Chapter 1

Multivariate Statistical Process Control with `mvMonitoring`

This document serves to introduce engineers to the workflow necessary to follow in order to analyze multivariate process monitoring data via the `mvMonitoring` package.

1.1 Introduction

This is the accompanying package to the research published by [**?**] and the previous chapter. The `mvMonitoring` package is designed to make simulation of multi-state multivariate process monitoring statistics easy and straightforward, as well as streamlining the statistical process monitoring component. This package can be downloaded from GitHub by running the following:

```
devtools::install_github("gabrielodom/mvMonitoring", auth_token = "tokenHere")
```

where you create the value of "tokenHere" by generating a personal access token (PAT) at `https://github.com/settings/tokens` and copying the quoted string to this argument.

We outline this vignette as follows. In Section **??**, we discuss the motivation for creating this package, and briefly explain the utility of multi-state process monitoring. In Section **??**, we offer a thorough explination of how to generate the synthetic data we employed to test and compare the new multi-state modification of the process monitoring setup with the `mspProcessData` function. In Section **??**, we describe the necessary information and data formatting to effectively train the fault detection algorithm using the `mspTrain` function. Then, in Section **??**, we list and explain the necessary inputs to the `mspMonitor`

and `mspWarning` functions and examine their outputs. Section **??** shows a step-by-step walkthrough of how to implement the four `mvMonitoring` functions in practice. We offer some concluding remarks in Section **??**.

1.2 Motivation

1.2.1 Why use `mvMonitoring`?

The `mvMonitoring` package can be used to detect outliers in a correlated, non-Gaussian multivariate process with non-linear, non-stationary, or autocorrelated feature behavior. These process outliers are often indicative of system fault. The naive (but unfortunately common) approach to multivariate process monitoring is to use expert opinion to identify a few important features to monitor visually, and raise an alarm if these features travel outside pre-defined normal operating boundaries.

However, this split univariate approach fails to account for the correlated nature of the process features, so some engineers have taken to monitoring the system as a single correlated multivariate process rather than a collection of independent univariate processes. In the literature review of the motivating papers, the authors cite how this approach has benefited the science of process monitoring as a whole. Unfortunately, this approach has its own shortcomings. One such complication is that monitoring a multivariate process in its original feature space can lead to exorbitant computational costs. Additionally, correlated multivariate process over many features may be noisy and often retain redundant information. To combat these issues, *principal components analysis* (PCA) and its many modifications have been employed to assist in monitoring multivariate processes as a whole. This package is an implementation of one such PCA modification.

1.2.2 Multi- or Single-State AD-PCA

Adaptive-Dynamic PCA (AD-PCA), thoroughly discussed in [**?**] as well as in the previous chapter, accounts for non-linearity, non-stationarity, and autocorrelation in non-Gaussian multivariate processes. As an additional layer of complexity within this model, consider such a process with multiple known system states. *Multi-State* monitoring is thus a modification to PCA which accounts for multiple process states and models them separately. States can be any mutually exclusive blocking factor, and states do not necessarily follow a strict order. Then, Multi-State AD-PCA (MSAD-PCA) allows process engineers to account for distinct process states. Specifically, this modification should be used when features under different process states have different means, correlations, variances, or some combination of these three.

1.3 Simulating Data with `mspProcessData`

The `mspProcessData` function generates three-dimensional multi-state or single-state non-linear, non-stationary, and autocorrelated process observations. We follow the seminal work of [**?**] for generation of the foundational stationary and independent features.

1.3.1 Latent Feature Creation

So that the simulated features have non-zero correlations, Dong and McAvoy created their three features all as polynomial functions of a single latent variable t_s, where $s = 1, \ldots, \omega$ is the observational index of the process.

Autocorrelated and Non-Stationary Error

The `mspProcesData` function induces autocorrelation in t through its errors, ε_s, where

$$\varepsilon_1 \sim \mathcal{N}\left(\frac{1}{2}(a+b)(1-\varphi), \frac{b-a}{12}(1-\varphi^2)\right),$$

where $a = 0.01$ and $b = 2$. Now, we define the first-order autoregressive process on ε_s by

$$\varepsilon_s = \varphi\varepsilon_{s-1} + (1-\varphi)\varepsilon,$$

where ε is a random innovation drawn from the Normal distribution defined in the previous expression, and the autocorrelation component $\varphi = 0.75$. The mean and variance multipliers are the mean and variance of a random variable from the $\text{uniform}_{[a,b]}$ distribution.

The Non-Linear Latent Process

This t vector will be sinusoidal with period $\omega = 7 * 24 * 60$ (signifying a weekly period in minute-level observations). We then synthesize a t by taking

$$t_s^* = -\cos\left(\frac{2\pi}{\omega}s\right) + \epsilon_s,$$

and then scaling t^* to

$$t = \frac{(b-a)(t_s^* - \min(t_s^*))}{\max(t_s^*) - \min(t_s^*)} + a.$$

Finally then, the t vector will lie entirely in $[a, b]$.

1.3.2 Single-State and Multi-State Features

Single State Features

First `mspProcessData` simulates three features, with each feature operating under k different states. Let $\langle x_k(t_s), y_k(t_s), z_k(t_s)\rangle$ be the process evaluated

at t_s within State k. These are the three features under State 1 (in control, or IC) as three functions of t:

$$x(\mathbf{t}) \equiv \mathbf{t} + \boldsymbol{\varepsilon}_1, \tag{1.1}$$

$$y(\mathbf{t}) \equiv \mathbf{t}^2 - 3 * \mathbf{t} + \boldsymbol{\varepsilon}_2, \tag{1.2}$$

$$z(\mathbf{t}) \equiv -\mathbf{t}^3 + 3 * \mathbf{t}^2 + \boldsymbol{\varepsilon}_3, \tag{1.3}$$

where $\varepsilon_i \sim N(0, 0.01),\ 1 = 1, 2, 3$. The `mspProcessData` function calls the internal `processNOCdata` function to generate IC single-state observations.

Multi-State Features

The multi-state feature expression is induced by rotation and scaling of certain sets of observations. To induce a three-state, hourly switching process (the default), the `mspProcessData` function will create a label column that switches from "1" to "2" to "3" every hour. State "1" will be the features generated under the single-state assumption, while State "2" and State "3" are generated as follows. These states will be scaled rotations of the current $\langle x, y, z\rangle$ set. The second state is yaw, pitch, and roll rotated by (0, 90, 30) degrees, and the scales are multiplied by (1, 0.5, 2). The third state is yaw, pitch, and roll rotated by (90, 0, -30) degrees, and the scales are multiplied by (0.25, 0.1, 0.75). That is,

1. $\mathcal{S}_1$: $\mathbf{X}(t_s) := \langle x(t_s), y(t_s), z(t_s)\rangle$.
2. $\mathcal{S}_2$: $\mathbf{X}(t_s) := \langle x(t_s), y(t_s), z(t_s)\rangle \cdot \mathbf{P}_1\boldsymbol{\Lambda}_1$, where

$$\mathbf{P}_1 = \begin{bmatrix} 0 & 0.50 & -0.87 \\ 0 & 0.87 & 0.50 \\ 1 & 0 & 0 \end{bmatrix}$$

   is the orthogonal rotation matrix for a yaw, pitch and roll degree change of $\langle 0°, 90°, 30°\rangle$, and $\boldsymbol{\Lambda}_1 = \text{diag}(1, 0.5, 2)$ is a diagonal scaling matrix.

3. $\mathcal{S}_3$: $\mathbf{X}(t_s) := \langle x(t_s), y(t_s), z(t_s)\rangle \cdot \mathbf{P}_2\boldsymbol{\Lambda}_2$, where

$$\mathbf{P}_2 = \begin{bmatrix} 0 & 0.87 & -0.50 \\ -1 & 0 & 0 \\ 0 & 0.50 & 0.87 \end{bmatrix}$$

   is the orthogonal rotation matrix for a yaw, pitch and roll degree change of $\langle 90°, 0°, -30°\rangle$, and $\boldsymbol{\Lambda}_2 = \text{diag}(0.25, 0.1, 0.75)$ is a diagonal scaling matrix.

These rotation matrices $\mathbf{P}_1$ and $\mathbf{P}_2$ turn the states in three-dimensional space so that the states are at right angles to each other in at least one dimension, and the scaling matrices $\boldsymbol{\Lambda}_1$ and $\boldsymbol{\Lambda}_2$ inflate or deflate the process variances along each principal component. The `mspProcessData` function calls the internal function `dataStateSwitch` which splits the observations by state and applies the state-specific rotation and scaling through the internal `rotateScale3D` function.

1.3.3 Synthetic Fault Induction

Faults can be introduced to single- or multi-state data via the `mspProcessData` function. The default fault start index is 8500, or roughly 84% through the cycle of 10,080 observations. These faults are added through the internal `faultSwitch` function.

1. Fault 1A is a positive shift to all three features before state rotation: $\mathbf{X}^*(t_s) = \mathbf{X}(t_s) + 2,\ s \geq 8500$.
2. Fault 1B is a positive shift to the x feature before state rotation: $x^*(t_s) = x(t_s) + 2,\ s \geq 8500$.
3. Fault 1C is a positive shift to the x and z features in State 3 only and *after* state rotation: $x^*(t_s) = x(t_s) + 2,\ z^*(t_s) = z(t_s) + 2,\ s \geq 8500$.
4. Fault 2A is a positive drift across all the process monitoring features before state rotation: $\mathbf{X}^*(t_s) = \mathbf{X}(t_s) + (s - 8500) \times 10^{-3},\ s > 8500$.
5. Fault 2B is a positive drift across the y and z process monitoring features before state rotation: $y^*(t_s) = y(t_s) + (s - 8500) \times 10^{-3},\ z^*(t_s) = z(t_s) + (s - 8500) \times 10^{-3},\ s > 8500$.
6. Fault 2C is a negative drift in the y process monitoring feature in State 2 only and *after* state rotation: $y^*(t_s) = y(t_s) - 1.5 \times \frac{s-8500}{10080-8500}$, for $s > 8500$.
7. Fault 3A is an amplification of the underlying latent variable t for all features. The maximum latent drift of this fault will be 5 + 1: $\mathbf{X}^*(t_s) = \mathbf{X}(t_s^*),\ s > 8500$, where $t_s^* = \left[\frac{5(s-8500)}{\omega-8500} + 1\right] t_s$.
8. Fault 3B is a mutation of the underlying latent variable t for the z feature: $z^*(t_s) = z(\log t_s^*),\ s \geq 8500$. This fault will dampen the underlying latent effect for z if $t_s > 1$ and amplify this effect if $t_s < 1$.
9. Fault 3C is a polynomial mutation of the error for the y feature in State 2 only and *after* state rotation: $y^*(t_s) = y(t_s) + 2 * \mathbf{e}_3(s) - 0.25$, for $s > 8500$.

1.3.4 Putting it all together...

The `mspProcessData` function can generate weeks' worth of non-linear, non-stationary, autocorrelated, multi-state, multivariate process data useful to test new process monitoring techniques. Users can generate IC observations to measure false alarm rates, or induce one of nine pre-built faults to test detection time and consistency with repeated Monte Carlo sampling. We expect this function will generate interesting data useful enough to compare new and improved process monitoring techniques with existing methods. A detailed description of this function's inputs and outputs are shown in Section **??**.

1.4 Training with `mspTrain`

The `mspTrain` function will generate projection matrices and test statistic thresholds from training data matrices. It requires data matrices to be in the `xts` matrix format.

1.4.1 The `xts` Data Matrix

An `xts` data matrix is first and foremost a matrix, *not* a data frame. For users very familiar with data frame manipulation (with `dplyr` for instance), the slight but profound differences between manipulating matrices and data frames quickly become apparent. Because of the class requirements for matrices, *all* features must be integer or double vectors. The `mspTrain` function cannot train on non-numeric information.

The `xts` object class stands for *extendible time series* and comes from the package `xts`, which is built on the package `zoo`. The date and time information for the multivariate stochastic process (necessarily as `POSIX` objects) are stored as the row indices of `xts` matrices. We recommend the package `lubridate` for manipulating `POSIX` objects.

1.4.2 The State Vector

The state membership (or *class*) vector of the data matrix is the column of integer values corresponding to the generation state of the observation. When implementing single-state AD-PCA, this class vector will simply be a numeric column of the same value – for instance, 1. However, for MSAD-PCA, the class vector indicates from which state each observation has been drawn.

The `mspTrain` function will split the observations by the class vector, apply single-state AD-PCA to each class, then return the class-specific projection matrices and thresholds. Because of this split-apply-combine strategy, users must ensure that one or more classes are not too "rare" – that is, the class sample sizes should be sufficiently large to allow for stable covariance matrix inversion. For p features (including lags), stable covariance inversion requires a class sample size near $p^2/2$ at minimum. Because of this, attention must be given to model parsimony – we recommend against blocking observations on a factor unless that factor has sufficent observations and significantly affects the observations' mean vector or covariance matrix.

1.4.3 Adaptive and Dynamic Modeling

Because of the non-linear, non-stationary, and autocorrelated nature of some process monitoring applications, the `mspTrain` function allows users to include lags of all feature variables. Including lags of the features in the data matrix (dynamic) can significantly reduce the negative effects of modeling autocorrelated observations. Further, the `mspTrain` function includes the option to update the training window over time. Re-estimating the projection matrix

and test statistic thresholds at pre-specified time intervals (adaptive) can reduce the negative effects of non-linearity and non-stationarity when modeling the observations. The idea is to divide a non-linear and non-stationary process between some equally-spaced boundaries (every day, for instance), so that the process becomes locally linear and stationary within these boundaries. As time progresses, the oldest observations are "forgotten" and the newest observations are "learned". This causes the projection and IC thresholds to "adapt" over time.

1.4.4 Model Training

After the observations have been split by class, the `mspTrain` function will call the internal function `processMonitor`, which subsequently calls the internal function `faultFilter`. This function will calculate a linear projection matrix of the data by taking the PCA of the training data matrix. The observations will then be projected linearly into a reduced-feature subspace which preserves a chosen proportion of the energy of the training data, where the energy of a matrix is the relative sum of eigenvalues of that matrix. The default proportion is 90%. This projection is calculated by the internal `pca` function.

Furthermore, non-parametric threshold values are calculated for the two process monitoring statistics – *Squared Prediction Error* (SPE) and *Hotelling's* T^2 (T^2). These monitoring statistics are described in the motivating paper (Chapter **??**). These threshold values are calculated by the internal `threshold` function, and passed up through the function pipe to be returned by `mspTrain`. The α-level of the nonparametric threshold is controlled by the user, but defaults to 0.001.

Finally, the `mspTrain` function will remove any observation(s) which would cause an alarm from the training data set. The alarm-free observations will be returned in one `xts` matrix, while the alarmed observations will be returned by another. The alarm criteria are discussed in Section **??**. When training the process monitor, pay attention to any observations flagged as alarms. The proportion of observations flagged as faults may be higher than the α-level specified, so some tuning may be necessary. As with the `mspProcessData` function, a detailed description of the inputs and outputs from the `mspTrain` function are shown in Section **??**.

1.5 Monitor and Issue Alarms with `mspMonitor` and `mspWarning`

After training the model with `mspTrain`, the projection matrices and non-parametric monitoring statistic thresholds can be used to flag incoming observations which are potentially out of control. The `mspMonitor` function was instead designed to test a single incoming observation at a time via a script or batch file, but *can* check every observation in a test matrix, which is useful for analyzing past data. To this end, the `mspMonitor` function projects a single

observation with the class projection matrices returned by `mspTrain` and checks the observation's *SPE* and T^2 statistics against the thresholds also returned by `mspTrain`. The `mspMonitor` function will then append the monitoring statistic values and indicators to the end of the observation row. Indicator values will be returned as a "1" if the values of these statistics exceed their IC thresholds. This new appended observation will be passed to the `mspWarning` function.

The `mspWarning` function takes in an observation returned by the `mspMonitor` function and an integer value (denoted r for this discussion) dictating the number of sequential flagged observations observed before an alarm is raised. If an observation returned by the `mspMonitor` function has positive statistic indicator values for either the SPE or T^2 monitoring statistics, then the `mspWarning` function will query the most recent r observations for other flags. If all r observations are positive for anomalies, then the `mspWarning` function will issue an alarm. The number of flags necessary to trigger an alarm defaults to 5. However, this default value depends heavily on the scale of the data: for continuous observations aggregated and recorded every five seconds, the number of sequential flags necessary to trigger an alarm could be much higher, perhaps even 20 or more. In contrast, for observations aggregated to the 10-minute-scale, only three sequential flags may be necessary.

In future updates of this package, this function will also have an option to issue an alarm if a critical mass of non-sequential flags is reached in a set period of observations. This modification may be necessary if the observation level becomes more fine than the 1-minute-level. Additionally, this function will also be equipped to take in a cell phone number and service provider and issue an alarm via SMS through email. Detailed descriptions of the inputs and outputs from the `mspMonitor` and `mspWarning` functions are shown in the next section.

1.6 Example Simulation Workflow

This section provides a fully commented code walk-through for the main `msp*` functions in the `mvMonitoring` package. This package depends on a few other packages, but most of these dependencies are on commonly-used packages. The exception is `BMS`.

```
library(devtools)
library(xts)
# install.packages("BMS") # If necessary
build("C:/Users/gabriel_odom/Documents/GitHub/mvMonitoring/mvMonitoring")

## [1] "C:/Users/gabriel_odom/Documents/GitHub/mvMonitoring/mvMonitoring_0.1.1.tar.gz"

library(mvMonitoring)
```

1.6.1 Generating Synthetic Data

First begin by generating multi-state data from a fault scenario. This code will yield observations under Fault 2A, as described the Synthetic Fault Induction section. We choose the default options for the period length (7 days * 24 hours * 60 minutes = 10,080 observations), the starting index of the fault (8500 out of 10080), and the time stamp for beginning the data is 16 May of 2016 at 10:00AM (my wedding anniversary). As we can see from the `str()` function, we have an `xts` matrix with 10080 rows and four columns (the state indicator and the three features).

```
fault1A_xts <- mspProcessData(faults = "A1",
                              period = 7 * 24 * 60,
                              faultStartIndex = 8500,
                              startTime = "2015-05-16 10:00:00 CST")
str(fault1A_xts)

## An 'xts' object on 2015-05-16 10:00:00/2015-05-23 09:59:00 containing:
##   Data: num [1:10080, 1:4] 1 1 1 1 1 1 1 1 1 1 ...
##  - attr(*, "dimnames")=List of 2
##   ..$ : NULL
##   ..$ : chr [1:4] "state" "x" "y" "z"
##   Indexed by objects of class: [POSIXct,POSIXt] TZ:
##   xts Attributes:
##  NULL
```

1.6.2 Train the Fault Detection Threshold

Now that these observations are generated and stored in memory, the `mspTrain` function can train the MSAD-PCA model. The last 1620 observations (27 hours' worth) will be saved for testing. The `mspTrain` function takes in the training data set partitioned between the observation and the label column. If this function errors, make sure the label column is not included in the data matrix – this will cause a singularity in the data. The function will

1. Train on the first three days' worth of observations, as set by `trainObs`.
2. Scan the fourth day for anomalies, as set by `updateFreq`.
3. Remove any alarmed observations.
4. "Forget" the first day's observations.
5. "Learn" the non-alarmed observations from the fourth day.
6. Retrain and repeat until the end of the data matrix

Furthermore, the `Dynamic = TRUE` option means that the `mspTrain` function will include the lags specified by the `lagsIncluded` argument. Finally, the number of sequential anomalous observations necessary to raise an alarm is set at 5 by the `faultsToTriggerAlarm` argument. These last three arguments are set to their defaults.

```
train1A_xts <- fault1A_xts[1:8461,]
# This function will run in 13 seconds on the author's machine.
train1A_ls <- mspTrain(data = train1A_xts[,-1],
                       labelVector = train1A_xts[,1],
                       trainObs = 3 * 24 * 60,
                       updateFreq = 1 * 24 * 60,
                       Dynamic = TRUE,
                       lagsIncluded = 0:1,
                       faultsToTriggerAlarm = 5)
str(train1A_ls)

## List of 4
##  $ FaultChecks    :An 'xts' object on 2015-05-19 08:59:00/2015-05-22 07:00:00 containing:
##   Data: num [1:4142, 1:5] 0.28 0.201 1.208 0.919 0.418 ...
##  - attr(*, "dimnames")=List of 2
##   ..$ : NULL
##   ..$ : chr [1:5] "SPE" "SPE_Flag" "T2" "T2_Flag" ...
##   Indexed by objects of class: [POSIXct,POSIXt] TZ:
##   xts Attributes:
##  NULL
##  $ Non_Alarmed_Obs:An 'xts' object on 2015-05-19 08:59:00/2015-05-22 07:00:00 containing:
##   Data: num [1:4142, 1:7] 2 3 1 1 1 1 1 1 1 1 ...
##  - attr(*, "dimnames")=List of 2
##   ..$ : NULL
##   ..$ : chr [1:7] "state" "x" "y" "z" ...
##   Indexed by objects of class: [POSIXct,POSIXt] TZ:
##   xts Attributes:
##  NULL
##  $ Alarms         :An 'xts' object of zero-width
##  $ TrainingSpecs  :List of 3
##   ..$ 1:List of 6
##   .. ..$ SPE_threshold   : Named num 4.29
##   .. .. ..- attr(*, "names")= chr "99.9%"
##   .. ..$ T2_threshold    : Named num 62.6
##   .. .. ..- attr(*, "names")= chr "99.9%"
##   .. ..$ projectionMatrix: num [1:6, 1:2] 0.422 -0.386 0.435 0.431 -0.318 ...
##   .. ..$ LambdaInv       : num [1:2, 1:2] 0.204 0 0 1.606
##   .. ..$ muTrain         : Named num [1:6] 1.37 -2.1 2.95 1.36 -2.06 ...
##   .. .. ..- attr(*, "names")= chr [1:6] "x" "y" "z" "x.1" ...
##   .. ..$ RootPrecisTrain : num [1:6, 1:6] 2.64 0 0 0 0 ...
```

```
##   .. ..- attr(*, "class")= chr [1:2] "threshold" "pca"
##   ..$ 2:List of 6
##   .. ..$ SPE_threshold   : Named num 1.16
##   .. .. ..- attr(*, "names")= chr "99.9%"
##   .. ..$ T2_threshold    : Named num 74.2
##   .. .. ..- attr(*, "names")= chr "99.9%"
##   .. ..$ projectionMatrix: num [1:6, 1:3] -0.4957 -0.0668 0.4887 -0.5236 -0.1866 ...
##   .. ..$ LambdaInv       : num [1:3, 1:3] 0.284 0 0 0 0.712 ...
##   .. ..$ muTrain         : Named num [1:6] 2.978 -0.566 -4.508 2.95 -0.592 ...
##   .. .. ..- attr(*, "names")= chr [1:6] "x" "y" "z" "x.1" ...
##   .. ..$ RootPrecisTrain : num [1:6, 1:6] 1.08 0 0 0 0 ...
##   .. ..- attr(*, "class")= chr [1:2] "threshold" "pca"
##   ..$ 3:List of 6
##   .. ..$ SPE_threshold   : Named num 8.04
##   .. .. ..- attr(*, "names")= chr "99.9%"
##   .. ..$ T2_threshold    : Named num 80.8
##   .. .. ..- attr(*, "names")= chr "99.9%"
##   .. ..$ projectionMatrix: num [1:6, 1:2] -0.392 -0.452 -0.454 0.163 -0.462 ...
##   .. ..$ LambdaInv       : num [1:2, 1:2] 0.278 0 0 0.493
##   .. ..$ muTrain         : Named num [1:6] 0.528 0.268 1.413 0.57 0.254 ...
##   .. .. ..- attr(*, "names")= chr [1:6] "x" "y" "z" "x.1" ...
##   .. ..$ RootPrecisTrain : num [1:6, 1:6] 17.4 0 0 0 0 ...
##   .. ..- attr(*, "class")= chr [1:2] "threshold" "pca"
```

The `mspTrain` function returns a list of four objects:

1. `FaultChecks`: An `xts` matrix of monitoring statistics and associated indicators for all observations after the burn-in of `trainObs`. It will have 10080 - `trainObs` number of rows and five columns:
   (a) `SPE`: the SPE statistic for each observation.
   (b) `SPE_Flag`: an indicator showing if the SPE statistic for that observation is beyond the calculated threshold; 0 is normal, 1 is flagged.
   (c) `T2`: the T^2 statistic for each observation.
   (d) `T2_Flag`: an indicator showing if the T^2 statistic for that observation is beyond the calculated threshold; 0 is normal, 1 is flagged.
   (e) `Alarm`: an indicator showing if the observation is in a sequence of flagged observations; 0 is normal, 1 is alarmed.
2. `Non_Alarmed_Obs`: An `xts` matrix will all observations with an alarm code of 0 from `FaultChecks`. Of note, this matrix contains the data, while the `FaultChecks` matrix only contains the monitoring statistics and indicators.
3. `Alarms`: An `xts` matrix of all the observations removed from the training data matrix.

4. `TrainingSpecs`: a list with length equal to the number of classes – in this case 3. For each class, this list contains a list of six objects:

   (a) `SPE_Threshold`: a named numeric scalar of the $1-\alpha$ percentile of the non-parametric estimate of the SPE statistic density.

   (b) `T2_Threshold`: a named numeric scalar of the $1-\alpha$ percentile of the non-parametric estimate of the T^2 statistic density.

   (c) `projectionMatrix`: The $p \times q$ matrix of eigenvectors necessary to project a p-dimensional observation to q dimensions. This is necessary to reduce the dimension of any test observation, and is used in calculating the SPE statistic for test observations.

   (d) `LambdaInv`: The inverse of the diagonal $q \times q$ matrix of eigenvalues. This matrix is used to calculate the T^2 statistic for test observations.

   (e) `muTrain`: The mean vector of the training observations. This is used to center the test observations on the training mean.

   (f) `RootPrecisTrain`: The $p \times p$ diagonal matrix of the inverse square roots of the feature variances. This is used to scale the test observations into the training scale.

1.6.3 Test New Observations for Anomalies

The training data summary was given by `mspTrain`, so this information can now be used to monitor incoming observations for system faults.

Adding Lagged Features

First, concatenate the last given observation from the training set as "row 0" of the test data set. This will enable `mspMonitor` to include lag-1 features. Similarly, one would include the last k observations of the training set should the process dictate the need for any lag-k features. Because the Fault Start Index was set to 8500, this testing window will show the change point between observations generated under normal conditions and those generated under a fault state.

```
test1A_xts <- fault1A_xts[8460:8520, -1]
lagTest1A_xts <- lag.xts(test1A_xts, 0:1)
lagTest1A_xts <- cbind(fault1A_xts[8461:8520,1],
                       lagTest1A_xts[-1,])
head(lagTest1A_xts)
```

```
##                     state         x         y         z       x.1
## 2015-05-22 07:00:00     1 0.5454353 -1.494528 0.9296569 0.4159290
## 2015-05-22 07:01:00     1 0.5752773 -1.638525 1.1806822 0.5454353
## 2015-05-22 07:02:00     1 0.5381703 -1.448424 1.0029127 0.5752773
## 2015-05-22 07:03:00     1 0.6097887 -1.320156 0.9345597 0.5381703
```

```
## 2015-05-22 07:04:00     1 0.8534018 -1.555631 1.0922745 0.6097887
## 2015-05-22 07:05:00     1 0.7252695 -1.792569 1.1605990 0.8534018
##                            y.1       z.1
## 2015-05-22 07:00:00  0.1063122 0.3767823
## 2015-05-22 07:01:00 -1.4945284 0.9296569
## 2015-05-22 07:02:00 -1.6385255 1.1806822
## 2015-05-22 07:03:00 -1.4484243 1.0029127
## 2015-05-22 07:04:00 -1.3201558 0.9345597
## 2015-05-22 07:05:00 -1.5556311 1.0922745
```

Monitoring the Test Data

With the lagged test observations in the working environment, the `mspMonitor` function can be applied. This function (similarly to `mspTrain`) takes in the label information as a separate argument from the input data. Further, the `mspMonitor` function takes in the `TrainingSpecs` object returned in the results list from `mspTrain`. Notice that the first six rows of the monitor matrix are the exact same as the first six rows of the lagged test matrix, except that the rows of the monitor matrix have the monitoring statistic and corresponding indicator columns appended.

```
monitor1A_xts <- mspMonitor(observations = lagTest1A_xts[,-1],
                            labelVector = lagTest1A_xts[,1],
                            trainingSummary = train1A_ls$TrainingSpecs)
head(monitor1A_xts)
```

```
##                             x         y         z       x.1        y.1
## 2015-05-22 07:00:00 0.5454353 -1.494528 0.9296569 0.4159290  0.1063122
## 2015-05-22 07:01:00 0.5752773 -1.638525 1.1806822 0.5454353 -1.4945284
## 2015-05-22 07:02:00 0.5381703 -1.448424 1.0029127 0.5752773 -1.6385255
## 2015-05-22 07:03:00 0.6097887 -1.320156 0.9345597 0.5381703 -1.4484243
## 2015-05-22 07:04:00 0.8534018 -1.555631 1.0922745 0.6097887 -1.3201558
## 2015-05-22 07:05:00 0.7252695 -1.792569 1.1605990 0.8534018 -1.5556311
##                            z.1       SPE SPE_Flag        T2 T2_Flag Alarm
## 2015-05-22 07:00:00 0.3767823 0.8309122        0 27.233955       0    NA
## 2015-05-22 07:01:00 0.9296569 0.0575549        0  4.361359       0    NA
## 2015-05-22 07:02:00 1.1806822 0.4782941        0  5.062568       0    NA
## 2015-05-22 07:03:00 1.0029127 1.0857386        0  5.471633       0    NA
## 2015-05-22 07:04:00 0.9345597 0.2959896        0  4.626291       0    NA
## 2015-05-22 07:05:00 1.0922745 0.3100297        0  2.913877       0    NA
```

1.6.4 Warn Operators During Alarms

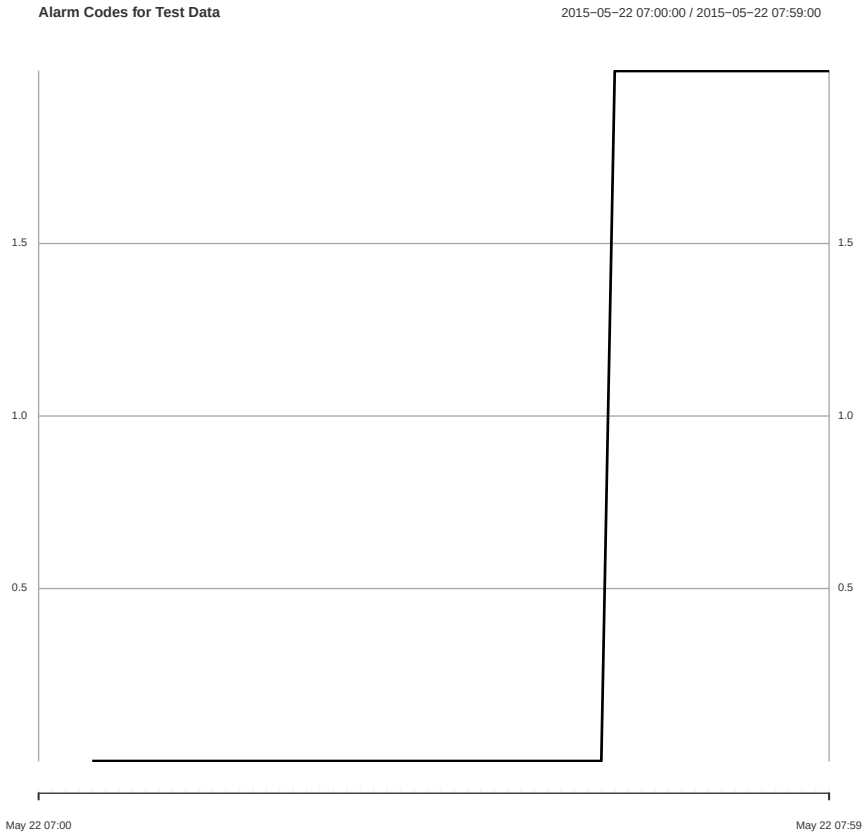
Note that all 'Alarm' codes in the monitor matrix above are recorded with `NA` values. This is because the `mspMonitor` function does not check the sequen-

tial flag conditions of the monitor matrix. This is the responsibility of the **mspWarning** function. Because the **mspWarning** function is designed to test one incoming observation at a time through a script or batch file, the following example is designed to mimic the behavior of the **mspWarning** function as each new observation comes online.

```
alarm1A_xts <- monitor1A_xts
for(i in 1:nrow(alarm1A_xts)){
  if(i < (5 + 1)){
    alarm1A_xts[1:i,] <- mspWarning(alarm1A_xts[1:i,])
  }else{
    alarm1A_xts[(i - 5):i,] <- mspWarning(alarm1A_xts[(i - 5):i,])
  }
}
```

The fault was introduced at index 8500, which corresponds to about 40 minutes into the test hour.

```
plot(alarm1A_xts[, ncol(alarm1A_xts)],
     main = "Alarm Codes for Test Data")
```



The alarm codes are

1. "0": No alarm.
2. "1": Hotelling's T^2 alarm.
3. "2": Squared Prediction Error alarm.
4. "3": Both alarms.

As we can see, the monitoring function detects a process anomaly after 40 minutes into the test hour, and the warning function issues the corresponding alarms.

1.7 Conclusion

We have supplied our motivation for this package, and we have discussed implementing a multivariate process monitoring scheme with this package using

the example of a decentralized WWT plant in Golden, CO. We believe that this software will provide system engineers with the tools necessary to quickly and accurately detect abnormalities in multivariate, autocorrelated, non-stationary, non-linear, and multi-state water treatment systems. Further, we have given a synthetic example showing how the functions within this package would be implemented and tuned in practice.