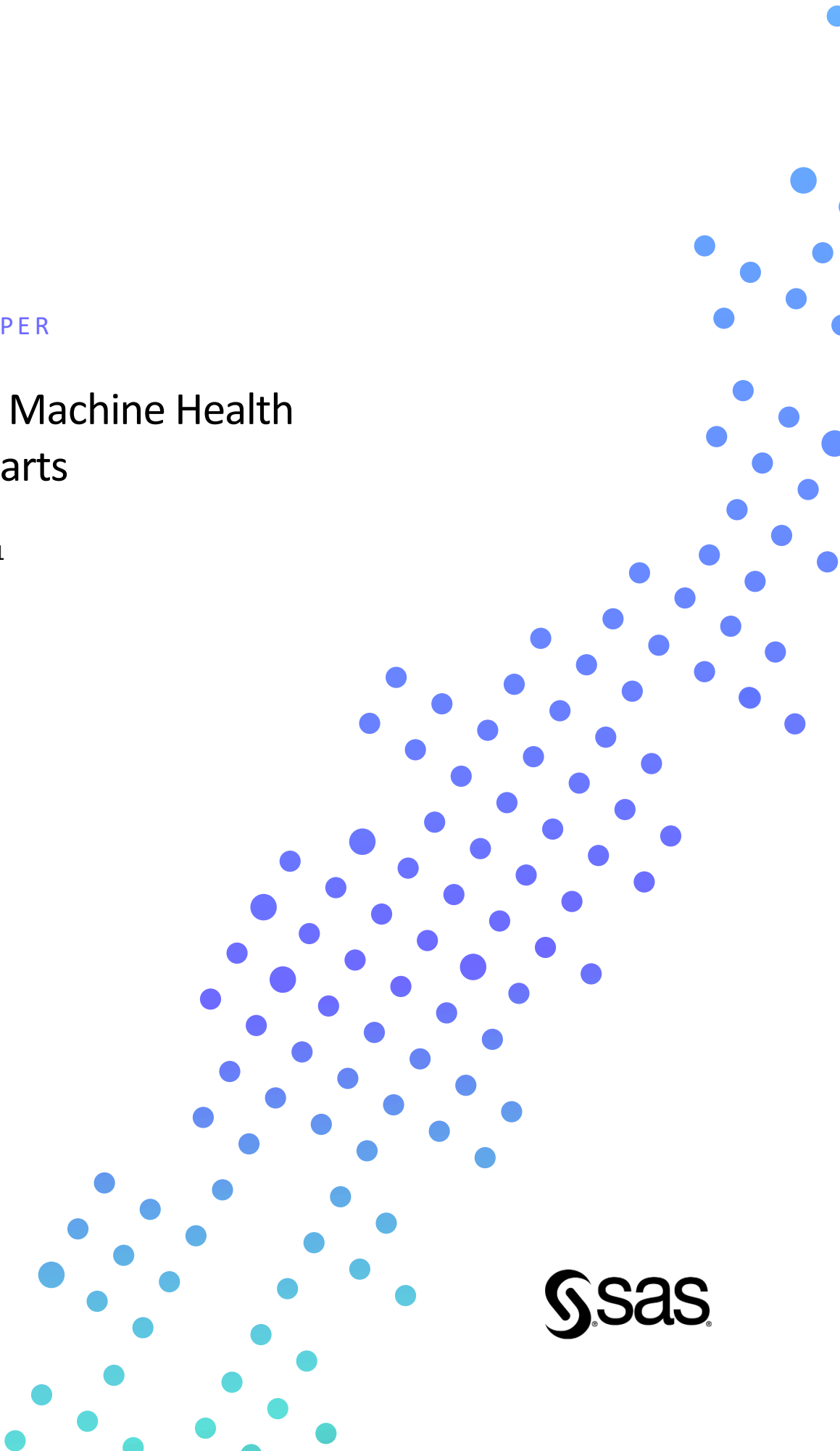
TECHNICAL PAPER

# Monitoring Machine Health Using $K_T$ Charts

Last update: June 2021



# Contents



# Relevant Product and Release

- SAS® Visual Forecasting
  - 2020.1.4

# Introduction

## Predictive Maintenance with IIoT

The industrial Internet of Things (IIoT) refers to the application of IoT technology to industrial assets. The technology involves sets of sensors, communication devices, and computing capabilities installed on a high-value industrial asset. The asset can be a wind turbine, an aircraft engine, or a manufacturing plant, to name just a few. The two main application areas of IIoT are predictive maintenance and predictive quality. Predictive maintenance refers to using IIoT technology to perform the proper maintenance at the proper time, based on the condition of the equipment. The equipment condition, or health, is assessed by the computing devices on the asset, using the sensor measurements. An early indication of equipment health degradation is helpful, because it can provide a window of opportunity to perform required actions. In the case of predictive quality applications, sensors measure important manufacturing process parameters, either controllable or uncontrollable, and then the data that they collect is used to identify a stable process. Deviations from a stable process can help engineers identify the presence of a process disturbance, which can be a precursor to poor quality. In discrete manufacturing, sensor data can also be used to develop a predictive model to assess the quality of the end product.

All these IIoT use cases have certain commonalities. One is the presence of multiple sensors, each collecting a different process measurement. For example, a chemical process can have sensors that measure temperature, pressure, flow rate, and so on. For an industrial asset, in health monitoring applications, sensors can collect data such as vibration measurements from rotating parts, oil temperature, and power consumption. Another commonality is that data from normal operations are available in abundance. This is because any mature manufacturing process or equipment generally operates as expected, and poor-quality products and degraded performance are more of an exception than a rule.

These days, the computing power of modern equipment can be exploited to perform analytical modeling to assess the health of the equipment. Traditionally, many types of equipment had some capability to monitor performance parameters against a reference value, usually one parameter at a time. With the advent of computing capability and advances in analytical techniques, with respect to both accuracy and reduction memory footprint, more sophisticated analysis can now be performed to monitor the process or health parameters and can often generate useful predictions.

## Monitoring with $K_T$ Chart Procedures in SAS Forecast Studio

This paper introduces new functionality available in SAS® Forecast Studio, known as $K_T$ chart procedures, which can be useful for monitoring process or equipment data and which has applications in both predictive maintenance and predictive quality. In SAS Forecast Studio, the $K_T$ chart procedures, namely the KTTTRAIN (KTTRAIN Procedure, 2021) and KTMONITOR (KTMONITOR Procedure, 2021) procedures, implement $K_T$ charts as discussed in Kakde, Peredriy, and Chaudhuri (2017). Before delving into the details of these SAS procedures, we will first explain what $K_T$ charts are.

## Implementation of $K_T$ Charts

$K_T$ charts are useful for monitoring the central tendency and spread of multivariate data that are generated from a device or process. Two charts together constitute the $K_T$ chart family. One chart, called an *a chart*, is useful for

monitoring the central tendency. The other chart, an $R^2$ *chart,* is useful for monitoring the spread of multivariate data. In process monitoring, the statistics that are related to central tendency and spread are both important. An engineer responsible for monitoring the process wants to have minimal variation around the expected position of the process center, or process target value, and at the same time to keep the overall spread of the process data to a minimum. Further, some variation in the process center as well as process spread is inherently present in any manufacturing process; Montgomery (2019) refers to this as inherent variation, and it is impossible to eliminate. In control chart methodology, inherent or chance variation is acknowledged, as are attempts to flag any variation that is greater than the expected variation. With this in mind, the control charts are implemented in two phases. In phase 1, analysis is performed to quantify the inherent variation. In control chart terminology, this variation is a characteristic of a process in the state of statistical control and is devoid of any special or assignable causes of variation. In phase 2, the process data are measured, and the variation and central tendency are compared to the expected values that are developed during phase 1. Significant deviation from expected natural process variation can be a cause for concern, and it is prudent to investigate the cause of such deviation. Phase 1 analysis can be performed using PROC KTTRAIN, and PROC KTMONITOR can be used for phase 2 analysis.

$K_T$ charts incorporate the concept of a window to compute the central tendency and spread. A window consists of a set of observations that are contiguous in time. For each window, a measure of central tendency and a measure of spread are computed. Using $K_T$ charts, you can monitor these two measures over time. The window approach is advantageous in IIoT use cases where data are generated at a very high speed. A good window size is one in which the sensor data variation within the window is minimal but any process changes can be captured across windows. Windows can be overlapped, in order to capture process changes early, but too much overlap can produce too many false alarms.

The foundation of $K_T$ charts is a machine learning technique known as support vector data description, or SVDD (Tax & Duin, 2004). SVDD is a one-class classification technique, and it is useful for detecting deviations from data that belong to a single class. In the context of IIoT, these data are the multivariate sensor data that are generated from normal or stable operations of the machine or manufacturing process. Because machines doing any nontrivial work are generally reliable and seldom fail, these data are abundant. SVDD builds a geometric description of these data and provides a method to identify whether any new observation is similar to the normal data. SVDD does not make distributional assumptions, so it can model a variety of data shapes. Additionally, SVDD has been observed to effectively address many peculiarities of sensor data, such as the presence of multiple clusters. When SVDD is trained, it provides two important statistics: the center $a^*$ and the threshold $R^2$. The center provides a measure of central tendency of the training data. The $R^2$, or radius, measures the spread of the data. If data variation is low, $R^2$ is small, and vice versa.

$K_T$ charts perform SVDD training for each data window and compute the center $a$ and the threshold $R^2$. These two statistics represent the central tendency and spread of the current window observations. The same computations are done for the next window and all subsequent windows. These centers and $R^2$ values are then monitored using an $a$ chart and an $R^2$ chart. During phase 1, the expected variation in centers and the $R^2$ value are determined. During phase 2, the same statistics for new windows are compared to what was observed in phase 1. Any significant deviation is flagged because it indicates a change in the process from the expected behavior and warrants further investigation.

## Audience for this Paper

The audience for this paper is data scientists and engineers who work in the area of predictive quality and prognostics and health management (PHM). The methodology outlined in this paper can be used to analyze

sensor data and to get early warnings about equipment faults and poor product quality.

# Overview of $K_T$ Chart Monitoring

The $K_T$ chart monitoring method is intended to provide the means to monitor high-frequency multivariate data. The method is based on the SVDD (Tax & Duin, 2004) algorithm that is applied to a moving window of observations. This approach enables you to reduce noise that can be present in high-frequency data and eliminates the need to monitor individual observations. $K_T$ chart monitoring has two steps:

1. Training, which is implemented in the KTTRAIN procedure. The training process uses the data from normal operations to define the state of statistical control—that is, to determine the central tendency and the spread of the process. The $K_T$ chart training calculates the control limits for allowed deviations in both the central tendency and the spread. The $K_T$ chart for monitoring the process center is called the *a chart*, and the $K_T$ chart for monitoring the process variation is called the $R^2$ *chart*.

2. Monitoring, which is implemented in the KTMONITOR procedure. The monitoring process uses the data from the ongoing process to monitor the process for stability in both the central tendency and the spread by using the control limits that are calculated in the training step.

For $K_T$ chart training, you specify a window size $n$—that is, the number of consecutive observations from the input data to be processed by SVDD at a time. If $w_i$ is an observation in the $i$th window, then $w_1 = 1, \ldots, n$, $w_2 = n+1, \ldots, 2n$, and so on. You can also specify an optional window overlap $m$, where $0 \le m < n$. In this case, $w_1 = 1, \ldots, n$, $w_2 = n+1-m, \ldots, 2n-m$, and so on. The number of windows is $k = \lfloor p/(n-m) \rfloor$, where $p$ is the number of observations in the data. The same window size and overlap are used later in the monitoring by PROC KTMONITOR.

For each window $i$, where $i = 1, \ldots, k$, the $K_T$ chart training process runs SVDD training and obtains the threshold $R_i^2$ and the center $a_i$. This process is referred to throughout this paper as *window SVDD*. Let $A$ be the data set that contains the centers $a_i$ of all windows.

After SVDD training is run for all windows and data set $A$ is complete, the $K_T$ chart training process runs SVDD training on data set $A$ and obtains the threshold $R_a^2$ and the center $a^*$.

This process is referred to throughout the paper as *centers SVDD*, and the resulting $a^*$ is referred to as the *center of the centers*.

## $R^2$ Chart

The $K_T$ chart training process computes the mean $\overline{R^2}$ and the standard deviation $\sigma_R^2$ of the $R_i^2$ series. These two measures are used to construct the control limits for the $R^2$ chart:

$$\mathrm{UCL} = \overline{R^2} + 3\sigma_R^2$$
$$\mathrm{CL} = \overline{R^2}$$
$$\mathrm{LCL} = \overline{R^2} - 3\sigma_R^2$$

The $R^2$ chart plots the SVDD threshold $R_i^2$ for each window *i*, where *i* = 1, …, *k*. For convenience, the X axis has the time ID variable, and each point on the X axis is the beginning of the *i*th window.

The $R^2$ chart is used to monitor the change in the variability of the process. The default control limits can be modified.

### *a* Chart

The control limits of the *a* chart are derived from the threshold $R_a^2$ for centers SVDD. The *a* chart plots an SVDD distance $dist^2(a_i)$ from each window's center $a_i$ to the center of the centers $a^*$.

As with the $R^2$ chart, the X axis has the time ID variable, and each point on the X axis is the beginning of the *i*th window.

The *a* chart is used to monitor changes in the central tendency. Based on experimentation on a variety of data sets, it has been observed that the magnitude of variability of the process center is much smaller than $R_a^2$. For better scaling of graphics, the LCL of the *a* chart is set to 0.6. It has been observed that when the bandwidth is selected using the automated bandwidth selection methods, an LCL of 0.6 serves as a good value for the majority of the data sets. For these reasons, the following control limit defaults are selected in PROC KTTRAIN for the *a* chart:

$$\mathrm{UCL} = R_a^2$$
$$\mathrm{LCL} = \min\left(0.6, \min(dist^2 a_i)\right)$$

The control limits of the *a* chart can be adjusted.

## Tennessee Eastman Process Example

This example demonstrates the use of $K_T$ monitoring on the Tennessee Eastman process data. The Tennessee Eastman (TE) process is a realistic model of a typical chemical industrial process that consists of five main process units: a two-phase reactor where an exothermic reaction occurs, a separator, a stripper, a compressor, and a mixer. This is a nonlinear, open-loop, unstable process that is widely used in the academic community as a case study of multivariate statistical process control, plant-wide control, and sensor fault detection. MATLAB simulation code from Ricker (2002) was used to generate the TE process data. Data were generated for the normal operations of the process and for 20 different fault conditions[1].

Each observation consists of 41 sensor response variables, x1 through x41, and the time ID variable time. For

[1] All the code and the corresponding data in this document are available at the GitHub repository (KT Chart Sample Code).

sensor response variables, 22 variables are measured continuously (every 6 seconds on average), and 19 variables are sampled at a specified interval of either 0.1 or 0.25 hour. The time ID variable measures time in seconds. Table 1 shows the total number of observations for normal operation and for each fault condition.

***Table 1.*** *The Tennessee Eastman Data*

| Operation Mode | Number of Observations | Operation Mode | Number of Observations |
|---|---|---|---|
| Normal | 900 | Fault 11 | 372 |
| Fault 1 | 294 | Fault 12 | 427 |
| Fault 2 | 1,007 | Fault 13 | 465 |
| Fault 3 | 653 | Fault 14 | 1,729 |
| Fault 4 | 807 | Fault 15 | 1,729 |
| Fault 5 | 603 | Fault 16 | 1,124 |
| Fault 6 | 280 | Fault 17 | 1,953 |
| Fault 7 | 217 | Fault 18 | 1,741 |
| Fault 8 | 759 | Fault 19 | 1,729 |
| Fault 9 | 1,465 | Fault 20 | 1,473 |
| Fault 10 | 1,362 | | |

The following SAS DATA step creates the TE data set for normal operations (which is stored in the Work library by default):

```
data normal;
input time x1-x41;
datalines;
... more lines ...
```

To demonstrate the capabilities of the $K_T$ chart procedures, the data set was expanded to have 20 observations per second, using the following macro:

```
%macro expand(input=, output=);
   proc expand data=&input out=tmp to=second method=spline(natural);
      var x1-x41;
      id time;
```

```
    run;

    proc expand data=tmp out=&output factor=20:1 method=spline(natural);
       var x1-x41;
       id time;
    run;
%mend;
```

The following code creates the training data and loads it into a SAS® Cloud Analytic Services (CAS) session:

```
%expand(input=normal, output=mycas.train);
```

The output data table **mycas.train** has 109,681 observations for 1 hour, 31 minutes, and 24 seconds of normal operations. The macro assumes that the CAS engine libref is named mycas, but you can substitute any appropriately defined CAS engine libref.

Next, $K_T$ chart training is run using the KTTRAIN procedure:

```
ods graphics / imagemap=on;

proc kttrain data=mycas.train window=600;
input x1-x41;
timeid time;
output out = mycas.out
centers = mycas.centers
kt = mycas.outkt
scoreInfo = mycas.scoreinfo
SV = mycas.SV;
run;
```

***Table 2.*** *Control Limits for TE Data*

| KT Chart Statistics | |
|---|---|
| **Name** | **Value** |
| R-square mean | 0.8038 |
| R-square UCL | 0.9451 |
| R-square LCL | 0.6625 |
| A UCL | 0.9346 |

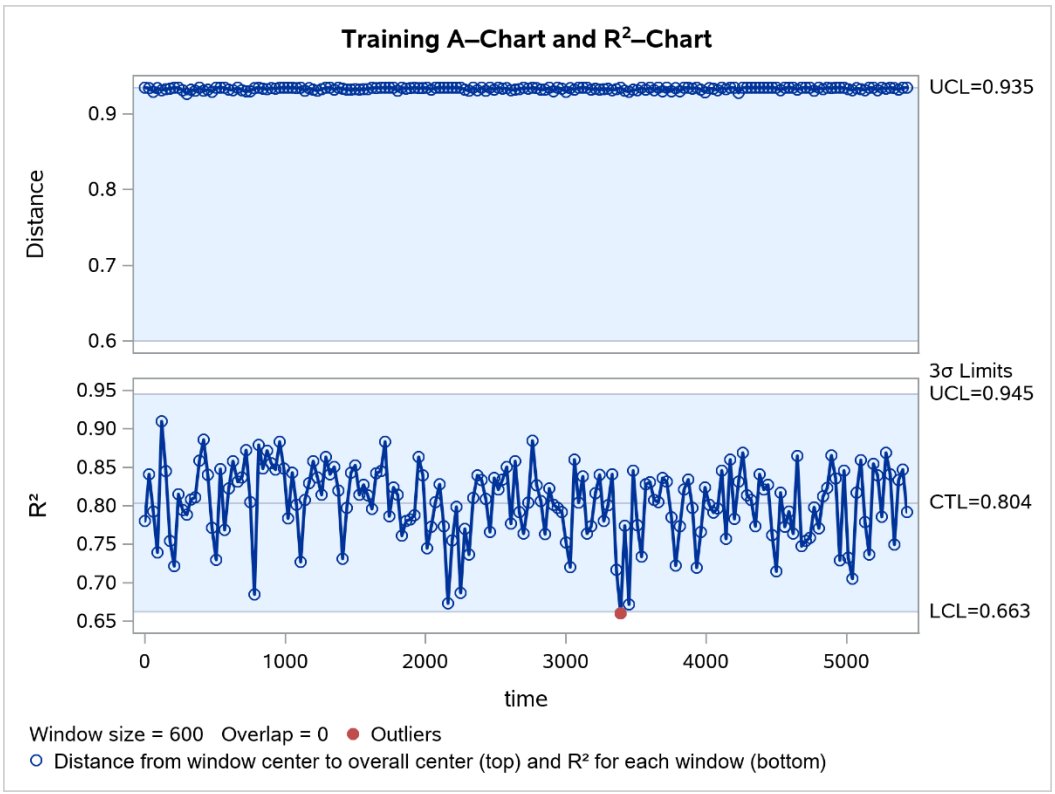
PROC KTTRAIN looks for the input data table **mycas.train**. The window size is set to 600. Because there are 20 observations per second, this means that PROC KTTRAIN generates one point for every 30 seconds on the $a$ and $R^2$ charts. Because the OVERLAP= option is omitted, there is no overlap between consecutive windows. The default control limits are used for the $a$ chart: UCL=COMPUTED and LCL=0.6. The training specifies 41 explanatory variables in the INPUT statement, x1 through x41, and time is specified as the time ID variable in the TIMEID statement. The OUTPUT statement generates the following SAS output tables: main output used by the charts,

centers of SVDD for each window, $K_T$ chart-specific output (such as control limits), scoring information, and support vectors. PROC KTTRAIN produces an ODS table that contains the chart control limits. The ODS plot of the training charts (*a* chart and $R^2$ chart) is also produced.

Control limits for the *a* chart and $R^2$ chart are shown in Table 2.

The a chart and R2 chart for the training data are shown in Figure 1. As we can see from the a chart, the center of the process shows very little variability. From the R2 chart, we can see that the process dispersion changes over time, and most of the values lie within six standard deviations of the mean window SVDD threshold $\overline{R^2}$.

***Figure 1.*** *Training Charts for TE Data*




The following code creates the monitoring data for Fault 1:

```
%expand(input=fault1, output=mycas.score);
```

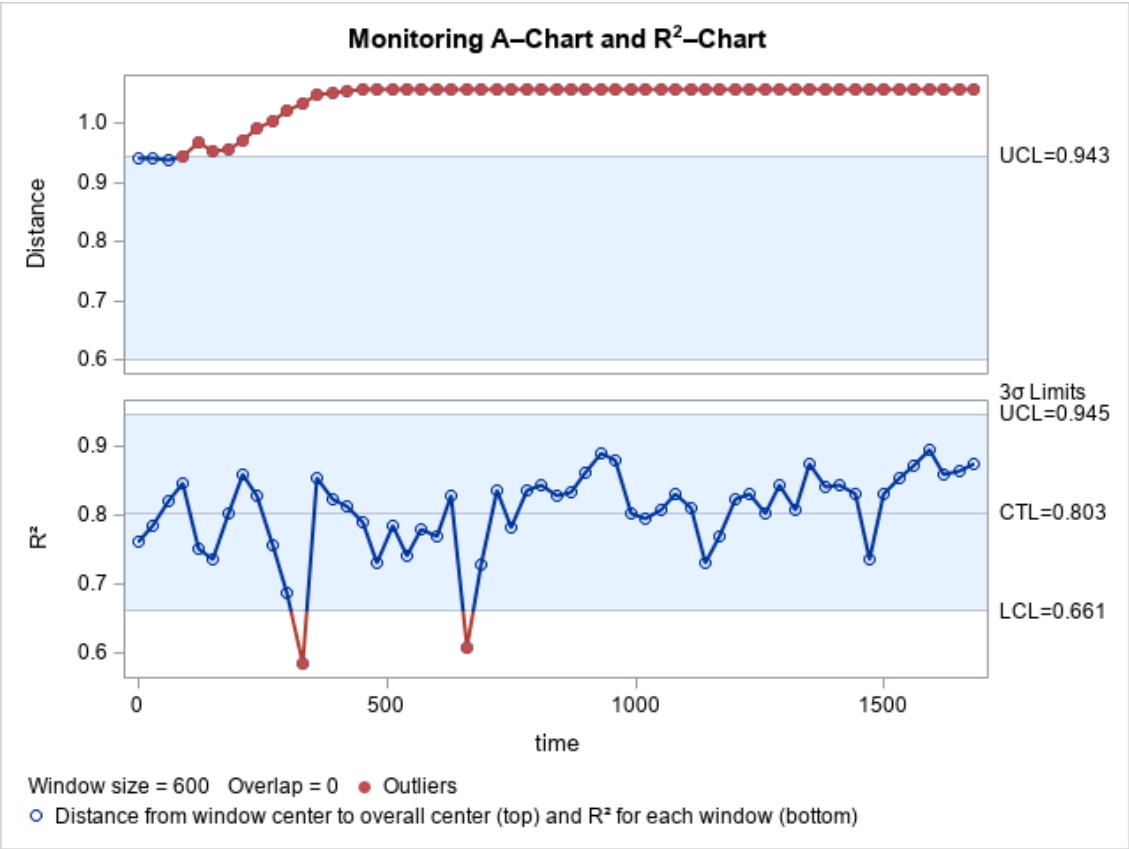
Next, monitoring is run using the KTMONITOR procedure:

```
proc ktmonitor data=mycas.score sv=mycas.SV scoreInfo = mycas.scoreinfo;
   output out = mycas.scoreout
          centers = mycas.scorecenters;
run;
```

The PROC KTMONITOR statement looks for the input data set **`mycas.score`**, the scoring information data set **`mycas.scoreinfo`**, and the data set **`mycas.sv`** with support vectors for centers SVDD. In the OUTPUT statement, the OUT= option stores the output in the **`mycas.scoreout`** data table, and the CENTERS= option stores the coordinates of the centers of each window in the **`mycas.scorecenters`** data table.

The monitoring charts are shown in Figure 2. As you can see from the a chart, which shows the distance from the center of the window SVDD to the center of the centers, the process center begins drifting from the center of the centers, and starting from the fourth window, the distance exceeds the upper control limit, indicating the fault condition.

***Figure 2.*** *Monitoring Charts for TE Data, Fault 1*




Let's consider another fault from the same data. The following code creates the monitoring data for Fault 13:

```
%expand(input=fault13, output=mycas.score);
```

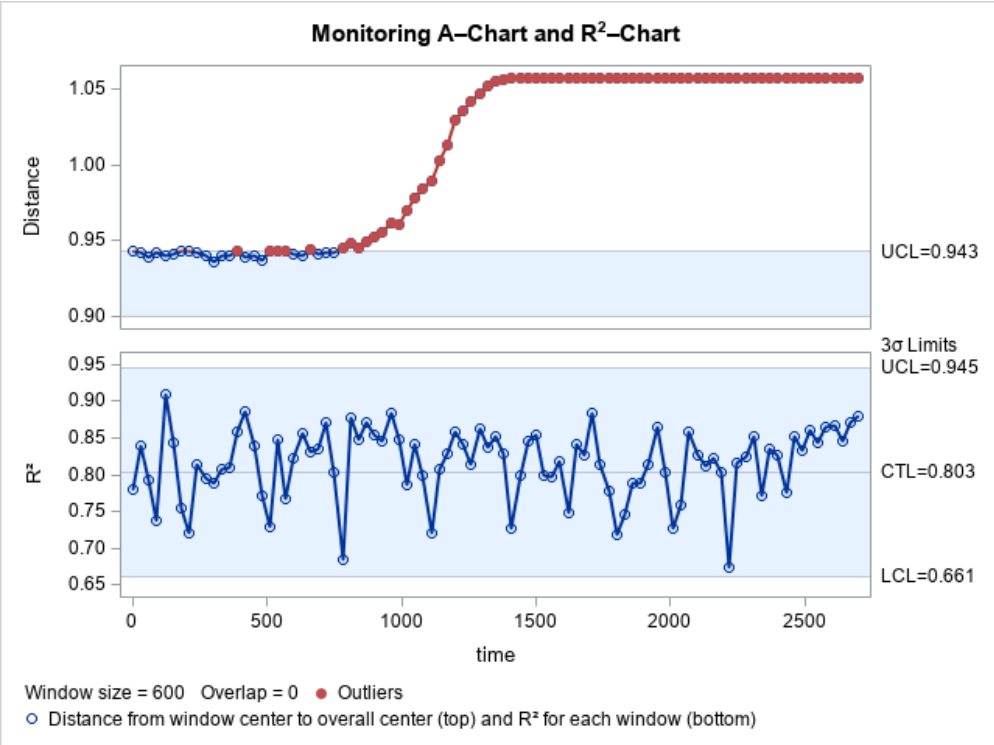
Next, monitoring is run using PROC KTMONITOR:

```
proc ktmonitor data=mycas.score
               sv=mycas.SV
               scoreInfo = mycas.scoreinfo
               a_lcl=.9;
   output out = mycas.scoreout
          centers = mycas.scorecenters;
run;
```

The only change from the previous call of the KTMONITOR procedure is that the lower limit of the *a* chart is set to 0.9 for better scaling of the graphics by specifying the A_LCL= option in the PROC KTMONITOR statement.

The monitoring charts are shown in Figure 3. As with Fault 1, the *a* chart shows the drift of the process center from the center of the centers to the outside of the upper control limit, indicating the fault condition.

***Figure 3.*** *Monitoring Charts for TE Data, Fault 13*




# Drone Data Example

This example demonstrates the use of $K_T$ monitoring on the Air Lab Fault and Anomaly (ALFA) data set. The ALFA data set is a newly published real-flight data set (Keipour, Mousaei, & Scherer, ALFA: A dataset for UAV Fault and Anomaly Detection. DOI:10.1184/R1/12707963, 2020) for unmanned aerial vehicle (UAV) anomaly detection research. It contains processed data, recorded using a fixed-wing UAV platform, of 47 autonomous flights with 23 scenarios of sudden full engine failure and 24 scenarios of seven other types of sudden control surface (actuator) faults. Each fault scenario is detailed using a flight data sequence in which ground truth of fault simulation is provided. The flight sequences in total provide 66 minutes of flight data in normal condition and 13 minutes in post-fault condition. Each flight sequence includes measurements of the following variables:

- Flight and state information, including the GPS information and wind estimation. The data are available at 4 Hz or higher.
- The measured (by sensors) and commanded (by autopilot) roll, pitch, yaw, velocity, and airspeed. The data frequency for these variables is between 20 and 25 Hz.
- The failure status variables for the following control surfaces: engine, aileron, rudder, and elevator. Fault-free condition is indicated by a value of 0 for each status variable. For engine failure, the status variable takes a value of 1 when the failure occurs; for the other three control surfaces, the value of the status variable indicates the type of fault (three types for aileron and rudder, and two types for elevator). The data frequency is around 5 Hz.

Among these, we consider 17 variables. They are datetime, a reference timestamp (in seconds) for all measurements; commanded and measured values of roll, pitch, yaw, velocity (vx, vy, vz), and airspeed (aspd); airspeed and altitude error (aspd_error, alt_error); and fault indicator. We use the timestamp of roll as the reference timestamp, and values of all other measurement variables are interpolated if necessary. For each command and measure pair, the commanded value is indicated by the suffix "_c", and the measured value is indicated by the suffix "_m". The data are provided in various formats by the original authors. We worked with binary MATLAB files (.mat) and, for each fault scenario, extracted all variables of interest into a CSV file by using the script available from Keipour (ALFA Dataset Tools, 2020).

We analyze the data for two flight sequences with simulated engine failure. The following code creates the data set **`trainscore`**, which contains the flight information for flight 1:

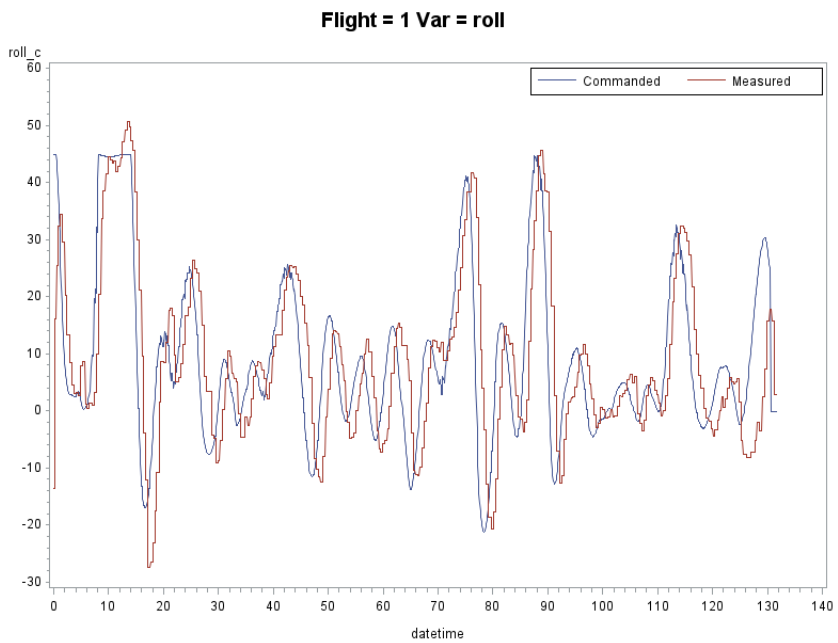
```
data trainscore;
input datetime roll_c roll_m yaw_c yaw_m pitch_c pitch_m vx_c vx_m vy_c vy_m
vz_c vz_m fault;
datalines;
... more lines ...
```

The following code generates the comparative plot of commanded and measured roll:

```
legend1 label=none position=(top right inside) value=('Commanded' 'Measured')
mode=share cborder=black;

title 'Flight = 1 Var = roll';
proc gplot data=trainscore;
   plot (roll_c roll_m)*datetime/overlay legend=legend1;
run; quit;
```

Figure 4 shows the graph generated by the code. We can observe that there is a lag between the command and the measurement. Similar graphs can be obtained for two other angle variables: pitch and yaw.

***Figure 4*** *Commanded and Measured Roll for Flight 1*

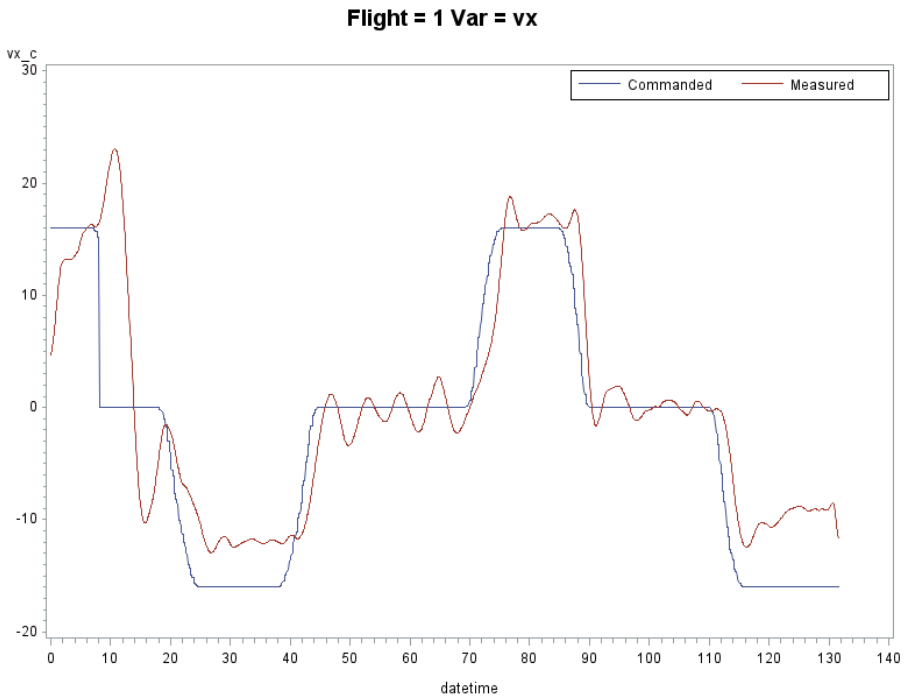


Now let's compare the commanded and measured velocity component, vx, by using the following code:

```
title 'Flight = 1 Var = vx';
proc gplot data=trainscore;
   plot (vx_c vx_m)*datetime/overlay legend=legend1;
run; quit;
```

The graph that it produces is shown in Figure 5.

***Figure 5.*** *Commanded and Measured vx for Flight 1*




We can see from the graph that the behavior of the measured variable, vx_m, cannot be described simply as a lag of the commanded variable, vx_c. We can observe the following:

- The lag changes over time.
- The measured variable, vx_m, tends to oscillate around a constant value of vx_c.
- A change in vx_m tends to temporarily overshoot (with a lag) the corresponding change in vx_c.

Similar observations can be drawn from the comparative graph of commanded and measured values of the vy velocity component. For the vz component, the commanded value is 0 for the whole flight (and for all other flights) even though the measured component changes over time.

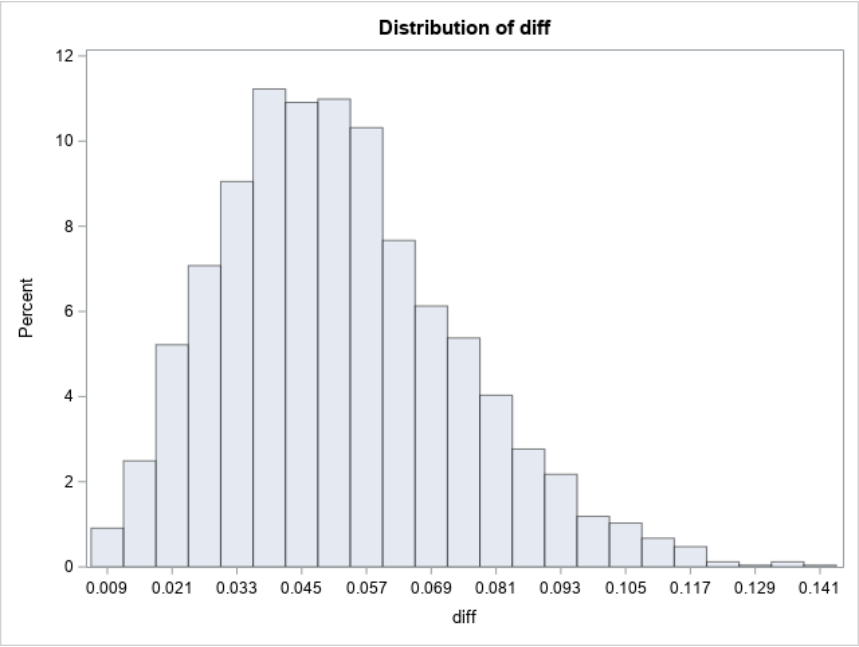
Given these observations, it might make sense to construct variables that are lagged differences between commanded and measured angle variables (roll, pitch, and yaw) in order to use them for modeling. For that, we need to determine the lag for each pair of variables. To compute a lag between two variables, the variables must be proper time series—that is, we need to have the same time difference between two consecutive observations. The following code generates a histogram of time difference for flight 1:

```
data tsd;
set trainscore;
retain t;
```

```
diff = datetime-t;
t=datetime;
run;

title "Flight = 1";
proc univariate data=tsd;
histogram diff;
run;
```

The plot of time difference is shown in Figure 6. We can see that the time difference is not constant.

***Figure 6.*** *Distribution of Time Difference between Two Consecutive Observations for Flight 1*




The following code finds the lag for each pair of angle variables:

```
%macro lag;
%local i var vars all_vars;
%let vars=roll yaw pitch;

%let i=1;
%let var=%scan(&vars,&i);
%do %while (&var ne);
   %let all_vars = &all_vars &var._c &var._m;
   %let i =%eval(&i+1);
   %let var=%scan(&vars,&i);
%end;

proc sql noprint;
select min(datetime), max(datetime) into :fault_time, :tot_time from trainscore
where fault=1;
quit;

data ts; set trainscore; datetime=datetime*100; run;
```

```
proc expand data=ts out=tsn to=second method=step;
var &all_vars;
id datetime;
format datetime best12.;
run;

data tsn; set tsn; datetime=datetime/100; run;

data mycas.ts; set tsn; where datetime<&fault_time; run;

%let i=1;
%let var=%scan(&vars,&i);
%do %while (&var ne);

proc cas;
 action tsInfo.selectLag result=r /
   casOut={name="lagOut_&var" replace="yes"}
   dttmName="datetime"
   minLag=0
   maxLag=400
   timeSeriesTable={name="ts"}
   targetName="&var._m"
   xVarNames={"&var._c"};
 run;
 quit;

proc sql noprint;
select distinct lags into :&var._lag from mycas.lagout_&var having
ccf=max(ccf);
quit;

%let i =%eval(&i+1);
%let var=%scan(&vars,&i);

%end;

data lags;
%let i=1;
%let var=%scan(&vars,&i);
%do %while (&var ne);
   &var = &&&var._lag;
   %let i =%eval(&i+1);
   %let var=%scan(&vars,&i);
%end;
fault_time = &fault_time; tot_time = &tot_time;
run;

%mend;

%lag;
```

First, we interpolate the data to have a frequency of 100 Hz by using the STEP method of the EXPAND procedure. Then we determine the lag for each pair of variables (commanded vs. measured) by using the tsInfo.selectLag CAS action. Input data are the interpolated flight data before the fault. The action computes the cross-correlation

function (CCF) for a specified lag range. The lag value is then selected using the maximum of the CCF function. The selected lags together with the times of failure and the total flight time are stored in the **work.lags** data set. Table 3 lists the selected lags for seven flights that experienced engine faults. For flight 1, the selected lag value for roll is 106, which means that on average there is a delay of 1.06 seconds between the command and the measurement for roll.

***Table 3.*** *Selected Lags for Interpolated Data (to 100 Hz) for Seven Flights*

| Flight | Roll | Yaw | Pitch | Fault Time, s | Total Time, s |
|---|---|---|---|---|---|
| 1 | 106 | 66 | 137 | 116.82 | 131.55 |
| 2 | 86 | 176 | 117 | 73.43 | 88.74 |
| 3 | 78 | 146 | 140 | 116.35 | 132.35 |
| 4 | 100 | 149 | 96 | 91.66 | 106.04 |
| 5 | 92 | 179 | 139 | 103.63 | 124.39 |
| 6 | 97 | 167 | 148 | 104.90 | 114.12 |
| 7 | 100 | 200 | 151 | 49.94 | 62.34 |

Next, we construct the input data by using the interpolated data and the selected lags:

```
%let ttime = 90;
proc sql noprint;
select roll, yaw, pitch into :roll_lag, :yaw_lag, :pitch_lag
from lags;
quit;

%macro diff;
%let vars=roll yaw pitch;

data tsdiff;
set tsn;
%let i=1;
%let var=%scan(&vars,&i);
%do %while (&var ne);
&var._d = &var._m-lag%sysfunc(strip(&&&var._lag))(&var._c);
%let i =%eval(&i+1);
%let var=%scan(&vars,&i);
%end;
run;
%mend;
%diff;

data work.train work.score;
set tsdiff;
if datetime<&ttime then output work.train;
else output work.score;
run;

data mycas.train; set train; run;
data mycas.score; set score; run;
```

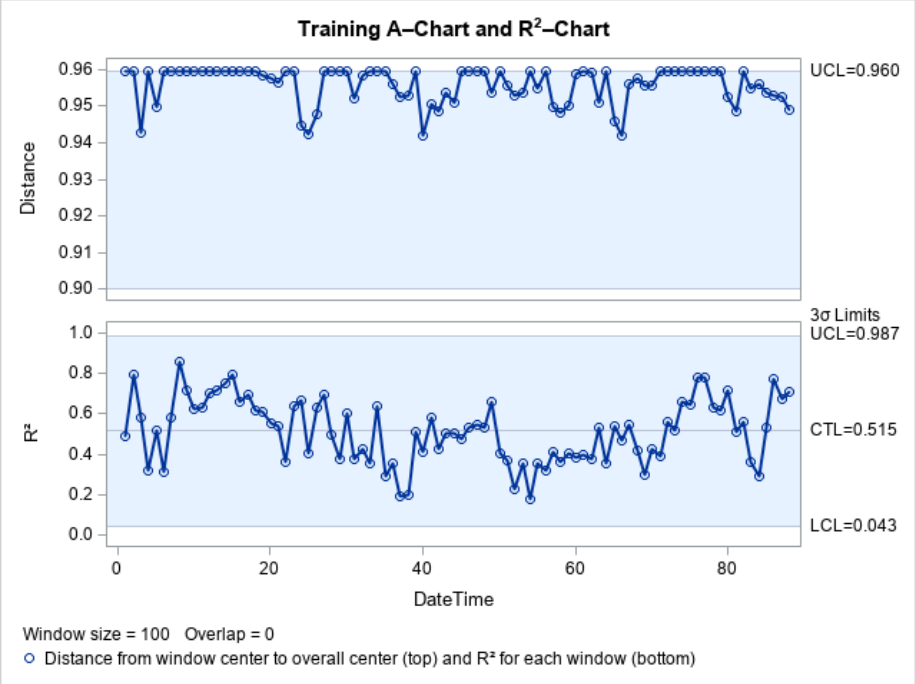
We chose the first 90 seconds of flight for the training data. Next, $K_T$ chart training is run using PROC KTTRAIN:

```
ods graphics / imagemap=on;

proc kttrain data=mycas.train window=100 a_lcl=.9;
input roll_d yaw_d pitch_d;
output out = mycas.out
       centers = mycas.centers
       kt = mycas.outkt
       scoreInfo = mycas.scoreinfo
       SV = mycas.SV;
run;
```

PROC KTTRAIN looks for the input data table **mycas.train**. The window size is set to 100. Because there are 100 observations per second, this means that PROC KTTRAIN generates one point for each second on the *a* and $R^2$ charts. Because the OVERLAP= option is omitted, there is no overlap between consecutive windows. The upper control limit for the *a* chart is set to 0.9 for better scaling of graphics. The training specifies three explanatory variables in the INPUT statement—roll_d, yaw_d, and pitch_d—which are the constructed lagged difference variables between commanded and measured angles. Because the TIMEID statement is omitted, the datetime is assumed to be the time ID variable. The OUTPUT statement generates the following SAS output tables: main output used by the charts, centers of SVDD for each window, $K_T$ chart-specific output (such as control limits), scoring information, and support vectors. PROC KTTRAIN produces an ODS table that contains the chart control limits. The ODS plot of the training charts (*a* chart and $R^2$ chart) is also produced.

The training charts are shown in Figure 7. All points on both training charts lie within the control limits.

***Figure 7.*** *Training Charts for Flight 1*




Next, $K_T$ chart monitoring is run using the data from 90 seconds on:

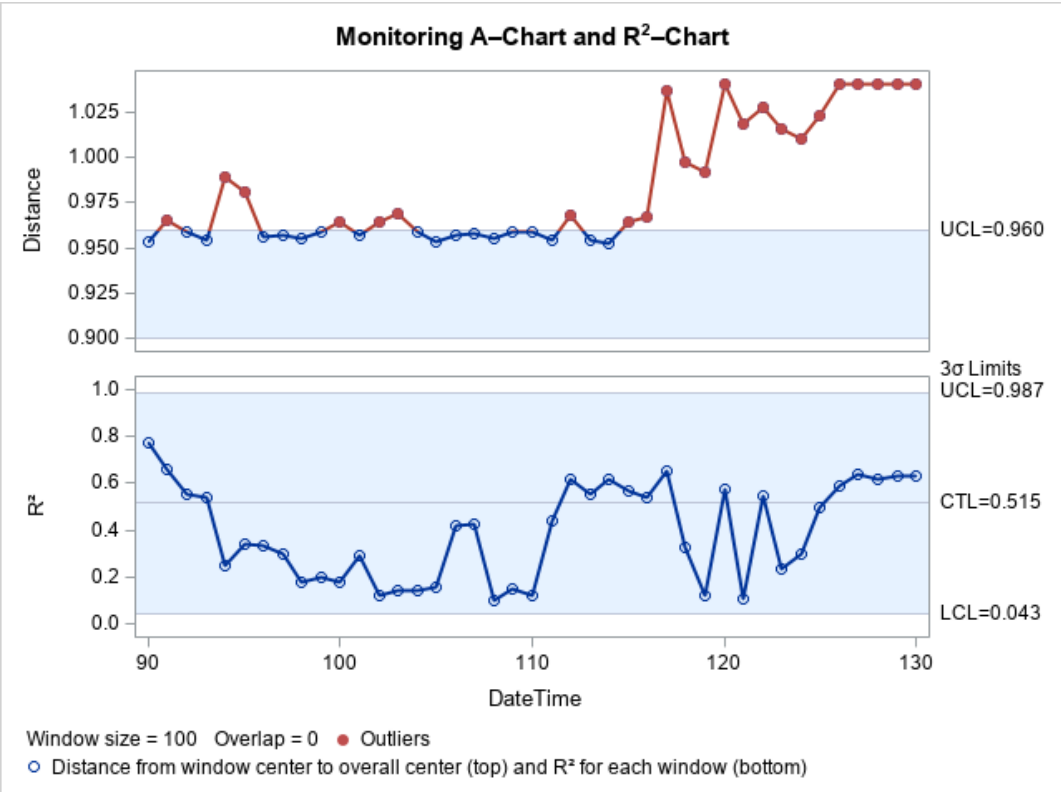
```
proc ktmonitor data=mycas.score sv=mycas.SV scoreInfo = mycas.scoreinfo;
```

```
output out = mycas.scoreout
       centers = mycas.scorecenters;
run;
```

The PROC KTMONITOR statement looks for the input data set **mycas.score**, the scoring information data set **mycas.scoreinfo**, and the data set **mycas.sv** with support vectors for centers SVDD. In the OUTPUT statement, the OUT= option stores the output in the **mycas.scoreout** data table, and the CENTERS= option stores the coordinates of the centers of each window in the **mycas.scorecenters** data table.

The monitoring charts for flight 1 are shown in Figure 8. As we can see from the *a* chart, starting at 117 seconds, the process center significantly exceeds the upper control limit, indicating the fault condition (the fault occurs at 116.8 seconds). There are a few false positives before that time. This might indicate the potential for improving the model further. The upper control limit for the *a* chart can also be relaxed by specifying the A_UCL= option in the PROC KTMONITOR statement. This option should be used with caution, because relaxing the upper control limit can prevent or delay the monitoring process from signaling the fault condition.

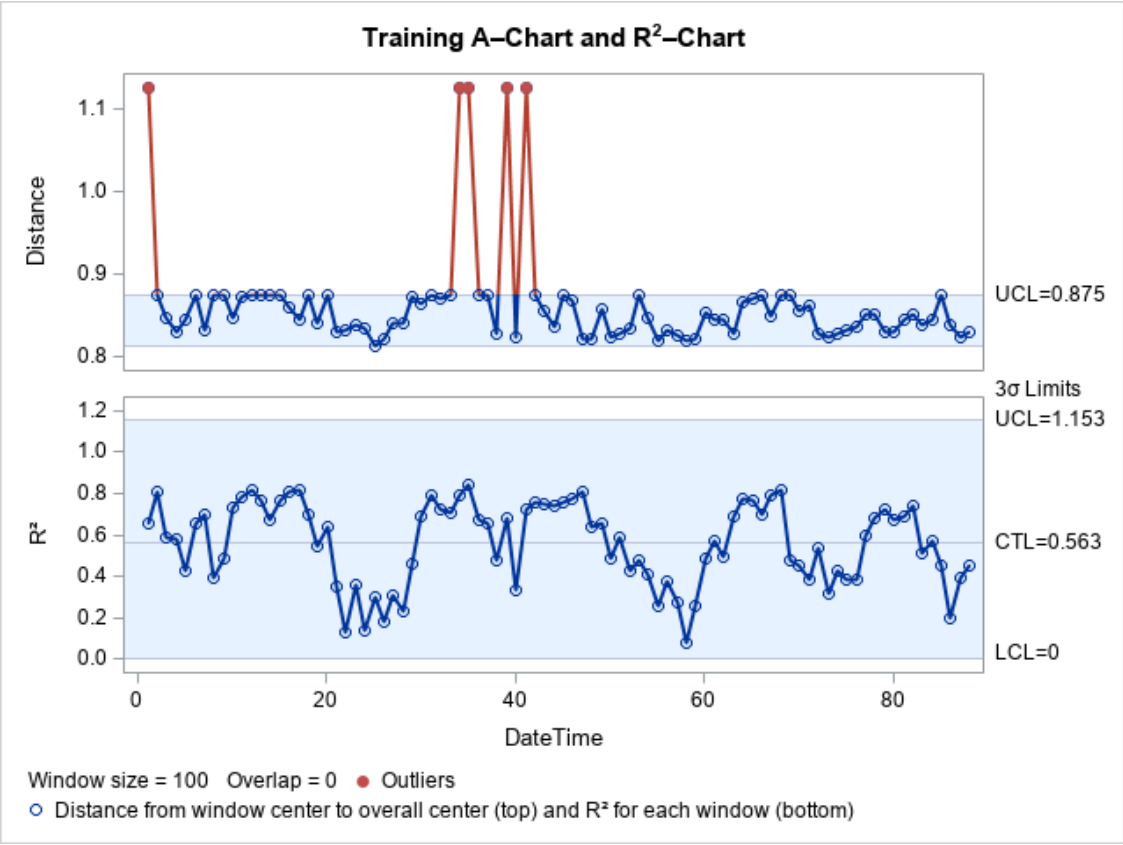
***Figure 8.*** *Monitoring Charts for Flight 1*




Let's consider another flight from this group, flight 6. The input data for $K_T$ chart procedures are constructed by applying the same data transformation and lag selection code to the original flight data that we applied to the flight 1 data. Then, $K_T$ chart training and monitoring were done using the same code that we used for flight 1. We used the same threshold—90 seconds of flight—to separate the training data from the scoring data. Figure 9

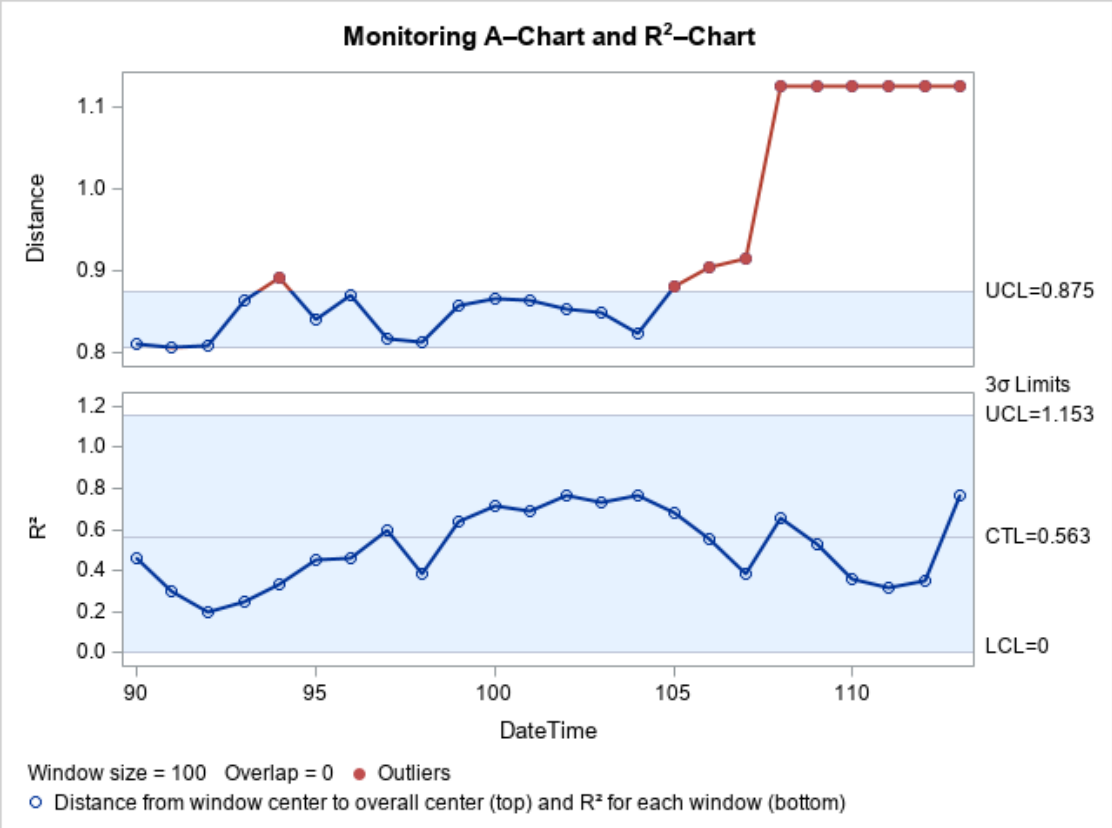
shows the training charts for flight 6. We can see that there are a few outliers in the training *a* chart. If we refer to the original data, we see that the outliers correspond to the times when the commanded yaw was changed by the maximum possible value, from −179 to 179 degrees (or from 179 to −179 degrees), and the measured yaw reacted with a lag that was different from the estimated lag value, resulting in a very large difference variable for yaw for these moments of time.

***Figure 9.*** *Training Charts for Flight 1*



The monitoring charts for flight 6 are shown in Figure 10. As the *a* chart shows, starting from 105 seconds of flight, the process center starts to go outside the upper control limit, indicating a fault condition (the actual fault occurs at 104.9 seconds).

***Figure 10.*** *Monitoring Charts for Flight 6*



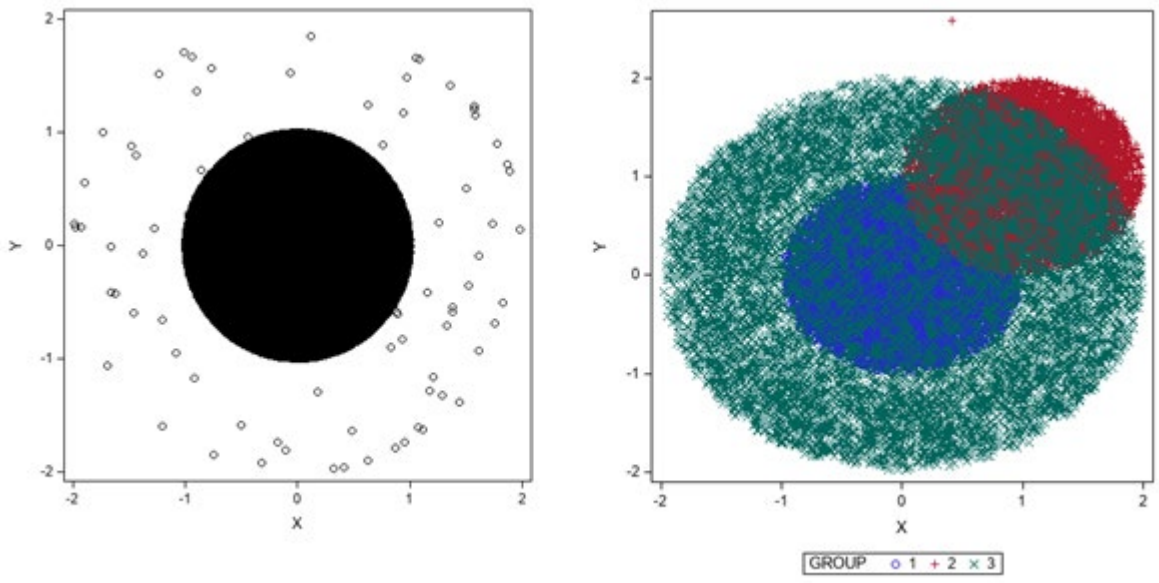

# Simulated Data Example

In this section, we assess the performance of $K_T$ monitoring by using simulated data from multiple scenarios. Unlike application to real data sets, application to simulated data sets yields results that are more predictable, because the data generation scheme is completely known. Therefore, by comparing the observed results with the expected results, we can determine whether the implementation is functioning correctly.

## Hypersphere Data

For the training data set, we generated 1,000,000 uniform samples from a three-dimensional sphere centered at (0, 0, 0) with a radius of 1. Some outliers were also generated to stabilize the $K_T$ chart training process. The outliers are uniform samples from the surface of a three-dimensional sphere with a radius of 2. The percentage of outliers is fixed at 0.0001. The testing data set contains three groups of data, each having 10,000 uniform samples. The first group of data is generated in the same way as the training data. The second group is also generated from a unit sphere but is centered at (1, 1, 1). The third group is generated from a sphere centered at (0, 0, 0) but with a radius of 2. Therefore, the difference between group 1 and group 2 reflects a location change, while the difference between group 1 and group 3 reflects a scale change. We expect that the *a* chart will classify groups 1 and 3 as normal but group 2 as an anomaly, whereas the $R^2$ chart will classify groups 1 and 2 as normal but group 3 as an anomaly. Figure 11 shows the data that were used for the simulation.

***Figure 11.*** *Training Data (left) and Testing Data (right) for Three-Dimensional Sphere Simulation*




The following code generates training and testing data for the three-dimensional sphere simulation:

```
%macro generate_sphere(window_size = 1000,
 window_num = 1000,
 dimension = 3,
 center_shift = 1,
 radius_multiplier = 2,
 outlier_fraction = 0.0001,
 seed = 12345);

proc iml;
call randseed(&seed);
train_size = &window_size*&window_num;
X = j(train_size,&dimension,.);
call randgen(X,"Normal",0,1);
r = randfun(train_size,"Uniform",0,1)##(1/&dimension);
```

```
norm = X[,##]##(0.5);
g = randfun(train_size,"Bernoulli",&outlier_fraction);
g = g#2 - ((g-1)#r);
X = g#X/norm;
run Scatter(X[,1],X[,2]);
varNames = cats("x",char(1:ncol(X)));
create trainSphere from X [colname = varNames];
append from X;
close trainSphere;
group_size = &window_size*10;
test_size = group_size*3;
X = j(test_size,3,.);
group = j(test_size,1,.);
call randgen(X,"Normal",0,1);
r = randfun(test_size,"Uniform",0,1)##(1/3);
r[(test_size-group_size+1):test_size] = r[(test_size-group_size+1):test_size] *
&radius_multiplier;
norm = X[,##]##(0.5);
g = randfun(test_size,"Bernoulli",&outlier_fraction);
g = g#2 - ((g-1)#r);
X = g#X/norm;
do i = 1 to 3;
group[((i-1)*group_size+1):(i*group_size)] = i;
end;
X[(group_size+1):(2*group_size),] = X[(group_size+1):(2*group_size),] +
&center_shift;
run Scatter(X[,1],X[,2]) group = group;
varNames = cats("x",char(1:ncol(X)));
create testSphere from X [colname = varNames];
append from X;
close testSphere;
quit;
data mycas.train;
set trainSphere;
datetime = _N_;
run;

data mycas.test;
set testSphere;
datetime = _N_;
run;
%mend generate_sphere;

%generate_sphere;
```

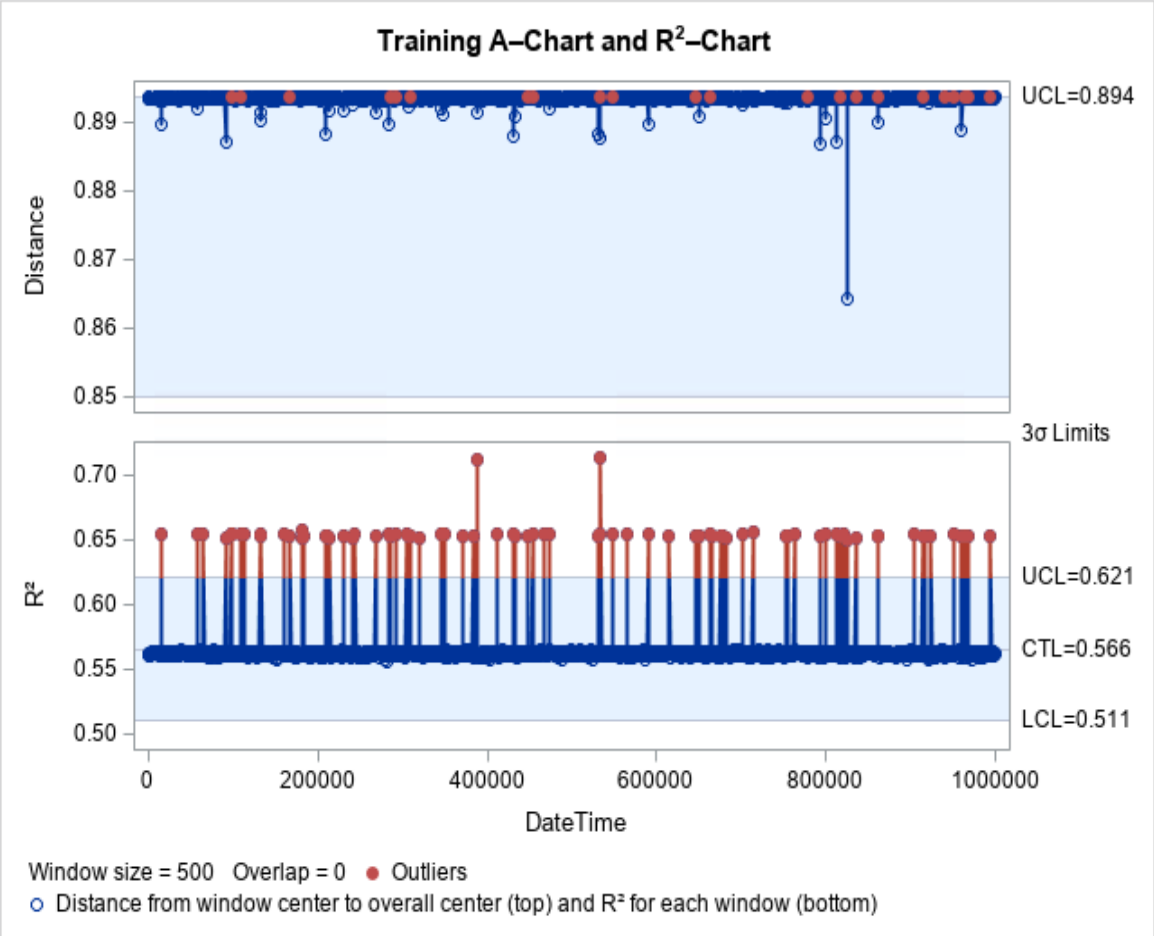
$K_T$ chart training and monitoring are then run using the following settings:

```
proc kttrain data=mycas.train window=500 overlap=0 frac=1e-4 a_lcl=0.85;
input x1-x3;
kernel / bww = 1 bwc = trace;
output out = mycas.out
       centers = mycas.centers
       kt = mycas.outkt
       scoreInfo = mycas.scoreinfo
       SV = mycas.SV;
run;
```

```
proc ktmonitor data = mycas.test sv = mycas.sv scoreinfo = mycas.scoreinfo
a_lcl=0.8;
    output out = mycas.outmon centers = mycas.centersmon;
run;
```

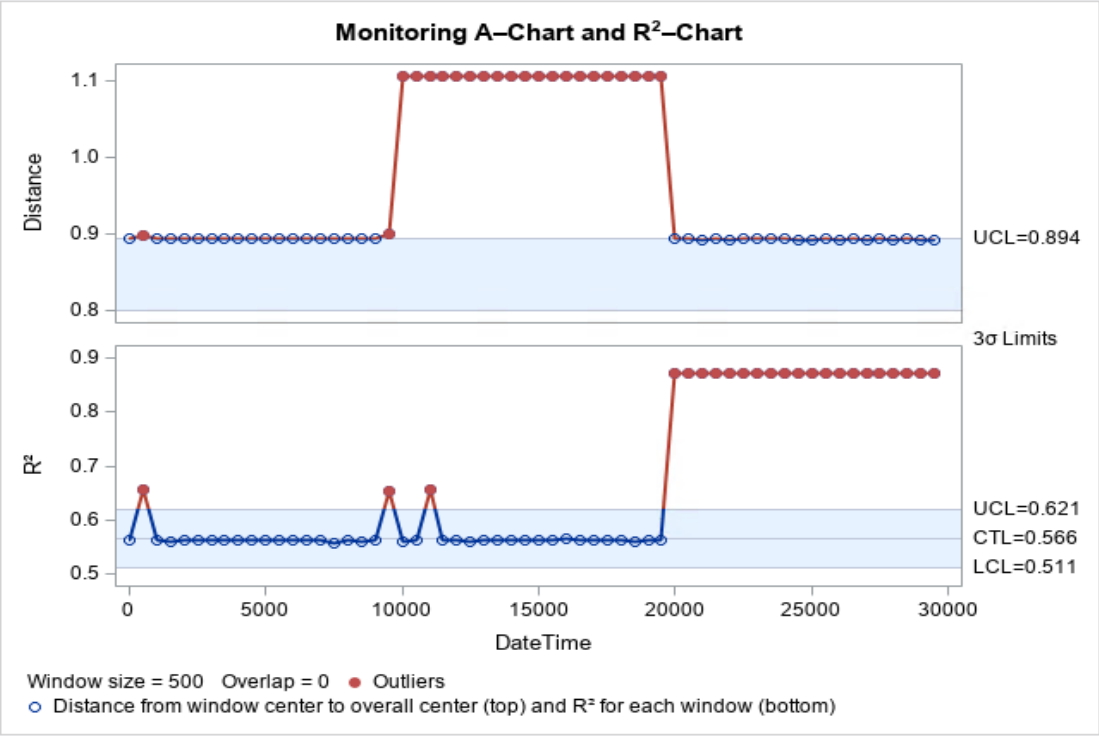
The training and testing charts of the three-dimensional sphere simulation are shown in Figures 12 and 13, respectively. In Figure 12, the majority of the training observations, representing samples from the unit sphere, are classified as inliers, as expected. In the $R^2$ chart, a small number of points are classified as outliers. These points represent outlying samples that were generated from the sphere centered at (0, 0, 0) with a radius of 2.

***Figure 12.*** *Training Charts for Three-Dimensional Sphere Data*



In Figure 13, the *a* chart recognized the second group of observations as outliers since they are from a distribution that differs from the training distribution by a location shift. A location shift results in a change in the centers of individual windows as well as the overall center, to which the *a* chart is sensitive. The $R^2$ chart, on the other hand, is sensitive to a change in variation in the data. The distribution of the second group has the same radius as the training distribution; therefore, the second group of observations are not classified as outliers in the $R^2$ chart. In contrast, the third group of observations are from a distribution that differs from the training distribution by a radius change. Thus, the points in this group are classified as outliers in the $R^2$ chart but not in the *a* chart.

***Figure 13.*** *Testing Charts for Three-Dimensional Sphere Data*




## Hyperellipsoid Data

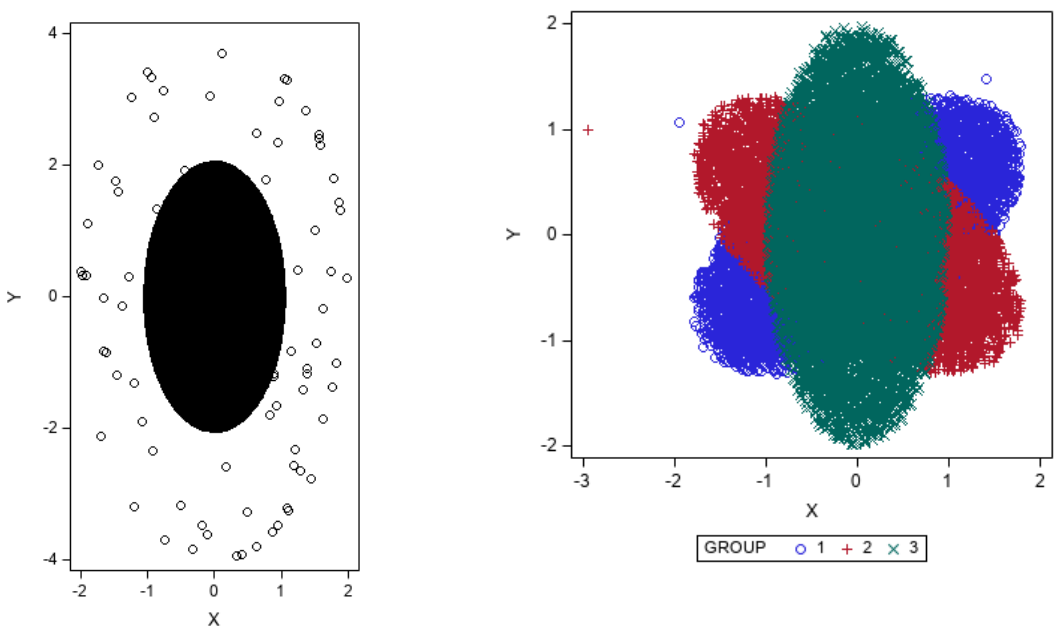
We can easily obtain uniform samples from a three-dimensional ellipsoid by multiplying a scale matrix to uniform samples from a three-dimensional sphere. For the training data, we set the x, y, and z radii to 1, 2, and 5, respectively. The following code generates training data for the three-dimensional ellipsoid simulation:

```
%macro generate_ellipsoids(window_size = 1000, window_num = 1000, x_radius = 1,
y_radius = 2, z_radius = 5, center_shift = 1, radius_multiplier = 2,
outlier_fraction = 1e-4, seed = 12345);
proc iml;
    call randseed(&seed);
    train_size = &window_size*&window_num;
    X = j(train_size,3,.);
    call randgen(X,"Normal",0,1);
    r = randfun(train_size,"Uniform",0,1)##(1/3);
```

```
    pi = constant('pi');
    norm = X[,##]##(0.5);
    g = randfun(train_size,"Bernoulli",&outlier_fraction);
    g = g#2 - ((g-1)#r);
    X = g#X/norm;
    M = I(3);
    M[1,1] = &x_radius;
    M[2,2] = &y_radius;
    M[3,3] = &z_radius;
    X = X*M;
    run Scatter(X[,1],X[,2]);
    varNames = cats("x",char(1:ncol(X)));
    create trainEllipsoid from X [colname = varNames];
    append from X;
    close trainEllipsoid;

data mycas.train;
    set trainEllipsoid;
    datetime = _N_;
run;
%mend generate_ellipsoids;

%generate_ellipsoids;
```

Figure 14 shows the data that were used for the simulation.

***Figure 14.*** *Training and Testing Data for Three-Dimensional Ellipsoid Simulation*




It is well known that distance metrics are invariant to rotation. Because $K_T$ charts use a distance metric to differentiate outliers from inliers, they are not capable of detecting the difference of a distribution before and after some rotation. To demonstrate this, we generated a testing data set that contains samples from the training

distribution after rotations of 60, 120, and 180 degrees, respectively. The following code generates this data set:

```
%macro generate_rotating_ellipsoids(window_size = 1000, window_num = 1000,
rotation_number = 3, x_radius = 1, y_radius = 2, z_radius = 5, outlier_fraction
= 1e-4, seed = 12345);
proc iml;
    call randseed(&seed);
    group_size = &window_size*10;
    test_size = group_size*&rotation_number;
    X = j(test_size,3,.);
    group = j(test_size,1,.);
    call randgen(X,"Normal",0,1);
    r = randfun(test_size,"Uniform",0,1)##(1/3);
    pi = constant('pi');
    norm = X[,##]##(0.5);
    g = randfun(test_size,"Bernoulli",&outlier_fraction);
    g = g#2 - ((g-1)#r);
    X = g#X/norm;
    M[1,1] = &x_radius;
    M[2,2] = &y_radius;
    M[3,3] = &z_radius;
    X = X*M;
    R = I(3);
    do i = 1 to 3;
          R[1,1] = cos(i*pi/3);
          R[2,2]= cos(i*pi/3);
          R[1,2]= -sin(i*pi/3);
          R[2,1]= sin(i*pi/3);
          X[((i-1)*group_size+1):(i*group_size),] = X[((i-
1)*group_size+1):(i*group_size),]*R;
          group[((i-1)*group_size+1):(i*group_size)] = i;
    end;
    run Scatter(X[,1],X[,2]) group = group;
    varNames = cats("x",char(1:ncol(X)));
    create testEllipsoid from X [colname = varNames];
    append from X;
    close testEllipsoid;
quit;

data mycas.test;
    set testEllipsoid;
    datetime = _N_;
run;
%mend generate_rotating_ellipsoids;

%generate_rotating_ellipsoids;
```

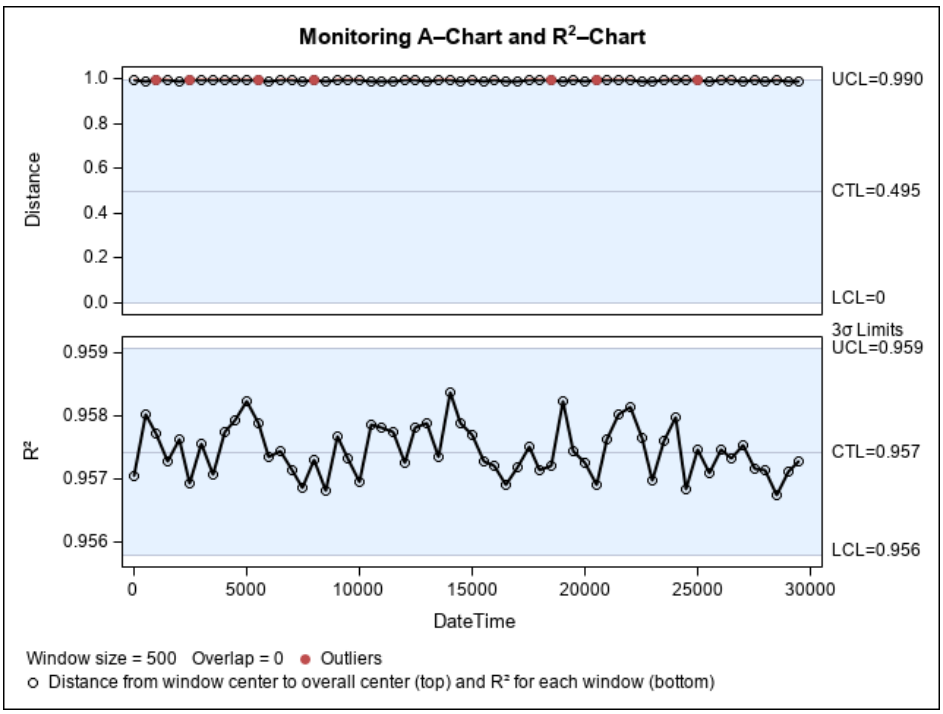
$K_T$ chart training and monitoring are then run using the following settings:

```
ods graphics / imagemap=on;

proc kttrain data=mycas.train window=500 overlap=0 frac=1e-4 a_lcl=0.85;
input x1-x3;
kernel / bww = 0.5 bwc = trace;
output out = mycas.out
       centers = mycas.centers
       kt = mycas.outkt
```

```
        scoreInfo = mycas.scoreinfo
        SV = mycas.SV;
run;

proc ktmonitor data = mycas.test sv = mycas.sv scoreinfo = mycas.scoreinfo
a_lcl=0.8;
output out = mycas.outmon centers = mycas.centersmon;
run;
```

The monitoring charts are shown in Figure 15. Both the *a* chart and the $R^2$ chart classified a great majority of testing observations (if not all) as inliers. This demonstrates that the $K_T$ charts are indeed incapable of detecting anomalies caused by rotating the distribution of the normal data.

***Figure 15.*** *Testing Charts for Three-Dimensional Ellipsoid Simulation*




## Two-Spheres Data

Many industrial processes have multiple normal operating modes. To demonstrate the ability of the $K_T$ chart to monitor such processes, we simulated our training data as uniform samples of two spheres. In particular, 80% of the training data come from a unit sphere centered at (0, 0, 0) (mode 1), and the rest come from a unit sphere centered at (2, 2, 2) (mode 2). We then simulated five groups of samples in the testing data set. The distribution of group 1 is identical to that of mode 1; the distribution of group 2 differs from that of mode 1 by a location change; the distribution of group 3 is identical to that of mode 2; the distribution of group 4 differs from that of mode 2 by a location change; and the distribution of group 5 is a unit sphere whose center is different from those of mode 1 and mode 2. Ideally, the *a* chart should classify group 5 as outliers, whereas the $R^2$ chart should classify groups 2 and 4 as outliers.

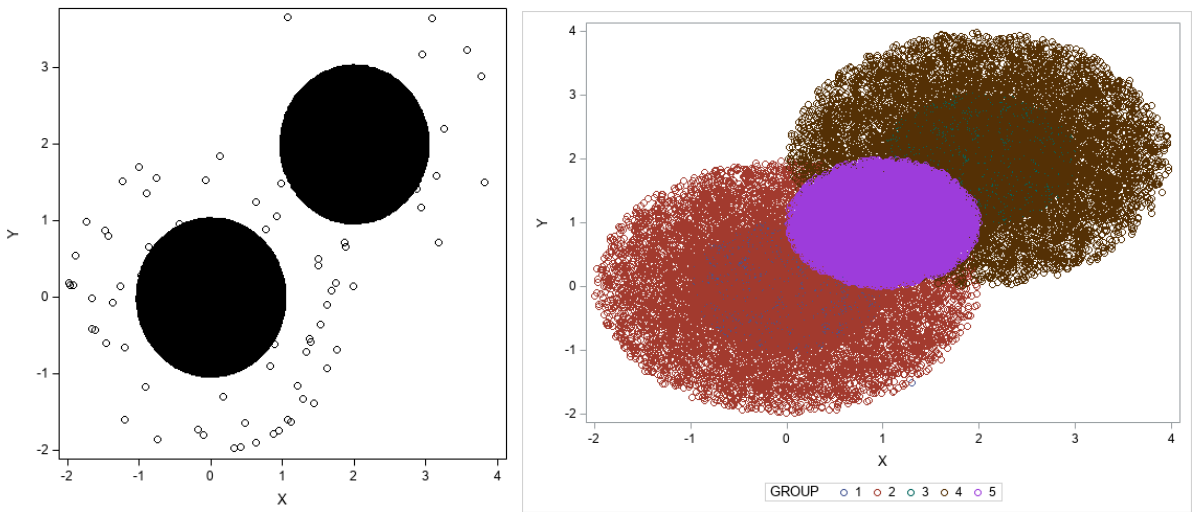
The following code generates training and testing data for two-spheres simulation:

```
%macro generate_two_modes(window_size = 1000, window_num = 1000, dimension = 3,
center_shift = 2, radius_multiplier = 2, outlier_fraction = 0.0001, seed =
12345);
proc iml;
    call randseed(&seed);
    train_size = &window_size*&window_num;
    X = j(train_size,&dimension,.);
    call randgen(X,"Normal",0,1);
    r = randfun(train_size,"Uniform",0,1)##(1/&dimension);
    norm = X[,##]##(0.5);
    g = randfun(train_size,"Bernoulli",&outlier_fraction);
    g = g#2 - ((g-1)#r);
    X = g#X/norm;
    do i = 1 to &window_num;
          res = mod(i,5);
          if res = 0 then X[((i-1)*&window_size+1):(i*&window_size),] = X[((i-
1)*&window_size+1):(i*&window_size),] + &center_shift;
    end;
    run Scatter(X[,1],X[,2]);
    varNames = cats("x",char(1:ncol(X)));
    create trainDisk from X [colname = varNames];
    append from X;
    close trainDisk;
    test_group = 5;
    group_size = &window_size * 10;
    test_size = group_size*test_group;
    X = j(test_size,&dimension,.);
    group = j(test_size,1,.);
    call randgen(X,"Normal",0,1);
    r = randfun(test_size,"Uniform",0,1)##(1/&dimension);
    r[(group_size+1):(2*group_size)] = r[(group_size+1):(2*group_size)] *
&radius_multiplier;
    r[(3*group_size+1):(4*group_size)] = r[(3*group_size+1):(4*group_size)] *
&radius_multiplier;
    norm = X[,##]##(0.5);
    g = randfun(test_size,"Bernoulli",&outlier_fraction);
    g = g#2 - ((g-1)#r);
    X = g#X/norm;
    X[(2*group_size+1):(4*group_size),] = X[(2*group_size+1):(4*group_size),] +
2;
    X[(4*group_size+1):(5*group_size),] = X[(4*group_size+1):(5*group_size),] +
1;
    do i = 1 to test_group;
          group[((i-1)*group_size+1):(i*group_size)] = i;
    end;
    run Scatter(X[,1],X[,3]) group = group;
    varNames = cats("x",char(1:ncol(X)));
    create testDisk from X [colname = varNames];
    append from X;
    close testDisk;
quit;

data mycas.train;
    set trainDisk;
    datetime = _N_;
run;
```

```
   data mycas.test;
       set testDisk;
       datetime = _N_;
   run;

   %mend generate_two_modes;

%generate_two_modes;
```

Figure 16 shows the results.

***Figure 16.*** *Training and Testing Data for Two-Spheres Simulation*

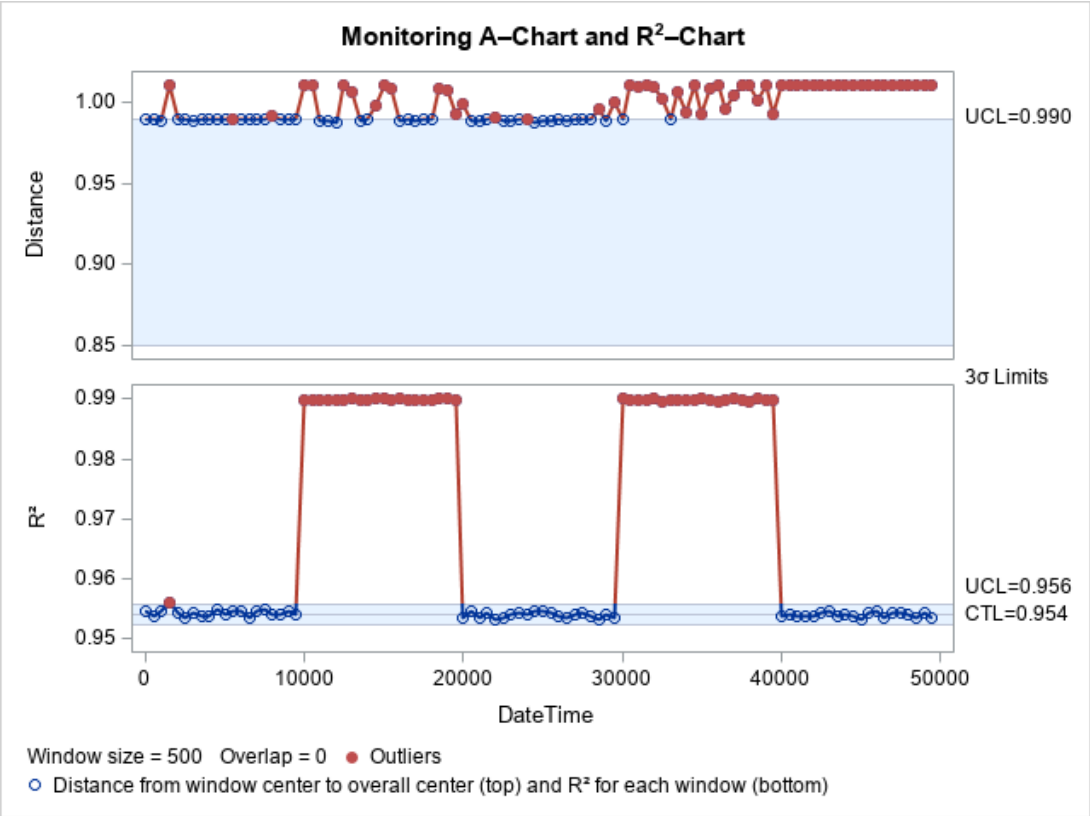



$K_T$ chart training and monitoring are then run using the following code:

```
   ods graphics / imagemap=on;

   proc kttrain data=mycas.train window=500 overlap=0 frac=1e-4 a_lcl=0.85;
   input x1-x3;
   kernel / bww = 0.3 bwc = trace;
   output out = mycas.out
          centers = mycas.centers
          kt = mycas.outkt
          scoreInfo = mycas.scoreinfo
          SV = mycas.SV;
   run;

   proc ktmonitor data = mycas.test sv = mycas.sv scoreinfo = mycas.scoreinfo;
   output out = mycas.outmon centers = mycas.centersmon;
   run;
```

The monitoring charts are shown in Figure 17. The *a* chart classifies most observations in group 5 as outliers, whereas the $R^2$ chart classifies most observations in groups 2 and 4 as outliers. These results are similar to what we expected, and they demonstrate the effectiveness of the $K_T$ charts in multiple operating modes.

***Figure 17.*** *Testing Charts for Two-Spheres Simulation*




# Conclusion

We have demonstrated that $K_T$ chart monitoring can be effective in monitoring machine or process condition through the use of high-frequency data generated by multiple sensors. The monitoring technique is easy to implement, and it can be used to detect impending failure or a fault condition in a machine or process. The *a* chart and the $R^2$ chart can be used in concert to monitor the central tendency and spread of the process. $K_T$ charts make no restrictive distributional assumptions and can handle multimodal data. We provided an example of monitoring process data from the Tennessee Eastman data set, where $K_T$ chart monitoring indicated the presence of a fault through the *a* chart. The Drone data example showed the utility of $K_T$ charts for monitoring equipment health. Although the examples provided in this paper are from industrial process control and predictive maintenance, $K_T$ charts can be used in any applications where multivariate data must be monitored for deviations from normal class or stable conditions.

Many processes exhibit transient behaviors that are generally addressed quickly by the control system through the use of feedback or feedforward mechanisms. For example, in a system designed to heat water to a constant temperature, variation of the inlet water temperature in a system is reflected by changes in the temperature of the heating element. Sensor data from such a system, if individual observations are monitored, can lead to false alarms. The window approach that is used in $K_T$ charts helps reduce the impact of such transient changes.

$K_T$ charts involve training an SVDD model for each window. The SVDD solvers that $K_T$ charts use can perform this

training efficiently. In using $K_T$ charts, the most important parameter that the analyst is expected to decide is the window size and overlap, if any. Other than that, the default parameters of KT charts can be used. The use of defaults reduces overhead in applying the technique and can lead to faster implementation.

# References


Kakde, D., Peredriy, S., & Chaudhuri, A. (2017). A Non-parametric Control Chart for High Frequency Multivariate Data. *Annual Reliability and Maintainability Symposium (RAMS).* Piscataway, NJ: Institute of Electrical and Electronics Engineers.

Keipour, A. (2020). *ALFA Dataset Tools*. Retrieved from github.com: **https://github.com/castacks/alfa-dataset-tools/**

Keipour, A., Mousaei, M., & Scherer, S. (2020). *ALFA: A dataset for UAV Fault and Anomaly Detection. DOI:10.1184/R1/12707963*. Retrieved from **https://doi.org/10.1184/R1/12707963**

Montgomery, D. C. (2019). *Introduction to Statistical Quality Control.* John Wiley & Sons.

Ricker, N. L. (2002). *Tennessee Eastman Challenge Archive, MATLAB 7.x Code.* Retrieved from University of Washington, Seattle, Department of Chemical Engineering.: **http://depts.washington.edu/control/LARRY/TE/download.html**

SAS Institute Inc. (2021). *KT Chart Sample Code.* Retrieved from github.com: **https://github.com/sassoftware/kt-chart-sample-code**

SAS Institute Inc. (2021). *KTMONITOR Procedure.* Retrieved from SAS® Viya® Programming Documentation: **https://documentation.sas.com/doc/en/pgmsascdc/v_010/casforecast/casforecast_ktmonitor_toc.htm**

SAS Institute Inc. (2021). *KTTRAIN Procedure.* Retrieved from SAS® Viya® Programming Documentation: **https://documentation.sas.com/doc/en/pgmsascdc/v_010/casforecast/casforecast_kttrain_toc.htm**

Tax, D. M., & Duin, R. P. (2004). Support Vector Data Description. *Machine Learning*, 54: 45-66.

**Release Information** Content Version: 1.0 June 2021



To contact your local SAS office, please visit: sas.com/offices