         downhill simplex algorithm. It's very sensitive to
1194         the initial starting position. t and y must be a vector
1195         of time stamps and values
1196         initial_pos is a vector of initial positions of break points
1197         bounds is a list of upper and lower bounds for each break
1198          point in initial _pos
1199      '''
1200
1201     if bounds is None:
1202         bounds = []
1203         for i in range(len(initial_pos)):
1204 #             upper = initial_pos[i] + int(len(t)*.1)
1205 #             lower = initial_pos[i] - int(len(t)*.1)
1206             upper = initial_pos[i] + 5
1207             lower = initial_pos[i] - 5
1208             if lower < 0:
1209                 lower = 0
1210             if upper > len(t):
1211                 upper = len(t)
1212             bounds.append([lower,upper])
1213         for b1,b2 in zip(bounds[:-1],bounds[1:]):
1214             if b1[1] > b2[0]:
1215                 v1 = int((b1[1]+b2[0])/2.)
1216                 b1[1] = v1
1217                 b2[0] = v1
1218     def calc_stddev_of_resid(T,Y):
1219         T = np.array(T)
1220         Y = np.array(Y)
1221         A = np.vstack((T,np.ones(len(T))))
1222         fit = np.linalg.lstsq(A.T,Y)
1223         resid = np.dot(fit[0],A)-Y
1224         return np.std(resid)
1225     def func(x):
1226         x0 = x
1227         x = np.array(x,dtype=np.int)
1228         x1 = x
```

```
1229            r = 0.
1230            for i in range(len(bounds)):
1231                if x[i] > bounds[i][1]:
1232                    r += 100+(x[i]-bounds[i][1])*10
1233                    x[i] = bounds[i][1]
1234                if x[i] < bounds[i][0]:
1235                    r += 100+(bounds[i][1]-x[i])*10
1236                    x[i] = bounds[i][0]
1237            for i in range(len(x)-1):
1238                if x[i] == x[i+1]:
1239                    x[i+1] = x[i] + 1
1240            #print x,x1,x0
1241            brks = np.hstack((t[0],x,t[-1]))
1242            for ind in zip(brks[:-1],brks[1:]):
1243                r += calc_stddev_of_resid(t[ind[0]:ind[1]],
1244                                          y[ind[0]:ind[1]])
1245            return r
1246        result = scipy.optimize.fmin(func,initial_pos,
1247                                     maxiter=100,disp=0,xtol=1)
1248        result = np.hstack((0,np.array(result,dtype=np.int),len(t)))
1249        t_out = []
1250        y_out = []
1251        for i in zip(result[:-1],result[1:]):
1252            t_out.append(t[i[0]:i[1]])
1253            y_out.append(y[i[0]:i[1]])
1254        return t_out,y_out
1255
1256    def make_dataset(ts_fn,X,Z,mc,t,sd,filename,**kwargs):
1257        ''' Makes a dataset for given X and Z inputs.
1258            ts_fn is the time series class object that makes time
1259            series data from the ts_segments module.
1260            Options are: sine, square, triangle, sawtooth, flat,
1261                         sloped and step
1262            X is the pandas Data Frame for inputs
1263            Z is the pandas Data Frame for values for variables
1264                that need to be varied.
1265            The column names should be keywords into the time series
1266                function.
1267            mc is the number of repititions of the time series
1268            t is for list of time steps
1269            sd is the standard deviation for noise that is normally
1270                distributed
1271            filename is the name of the saved file
1272            kwargs has parameters for other parts of the ts generating
1273                function
1274                wave fn: amplitude, n_waves, phase, (square: ratio)
1275                sloped: (slope and ystart) or (ystart and yend)
1276                step: separation, loc
1277        '''
1278        t = np.array(t)
1279        ts_attr = {}
1280        ts_attr['t'] = t
1281        ts_attr['sd'] = sd
1282
```

```
1283      ts_attr.update(kwargs)
1284
1285      df = None
1286      design_input_names = list(X.columns)
1287      design_input_names.append('mc')
1288      ts_input_names = list(Z.columns)
1289      design_input_col_index = pandas.Index(design_input_names,
1290                                            name='inputs')
1291      attributes_col_index = pandas.Index(['case'],name='attributes')
1292      time_step_col_index = pandas.Index(t,name='time')
1293
1294      for i_doe in range(len(X)):
1295          # Grab the DoE variable values
1296          ins = X.take([i_doe]).values
1297          # Prepare the attributes
1298          df_attr = pandas.DataFrame([i_doe],
1299                                     columns = attributes_col_index)
1300          # Modify inputs into time series function
1301          d = dict(zip(ts_input_names,Z.take([i_doe]).values))
1302          ts_attr.update(d)
1303
1304          for i_mc in range(mc):
1305              ts = ts_fn(**ts_attr)
1306              # Prepare the doe inputs to be merged
1307              data1 = np.asarray(ins).flatten()
1308              data1 = np.append(data1,i_mc)
1309              df1 = pandas.DataFrame([data1],
1310                                     columns=design_input_col_index)
1311              # Prepare the ts outputs to be merged
1312              data2 = np.round(ts.y,3)
1313              df2 = pandas.DataFrame([data2],
1314                                     columns=time_step_col_index)
1315              # Merge the inputs, attributes and ts outputs
1316              data3 = {'inputs':df1,
1317                       'attributes':df_attr,'outputs':df2}
1318              df3 = pandas.concat(data3,axis=1,names=['L1','L2'])
1319
1320              if isinstance(df, pandas.DataFrame):
1321                  df = pandas.concat([df,df3],
1322                                     axis=0, ignore_index=True)
1323              else:
1324                  df = df3.copy()
1325      df.to_csv(filename)
1326
1327  def from_pandas_csv(filename):
1328      ''' This function takes csv files exported via pandas
1329          and returns a TimeSeriesDB object.
1330      '''
1331      df = pandas.DataFrame.from_csv(filename)
1332      # Make columns multi level index
1333      cols = df.columns
1334      cols = [eval(x) for x in cols]
1335      cols = pandas.MultiIndex.from_tuples(cols)
1336      df.columns = cols
```

```
    df_in = df.xs('inputs',axis=1)
    inputs_names = list(df_in.columns)
    ins = df_in.values
    outs = df.xs('outputs',axis=1).values
    outputs_names = [str(x) for x in range(len(outs[0,:]))]
    if 'attributes' in df.columns.levels[0]:
        df_attr = df.xs('attributes',axis=1)
        attr_names = list(df_attr.columns)
        attr = df_attr.values
    else:
        attr = None

    tsDB = TimeSeriesDB.TimeSeriesDB()
    for i in range(len(df_in)):
        x_in = ins[i,:]
        x_in = dict(zip(inputs_names,x_in))
        y_in = outs[i,:]
        t = np.arange(len(y_in))
        if attr is None:
            d = {'inputs':x_in,'t':t,'y':y_in,'case':i}
        else:
            attr_in = attr[i,:]
            attr_in = dict(zip(attr_names,attr_in))
            d = {'inputs':x_in,'t':t,'y':y_in}
            d.update(attr_in)
        tsDB.record(d)
    return tsDB
```

A.3 my_functions.py

```
import matplotlib.pyplot as plt
from matplotlib import offsetbox
import numpy as np

# Scale and visualize the embedding vectors
def plot_embedding_1D(X, images=None, title=None,
                      xlim = [-.1,1.1], fig=None):
    x_min, x_max = np.min(X, 0), np.max(X, 0)
    X = (X - x_min) / (x_max - x_min)

    sweep = np.linspace(0, 1, 11)
    if fig is None:
        fig = plt.figure(figsize=(20,5))
    ax = plt.subplot(111)
#    yloc = np.sin(np.arange(len(X))*2)/4
    yloc = np.random.uniform(-.25,.25,len(X))
#    yloc = 0
    for i in range(len(X)):
        if X[i] > xlim[0]:
            plt.plot(X[i], yloc[i],'o')

    if images is not None:
        if hasattr(offsetbox, 'AnnotationBbox'):
            # only print thumbnails with matplotlib > 1.0
            for sw in sweep:
```

```
                i = arg_closest(sw, X)
                if np.all((X[i] >= xlim[0],X[i] <= xlim[1])):
                    imagebox = offsetbox.AnnotationBbox(
                        offsetbox.OffsetImage(images[i],
                                              cmap = plt.cm.gray_r,
                                              zoom=.9),
                        (X[i],.8),pad=0.1)
                    ax.add_artist(imagebox)

    plt.xlim(xlim)
#    plt.ylim((-.5,1.5))
    plt.ylim((-.5,1.5))
    #plt.xticks([])
    plt.yticks([])
    if title is not None:
        plt.title(title)
    return fig


def plot_pca_1D_hist(X, dat, ts_resample='auto',
                     dot_size=0, fig=None,
                     font_settings=None, yticks=None):
    ''' plots the image of the time series along the first
        PCA axis with equal spacing '''
    minimum = np.min(np.min(dat))
    minimum -= np.abs(minimum)*.1
    maximum = np.max(np.max(dat))
    if ts_resample == 'auto':
        L = len(dat[0, :])
        ts_resample = int(L / 65)

    x_min, x_max = np.min(X, 0), np.max(X, 0)
    X = (X - x_min) / (x_max - x_min)

    sweep = np.linspace(0, 1, 11)
    if fig is None:
        fig = plt.figure(figsize=(20, 5))
    if font_settings is None:
        font_settings = {'fontname': 'sans', 'fontsize': 12}
    ax2 = plt.subplot(212)

    ax2.hist(X, bins=50, normed=False, color='b', edgecolor='b')
    plt.xlim([-.1, 1.1])
    ax2.yaxis.labelpad = -10
    plt.xlabel('PC 1', **font_settings)
    if yticks:
        plt.yticks(yticks)

    ax1 = plt.subplot(211)

#    if images is not None:
    if True:
        if hasattr(offsetbox, 'AnnotationBbox'):
            # only print thumbnails with matplotlib > 1.0
```

```
80              for sw in sweep:
81                  i = arg_closest(sw, X)
82                  y = dat[i,:]
83                  img = make_ts_img(y, ts_resample, dot_size,
84                                    minimum, maximum)
85                  imagebox = offsetbox.AnnotationBbox(
86                      offsetbox.OffsetImage(img, cmap=plt.cm.gray_r,
87                                            zoom=1),
88                      (X[i],0),pad=0.1)
89                  ax1.add_artist(imagebox)
90      plt.ylim([-1,1])
91      plt.xlim([-.1,1.1])
92      plt.xticks([])
93      plt.yticks([])
94      ax2.set_xticks(np.arange(0,1.1,.2))
95      plt.setp(ax2.get_xticklabels(),**font_settings)
96
97      ax2.spines['top'].set_visible(False)
98      ax1.spines['bottom'].set_visible(False)
99      fig.subplots_adjust(hspace=0)
100     return fig
101
102 def arg_closest(target,lst):
103     ta = np.array(target)
104     ls = np.array(lst)
105     ls = ls-ta
106     ls = np.absolute(ls)
107     ls1 = np.array((range(len(ls)),ls))
108     ls1 = ls1.T
109     ls1 = ls1[np.argsort(ls1[:,1])]
110     return int(ls1[0,0])
111
112 def make_ts_img(yvals,resample=3,dot_size=0,minimum=0,maximum=1):
113     yvals = yvals[::resample]
114     yvals = normalize(yvals,minimum,maximum)
115     dot_size=int(dot_size)
116     l = len(yvals)
117     buff = np.floor(l*.05)
118     if buff < 1:
119         buff = 1
120     img_temp = np.zeros((l,l))
121     y = yvals/(np.max(yvals,0))*(l-buff*2)
122     y = np.floor(y)
123     for i in range(l):
124         r1 = l-y[i] - dot_size
125         r2 = l-y[i] + 1+dot_size
126         c1 = i - dot_size
127         c2 = i + 1+dot_size
128         img_temp[r1:r2,c1:c2] = 1
129 #         img_temp[l-y[i]][i] = 1
130     return img_temp
131
132 def normalize(vec,minimum=None,maximum=None):
133     ''' Normalize a Vector of values to 0 to 1 or given values '''
```

```
134     vec = np.array(vec)
135     if minimum is None:
136         mi = np.float(np.min(vec))
137     else:
138         mi = np.float(minimum)
139     if maximum is None:
140         ma = np.float(np.max(vec))
141     else:
142         ma = np.float(maximum)
143     myOut = (vec-mi)/(ma-mi)
144     return myOut
145
146 def normalize2(vec,minimum,maximum):
147     ''' Normalize a Vector of values where min and max are
148         specified
149     '''
150     normalized = normalize(vec)
151     mi = np.float64(minimum)
152     ma = np.float64(maximum)
153     myOut = (normalized)*np.abs(ma-mi) + mi
154     return myOut
155
156 def movingaverage(interval, window_size,mode='valid'):
157     window = np.ones(int(window_size))/float(window_size)
158     return np.convolve(interval, window, mode)
159
160 def combine_mu(mus,ns):
161     assert len(mus) == len(ns)
162     mus = np.array(mus,dtype=np.float)
163     ns = np.array(ns,dtype=np.float)
164     return np.sum(np.dot(mus,ns))/np.sum(ns)
165
166 def combine_sd(mus,sds,ns):
167     assert len(mus) == len(ns)
168     mus = np.array(mus,dtype=np.float)
169     sds = np.array(sds,dtype=np.float)
170     ns = np.array(ns,dtype=np.float)
171     n_tot = np.sum(ns)
172     MU = combine_mu(mus,ns)
173     sum1 = np.sum(np.dot((ns-1),sds**2) + np.dot(ns,mus**2))
174     sum2 = n_tot*MU**2
175     return np.sqrt((sum1-sum2)/(n_tot-1))
176
177 def find_overlap_partial(x1, x2):
178     ''' This method finds the partial overlapping region along a
179         1-D axis of two vectors x1 and x2
180
181         inputs
182         ------
183         x1, x2 - list or numpy array of real numbers that are
184                  partially overlapping with x1 being less than x2
185
186         outputs
187         -------
```

```
188         idx1, idx2 - index of x1 and x2 that are overlapping
189     '''
190     x1 = np.array(x1)
191     x2 = np.array(x2)
192     assert np.min(x1) < np.min(x2)
193     assert np.max(x1) < np.max(x2)
194     idx1 = np.argwhere( x1 >= np.min(x2) ).ravel()
195     idx2 = np.argwhere( x2 <= np.max(x1) ).ravel()
196     return idx1, idx2
197
198 def find_overlap_complete(x1, x2):
199     ''' This method finds the complete overlapping region along a
200         1-D axis of two vectors x1 and x2
201
202         inputs
203         ------
204         x1, x2 - list or numpy array of real numbers that are
205                  completely overlapping with x1 encompassing x2
206
207         outputs
208         -------
209         idx1, idx2 - index of x1 and x2 that are overlapping,
210                      values that are the same are also returned
211     '''
212     x1 = np.array(x1)
213     x2 = np.array(x2)
214     assert np.min(x1) < np.min(x2)
215     assert np.max(x1) > np.max(x2)
216     min1, min2 = np.min(x1), np.min(x2)
217     max1, max2 = np.max(x1), np.max(x2)
218
219     idx1a = np.argwhere(x1 >= min2).ravel()
220     idx1b = np.argwhere(x1 <= max2).ravel()
221     #print idx1a, idx1b
222     idx1b_, idx1a_ = find_overlap_partial(idx1b, idx1a)
223     idx1 = idx1b[idx1b_]
224     idx2 = range(len(x2))
225     return idx1, idx2
226
227 def find_overlap(x1, x2):
228     ''' This method finds the overlapping region along a 1-D axis
229         of two vectors x1 and x2
230
231         inputs
232         ------
233         x1, x2 - list or numpy array of real numbers
234
235         outputs
236         -------
237         idx1, idx2 - index of x1 and x2 that are overlapping
238     '''
239
240     x1 = np.array(x1)
241     x2 = np.array(x2)
```

```
242
243     min1, min2 = np.min(x1), np.min(x2)
244     max1, max2 = np.max(x1), np.max(x2)
245
246     # Check how it overlaps
247     # There are 6 possible ways (3 unique) that they can be arranged
248     # There are 4 possible ways (2 unique) that they can overlap
249     mat = np.array([[1, min1],
250                     [1, max1],
251                     [2, min2],
252                     [2, max2]])
253     mat_sorted = mat[ np.argsort(mat[:, 1]).ravel(), :]
254     my_order = mat_sorted[:,0]
255
256     # If my_order is [1 1 2 2] or [2 2 1 1], then no overlap
257     if all(my_order == [1, 1, 2, 2]) or \
258            all(my_order == [2, 2, 1, 1]):
259         return [], []
260     # If my_order is [1 2 1 2],
261     # then partial overlap with max(x1) > min(x2)
262     elif all(my_order == [1, 2, 1, 2]):
263         return find_overlap_partial(x1, x2)
264     # If my_order is [2 1 2 1],
265     # then partial overlap with max(x2) > min(x1)
266     elif all(my_order == [2, 1, 2, 1]):
267         idx2, idx1 = find_overlap_partial(x2, x1)
268         return idx1, idx2
269     # If my_order is [1 2 2 1],
270     # then complete overlap w/ x1 encompassing x2
271     elif all(my_order == [1, 2, 2, 1]):
272         return find_overlap_complete(x1, x2)
273     # If my_order is [2 1 1 2],
274     # then complete overlap w/ x2 encompassing x1
275     elif all(my_order == [2, 1, 1, 2]):
276         idx2, idx1 = find_overlap_complete(x2, x1)
277         return idx1, idx2
278
279 def make_white(fig):
280     '''
281     Sets figure parameters to white for presentations with
282     dark background
283
284     :param fig: matplotlib figure object
285     '''
286     all_axes = fig.get_axes()
287     col = 'w'
288     for ax in all_axes:
289         ya = ax.yaxis
290         # you modify x ticks indepently
291         xa = ax.xaxis
292         ya.set_tick_params(labelcolor=col)  # label
293         ya.set_tick_params(color=col)       # ticks
294         xa.set_tick_params(labelcolor=col)  # label
295         xa.set_tick_params(color=col)       # ticks
```

```
296         ax.spines['bottom'].set_color(col)
297         ax.spines['top'].set_color(col)
298         ax.spines['right'].set_color(col)
299         ax.spines['left'].set_color(col)
300         ax.yaxis.label.set_color(col)
301         ax.xaxis.label.set_color(col)
302     return(fig)
```

A.4 smoothers.py

```
1  import numpy
2  import rpy2 as rp
3  import rpy2.robjects as robjects
4  import pandas
5  import pandas.rpy.common
6  from rpy2.robjects.packages import importr
7  from sklearn import svm
8  from math import factorial
9  import copy
10 from sklearn.decomposition import PCA
11
12 def moving_average(data, window_size=3,mode='same'):
13     window = numpy.ones(int(window_size))/float(window_size)
14     return numpy.convolve(data, window, mode)
15
16 def moving_triangle(data,window_size=3,mode='same'):
17     weight = []
18     degree = window_size/2.
19     for x in range(1,window_size):
20         weight.append(degree-abs(degree-x))
21     w=numpy.array(weight)/(numpy.sum(weight))
22     return numpy.convolve(data, w, mode)
23
24 def moving_gaussian(data,window_size=5,mode='same'):
25     weight = []
26     degree = window_size/2.
27     weightGauss=[]
28     for i in range(window_size):
29         i2=i-degree+0.5
30         frac=i2/float(window_size)
31         gauss=1/(numpy.exp((4*(frac))**2))
32         weightGauss.append(gauss)
33     weight=numpy.array(weightGauss)/(numpy.sum(weightGauss))
34 #    print weight
35     return numpy.convolve(data, weight, mode)
36
37 def moving_median(data,window_size=7):
38     ''' Performs moving median smoothing.
39         window_size must be odd '''
40     if window_size == 1:
41         return data
42     else:
43         return moving_median_set(data,window_size)
44
45 def svr_1d(data,t=None,t_pred=None):
```

```
46      y = numpy.array(data)
47      if t is None:
48          t = numpy.arange(len(data))
49      t_fit = t.reshape((-1,1))
50      svr_fit = svm.SVR()
51      svr_fit.fit(t_fit,y)
52
53      if t_pred is None:
54          t_pred = t_fit
55      elif len(t_pred.shape) < 2:
56          t_pred = t_pred.reshape((-1,1))
57      y_new = svr_fit.predict(t_pred)
58      return numpy.array(y_new)
59
60  def mars_1d(data,t=None,t_pred=None):
61      y = numpy.array(data)
62      if t is None:
63          t = numpy.arange(len(data))
64      d_in = {'t':t}
65      d_ou = {'y':y}
66      df_in = pandas.DataFrame.from_dict(d_in)
67      df_ou = pandas.DataFrame.from_dict(d_ou)
68
69      R = robjects.r
70      earth = importr('earth')
71      R_df_in = pandas.rpy.common.convert_to_r_dataframe(df_in)
72      R_df_ou = pandas.rpy.common.convert_to_r_dataframe(df_ou)
73      fit_mars = R.earth(x=R_df_in,y=R_df_ou,degree=2)
74
75      if t_pred is None:
76          R_df_pred = R_df_in
77      else:
78          d_pred = {'t':t_pred}
79          df_pred = pandas.DataFrame.from_dict(d_pred)
80          temp_fn = pandas.rpy.common
81          R_df_pred = temp_fn.convert_to_r_dataframe(df_pred)
82      y_new = R.predict(fit_mars,newdata=R_df_pred)
83      y_new = numpy.array(list(y_new))
84      return y_new
85
86  def kalman_filter_1d(data,R=0.0001):
87      L = len(data)
88      sz = (L,)
89      xhat=numpy.zeros(sz)       # a posteri estimate of x
90      P=numpy.zeros(sz)          # a posteri error estimate
91      xhatminus=numpy.zeros(sz)  # a priori estimate of x
92      Pminus=numpy.zeros(sz)     # a priori error estimate
93      K=numpy.zeros(sz)          # gain or blending factor
94      z = data
95
96      Q = 1e-5 # process variance
97      # estimate of measurement variance, change to see effect
98  #     R = 0.01**2
99
```

```
100     xhat[0] = 0.0
101     P[0] = 1.0
102
103     for k in range(1,L):
104         # time update
105         xhatminus[k] = xhat[k-1]
106         Pminus[k] = P[k-1]+Q
107
108         # measurement update
109         K[k] = Pminus[k]/( Pminus[k]+R )
110         xhat[k] = xhatminus[k]+K[k]*(z[k]-xhatminus[k])
111         P[k] = (1-K[k])*Pminus[k]
112     return xhat
113
114 def savitzky_golay_simple(y):
115     return savitzky_golay(y, 3, 1)
116
117 def savitzky_golay_auto(y):
118     window_size = int(numpy.max((3,numpy.round(len(y)*.01)*2+1)))
119     order = int(numpy.max((1,numpy.floor(window_size**.5))))
120     return savitzky_golay(y,window_size,order)
121
122 def savitzky_golay(y, window_size, order, deriv=0, rate=1):
123     """ Smooth (and optionally differentiate) data with a
124         Savitzky-Golay filter.  The Savitzky-Golay filter removes
125         high frequency noise from data.
126
127         It has the advantage of preserving the original shape and
128         features of the signal better than other types of filtering
129         approaches, such as moving averages techniques.
130         Parameters
131         ----------
132         y : array_like, shape (N,)
133             the values of the time history of the signal.
134         window_size : int
135             the length of the window. Must be an odd integer number.
136         order : int
137             the order of the polynomial used in the filtering.
138             Must be less then 'window_size' - 1.
139         deriv: int
140             the order of the derivative to compute
141             (default=0 means only smoothing)
142         Returns
143         -------
144         ys : ndarray, shape (N)
145             the smoothed signal (or it's n-th derivative).
146         Notes
147         -----
148         The Savitzky-Golay is a type of low-pass filter,
149         particularly suited for smoothing noisy data. The main idea
150         behind this approach is to make for each point a
151         least-square fit with a polynomial of high order over a
152         odd-sized window centered at the point.
153         Examples
```

```
154         --------
155         t = numpy.linspace(-4, 4, 500)
156         y = numpy.exp( -t**2 ) + numpy.random.normal(0, 0.05,
157                                                      t.shape)
158         ysg = savitzky_golay(y, window_size=31, order=4)
159         import matplotlib.pyplot as plt
160         plt.plot(t, y, label='Noisy signal')
161         plt.plot(t, numpy.exp(-t**2), 'k',
162                  lw=1.5, label='Original signal')
163         plt.plot(t, ysg, 'r', label='Filtered signal')
164         plt.legend()
165         plt.show()
166         References
167         ----------
168         .. [1] A. Savitzky, M. J. E. Golay,
169                Smoothing and Differentiation of
170                Data by Simplified Least Squares Procedures.
171                Analytical Chemistry, 1964, 36 (8), pp 1627-1639.
172         .. [2] Numerical Recipes 3rd Edition:
173                The Art of Scientific Computing
174                W.H. Press, S.A. Teukolsky, W.T. Vetterling,
175                B.P. Flannery Cambridge University Press
176                ISBN-13: 9780521880688
177         ---------
178         http://www.scipy.org/Cookbook/SavitzkyGolay
179     """
180     try:
181         window_size = numpy.abs(numpy.int(window_size))
182         order = numpy.abs(numpy.int(order))
183     except ValueError, msg:
184         myMsg = "window_size and order have to be of type int"
185         raise ValueError(myMsg)
186     if window_size % 2 != 1 or window_size < 1:
187         myMsg = "window_size size must be a positive odd number"
188         raise TypeError(myMsg)
189     if window_size < order + 2:
190         myMsg = "window_size is too small for the polynomials order"
191         raise TypeError(myMsg)
192     order_range = range(order+1)
193     half_window = (window_size -1) // 2
194     # precompute coefficients
195     b = numpy.mat([[k**i for i in order_range] for k in \
196                    range(-half_window, half_window+1)])
197     m = numpy.linalg.pinv(b).A[deriv] * \
198         rate**deriv * factorial(deriv)
199     # pad the signal at the extremes with
200     # values taken from the signal itself
201     firstvals = y[0] - numpy.abs( y[1:half_window+1][::-1] - y[0] )
202     lastvals = y[-1] + numpy.abs(y[-half_window-1:-1][::-1] - y[-1])
203     y = numpy.concatenate((firstvals, y, lastvals))
204     return numpy.convolve( m[::-1], y, mode='valid')
205
206
207 def _set_up_ndarray(data,window_size,w=None):
```

```
208     assert window_size%2 == 1
209     L = window_size
210     if w is None:
211         w = numpy.ones(L)
212     ys = numpy.asmatrix(data)
213     nr,nc = ys.shape
214     ys2 = None
215     for i in range(L):
216         ys1 = numpy.hstack((numpy.ones((nr,i))*numpy.nan,ys,
217                             numpy.ones((nr,L-i-1))*numpy.nan))
218         ys1 *= w[i]
219         if ys2 is None:
220             ys2 = ys1
221         else:
222             ys2 = numpy.vstack((ys2,ys1))
223     L2 = (L-1)/2
224     ys2 = numpy.asarray(ys2[:,L2:-L2])
225     ys2_masked = numpy.ma.masked_array(ys2)
226     ys2_masked[numpy.isnan(ys2)] = numpy.ma.masked
227     return ys2_masked
228
229
230 def moving_average_set(data, window_size=3):
231     data = numpy.matrix(data)
232     if window_size == 1:
233         myOut = numpy.mean(data,axis=0)
234         myOut = numpy.asarray(myOut).ravel()
235     else:
236         ys2_masked = _set_up_ndarray(data, window_size)
237         myOut = numpy.ma.mean(ys2_masked,axis=0)
238     return numpy.asarray(myOut)
239
240 def moving_triangle_set(data,window_size=3,mode='same'):
241     weight = []
242     degree = (window_size+1)/2.
243     for x in range(1,window_size+1):
244         weight.append(degree-abs(degree-x))
245     w=numpy.array(weight)/(numpy.sum(weight))/len(data)
246 #    print w
247     y_masked = _set_up_ndarray(data, window_size, w)
248     return numpy.ma.sum(y_masked,axis=0)
249
250 def moving_gaussian_set(data,window_size=5,mode='same'):
251     weight = []
252     degree = window_size/2.
253     weightGauss=[]
254     for i in range(window_size):
255         i2=i-degree+0.5
256         frac=i2/float(window_size)
257         gauss=1/(numpy.exp((4*(frac))**2))
258         weightGauss.append(gauss)
259     weight = numpy.array(weightGauss)/(numpy.sum(weightGauss))
260     weight /= len(data)
261 #    print weight
```

```
262     y_masked = _set_up_ndarray(data,window_size,weight)
263     return numpy.ma.sum(y_masked,axis=0)
264
265 def moving_median_set(ys,window_size=7):
266     ''' Performs moving median smoothing.
267         window_size must be odd '''
268     if window_size == 1:
269         myOut = numpy.median(ys,axis=0)
270     else:
271         ys2_masked = _set_up_ndarray(ys,window_size)
272         #print ys2_masked
273         myOut = numpy.ma.median(ys2_masked,axis=0)
274     return numpy.asarray(myOut)
275
276 def moving_max_set(ys,window_size=3):
277     ''' Perform moving max smoothing. '''
278     if window_size == 1:
279         myOut = numpy.max(ys,axis=0)
280     else:
281         ys2_masked = _set_up_ndarray(ys,window_size)
282         #print ys2_masked
283         myOut = numpy.ma.max(ys2_masked,axis=0)
284     return numpy.asarray(myOut)
285
286 def moving_min_set(ys,window_size=3):
287     ''' Perform moving min smoothing. '''
288     if window_size == 1:
289         myOut = numpy.min(ys,axis=0)
290     else:
291         ys2_masked = _set_up_ndarray(ys,window_size)
292         #print ys2_masked
293         myOut = numpy.ma.min(ys2_masked,axis=0)
294     return numpy.asarray(myOut)
295
296 def svr_1d_set(data,t=None):
297     ''' Performs SVR on a set of repeated time series data.
298         Each time series should be in the row and the columns
299         are the time steps
300     '''
301     data = numpy.array(data)
302     nr,nc = data.shape
303     if t is None:
304         t = numpy.array(range(nc)*nr)
305     else:
306         t = numpy.array(t)
307         t = t.ravel()
308     return svr_1d(data.ravel(),t,numpy.arange(nc))
309
310 def mars_1d_set(data,t=None):
311     ''' Perform MARS on a set of repeated time series data '''
312     data = numpy.array(data)
313     nr,nc = data.shape
314     if t is None:
315         t = numpy.array(range(nc)*nr)
```

```
316     else:
317         t = numpy.array(t)
318         t = t.ravel()
319     return mars_1d(data.ravel(),t,numpy.arange(nc))
320
321 def calculate_residuals(func, data, **kwargs):
322     ''' Calculates the residuals by estimating the signal
323         using the func and taking the difference of data and
324         estimated signal
325     '''
326     data = numpy.asarray(data)
327     est = func(data,**kwargs)
328     resid = data - est
329     return resid
```

A.5 FeatureExtract.py

```
1  import numpy as np
2  from itertools import izip
3  from sklearn.decomposition import PCA, ProbabilisticPCA
4  from sklearn.decomposition import RandomizedPCA
5  from sklearn.manifold import LocallyLinearEmbedding, Isomap
6  import warnings
7  import scipy.stats
8
9  import my_functions
10 import ApplyClustering
11 import copy
12
13 def ApplyAll(tsDB, n_components=3):
14     ApplyPCA(tsDB, n_components=n_components, downsample=1)
15     ApplyPCAmod(tsDB)
16     ApplyMean(tsDB)
17     ApplyStd(tsDB)
18     ApplyFFT(tsDB, n_components=n_components)
19     ApplyLLE(tsDB, n_components=n_components, downsample=1)
20     ApplyIsomap(tsDB, n_components=n_components, downsample=1)
21     ApplyPCA_Breakpoints(tsDB, n_components=n_components)
22     ApplyBase2_Breakpoints(tsDB)
23
24 def ApplyAll_auto(tsDB, reps=100):
25     dat = tsDB.get_y().xs('y',axis=1).values
26     n_components = determine_n_components(dat,reps)
27     print ApplyAll_auto.func_name, 'n_components:', n_components
28     # if n_components is 0, there is no point in
29     # doing feature extraction
30     if n_components > 0:
31         ApplyPCA(tsDB, n_components=n_components, downsample=1)
32         ApplyPCAmod(tsDB)
33         ApplyFFT(tsDB, n_components=n_components)
34         ApplyLLE(tsDB, n_components=n_components, downsample=1)
35         ApplyIsomap(tsDB, n_components=n_components, downsample=1)
36     ApplyMean(tsDB)
37     ApplyStd(tsDB)
38     ApplyPCA_Breakpoints(tsDB, n_components=1)
```

```
39      ApplyBase2_Breakpoints(tsDB)
40      return n_components
41
42  class GrabY(object):
43      def __init__(self, tsDB, name):
44          self.tsDB = tsDB
45          self.name = name
46      def __enter__(self):
47  #          print 'enter'
48          df = self.tsDB.get_y()
49          self.run_ids = df.xs('run_id', axis=1).values
50          self.y = df.xs('y', axis=1).values
51          return self
52      def normalize_X(self):
53          shape = self.X.shape
54          if len(shape) < 2:
55              self.X = my_functions.normalize(self.X)
56          else:
57              X = self.X.reshape((-1,1))
58              X = my_functions.normalize(X)
59              self.X = X.reshape(shape)
60  #            self.X = map(my_functions.normalize, self.X.T)
61  #            self.X = np.array(self.X).T
62      def __exit__(self, type_, value, tb):
63          if np.any([y == 0 for y in self.X.shape]):
64              self.X = np.zeros((len(self.run_ids)))
65          self.normalize_X()
66          self.tsDB.add_val_list(tab='db_attr',
67                                 run_ids=self.run_ids,
68                                 name=self.name,
69                                 vals=self.X)
70
71  def assign_val(db, run_ids, vals, name):
72      print 'assign'
73      assert len(run_ids) == len(vals)
74      for run_id, v in izip(run_ids, vals):
75          db.add_val(tab='db_attr', run_id=run_id, name=name, val=v)
76
77  def ApplyPCA(tsDB, n_components=1, downsample=2):
78      with GrabY(tsDB, ApplyPCA.func_name) as g:
79          # Downsample data
80          data_small = g.y[::, ::downsample]
81          # PCA
82          fit_pca = PCA(n_components=n_components)
83          g.X = fit_pca.fit_transform(data_small)
84
85  def ApplyPCAmod(tsDB):
86      with GrabY(tsDB, ApplyPCAmod.func_name) as g:
87          fit_pca = PCA(n_components=2)
88          X = fit_pca.fit_transform(g.y)
89          temp = np.arange(len(X)).reshape((-1,1))
90          X = np.hstack((temp,X))
91          X = X[np.argsort(X[:,1]),:]
92          new_order = np.hstack((X,temp))
```

```
93          new_order = new_order[np.argsort(new_order[:,0]),:]
94          g.X = new_order[:,3]
95
96  def ApplyMean(tsDB):
97      with GrabY(tsDB, ApplyMean.func_name) as g:
98          g.X = np.mean(g.y, axis=1)
99
100 def ApplyStd(tsDB):
101     with GrabY(tsDB, ApplyStd.func_name) as g:
102         g.X = np.std(g.y, axis=1)
103
104 def ApplyFFT(tsDB, n_components=3):
105     with GrabY(tsDB, ApplyFFT.func_name) as g:
106         n_comp = 2 * n_components - 1
107         X = np.fft.fft(g.y, n=n_components, axis=1)
108         g.X = np.real(X[:, ::2])
109
110 def ApplyLLE(tsDB, n_components=1, downsample=2):
111     with GrabY(tsDB, ApplyLLE.func_name) as g:
112         # Downsample data
113         data_small = g.y[::, ::downsample]
114         # LLE standard
115         lle = LocallyLinearEmbedding(n_components=n_components,
116                                      method='standard')
117         g.X = lle.fit_transform(data_small)
118
119 def ApplyIsomap(tsDB, n_components=1, downsample=2):
120     with GrabY(tsDB, ApplyIsomap.func_name) as g:
121         # Downsample data
122         data_small = g.y[::, ::downsample]
123         # Isomap
124         iso = Isomap(n_components=n_components)
125         g.X = iso.fit_transform(data_small)
126
127 # Make different breakpoint representations
128 def breakpoint_formatting(bp, style='short'):
129     ''' bp is a list of pairs of segment beginning and
130         ending indices. This outputs the breakpoints in the short
131         and long format. Short is just the beginning locations,
132         padded with zeroes. Long is in 0,1 representation '''
133     t_len = np.max(bp[0]) + 1
134     st = []
135     break_points_long = np.zeros((len(bp), t_len), dtype=np.int)
136     for i, row in enumerate(bp):
137         st_ = np.zeros(len(row), dtype=np.int)
138         for i2, pair in enumerate(row):
139             st_[i2] = pair[0]
140         st.append(st_)
141         break_points_long[i, st_[1:]] = 1
142     if style == 'short' or style == 'both':
143         max_L = np.max([len(x) for x in st])
144         break_points_short = np.zeros((len(bp), max_L))
145         for i, row in enumerate(st):
146             x = max_L - len(row)
```

```
147             break_points_short[i, x:] = row
148         if style == 'both':
149             return break_points_short, break_points_long
150 #             return break_points_short[:, 1:], break_points_long
151         else:
152             return break_points_short
153 #             return break_points_short[:, 1:]
154     else:
155         return break_points_long
156 
157 class GrabBreakpoints(GrabY):
158     def __enter__(self):
159 #         print 'enter'
160         segs = self.tsDB.get_val_all('db_attr', 'segs',
161                                      out_style='list')
162         self.run_ids = [row[0] for row in segs]
163         self.segs = [row[1] for row in segs]
164         self.break_points_short, self.break_points_long = \
165             breakpoint_formatting(self.segs, style='both')
166         return self
167 
168 def ApplyPCA_Breakpoints(tsDB, n_components=1):
169     with GrabBreakpoints(tsDB, ApplyPCA_Breakpoints.func_name) as g:
170         if n_components >= g.break_points_short.shape[1]:
171             n_components = g.break_points_short.shape[1] - 1
172         if n_components < 1:
173             n_components = 1
174         fit_pca = PCA(n_components=n_components)
175         g.X = fit_pca.fit_transform(g.break_points_short)
176 
177 def sum_base2(x):
178     ''' Given a list of 1 and 0, it produces a base 2 sum
179         where the location of the 1 is the power, so 2^i.
180         This is processed from left to right '''
181     myOut = 2.
182     for i, idx in enumerate(x):
183         if idx == 1:
184             myOut += 2 ** i
185     return myOut
186 
187 def breakpoint_metric_sum_base2(bp):
188     ''' Calculates a metric based on breakpoint locations
189         bp is in the long format '''
190     l_sum = np.zeros(len(bp))
191     r_sum = np.zeros(len(bp))
192     for i, row in enumerate(bp):
193         l_sum[i] = np.float64(sum_base2(row))
194         r_sum[i] = np.float64(sum_base2(row[::-1]))
195     l_sum, r_sum = map(np.log2, [l_sum, r_sum])
196     l_sum, r_sum = l_sum / np.max(l_sum), r_sum / np.max(r_sum)
197     return (l_sum - r_sum * .5)
198 
199 def ApplyBase2_Breakpoints(tsDB):
200     with GrabBreakpoints(tsDB,
```

```
                        ApplyBase2_Breakpoints.func_name) as g:
        g.X = breakpoint_metric_sum_base2(g.break_points_long)


def make_comparison_eigs(r, c, reps=100,
                         n_eigs_2_keep=10,
                         out_option='stats'):
    ''' Calculates the average eigenvalues from rxr
        correlation matrix.

        Correlation matrix is calculated from a rxc matrix,
        where each element is N(0,1).
        Also calculates the Shapiro-Wilk test for normality on each
        component

        inputs
        ------
        r - number of rows of random matrix
        c - number of columns of random matrix
        reps - number of repetitions
        n_eigs_2_keep - number of eigenvalues to keep
                        ( n_eigs_2_keep <= c)
        out_option - ['stats', 'vals']: stats gives
                                        back mean and std
                     vals gives back original values

        outputs
        -------
        myOut - (if out_option=='stats') - (n_eigs_2_keep)x2 matrix,
                1st col - mean, 2nd col - std dev
        myOut - (if out_option=='vals') - reps x n_eigs_2_keep -
                matrix of eigenvalues
        shapiro_wilk - list of tuples (Shapiro-Wilk statistic,
                        probability for hypothesis test)
    '''
    # Check if number of components to keep is >= c
    if n_eigs_2_keep > c:
        n_eigs_2_keep = c
        myMsg = "# of eigenvalues is more than # of columns"
        warnings.warn(myMsg)
    # Calculate eigenvalues
    matrix_of_eigenvalues = np.zeros((reps, n_eigs_2_keep),
                                     dtype=float)
    for i in range(reps):
        test_data = np.random.normal(0, 1., (r,c))
        R = np.corrcoef(test_data) # makes rxr matrix
        eigenvalues = np.linalg.eigvals(R)
        matrix_of_eigenvalues[i,:] = \
                        np.real(eigenvalues[:n_eigs_2_keep])
    # Format Output
    if out_option == 'stats':
        myOut = np.zeros((n_eigs_2_keep, 2), dtype=float)
        myOut[:,0] = np.mean(matrix_of_eigenvalues, axis=0)
        myOut[:,1] = np.std(matrix_of_eigenvalues, axis=0)
```

```
255     elif out_option == 'vals':
256         myOut = matrix_of_eigenvalues
257     # Perform Shapiro-Wilk test for normality
258     shapiro_wilk = []
259     for i in range(n_eigs_2_keep):
260         shapiro_wilk.append( \
261                 scipy.stats.shapiro(matrix_of_eigenvalues[:,i]) )
262 
263     return myOut, shapiro_wilk
264 
265 def determine_n_components(dat, reps=100):
266     ''' Determines the number of components to keep using the
267         parallel analysis method.
268 
269         inputs
270         ------
271         dat - (n x p) - matrix of n samples and p features
272                         (time steps)
273 
274         outputs
275         -------
276         n_components - the number of components to keep for PCA
277     '''
278     dat_ = copy.deepcopy(dat)
279     # Center and Standardize Data
280     # Var need to be in the rows when for correlation matrix
281     dat_ = dat_.T
282     dat_ -= np.mean(dat_)
283     dat_ /= np.std(dat_)
284     p, n = dat_.shape
285 #    print 'new n, p:', n, p
286 
287     # Calculate Eigenvalues of Correlation Matrix
288     R = np.corrcoef(dat_)
289     eigs = np.linalg.eigvals(R)
290 
291     # Calculate test eigenvalues (mean, std)
292     n_consider = np.min([p, 10])
293     p_significance = 0.95    # Perform one-tail significance test
294     eig_stats, shapiro_wilk_stats = make_comparison_eigs(p, n,
295                                         reps=100,
296                                         n_eigs_2_keep=n_consider,
297                                         out_option='stats')
298 
299     # Count how many principal components to keep
300     n_keep = 0
301     for i in range(n_consider):
302         av = eig_stats[i,0]
303         sd = eig_stats[i,1]
304         test_stat = (np.real(eigs[i]) - av)/sd
305         my_prob = scipy.stats.zprob(test_stat)
306         if my_prob > p_significance:
307             n_keep += 1
308         else:
```

```
309             # As soon as it hits false, break and return
310             break
311         return cl_sep
```

A.6 ApplyClustering.py

```
1  import numpy as np
2  import pandas as pd
3  import copy
4  import rpy2 as rp
5  import rpy2.robjects as robjects
6  from sklearn.decomposition import PCA
7  from sklearn.cluster import DBSCAN
8  from sklearn import cluster
9  from scipy.spatial import distance
10 from collections import Counter
11 import gc
12
13 def GrabColumns(dat, columns=None):
14     if columns is None:
15         return dat
16     if isinstance(columns, int):
17         columns = range(columns)
18     _, c0 = dat.shape
19     columns = list(set(range(c0)) & set(columns))
20     if isinstance(dat, np.ndarray):
21         return dat[:, columns]
22     elif isinstance(dat, pd.DataFrame):
23         return dat.ix[:, columns]
24
25 def GrabAttributes(tsDB, attrs, out_style='data frame'):
26     ''' This grabs the features data from the database
27
28         inputs
29         ------
30         attrs - list of attributes to be extracted from the database
31         out_style - ['data frame', 'matrix']
32
33         outputs
34         -------
35         Returns a pandas data frame or numpy ndarray of the
36             attributes.
37         If data frame, the row index are the run_ids.
38     '''
39     df = None
40     for attr in attrs:
41         # Check if attr is a list
42         if isinstance(attr, list):
43             attr_name = attr[0]
44             cols = attr[1]
45         else:
46             attr_name = attr
47             cols = False
48         # check if attribute exists
49         if attr_name in tsDB.attr_var_names:
```

```
            # Grab the attribute values
            ls = tsDB.get_val_all('db_attr', attr_name,
                                  out_style='list')
            # Separate run_ids and attribute values
            run_ids = np.zeros(len(ls))
            vals = []
            for i, (k, v) in enumerate(ls):
                run_ids[i] = k
                vals.append(v)
            # reformat vals to be column vector
            vals = np.array(vals)
            if cols:
                vals = vals[:, cols]
            if len(vals.shape) < 2:
                vals = vals.reshape((-1, 1))
        else:
            run_ids = range(tsDB.n_runs)
            vals = np.zeros((len(run_ids),1))
        # set up indices and column headers
        idx = pd.Index(run_ids, name='run_ids', dtype=int)
        if vals.shape[1] > 1:
            headers = [ '.'.join([ attr_name,
                        str(x) ]) for x in xrange(vals.shape[1])]
        else:
            headers = [attr_name]
        # create and concat as pandas data frame
        if df is None:
            df = pd.DataFrame(data=vals, index=idx, columns=headers)
        else:
            df_ = pd.DataFrame(data=vals, index=idx,
                               columns=headers)
            df = pd.concat((df, df_), axis=1)
    if out_style == 'data frame':
        return df
    elif out_style == 'matrix':
        return df.values
    else:
        return df

def _check_clean_data(dat):
    # check data
    dat = np.atleast_2d(dat)
    row = dat[0,:]
    keepcol = []
    for i, x in enumerate(row):
        if x is not None:
            if not np.isnan(x):
                keepcol.append(i)
    dat = dat[:,keepcol]
    return dat

def dbscan_wrapper(dat, **kwargs):
    # check and remove None and nan
    dat = _check_clean_data(dat)
```

```
104     # format data
105     D = distance.squareform(distance.pdist(dat))
106     S = 1 - (D / np.max(D))
107     if np.isnan(np.sum(S)):
108         return np.ones(len(dat))*-1
109 #    print np.sum(S)
110     _, labels = cluster.dbscan(S, **kwargs)
111     return labels
112 
113 def dbscan_wrapper_mod(dat, **kwargs):
114     ''' Performs a linear regression clustering after the dbscan '''
115     # Perform DBSCAN and get labels
116     dat = _convert_data_format(dat, out_format='ndarray')
117     lbls = dbscan_wrapper(dat, **kwargs)
118     lbl_set = list(set(lbls))
119 #    print 'lbl_set:', lbl_set
120 
121     lbls_3 = np.zeros(lbls.shape, dtype=np.float64) - 1
122     for i, lbl_ in enumerate(copy.copy(lbl_set)):
123         # Skip outliers
124         if lbl_ == -1: continue
125 #        run_ids = tsDB.get_matching_ids(tab='db_attr',
126 #                                        name=label_name, val=x)
127 #        temp3 = temp.ix[run_ids]
128 #        dat_3 = temp3.values
129         #print lbls
130         row_idx = np.argwhere( lbls == lbl_ ).ravel()
131         row_idx = np.array(row_idx, dtype=int)
132 #        print 'lbl_:', lbl_
133 #        print 'row_idx:', row_idx
134 #        print dat
135         dat_3 = dat[row_idx,:]
136 #        print dat_3
137         lbl_3 = lin_reg_cluster(dat_3)
138         lbl_3 = np.array(lbl_3)
139         lbl_3_set = list(set(lbl_3))
140         if len(lbl_3_set) > 1:
141 #            print 'lbl_3_set:', lbl_3_set
142             lbl_3 += np.max(lbl_set) + 1
143             lbl_3_set = list(set(lbl_3))
144             lbl_set.extend(lbl_3_set)
145 #            print 'lbl_set:', lbl_set
146         else:
147             lbl_3[:] = lbl_
148         lbls_3[row_idx] = np.array(lbl_3, dtype=np.float64)
149 #    print 'lbls_3:', set(lbls_3)
150     return lbls_3
151 
152 def CheckGrouping(tsDB, attr_names=None):
153     ''' Checks the grouping accuracy of the data
154 
155         inputs
156         ------
157         tsDB - TimeSeriesDB instance
```

```
158
159         outputs
160         -------
161         groupings - returns the group values ordered by run_id
162         check_accuracy - for each grouping, gives a 0 for right
163                          grouping
164                        and a -1 for wrong grouping
165         description
166         -----------
167         The method looks for a "right_group" attribute in the tsDB,
168         and this acts as the baseline. Then it looks for attributes
169         with "label:" in the name, which is a flag for comparison.
170         Then for each attribute it finds, it extracts the grouping
171         and compares that value against the "right_group".
172     '''
173     if attr_names is None:
174         attr_names = tsDB.attr_var_names
175         attr_names = [x for x in attr_names if 'label:' in x]
176 #    print attr_names
177
178     right_group = tsDB.get_val_all(tab='db_attr',
179                                    name='right_group',
180                                    out_style='data frame')
181     groupings = right_group.copy()
182 #    check_accuracy = right_group.copy()
183     check_accuracy = pd.DataFrame(data=np.zeros(len(groupings)),
184                                   index=groupings.index,
185                                   columns=['right_group'])
186     for attr_name in attr_names:
187         df = tsDB.get_val_all(tab='db_attr', name=attr_name,
188                               out_style='data frame')
189         groupings = pd.merge(groupings, df, on='run_id')
190         g0 = groupings.xs('right_group', axis=1).values
191         g1 = groupings.xs(attr_name, axis=1).values
192 #         print np.vstack((g0,g1))
193         grouping_check = check_set(g0, g1)
194 #         print grouping_check
195
196         df1 = pd.DataFrame(grouping_check, index=right_group.index,
197                            columns=[attr_name])
198         check_accuracy = pd.merge(check_accuracy, df1,
199                                   left_index=True,
200                                   right_index=True)
201     return groupings, check_accuracy
202
203 def check_set(s1, s2):
204     ''' This method tests the mismatch in groupings between the
205         first and second sets of values. The grouping labels between
206         the two maybe different so the method will take a guess at
207         it. It uses the first list as the "right" grouping and
208         checks the second one against it.
209
210         inputs
211         ------
```

```
212         s1, s2 - list or np.array of values
213
214         outputs
215         -------
216         accuracy_lbl - numpy array of 0 and -1. 0 means correct
217                        grouping
218                         and -1 means wrong grouping
219     '''
220     # s1 is the 'right' set so there shouldn't be a -1 in the set
221     s1 = np.array(s1)
222     s2 = np.array(s2)
223 #    set1 = set(s1)
224     counts1 = Counter(s1)
225     counts1 = np.asarray(counts1.items())
226     counts1 = counts1[np.argsort(counts1[:, 1]), :]
227     set1 = counts1[::-1,0]
228 #    print counts1
229 #    print set1
230     numbers_used = {-1:-1}
231     accuracy_lbl = np.zeros(len(s1))
232     for i in set1:
233 #        print 'numbers_used:', numbers_used
234         # Find indices where value is i in s1
235         idx = np.argwhere(s1 == i).flatten()
236 #        print 'idx:', idx
237         # Grab indices from s2
238         s2_ = s2[idx]
239         set2 = list(set(s2_))
240 #        print 's2_, set2:', s2_, set2
241         if len(set2) == 1:
242             if set2[0] in numbers_used.values():
243                 accuracy_lbl[idx] = -1
244             else:
245                 numbers_used[i] = set2[0]
246         else:
247             counts = Counter(s2[idx])
248 #            print 'counts:', counts
249             for k, v in counts.items():
250 #                print 'k, v:', k,v
251                 if k in numbers_used.values():
252                     idx2 = np.argwhere(s2 == k).flatten()
253                     idx3 = list(set(idx) & set(idx2))
254 #                    print 'idx2:', idx2
255 #                    print 'idx3:', idx3
256                     accuracy_lbl[idx3] = -1
257                     counts.pop(k)
258             counts = np.asarray(counts.items())
259 #            print 'counts2:', counts
260             if len(counts) > 1:
261 #                counts = counts[np.argsort(counts[:, 1]), :]
262                 counts = counts[np.argsort(counts[:, 1]), :]
263 #                print 'counts3:', counts
264                 numbers_used[i] = counts[-1][0]
265                 for k in counts[:-1, 0]:
```

```
                    idx2 = np.argwhere(s2 == k).flatten()
                    idx3 = list(set(idx) & set(idx2))
                    accuracy_lbl[idx3] = -1
            elif len(counts) == 1:
                numbers_used[i] = counts[0][0]
            else:
                pass
    return accuracy_lbl


def bottom_up_aggr_multivariate(X, sigma=.1):
    ''' A bottom up aggregation using multivariate linear
        regression. X is a (n x m) matrix, where n is the number of
        points, and m is the number of variables. So for a 3
        dimensional line, where coordinates in cartesian are
        [x,y,z], then X is (n x 3)
    '''
    L1, W1 = X.shape
    L2 = int(np.floor(L1/2.))
    Y = np.array(X)
    Z = np.vstack(( np.ones(L1), range(L1) )).T

    def calc_stddev_of_resid(Z, Y):
        Z = np.array(Z)
        Y = np.array(Y)
        fit = np.linalg.lstsq(Z, Y)
        resid = np.dot(Z, fit[0]) - Y
        return np.std(resid)

    sd = calc_stddev_of_resid(Z, Y)
#    print sd
    if sd < sigma:
        return [Z], [Y]

    ### Pair Up Points ###
    z2, y2 = [], []
    for i in range(L2):
        z2.append( Z[2*i:2*i+2, :] )
        y2.append( Y[2*i:2*i+2, :] )
    # Check to see if sequence is odd
    if np.mod(L1,2) == 1:
        z2.append([Z[-1,:]])
        y2.append([Y[-1,:]])

    ### Calculate Linear Regression of adj. sets
    ### Calculate std. dev.
    def calc_local_sd(X, Y, idx):
        x = np.array( X[idx] )
        x = np.vstack( (x, np.array(X[idx+1])) )
        y = np.array( Y[idx] )
        y = np.vstack( (y, np.array(Y[idx+1])) )
        return calc_stddev_of_resid(x, y)

    y3sd = []
```

```
320     for i in xrange(len(y2)-1):
321         y3sd.append( calc_local_sd(z2, y2, i) )
322 
323     myMins = [0]
324 
325     def merge_and_pop(vec, idx):
326         v = copy.copy(vec)
327         v1 = v.pop(idx)
328         v2 = v.pop(idx)
329         v3 = np.vstack(( v1, v2))
330         v.insert(idx, v3)
331         return v
332 
333     while min(y3sd) < sigma:
334         myMins.append(np.min(y3sd))
335         myIndex = y3sd.index(myMins[-1])
336 
337         y2 = merge_and_pop(y2, myIndex)
338         z2 = merge_and_pop(z2, myIndex)
339         y3sd.pop(myIndex)
340 
341         if myIndex == len(y2)-1:
342             i3 = [-1]
343         elif myIndex == 0:
344             i3 = [0]
345         else:
346             i3 = [-1,0]
347         for i in i3:
348             y3sd[myIndex+i] = calc_local_sd(z2, y2, myIndex+i)
349 
350         if len(y2) < 3:
351             break
352     return z2, y2
353 
354 
355 def lin_reg_cluster(dat, is_sorted=False):
356     ''' Clusters points in a line using a modified time series
357         segmentation algorithm. It's assumed that the data is
358         organized as a line in n-dimensional space.
359 
360         Inputs
361         ------
362         dat - (n x m) matrix where n is the number of samples and
363                 m is number of features
364         is_sorted - if False, the data is sorted using the first PC
365                     of PCA
366 
367         Outputs
368         -------
369         labels - (n) list of labels
370     '''
371 
372     dat = _check_clean_data(dat)
373 
```

```
374     # Sort the data using PCA
375     if not is_sorted:
376         pca = PCA(n_components=1)
377         result = pca.fit_transform(dat)
378 
379         mat1 = np.hstack(( np.arange(len(result)).reshape((-1,1)),
380                            result.reshape((-1,1)) ))
381         mat2 = np.hstack(( mat1, dat ))
382         mat3 = mat2[np.argsort(mat1[:,1]),:]
383         mat_ordered = mat3[:,2:]
384     else:
385         mat_ordered = dat
386 
387     Z, Y = bottom_up_aggr_multivariate(mat_ordered, sigma=.01)
388     labels = np.zeros(len(dat), dtype=np.float64)
389     idx = 0
390 
391     for i, z in enumerate(Z):
392         idx2 = idx + len(z)
393         labels[idx:idx2] =  i
394         idx = idx2
395 
396     if not is_sorted:
397         mat4 = np.hstack(( mat3[:,[0]], labels.reshape((-1,1)) ))
398         mat4 = mat4[np.argsort(mat4[:,0]), :]
399         return mat4[:,1]
400     else:
401         return labels
402 
403 def _convert_data_format(dat, out_format='ndarray'):
404     ''' Converts data into desired format
405 
406         inputs
407         ------
408         dat - data, accepts numpy ndarray and pandas data frame
409         out_format - ['ndarray', 'data frame', 'list']
410 
411         outputs
412         -------
413         returns data as specified by out_format
414     '''
415     columns = None
416     idx = None
417     if isinstance(dat, np.ndarray):
418         mat = dat
419     elif isinstance(dat, pd.DataFrame):
420         mat = dat.values
421         columns = dat.columns
422         idx = dat.index
423     else:
424         raise TypeError('dat is not right format')
425 
426     if out_format == 'ndarray':
427         return mat
```

```
428     if out_format == 'data frame':
429         if not columns:
430             col_name = [ ''.join(('x', str(i))) for i in \
431                         range(dat.shape[1])]
432             return pd.DataFrame(mat, columns=col_name)
433         else:
434             return pd.DataFrame(mat, index=idx, columns=columns)
435     if out_format == 'list':
436         return [x.tolist() for x in mat]
437     else:
438         raise TypeError('out_format is not recognized format')
439
440
441 def model_based_clustering(dat):
442     ''' Wrapper for the Mclust method from mclust package in R
443         Performs model-basee clustering
444     '''
445     dat_ = _convert_data_format(dat, out_format='ndarray')
446     dat_ = _check_clean_data(dat_)
447     if dat_.size < 1:
448         return np.ones(len(dat))*-1
449     df = _convert_data_format(dat_, out_format='data frame')
450 #    print '[model_based_clustering] df:', df
451     R_df = pd.rpy.common.convert_to_r_dataframe(df)
452     R = rp.robjects.r
453     R('library(mclust)')
454     rp.robjects.globalenv['R_df'] = R_df
455     if dat_.shape[1] > 1:
456         R('mclust <- Mclust( R_df, modelNames=c("VVV") )' )
457 #        R('mclust <- Mclust( R_df, modelNames=c("VVV","VEV") )' )
458 #        R('mclust <- Mclust( R_df )' )
459     else:
460         R('mclust <- Mclust( R_df )' )
461 #        R('mclust <- Mclust( R_df, modelNames=c("V") )' )
462     print R('mclust$modelName')
463     lbls = list(R('mclust$classification'))
464     garbage_collect()
465 #    print 'done mclust'
466     return lbls
467
468 def garbage_collect():
469     gc.collect()
470     R = robjects.r
471     R.gc()
472     gc.collect()
```

REFERENCES

[1] Bandte, O., *A probabilistic multi-criteria decision making technique for conceptual and preliminary aerospace systems design*. PhD thesis, Georgia Institute of Technology, 2000.

[2] Basilevsky, A., *Statistical factor analysis and related methods: theory and applications*. Wiley series in probability and mathematical statistics, Wiley-Interscience, 1994.

[3] Beaney, M., "Analysis," in *The Stanford Encyclopedia of Philosophy* (Zalta, E. N., ed.), Stanford University, winter 2012 ed., 2012.

[4] Beisecker, E. K., *Framework for robust design: a forecast environment using intelligent discrete event simulation*. PhD thesis, Georgia Institute of Technology, 2012.

[5] Bellman, R., *Adaptive Control Processes*. Princeton University Press, 1961.

[6] Berkhin, P., "Survey of clustering data mining techniques," in *Grouping Multidimensional Data: Recent Advances in Clustering* (Kogan, J., Nicholas, C. K., and Teboulle, M., eds.), Taylor & Francis US, 2006.

[7] Biltgen, P. T., *A methodology for capability-based technology evaluation for systems-of-systems*. PhD thesis, Georgia Institute of Technology, 2007.

[8] Boyle, E., "LCOM explained," tech. rep., Air Force Human Resources Lab Brooks AFB TX, 1990.

[9] Breiman, L. and Meisel, W. S., "General estimates of the intrinsic variability of data in nonlinear regression models," *Journal of the American Statistical Association*, vol. 71, no. 354, pp. pp. 301–307, 1976.

[10] Breiman, L., Friedman, J. H., Olshen, R. A., and Stone, C. J., *Classification and Regression Trees*. Belmot, CA: Wadsworth & Brooks/Cole Advanced Books & Software, 1984.

[11] Brockwell, P. J. and Davis, R. A., *Introduction to time series and forecasting*. Springer New York, 2002.

[12] Butler, A., "SAR underscores F-35 sustainment cost confusion." `http://www.aviationweek.com/Article.aspx?id=/article-xml/AW_06_03_2013_p26-582961.xml`, June 3, 2013.

[13] Chairman of the Joint Chiefs of Staff, *Manual for the Operation of the Joint Capabilities Integration and Development System (JCIDS)*, 2012.

[14] Chance, F., Robinson, J., and Fowler, J., "Supporting manufacturing with simulation: model design, development, and deployment," in *Proceedings of the 1996 Winter Simulations Conference*, pp. 114–121, IEEE, 1996.

[15] Cheong, S., Oh, S.-H., and Lee, S.-Y., "Support vector machines with binary tree architecture for multi-class classification," *Neural Information Processing Letters and Reviews*, vol. 2, pp. 47–51, March 2004.

[16] Chipman, H. A., George, E. I., and McCulloch, R. E., "Bayesian CART model search," *Journal of the American Statistical Association*, vol. 93, pp. 935–948, September 1998.

[17] Chipman, H. A., George, E. I., and Mcculloch, R. E., "Managing multiple models," in *Artificial Intelligence and Statistics 2001, (eds. Tommi Jaakkola, Thomas Richardson)*, Morgan Kaufmann, 2001.

[18] Chipman, H. A., George, E. I., and Mcculloch, R. E., "BART: Bayesian additive regression trees," *The Annals of Applied Statistics*, vol. 4, no. 1, pp. 266–298, 2010.

[19] Defence Acquisition University, "Aeronautical systems center (ASC) logistics composite model (LCOM)." `https://dap.dau.mil/aphome/das/Lists/Software%20Tools/DispForm.aspx?ID=53`, January 2011.

[20] Devroye, L., Gyorfi, L., Krzyzak, A., and Lugosi, G., "On the strong universal consistency of nearest neighbor regression function estimates," *The Annals of Statistics*, pp. 1371–1385, 1994.

[21] Dietz, D. C. and Jenkins, R. C., "Analysis of aircraft sortie generation with the use of a fork-join queueing network model," *Naval Research Logistics (NRL)*, vol. 44, no. 2, pp. 153–164, 1997.

[22] Eilers, P. H. C. and Marx, B. D., "Flexible smoothing with B-splines and penalties," *Statistical Science*, vol. 11, no. 2, pp. pp. 89–102, 1996.

[23] Ester, M., Kriegel, H.-P., Sander, J., and Xu, X., "A density-based algorithm for discovering clusters in large spatial databases with noise," in *Second International Conference on Knowledge Discovery and Data Mining*, KDD, 1996.

[24] Faas, P. D., "Simulation of autonomic logistics system (als) sortie generation," Master's thesis, Air Force Institute of Technology, 2003.

[25] Faas, P. D. and Miller, J. O., "Logistics: impact of an autonomic logistics system (als) on the sortie generation process," in *Proceedings of the 35th conference on Winter simulation: driving innovation*, WSC '03, pp. 1021–1025, Winter Simulation Conference, 2003.

[26] Fei, B. and Liu, J., "Binary tree of svm: a new fast multiclass training and classification algorithm," *Neural Networks, IEEE Transactions on*, vol. 17, pp. 696 –704, may 2006.

[27] Fisher, W. W., "Issues and models in maintenance systems incorporating cannibalization: A review," *INFOR*, vol. 28, no. 1, pp. 67 – 88, 1990.

[28] Fisher, W., "Markov process modelling of a maintenance system with spares, repair, cannibalization and manpower constraints," *Mathematical and Computer Modelling*, vol. 13, no. 7, pp. 119 – 125, 1990.

[29] Forrester, A. I. J., Sóbester, A., and Keane, A. J., *Engineering Design via Surrogate Modelling: A Practical Guide*, vol. 226 of *Progress in Astronautics and Aeronautics*. Wiley, 2008.

[30] Fraley, C. and Raftery, A. E., "Model-based clustering, discriminant analysis and density estimation," *Journal of the American Statistical Association*, vol. 97, pp. 611–631, 2002.

[31] Fraley, C., Raftery, A. E., Murphy, T. B., and Scrucca, L., *mclust Version 4 for R: Normal Mixture Modeling for Model-Based Clustering, Classification, and Density Estimation*, 2012.

[32] Friedman, J., "A tree-structured approach to nonparametric multiple regression," in *Smoothing Techniques for Curve Estimation* (Gasser, T. and Rosenblatt, M., eds.), vol. 757 of *Lecture Notes in Mathematics*, pp. 5–22, Springer Berlin / Heidelberg, 1979. 10.1007/BFb0098488.

[33] Friedman, J. H., "Multivariate adaptive regression splines," *The Annals of Statistics*, vol. 19, no. 1, pp. 1–67, 1991.

[34] Friedman, J. H. and Meulman, J. J., "Multiple additive regression trees with application in epidemiology," *Statistics in Medicine*, vol. 22, no. 9, pp. 1365–1381, 2003.

[35] Gallasch, G. E., Francis, B., Moon, C., and Billington, J., "Modelling personnel within a defence logistics maintenance process," in *Proceedings of the 1st international conference on Simulation tools and techniques for communications, networks and systems & workshops*, Simutools '08, (ICST, Brussels, Belgium, Belgium), pp. 18:1–18:10, ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2008.

[36] Gnanadesikan, R., *Methods for statistical data analysis of multivariate observations*. Wiley-Interscience, 1977.

[37] Goel, T., Haftka, R., Shyy, W., and Queipo, N., "Ensemble of surrogates," *Structural and Multidisciplinary Optimization*, vol. 33, pp. 199–216, 2007. 10.1007/s00158-006-0051-9.

[38] Gossard, T., Brown, N., Powers, S., and Crippen, D., "Scalable integration model for objective resource capability evaluations (sim-force)," in *Simulation Conference Proceedings, 1999 Winter*, vol. 2, pp. 1316 –1323 vol.2, 1999.

[39] Gramacy, R. B. and Lee, H. K. H., "Bayesian treed gaussian process models with an application to computer modeling," *Journal of the American Statistical Association*, vol. 103, no. 483, pp. 1119–1130, 2008.

[40] Gramacy, R. B., Lee, H. K. H., and Macready, W. G., "Parameter space exploration with gaussian process trees," in *Proceedings of the twenty-first international conference on Machine learning*, ICML '04, (New York, NY, USA), pp. 45–, ACM, 2004.

[41] Gramacy, R. B. and Taddy, M., "Categorical inputs, sensitivity analysis, optimization and importance tempering with tgp version 2, an R package for treed gaussian process models," *Journal of Statistical Software*, vol. 33, no. 6, pp. 1–48, 2010.

[42] Guyon, I. and Elisseeff, A., "An introduction to variable and feature selection," *The Journal of Machine Learning Research*, vol. 3, pp. 1157–1182, 2003.

[43] Han, J., Kamber, M., and Pei, J., *Data mining: concepts and techniques*. Morgan kaufmann, 2006.

[44] Healy, K. J. and Kilgore, R. A., "Introduction to silk and java-based simulation," in *Proceedings of the 30th conference on Winter simulation*, WSC '98, (Los Alamitos, CA, USA), pp. 327–334, IEEE Computer Society Press, 1998.

[45] Hetland, M. L., *Data Mining in Time Series Databases*, vol. 57 of *Series in Machine Learning and Artificial Intelligence*, ch. A Survey of Recent Methods for Efficient Retrieval of Similar Time Sequences, pp. 23–42. World Scientific Publishing Co., 2004.

[46] Hill, R. R., Miller, J. O., and McIntyre, G. A., "Simulation analysis: applications of discrete event simulation modeling to military problems," in *Proceedings of the 33nd conference on Winter simulation*, WSC '01, (Washington, DC, USA), pp. 780–788, IEEE Computer Society, 2001.

[47] Himes, D. M., Storer, R. H., and Georgakis, C., "Determination of the number of principal components for disturbance detection and isolation," in *American Control Conference, 1994*, vol. 2, pp. 1279–1283, IEEE, 1994.

[48] Hyndman, R., "Time series data library." `http://robjhyndman.com/TSDL`, n.d. Accessed on: 2nd March 2012.

[49] Iakovidis, K., "Comparing F-16 maintenance scheduling philosophies," Master's thesis, Air Force Institute of Technology, 2005.

[50] Iwata, C. and Mavris, D., "Object-oriented discrete event simulation modeling environment for aerospace vehicle maintenance and logistics process," *Procedia Computer Science*, vol. 16, pp. 187 – 196, 2013.

[51] Jackson, D. A., "Stopping rules in principal components analysis: a comparison of heuristical and statistical approaches," *Ecology*, pp. 2204–2214, 1993.

[52] Jackson, J. E., *A user's guide to principal components*, vol. 244. John Wiley & Sons, 1991.

[53] Jolliffe, I., *Principal Components Analysis*. Springer, 2nd ed. ed., 2002.

[54] Kalman, R. E. and others, "A new approach to linear filtering and prediction problems," *Journal of basic Engineering*, vol. 82, no. 1, pp. 35–45, 1960.

[55] Keane, A. J. and Nair, P. B., *Computational Approaches for Aerospace Design: The Pursuit of Excellence*. John Wiley & Sons, Ltd, 2005.

[56] Keogh, E., Chu, S., Hart, D., and Pazzani, M., "An online algorithm for segmenting time series," in *Data Mining, 2001. ICDM 2001, Proceedings IEEE International Conference on*, pp. 289 –296, 2001.

[57] Keogh, E., Chakrabarti, K., Pazzani, M., and Mehrotra, S., "Dimensionality reduction for fast similarity search in large time series databases," *Knowledge and information Systems*, vol. 3, no. 3, pp. 263–286, 2001.

[58] Keogh, E., Chu, S., Hart, D., and Pazzani, M., "Segmenting time series: a survey and novel approach," in *Data Mining in Time Series Databases* (Last, M., Kandel, A., and Bunke, H., eds.), vol. 57 of *Series in Machine Perception and Artificial Intelligence*, ch. 1, pp. 1 – 22, World Scientific, 2004.

[59] Koch, P., *Hierarchical modeling and robust synthesis for the preliminary design of large scale complex systems*. PhD thesis, Georgia Institute of Technology, December 1997.

[60] Kriegel, H.-P., Kröger, P., and Zimek, A., "Clustering high-dimensional data: a survey on subspace clustering, pattern-based clustering, and correlation clustering," *ACM Transactions on Knowledge Discovery from Data (TKDD)*, vol. 3, no. 1, p. 1, 2009.

[61] Ku, W., Storer, R. H., and Georgakis, C., "Disturbance detection and isolation by dynamic principal component analysis," *Chemometrics and intelligent laboratory systems*, vol. 30, no. 1, pp. 179–196, 1995.

[62] Lefebvre, C. et. al., "Linux mint." `http://www.linuxmint.com/about.php`, 2013.

[63] Liao, W. T., "Clustering of time series dataa survey," *Pattern Recognition*, vol. 38, no. 11, pp. 1857–1874, 2005.

[64] Ljung, L., *System Identification: Theory for the User*. Pearson Education, 1998.

[65] Ljung, L., "Perspectives on system identification," *Annual Reviews in Control*, vol. 34, pp. 1–12, April 2010.

[66] MacKenzie, A., Miller, J., and Hill, R., "An exploration of the effects of maintenance manning on combat mission readiness utilizing agent based modeling," in *Simulation Conference (WSC), Proceedings of the 2010 Winter*, pp. 1376 –1382, dec. 2010.

[67] MacKenzie, A., Miller, J., Hill, R. R., and Chambal, S. P., "Application of agent based modelling to aircraft maintenance manning and sortie generation," *Simulation Modelling Practice and Theory*, vol. 20, no. 1, pp. 89 – 98, 2012.

[68] Malley, M., *A methodology for simulating the joint strike fighter's (JSF) prognostics and health management system*. PhD thesis, Air Force Institute of Technology, 2001. AFIT/GOR/ENS/0 IM-11.

[69] Matloff, N., "A discrete-event simulation course based on the simpy language." `http://heather.cs.ucdavis.edu/~matloff/simcourse.html`, Feb. 2008.

[70] Mavris, D. N., Kirby, M. R., and Qiu, S., "Technology impact forecasting for a high speed civil transport," *SAE Transactions*, 1998.

[71] Miller, J. O., Bauer, K. W., Faas, P., Pawling, C. R., and Sterling, S. E., "Multivariate analysis of a simulated prognostics and health management system for military aircraft maintenance," *International Journal of Logistics*, vol. 10, no. 1, pp. 1–10, 2007.

[72] Moon, K., *Self-reconfigurable ship fluid-network modeling for simulation-based design*. PhD thesis, Georgia Institute of Technology, 2010.

[73] Muckstadt, J. A., "A model for a multi-item, multi-echelon, multi-indenture inventory system," *Management Science*, vol. 20, no. 4-Part-I, pp. 472–481, 1973.

[74] Muller, K. G., "Advanced systems simulation capabilities in SimPy." Presentation, n.d.

[75] Myers, R. H. and Montgomery, D., *Response Surface Methodology: Process and Product Optimization Using Designed Experiments*. Wiley Series in Probability and Statistics, Wiley Interscience, 2nd ed., 2002.

[76] Myung, I. J., "The importance of complexity in model selection," *Journal of Mathematical Psychology*, vol. 44, no. 1, pp. 190–204, 2000.

[77] OED Online, "series, n.." http://www.oed.com/view/Entry/176458, March 2013.

[78] O'Sullivan, F., Yandell, B. S., and Raynor, William J., J., "Automatic smoothing of regression functions in generalized linear models," *Journal of the American Statistical Association*, vol. 81, no. 393, pp. 96–103, 1986.

[79] Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., and Duchesnay, E., "Scikit-learn: machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[80] Pendley, S. A., Thoele, B. A., Albrecht, T. W., Howe, J. A., Antoline, A. F., and Golden, R. D., "Aligning maintenance metrics.," *Air Force Journal of Logistics*, vol. 33, no. 1, pp. 134 – 145, 2009.

[81] Peres-Neto, P. R., Jackson, D. A., and Somers, K. M., "Giving meaningful interpretation to ordination axes: assessing loading significance in principal component analysis," *Ecology*, vol. 84, no. 9, pp. 2347–2363, 2003.

[82] Peres-Neto, P. R., Jackson, D. A., and Somers, K. M., "How many principal components? stopping rules for determining the number of non-trivial axes revisited," *Computational Statistics & Data Analysis*, vol. 49, no. 4, pp. 974–997, 2005.

[83] Phan, L. L., *A methodology for the efficient integration of transient constraints in the design of aircraft dynamic systems*. PhD thesis, Georgia Institute of Technology, 2010.

[84] Pidd, M., *Computer Simulation in Management Science*. John Wiley & Sons, 3rd ed., 1992.

[85] Press, W. H., Teukolsky, S. A., Vetterling, W. T., and Flannery, B. P., *Numerical recipes 3rd edition: The art of scientific computing*. Cambridge University Press, 2007.

[86] Python Software Foundation, "Python programming language." http://www.python.org/, n.d.

[87] Queipo, N. V., Haftka, R. T., Shyy, W., Goel, T., Vaidyanathan, R., and Tucker, P. K., "Surrogate-based analysis and optimization," *Progress in Aerospace Sciences*, vol. 41, no. 1, pp. 1 – 28, 2005.

[88] Rainey, J. C., Young, C., and Golden, R. D., eds., *Maintenance Metrics U.S. Air Force*. Air Force Logistics Management Agency, March 2009.

[89] Ratanamahatana, C. A., Lin, J., Gunopulos, D., Keogh, E., Vlachos, M., and Das, G., "Mining time series data," in *Data Mining and Knowledge Discovery Handbook* (Maimon, O. and Rokach, L., eds.), pp. 1049–1077, Springer US, 2010.

[90] Razali, N. M. and Wah, Y. B., "Power comparisons of shapiro-wilk, kolmogorov-smirnov, lilliefors and anderson-darling tests," *Journal of Statistical Modeling and Analytics*, vol. 2, no. 1, pp. 21–33, 2011.

[91] Rebulanan, R., "Simulation the joint strike fighter's (JSF) autonomic logistics system (ALS) using the JAVA programming language," Master's thesis, Air Force Institute of Technology, 2000.

[92] Robinson, S., "Discrete-event simulation: from the pioneers to the present, what next?," *Operational Research Society*, vol. 56, no. 6, pp. 619–629, 2005.

[93] Rodrigues, M., Karpowicz, M., and Kang, K., "A readiness analysis for the argentine air force and the brazilian navy a-4 fleet via consolidated logistics support," in *Simulation Conference, 2000. Proceedings. Winter*, vol. 1, pp. 1068 –1074 vol.1, 2000.

[94] Rossetti, M. D. and Thomas, S., "Object-oriented multi-indenture multi-echelon spare parts supply chain simulation model," *International Journal of Modelling & Simulation*, vol. 26, no. 4, 2006.

[95] Roweis, S. T. and Saul, L. K., "Nonlinear dimensionality reduction by locally linear embedding," *Science*, vol. 290, no. 5500, pp. 2323–2326, 2000.

[96] Sadoun, B., "Applied system simulation: a review study," *Information Sciences*, vol. 124, pp. 173–192, 2000.

[97] Salman, S., Cassady, C. R., Pohl, E. A., and Ormon, S. W., "Evaluating the impact of cannibalization on fleet performance," *Quality and Reliability Engineering International*, vol. 23, no. 4, pp. 445–457, 2007.

[98] Salmon, J., Iwata, C., Mavris, D., Weston, N., and Fahringer, P., "Comparative assessment and decision support system for strategic military airlift capability," in *Selected Papers and Presentations Presented at MODSIM World 2010 Conference and Expo*, pp. 48–67, 2010.

[99] Sarma, V. V. S. and Ramchand, K., "Air fleet and facility planning via optimal control models," *Systems, Man and Cybernetics, IEEE Transactions on*, vol. 9, pp. 131 –142, march 1979.

[100] Sarma, V. V. S., Ramchand, K., and Rao, A. K., "Queuing models for estimating aircraft fleet availability," *IEEE Transactions on Reliability*, vol. R-26, pp. 253 –256, oct. 1977.

[101] SAS Institute Inc., "JMP, version 10," 1989-2013.

[102] SAS Institute Inc., "Box plots." http://www.jmp.com/academic/pdf/learning/02_box_plots.pdf, 2012.

[103] Schaefer, C. G. and Haas, D. J., "A simulation model to investigate the impact of health and usage monitoring systems (HUMS) on helicopter operations and maintenance," in *American Helicopter Society 58th Annual Forum*, American Helicopter Society International, 2002.

[104] Schrage, D. P., "Technology for rotorcraft affordability through integrated product/process development (IPPD)," in *Presented at the American Helicopter Society 55th Annual Forum*, American Helicopter Society, 1999.

[105] Shalal-Esa, A., "Pentagon, contractors aim at F-35 operating costs." http://www.reuters.com/article/2013/06/18/us-air-show-lockheed-fighter-idUSBRE95H0LU20130618, June 18, 2013.

[106] Shan, S. and Wang, G. G., "Survey of modeling and optimization strategies to solve high-dimensional design problems with computatinoally-expensive black-box functions," *Structural and Multidisciplinary Optimization*, vol. 41, pp. 219–241, 2010.

[107] Shatkay, H., "The fourier transform - a primer," tech. rep., Brown University, 1995.

[108] Sherbrooke, C., *Optimal Inventory Modeling of Systems: Multi-Echelon Techniques*. New Dimensions in Engineering, John Wiley & Sons, 1992.

[109] Sherbrooke, C. C., "An evaluator for the number of operationally ready aircraft in a multilevel supply system," *Operations Research*, vol. 19, pp. 618–635, May/June 1971.

[110] Sherbrooke, C. C., "Vari-metric: Improved approximations for multi-indenture, multi-echelon availability models," *Operations Research*, vol. 34, no. 2, pp. pp. 311–319, 1986.

[111] Simpson, T. W., Peplinski, J. D., Koch, P. N., and Allen, J. K., "Metamodels for computer-based engineering design : Survey and recommendations," *Engineering with Computers*, vol. 17, pp. 129–150, 2001.

[112] Simpson, T. W., Lin, D. K., and Chen, W., "Sampling strategies for computer experiments: design and analysis," *International Journal of Reliability and Applications*, vol. 2, no. 3, pp. 209–240, 2001.

[113] Smola, A. J. and Schlkopf, B., "A tutorial on support vector regression," *Statistics and Computing*, vol. 14, pp. 199–222, 2004. 10.1023/B:STCO.0000035301.49549.88.

[114] Subramanian, R., Scheff, R. P., Quillinan, J. D., Wiper, D. S., and Marsten, R. E., "Coldstart: fleet assignment at delta air lines," *Interfaces*, vol. 24, no. 1, pp. 104–120, 1994.

[115] Sullivan, M. J., "F-35 joint strike fighter, restructuring has improved the program, but affordability challenges and other risks remain," tech. rep., United States Government Accountability Office, 2013.

[116] Tako, A. A. and Robinson, S., "The application of discrete event simulation and system dynamics in the logistics and supply chain context," *Decision Support Systems*, vol. 52, no. 4, pp. 802–815, 2012.

[117] Tenenbaum, J. B., "Mapping a manifold of perceptual observations," *Advances in neural information processing systems*, pp. 682–688, 1998.

[118] Tenenbaum, J. B., De Silva, V., and Langford, J. C., "A global geometric framework for nonlinear dimensionality reduction," *Science*, vol. 290, no. 5500, pp. 2319–2323, 2000.

[119] Torkkola, K., "Feature extraction by non parametric mutual information maximization," *The Journal of Machine Learning Research*, vol. 3, pp. 1415–1438, 2003.

[120] Tsoutis, A., "An analysis of the joint strike fighter autonomic logistics system," Master's thesis, Naval Postgraduate School, 2006.

[121] United States Departement of Defense, Office of the Secretary of Defense, Cost Assessment and Program Evaluation Office, "O&S Cost as Percent of Total Ownership Cost (TOC)." acquired through correspondence with Bill Kobren.

[122] United States Departement of Defense, Office of the Undersecretary of Defense, "Operation and maintenance programs (o-1) revolving and management funds (rf-1)," tech. rep., US DoD, 2012.

[123] Vieira, V., "Permutation tests to estimate significances on principal components analysis," *Computational Ecology and Software*, vol. 2, no. 2, pp. 103–123, 2012.

[124] Yager, N. A., *Models for Sortie Generation with Autonomic Logistics Capabilities*. PhD thesis, Air Force Institute of Technology, 2003. AFIT/GOR/ENS/03-25.

VITA

Curtis Iwata was born in Iwakuni, Japan and grew up on U.S. military bases overseas. He received his Bachelor's degree in Mechanical and Aerospace Engineering from University of California, Irvine in 2005, Master's degree in Aerospace Engineering from Georgia Institute of Technology in 2008, and Master's in Space Management from the International Space University in Strasbourg, France in 2009. Under the guidance of Dr. Dimitri Mavris, his research at Georgia Tech ASDL focused on systems engineering decision support applications using visual analytics and computer simulation models. Curtis was also a pre-doctoral fellow of the Sam Nunn Security Program.