## Contents

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>What's New in SAS Visual Forecasting 8.5 Time Series Packages</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>Introduction to Packages for the TSMODEL Procedure</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>Automatic Time Series Modeling Package</td>
<td>11</td>
</tr>
<tr>
<td>4</td>
<td>External Languages Package</td>
<td>121</td>
</tr>
<tr>
<td>5</td>
<td>Multivariate Singular Spectrum Analysis Package</td>
<td>165</td>
</tr>
<tr>
<td>6</td>
<td>Simple Forecast Service Package</td>
<td>191</td>
</tr>
<tr>
<td>7</td>
<td>Singular Spectrum Analysis Package</td>
<td>211</td>
</tr>
<tr>
<td>8</td>
<td>Subspace Tracking Package</td>
<td>239</td>
</tr>
<tr>
<td>9</td>
<td>Time Filters Package</td>
<td>257</td>
</tr>
<tr>
<td>10</td>
<td>Time-Frequency Analysis Package</td>
<td>323</td>
</tr>
<tr>
<td>11</td>
<td>Time Series Analysis Package</td>
<td>353</td>
</tr>
<tr>
<td>12</td>
<td>Time Series Dimension Reduction Package</td>
<td>425</td>
</tr>
<tr>
<td>13</td>
<td>Time Series Distance Measure Package</td>
<td>451</td>
</tr>
<tr>
<td>14</td>
<td>Time Series Model Package</td>
<td>489</td>
</tr>
<tr>
<td>15</td>
<td>Time Series Motif Discovery Package</td>
<td>571</td>
</tr>
<tr>
<td>16</td>
<td>Utility Package</td>
<td>605</td>
</tr>
</tbody>
</table>
Chapter 1
What’s New in SAS Visual Forecasting 8.5
Time Series Packages

Contents

| Overview | 1 |
| New Packages | 1 |
| External Languages (EXTLANG) Package | 1 |
| Time Series Dimension Reduction (TDR) Package | 2 |
| Package Enhancements | 2 |
| Automatic Time Series Modeling (ATSM) Package | 2 |

Overview

SAS Visual Forecasting 8.5 adds two new packages and enhancements to an existing package.

New Packages

SAS Visual Forecasting 8.5 adds the external languages (EXTLANG) and time series dimension reduction (TDR) packages.

External Languages (EXTLANG) Package

- The external languages (EXTLANG) package enables you to integrate external-language code that is written in Python and R into your SAS program.

- The package enables you to call arbitrary external-language code from within your SAS program and automatically handles the exchange of data among the different environments.

- By virtue of being integrated into the TSMODEL procedure, Python and R code are seamlessly run in parallel across all time series.

- The package supports various versions of the Python 2, Python 3, and R languages.
Time Series Dimension Reduction (TDR) Package

- The time series dimension reduction (TDR) package enables you to reduce the dimensionality of time series.

- The dimension reduction methods include piecewise aggregate approximation, symbolic aggregate approximation, discrete Fourier transformation, discrete wavelet transformation, random projection, and singular value decomposition.

- The resulting reduced-dimension time series can be used to efficiently perform various tasks such as similarity, classification, clustering, and so on.

Package Enhancements

SAS Visual Forecasting 8.5 adds enhancements to one package: the ATSM package.

Automatic Time Series Modeling (ATSM) Package

- The new EventGroup method of the EVENT object enables you to specify a group event definition that is stored in the EVENT object.

- The new 'CHOOSE' argument of the FORENG object enables you to specify the name of the winning forecast model.

- The new 'ENDZEROS.MAXNUM', 'ENDZEROS.MAXPCT', and 'ENDZEROS.MINOBS' arguments of the FORENG object enable you to configure the _ZERO_ model test, which determines whether the FORENG object produces an all-zero forecast when trailing zeros are present in the dependent series.

- The new 'ZEROMISS' argument of the TSDF.AddSeries, TSDF.AddX, and TSDF.AddY methods enables you to specify how to interpret beginning and ending zero values in the ancillary, independent, and dependent series that you provide for the generation of forecast models.
Overview

The chapter describes the use of packages in the TSMODEL procedure. It defines the main concepts of packages and describes the steps for using packages in the TSMODEL procedure, the package syntax conventions and terminology, and the structural organization of each package’s chapter.


---

Contents

Overview .................................................. 3
What Is a Package? ...................................... 4
What Are Objects, Classes, Instances, Functions, and Methods? ................. 5
  What Are Objects? .................................. 5
  What Are Classes? ................................. 6
  What Are Instances? ............................... 6
  What Are Functions? .............................. 6
  What Are Methods? ............................... 6
How Do You Use Packages? ................................ 7
  The REQUIRE Statement .......................... 7
  The DECLARE OBJECT Statement ............... 7
Chapter Organization .................................. 8
Syntax Conventions ................................... 9
  Object Names .................................... 9
  Object Method Names ............................ 9
Return Codes .......................................... 10
What Is a Package?

A package is a set of related specialized objects and functions that tackles a unique facet of the time series analysis problem. You can use these specialized objects and functions when you write your custom SAS code in order to gain access both to cutting-edge data analysis tools and to utilities that are designed to significantly speed up code development and improve the quality of the resulting code.

Table 2.1 shows the packages that you can use with the TSMODEL procedure. Each package has a unique abbreviation that contains up to five alphabetic characters.

<table>
<thead>
<tr>
<th>Package Name</th>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Automatic time series analysis and forecasting</td>
<td>ATSM</td>
<td>Tools for automatic modeling and forecasting of time series using various model families such as exponential smoothing (ESM), ARIMA, intermittent demand (IDM), and unobserved component models (UCM)</td>
</tr>
<tr>
<td>External Languages</td>
<td>EXTLANG</td>
<td>Tools for enabling the seamless integration of external language programs into SAS environments</td>
</tr>
<tr>
<td>Time series motif discovery</td>
<td>MOTIF</td>
<td>Tools for the discovery of frequent patterns or repeated subsequences in time series</td>
</tr>
<tr>
<td>Multivariate singular spectrum analysis</td>
<td>MSSA</td>
<td>Tools for decomposing a multivariate time series into additive components and categorizing those components based on the magnitudes of their contributions</td>
</tr>
<tr>
<td>Simple forecast service</td>
<td>SFS</td>
<td>Tools for automatic forecasting of time series with a simple-to-use interface and using only ARIMA models</td>
</tr>
<tr>
<td>Singular spectrum analysis</td>
<td>SSA</td>
<td>Tools for decomposing a time series into ARIMA models</td>
</tr>
<tr>
<td>Subspace tracking</td>
<td>SST</td>
<td>Tools for tracking the principal subspace of multiple time series</td>
</tr>
<tr>
<td>Time filters</td>
<td>TIMFIL</td>
<td>Tools for filtering and aggregation of time series</td>
</tr>
<tr>
<td>Time frequency analysis</td>
<td>TFA</td>
<td>Tools for efficient analysis of time series in both time and frequency domains</td>
</tr>
<tr>
<td>Time series analysis</td>
<td>TSA</td>
<td>Tools for efficient statistical analysis of time series (transformations, decompositions, statistical tests for intermittency, seasonality, stationarity, and forecast bias, and so on)</td>
</tr>
<tr>
<td>Time series dimension reduction</td>
<td>TDR</td>
<td>Tools for reducing the dimensionality of time series</td>
</tr>
</tbody>
</table>
What Are Objects, Classes, Instances, Functions, and Methods?

This section describes the computer programming terminology that is used in the documentation of packages for the TSMODEL procedure.

What Are Objects?

A programming object, a fundamental concept in object-oriented programming (OO) and analogous to an object found in nature, is the description of the state and behavior of a system. For example, a dog is a real-world object that has both a state (name, age, weight, fur color, tail length, running speed, number of teeth, friendliness level, and so on) and behavior (running, biting, eating, hunting, wagging its tail, whining, sleeping, tearing the sofa, and so on). A programming object is similar to a real-world object, but it stores its state as internal variables called fields and communicates its behavior to other objects via a set of functions called methods.

Many of the objects in the TSMODEL procedure packages are collector objects that are specifically designed to postprocess the output of other objects and then store the results in a CAS table. This postprocessing and storing the results can help accelerate your development process. These objects are called collectors because they “gather” the results of other objects and store them for future reference.

Conversely, some other objects are specifically designed to retrieve the data that have been stored in a CAS table by collectors and make those data available to your custom SAS code. These objects are called repeater objects.

<table>
<thead>
<tr>
<th>Package Name</th>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time series distance measure</td>
<td>TSD</td>
<td>Tools for measuring the distance between two time series or among sequences in temporal data</td>
</tr>
<tr>
<td>Time series model</td>
<td>TSM</td>
<td>Tools for efficient time series modeling and forecasting</td>
</tr>
<tr>
<td>Utility</td>
<td>UTL</td>
<td>Tools for performing basic statistical computations on pairs of actual and predicted time series</td>
</tr>
</tbody>
</table>
What Are Classes?

A class is a programming recipe for creating objects. Whereas real objects can be described by an unlimited number of features, a class uses a limited number of features to describe as faithfully as possible both the state and behavior of an object inside a computer program. The size and complexity of the feature set that a class uses (that is, how complex the recipe for the object is) depends on the requirements of each application. A class uses fields of varying data types to represent the features that describe the state of an object. Similarly, a class provides a set of methods that enables external objects to query and even modify the object’s behavior. Furthermore, a class also generally provides the recipe for a constructor, which is a special method that assigns an initial state to the class fields.

The words “class” and “object” are sometimes used interchangeably because a class is the representation of an object within the constraints of a programming language.

What Are Instances?

An instance of a class is a unique replica of a class that represents an object inside a computer program. A computer program creates instances of a class by allocating a block of memory and using different portions of that block to represent the different fields and methods that are described in the recipe given by the class. A computer program can create multiple unique instances of the same class in memory to represent many independent objects.

What Are Functions?

A function, a fundamental concept in computer programming, is a named block of instructions that performs a specific task. Functions allow programs to be modularized into reusable blocks of code that are written once and can then be invoked many times. In addition, modularization makes a program more readable and enables developers to work simultaneously on different portions of the same program.

What Are Methods?

A method is similar to a function, but it belongs to a specific class. In a computer program, all instances of a class contain the same methods. However, the methods of each instance can access only the instance’s own internal variables. Functions can interact with instances of a class by invoking its methods.
How Do You Use Packages?

There are two steps involved in using packages in the TSMODEL procedure. First, you specify in the REQUIRE statement the packages that you want to use in your SAS code. Second, you declare instances of package objects in your SAS code by using a DECLARE OBJECT statement for each instance. The following sections further elaborate on these steps.

The REQUIRE Statement

The REQUIRE statement in the TSMODEL procedure enables your custom SAS code to access the objects and functions in the packages that are specified in the statement. You can use specify multiple packages in a single REQUIRE statement, as follows:

```
REQUIRE package-1 < package-2 . . . package-n > ;
```

You can selectively identify which objects within a package you intend to use, as shown in the following statement:

```
REQUIRE package (class-1 . . . class-n ) ;
```

In contrast to objects, the functions offered by a package are always available when a package is registered via the REQUIRE statement; they do not have to be individually specified.

The REQUIRE statement does not automatically create any instances of the objects in the packages that you specify. For more information about the REQUIRE statement, see the Chapter 11, “The TSMODEL Procedure” (SAS Visual Forecasting: Forecasting Procedures).

The DECLARE OBJECT Statement

You use the DECLARE OBJECT statement in your custom SAS code to create one or more unique instances of package objects. Instances of the same object are completely independent from each other, and there is no limit on how many instances of an object you can create in your program. You can access package functions by simply invoking them directly from your custom SAS code, but you must declare instances of package objects in order to use them in your custom SAS code. You interact with instances of objects by invoking their methods, as is illustrated in the following statements:

```
DECLARE OBJECT name-1 (class ) ;
DECLARE OBJECT name-2 (class ) ;
rc1=name1.SomeMethod (Argument-1 . . . Argument-n ) ;
rc2=name2.SomeMethod (Argument-1 . . . Argument-n ) ;
```

These statements declare two unique instances of the class object: name1 and name2. You can interact with these two instances independently by invoking their SomeMethod method and passing in the required arguments. Method calls always return a numeric status code (rc) to indicate successful or failure. By convention, the objects in the TSMODEL procedure packages return a negative number to indicate that a computational failure has occurred.
Chapter Organization

This book is organized as follows.

This chapter provides an overview that describes what packages for the TSMODEL procedure are and how they are used.

The remaining chapters describe the various packages that are available for the TSMODEL procedure. Each of these chapters is organized as follows:

- The “Overview” section describes the broad purpose of the package; it describes which specific facet of the time series analysis problem the package is designed to tackle. It also provides a table that lists all the objects and functions that are offered by the package along with a brief description of each object’s functionalities. You can use the contents of this table to quickly locate an object that offers the functionalities you need. For packages that contain objects that interact with each other to perform larger and more complex tasks, a data flow diagram is also provided in this section to succinctly describe the interactions between the various objects via their methods. The direction of the arrows in this diagram indicates how the output of object methods feeds into other object methods. This section also describes the status codes that can be returned by an object methods and functions in the packages.

- A section for each object follows the “Overview” section. Sections for objects have the following structure:
  - The “Object Summary” section is divided into three parts: “Synopsis,” “Data Flow Diagram,” and “Methods.” Each object that is offered by a package has its own “Object Summary” section, which starts with a detailed description of the functionality of the object. Read this section if you want to understand more about the inner workings of an object (for example, to find out mathematical techniques that an object uses internally). This section also includes a table that summarizes all the available object methods along with a brief description of their functionalities. Refer to this table to quickly determine how to control the behavior of an object. You can also search for the methods that are prefixed with “Set” and “Get.” In addition, if the object is a collector, this section provides a table that describes the corresponding CAS table schema that the object generates. The table schema description includes the name, SAS data type, and a brief description of each CAS table column.
  - The “Synopsis” section shows how you can declare an instance of the object in your custom SAS code. This section describes the correct sequence of method calls (when a particular sequence is required) and indicates which method arguments are optional or mandatory. For example, for the FORENG object in the ATSM package, you provide forecast data to the FORENG object via its Initialize method, optionally set the forecast lead via the SetOption method, then run the forecast via its Run method, and optionally retrieve a forecast series via the GetForecast method. The “Synopsis” section also optionally offers a data flow diagram of the object that depicts the flow of data into and out of the object via its methods and the relevant data types that are required as arguments by each method. This diagram provides you with a visual overview of how to interact with the object from your program.
  - The “Methods” section provides a detailed description of the object’s methods. A subsection for each method includes its functionality and arguments. Method arguments are divided into either
input or output arguments. Input arguments are not modified by a method internally. Output arguments can be modified by a method internally. The size and value of an output argument after the execution of a method might differ from its original size and value before the method call, but the data type always remains the same. The description of each argument also lists the SAS data types that are acceptable for that argument. Each method argument can accept one or more SAS data types.

- The “Examples” section follows the detailed description of all the package’s objects. This section includes various examples that depict how the various objects provided by the package interact with each other and how they interact with objects in other packages to solve problems. These examples generally use almost all methods in each object and can provide a good starting point for your own custom implementations.

- The “References” section contains references for the methodology and for examples of the procedure.

Syntax Conventions

This section describes the common terminology and design patterns that are used by all packages that can be used with the TSMODEL procedure.

Object Names

Names of collector objects have the prefix “OUT” followed by a descriptive object name. For example, the OUTFREQ collector object is designed to store the results of an instance of the FREQ object in the TSA package. Similarly, the OUTTEST object is designed to store the time series model parameter estimates that are computed by an instance of the FORENG object in the ATSM package.

Similarly, names of repeater objects have the prefix “IN” followed by a descriptive object name. For example, the INDIAG object is used to read in the set of diagnostic control specifications that were previously stored into a CAS table by an instance of the OUTDIAG collector object in the ATSM package.

Object Method Names

Many of the objects in the TSMODEL procedure packages allow their behaviors to be controlled via the specification of options. For example, the FORENG object in the ATSM package provides automatic time series forecasting capabilities and enables you to specify the lead time, forecast horizon, and confidence level size of the computed forecast confidence limits, among many other options that affect the forecast. Names of object methods that are designed specifically for setting options have the prefix “Set” followed by the option name, as in the following example:

```
DECLARE OBJECT s (SFS) ;
rc=s.SetY (y ) ;
```
In addition, various objects offer a single overarching method called “SetOption” that is used to set all object properties, with the exception of the input time series data. To quickly determine how to control the behavior of objects, you can search for the description of an overarching “SetOption” method in the documentation of the objects that you plan to use in your custom SAS code. Similarly, names of object methods that are designed specifically to enable you to retrieve both data and properties from objects have the prefix “Get” followed by a descriptive name. For example, the “GetForecast” method in the FORENG object enables you to retrieve various forecast series that are computed by the object and include them in your custom SAS code for further processing.

Return Codes

All object methods and functions offered by the TSMODEL procedure packages return a status code upon their execution. Negative status codes are used to indicate the occurrence of computational failures. Positive return codes, including 0, are customized to the needs of individual objects and functions. Generally, a status code of 0 indicates unconditional success, and a status code greater than 0 indicates conditional success.
Chapter 3
Automatic Time Series Modeling Package

Contents

Overview: ATSM Package ......................................................... 13
ATSM Package Summary ....................................................... 13
Return Codes ................................................................. 16
TSDF Object ................................................................. 16
TSDF Synopsis ............................................................... 17
TSDF Methods .............................................................. 17
DIAGNOSE Object .......................................................... 24
DIAGNOSE Synopsis ....................................................... 24
DIAGNOSE Methods ......................................................... 25
FORENG Object .............................................................. 27
FORENG Synopsis .......................................................... 28
FORENG Methods .......................................................... 28
SCORE Object ................................................................. 35
SCORE Synopsis ............................................................. 36
SCORE Methods ............................................................. 36
DIAGSPEC Object ............................................................ 39
DIAGSPEC Synopsis ......................................................... 41
DIAGSPEC Methods ......................................................... 42
EVENT Object ................................................................. 56
EVENT Synopsis ............................................................ 57
EVENT Methods .............................................................. 57
SELSPEC Object .............................................................. 65
SELSPEC Synopsis .......................................................... 67
SELSPEC Methods .......................................................... 67
COMBSPEC Object .......................................................... 69
COMBSPEC Synopsis ....................................................... 72
COMBSPEC Methods ......................................................... 72
OUTDIAG Object .............................................................. 78
OUTDIAG Synopsis .......................................................... 78
OUTDIAG Methods .......................................................... 79
OUTEST Object ............................................................... 79
OUTEST Synopsis ........................................................... 82
OUTEST Methods ............................................................ 82
OUTEVENT Object ........................................................... 83
OUTEVENT Synopsis ........................................................ 84
<table>
<thead>
<tr>
<th>Object</th>
<th>Synopsis</th>
<th>Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>OUTEVENT Dummy</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OUTEVENT Dummy Synopsis</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OUTEVENT Dummy Methods</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OUTFOR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OUTFOR Synopsis</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OUTFOR Methods</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OUTCOMP</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OUTCOMP Synopsis</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OUTCOMP Methods</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OUTINDEP</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OUTINDEP Synopsis</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OUTINDEP Methods</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OUTMODELINFO</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OUTMODELINFO Synopsis</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OUTMODELINFO Methods</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OUTSELECT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OUTSELECT Synopsis</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OUTSELECT Methods</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OUTSTAT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OUTSTAT Synopsis</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OUTSTAT Methods</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OUTFMSG</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OUTFMSG Synopsis</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OUTFMSG Methods</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OUTSCORE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OUTSCORE Synopsis</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OUTSCORE Methods</td>
<td></td>
<td></td>
</tr>
<tr>
<td>INDIAG</td>
<td></td>
<td></td>
</tr>
<tr>
<td>INDIAG Synopsis</td>
<td></td>
<td></td>
</tr>
<tr>
<td>INDIAG Methods</td>
<td></td>
<td></td>
</tr>
<tr>
<td>INFMSG</td>
<td></td>
<td></td>
</tr>
<tr>
<td>INFMSG Synopsis</td>
<td></td>
<td></td>
</tr>
<tr>
<td>INFMSG Methods</td>
<td></td>
<td></td>
</tr>
<tr>
<td>INEST</td>
<td></td>
<td></td>
</tr>
<tr>
<td>INEST Synopsis</td>
<td></td>
<td></td>
</tr>
<tr>
<td>INEST Methods</td>
<td></td>
<td></td>
</tr>
<tr>
<td>INEVENT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>INEVENT Synopsis</td>
<td></td>
<td></td>
</tr>
<tr>
<td>INEVENT Methods</td>
<td></td>
<td></td>
</tr>
<tr>
<td>INEVENTBY</td>
<td></td>
<td></td>
</tr>
<tr>
<td>INEVENTBY Synopsis</td>
<td></td>
<td></td>
</tr>
<tr>
<td>INEVENTBY Methods</td>
<td></td>
<td></td>
</tr>
<tr>
<td>INSCORE</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Overview: ATSM Package

The automatic time series modeling (ATSM) package provides objects that are designed to support automatic
time series modeling and automatic forecasting. For more information about the statistical methodology
that underlies this package, see the chapters about the HPFDIAGNOSE, HPENGINE, HPFEVENTS, and
HPFSELECT procedures in SAS Forecast Server Procedures: User’s Guide. In addition, it is helpful to
review Chapter 11, “The TSMODEL Procedure” (SAS Visual Forecasting: Forecasting Procedures), and
Chapter 14, “Time Series Model Package.” Each of the objects in the ATSM package is designed to carry out
a particular task in the time series analysis process.

ATSM Package Summary

Table 3.1 summarizes the objects in the ATSM package.

<table>
<thead>
<tr>
<th>Object</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Time Series Modeling and Forecasting Objects</strong></td>
<td></td>
</tr>
<tr>
<td>DIAGNOSE</td>
<td>Time series model generator</td>
</tr>
<tr>
<td>FORENG</td>
<td>Time series forecasting engine</td>
</tr>
<tr>
<td>SCORE</td>
<td>Time series forecast from model score</td>
</tr>
<tr>
<td>TSDF</td>
<td>Time series data frame</td>
</tr>
<tr>
<td><strong>Objects for Controlling Model Identification</strong></td>
<td></td>
</tr>
<tr>
<td>COMBSPEC</td>
<td>Model combination list specification</td>
</tr>
<tr>
<td>DIAGSPEC</td>
<td>Time series model diagnosis control specification</td>
</tr>
<tr>
<td>EVENT</td>
<td>Time series event object</td>
</tr>
<tr>
<td>SELSPEC</td>
<td>Model selection list specification</td>
</tr>
<tr>
<td><strong>Collector Objects (Note 1)</strong></td>
<td></td>
</tr>
<tr>
<td>OUTCOMP</td>
<td>Time series model components collector</td>
</tr>
<tr>
<td>OUTDIAG</td>
<td>Persistent diagnostic control specifications</td>
</tr>
<tr>
<td>OUTEST</td>
<td>Time series model parameter estimates collector</td>
</tr>
<tr>
<td>OUTEVENT</td>
<td>Time series EVENT object event variable definition collector</td>
</tr>
</tbody>
</table>
### Table 3.1  
*continued*

<table>
<thead>
<tr>
<th>Object</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OUTEVENTDUMMY</td>
<td>Time series TSDF object event dummy variable collector</td>
</tr>
<tr>
<td>OUTFMSG</td>
<td>Persistent forecast model selection graph XML</td>
</tr>
<tr>
<td>OUTFOR</td>
<td>Time series model forecast collector</td>
</tr>
<tr>
<td>OUTINDEP</td>
<td>Time series model input series collector</td>
</tr>
<tr>
<td>OUTMODELINFO</td>
<td>Time series model information collector</td>
</tr>
<tr>
<td>OUTSCORE</td>
<td>Forecast score XML persistence</td>
</tr>
<tr>
<td>OUTSELECT</td>
<td>Time series model selection statistics collector</td>
</tr>
<tr>
<td>OUTSTAT</td>
<td>Time series model fit statistics collector</td>
</tr>
</tbody>
</table>

**Repeater Objects (Note 2)**

<table>
<thead>
<tr>
<th>Object</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>INDIAG</td>
<td>Replay of time series diagnostic control specifications</td>
</tr>
<tr>
<td>INFMSG</td>
<td>Forecast model selection graph (FMSG) repeater</td>
</tr>
<tr>
<td>INEST</td>
<td>Automatic time series model parameter estimates repeater</td>
</tr>
<tr>
<td>INEVENT</td>
<td>Event definition repeater</td>
</tr>
<tr>
<td>INEVENTBY</td>
<td>Add event repeater</td>
</tr>
<tr>
<td>INSCORE</td>
<td>Forecast model score repeater</td>
</tr>
</tbody>
</table>

**Notes:**

1. An ATSM collector object enables you to create a snapshot of results from the ATSM objects and save the results to a CAS table. Many of the CAS tables to which ATSM collector objects save results contain the same variables as their data set counterparts that are used by the HPFENGINE procedure. In some cases, these tables represent a superset of the columns in the corresponding PROC HPFENGINE data sets. All collector objects’ CAS tables are automatically appended with the BY variable columns that are specified in the procedure invocation.

2. ATSM repeater objects enable you to restore rows from a CAS table that was previously defined by a collector object to make them available for use by other ATSM objects. Repeater objects perform the inverse function of the collector objects. Not all ATSM collector objects have associated repeater objects. Repeater objects must be bound to an existing CAS table that is compatible with its purpose.

Figure 3.1 diagrams the relationships among the objects in the ATSM package. The object labeled TSM:XXXXSpec represents the following TSM package model specification objects: ARIMASPEC, ESMSPEC, UCMSPEC, IDMSPEC, and EXMSPEC. The object labeled OUTXXX represents the following ATSM package objects: OUTCOMP, OUTINDEP, OUTMODELINFO, and OUTSCORE.
**Figure 3.1 ATSM Data Flow**

**Diagram Description:**
- The diagram illustrates the data flow and method calls in the ATSM (Automatic Test System Management) package.
- Key components include:
  - **OUTDIAG**: An output table.
  - **INDIAG**: An input table.
  - **OUTDIAG**: A collection method.
  - **DIAGSPEC**: A diagnosis code.
  - **TSDF**: Time data.
  - **OUTFMSG**: An output message.
  - **INFMSG**: An input message.
  - **OUTXXX**: An output table.
  - **OUTFOR**: An output file.
  - **OUTSELECT**: An output selection.
  - **OUTSTAT**: An output statistic.
  - **INES**: An input event.
  - **OUTTEST**: An output test.
  - **INFIN**: An input file.
  - **FOREN**: A forensic event.

- The diagram shows the flow of data and method calls through these components, highlighting the interactions and dependencies.
- Key method calls include `SetDiagSpec`, `Collect`, and `Replay(*)`.
Return Codes

Table 3.2 shows the return code \((rc\) in method statements) status values that are used in this package. These status code values are returned after a method that is associated with an object is called; they can help determine whether the method executed successfully.

<table>
<thead>
<tr>
<th>Status</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt; 0</td>
<td>An unrecoverable error occurred. No result was produced.</td>
</tr>
<tr>
<td>= 0</td>
<td>Unconditional success. The requested action completed and a normal result was produced.</td>
</tr>
<tr>
<td>&gt; 0</td>
<td>Conditional success or warning. A result was produced subject to conditions.</td>
</tr>
</tbody>
</table>

TSDF Object

The TSDF object groups time series variables to be used as input for the other ATSM package objects. As such, TSDF instances are time series data frames.

Table 3.3 summarizes the methods that are associated with the TSDF object.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AddEvent</td>
<td>Add series defined by event definition</td>
</tr>
<tr>
<td>AddSeries</td>
<td>Add ancillary series</td>
</tr>
<tr>
<td>AddX</td>
<td>Add X series</td>
</tr>
<tr>
<td>AddY</td>
<td>Add Y series</td>
</tr>
<tr>
<td>GetSeries</td>
<td>Get series</td>
</tr>
<tr>
<td>Initialize</td>
<td>Initialize TSDF instance</td>
</tr>
<tr>
<td>ntid</td>
<td>Return the observation count of the time ID variable</td>
</tr>
<tr>
<td>ReplayEventby</td>
<td>Replay the INEVENTBY object to add events to the data frame</td>
</tr>
<tr>
<td>SetEventby</td>
<td>Set the eventby filter to the INEVENTBY object</td>
</tr>
<tr>
<td>SetOption</td>
<td>Set named option</td>
</tr>
</tbody>
</table>
DECLARE OBJECT obj (TSDF) ;

Method syntax, in order of typical usage:

\[ rc = \text{obj.Initialize}() ; \]
\[ rc = \text{obj.ReplayEventby} (\text{EventObj}, \text{IneventbyObj}) ; \]
\[ rc = \text{obj.AddY} (\text{YSeries} < 'Name', \text{Value}, \ldots >) ; \]
\[ rc = \text{obj.AddX} (\text{XSeries} < 'Name', \text{Value}, \ldots >) ; \]
\[ rc = \text{obj.SetEventby} (\text{IneventbyObj}) ; \]
\[ rc = \text{obj.AddEvent} (\text{EventObj}, \text{eventName} < 'Name', \text{Value}, \ldots >) ; \]
\[ rc = \text{obj.AddSeries} (\text{Series} < 'Name', \text{Value}>) ; \]
\[ rc = \text{obj.SetOption} ('Name', \text{Value} < 'Name', \text{Value}, \ldots >) ; \]
\[ rc = \text{obj.GetSeries} ('Name', \text{Result} < 'Name', \text{Value}>) ; \]
\[ \text{ntids} = \text{obj.ntid}() ; \]

**TSDF Methods**

### TSDF.AddEvent Method

\[ rc = \text{obj.AddEvent} (\text{EventObj}, \text{eventName} < 'Name', \text{Value}, \ldots >) ; \]

Adds an event to the TSDF instance. Each call to the AddEvent method adds the specified event variable to the TSDF instance. This method can be called as many times as needed to specify all the events that are needed. The specified events will be incorporated into models that use events.

#### Input Arguments
You must specify the following input arguments:

* **EventObj** specifies the object name that contains the event definition. *EventObj* must be declared and initialized prior to being specified in an AddEvent method.

* **eventName** specifies the name for the predefined event or the name of an event in the object, or you can specify _ALL_ to indicate all events in the object. _ALL_ adds all events in the event definition object subject to the values of the CLASS option. A numeric variable with the *eventName* is created for the TSDF instance. Only ARIMA and UCM models support events. If _ALL_ is specified, multiple events might be created. Both simple predefined events and complex (group) predefined events can be specified. If a predefined group event is specified, the multiple simple events that comprise the group event are created.

You can also specify one of the following *Names* and its associated *Value*:

* **'REQUIRED'** takes a string *Value* that specifies a variable selection hint for model diagnosis. This argument provides a hint about the importance of the event dummy variable when the TSDF is used in the DIAGNOSE object. For more information, see the HPFDIAGNOSE procedure in *SAS Forecast Server Procedures: User’s Guide*. This argument takes one of the following values:
MAYBE includes the event dummy variable unless it causes model fit failure and there are no numerical issues in the model fit.

NO includes the event dummy variable if it sufficiently improves the model’s information criterion.

YES includes the event dummy variable unless it causes model fit failure.

The default value is NO.

'POSITIVE' takes a Boolean Value that indicates whether to keep an event in a model. When the value of 'POSITIVE' is 1, the event is included in the model only if the parameter estimate is positive. Otherwise, the event is not included. The default value is 0.

'NEGATIVE' takes a Boolean Value that indicates whether to keep an event in a model. When the value of 'NEGATIVE' is 1, the event is included in the model only if the parameter estimate is negative. Otherwise, the event is not included. The default value is 0.

'CLASS' takes a string Value that specifies the type of events to create for the TSDF instance when _ALL_ is specified. Since a combination events and its composite simple events often cause conflicts in the model, this argument can be used to avoid those conflicts. For more information, see the HPFEVENTS procedure in SAS Forecast Server Procedures: User’s Guide. This argument takes one of the following values:

- DEFAULT excludes combination events.
- ALL includes all events.
- SIMPLE includes the simple events.
- COMBINATION includes the combination events.

The default value is DEFAULT, which excludes combination events. The default is similar to the behavior of an EVENT _ALL_; statement in the HPFDIAGNOSE procedure.

**TSDF.AddSeries Method**

```plaintext
rc=obj.AddSeries (Series <,'Name',Value,..>)
```

Adds an ancillary time series array (Series) for the TSDF instance. Ancillary series are used by other computational objects in the program flow. For example, you might include an EXMSPEC object (which is a part of the TSM package) in the model selection process for some FORENG object, where the EXMSPEC object references specific variables that supply the forecast-related series that are needed by the EXMSPEC object.

**Input Arguments**

You must specify the following input argument:

- **Series** specifies a numeric array that contains an ancillary series for the TSDF instance.

You can also specify the following 'Names' and their associated Values:

- **'ALIAS'** takes a string Value that specifies the name of the series in the TSDF instance. The default is the argument name.

- **'ZEROMISS'** takes a string Value that specifies how to interpret beginning and ending zero values in Series. You can specify one of the following Values:
NONE does not change any beginning or ending zeros.
LEFT sets all beginning zeros to missing, but leaves ending zeros unchanged.
RIGHT sets all ending zeros to missing, but leaves beginning zeros unchanged.
BOTH sets all beginning and ending zeros to missing.

The default is NONE.

**TSDF.AddX Method**

```plaintext
rc=obj.AddX (XSeries <,'Name',Value, . . . >) ;
```

Adds an independent time series array (XSeries) for the TSDF instance. Each call to the TSDF.AddX method adds the specified XSeries variable to the TSDF instance. You can call this method as many times as you need to in order to specify all the independent variables that you need. You can use any method for forecasting the independent series and supply all nonmissing values for it.

**Input Arguments**

You must specify the following input argument:

`XSeries` specifies a numeric array that contains an independent series for the TSDF instance. When this method is used with user-defined models, the name of the XSeries variable must match the name of an input symbol (predictor) in any user-defined time series model. Only ARIMA and UCM models support predictors.

You can also specify one of the following 'Names' and its associated Value:

- 'ALIAS' takes a string Value that specifies an alias for XSeries in lieu of the array name for the TSDF instance.
- 'CONTROL' takes a Boolean Value that indicates whether the variable is controllable in forecast functions. The default value is 0.
- 'EXTEND' takes a string Value that specifies an independent series forecast method for extension or replacement. If you supply nonmissing values, it does not matter how you specify this argument; the independent series is always used as specified. This argument controls the generation of future values for the independent variable; it takes one of the following Values:
  - AVERAGE uses the average of historical values.
  - FIRST uses the first nonmissing value.
  - LAST uses the last nonmissing value.
  - MAX uses the maximum value.
  - MEDIAN uses the median value.
  - MIN uses the minimum value.
  - NONE uses as specified.
Chapter 3: Automatic Time Series Modeling Package

STOCHASTIC uses the forecast values of the best suited exponential smoothing model. For more information, see the STOCHASTIC statement in the HPFENGINE statement in SAS Forecast Server Procedures: User’s Guide.

The default is STOCHASTIC.

'KEEP' takes a Boolean Value that indicates whether to keep a variable if it is referenced in a model. This argument forces the inclusion of the independent variable when it is referenced from a model in the FORENG object during model selection. When the value of 'KEEP' is 0, independent variables that might cause a model fit to fail are selectively removed from the FORENG model selection process. When the value of 'KEEP' is 1, the independent variable is always kept and a model fit failure excludes the model from consideration during model selection. The 'KEEP' argument functions the same as the REQUIRED= option for PROC HPFENGINE. The default value is 0.

'NODIFF' takes a Boolean Value that indicates whether the variable automatically follows Y differencing. The default value is 0 (the variable does not automatically follow differencing).

'REPMISS' takes a Boolean Value that controls whether embedded missing values in the historical region of the independent series are replaced via the 'EXTEND' argument. The default value is 1 (missing values are replaced).

'REQUIRED' takes a string Value that specifies a variable selection hint for model diagnosis. This argument provides a hint about the importance of the independent variable when the TSDF object is used in the DIAGNOSE object. For more information, see the REQUIRED option in the HPFDIAGNOSE procedure in SAS Forecast Server Procedures: User’s Guide. You can specify one of the following Values:

- MAYBE includes the independent variable unless it causes model fit failure and there are no numerical issues in the model fit.
- NO includes the independent variable if it sufficiently improves the model’s information criterion.
- YES includes the independent variable unless it causes model fit failure.

The default is NO.

'SIGN' takes a string Value that specifies whether the variable should be dropped from the model based on the sign of its computed regression coefficient. You can specify one of the following Values:

- IGNORE ignores the sign of the regression coefficient of the variable in the model.
- NEGATIVE retains the variable in the model if its regression coefficient is negative.
- POSITIVE retains the variable in the model if its regression coefficient is positive.

The default is IGNORE.

'ZEROMISS' takes a string Value that specifies how to interpret beginning and ending zero values in XSeries. You can specify one of the following Values:
NONE does not change any beginning or ending zeros.
LEFT sets all beginning zeros to missing, but leaves ending zeros unchanged.
RIGHT sets all ending zeros to missing, but leaves beginning zeros unchanged.
BOTH sets all beginning and ending zeros to missing.

The default is NONE.

**TSDF.AddY Method**

```c
rc=obj.AddY (YSeries <,'Name',Value,...>);
```

Adds a dependent time series array (YSeries) for the TSDF instance.

**Input Arguments**

You must specify the following input argument:

*YSeries* specifies a numeric array that is used to specify the dependent series for the TSDF instance.

You can also specify the following *Names* and their associated *Values*:

- **'REPMISS'** takes a Boolean *Value* that when set to 1, replaces missing values over the fit range with values that are obtained by applying the method specified in the TSDF.AddX method. For more information, see the REPLACEMISSING option in the HPFENGINE procedure in *SAS Forecast Server Procedures: User’s Guide*. The default value is 1.
- **'ZEROMISS'** takes a string *Value* that specifies how to interpret beginning and ending zero values in YSeries. You can specify one of the following *Values*:
  - NONE does not change any beginning or ending zeros.
  - LEFT sets all beginning zeros to missing, but leaves ending zeros unchanged.
  - RIGHT sets all ending zeros to missing, but leaves beginning zeros unchanged.
  - BOTH sets all beginning and ending zeros to missing.

The default is NONE.

**TSDF.GetSeries Method**

```c
rc=obj.GetSeries ('Name', Result <,'Name',Value>);
```

Retrieves the specified series by its name (*Name* or alias) from the TSDF instance and stores it in the specified numeric array (*Result*). You can optionally specify that the retrieved series be automatically extended forward in time. The length of the extended series depends on the values of the 'LEAD' and 'HORIZON' options that are set via the SetOption method and also on the length of the dependent series that is set via the AddY method (up to the last nonmissing value). Only independent variables added via the AddX method and events added via the AddEvent method can be extended forward in time. For independent variables, you can specify the method used to extended the series via the 'EXTEND', 'REPMISS', and 'ZEROMISS' options of the AddX method.
Chapter 3: Automatic Time Series Modeling Package

Input Arguments
You must specify the following input argument:

'Name' specifies a character string that contains the variable to be returned.

You can also specify the following 'Name' and a Value for it:

'ADJUST' takes a string Value that specifies whether the retrieved series should be automatically extended forward in time. Only independent and event series can be extended. This option is silently ignored otherwise. You can specify one of the following Values:

YES retrieve an extended version of the series.
NO retrieve the original series.

The default is NO.

Output Arguments
You must specify the following output argument:

Result specifies a numeric array to receive the variable’s series.

TSDF.Initialize Method

\[ rc = obj.\text{Initialize}() ; \]

Initializes a TSDF instance to an empty state. This method must be called before the time series arrays and other attributes are specified for the TSDF instance.

Arguments
There are no arguments associated with this method.

TSDF.ntid Method

\[ ntids = obj.\text{ntid}() ; \]

Returns the length (observation count) of the time ID variable for the TSDF instance. A missing value indicates that the TSDF instance has no time ID variable.

Arguments
There are no arguments associated with this method.

TSDF.ReplayEventby Method

\[ rc = obj.\text{ReplayEventby}(\text{EventObj, IneventbyObj}) ; \]

Specifies an EVENT object and INEVENTBY object to create events for the TSDF instance for each BY group. The ReplayEventby method adds all events associated with the BY group to the TSDF instance. If a DIAGNOSE Object is specified without the SetEventby method being specified, then all the events associated with the BY group are candidates for the model. If a DIAGNOSE object is specified and the SetEventby method is specified, then the events associated with the Y variable specified in the SetY method and BY group are candidates for the model.
**Input Arguments**

You must specify the following input arguments:

- **EventObj** specifies the object name that contains the event definition. *EventObj* must be declared and initialized prior to being specified in a ReplayEventby method.

- **IneventbyObj** specifies the INEVENTBY object that provides a mapping between the Y series identified by BY group and series name and the event names. Since the ReplayEventby method is adding events to the TSDF object, the Y series mapping is ignored by ReplayEventby, and all events associated with the BY group are added to the TSDF instance for that BY group. See SetEventby for information on mapping the events to a Y series.

**TSDF.SetEventby Method**

```plaintext
rc = obj.SetEventby (IneventbyObj ) ;
```

Specifies an INEVENTBY object for the TSDF instance. When the DIAGNOSE Object receives the data frame, the events will be filtered using the information in the INEVENTBY object.

**Input Arguments**

You must specify the following input argument:

- **IneventbyObj** specifies the INEVENTBY object that provides a mapping between the Y series identified by BY group and series name and the event names. The events specified with the BY group and series name are candidates for inclusion in models considered by the DIAGNOSE object. The events are filtered based on the information in the SetY method of the TSDF object and the BY variables, the _NAME_ variable, and the _EVENT_ variable in the table specified by the INEVENTBY object.

**TSDF.SetOption Method**

```plaintext
rc = obj.SetOption ('Name', Value <,'Name',Value,.. >) ;
```

Specifies named options for the TSDF instance.

**Input Arguments**

You must specify at least one of the following 'Names' and its associated Value:

- **'HORIZON'** takes a numeric Value that specifies the forecast horizon reference time. When set to missing value, the forecast horizon reference time is automatically set as the first time period following the last nonmissing observation of the dependent series. The default value is missing value.

- **'LEAD'** takes a nonnegative integer Value that specifies the forecast lead. The default value is the value of the LEAD= option in the PROC TSMODEL statement.

- **'SEASONALITY'** takes a positive integer Value that specifies the seasonal cycle length. By default, SEASONALITY is inferred from the INTERVAL= option that you specify in the procedure invocation.
**DIAGNOSE Object**

The DIAGNOSE object generates (diagnoses) time series models for a time series data. The DIAGNOSE object can be specified as input to perform automatic model selection and forecasting in the forecasting engine (FORENG) object.

Table 3.4 summarizes the methods that are associated with the DIAGNOSE Object.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initialize</td>
<td>Initialize the DIAGNOSE object</td>
</tr>
<tr>
<td>nmodels</td>
<td>Get the number of model families diagnosed</td>
</tr>
<tr>
<td>Replay</td>
<td>Replay the diagnostic control specification</td>
</tr>
<tr>
<td>Run</td>
<td>Perform model generation</td>
</tr>
<tr>
<td>SetOption</td>
<td>Set named option</td>
</tr>
<tr>
<td>SetSpec</td>
<td>Set diagnostic controls</td>
</tr>
</tbody>
</table>

**DIAGNOSE Synopsis**

```plaintext
DECLARE OBJECT obj (DIAGNOSE) ;
```

Method syntax, in order of typical usage:

```plaintext
rc=obj.Initialize (TSDFObject ) ;
rc=obj.SetSpec (DIAGSPECObject) ;
rc=obj.Replay (INDIAGOObject) ;
rc=obj.Run () ;
rc=obj.SetOption ('Name', Value <,'Name', Value,...) ;
nmodels=obj.nmodels () ;
```
DIAGNOSE Methods

DIAGNOSE.Initialize Method

\[ rc = \text{obj}.\text{Initialize} \left( TSDFObj \right) ; \]

Initializes a DIAGNOSE instance and specifies the time series data for the DIAGNOSE instance.

**Input Arguments**

You must specify the following input argument:

- **TSDFObj** specifies the TSDF object that holds the time series data to be diagnosed by the DIAGNOSE instance. This method retains only a reference to the TSDF object’s time series data frame and does not make a deep copy of its time series data.

DIAGNOSE.nmodels Method

\[ \text{nmodels} = \text{obj}.\text{nmodels} () ; \]

Returns the number of models that are generated from the DIAGNOSE.Run method. A missing value indicates that a DIAGNOSE.Run method has not been successfully completed since the last call to the DIAGNOSE.Initialize method.

**Arguments**

There are no arguments associated with this method.

DIAGNOSE.Replay Method

\[ rc = \text{obj}.\text{Replay} \left( \text{INDIAGOBJ} \right) ; \]

Restores a previously stored diagnostic control specification from the specified INDIAGOBJ. This method establishes the diagnostic control settings in a model identification run of the DIAGNOSE instance. For more information about storing diagnostic control specifications, see the section “OUTDIAG Object” on page 78.

**Input Arguments**

You must specify the following input argument:

- **INDIAGOBJ** specifies the INDIAG instance that defines the source of the diagnostic control specification to restore.

DIAGNOSE.Run Method

\[ rc = \text{obj}.\text{Run} () ; \]

Runs model diagnosis for the time series data frame that is specified for the DIAGNOSE instance.

**Table 3.5** shows the model families that might be considered during the diagnostic process.
Table 3.5  Model Families for the DIAGNOSE.Run Method

<table>
<thead>
<tr>
<th>Family</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARIMAX</td>
<td>ARIMA model that includes predictors that use ARIMA-REG identification order</td>
</tr>
<tr>
<td>ESM</td>
<td>Seasonal and nonseasonal exponential smoothing models</td>
</tr>
<tr>
<td>IDM</td>
<td>Intermittent demand (IDM) model</td>
</tr>
<tr>
<td>REGARIMA</td>
<td>ARIMA model that includes predictors that use REG-ARIMA identification order</td>
</tr>
<tr>
<td>UCM</td>
<td>UCM model with predictors</td>
</tr>
</tbody>
</table>

By default, the ARIMAX and ESM families are considered.

Arguments
There are no arguments associated with this method.

DIAGNOSE.SetOption Method

rc=obj.SetOption('Name', Value < ', 'Name', Value, ... );

Specifies named options for the DIAGNOSE instance.

Input Arguments
You must specify at least one of the following 'Names' and its associated Value:

'BACK'  takes a nonnegative integer Value that specifies the back region for model performance. If BACK=n and the number of observations is T, then the first T – n observations are used to diagnose a series. The default is value 0.

'CRIERION'  takes a string Value that specifies the selection statistic mnemonic. The default is 'RMSE'. For a list of valid values, see the CRITERION= option in the HPFDIAGNOSE procedure in SAS Forecast Server Procedures: User's Guide.

'HOLDOUT'  takes a nonnegative integer Value that specifies the holdout region for model selection. The default value is 0.

'HOLDOUTPCT'  takes a numeric Value between 0 and 100 that specifies the size of the holdout sample as a percentage of the length of the dependent time series. If HOLDOUT=5 and HOLDOUTPCT=10, the size of the holdout sample is min(5, 0.1T), where T is the length of the dependent time series after the beginning and ending missing values are removed. The default value is a missing value.

'MINOBSTREND'  takes a numeric Value greater than 0 that specifies the minimum number of nonmissing observations needed for a trend model to be fitted to any series. The default value is 1.

'MINOBSEASON'  takes a numeric Value greater than 0 that specifies the minimum number of nonmissing observations needed for a seasonal model to be fitted to any series. The default value is 2.
**DIAGNOSE.SetSpec Method**

```cpp
rc=objc.SetSpec (DIAGSPECObject) ;
```

Specifies diagnostic control options for the DIAGNOSE instance. Modified control settings in the `DIAGSPECObject` are copied into the control settings for the DIAGNOSE instance for use by the next DIAGNOSE.Run method call.

**Input Arguments**

You must specify the following input argument:

- `DIAGSPECObject` specifies the DIAGSPEC instance that defines the diagnostic options to be used by the DIAGNOSE instance.

---

**FORENG Object**

The FORENG object automates time series model selection and forecasting. The FORENG object can be used with the DIAGNOSE object and with various model specification objects from the TSM package.

Table 3.6 summarizes the methods that are associated with the FORENG object.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AddFrom</td>
<td>Add models to the FORENG instance</td>
</tr>
<tr>
<td>criterion</td>
<td>Get final forecast fit statistic</td>
</tr>
<tr>
<td>GetForecast</td>
<td>Get forecast series</td>
</tr>
<tr>
<td>GetXSeries</td>
<td>Get X series that is used in final forecast</td>
</tr>
<tr>
<td>Initialize</td>
<td>Initialize the FORENG object with a DIAGNOSE object</td>
</tr>
<tr>
<td>Initialize</td>
<td>Initialize the FORENG object with a TSDF object</td>
</tr>
<tr>
<td>model</td>
<td>Get the name of the selected model</td>
</tr>
<tr>
<td>nfor</td>
<td>Get forecast series length</td>
</tr>
<tr>
<td>Replay</td>
<td>Replay models and parameter estimates for the FORENG instance</td>
</tr>
<tr>
<td>Run</td>
<td>Run automatic model selection and forecast</td>
</tr>
<tr>
<td>SetOption</td>
<td>Set named option</td>
</tr>
</tbody>
</table>
FORENG Synopsis

DECLARE OBJECT obj (FORENG) ;

Method syntax, in order of typical usage:

\[ rc = obj.\text{Initialize} \left( \text{TSDFObj ect} \right) ; \]
\[ rc = obj.\text{Initialize} \left( \text{DIAGNOSEObject} \right) ; \]
\[ rc = obj.\text{AddFrom} \left( \text{specObject} \right) ; \]
\[ rc = obj.\text{Replay} \left( \text{infmsgObject} \lt .\text{inestObject} \gt \right) ; \]
\[ rc = obj.\text{Run} () ; \]
\[ rc = obj.\text{GetForecast} \left( \text{Which}, \text{Result} \right) ; \]
\[ rc = obj.\text{GetXSeries} \left( \text{WhichX}, \text{XSeries} \right) ; \]
\[ rc = obj.\text{SetOption} \left( 'Name', Value < ,'Name', Value, ... > \right) ; \]
\[ model = obj.\text{model} () ; \]
\[ nfor = obj.\text{nfor} () ; \]
\[ criterion = obj.\text{criterion} () ; \]

FORENG Methods

FORENG.AddFrom Method

\[ rc = obj.\text{AddFrom} \left( \text{specObject} \right) ; \]

Adds models from a source instance into the FORENG object’s model selection graph.

For a model selection list, the list’s models are appended to the FORENG instance’s root selection list. For a model combination list, the combination is appended to the FORENG instance’s root selection list.

Calling FORENG.AddFrom subsequent to a FORENG.Run instance results in a replay that augments the restored model selection graph with the models from the model specification object that was passed as an argument to the AddFrom method. Consequently, this also forces the FORENG.Run method to re-execute the model selection step, followed by the parameter estimation step for the selected model, and finally the forecast step.

Input Arguments

You must specify the following input argument:

\[ \text{specObject} \quad \text{specifies the source of models to add. Models can be included from the objects shown in Table 3.7.} \]
Table 3.7  Model Objects for the FORENG.AddFrom Method

<table>
<thead>
<tr>
<th>specObject</th>
<th>Package</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARIMASPEC</td>
<td>TSM</td>
<td>Includes the ARIMA model specification</td>
</tr>
<tr>
<td>COMBSPEC</td>
<td>ATSM</td>
<td>Includes the combination model specification</td>
</tr>
<tr>
<td>ESMSPEC</td>
<td>TSM</td>
<td>Includes the exponential smoothing model specification</td>
</tr>
<tr>
<td>IDMSPEC</td>
<td>TSM</td>
<td>Includes the IDM model specification</td>
</tr>
<tr>
<td>SELSPEC</td>
<td>ATSM</td>
<td>Includes the model selection specification</td>
</tr>
<tr>
<td>TSM</td>
<td>TSM</td>
<td>Include the TSM model specification</td>
</tr>
<tr>
<td>UCMSPEC</td>
<td>TSM</td>
<td>Includes the UCM model specification</td>
</tr>
</tbody>
</table>

FORENG.criterion Method

```
criterion = obj.criterion () ;
```

Returns the fit statistic value for the final forecast for the FORENG instance. The criterion is set via the FORENG.SetOption method. A missing value indicates that the FORENG instance has not produced a successful forecast.

**Arguments**
There are no arguments associated with this method.

FORENG.GetForecast Method

```
rc = obj.GetForecast (Which, Result) ;
```

Gets the specified forecast series (Which) from the FORENG instance and stores it in the specified numeric array (Result).

**Input Arguments**
You must specify the following input argument:

`Which` is a case-insensitive character string that specifies the forecast series to return. You can specify one of the following values:

- **ERROR** returns prediction errors.
- **LOWER** returns lower confidence limit series.
- **STDERR** returns a prediction standard error series.
- **PREDICT** returns prediction series.
- **UPPER** returns upper confidence limit series.
Output Arguments
You must specify the following output argument:

Result specifies a numeric array to receive the forecast series.

FORENG.GetXSeries Method

\[ rc = obj.GetXSeries(WhichX, XSeries) ; \]

Gets the specified X series (WhichX) from the FORENG instance and stores it in the specified numeric array (XSeries).

Input Arguments
You must specify the following input argument:

WhichX specifies a case-sensitive character string that contains the name of the X series to return. If the specified X variable is not available in the FORENG instance, a failure status is returned. If the specified X variable is available and not used in the final forecast, the X series is returned and a warning status is returned.

Output Arguments
You must specify the following output argument:

XSeries specifies a numeric array to receive the X series.

FORENG.Initialize Method (Using a TSDF Object)

\[ rc = obj.Initialize(TSDFOBJECT) ; \]

Initializes the FORENG instance with the time series data frame to supply the data that it will use to forecast. Models can be added via the FORENG.AddFrom method prior to calling the FORENG.Run method. If no models are added before calling FORENG.Run, the forecast is generated using the best candidate exponential smoothing model (ESMBEST). See the METHOD argument in the “DIAGSPEC.SetESM Method” on page 50 for details regarding how the ESMBEST model is selected.

Input Arguments
You must specify the following input argument:

TSDFOBJECT specifies the TSDF object that holds the time series data to be forecast by the FORENG instance.
FORENG.Initialize Method (Using a DIAGNOSE Object)

\[ rc = obj.\text{Initialize} \left( \text{DIAGNOSEObject} \right) ; \]

Initializes the FORENG instance to use the generated models and time series data frame from the specified DIAGNOSE instance.

**Input Arguments**

You must specify the following input argument:

- **DIAGNOSEObject** specifies the DIAGNOSE object that holds the time series data and generated model information that are used to initialize the FORENG instance.

FORENG.model Method

\[ model = obj.\text{model} () ; \]

Returns the name of the model that was selected by the FORENG instance. A zero-length name indicates that the FORENG instance has not produced a successful forecast.

**Arguments**

There are no arguments associated with this method.

FORENG.nfor Method

\[ nfor = obj.\text{nfor} () ; \]

Returns the length (observation count) of the forecast series for the FORENG instance. A missing value indicates that the FORENG instance has not produced a successful forecast.

**Arguments**

There are no arguments associated with this method.

FORENG.Replay Method

\[ rc = obj.\text{Replay} \left( \text{infmsgObject} < , \text{inestObject} > \right) ; \]

Restores a previously stored forecast model selection graph (FMSG) from the specified \text{infmsgObject}. This method establishes the context to allow for a model selection and forecasting run of the FORENG instance. Optionally, you can specify \text{inestObject} that supplies previously stored parameter estimates for the restored FMSG. The \text{inestObject} determines the selected path (model set) from the restored FMSG and supplies the selected models with parameter estimates that are fixed for forecast-only runs or that serve as initial values for update runs. This method effectively provides the capability that is inherent in the TASK= modes in the HPFENGINE procedure. If you restore a previously saved FMSG, you can call the FORENG.AddFrom method to augment the restored model set with other models. When you restore the FMSG with a parameter set (INEST instance), the restored FMSG is pruned to reflect the selected path (model set), based on the parameter set. In either case, calling FORENG.AddFrom subsequent to a FORENG.Replay method always forces a model selection mode (TASK=SELECT) for the FORENG object. For more information about storing FMSG specifications, see the section “OUTFMSG Object” on page 102; for more information about storing FMSG model parameter estimates, see the section “OUTEST Object” on page 79.
Input Arguments
You must specify the following input argument:

infmsgObject specifies the source of the FMSG XML to restore. By default, restoring only the FMSG forces a model selection mode for the FORENG.Run method.

You can also specify the following input argument:

inestObject specifies the source of the model parameter estimates to restore. This both subsets the FMSG model set to those models that include parameter estimates and provides those selected models with their parameter estimates. By default, restoring both FMSG and parameter estimates forces a forecast-only mode for the FORENG. You can change this by using the FORENG.SetOption method.

FORENG.Run Method

\[ rc = obj.Run(); \]

Runs model selection for the time series data frame that is specified for the FORENG instance. The FORENG instance model selection is driven by its FMSG content. Model selection is followed by a final forecast that uses the best performing model from the model selection step.

Arguments
There are no arguments associated with this method.

FORENG.SetOption Method

\[ rc = obj.SetOption('Name', Value <,'Name', Value, ... >); \]

Specifies named options for the FORENG instance.

Input Arguments
You must specify at least one of the following 'Names' and its associated Value:

- **'ALPHA'**
  - takes a numeric Value between 0 and 1, exclusive, that specifies the significance level for forecast confidence bands. The default value is 0.05.

- **'BACK'**
  - takes a nonnegative integer Value that specifies the number of observations before the end of the data where the multistep forecasts are to begin. This option is often used to obtain performance statistics. The default value is 0.

- **'COMPINTEGRATE'**
  - takes a Boolean Value that when set to 1, forces the generated component series to sum to the forecast series. This option affects only the components that are produced by ARIMA models and include differencing. The default value is 0.

- **'CHOOSE'**
  - takes a case-insensitive string Value that specifies the name of the winning model. You can disable this option by specifying a missing value (.). The default value is a missing value.

  When this option is specified, then only the winning model that is specified by Value is evaluated as part of the model selection process. All other competing models that are present in the FORENG object’s forecast message graph (FMSG)
are ignored. Consequently, the OUTSELECT object’s CAS table, which usually contains one row of model selection statistics per competing model, will contain only a single row that corresponds to the model selection statistics of the specified winning model.

The winning model name specified by Value is automatically propagated to the XML string that is generated from the contents of the FORENG object’s FMSG. This XML string is stored in the OUTFMSG object’s CAS table. When you replay the model selection list into the FORENG object via the FORENG.Replay method, the winning model name that is present in the replayed model selection list will automatically become the default value of the ‘CHOOSE’ option in the FORENG object.

When you invoke the FORENG.Replay method and pass in valid INFMSG and INEST object instances as arguments, the FORENG object automatically enters into a forecast mode. In this mode, the name of the winning model is extracted directly from the INEST object, which causes the model selection process to be skipped and the value of the ‘CHOOSE’ argument to be ignored. However, if the FORENG object’s FMSG is augmented with new models via calls to the FORENG.AddFrom method that are performed after the FORENG.Replay method has been called, then the FORENG object will automatically enter into a model selection mode. In this mode, the value of the ‘CHOOSE’ argument becomes relevant once again because the INEST object no longer dictates the winning model.

'CRIERION' takes a string Value that specifies the model selection criterion (statistic of fit) for selecting from several candidate models. The default is RMSE. For a list of valid values, see the CRITERION= option in the HPFDIAGNOSE procedure in SAS Forecast Server Procedures: User’s Guide.

'ENDZEROS.MAXNUM' takes a numeric Value greater than 1 that specifies the maximum allowed number of trailing zero values in the dependent series in order for a nonzero model to be considered. This determination is part of the _ZERO_ model test, which determines whether to produce an all-zero forecast when trailing zeros are present in the dependent series. If the number of trailing zero observations in the series is less than or equal to Value, then a nonzero model is considered. For example, if you specify Value as 10, then a nonzero model is considered only if the dependent series has 10 or fewer trailing zero values. The default value is the length of the dependent series.

'ENDZEROS.MAXPCT' takes a numeric Value (0 < Value ≤ 100) that specifies the maximum allowed percentage of trailing zero values in the dependent series, relative to the number of nonzero values in the series, in order for a nonzero model to be considered. This determination is part of the _ZERO_ model test, which determines whether an all-zero forecast should be produced when trailing zeros are present in the dependent series. For example, if you specify Value as 20, then a nonzero model is considered only if the number of trailing zero values is less than or equal to 20% of the number of nonmissing and nonzero values of the entire series. The default value is 100.

'ENDZEROS.MINOBS' takes a numeric Value greater than 0 that specifies the minimum series length that is required to perform the _ZERO_ model test, which determines whether an all-zero forecast should be produced when trailing zeros are present in the dependent
series. For example, if you specify Value as 8, then the _ZERO_ model test is not performed on the dependent series if its length is fewer than eight observations. The default value is 0 (that is, no minimum length exclusion).

'FCST.BD.LOWER' takes a numeric Value that specifies a lower bound for the forecast. Forecast values that fall below the specified value are truncated to Value and the lower confidence limit of the forecast is correspondingly shifted up. A missing value indicates that no lower bound truncation should occur. The default is a missing value.

'FCST.BD.UPPER' takes a numeric Value that specifies an upper bound for the forecast. Forecast values that fall above the specified value are truncated to Value and the upper confidence limit of the forecast is correspondingly shifted down. A missing value indicates that no upper bound truncation should occur. The default is a missing value.

'HOLDOUT' takes a nonnegative integer Value that specifies the size of the holdout sample to be used for model selection. The holdout sample is a subset of actual time series that ends at the last nonmissing observation. The default value is 0.

'HOLDOUTPCT' takes a numeric Value between 0 and 100 that specifies the size of the holdout sample as a percentage of the length of the dependent time series. If HOLDOUT=5 and HOLDOUTPCT=10, the size of the holdout sample is \(\min(5, 0.1T)\), where \(T\) is the length of the dependent time series after the beginning and ending missing values are removed. The default value is a missing value.

'HORIZON' takes a numeric Value that specifies the forecast horizon reference time. When set to missing value, the forecast horizon reference time is automatically set as the first time period following the last nonmissing observation of the dependent series. The default value is the value of the 'HORIZON' option specified via the TSDF.SetOption method.

'LEAD' takes a nonnegative integer Value that specifies the forecast lead. The default value is the value of the 'LEAD' option specified via the TSDF.SetOption method.

'MINOBS.MEAN' takes a numeric Value greater than 0 such that any series that has fewer than Value nonmissing values is not fit using the models in the selection list, but instead is forecast as the mean of the observations in the series. The default value is 1.

'MINOBS.TREND' takes a numeric Value greater than 0 that specifies the minimum number of nonmissing observations needed for a trend model to be fitted to any series. The default value is 1.

'MINOBS.SEASON' takes a numeric Value greater than 0 that specifies the minimum number of nonmissing observations needed for a seasonal model to be fitted to any series. The default value is 2.

'SCORE' takes a Boolean Value that when set to 1, produces the score XML that is required by the SCORE and OUTSCORE objects to function properly. The default value is 0.

'TASK' takes a string Value that directs the automatic forecasting process of the FORENG object. The FORENG object is a robust system that was built to deliver automatic forecasts with minimal user intervention. This argument gives you some control over the automatic forecasting process. You can specify the following Values:
FIT causes the parameter set of the selected model to be estimated based on the time series data that are currently available to the FORENG object. The forecast that is subsequently produced is based on the new estimated parameters. No model selection occurs.

UPDATE is the same as FIT, but causes the restored parameter set to be used as starting values for the ensuing parameter estimation.

A forecast is always generated regardless of the specified Value. Note that both Values require as a prerequisite the successful execution of the FORENG.Replay method with both INFMSG and INEST objects passed in as arguments. Otherwise, the FORENG.Run method ignores the request and proceeds to automatically determine the steps required in order to successfully produce a reasonable forecast.

There are various reasons why the FORENG object might automatically override the Value you specify. For example, if you replay parameter estimates into the FORENG object using the FORENG.Replay method, but then subsequently invoke the FORENG.AddFrom method to insert new model specifications into the FORENG object, then the FORENG.Run method determines that its forecast message graph has changed and forces the model selection process to start over regardless of the specified Value. Similarly, the FORENG object can automatically determine when it can generate a better model, given the current data set, than the model that was provided by the replayed parameter estimates. In this scenario, the FORENG object forces the model selection process to start over in lieu of the improved predictions that it expects to achieve. This feature enables the FORENG object to continuously improve its predictions with minimal user supervision.

SCORE Object

The SCORE object generates a forecast by using the score (saved state) from a FORENG object instance. Table 3.8 summarizes the methods that are associated with the SCORE object.
Table 3.8  Methods of the SCORE Object

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>convar</td>
<td>Get name of variables in SCORE that can be controlled by the model</td>
</tr>
<tr>
<td>GetForecast</td>
<td>Get forecast series</td>
</tr>
<tr>
<td>Initialize</td>
<td>Initialize SCORE instance from the score context in the specified FORENG object</td>
</tr>
<tr>
<td>Initialize</td>
<td>Initialize SCORE instance from the score context in the specified INSCORE object</td>
</tr>
<tr>
<td>ncon</td>
<td>Specify the number of controllable independent variables in score</td>
</tr>
<tr>
<td>nfor</td>
<td>Specify the forecast series length</td>
</tr>
<tr>
<td>Run</td>
<td>Run SCORE instance</td>
</tr>
<tr>
<td>SetControl</td>
<td>Set controllable series</td>
</tr>
<tr>
<td>SetOption</td>
<td>Set named option</td>
</tr>
</tbody>
</table>

SCORE Synopsis

DECLARE OBJECT obj (SCORE) ;

Method syntax, in order of typical usage:

```plaintext
rc=obj.Initialize (FORENGObject) ;
rc=obj.Initialize (INSCOREObject <,YName>) ;
rc=obj.Run () ;
rc=obj.SetControl (XSeries <,SymbolName>) ;
rc=obj.SetOption ('Name', Value <,'Name', Value,. . . >) ;
rc=obj.GetForecast (Which, Result) ;
ncon=obj.ncon () ;
nfor=obj.nfor () ;
xname=obj.convar (First) ;
```

SCORE Methods

SCORE.convar Method

```plaintext
xname=obj.convar (First) ;
```

Returns the name of the next controllable X variable in the SCORE instance. A zero-length name indicates that either the SCORE instance has not been initialized or that it has no more controllable variables to list.

**Input Arguments**

You must specify the following input argument:
First specifies a numeric variable to control iteration of names. When First is equal to 1, the iteration is reset and the name of the first controllable variable is returned. When First is not equal to 1, the name of the next controllable variable is returned.

**SCORE.GetForecast Method**

```plaintext
rc = obj.GetForecast (Which, Result);
```

Retrieves the specified forecast component series (Which) from the SCORE instance and saves it in the specified numeric array (Result).

**Input Arguments**

You must specify the following input argument:

*Which* is a case-insensitive character string that specifies the forecast series to return.

**Output Arguments**

You must specify the following output argument:

*Result* takes a character value that specifies the type of a numeric array to receive the forecast series. You can specify the following values:

- **LOWER** requests that the array receive lower confidence limit series.
- **PREDICT** requests that the array receive prediction scores.
- **STDERR** requests that the array receive prediction standard error series.
- **UPPER** requests that the array receive upper confidence limit series.

**SCORE.Initialize Method**

```plaintext
rc = obj.Initialize (FORENObject);
```

Initializes the SCORE instance by using the score XML from the FORENG instance.

**Input Arguments**

You must specify the following input argument:

*FORENObject* specifies the FORENG object instance to use as the source of score context (XML).

**SCORE.Initialize Method**

```plaintext
rc = obj.Initialize (INSCOREObject <, YName>);
```

Initialize a SCORE instance by using the score XML from the INSCORE instance.

**Input Arguments**

You must specify the following input argument:
INSCOREObject specifies the INSCORE object instance to use as the source of score context (XML).

You can also specify the following input argument:

YName takes a string variable that specifies the name of the dependent variable to use in selecting the INSCORE row that provides the score XML for the Initialize method. If YName is specified, a score XML specification with an exact match must be found in the INSCORE object. If YName is not specified, the SCORE instance name is used to find a matching INSCORE score specification, and an INSCORE specification with a missing name value is also considered a match.

SCORE.ncon Method

\[
ncon = obj.ncon() ;
\]

Returns the number of controllable X variables in the SCORE instance. A missing value indicates that the SCORE instance has not been initialized.

Arguments
There are no arguments associated with this method.

SCORE.nfor Method

\[
nfor = obj.nfor() ;
\]

Returns the length (observation count) of the forecast series for the SCORE instance. A missing value indicates that the SCORE instance has not produced a successful forecast.

Arguments
There are no arguments associated with this method.

SCORE.Run Method

\[
rc = obj.Run() ;
\]

Runs the SCORE instance to generate a forecast from the future values of the controllable variables. Upon successful completion, various results can be extracted from the SCORE instance.

Arguments
There are no arguments associated with this method.

SCORE.SetControl Method

\[
rc = obj.SetControl(XSeries < .SymbolName >) ;
\]

Adds the future values for a controllable variable before running score.

Input Arguments
You must specify the following input argument:
**XSeries**
takes a string variable that specifies a numeric array that contains the future values of a controllable variable that is referenced in the model’s score.

You can also specify the following input argument:

**SymbolName**
takes a string variable that specifies the name of the controllable variable in the model’s score. If this argument is not specified, a controllable variable that has the same variable name as the **XSeries** argument must exist.

**SCORE.SetOption Method**

```
rc = obj.SetOption ('Name', Value <,'Name', Value,. . .>);
```

Specifies named options for the SCORE instance.

**Input Arguments**
You must specify at least one of the following 'Names' and its associated Value:

- **'ALPHA'** takes a numeric Value between 0 and 1 that specifies the significance level for forecast confidence bands.
- **'HORIZON'** takes a numeric Value that specifies forecast horizon reference time.
- **'LEAD'** takes a nonnegative integer Value that specifies the forecast horizon.

---

### DIAGSPEC Object

The DIAGSPEC object controls specification for a time series model identification process. You can configure this object so that appropriate time series model families are included in the search process. You can also configure this object to set various other search options.

Table 3.9 lists the time series model families that are supported.

**Table 3.9  Model Families for the DIAGSPEC Object**

<table>
<thead>
<tr>
<th>Family</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARIMA</td>
<td>ARIMA/ARIMAX models</td>
</tr>
<tr>
<td>ESM</td>
<td>Exponential smoothing models</td>
</tr>
<tr>
<td>IDM</td>
<td>Intermittent demand models (Croston’s model and average demand model)</td>
</tr>
<tr>
<td>UCM</td>
<td>Unobserved component models</td>
</tr>
</tbody>
</table>

Table 3.10 summarizes the methods that are associated with the DIAGSPEC object.
Table 3.10  Methods of the DIAGSPEC Object

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Close</td>
<td>Close the DIAGSPEC object</td>
</tr>
<tr>
<td>Open</td>
<td>Open the DIAGSPEC object</td>
</tr>
<tr>
<td>SetARIMAX</td>
<td>Set ARIMA diagnostic options</td>
</tr>
<tr>
<td>SetARIMAXOutlier</td>
<td>Set ARIMA outlier detection options.</td>
</tr>
<tr>
<td>SetARIMAXRefine</td>
<td>Set ARIMA parameter refinement options</td>
</tr>
<tr>
<td>SetCombine</td>
<td>Configure automatic combination options</td>
</tr>
<tr>
<td>SetESM</td>
<td>Set ESM diagnostic options</td>
</tr>
<tr>
<td>SetIDM</td>
<td>Set IDM diagnostic options</td>
</tr>
<tr>
<td>SetOption</td>
<td>Set overall control options</td>
</tr>
<tr>
<td>SetTransform</td>
<td>Set transform test options</td>
</tr>
<tr>
<td>SetTrend</td>
<td>Set trend test options</td>
</tr>
<tr>
<td>SetUCM</td>
<td>Set UCM diagnostic options</td>
</tr>
<tr>
<td>SetUCMRefine</td>
<td>Set UCM parameter refinement options</td>
</tr>
</tbody>
</table>

Once configured, the DIAGSPEC object can be used to specify the model optimization and evaluation settings of the DIAGNOSE object (DIAGNOSEObject.SetSpec(DIAGSPECObject)). In turn, this DIAGNOSE object finds the best suited model for the target series in the data frame that initialized the DIAGNOSE object.

Figure 3.2 diagrams the methods of the DIAGSPEC object.
DIAGSPEC Synopsis

DECLARE OBJECT obj (DIAGSPEC) ;

Method syntax, in order of typical usage:

\[
rc = obj.Open () ;
rc = obj.SetOption ("Name", Value <,"Name", Value, ... >) ;
rc = obj.SetARIMAX (<"Name", Value,"Name", Value, ... >) ;
rc = obj.SetARIMAXRefine (<"Name", Value,"Name", Value, ... >) ;
rc = obj.SetESM (<"Name", Value,"Name", Value, ... >) ;
rc = obj.SetCombine (<"Name", Value,"Name", Value, ... >) ;
rc = obj.SetIDM (<"Name", Value,"Name", Value, ... >) ;
rc = obj.SetTransform (<"Name", Value <,"Name", Value, ... >) ;
rc = obj.SetTrend (<"Name", Value <,"Name", Value, ... >) ;
rc = obj.SetUCM (<"Name", Value,"Name", Value, ... >) ;
rc = obj.SetUCMRefine (<"Name", Value, ... >) ;
rc = obj.Close () ;
\]

The following remarks apply to 'Name', Value pairs in the various DIAGSPEC methods:

- The 'Name' argument values are always case-insensitive.
• The Value argument values that require string types are case-insensitive unless otherwise noted.

The DIAGSPEC.Open method always resets the object’s diagnostic control options to their default settings. By default, ARIMA and ESM model family diagnostics are enabled. Sometimes you might want to enable other model families with their default diagnostic control options. You can call the DIAGSPEC methods that are related to the different model families without any arguments to enable diagnostic tests for that model family to be tested with the associated default diagnostic control options. These property methods differ from the others that require at least one 'Name', Value pair.

The following methods can be called with no arguments:

- SetARIMAX
- SetARIMAXOutlier
- SetARIMAXRefine
- SetCombine
- SetESM
- SetIDM
- SetUCM
- SetUCMRefine

In all these cases, calling the DIAGSPEC methods with no arguments always resets the associated diagnostic control options to defaults regardless of preceding calls to the DIAGSPEC method that you might have made to change associated settings. Further details are provided in the descriptions of the various DIAGSPEC methods.

### DIAGSPEC Methods

**DIAGSPEC.Close Method**

```csharp
cr = obj.Close();
```

Finalizes the diagnostic settings in the DIAGSPEC instance. This prepares DIAGSPEC to be used in a DIAGNOSE instance or stored via an OUTDIAG collector object.

**Arguments**

There are no arguments associated with this method.
**DIAGSPEC.Open Method**

```c
rc = obj.Open();
```

Opens the DIAGSPEC instance for configuration and initializes all options to their default values.

**Arguments**

There are no arguments associated with this method.

**DIAGSPEC.SetARIMAX Method**

```c
rc = obj.SetARIMAX(<'Name', Value,'Name', Value,...>);
```

Specifies control options for performing ARIMAX model diagnostics. A SetARIMAX method call with no arguments enables ARIMAX diagnosis with the default options, which includes defaults for ARIMAX refinement. By default, this method disables ARIMAX outlier detection. This method is equivalent to the ARIMAX statement in the HPFDIAGNOSE procedure in *SAS Forecast Server Procedures: User’s Guide*.

**Input Arguments**

You can specify one or more of the following `Names` and its associated `Value`:

- **'CRITERION'**
  - takes a string `Value` that specifies the identification criterion. You can specify one of the following `Values`:
    - **AIC**
      - specifies Akaike’s information criterion.
    - **SBC**
      - specifies the Schwarz Bayesian information criterion.
  - The default is SBC.

- **'ESTMETHOD'**
  - takes a string `Value` that specifies the ARIMA estimation method. You can specify one of the following `Values`:
    - **CLS**
      - specifies the conditional least squares method.
    - **ML**
      - specifies the maximum likelihood method.
    - **ULS**
      - specifies the unconditional least squares method.
  - The default is CLS.

- **'IDENTIFY'**
  - takes a string `Value` that specifies the identification order. You can specify one of the following `Values`:
    - **ARIMA**
      - finds an ARIMA model for the error series first and then chooses significant inputs and events.
    - **BOTH**
      - fits models by using the two methods and determines the better model.
    - **REG**
      - finds a regression model first and then decides the AR and MA polynomial orders.
  - The default is ARIMA.

- **'METHOD'**
  - takes a string `Value` that specifies the tentative ARMA orders. You can specify one of the following `Values`:
Chapter 3: Automatic Time Series Modeling Package

ESACF specifies the extended sample autocorrelation function.
MINIC specifies the minimum information criterion.
SCAN specifies the smallest canonical correlation analysis.

The default is MINIC.

'NOINT' takes a numeric Value that when set to 1, suppresses the constant term. The default value is 0 (no constant term).

'P' takes a Value that specifies a two-dimensional array that contains the nonseasonal AR order range. The first element in the array contains the minimum nonseasonal order, and the second element in the array contains the maximum nonseasonal order. The default range is [0,5].

'PERROR' takes a Value that specifies a two-dimensional array that contains the AR order for MINIC error. The first element in the array contains the minimum AR order for MINIC error, and the second element in the array contains the maximum AR order for MINIC error. The default range is [5,10].

'PS' takes a Value that specifies a two-dimensional array that contains the seasonal AR order range. The first element in the array contains the minimum seasonal order, and the second element in the array contains the maximum seasonal order. The default range is [0,2].

'Q' takes a Value that specifies a two-dimensional array that contains the nonseasonal MA order range. The first element in the array contains the minimum nonseasonal order, and the second element in the array contains the maximum nonseasonal order. The default range is [0,5].

'QS' takes a Value that specifies a two-dimensional array that contains the seasonal MA order range. The first element in the array contains the minimum seasonal order, and the second element in the array contains the maximum seasonal order. The default range is [0,2].

'SIGLEVEL' takes a numeric Value between 0 and 1 that specifies the significance level to use as a cutoff value to decide the AR and MA orders. The default value is 0.05.

'XDEN' takes a Value that specifies a two-dimensional array that contains the transfer function denominator order range. The first element in the array contains the minimum transfer function denominator order, and the second element in the array contains the maximum transfer function denominator order. The default range is [0,2].

'XNUM' takes a Value that specifies a two-dimensional array that contains the transfer function numerator order range. The first element in the array contains the minimum transfer function numerator order, and the second element in the array contains the maximum transfer function numerator order. The default range is [0,2].

**DIAGSPEC.SetARIMAXOutlier Method**

```
rc = obj.SetARIMAXOutlier ( '<Name', Value;'Name', Value,...> ) ;
```

Specifies outlier detection options in an ARIMAX model. These options enables the automatic detection of significant deviations (outliers) between the dependent and forecast series of a diagnosed ARIMAX model. These outliers represent significant events (sudden changes or patterns) in the dependent series that could not be properly modeled using only the original set of input variables. In order to improve the performance of the
ARIMAX model, the recognized events are automatically converted into dummy input variables and added to the model. A SetARIMAXOutlier method call with no arguments enables ARIMAX model diagnosis and sets default options for ARIMAX outlier tests. This method is equivalent to the OUTLIER= option in the ARIMAX statement in the HPFDIAGNOSE procedure in *SAS Forecast Server Procedures: User’s Guide*.

**Input Arguments**
You can specify one or more of the following *Names* and its associated *Value*:

- **'ALLOWAO'** takes a Boolean *Value* that indicates whether to detect additive outliers. When the value of 'ALLOWAO' is 1, additive outlier detection is enabled. Otherwise, detection is disabled. The default value is 1.

- **'ALLOWLS'** takes a Boolean *Value* that indicates whether to detect level shift outliers. When the value of 'ALLOWLS' is 1, level shift outlier detection is enabled. Otherwise, detection is disabled. The default value is 1.

- **'ALLOWTLS'** takes a Boolean *Value* that indicates whether to detect temporary level shift outliers. When the value of 'ALLOWTLS' is 1, temporary level shift outlier detection is enabled. Otherwise, detection is disabled. You can use the 'TLSVALS' option to specify the valid durations of temporary level shifts that should be detected. The default value is 0.

- **'DETECT'** takes a string *Value* that specifies the criterion for automatic modeling of outliers in a model. You can specify one of the following *Values*:
  - **YES** specifies that detected outliers should be included in a model if the model that models the effect of the outliers is successfully diagnosed.
  - **MAYBE** specifies that detected outliers should be included in a model if the model that models the effect of the outliers is successfully diagnosed and has a smaller criterion than the model without outliers.
  - **NO** specifies that no outlier detection is performed.

  The default is MAYBE.

- **'ENTRYPCT'** takes a numeric *Value* between 0 and 100 that specifies the percentage of criterion improvement between two candidate models. This option overrides the value of the ENTRYPCT option in the SetOption method. The default value is 0.1.

- **'FILTER'** takes a string *Value* that specifies a model for outlier detection. You can specify one of the following *Values*:
  - **FULL** specifies that outliers are detected using a full model.
  - **SUBSET** specifies that outliers are detected using a subset model that includes nonseasonal AR and MA filters only.

  If the data have no seasonality, then the outlier detection is not affected by the FILTER option. The default is FULL.

- **'MAXNUM'** takes a nonnegative integer *Value* that specifies the maximum number of outliers to include in a model. The default value is 2.

- **'MAXPCT'** takes a numeric *Value* between 0 and 100 that specifies the maximum number of outliers to include in a model as a percentage of the length of the dependent time series. If
MAXNUM=5 and MAXPCT=10, the maximum number of outliers is \( \min(5, 0.1T) \), where \( T \) is the length of the dependent time series after the beginning and ending missing values are removed. The default value is 2.

'SIGLEVEL' takes a numeric Value between 0 and 1 that specifies the cutoff value for outlier detection. This option overrides the value of the SIGLEVEL option in the SetOption method. The default value is 0.01.

'TLSVALS' takes a numeric array Value that specifies the valid durations of any detected temporary level shifts. The duration of a temporary level shift should be greater than or equal to 2. An empty Value array or missing values in the array are not allowed. This option is ignored when the 'ALLOWTLS' option is set to 0 (default). If this option is not specified, then the detection of temporary level shift outliers is automatically disabled. The default value is missing value.

**DIAGSPEC.SetARIMAXRefine Method**

\[
rc = \text{obj.SetARIMAXRefine}(<\text{'Name'}, \text{Value},\text{'Name'}, \text{Value}, \ldots>)
\]

Specifies ARIMA parameter refinement options. These options enable the refinement of insignificant parameters of the final model, identification of the factors to refine, and identification of the order of factors. A SetARIMAXRefine method call with no arguments enables ARIMA model diagnosis and sets default options for ARIMA refinement. This method is equivalent to the REFINEPARMS= option in the ARIMAX statement in the HPFDIAGNOSE procedure in *SAS Forecast Server Procedures: User’s Guide*.

**Input Arguments**

You can specify one or more of the following 'Names' and its associated Value:

'ORDER' takes a string Value that specifies the order of diagnosing model components. You can specify one of the following Values:

- **ALL** is equivalent to ARMA:INPUT.
- **ARMA** tests ARMA coefficients.
- **INPUT** tests input variable coefficients.
- **ARMA:INPUT** tests ARMA coefficients before predictor coefficients.
- **INPUT:ARMA** tests predictor coefficients before ARMA coefficients.

The default is ALL.

'SIGLEVEL' takes a numeric Value between 0 and 1 that specifies the cutoff value for refining all insignificant parameters. The default value is 0.4.
**DIAGSPEC.SetCombine Method**

```plaintext
rc = obj.SetCombine (< 'Name', Value,'Name',Value, ... >);
```

Specifies combination options for the DIAGSPEC instance. A SetCombine method call with no arguments enables automatic combination with the default options. This method is equivalent to the COMBINE statement in the HPFDIAGNOSE procedure in *SAS Forecast Server Procedures: User’s Guide*.

**Input Arguments**

You can specify one or more of the following 'Names' and its associated Value:

- **'AICC.ABSWGT'** takes a numeric Value between 0 and 1 that specifies a lower bound for computed weights. Computed weights with values less than Value are omitted, and the remaining weights are normalized to sum to 1. This argument is used along with the AICC value for the 'WEIGHT' argument. For more information, see the 'WEIGHT' argument.

- **'AICC.BESTPCT'** takes a numeric Value between 0 and 100 that specifies the percentage of the number of candidate models to retain in the combination after any specified forecast exclusion tests have been performed. The remaining weights are normalized to sum to 1. This argument is used along with the AICC value for the 'WEIGHT' argument. For more information, see the 'WEIGHT' argument.

- **'AICC.BESTN'** takes a numeric Value that specifies the number of candidates to be retained in the combination as a percentage of the total weighted number. The remaining weights are normalized to sum to 1. This argument is used along with the AICC value for the 'WEIGHT' argument. For more information, see the 'WEIGHT' argument.

- **'AICC.LAMBDA'** takes a numeric Value that specifies the scale factor that is used in the computation of the AICC weights. The default value is 1, which results in the usual Akaike weights. This argument is used along with the AICC value for the 'WEIGHT' argument. For more information, see the 'WEIGHT' argument.

- **'ENCALPHA'** takes a numeric Value that specifies the encompassing test significance level. The default value is 0.05.

- **'ENCTEST'** takes a string Value that specifies the encompassing test type. The encompassing test attempts to eliminate from consideration forecasts that fail to add significant information to the final forecast. You can specify the following Values:

  - **HLN** uses the Harvey-Leybourne-Newbold (HLN) test to estimate pairwise encompassing between candidate forecasts.
  - **NONE** performs no encompassing tests.
  - **OLS** uses an OLS-based regression test to estimate pairwise encompassing between candidate forecasts.

  The default is NONE.

- **'HMISSPCT'** takes a numeric Value between 0 and 100 that specifies a threshold for the percentage of missing forecast values in the combination horizon. This threshold is used to exclude a candidate forecast from consideration in the final combination. By default, no horizon missing percentage test is performed on candidate forecasts. The forecast horizon is the region of time in which multistep forecasts are generated.
'LAD.OBJTYPE' takes a string Value that specifies the form of the objective function. This argument is used along with the LAD value for the 'WEIGHT' argument. For more information, see the 'WEIGHT' argument. You can specify the following Values:

- **L1**: specifies that the objective is an \( \ell_1 \) norm that involves loss series.
- **LINF**: specifies that the objective is an \( \ell_\infty \) norm that involves the loss series.

The default is L1.

'LAD.ERRTYPE' takes a string Value that specifies the form of the loss series in the objective function. This argument is used along with the LAD value for the 'WEIGHT' argument. For more information, see the 'WEIGHT' argument. You can specify the following Values:

- **ABS**: specifies that loss series terms are deviations.
- **APE**: specifies that loss series terms are percentage deviations.
- **RAE**: specifies that loss series terms are relative error deviations.

The default is ABS.

'MISSMODE' takes a string Value that specifies a method for treating missing values in the forecast combination. In a particular time slice across the combination ensemble, one or more combination contributors can have a missing value. This setting determines the treatment of those contributors in the final combination for such time indices. You can specify the following Values:

- **MISSING**: generates a missing combined forecast at each time index with one or more missing contributors.
- **RESCALE**: rescales the combination weights for the nonmissing contributors at each time index to sum to 1. You cannot specify RESCALE when the values of the 'WEIGHT' argument is OLS or NRLS.

The default value depends on the combination weight method specified by the 'WEIGHT' argument. RESCALE is the default for simple average, user-specified weights, ranked user weights, ranked weights, and RMSE weights. MISSING is the default for all other methods.

'MISSPCT' takes a numeric Value between 0 and 100 that specifies a threshold for the percentage of missing forecast values in the combination estimation region. This threshold is used to exclude a candidate forecast from consideration in the final combination. By default, no missing percentage test is performed on candidate forecasts.

'RANKING' takes a string Value that specifies the forecast ranking criterion (statistics of fit) mnemonic to be used when ranking forecast candidates. For a list of valid values, see the CRITERION= option in the HPFDIAGNOSE procedure in SAS Forecast Server Procedures: User's Guide. The default value is RMSE.

'SEMODE' takes a string Value that specifies the method for computing the prediction error variance series. This series is used to compute the prediction standard error, which in turn is used to compute confidence bands on the combined forecast. You can specify the following Values:
**DIAG** computes the prediction error variance by assuming the forecast errors at time $t$ are uncorrelated so that the simple diagonal form of $\Sigma_t$ is used.

**ESTCORR** computes the prediction error variance by using estimates of $\rho_{i,j,t}$, the sample cross-correlation between $e_{i,t}$ and $e_{j,t}$ over the time span $t = 1, \ldots, T$, where $t$ denotes the last time index of the actual series $y_t$. This Value implies that the error series $e_{i,t}$ and $e_{j,t}$ are assumed to be jointly stationary.

The default is DIAG.

**'USERWEIGHTS'** takes a numeric array Value that specifies the combinations weights to be used when the value of the 'WEIGHT' argument is RANKWGT or USERDEF. For more information, see the 'WEIGHT' argument.

**'WEIGHT'** takes a string Value that specifies the method for determining the combination weights used in the weighted average of the candidate forecasts in the combination list. You can specify the following Values:

- **AICC** computes the combination weights based on corrected AIC weights. By default, all AICC scored candidate forecasts are combined. Frequently there is considerable disparity between the weights because of the exponential weighting scheme, so additional arguments are provided to affect the scaling and to cull low-scoring candidates from consideration for computational efficiency. For more information, see the 'AICC.ABSWGT', 'AICC.BESTPCT', and 'AICC.LAMBDA' arguments. You can specify one of these additional arguments when you specify a value of AICC for the 'WEIGHT' argument. By default, the 'AICC.LAMBDA' argument is chosen and set to 1 when you specify the value of AICC for the 'WEIGHT' argument.

- **AVERAGE** computes the simple average of the forecasts that are selected for combination.

- **ERLS** computes the combination weights based on a constrained least squares problem to minimize the $\ell_2$ norm of the combined forecast residuals subject to the constraint that the weights sum to 1.

- **LAD** computes the weights based on a least absolute deviations (LAD) measure of fit for the combined forecast. A linear program is formulated in which an objective function to be minimized is expressed in terms of the absolute values of a loss series subject to constraints that the weights sum to 1 and be nonnegative. You can specify the form of the objective function via the 'LAD.OBJTYPE' argument and the form of the loss series in the objective function via the 'LAD.ERRTYPE' argument.

- **NERLS** computes the combination weights based on a constrained least squares problem to minimize the $\ell_2$ norm of the combined forecast residuals subject to the constraints that the weights sum to 1 and be nonnegative.
NRLS is equivalent to NERLS except that the resulting combination weights are not constrained to summing up to 1.

OLS computes the combination weights that result from the ordinary least squares problem to minimize the $\ell_2$ norm of the combined forecast residuals.

RANKWGT assigns weights by using the rank of the candidate forecasts at the time the combination is performed. You must specify the weights assigned via the 'USERWEIGHTS' argument, where the number of specified values must agree with the number of model families that are specified in the current instance of the DIAGSPEC object. These weights must sum to 1. The weights are assigned by ranking the candidate forecasts from best to worst. The best uses the first weight, W1, and so on. The set of weights that is used is normalized to account for candidates that fail to forecast or for candidates that are omitted from the final combination.

RMSEWGT computes the combination weights based on the RMSE statistic of fit for the forecast contributors. The weights are normalized to sum to 1.

USERDEF assigns weights by using the list of user-specified values. You must specify the weights assigned via the 'USERWEIGHTS' argument, where the number of values that are specified must agree with the number of model families that are specified in the current instance of the DIAGSPEC object. These weights must sum to 1. The set of weights that is used is normalized to account for candidates that fail to forecast or for candidates that are omitted from the final combination.

The default is AVERAGE.

**DIAGSPEC.SetESM Method**

```r
rc=obj.SetESM (<'Name', Value,'Name',Value,. . >) ;
```

Specifies control options for performing ESM testing. A SetESM method call with no arguments enables ESM diagnosis with the default options. This method is equivalent to the ESM statement in the HPFDIAGNOSE procedure in *SAS Forecast Server Procedures: User’s Guide*.

**Input Arguments**

You can specify one or more of the following 'Names' and its associated *Value*:

- 'METHOD' takes a string *Value* that specifies the ESM method. You can specify the following *Values*:
  - **BEST** requests the best candidate smoothing model among the SIMPLE, LINEAR, DAMPTREND, SEASONAL, ADDWINTERS, or WINTERS methods that are described in the HPFESMSPEC procedure in *SAS Forecast Server Procedures: User’s Guide*.
BESTN requests the best candidate nonseasonal smoothing model among the SIMPLE, LINEAR, or DAMPTREND methods that are described in the HPFESMSPEC procedure in *SAS Forecast Server Procedures: User’s Guide*.

BESTS requests the best candidate seasonal smoothing model among the SEASONAL, ADDWINTERS, or WINTERS methods that are described in the HPFESM-SPEC procedure in *SAS Forecast Server Procedures: User’s Guide*.

The default is BEST.

'SIGLEVEL' takes a numeric *Value* between 0 and 1 that specifies the significance level. The default value is 0.05.

**DIAGSPEC.SetIDM Method**

```plaintext
rc=obj.SetIDM (< 'Name', Value,'Name',Value,... >) ;
```

Specifies control options for IDM testing to be performed. A SetIDM method call with no arguments enables IDM diagnosis with the default options. This method is equivalent to the IDM statement in the HPFDIAGNOSE procedure in *SAS Forecast Server Procedures: User’s Guide*.

**Input Arguments**

You can specify one or more of the following *Names* and its associated *Value*:

'BASE' takes a numeric *Value* that specifies the base value of the time series that is used to determine the demand series components as departures from this value. A missing value causes automatic detection of the base value. The default is a missing value.

'INTERMITTENT' takes a numeric *Value* greater than 1 that specifies the intermittency threshold. The default value is 2.

'METHOD' takes a string *Value* that specifies the IDM method. You can specify one of the following *Values*:

- **AVERAGE** requests the extended sample autocorrelation function.
- **BEST** uses the single smoothing model to fit the average demand component.
- **CROSTON** uses the two smoothing models to fit the demand interval component and the demand size component.

The default is BEST.

'TRANSFORM' takes a string *Value* that specifies the transform to use. You can specify the following *Values*:

- **AUTO** automatically chooses between NONE and LOG based on model selection criteria.
- **BOXCOX(value)** requests Box-Cox transformation with a parameter *value* between –5 and 5. The default is BOXCOX(1).
- **LOG** requests logarithmic transformation.
- **LOGIT** requests logistic transformation.
Chapter 3: Automatic Time Series Modeling Package

NONE does not apply a transformation.

SQRT specifies square-root transformation.

The default is NONE.

'TRANSPARM' takes a numeric Value that specifies the transform parameter (Box-Cox only).

DIAGSPEC.SetOption Method

\[ rc = obj.SetOption( 'Name', \text{Value} <, 'Name', \text{Value}, \ldots >) \];

Sets a named option for the DIAGSPEC instance.

Input Arguments

You must specify at least one of the following 'Names' and its associated Value:

'CRI TERION' takes a string Value that specifies the fit statistic mnemonic. For a list of valid values, see the CRITERION= option in the HPFDIAGNOSE procedure in SAS Forecast Server Procedures: User's Guide. The default is RMSE.

'DELAYEVENT' takes a nonnegative numeric Value that specifies the event variable lag.

'DELAYINPUT' takes a nonnegative numeric Value that specifies the input variable lag. If not specified, the delay lags for the inputs are automatically chosen.

'ENTRYPCT' takes a numeric Value between 0 and 100 that specifies the percentage of AIC or SBC improvement between two candidate models. The default value is 0.1.

'INPUTMISSPCT' takes a numeric Value between 0 and 100 that specifies the number of the missing observations as a percentage of the length of the input time series. If INPUTMISSINGPCT=50, then the input time series that has more than 50% missing data is ignored in the model. The default value is 10 (10%).

'PREFILTER' takes a string Value that specifies how missing and extreme values are handled prior to diagnostic tests. You can specify one of the following Values:

    BOTH is equivalent to both YES and EXTREME.
    EXTREME requests that extreme values be set to missing for a tentative ARIMA model and extreme values be used for the final ARIMAX model diagnostics.
    MISSING requests that smoothed values for missing data be applied for tentative order selection and missing values be used for the final diagnostics.
    YES requests that smoothed values for missing data be applied to overall diagnoses.

The default is YES.

'SELECTINPUT' takes a numeric Value, no less than 1, that specifies the number of best input variables to be selected; or takes a string Value that specifies the maximum number of input variables to select. You can specify one of the following string Values:
**SELECT** selects the input variables that satisfy the criteria of noncollinearity, nonnegative delay, and smaller AIC.

**ALL** selects the input variables that satisfy the criteria of noncollinearity and nonnegative delay.

The default is **SELECT**.

**'SIGLEVEL'** takes a numeric **Value** between 0 and 1 that specifies the cutoff value for all diagnostic tests such as log transformation, stationarity, tentative ARMA order selection, ARIMAX outlier detection, and significance of UCM components. The default is 0.05.

**'TESTINPUT'** takes a string **Value** that specifies the test input control. You can specify one of the following **Values**:

- **BOTH** requests that the log transform and trend testing of the input variables be applied independently of the variable to be forecast.
- **NONE** does not apply a transformation. The same differencing is applied to the input variables as is used for the variable to be forecast.
- **TRANSFORM** requests that the log transform testing of the input variables be applied independently of the variable to be forecast.
- **TREND** requests that the trend testing of the input variables be applied independently of the variable to be forecast.

The default is **NONE**.

---

**DIAGSPEC.SetTransform Method**

```plaintext
rc=obj.SetTransform ( 'Name', Value < 'Name', Value, ... ) ;
```

Specifies control options for functional transform testing. This method is equivalent to the TRANSFORM statement in the HPFDIAGNOSE procedure in *SAS Forecast Server Procedures: User’s Guide*.

**Input Arguments**

You must specify at least one of the following **Names** and its associated **Value**:

- **'P'** takes a numeric **Value** that specifies the AR order for log transform test. The default value is 2.
- **'SIGLEVEL'** takes a numeric **Value** between 0 and 1, inclusive, that specifies the significance level to use as a cutoff value to decide whether the series requires a log transformation. The default value is 0.05.
- **'TRANSFORM'** takes a string **Value** that specifies the transform to use. You can specify the following **Values**:
  - **AUTO** automatically chooses between NONE and LOG based on model selection criteria.
  - **BOXCOX(value)** requests Box-Cox transformation with a parameter **value** between –5 and 5. You can specify the parameter value via the TRANSPARM option. The default is BOXCOX(1).
Chapter 3: Automatic Time Series Modeling Package

LOG requests logarithmic transformation.

LOGIT requests logistic transformation.

NONE does not apply a transformation.

SQRT specifies square-root transformation.

The default is NONE.

'TRANSPORT' takes a string Value that specifies inverse forecasts. You can specify one of the following values:

MEAN requests that the inverse transform produce mean forecasts.

MEDIAN requests that the inverse transform produce median forecasts.

The default is MEAN.

'TRANSPARM' takes a numeric Value that specifies the transform parameter (Box-Cox only). The default value is 1.

**DIAGSPEC.SetTrend Method**

```plaintext
rc=objc.SetTrend ('Name', Value <,'Name',Value,...>);
```

Specifies control options for trend testing. This method is equivalent to the TREND statement in the HPFDIAGNOSE procedure in *SAS Forecast Server Procedures: User’s Guide*.

**Input Arguments**

You must specify at least one of the following 'Names' and its associated Value:

'DIFF' takes a string Value that specifies simple differencing. You can specify one of the following values:

AUTO tests for simple differencing.

NONE requests that no simple differencing be used.

The default is AUTO.

'DIFFN' takes a numeric Value that specifies the forced simple differencing order. For more information, see the TREND statement in *SAS Forecast Server Procedures: User’s Guide*.

'DIFFRAN' takes a numeric Value that specifies the range of simple differencing order for testing. For more information, see the TREND statement in *SAS Forecast Server Procedures: User’s Guide*.

'P' takes a numeric Value that specifies the autoregressive order for the augmented unit root tests and a seasonality test. The default value is 5.

'SDIFF' takes a string Value that specifies seasonal differencing. You can specify one of the following values:
AUTOMATICALLY tests for seasonal differencing.
NONE requests that no seasonal differencing be used.

The default is AUTO.

'SDIFFN' takes a numeric Value that specifies the forced seasonal differencing order. For more information, see the TREND statement in SAS Forecast Server Procedures: User’s Guide.

'SIGLEVEL' takes a numeric Value between 0 and 1, inclusive, that specifies the significance level to use as a cutoff value to decide whether the series needs differencing. The default value is 0.05.

DIAGSPEC.SetUCM Method

rc=obj.SetUCM (< 'Name', Value,'Name',Value,...> ) ;

Specifies control options for performing UCM model diagnostics. A SetUCM method call with no argument enables UCM diagnosis with the default options, which includes defaults for UCM refinement. This method is equivalent to the UCM statement in the HPFDIAGNOSE procedure in SAS Forecast Server Procedures: User’s Guide.

Input Arguments
You can specify one or more of the following 'Names' and its associated Value:

'ALL' takes a Boolean Value (0 or 1) that when set to 1, tests all components. The default value is 0.

'AUTOREG' takes a Boolean Value (0 or 1) that when set to 1, tests whether an autoregressive component is significant in the model. The default value is 0.

'CYCLE' takes a Boolean Value (0 or 1) that when set to 1, tests whether two cycle components are significant in the model. The two CYCLE components are included, and the LEVEL component is added. When the series has the seasonality information, the CYCLE component is not tested. The default value is 0.

'DEPLAG' takes a Boolean Value (0 or 1) that when set to 1, tests whether a dependent lag component is significant in the model. Only a single time lag (order 1) is considered. The model checks whether the value of the dependent variable in the previous forecast time step affects the forecast performance in the current forecast time step. The default value is 0.

'IRREGULAR' takes a Boolean Value (0 or 1) that when set to 1, tests whether an irregular component is significant in the model. The default value is 1.

'LEVEL' takes a Boolean Value (0 or 1) that when set to 1, tests whether a level component is significant in the model. The default value is 1.

'SEASON' takes a Boolean Value (0 or 1) that when set to 1, tests whether a season component is significant in the model. When the series has the seasonality information, the season component is not tested. The default value is 0.

'SIGLEVEL' takes a numeric Value between 0 and 1, inclusive, that specifies the significance level to use as a cutoff value to decide which component or variances (or both) are significant. The default value is 0.05.
'SLOPE' takes a Boolean Value (0 or 1) that when set to 1, tests whether a slope component is significant in the model. The default value is 1.

**DIAGSPEC.SetUCMRefine Method**

```plaintext
rc = obj.SetUCMRefine (<'Name', Value, ...>);
```

Specifies control options for UCM model parameter refinement. A SetUCMRefine method call with no specified arguments enables UCM diagnosis and uses default options for UCM refinement. This method is equivalent to the REFINEPARMS= option in the UCM statement in the HPF DIAGNOSE procedure in *SAS Forecast Server Procedures: User’s Guide*.

**Input Arguments**

You can specify one or more of the following 'Name' and its associated Value:

- **'ORDER'** takes a string Value that specifies the order of subsetting UCM components. You can specify one of the following Values:
  - **ALL** is equivalent to ARMA:INPUT.
  - **ARMA** tests ARMA coefficients.
  - **INPUT** tests input variable coefficients.
  - **ARMA:INPUT** tests ARMA coefficients before predictor coefficients.
  - **INPUT:ARMA** tests predictor coefficients before ARMA coefficients.

  The default is ALL.

- **'SIGLEVEL'** takes a numeric Value between 0 and 1, inclusive, that specifies the cutoff value for refining all insignificant parameters. The default is 0.4.

---

**EVENT Object**

The EVENT object defines an object that is used to make event definitions available to create dummy variables to be used in the TSDF object and other objects.

*Table 3.11* summarizes the methods that are associated with the EVENT object.
### EVENT Synopsis

**DECLARE OBJECT obj (EVENT) ;**

Method syntax, in order of typical usage:

```java
rc = obj.Initialize () ;
rc = obj.Replay (INEVENTObject) ;
rc = obj.EventComb (EventName,'EVENT',Value < , 'EVENT',Value... > < , 'Name',Value... >) ;
rc = obj.EventDef (EventName,'Name',Value < , 'Name',Value... >) ;
rc = obj.EventKey (EventName < , 'EVENT',event name > < , 'Name',Value>) ;
rc = obj.EventGroup (EventName,'EVENT',
  Value,'EVENT',Value < , 'EVENT',Value... > < , 'Name',Value... >) ;
rc = obj.EventGroup (EventName,'GROUP',Value < , 'Name',Value... >) ;
```

### EVENT Methods

**EVENT.Initialize Method**

```java
rc = obj.Initialize () ;
```

Initializes an instance of the EVENT `obj` to the default state, which includes predefined events. This method must be called before events are added by specifying the TSDF.AddEvent method.

**Arguments**

There are no arguments associated with this method.
EVENT.EventComb Method

```plaintext
rc=obj.EventComb (EventName,'EVENT',Value < ,'EVENT',Value... > < ,'Name',Value... > );
```

Adds a combination event definitions to the EVENT object instance. Each call to the EventComb method adds to `obj` the event definition that is specified for `EventName`. This method can be called as many times as needed to define all the combination events that are needed. The defined events are available to be added to the TSDF method and to modeling and forecasting methods. For more information about combination event definitions, see the EVENTCOMB statement in *SAS Forecast Server Procedures: User’s Guide*.

**Input Arguments**

You must specify the following input argument:

- `EventName` specifies the name for the event definition.

You must specify the following argument and its associated `Value` at least once:

- `'EVENT'` takes a string `Value` (for a SAS variable name) that specifies an event to be combined with other events to create the combination event. The event must either be defined in `obj` or be a SAS predefined event.

You can also specify one or more of the following `Names` and their associated `Values`:

- `'LABEL'` takes a string `Value` that specifies a label for the dummy variable for this event definition.

- `'LOCALE'` takes a string `Value` that specifies the locale that is associated with the event. The locale should be a POSIX locale value.

- `'RULE'` takes a string `Value` that specifies the action to take when the values of the events are combined at each observation. You can specify one of the following `Values`:
  
  - `ADD` adds the values.
  - `MAX` uses the maximum value.
  - `MIN` uses the minimum value.
  - `MINMAG` uses the value whose magnitude is the lowest.
  - `MINNZ` uses the minimum nonzero value.
  - `MULT` multiplies the values.

The default is `ADD`.

EVENT.EventDef Method

```plaintext
rc=obj.EventDef (EventName,'Name',Value < ,'Name',Value... > );
```

Adds an event definition to the EVENT object instance. Each call to the EventDef method adds to `obj` the event definition that is specified for `EventName`. This method can be called as many times as needed to define all the events that are needed. The defined events are available to be added to the TSDF method and to modeling and forecasting methods. For more information about event definitions, see the EVENTDEF statement in *SAS Forecast Server Procedures: User’s Guide*.
**Input Arguments**

You must specify the following input argument:

*eventName* specifies the name for the event definition.

You must specify at least one of the following *Names* and its associated *Value*:

- **'STARTDATE'** takes a numeric *Value*, or a string *Value* in SAS date format, that specifies the timing value of the event or the starting date of a do-list that produces timing values for the event.

- **'STARTDT'** takes a numeric *Value*, or a string *Value* in SAS datetime format, that specifies the timing value of the event or the starting datetime of a do-list that produces timing values for the event.

- **'STARTOBS'** takes a numeric *Value* that specifies an observation number as the timing value of the event or the starting observation of a do-list that produces timing values for the event.

- **'DATEKEY'** takes a string *Value* that specifies the timing values of the event. For a list of valid values, see the HPFEVENTS procedure in *SAS Forecast Server Procedures: User’s Guide*.

You can also specify one or more of the following *Names* and their associated *Values* with the restrictions as noted:

- **'ENDDATE'** takes a numeric *Value*, or a string *Value* in SAS date format, that specifies the ending date of a do-list that produces timing values for the event. An 'ENDDATE', *Value* pair must be immediately preceded by the corresponding 'STARTDATE', *Value* pair.

- **'ENDDT'** takes a numeric *Value*, or a string *Value* in SAS datetime format, that specifies the ending datetime of a do-list that produces timing values for the event. An 'ENDDT', *Value* pair must be immediately preceded by the corresponding 'STARTDT', *Value* pair.

- **'ENDOBS'** takes a numeric *Value* that specifies the ending observation of a do-list that produces timing values for the event. An 'ENDOBS', *Value* pair must be immediately preceded by the corresponding 'STARTOBS', *Value* pair.

- **'DATEINTRVL'** takes a string *Value* (in SAS date interval format) that specifies the BY interval of a do-list that produces timing values for the event. A 'DATEINTRVL', *Value* pair must be immediately preceded by the corresponding 'ENDDATE', *Value* pair.

- **'DTINTRVL'** takes a string *Value* (in SAS datetime interval format) that specifies the BY interval of a do-list that produces timing values for the event. A 'DTINTRVL', *Value* pair must be immediately preceded by the corresponding 'ENDDT', *Value* pair.

- **'OBSINTRVL'** takes a string *Value* that specifies the BY interval of a do-list that produces timing values for the event. An 'OBSINTRVL', *Value* pair must be immediately preceded by the corresponding 'ENDOBS', *Value* pair.

You can also specify one or more of the following *Names* and their associated *Values*:

- **'DUR_AFTER'** takes a numeric *Value* that specifies the event duration in number of observations (or periods if the 'PULSE' argument is also specified) after the timing value.
'DUR_BEFORE' takes a numeric Value that specifies the event duration in number of observations (or periods if the 'PULSE' argument is also specified) before the timing value.

'LABEL' takes a string Value that specifies a label for the dummy variable for this event definition.

'LOCALE' takes a string Value that specifies the locale that is associated with the event. The locale should be a POSIX locale value.

'PERIOD' takes a string Value that specifies the interval for periodic recurrence of the timing values.

'PULSE' takes a string Value that specifies the interval for the pulse unit of the timing values.

'RULE' takes a string Value that specifies the action to take when the effects of multiple timing values overlap. You can specify one of the following Values:

- ADD adds the values.
- MAX uses the maximum value.
- MIN uses the minimum value.
- MINMAG uses the value whose magnitude is the lowest.
- MINNZ uses the minimum nonzero value.
- MULT multiplies the values.

The default is ADD.

'SHIFT' takes a numeric Value that specifies the shift in observations or pulse units of the timing values.

'SLOPE_AFTER' takes a string Value that specifies the slope after the timing value for ramps and temporary change events. You can specify one of the following Values:

- DECAY creates a slope away from the peak value.
- GROWTH creates a slope toward the peak value.

The default is GROWTH for ramp events and DECAY for temporary change events.

'SLOPE_BEFORE' takes a string Value that specifies the slope before the timing value for ramps and temporary change events. You can specify one of the following Values:

- DECAY creates a slope away from the peak value.
- GROWTH creates a slope toward the peak value.

The default is GROWTH for ramp events and temporary change events.

'TCPARM' takes a numeric Value that specifies the rate of growth or decay, where Value must be between 0 and 1 inclusive.

'TYPE' takes a string Value that specifies the type (shape) of the event variable. You can specify one of the following Values:
EVENT Methods

POINT creates a point or pulse event.
LS creates a level shift event.
RAMP creates a ramp event.
TR | TEMPRAMP creates a temporary ramp event.
TC creates a temporary change event.
LIN | LINEAR creates a linear trend.
QUAD creates a quadratic trend.
CUBIC creates a cubic trend.
INV | INVERSE creates an inverse trend.
LOG | LOGARITHMIC creates a logarithmic trend.

The default is POINT.

'VALUE' takes a numeric Value that specifies the event indicator value. The default is 1.

EVENT.EventGroup Method

\[
rc = obj\_EventGroup(\textquote{EventName},\textquote{EVENT},Value,\textquote{EVENT},Value <, \textquote{EVENT},Value... > <, \textquote{Name},Value... > ) ;
\]

\[
rc = obj\_EventGroup(\textquote{EventName},\textquote{Group},Value <, \textquote{Name},Value... > ) ;
\]

Adds group event definitions to the EVENT object instance. Each call to the EventGroup method adds to \( \textit{obj} \) the group event definition that is specified for \( \textit{EventName} \). This method can be called as many times as needed to define all the group events that are needed. The defined events are available to be added to the TSDF method and to modeling and forecasting methods. For more information about group event definitions, see the EVENTGROUP statement in \textit{SAS Forecast Server Procedures: User’s Guide}.

The following example creates the new event group TradingDays which is based on the existing group event Days:

\[
rc = eventDB\_EventGroup(\textquote{TradingDays}, \textquote{group}, \textquote{Days}) ;
\]

The group event TradingDays will consist of the seven SAS predefined events Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, and Saturday. When this group event is applied to a monthly series, each event will count the number of weekdays in that month. For example, Sunday will count the number of Sundays in each month.

The following example creates the new event group Holidays, which consists of the two SAS predefined existing events Thanksgiving and Christmas:

\[
rc = eventDB\_EventGroup(\textquote{Holidays}, \textquote{event}, \textquote{Thanksgiving}, \textquote{event}, \textquote{Christmas}) ;
\]

Input Arguments
You must specify the following input argument:

\textit{EventName} specifies the name for the group event definition.
You must specify the events that comprise the group either by specifying an existing group in a 'GROUP' argument and its associated value or by specifying a list of events in one or more 'EVENT' arguments and their associated Values:

'EVENT' takes a string Value (for a SAS variable name) that specifies an event to be included in the group event. The event must either be defined in obj or be a SAS predefined event. You can specify this argument as many times as needed to include all the desired events in the group definition.

'GROUP' takes a string Value (for a SAS variable name) that specifies an existing group event. The group event must either be defined in obj or be a SAS predefined group event. The definition of the group event EventName will be based on this group event definition.

You can also specify one or more of the following 'Names' and their associated Values:

'LABEL' takes a string Value that specifies a label for the dummy variable for this group event definition.

'LOCALE' takes a string Value that specifies the locale that is associated with the group event. The locale should be a POSIX locale value.

EVENT.EventKey Method

rc = obj.EventKey (EventName <,'EVENT','event name'> <,'Name',Value> ) ;

Modifies or adds the event definition EventName to the EVENT instance. Each call to the EventKey method retrieves an existing simple event definition, optionally alters the parameters, and either saves the existing event or adds a new event that is based on the existing event and the new parameters.

The following example alters the existing events Sale and Anniversary:

    rc = eventDB.EventKey("Sale","event","Sale","DUR_AFTER",6);
    rc = eventDB.EventKey("Anniversary","PERIOD","YEAR");

When no 'EVENT','event name' argument is specified, the existing event is assumed to be the same as EventName.

The following example creates new events DeadWeek and PaymentDueDate, which are based on the existing events ExamWeek and InvoiceDate, respectively:

    rc = eventDB.EventKey("DeadWeek","event","ExamWeek","SHIFT",-1);
    rc = eventDB.EventKey("PaymentDueDate","event","InvoiceDate","PULSE","DAY","SHIFT",30);

When a new event is based on an existing event as in the previous example, the existing event (ExamWeek and InvoiceDate) is unchanged.

This method can be called as many times as needed to define or redefine all the events that are needed. The defined events are available to be added to the TSDF method and to modeling and forecasting methods.

Input Arguments

You must specify the following input argument:
EventName specifies the name for the new or updated event definition.

You can also specify the following 'Name' and its associated Value:

'EVENT' specifies a string Value that is the name of an existing event definition. If this argument is omitted, the default is the value of EventName.

You can also specify the following 'Names' and their associated Values; their defaults are the values that are defined in the existing event:

'DUR_AFTER' takes a numeric Value that specifies the event duration in number of observations (or periods if 'PULSE' is specified) after the timing value. The default is the value defined in the existing event.

'DUR_BEFORE' takes a numeric Value that specifies the event duration in number of observations (or periods if 'PULSE' is specified) before the timing value. The default is the value defined in the existing event.

'LABEL' takes a string Value that specifies a label for the dummy variable for this event definition. The default is the value defined in the existing event.

'LOCALE' takes a string Value that specifies the locale that is associated with the event. The locale should be a POSIX locale value. The default is the value defined in the existing event.

'PERIOD' takes a string Value that specifies the interval for periodic recurrence of the timing values. The default is the value defined in the existing event.

'PULSE' takes a string Value that specifies the interval for the pulse unit of the timing values. The default is the value defined in the existing event.

'RULE' takes a string Value that specifies the action to take when the effect of multiple timing values overlap. You can specify one of the following Values:

ADD adds the values.
MAX uses the maximum value.
MIN uses the minimum value.
MINMAG uses the value whose magnitude is the lowest.
MINNZ uses the minimum nonzero value.
MULT multiplies the values.

The default is the value defined in the existing event.

'Shift' takes a numeric Value that specifies the shift in observations or pulse units of the timing values. The default is the value defined in the existing event.

'SLOPE_AFTER' takes a string Value that specifies the slope after the timing value for ramps and temporary change events. You can specify one of the following Values:

DECAY creates a slope away from the peak value.
GROWTH creates a slope toward the peak value.
The default is the value defined in the existing event.

'SLOPE_BEFORE' takes a string Value that specifies the slope before the timing value for ramps and temporary change events. You can specify one of the following Values:

- DECAY creates a slope away from the peak value.
- GROWTH creates a slope toward the peak value.

The default is the value defined in the existing event.

'TCPARM' takes a numeric Value that specifies the rate of growth or decay. Value must be between 0 and 1 inclusive. The default is the value defined in the existing event.

'TYPE' takes a string Value that specifies the type (shape) of the event variable. You can specify one of the following Values:

- POINT creates a point or pulse event.
- LS creates a level shift event.
- RAMP creates a ramp event.
- TR creates a temporary ramp event.
- TEMPRAMP creates a temporary ramp event.
- TC creates a temporary change event.
- LIN creates a linear trend.
- LINEAR creates a linear trend.
- QUAD creates a quadratic trend.
- CUBIC creates a cubic trend.
- INV creates an inverse trend.
- INVERSE creates an inverse trend.
- LOG creates a logarithmic trend.
- LOGARITHMIC creates a logarithmic trend.

The default is the value defined in the existing event.

'VALUE' takes a numeric Value that specifies the event indicator value. The default is the value defined in the existing event.

EVENT.Replay Method

rc=obj.Replay (INEVENTObject) ;

Restores a previously stored set of event definitions from the specified INEVENTObject. The event definitions are loaded into the EVENT instance. For more information about creating event definitions, see the HPFEVENTS procedure in SAS Forecast Server Procedures: User’s Guide.
Input Arguments
You must specify the following input argument:

INEVENTObject specifies the INEVENT instance that defines the source of the event definitions to restore.

SELSPEC Object
The SELSPEC object defines and manipulates forecast model selection list (MSL) objects. The MSL object defines a model-based selection strategy for forecasting a dependent variable based on a historical information set (TSDF). The MSL is a directed acyclic graph (DAG) of time series models, model selection lists, and model combination lists that are evaluated to determine the best-performing forecast. Competing forecasts are evaluated subject to rules that are defined in the MSL and in the FORENG object that evaluates the MSL. Upon evaluation, a final forecast is produced from the best-performing forecast. Abstractly, the best-performing forecast is properly viewed as a path in the MSL DAG. The SELSPEC object offers functionality comparable to the HPFSELECT procedure. SELSPEC objects accept models and subgraphs from the sources that are shown in Table 3.12.

Table 3.12 Model Objects for the SELSPEC Object

<table>
<thead>
<tr>
<th>Object</th>
<th>Package</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARIMASPEC</td>
<td>TSM</td>
<td>ARIMA model to include</td>
</tr>
<tr>
<td>COMBSPEC</td>
<td>ATSM</td>
<td>Forecast combination to include</td>
</tr>
<tr>
<td>DIAGNOSE</td>
<td>ATSM</td>
<td>Generated models to include</td>
</tr>
<tr>
<td>ESMSPEC</td>
<td>TSM</td>
<td>ESM model to include</td>
</tr>
<tr>
<td>FORENG</td>
<td>ATSM</td>
<td>FORENG current FMSG to include</td>
</tr>
<tr>
<td>IDMSPEC</td>
<td>TSM</td>
<td>IDM model to include</td>
</tr>
<tr>
<td>INFMSG</td>
<td>ATSM</td>
<td>Restored SELSPEC models to include</td>
</tr>
<tr>
<td>SELSPEC</td>
<td>ATSM</td>
<td>Subgraph to include</td>
</tr>
<tr>
<td>TSM</td>
<td>TSM</td>
<td>TSM model to include</td>
</tr>
<tr>
<td>UCMSPEC</td>
<td>TSM</td>
<td>UCM model to include</td>
</tr>
</tbody>
</table>

Figure 3.3 diagrams the methods of the SELSPEC object. TSM:XXXSpec refers to any of the ESMSPEC, ARIMASPEC, UCMSPEC, IDMSPEC, and TSM objects, which are part of the TSM package.
Table 3.13 summarizes the methods that are associated with the SELSPEC object.

**Table 3.13** Methods of the SELSPEC Object

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AddFrom</td>
<td>Add from the specified source to the SELSPEC object</td>
</tr>
<tr>
<td>Close</td>
<td>Close SELSPEC for editing</td>
</tr>
<tr>
<td>Open</td>
<td>Open SELSPEC for editing</td>
</tr>
<tr>
<td>SetDiagnose</td>
<td>Set diagnose options</td>
</tr>
<tr>
<td>SetOption</td>
<td>Set named option</td>
</tr>
</tbody>
</table>
### SELSPEC Synopsis

```c
DECLARE OBJECT obj (SELSPEC) ;
```

Method syntax, in order of typical usage:

```c
rc=obj.Open (nspecs) ;
rc=obj.AddFrom (SourceObj <,ListIndex> ) ;
rc=obj.SetDiagnose ('Name', Value <,'Name',Value,...>) ;
rc=obj.SetOption ('Name', Value <,'Name',Value,...>) ;
rc=obj.Close () ;
```

### SELSPEC Methods

**SELSPEC.AddFrom Method**

```c
rc=obj.AddFrom (SourceObj <,ListIndex> ) ;
```

Adds FMSG nodes from the specified source object to the SELSPEC instance. The optional `ListIndex` enables you to specify the 1-relative index in the SELSPEC’s model list where the included DAG is to be placed.

#### Arguments

You must specify the following input arguments:

- **SourceObj**

  specifies the source for models to be added to the SELSPEC instance. You can specify any of the objects shown in Table 3.14:

<table>
<thead>
<tr>
<th>SourceObj</th>
<th>Package</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARIMASPEC</td>
<td>TSM</td>
<td>Model from an ARIMASPEC object</td>
</tr>
<tr>
<td>COMBSPEC</td>
<td>ATSM</td>
<td>COMBSPEC object’s DAG</td>
</tr>
<tr>
<td>DIAGNOSE</td>
<td>ATSM</td>
<td>DIAGNOSE object’s generated models</td>
</tr>
<tr>
<td>ESMSPEC</td>
<td>TSM</td>
<td>Model from an ESMSPEC object</td>
</tr>
<tr>
<td>FORENG</td>
<td>ATSM</td>
<td>Current FORENG FMSG</td>
</tr>
<tr>
<td>IDMSPEC</td>
<td>TSM</td>
<td>Model from an IDMSPEC object</td>
</tr>
<tr>
<td>INFMSG</td>
<td>ATSM</td>
<td>INFMSG XML</td>
</tr>
<tr>
<td>SELSPEC</td>
<td>ATSM</td>
<td>SELSPEC object’s DAG</td>
</tr>
<tr>
<td>TSM</td>
<td>TSM</td>
<td>Model from a TSM object</td>
</tr>
<tr>
<td>UCMSPEC</td>
<td>TSM</td>
<td>Model from an UCMSPEC object</td>
</tr>
</tbody>
</table>

You can also specify the following input argument:
ListIndex specifies the index in the SELSPEC instance list where the model DAG from the SourceObj is to be inserted. If this argument is not specified, the DAG from the SourceObj is inserted at the next available list index.

SELSPEC.Close Method

rc = obj.Close () ;

Closes the SELSPEC object for editing. Consistency checks are performed to ensure that the model set that is defined for the SELSPEC is completely determined. This means that models must have definitions in the SELSPEC instance before it is closed. Failure to ensure this means that the SELSPEC content is nullified for subsequent use in other ATSM package interactions. For example, if the SELSPEC instance includes a specification name for which no XML is defined, then a consistency check failure occurs.

Arguments
There are no arguments associated with this method.

SELSPEC.Open Method

rc = obj.Open (nspecs ) ;

Opens the SELSPEC object for construction of a model selection list.

Input Arguments
You must specify the following input argument:

nspecs is a numeric variable that specifies the number of model specification slots to be defined in the forecast model selection graph (FMSG) list.

SELSPEC.SetDiagnose Method

rc = obj.SetDiagnose (’Name’, Value <,’Name’,Value,…) ;

Specifies options for selection diagnostics.

Input Arguments
You must specify at least one of the following ’Names’ and its associated Value:

’IDMBASE’ takes a numeric alpha Value that specifies the base value of the time series. The base value is used to determine the demand series components for an intermittent demand model. If you specify a missing value for Value, then the base value is detected automatically. The default is a missing value.

’INTERMITTENT’ takes a numeric Value greater than or equal to 0 that is used to determine whether a time series is intermittent. If the average demand interval is greater than this number, then the series is assumed to be intermittent. The default value is 2.

’LOGTEST’ takes a Boolean Value that when set to 1, enables the use of the dependent series log transform test in order to eliminate candidate models during the model selection process. The default value is 1.
'SEASONTEST' takes a numeric Value between 0 and 1 that specifies the significance probability value to use in testing whether seasonality is present in the time series. A smaller value means that stronger evidence of a seasonal pattern in the data is required before seasonal models are produced to forecast the time series. The default value is 0.01.

**SELSPEC.SetOption Method**

rc=obj.SetOption ('Name', Value <,'Name',Value,... > ) ;

Specifies named options for SELSPEC object.

**Input Arguments**

You must specify at least one of the following 'Names' and its associated Value:

- **'ALPHA'** takes a numeric Value between 0 and 1 that specifies the significance level to use in computing the confidence limits of the forecast. The default value is 0.05.
- **'CRITERION'** takes a string Value that specifies the model selection criterion (statistic of fit) to be used to select from several candidate models. The default is RMSE. For a list of valid values, see the CRITERION= option in the SELECT statement in *SAS Forecast Server Procedures: User’s Guide*.
- **'HOLDOUT'** takes a positive integer Value that specifies the size of the holdout sample to be used for model selection. The default value is 0 (no holdout sample).
- **'HOLDOUTPCT'** takes a numeric Value between 0 and 100 that specifies the size of the holdout sample as a percentage of the length of the dependent time series. If HOLDOUT=5 and HOLDOUTPCT=10, the size of the holdout sample is \( \min(5, 0.1T) \), where \( T \) is the length of the dependent time series with beginning and ending missing values removed. The default value is 100 (100%), which means that the holdout sample size is not restricted based on the series length.

---

**COMBSPEC Object**

The COMBSPEC object defines and manipulates forecast model combination list (CML) objects. The CML defines a forecast combination strategy for forecasting a dependent variable; the strategy is based on a historical information set (TSDF) and a set of contributing forecasts. The CML is a directed acyclic graph (DAG) of time series models, model selection lists, and model combination lists that are evaluated to compute a forecast that is a weighted average of a subset of the CML candidates. Competing forecasts are evaluated subject to rules that are defined in the CML when the CML is evaluated in a FORENG object. To be used, a CML must be included in a model selection list specification as one of its candidates. The COMBSPEC object offers functionality comparable to the HPFSELECT procedure when used to construct model combinations. COMBSPEC objects accept models and subgraphs from the sources that are shown in Table 3.15.
Table 3.15  Model Objects for the COMBSPEC Object

<table>
<thead>
<tr>
<th>Family</th>
<th>Package</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARIMASPEC</td>
<td>TSM</td>
<td>ARIMA model to include</td>
</tr>
<tr>
<td>COMBSPEC</td>
<td>ATSM</td>
<td>Forecast combination to include</td>
</tr>
<tr>
<td>DIAGNOSE</td>
<td>ATSM</td>
<td>Generated models to include</td>
</tr>
<tr>
<td>ESMSPEC</td>
<td>TSM</td>
<td>ESM model to include</td>
</tr>
<tr>
<td>FORENG</td>
<td>ATSM</td>
<td>FORENG current FMSG to include</td>
</tr>
<tr>
<td>IDMSPEC</td>
<td>TSM</td>
<td>IDM model to include</td>
</tr>
<tr>
<td>INFMSG</td>
<td>ATSM</td>
<td>INFMSG FMSG to include</td>
</tr>
<tr>
<td>SELSPEC</td>
<td>ATSM</td>
<td>Subgraph to include</td>
</tr>
<tr>
<td>TSM</td>
<td>TSM</td>
<td>Object’s model to include</td>
</tr>
<tr>
<td>UCMSPEC</td>
<td>TSM</td>
<td>UCM model to include</td>
</tr>
</tbody>
</table>

Table 3.16 summarizes the methods that associated with the COMBSPEC Object. TSM:XXXSpec refers to any of the ESMSPEC, ARIMASPEC, UCMSPEC, IDMSPEC, and TSM objects, which are part of the TSM package.

Table 3.16  Methods of the COMBSPEC Object

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AddFrom</td>
<td>Add from the specified source to the COMBSPEC object</td>
</tr>
<tr>
<td>Close</td>
<td>Close COMBSPEC for editing</td>
</tr>
<tr>
<td>Open</td>
<td>Open COMBSPEC for editing</td>
</tr>
<tr>
<td>SetDiagnoze</td>
<td>Set diagnose options</td>
</tr>
<tr>
<td>SetOption</td>
<td>Set combination process options</td>
</tr>
</tbody>
</table>

Figure 3.4 diagrams the relationships among the methods of the COMBSPEC object.
Figure 3.4 COMBSPEC Data Flow
COMBSPEC Synopsis

DECLARE OBJECT obj (COMBSPEC ) ;

Method syntax, in order of typical usage:

   rc=obj.Open (nspecs ) ;
   rc=obj.AddFrom (SourceObj < ,ListIndex> ) ;
   rc=obj.SetDiagnose ('Name', Value <,'Name',Value,... > ) ;
   rc=obj.SetOption ('Name', Value <,'Name',Value,... > ) ;
   rc=obj.Close () ;

COMBSPEC Methods

COMBSPEC.AddFrom Method

   rc=obj.AddFrom (SourceObj < ,ListIndex> ) ;

Adds FMSG nodes from the specified source object to the COMBSPEC instance. The optional ListIndex argument enables you to specify the 1-relative index in the SELSPEC’s model list where the included DAG is to be placed.

Input Arguments

You must specify the following input arguments:

SourceObj specifies the source for models to be added to the COMBSPEC instance. You can specify any of the objects shown in Table 3.17:

<table>
<thead>
<tr>
<th>Object</th>
<th>Package</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARIMASPEC</td>
<td>TSM</td>
<td>Model from the ARIMASPEC object</td>
</tr>
<tr>
<td>COMBSPEC</td>
<td>ATSM</td>
<td>COMBSPEC object’s DAG</td>
</tr>
<tr>
<td>DIAGNOSE</td>
<td>ATSM</td>
<td>DIAGNOSE object’s generated models</td>
</tr>
<tr>
<td>ESMSPEC</td>
<td>TSM</td>
<td>Model from the ESMSPEC object</td>
</tr>
<tr>
<td>FORENG</td>
<td>ATSM</td>
<td>Current FORENG FMSG</td>
</tr>
<tr>
<td>IDMSPEC</td>
<td>TSM</td>
<td>Model from the IDMSPEC object</td>
</tr>
<tr>
<td>INFMSG</td>
<td>ATSM</td>
<td>INFMSG FMSG</td>
</tr>
<tr>
<td>SELSPEC</td>
<td>ATSM</td>
<td>SELSPEC object’s DAG</td>
</tr>
<tr>
<td>TSM</td>
<td>TSM</td>
<td>Model from the TSM object</td>
</tr>
<tr>
<td>UCMSPEC</td>
<td>TSM</td>
<td>Model from the UCMSPEC object</td>
</tr>
</tbody>
</table>

You can also specify the following input argument:
**ListIndex** specifies the index in the COMBSPEC instance list where the model DAG from the `SourceObj` is to be inserted. If this argument is not specified, the DAG from the `SourceObj` is inserted at the next available list index.

**COMBSPEC.Close Method**

```c
rc = obj.Close();
```

Closes the COMBSPEC object for editing. Consistency checks are performed to ensure that the model set that is defined for the COMBSPEC is completely determined. This means that models must have definitions in the COMBSPEC instance before it is closed. Failure to ensure this means that the COMBSPEC content is nullified for subsequent use in other ATSM package interactions. For example, if the COMBSPEC includes a specification name for which no XML is defined, then a consistency check failure occurs.

**Arguments**

There are no arguments associated with this method.

**COMBSPEC.Open Method**

```c
rc = obj.Open(nspecs);
```

Opens the COMBSPEC object for construction of a model selection list.

**Input Arguments**

You must specify the following input arguments:

- `nspecs` is a numeric variable that specifies the number of model specification slots to be defined in the FMSG list.

**COMBSPEC.SetDiagnose Method**

```c
rc = obj.SetDiagnose('Name', Value <,'Name',Value,... >);
```

Sets options for selection diagnostics.

**Input Arguments**

You must specify at least one of the following 'Names' and its associated `Value`:

- **'IDMBASE'** takes a numeric alpha `Value` that specifies the base value of the time series. The base value is used to determine the demand series components for an intermittent demand model. If you specify a missing value for `Value`, then the base value is detected automatically. The default is a missing value.

- **'INTERMITTENT'** takes a numeric `Value` greater than and equal to 0 that is used to determine whether a time series is intermittent. If the average demand interval is greater than this number, then the series is assumed to be intermittent. The default value is 2.

- **'LOGTEST'** takes a Boolean `Value` that when set to 1, enables the use of the dependent series log transform test in order to eliminate candidate models during the model selection process. The default value is 1.
'SEASONTEST' takes a numeric Value between 0 and 1 that specifies the significance probability value to use in testing whether seasonality is present in the time series. A smaller value means that stronger evidence of a seasonal pattern in the data is required before seasonal models are produced to forecast the time series. The default value is 0.01.

**COMBSPEC.SetOption Method**

```matlab
rc = obj.SetOption ('Name', Value <,'Name',Value,...>);
```

Specifies named options for the COMBSPEC object.

**Input Arguments**

You must specify at least one of the following 'Names' and its associated Value:

- `'AICC.ABSWGT'` takes a numeric Value between 0 and 1 that specifies a lower-bound for computed weights. Computed weights whose values are less than the specified value are omitted. The remaining weights are normalized to sum to 1. This argument is used along with the AICC value for the 'WEIGHT' argument. For more information, see the 'WEIGHT' argument.

- `'AICC.BESTPCT'` takes a numeric Value between 0 and 100 that specifies the percentage of the number of candidate models to retain in the combination after any specified forecast exclusion tests have been performed. The remaining weights are normalized to sum to 1. This argument is used along with the AICC value for the 'WEIGHT' argument. For more information, see the 'WEIGHT' argument.

- `'AICC.BESTN'` takes a numeric Value that specifies the number of candidates to be retained in the combination as a percentage of the total number that are weighted. The remaining weights are normalized to sum to 1. This argument is used along with the AICC value for the 'WEIGHT' argument. For more information, see the 'WEIGHT' argument.

- `'AICC.LAMBDA'` takes a numeric Value that specifies the scale factor that is used in the computation of the AICC weights. The default value is 1, which results in the usual Akaike weights. This argument is used along with the AICC value for the 'WEIGHT' argument. For more information, see the 'WEIGHT' argument.

- `'ENCALPHA'` takes a numeric Value that specifies the encompassing test significance level. The default value is 0.05.

- `'ENCTEST'` takes a string Value that specifies the encompassing test type. The encompassing test attempts to eliminate from consideration forecasts that fail to add significant information to the final forecast. You can specify the following Values:

  - **HLN** uses the Harvey-Leybourne-Newbold (HLN) test to estimate pairwise encompassing between candidate forecasts.
  - **NONE** performs no encompassing test. This is the default option.
  - **OLS** uses an OLS-based regression test to estimate pairwise encompassing between candidate forecasts.

The default is NONE.
"HMISSPCT" takes a numeric Value between 0 and 100 that specifies a threshold for the percentage of missing forecast values in the combination horizon. This threshold is used to exclude a candidate forecast from consideration in the final combination. By default, no horizon missing percentage test is performed on candidate forecasts. The forecast horizon is the region of time in which multistep forecasts are generated.

"LAD.OBJTYPE" takes a string Value that specifies the form of the objective function that is used when the value of the 'WEIGHT' argument is LAD. For more information, see the 'WEIGHT' argument. You can specify the following Values:

- L1 specifies that the objective is an $\ell_1$ norm that involves loss series.
- LINF specifies that the objective is an $\ell_\infty$ norm that involves the loss series.

The default is L1.

"LAD.ERRTYPE" takes a string Value that specifies the form of the loss series in the objective function that is used when the value of the 'WEIGHT' argument is LAD. For more information, see the 'WEIGHT' argument. You can specify the following Values:

- ABS specifies that loss series terms are deviations.
- APE specifies that loss series terms are percentage deviations.
- RAE specifies that loss series terms are relative error deviations.

The default is ABS.

"MISSMODE" takes a string Value that specifies a method for treating missing values in the forecast combination. In a particular time slice across the combination ensemble, one or more combination contributors can have a missing value. This Value determines the treatment of contributors in the final combination for such time indices. You can specify the following Values:

- MISSING generates a missing combined forecast at each time index with one or more missing contributors.
- RESCALE rescales the combination weights for the nonmissing contributors at each time index so that they sum to 1. You cannot specify RESCALE when the value of the 'WEIGHT' argument is OLS and NRLS.

The default value depends on the combination weight method specified in the 'WEIGHT' argument. RESCALE is the default for simple average, user-specified weights, ranked user weights, ranked weights, and RMSE weights. MISSING is the default for all other methods.

"MISSPCT" takes a numeric Value between 0 and 100 that specifies a threshold for the percentage of missing forecast values in the combination estimation region. This threshold is used to exclude a candidate forecast from consideration in the final combination. By default, no missing percentage test is performed on candidate forecasts.

"RANKING" takes a string Value that specifies the forecast ranking criterion (statistics of fit) mnemonic to be used when ranking forecast candidates. For a list of valid values, see the CRITERION= option in the HPFDIAGNOSE procedure in SAS Forecast Server Procedures: User’s Guide. The default value is RMSE.
Chapter 3: Automatic Time Series Modeling Package

'SEMODE' takes a string Value that specifies the method for computing the prediction error variance series. This series is used to compute the prediction standard error, which in turn is used to compute confidence bands on the combined forecast. You can specify the following Values:

**DIAG** computes the prediction error variance by assuming that the forecast errors at time t are uncorrelated so that the simple diagonal form of $\Sigma_t$ is used.

**ESTCORR** computes the prediction error variance by using estimates of $\rho_{i,j,t}$, the sample cross-correlation between $e_{i,t}$ and $e_{j,t}$ over the time span $t = 1, \ldots, T$, where $t$ denotes the last time index of the actual series $y_t$. Of course, this option implies that the error series $e_{i,t}$ and $e_{j,t}$ are assumed to be jointly stationary.

The default is DIAG.

'USERWEIGHTS' takes a numeric array Value that specifies the combinations weights that are used when the value of the 'WEIGHT' argument is RANKWGT or USERDEF. For more information, see the 'WEIGHT' argument.

'WEIGHT' takes a string Value that specifies the method for determining the combination of weights that are used in the weighted average of the candidate forecasts in the combination list. You can specify the following Values:

**AICC** computes the combination weights based on corrected AIC weights. By default, all AICC-scored candidate forecasts are combined. Frequently there is considerable disparity between the weights because of the exponential weighting scheme, so additional options are provided to affect the scaling and to cull low-scoring candidates from consideration for computational efficiency. For more information, see the 'AICC.ABSWGT', 'AICC.BESTPCT', and 'AICC.LAMBDA' arguments. You can specify one of these additional options when you specify the value AICC for the 'WEIGHT' argument. By default, the 'AICC.LAMBDA' argument is chosen and set to 1 when the value of the 'WEIGHT' argument is AICC.

**AVERAGE** computes the simple average of the forecasts that are selected for combination.

**ERLS** computes the combination weights based on a constrained least squares problem to minimize the $\ell_2$ norm of the combined forecast residuals subject to the constraint that the weights sum to 1.

**LAD** computes the weights based on a least absolute deviations (LAD) measure of fit for the combined forecast. A linear program is formulated, where an objective function to be minimized is expressed in terms of the absolute values of a loss series subject to constraints that the weights sum to 1 and be nonnegative. You can specify the form of the objective function in the 'LAD.OBJTYPE' argument and the form of the loss series in the objective function in the 'LAD.ERRTYPE' argument.
NERLS computes the combination weights based on a constrained least squares problem to minimize the $\ell_2$ norm of the combined forecast residuals subject to the constraints that the weights sum to 1 and be nonnegative.

NRLS is equivalent to NERLS except that the resulting combination weights are not constrained to sum up to 1.

OLS computes the combination weights that result from the ordinary least squares problem to minimize the $\ell_2$ norm of the combined forecast residuals.

RANKWGT assigns weights by using the rank of the candidate forecasts at the time the combination is performed. You must specify the assigned weights in the 'USERWEIGHTS' argument. The number of specified values must agree with the number of model families that are specified in the current instance of the DIAGSPEC object. These weights must sum to 1. The weights are assigned by ranking the candidate forecasts from best to worst. The best uses the first weight, $W_1$, and so on. The set of weights that are used is normalized to account for candidates that fail to forecast or for candidates that are omitted from the final combination.

RMSEWGT computes the combination weights based on the RMSE statistic of fit for the forecast contributors. The weights are normalized to sum to 1.

USERDEF assigns weights by using the list of user-specified values. You must specify the assigned weights in the 'USERWEIGHTS' argument. The number of specified values must agree with the number of model families that are specified in the current instance of the DIAGSPEC object. The weights correspond to the order of specification of the model families. These weights must sum to 1. The set of weights that are used is normalized to account for candidates that fail to forecast or for candidates that are omitted from the final combination.

The default is AVERAGE.
OUTDIAG Object

The OUTDIAG object collects and stores diagnostic control specifications.

Table 3.18 shows the contents of the OUTDIAG object.

<table>
<thead>
<tr>
<th>Column</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>DIAGSPEC</em></td>
<td>String</td>
<td>Diagnostic specification XML document</td>
</tr>
<tr>
<td><em>NAME</em></td>
<td>String</td>
<td>Name of the dependent variable (can be missing)</td>
</tr>
<tr>
<td><em>SPECLEN</em></td>
<td>Numeric</td>
<td>Length of the diagnostic specification XML</td>
</tr>
<tr>
<td><em>SPECNAME</em></td>
<td>String</td>
<td>Name of the diagnostic specification</td>
</tr>
</tbody>
</table>

Table 3.19 summarizes the methods that are associated with the OUTDIAG Object.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Collect</td>
<td>Collect diagnostic specification from the DIAGSPEC and DIAGNOSE objects</td>
</tr>
<tr>
<td>nrows</td>
<td>Query the number of rows in the OUTDIAG object</td>
</tr>
</tbody>
</table>

OUTDIAG Synopsis

DECLARE OBJECT obj (OUTDIAG) ;

Method syntax, in order of typical usage:

\[
rc=\text{obj}.\text{Collect} (\text{SourceObject} ) ;
nrows=\text{obj}.\text{nrows} () ;
\]
OUTDIAG Methods

OUTDIAG.Collect Method

rc=obj.Collect(SourceObject);

Collects the diagnostic control specification from the source object.

**Input Arguments**
You must specify the following input argument:

*SourceObject* specifies the instance to use as the source of the diagnostic control option specification to be stored. You can specify one of the following:

- **DIAGNOSE** renders DIAGSPEC XML from a DIAGNOSE instance.
- **DIAGSPEC** renders DIAGSPEC XML from a DIAGSPEC instance.

OUTDIAG.nrows Method

nrows=obj.nrows();

Queries the OUTDIAG object for its current row count.

**Arguments**
There are no arguments associated with this method.

OUTEST Object

The OUTEST object collects parameter estimates from a FORENG instance. The CAS table schema that is used for storing the parameter estimates is compatible with the schema that is used by the HPFENGINE procedure for its OUTEST= data set.

Table 3.20 shows the contents of the OUTEST object.
Chapter 3: Automatic Time Series Modeling Package

Table 3.20 Contents of the OUTEST Object

<table>
<thead>
<tr>
<th>Column</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>COMPMODEL</em></td>
<td>String</td>
<td>Component model name</td>
</tr>
<tr>
<td><em>COMPONENT</em></td>
<td>String</td>
<td>Component name within the component model</td>
</tr>
<tr>
<td><em>DSVAR</em></td>
<td>String</td>
<td>Corresponding variable name</td>
</tr>
<tr>
<td><em>EST</em></td>
<td>Numeric</td>
<td>Parameter estimate</td>
</tr>
<tr>
<td><em>FACTOR</em></td>
<td>Numeric</td>
<td>Factor number</td>
</tr>
<tr>
<td><em>LABEL</em></td>
<td>String</td>
<td>System-generated label for model</td>
</tr>
<tr>
<td><em>LAG</em></td>
<td>Numeric</td>
<td>Lag that is used</td>
</tr>
<tr>
<td><em>MODEL</em></td>
<td>String</td>
<td>Name of model in the selection list</td>
</tr>
<tr>
<td><em>MODELVAR</em></td>
<td>String</td>
<td>Symbol used in the model specification</td>
</tr>
<tr>
<td><em>NAME</em></td>
<td>String</td>
<td>Name of the dependent variable</td>
</tr>
<tr>
<td><em>PARM</em></td>
<td>String</td>
<td>Parameter name</td>
</tr>
<tr>
<td><em>PVALUE</em></td>
<td>Numeric</td>
<td>( p )-value for parameter estimate</td>
</tr>
<tr>
<td><em>SELECT</em></td>
<td>String</td>
<td>Name of the selection list</td>
</tr>
<tr>
<td><em>STATUS</em></td>
<td>Numeric</td>
<td>Indicates success/failure in estimating parameter.</td>
</tr>
<tr>
<td><em>SHIFT</em></td>
<td>Numeric</td>
<td>Shift that is used</td>
</tr>
<tr>
<td><em>STDERR</em></td>
<td>Numeric</td>
<td>Parameter estimate standard error</td>
</tr>
<tr>
<td><em>TRANSFORM</em></td>
<td>String</td>
<td>Transform that is used for the dependent variable</td>
</tr>
<tr>
<td><em>TVALUE</em></td>
<td>Numeric</td>
<td>( t ) statistic for parameter estimate</td>
</tr>
<tr>
<td><em>VARTYPE</em></td>
<td>String</td>
<td>Type of the variable (dependent or independent)</td>
</tr>
</tbody>
</table>

The _STATUS_ column of the OUTEST object gives the forecasting status of each candidate model that is diagnosed by the FORENG object. Table 3.21 lists the possible values of the _STATUS_ column along with brief explanations of their meaning.
Table 3.21  Description of Forecasting Status in _STATUS_ Column

<table>
<thead>
<tr>
<th>Status</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>The forecast was successfully completed.</td>
</tr>
<tr>
<td>3000</td>
<td>Model selection could not be completed. Forecast values are set to missing.</td>
</tr>
<tr>
<td>3001</td>
<td>Model selection could not be completed and the 'NOALTLIST' option prohibits use of default exponential smoothing. Forecast values are set to missing.</td>
</tr>
<tr>
<td>3002</td>
<td>The forecast was completed subject to qualification that one or more input variables were omitted from the selected model. This can only occur in the context of ARIMAX or UCM models.</td>
</tr>
<tr>
<td>3003</td>
<td>The desired model could not be forecast. The forecast reverted to the default exponential smoothing model.</td>
</tr>
<tr>
<td>3004</td>
<td>The attempt to forecast the desired model produced an arithmetic exception. The forecast reverted to the default exponential smoothing model.</td>
</tr>
<tr>
<td>3005</td>
<td>The attempt to forecast the desired model produced an arithmetic exception. The forecast reverted to a zero-drift random walk model.</td>
</tr>
<tr>
<td>3006</td>
<td>The attempt to forecast the desired model produced an arithmetic exception. Forecast values are set to missing.</td>
</tr>
<tr>
<td>3007</td>
<td>The mean value forecast is generated as a result of the 'MINOBS' option.</td>
</tr>
<tr>
<td>3008</td>
<td>There were insufficient non-missing observations in the variable to be forecast. A missing value forecast is produced.</td>
</tr>
<tr>
<td>3009</td>
<td>There were insufficient non-zero observations in the variable to be forecast. A zero-valued forecast is produced.</td>
</tr>
</tbody>
</table>

Table 3.22 summarizes the methods that are associated with the OUTEST Object.

Table 3.22  Methods of the OUTEST Object

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Collect</td>
<td>Collect parameter estimates from the FORENG instance</td>
</tr>
<tr>
<td>nrows</td>
<td>Get the OUTEST row count</td>
</tr>
</tbody>
</table>
OUTEST Synopsis

DECLARE OBJECT obj (OUTEST ) ;

Method syntax, in order of typical usage:

\[
rc = obj.\text{Collect} (\text{FORENGObj}) ;
\]

\[
nrows = obj.nrows () ;
\]

OUTEST Methods

OUTEST.Collect Method

\[
rc = obj.\text{Collect} (\text{FORENGObj}) ;
\]

Collects the parameter estimates from the FORENG instance \text{FORENGObj}.

Input Arguments

You must specify the following input argument:

\text{FORENGObj} specifies the FORENG object instance to use as the source of time series model parameter estimates.

OUTEST.nrows Method

\[
nrows = obj.nrows () ;
\]

Gets the current row count from the OUTEST instance.

Arguments

There are no arguments associated with this method.
OUTEVENT Object

The OUTEVENT object collects the definitions for event variables that are specified in an instance of the EVENT object. Table 3.23 shows the contents of the OUTEVENT object.

<table>
<thead>
<tr>
<th>Column</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>SEQ</em></td>
<td>Numeric</td>
<td>Sequence number to sort observations for the HPFEVENTS procedure. This variable is not required by the INEVENT object.</td>
</tr>
<tr>
<td><em>LOCALE</em></td>
<td>String</td>
<td>Locale to which the event applies. This variable is not required by the INEVENT object. when the 'VERSION','HPFEVENTS' pair is specified.</td>
</tr>
<tr>
<td><em>NAME</em></td>
<td>String</td>
<td>Name of the event variable</td>
</tr>
<tr>
<td><em>CLASS</em></td>
<td>String</td>
<td>Class of the event</td>
</tr>
<tr>
<td><em>KEYNAME</em></td>
<td>String</td>
<td>Name of date keyword or event</td>
</tr>
<tr>
<td><em>STARTDATE</em></td>
<td>Numeric</td>
<td>Starting date timing value</td>
</tr>
<tr>
<td><em>ENDDATE</em></td>
<td>Numeric</td>
<td>Ending date timing value</td>
</tr>
<tr>
<td><em>DATEINTRVL</em></td>
<td>String</td>
<td>Interval for date do-list</td>
</tr>
<tr>
<td><em>STARTDT</em></td>
<td>Numeric</td>
<td>Starting datetime timing value</td>
</tr>
<tr>
<td><em>ENDDT</em></td>
<td>Numeric</td>
<td>Ending datetime timing value</td>
</tr>
<tr>
<td><em>DTINTRVL</em></td>
<td>String</td>
<td>Interval for datetime do-list</td>
</tr>
<tr>
<td><em>STARTOBS</em></td>
<td>Numeric</td>
<td>Starting observation timing value</td>
</tr>
<tr>
<td>_ ENDOBS_</td>
<td>Numeric</td>
<td>Ending observation timing value</td>
</tr>
<tr>
<td><em>OBSINTRVL</em></td>
<td>Numeric or string</td>
<td>Interval for observation do-list. This column contains a numeric value (such as 5) when the 'VERSION','HPFEVENTS' pair is specified, or a string value (such as 'OBS5') when the 'VERSION','TSMODEL' pair is specified.</td>
</tr>
<tr>
<td><em>TYPE</em></td>
<td>String</td>
<td>Type (basic shape) of event</td>
</tr>
<tr>
<td><em>VALUE</em></td>
<td>Numeric</td>
<td>Full amplitude of nonzero observation</td>
</tr>
<tr>
<td><em>PULSE</em></td>
<td>String</td>
<td>Interval unit for <em>DUR_BEFORE</em> or <em>DUR_AFTER</em> values</td>
</tr>
<tr>
<td><em>DUR_BEFORE</em></td>
<td>Numeric</td>
<td>Number of durations before the timing value</td>
</tr>
<tr>
<td><em>DUR_AFTER</em></td>
<td>Numeric</td>
<td>Number of durations after the timing value</td>
</tr>
<tr>
<td><em>SLOPE_BEFORE</em></td>
<td>String</td>
<td>Type of slope (growth or decay) before the timing value</td>
</tr>
<tr>
<td><em>SLOPE_AFTER</em></td>
<td>String</td>
<td>Type of slope (growth or decay) after the timing value</td>
</tr>
<tr>
<td><em>SHIFT</em></td>
<td>Numeric</td>
<td>Number of pulse intervals to shift the timing value</td>
</tr>
</tbody>
</table>
Table 3.23 continued

<table>
<thead>
<tr>
<th>Column</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>TCPARM</em></td>
<td>Numeric</td>
<td>Rate of growth or decay when the 'TYPE',TC pair is specified</td>
</tr>
<tr>
<td><em>RULE</em></td>
<td>String</td>
<td>Rule for combining events or overlapping timing values</td>
</tr>
<tr>
<td><em>PERIOD</em></td>
<td>String</td>
<td>Frequency for repeating event</td>
</tr>
<tr>
<td><em>LABEL</em></td>
<td>String</td>
<td>Label for event</td>
</tr>
</tbody>
</table>

Table 3.24 summarizes the methods that are associated with the OUTEVENT object.

Table 3.24 Methods of the OUTEVENTDUMMY Object

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Collect</td>
<td>Collect the event variable definitions that are in the EVENT instance</td>
</tr>
<tr>
<td>nrows</td>
<td>Get the OUTEVENT row count</td>
</tr>
</tbody>
</table>

OUTEVENT Synopsis

DECLARE OBJECT obj (OUTEVENT < ('VERSION','source') > ) ;

You can specify one of the following values for source:

'HPFEVENTS' specifies the CAS table schema that is compatible with the schema that is used by the HPFEVENTS procedure for the data sets that are specified in the IN= and OUT= options in its EVENTDATA statement. The variable _OBSINTRVL_ contains a numeric value for the multiplier for the observation do-list interval, if any. For example, a value of 5 would represent 'OBS5'.

'TSMODEL' specifies the CAS table schema that is compatible with the TSMODEL procedure. This CAS table schema is similar to the schema that is used when 'VERSION','HPFEVENTS' is specified, with a few modifications. The variable _OBSINTRVL_ contains a string (such as 'OBS5') for the observation do-list interval, if any.

The default is 'TSMODEL' in order to promote use of the newer TSMODEL procedure. Usually, you can use the default outobj=< collector> object as input for the corresponding default inobj=< repeater>, but the default for the INEVENT object is 'HPFEVENTS'. So if you want to use your default OUTEVENT table as input to an INEVENT object, be sure to specify ('VERSION','TSMODEL') when you use an INEVENT table. The defaults might change in future releases.

Method syntax, in order of typical usage:
**OUTEVENT Methods**

**OUTEVENT.Collect Method**

\[ rc = obj.Collect(EVENTObj) ; \]

Collects the event variable definitions in the EVENT instance that is specified in the \( EVENTObj \) argument.

*Input Arguments*

You must specify the following input argument:

\( EVENTObj \) specifies the EVENT object instance to use as the source of event variable definitions.

**OUTEVENT.nrows Method**

\[ nrows = obj.nrows() ; \]

Gets the current row count from the specified OUTEVENT object.

*Arguments*

There are no arguments associated with this method.

---

**OUTEVENTDUMMY Object**

The OUTEVENTDUMMY object collects the series for event dummy variables that are in the TSDF instance. The CAS table schema that is used for storing the event variable series is compatible with the schema that is used by the HPFENGINE procedure for its OUTINDEP= data set.

Table 3.25 shows the contents of the OUTEVENTDUMMY object.

<table>
<thead>
<tr>
<th>Column</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>TIMEID</em></td>
<td>Numeric</td>
<td>Uniform time ID values for series</td>
</tr>
<tr>
<td><em>XVAR</em></td>
<td>String</td>
<td>Name of the event variable</td>
</tr>
<tr>
<td>X</td>
<td>Numeric</td>
<td>Event (X) variable value</td>
</tr>
</tbody>
</table>

Table 3.26 summarizes the methods that are associated with the OUTEVENTDUMMY Object.
### Table 3.26 Methods of the OUTEVENTDUMMY Object

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Collect</td>
<td>Collect the event variable series that are in the TSDF instance</td>
</tr>
<tr>
<td>nrows</td>
<td>Get the OUTEVENTDUMMY row count</td>
</tr>
</tbody>
</table>

### OUTEVENTDUMMY Synopsis

```plaintext
DECLARE OBJECT obj (OUTEVENTDUMMY) ;
```

Method syntax, in order of typical usage:

```plaintext
rc = obj.Collect (TSDFObj) ;
rc = obj.nrows () ;
```

### OUTEVENTDUMMY Methods

#### OUTEVENTDUMMY.Collect Method

```plaintext
rc = obj.Collect (TSDFObj) ;
```

Collects the event variable series that are in the TSDF instance `TSDFObj`.

**Input Arguments**

You must specify the following input argument:

- `TSDFObj` specifies the TSDF object instance to use as the source of event variable series.

#### OUTEVENTDUMMY.nrows Method

```plaintext
nrows = obj.nrows () ;
```

Gets the current row count from the OUTEVENTDUMMY instance.

**Arguments**

There are no arguments associated with this method.
OUTFOR Object

The OUTFOR object collects forecast series from a FORENG instance. The CAS table schema that is used for storing the set of forecast series variables is compatible with the schema that is used by the HPFENGINE procedure for its OUTFOR= data set.

Table 3.27 shows the contents of the OUTFOR object.

<table>
<thead>
<tr>
<th>Column</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>NAME</em></td>
<td>String</td>
<td>Name of the dependent variable</td>
</tr>
<tr>
<td><em>TIMEID</em></td>
<td>Numeric</td>
<td>Uniform time ID values for series</td>
</tr>
<tr>
<td>ACTUAL</td>
<td>Numeric</td>
<td>Accumulated values of dependent variable</td>
</tr>
<tr>
<td>ERROR</td>
<td>Numeric</td>
<td>Residuals</td>
</tr>
<tr>
<td>LOWER</td>
<td>Numeric</td>
<td>Lower confidence limit</td>
</tr>
<tr>
<td>PREDICT</td>
<td>Numeric</td>
<td>Forecasts of dependent variable</td>
</tr>
<tr>
<td>STD</td>
<td>Numeric</td>
<td>Prediction standard error</td>
</tr>
<tr>
<td>UPPER</td>
<td>Numeric</td>
<td>Upper confidence limit</td>
</tr>
</tbody>
</table>

Table 3.28 summarizes the methods that are associated with the OUTFOR Object.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Collect</td>
<td>Collect forecasts estimates from FORENG instance.</td>
</tr>
<tr>
<td>nrows</td>
<td>Get the OUTFOR row count.</td>
</tr>
</tbody>
</table>
OUTFOR Synopsis

DECLARE OBJECT obj (OUTFOR) ;

Method syntax, in order of typical usage:

\[ rc = \text{obj}.\text{Collect} (\text{FORENGObj} < , \text{Region} > ) ; \]
\[ rc = \text{obj}.\text{nrows} () ; \]

OUTFOR Methods

OUTFOR.Collect Method

\[ rc = \text{obj}.\text{Collect} (\text{FORENGObj} < , \text{Region} > ) ; \]

Collects the forecast series from the FORENG instance forecast object.

Input Arguments
You must specify the following input argument:

FORENGObj specifies the FORENG object instance to use as the source of forecast series.

You can also specify the following input argument:

Region specifies the time region over which to collect the forecast series. You can specify the following values for Region:

**string** specifies the collection region. You can specify the following strings:

- **ALL**: collects over the entire time span of the available data.
- **FIT**: collects over the time region that supplied observations for estimating model parameters (that is, the model fit region).
- **FORECAST**: collects over the time region that is subsequent to the FIT region and that did not contribute any data to the model parameter estimation process (that is, the model forecast region).

The default is ALL.

**numeric** is a two-valued numeric array in which the first value specifies the starting time ID and the second value specifies the ending time ID of the time region over which the forecast series are to be collected. Either the starting time ID or the ending time ID can be a missing value. If both are missing values, then the default value ALL is used.
OUTFOR.nrows Method

```
rc=obj.nrows();
```

Gets the current row count from the OUTFOR instance.

**Arguments**

There are no arguments associated with this method.

---

**OUTCOMP Object**

The OUTCOMP object collects component series from a FORENG instance. The CAS table schema that is used for storing the set of forecast component series variables is compatible with the schema that is used by the HPFENGINE procedure for its OUTCOMP= data set.

Table 3.29 shows the contents of the OUTCOMP object.

<table>
<thead>
<tr>
<th>Column</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>ACTUAL</em></td>
<td>Numeric</td>
<td>Accumulated values of dependent variable</td>
</tr>
<tr>
<td><em>COMP</em></td>
<td>String</td>
<td>Name of the forecast component series</td>
</tr>
<tr>
<td><em>LOWER</em></td>
<td>Numeric</td>
<td>Lower confidence limit</td>
</tr>
<tr>
<td><em>NAME</em></td>
<td>String</td>
<td>Name of the dependent variable</td>
</tr>
<tr>
<td><em>PREDICT</em></td>
<td>Numeric</td>
<td>Forecasts of dependent variable</td>
</tr>
<tr>
<td><em>STD</em></td>
<td>Numeric</td>
<td>Prediction standard error</td>
</tr>
<tr>
<td><em>TIMEID</em></td>
<td>Numeric</td>
<td>Uniform time ID values for series</td>
</tr>
<tr>
<td><em>UPPER</em></td>
<td>Numeric</td>
<td>Upper confidence limit</td>
</tr>
</tbody>
</table>

Table 3.30 summarizes the methods that are associated with the OUTCOMP Object.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Collect</td>
<td>Collect model component series from the FORENG instance</td>
</tr>
<tr>
<td>nrows</td>
<td>Get the OUTCOMP row count</td>
</tr>
</tbody>
</table>
OUTCOMP Synopsis

DECLARE OBJECT obj (OUTCOMP) ;

Method syntax, in order of typical usage:

\[
rc = obj.\text{Collect}(\text{FORENGObj}, \text{Region}) ;
\]

\[
rc = obj.\text{nrows}() ;
\]

OUTCOMP Methods

OUTCOMP.Collect Method

\[
rc = obj.\text{Collect}(\text{FORENGObj}, \text{Region}) ;
\]

Collects the model component series from the FORENG instance \text{FORENGObj}.

Input Arguments
You must specify the following input argument:

\text{FORENGObj} specifies the FORENG object instance to use as the source of time series model parameter estimates.

You can also specify the following input argument:

\text{Region} specifies the time region over which to collect the forecast series. You can specify the following values for \text{Region}:

\text{string} specifies the collection region. You can specify the following strings:

\text{ALL} collects over the entire time span of the available data.

\text{FIT} collects over the time region that supplied observations for estimating model parameters (that is, the model fit region).

\text{FORECAST} collects over the time region that is subsequent to the FIT region and that did not contribute any data to the model parameter estimation process (that is, the model forecast region).

The default is ALL.

\text{numeric} is a two-valued numeric array in which the first value specifies the starting time ID and the second value specifies the ending time ID of the time region over which the forecast series are to be collected. Either the starting time ID or the ending time ID can be a missing value. If both are missing values, then the default value ALL is used.
OUTINDEP Object

The OUTINDEP object collects the series for independent variables that are used in the forecast from a FORENG instance. The CAS table schema that is used for storing the independent variable series is compatible with the schema that is used by the HPFENGINE procedure for its OUTINDEP= data set.

Table 3.31 shows the contents of the OUTINDEP object.

<table>
<thead>
<tr>
<th>Column</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>NAME</em></td>
<td>String</td>
<td>Name of the dependent variable</td>
</tr>
<tr>
<td><em>TIMEID</em></td>
<td>Numeric</td>
<td>Uniform time ID values for series</td>
</tr>
<tr>
<td><em>XVAR</em></td>
<td>String</td>
<td>Name of the independent variable</td>
</tr>
<tr>
<td>X</td>
<td>Numeric</td>
<td>Independent (X) variable value</td>
</tr>
</tbody>
</table>

Table 3.32 summarizes the methods that are associated with the OUTINDEP Object.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Collect</td>
<td>Collect the independent variable series that are used in the forecast from the FORENG instance forecast object</td>
</tr>
<tr>
<td>nrows</td>
<td>Get the OUTINDEP row count</td>
</tr>
</tbody>
</table>
OUTINDEP Synopsis

DECLARE OBJECT obj (OUTINDEP) ;

Method syntax, in order of typical usage:

rc = obj.Collect (FORENGObj);
rc = obj.nrows();

OUTINDEP Methods

OUTINDEP.Collect Method

rc = obj.Collect (FORENGObj < , 'VARTYPE') ;

Collects the independent variable series that are used in the forecast from the FORENG instance FORENGObj.

Input Arguments

You must specify the following input argument:

FORENGObj specifies the FORENG object instance to use as the source of time series model independent variable series.

You can also specify the following input argument:

VARTYPE specifies the type of variable to collect from the model in the FORENG instance. You can specify the following values for VARTYPE:

string specifies the type of variable. You can specify the following strings:

ALL collects both inputs and events.
EVENT collects only event variables.
INPUT collects only input variables.

The default is ALL.

OUTINDEP.nrows Method

rc = obj.nrows();

Gets the current row count from the OUTINDEP instance.

Arguments

There are no arguments associated with this method.
OUTMODELINFO Object

The OUTMODELINFO object collects characteristics of the selected model from a FORENG instance. The CAS table schema that is used for storing model information is compatible with the schema that is used by the HPFENGINE for its OUTMODELINFO= data set.

Table 3.33 shows the contents of the OUTMODELINFO object.

<table>
<thead>
<tr>
<th>Column</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>DEPTRANS</em></td>
<td>String</td>
<td>Dependent variable transform that is used</td>
</tr>
<tr>
<td><em>EVENTS</em></td>
<td>Numeric</td>
<td>Number of events in the model</td>
</tr>
<tr>
<td><em>INPUTS</em></td>
<td>Numeric</td>
<td>Number of input variables in the model</td>
</tr>
<tr>
<td><em>MODEL</em></td>
<td>String</td>
<td>Name of the selected model specification</td>
</tr>
<tr>
<td><em>MODELTYPE</em></td>
<td>String</td>
<td>Type of model (ESM, ARIMA, UCM, or IDM)</td>
</tr>
<tr>
<td><em>NAME</em></td>
<td>String</td>
<td>Name of the dependent variable</td>
</tr>
<tr>
<td><em>OUTLIERS</em></td>
<td>Numeric</td>
<td>Number of outlier events in the model</td>
</tr>
<tr>
<td><em>SEASONAL</em></td>
<td>Numeric</td>
<td>Seasonal model (0 or 1 indicator)</td>
</tr>
<tr>
<td><em>SOURCE</em></td>
<td>String</td>
<td>Named source of the model</td>
</tr>
<tr>
<td><em>STATUS</em></td>
<td>Numeric</td>
<td>Execution status of the model</td>
</tr>
<tr>
<td><em>TREND</em></td>
<td>Numeric</td>
<td>Trend model (0 or 1 indicator)</td>
</tr>
</tbody>
</table>

The _STATUS_ column of the OUTMODELINFO object gives the forecasting status of the selected model. See Table 3.21 for a list of possible values of the _STATUS_ column along with brief explanations of their meaning. Table 3.34 summarizes the methods that are associated with the OUTMODELINFO Object.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Collect</td>
<td>Collect the selected model information from the FORENG instance FORENGObj</td>
</tr>
<tr>
<td>nrows</td>
<td>Get the current row count from the OUTMODELINFO instance</td>
</tr>
</tbody>
</table>
OUTMODELINFO Synopsis

DECLARE OBJECT obj (OUTMODELINFO) ;

Method syntax, in order of typical usage:

rc = obj.Collect (FORENGObj) ;
rc = obj.nrows () ;

OUTMODELINFO Methods

OUTMODELINFO.Collect Method

rc = obj.Collect (FORENGObj) ;

Collects the selected model information from the FORENG instance FORENGObj.

Input Arguments
You must specify the following input argument:

FORENGObj specifies the FORENG object instance to use as the source of selected model information.

OUTMODELINFO.nrows Method

rc = obj.nrows () ;

Gets the current row count from the OUTMODELINFO instance.

Arguments
There are no arguments associated with this method.

OUTSELECT Object

The OUTSELECT object collects model selection statistics from a FORENG instance. This information is useful for comparing the performance of various models. The CAS table schema that is used for storing the fit statistics is compatible with the schema that is used by the HPFENGINE procedure for its OUTSTATSELECT= data set.

Table 3.35 shows the contents of the OUTSELECT object.

<table>
<thead>
<tr>
<th>Column</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>MODEL</em></td>
<td>String</td>
<td>Model specification name</td>
</tr>
<tr>
<td><em>NAME</em></td>
<td>String</td>
<td>Variable name</td>
</tr>
</tbody>
</table>
**Table 3.35  continued**

<table>
<thead>
<tr>
<th>Column</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>REGION</em></td>
<td>String</td>
<td>Region in which the statistics are calculated. Values in the <em>REGION</em> variable include:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>FIT</td>
</tr>
<tr>
<td></td>
<td></td>
<td>indicates that fit statistics were calculated over the fit region.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>FORECAST</td>
</tr>
<tr>
<td></td>
<td></td>
<td>indicates that fit statistics were calculated over the forecast region.</td>
</tr>
<tr>
<td><em>SELECT</em></td>
<td>String</td>
<td>Name of model selection list to which <em>MODEL</em> belongs</td>
</tr>
<tr>
<td><em>SELECTED</em></td>
<td>String</td>
<td>Indicates whether <em>MODEL</em> was chosen to forecast the dependent series or used by the chosen forecast when the chosen forecast is a combined model. Values in the <em>SELECTED</em> variable include:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>NO</td>
</tr>
<tr>
<td></td>
<td></td>
<td>indicates that <em>MODEL</em> is neither selected nor used.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>YES</td>
</tr>
<tr>
<td></td>
<td></td>
<td>indicates that <em>MODEL</em> is the primary model selected for the forecast.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>USED</td>
</tr>
<tr>
<td></td>
<td></td>
<td>indicates that <em>MODEL</em> is used by the primary model in producing the final forecast.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>USED_SELECT</td>
</tr>
<tr>
<td></td>
<td></td>
<td>indicates that <em>MODEL</em> is used by the primary model in the model selection region, but not in producing the final forecast.</td>
</tr>
<tr>
<td>AADJRSQ</td>
<td>Numeric</td>
<td>Amemiya’s adjusted R-square</td>
</tr>
<tr>
<td>ADJRSQ</td>
<td>Numeric</td>
<td>Adjusted R-square</td>
</tr>
<tr>
<td>AIC</td>
<td>Numeric</td>
<td>Akaike’s information criterion</td>
</tr>
<tr>
<td>AICC</td>
<td>Numeric</td>
<td>Finite sample corrected AIC</td>
</tr>
<tr>
<td>APC</td>
<td>Numeric</td>
<td>Amemiya’s prediction criterion</td>
</tr>
<tr>
<td>DFE</td>
<td>Numeric</td>
<td>Degrees of freedom error</td>
</tr>
<tr>
<td>GMAPE</td>
<td>Numeric</td>
<td>Geometric mean absolute percentage error</td>
</tr>
<tr>
<td>GMAPES</td>
<td>Numeric</td>
<td>Geometric mean absolute error percentage of standard deviation</td>
</tr>
<tr>
<td>GMAPPE</td>
<td>Numeric</td>
<td>Geometric mean absolute predictive percentage error</td>
</tr>
<tr>
<td>GMRAE</td>
<td>Numeric</td>
<td>Geometric mean relative absolute error</td>
</tr>
<tr>
<td>GMASPE</td>
<td>Numeric</td>
<td>Geometric mean absolute symmetric percentage error</td>
</tr>
<tr>
<td><em>LABEL</em></td>
<td>String</td>
<td>Descriptive label for the variable name in <em>NAME</em></td>
</tr>
<tr>
<td>Column</td>
<td>Type</td>
<td>Description</td>
</tr>
<tr>
<td>----------</td>
<td>--------</td>
<td>------------------------------------------------------------------</td>
</tr>
<tr>
<td>MAE</td>
<td>Numeric</td>
<td>Mean absolute error</td>
</tr>
<tr>
<td>MAPE</td>
<td>Numeric</td>
<td>Mean absolute percentage error</td>
</tr>
<tr>
<td>MAPES</td>
<td>Numeric</td>
<td>Mean absolute error percentage of standard deviation</td>
</tr>
<tr>
<td>MAPPE</td>
<td>Numeric</td>
<td>Symmetric mean absolute predictive percentage error</td>
</tr>
<tr>
<td>MASE</td>
<td>Numeric</td>
<td>Mean absolute scaled error</td>
</tr>
<tr>
<td>MAXAPES</td>
<td>Numeric</td>
<td>Maximum absolute error percentage of standard deviation</td>
</tr>
<tr>
<td>MAXERR</td>
<td>Numeric</td>
<td>Maximum error</td>
</tr>
<tr>
<td>MAXPE</td>
<td>Numeric</td>
<td>Maximum percentage error</td>
</tr>
<tr>
<td>MAXPPE</td>
<td>Numeric</td>
<td>Maximum predictive percentage error</td>
</tr>
<tr>
<td>MAXRE</td>
<td>Numeric</td>
<td>Maximum relative error</td>
</tr>
<tr>
<td>MAXSPE</td>
<td>Numeric</td>
<td>Maximum symmetric percentage error</td>
</tr>
<tr>
<td>MDAPE</td>
<td>Numeric</td>
<td>Median absolute percentage error</td>
</tr>
<tr>
<td>MDAPES</td>
<td>Numeric</td>
<td>Median absolute error percentage of standard deviation</td>
</tr>
<tr>
<td>MDAPPE</td>
<td>Numeric</td>
<td>Median absolute predictive percentage error</td>
</tr>
<tr>
<td>MDASPE</td>
<td>Numeric</td>
<td>Median absolute symmetric percentage error</td>
</tr>
<tr>
<td>MDRAE</td>
<td>Numeric</td>
<td>Median relative absolute error</td>
</tr>
<tr>
<td>MPPE</td>
<td>Numeric</td>
<td>Mean predictive percentage error</td>
</tr>
<tr>
<td>ME</td>
<td>Numeric</td>
<td>Mean error</td>
</tr>
<tr>
<td>MINAPES</td>
<td>Numeric</td>
<td>Minimum absolute error percentage of standard deviation</td>
</tr>
<tr>
<td>MINERR</td>
<td>Numeric</td>
<td>Minimum error</td>
</tr>
<tr>
<td>MINPE</td>
<td>Numeric</td>
<td>Minimum percentage error</td>
</tr>
<tr>
<td>MINPPE</td>
<td>Numeric</td>
<td>Minimum predictive percentage error</td>
</tr>
<tr>
<td>MINRE</td>
<td>Numeric</td>
<td>Minimum relative error</td>
</tr>
<tr>
<td>MINSPE</td>
<td>Numeric</td>
<td>Minimum symmetric percentage error</td>
</tr>
<tr>
<td>MPE</td>
<td>Numeric</td>
<td>Mean percentage error</td>
</tr>
<tr>
<td>MRAE</td>
<td>Numeric</td>
<td>Mean relative absolute error</td>
</tr>
<tr>
<td>MRE</td>
<td>Numeric</td>
<td>Mean relative error</td>
</tr>
<tr>
<td>MSE</td>
<td>Numeric</td>
<td>Mean square error</td>
</tr>
<tr>
<td>MSPE</td>
<td>Numeric</td>
<td>Mean symmetric percentage error</td>
</tr>
</tbody>
</table>
Table 3.35  continued

<table>
<thead>
<tr>
<th>Column</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>N</td>
<td>Numeric</td>
<td>Number of observations that were used</td>
</tr>
<tr>
<td>NMISSA</td>
<td>Numeric</td>
<td>Number of missing actual values</td>
</tr>
<tr>
<td>NMISSP</td>
<td>Numeric</td>
<td>Number of missing predicted values</td>
</tr>
<tr>
<td>NOBS</td>
<td>Numeric</td>
<td>Number of observations</td>
</tr>
<tr>
<td>NPARMS</td>
<td>Numeric</td>
<td>Number of parameters</td>
</tr>
<tr>
<td>RMSE</td>
<td>Numeric</td>
<td>Root mean square error</td>
</tr>
<tr>
<td>RSQUARE</td>
<td>Numeric</td>
<td>R-square</td>
</tr>
<tr>
<td>RWRSQ</td>
<td>Numeric</td>
<td>Random walk R-square</td>
</tr>
<tr>
<td>SBC</td>
<td>Numeric</td>
<td>Schwarz Bayesian information criterion</td>
</tr>
<tr>
<td>SMAPE</td>
<td>Numeric</td>
<td>Symmetric mean absolute percentage error</td>
</tr>
<tr>
<td>SSE</td>
<td>Numeric</td>
<td>Sum of square error</td>
</tr>
<tr>
<td>SST</td>
<td>Numeric</td>
<td>Corrected total sum of squares</td>
</tr>
<tr>
<td>TSS</td>
<td>Numeric</td>
<td>Total sum of squares</td>
</tr>
<tr>
<td>UMSE</td>
<td>Numeric</td>
<td>Unbiased mean square error</td>
</tr>
<tr>
<td>URMSE</td>
<td>Numeric</td>
<td>Unbiased root mean square error</td>
</tr>
</tbody>
</table>

Table 3.36 summarizes the methods that are associated with the OUTSELECT Object.

Table 3.36  Methods of the OUTSELECT Object

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Collect</td>
<td>Collect the model selection fit statistics from the FORENG instance FORENGObj</td>
</tr>
<tr>
<td>nrows</td>
<td>Get the current row count from the OUTSELECT instance</td>
</tr>
</tbody>
</table>
OUTSELECT Synopsis

DECLARE OBJECT obj (OUTSELECT) ;

Method syntax, in order of typical usage:

\( rc = \text{obj}.\text{Collect}(\text{FORENGObj}) ; \)
\( rc = \text{obj}.\text{nrows}() ; \)

OUTSELECT Methods

OUTSELECT.Collect Method

\( rc = \text{obj}.\text{Collect}(\text{FORENGObj}) ; \)
Collects the model selection fit statistics from the FORENG instance \text{FORENGObj}.

Input Arguments
You must specify the following input argument:

\text{FORENGObj} specifies the FORENG object instance to use as the source of time series model selection fit statistics.

OUTSELECT.nrows Method

\( rc = \text{obj}.\text{nrows}() ; \)
Gets the current row count from the OUTSELECT instance.

Arguments
There are no arguments associated with this method.
OUTSTAT Object

The OUTSTAT object collects from a FORENG instance the statistics of fit for the selected model. This information is useful for evaluating how well the selected model fits the dependent series. The CAS table schema that is used for storing the fit statistics is compatible with the schema that is used by the HPFENGINE procedure for its OUTSTAT= data set.

Table 3.37 shows the contents of the OUTSTAT object.

<table>
<thead>
<tr>
<th>Column</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>MODEL</em></td>
<td>String</td>
<td>Model specification name</td>
</tr>
<tr>
<td><em>NAME</em></td>
<td>String</td>
<td>Variable name</td>
</tr>
<tr>
<td><em>REGION</em></td>
<td>String</td>
<td>Region in which the statistics are calculated. Values in the <em>REGION</em> variable include:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>FIT indicates that fit statistics were calculated over the fit region.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>FORECAST indicates that fit statistics were calculated over the forecast region.</td>
</tr>
<tr>
<td><em>SELECT</em></td>
<td>String</td>
<td>Name of model selection list to which <em>MODEL</em> belongs</td>
</tr>
<tr>
<td>AADJRsq</td>
<td>Numeric</td>
<td>Amemiya’s adjusted R-square</td>
</tr>
<tr>
<td>ADJRsq</td>
<td>Numeric</td>
<td>Adjusted R-square</td>
</tr>
<tr>
<td>AIC</td>
<td>Numeric</td>
<td>Akaike’s information criterion</td>
</tr>
<tr>
<td>AICC</td>
<td>Numeric</td>
<td>Finite sample corrected AIC</td>
</tr>
<tr>
<td>APC</td>
<td>Numeric</td>
<td>Amemiya’s prediction criterion</td>
</tr>
<tr>
<td>DFE</td>
<td>Numeric</td>
<td>Degrees of freedom error</td>
</tr>
<tr>
<td>GMAPE</td>
<td>Numeric</td>
<td>Geometric mean absolute percentage error</td>
</tr>
<tr>
<td>GMAPES</td>
<td>Numeric</td>
<td>Geometric mean absolute error percentage of standard deviation</td>
</tr>
<tr>
<td>GMAPPE</td>
<td>Numeric</td>
<td>Geometric mean absolute predictive percentage error</td>
</tr>
<tr>
<td>GMRAE</td>
<td>Numeric</td>
<td>Geometric mean relative absolute error</td>
</tr>
<tr>
<td>GMASPE</td>
<td>Numeric</td>
<td>Geometric mean absolute symmetric percentage error</td>
</tr>
<tr>
<td>MAE</td>
<td>Numeric</td>
<td>Mean absolute error</td>
</tr>
<tr>
<td>MAPE</td>
<td>Numeric</td>
<td>Mean absolute percentage error</td>
</tr>
<tr>
<td>MAPES</td>
<td>Numeric</td>
<td>Mean absolute error percentage of standard deviation</td>
</tr>
<tr>
<td>Column</td>
<td>Type</td>
<td>Description</td>
</tr>
<tr>
<td>-----------</td>
<td>-------</td>
<td>------------------------------------</td>
</tr>
<tr>
<td>MAPPE</td>
<td>Numeric</td>
<td>Symmetric mean absolute predictive percentage error</td>
</tr>
<tr>
<td>MASE</td>
<td>Numeric</td>
<td>Mean absolute scaled error</td>
</tr>
<tr>
<td>MAXAPES</td>
<td>Numeric</td>
<td>Maximum absolute error percentage of standard deviation</td>
</tr>
<tr>
<td>MAXERR</td>
<td>Numeric</td>
<td>Maximum error</td>
</tr>
<tr>
<td>MAXPE</td>
<td>Numeric</td>
<td>Maximum percentage error</td>
</tr>
<tr>
<td>MAXPPE</td>
<td>Numeric</td>
<td>Maximum predictive percentage error</td>
</tr>
<tr>
<td>MAXRE</td>
<td>Numeric</td>
<td>Maximum relative error</td>
</tr>
<tr>
<td>MAXSPE</td>
<td>Numeric</td>
<td>Maximum symmetric percentage error</td>
</tr>
<tr>
<td>MDAPE</td>
<td>Numeric</td>
<td>Median absolute percentage error</td>
</tr>
<tr>
<td>MDAPES</td>
<td>Numeric</td>
<td>Median absolute error percentage of standard deviation</td>
</tr>
<tr>
<td>MDAPPE</td>
<td>Numeric</td>
<td>Median absolute predictive percentage error</td>
</tr>
<tr>
<td>MDASPE</td>
<td>Numeric</td>
<td>Median absolute symmetric percentage error</td>
</tr>
<tr>
<td>MDRAE</td>
<td>Numeric</td>
<td>Median relative absolute error</td>
</tr>
<tr>
<td>MPPE</td>
<td>Numeric</td>
<td>Mean predictive percentage error</td>
</tr>
<tr>
<td>ME</td>
<td>Numeric</td>
<td>Mean error</td>
</tr>
<tr>
<td>MINAPES</td>
<td>Numeric</td>
<td>Minimum absolute error percentage of standard deviation</td>
</tr>
<tr>
<td>MINERR</td>
<td>Numeric</td>
<td>Minimum error</td>
</tr>
<tr>
<td>MINPE</td>
<td>Numeric</td>
<td>Minimum percentage error</td>
</tr>
<tr>
<td>MINPPE</td>
<td>Numeric</td>
<td>Minimum predictive percentage error</td>
</tr>
<tr>
<td>MINRE</td>
<td>Numeric</td>
<td>Minimum relative error</td>
</tr>
<tr>
<td>MINSPE</td>
<td>Numeric</td>
<td>Minimum symmetric percentage error</td>
</tr>
<tr>
<td>MPE</td>
<td>Numeric</td>
<td>Mean percentage error</td>
</tr>
<tr>
<td>MRAE</td>
<td>Numeric</td>
<td>Mean relative absolute error</td>
</tr>
<tr>
<td>MRE</td>
<td>Numeric</td>
<td>Mean relative error</td>
</tr>
<tr>
<td>MSE</td>
<td>Numeric</td>
<td>Mean square error</td>
</tr>
<tr>
<td>MSPE</td>
<td>Numeric</td>
<td>Mean symmetric percentage error</td>
</tr>
<tr>
<td>N</td>
<td>Numeric</td>
<td>Number of observations that were used</td>
</tr>
<tr>
<td>NMISSA</td>
<td>Numeric</td>
<td>Number of missing actual values</td>
</tr>
<tr>
<td>NMISSP</td>
<td>Numeric</td>
<td>Number of missing predicted values</td>
</tr>
</tbody>
</table>
**Table 3.37 continued**

<table>
<thead>
<tr>
<th>Column</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NOBS</td>
<td>Numeric</td>
<td>Number of observations</td>
</tr>
<tr>
<td>NPARMS</td>
<td>Numeric</td>
<td>Number of parameters</td>
</tr>
<tr>
<td>RMSE</td>
<td>Numeric</td>
<td>Root mean square error</td>
</tr>
<tr>
<td>RSQUARE</td>
<td>Numeric</td>
<td>R-square</td>
</tr>
<tr>
<td>RWRSQ</td>
<td>Numeric</td>
<td>Random walk R-square</td>
</tr>
<tr>
<td>SBC</td>
<td>Numeric</td>
<td>Schwarz Bayesian information criterion</td>
</tr>
<tr>
<td>SMAPE</td>
<td>Numeric</td>
<td>Symmetric mean absolute percentage error</td>
</tr>
<tr>
<td>SSE</td>
<td>Numeric</td>
<td>Sum of square error</td>
</tr>
<tr>
<td>SST</td>
<td>Numeric</td>
<td>Corrected total sum of squares</td>
</tr>
<tr>
<td>TSS</td>
<td>Numeric</td>
<td>Total sum of squares</td>
</tr>
<tr>
<td>UMSE</td>
<td>Numeric</td>
<td>Unbiased mean square error</td>
</tr>
<tr>
<td>URMSE</td>
<td>Numeric</td>
<td>Unbiased root mean square error</td>
</tr>
</tbody>
</table>

Table 3.38 summarizes the methods that are associated with the OUTSTAT Object.

**Table 3.38 Methods of the OUTSTAT Object**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Collect</td>
<td>Collect the statistics of fit from the FORENG instance FORENGObj</td>
</tr>
<tr>
<td>nrows</td>
<td>Get the current row count from the OUTSTAT instance</td>
</tr>
</tbody>
</table>

**OUTSTAT Synopsis**

```
DECLARE OBJECT obj (OUTSTAT) ;
```

Method syntax, in order of typical usage:
```
rc=obj.Collect (FORENGObj ) ;
rc=obj.nrows () ;
```
OUTSTAT Methods

OUTSTAT.Collect Method

\[ rc=obj.Collect \left( FORENGObj \right); \]

Collects the statistics of fit from the FORENG instance \( FORENGObj \).

**Input Arguments**

You must specify the following input argument:

\( FORENGObj \) specifies the FORENG object instance to use as the source of time series statistics of fit.

OUTSTAT.nrows Method

\[ rc=obj.nrows \left( \right); \]

Gets the current row count from the OUTSTAT instance.

**Arguments**

There are no arguments associated with this method.

OUTFMSG Object

The OUTFMSG collector object stores the forecast model selection graph (FMSG) XML in a CAS table. Table 3.39 shows the contents of the OUTFMSG object.

<table>
<thead>
<tr>
<th>Column</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>FMSGPEC</em></td>
<td>String</td>
<td>FMSG specification XML document</td>
</tr>
<tr>
<td><em>NAME</em></td>
<td>String</td>
<td>Name of the dependent variable (can be missing)</td>
</tr>
<tr>
<td><em>SPECLEN</em></td>
<td>Numeric</td>
<td>Length of the XML specification for FMSG</td>
</tr>
<tr>
<td><em>SPECNAME</em></td>
<td>String</td>
<td>Name of the FMSG specification</td>
</tr>
</tbody>
</table>

Table 3.40 summarizes the methods that are associated with the OUTFMSG Object.
Table 3.40  Methods of the OUTFMSG Object

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Collect</td>
<td>Collect forecast model selection graph for output to a CAS table</td>
</tr>
<tr>
<td>nrows</td>
<td>Query the OUTFMSG object for its current row count</td>
</tr>
</tbody>
</table>

OUTFMSG Synopsis

DECLARE OBJECT obj (OUTFMSG ) ;

Method syntax, in order of typical usage:

\[
rc=obj.\text{Collect} (SourceObject ) ;
\]

\[
rc=obj.\text{nrows} () ;
\]

OUTFMSG Methods

OUTFMSG.Collect Method

\[
rc=obj.\text{Collect} (SourceObject ) ;
\]

Collects forecast model selection graph for output to a CAS table.

Input Arguments

You must specify the following input argument:

SourceObject specifies the instance to be used as the source of the diagnostic control option specification to be stored. You can specify the following values for SourceObject:

- DIAGNOSE uses the FMSG XML that is generated from DIAGNOSE object model XML.
- FORENG uses the FMSG XML that is generated from the FORENG FMSG that is used to forecast.
- SELSPEC uses the FMSG XML that is generated from specification object content.

OUTFMSG.nrows Method

\[
rc=obj.\text{nrows} () ;
\]

Queries OUTFMSG object for its current row count.

Arguments

There are no arguments associated with this method.
OUTSCORE Object

The OUTSCORE collector object stores forecast model score XML to a CAS table.

Table 3.41 shows the contents of the OUTSCORE object.

<table>
<thead>
<tr>
<th>Column</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>NAME</em></td>
<td>String</td>
<td>Name of the dependent variable (might be missing)</td>
</tr>
<tr>
<td><em>SCORESPEC</em></td>
<td>String</td>
<td>Score specification XML document</td>
</tr>
<tr>
<td><em>SPECLEN</em></td>
<td>Numeric</td>
<td>Length of the score specification XML</td>
</tr>
<tr>
<td><em>SPECNAME</em></td>
<td>String</td>
<td>Name of the score specification</td>
</tr>
</tbody>
</table>

Table 3.42 summarizes the methods that are associated with the OUTSCORE object.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Collect</td>
<td>Collect forecast model score XML and store it in a CAS table</td>
</tr>
<tr>
<td>nrows</td>
<td>Query the OUTSCORE object for its current row count</td>
</tr>
</tbody>
</table>

OUTSCORE Synopsis

DECLARE OBJECT obj (OUTSCORE ) ;

Method syntax, in order of typical usage:

```
rc=obj.Collect (SourceObject ) ;
rc=obj.nrows () ;
```
OUTSCORE Methods

OUTSCORE.Collect Method

\[ rc = \text{obj}\!.\text{Collect} (\text{SourceObject}) \];

Generates score XML to be stored in a CAS table.

**Input Arguments**
You must specify the following input argument:

**SourceObject** specifies the instance to be used as the source of the score XML to be stored. You can specify the following value:

FORENG scores XML that is generated from FORENG object context.

OUTSCORE.nrows Method

\[ rc = \text{obj}\!.\text{nrows} () \];

Queries the OUTSCORE object for its current row count. A returned missing value indicates that the Collect method has not successfully completed.

**Arguments**
There are no arguments associated with this method.

INDIAG Object

The INDIAG repeater object replays (via the OUTDIAG collector object) diagnostic control specifications that have been stored in a CAS table for use in a DIAGNOSE object.

Table 3.43 summarizes the methods that are associated with the INDIAG Object.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>nrows</td>
<td>Query INDIAG object for its current row count</td>
</tr>
</tbody>
</table>
INDIAG Synopsis

DECLARE OBJECT obj (INDIAG) ;

Method syntax:

rc=obj.nrows () ;

INDIAG Methods

INDIAG.nrows Method

rc=obj.nrows () ;

Queries the INDIAG object for its current row count. A returned missing value indicates that the INDIAG object has not been successfully configured.

Arguments
There are no arguments associated with this method.

INFMSG Object

The INFMSG repeater object replays (via the OUTFMSG collector object) forecast model selection graph (FMSG) XML specifications from a CAS table for use in a FORENG instance.

Table 3.44 summarizes the methods that are associated with the INFMSG Object.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>nrows</td>
<td>Return the number of rows in the INFMSG object. A returned missing value indicates that the INFMSG object has not been successfully configured.</td>
</tr>
</tbody>
</table>

Table 3.44 Methods of the INFMSG Object
INFMSG Synopsis

DECLARE OBJECT obj (INFMSG ) ;

Method syntax:

    rc=obj.nrows () ;

INFMSG Methods

INFMSG.nrows Method

    rc=obj.nrows () ;

Returns the number of rows in the INFMSG object. A returned missing value indicates that the INFMSG object has not been successfully configured.

Arguments

There are no arguments associated with this method.

INesti Object

The INEST repeater object replays (via the OUTEST collector object) model parameter estimates from a CAS table for use in a FORENG instance.

Table 3.45 summarizes the methods that are associated with the INEST object.

Table 3.45  Methods of the INEST Object

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>nrows</td>
<td>Return the number of rows in the INEST object. A returned missing value indicates that the INEST object has not been successfully configured.</td>
</tr>
</tbody>
</table>
INEST Synopsis

```plaintext
DECLARE OBJECT obj (INEST) ;
```

Method syntax:

```plaintext
rc = obj.nrows () ;
```

INEST Methods

INEST.nrows Method

```plaintext
rc = obj.nrows () ;
```

Returns the number of rows in the INEST object. A returned missing value indicates that the INEST object has not been successfully configured.

**Arguments**

There are no arguments associated with this method.

INEVENT Object

The INEVENT repeater object replays event definitions from a CAS table for use in a TSDF instance.

Table 3.46 summarizes the methods that are associated with the INEVENT object.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>nrows</td>
<td>Return the number of rows in the INEVENT object. A returned missing value indicates that the INEVENT object has not been successfully configured.</td>
</tr>
</tbody>
</table>
**INEVENT Synopsis**

```plaintext
DECLARE OBJECT obj (INEVENT < ('VERSION','source') > ) ;
```

You can specify one of the following values for `source`:

- **'HPFEVENTS'** specifies the CAS table schema that is compatible with the HPFEVENTS procedure. The table is expected to contain event variable definitions that are compatible with the schema that is used by the HPFEVENTS procedure for the data sets that are specified in the IN= and OUT= options in the EVENTDATA statement. The variable `_OBSINTRVL_` must contain a numeric value for the multiplier for the observation do-list interval, if any. For example, a value of 5 would represent 'OBS5'. The variable `_LOCALE_` is not required and is ignored if present.

- **'TSMODEL'** specifies the CAS table schema that is compatible with the TSMODEL procedure. The CAS table is expected to contain event variable definitions that are compatible with the schema that is used by the TSMODEL procedure. The schema used by the TSMODEL procedure is similar to the schema that is used by the HPFEVENTS procedure, with a few modifications. The variable `_OBSINTRVL_` must contain a string (such as 'OBS5') for the observation do-list interval, if any. The variable `_LOCALE_` is required and is input into the event definition.

The default is 'HPFEVENTS'. The current default is designed to facilitate backwards compatibility and a change to the newer TSMODEL version. Thus, the default for OUTEVENT object is 'VERSION','TSMODEL'. The defaults might change in future releases.

Method syntax:

```plaintext
rc=obj.nrows () ;
```

**INEVENT Methods**

**INEVENT.nrows Method**

```plaintext
rc=obj.nrows () ;
```

Returns the number of rows in the INEVENT object. A returned missing value indicates that the INEVENT object has not been successfully configured.

**Arguments**

There are no arguments associated with this method.
**INEVENTBY Object**

The INEVENTBY repeater object replays event maps from a CAS table for use in a TSDF instance for each BY group specified in the CAS table. Replaying the repeater object is equivalent to specifying the AddEvent method for each BY group and event as specified in the CAS table.

Table 3.47 summarizes the methods that are associated with the INEVENT object.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>nrows</td>
<td>Return the number of rows in the INEVENTBY object. A returned missing value indicates that the INEVENTBY object has not been successfully configured.</td>
</tr>
</tbody>
</table>

**INEVENTBY Synopsis**

```plaintext
DECLARE OBJECT obj (INEVENTBY);
```

Method syntax:

```plaintext
rc = obj.nrows();
```

**INEVENTBY Methods**

**INEVENTBY.nrows Method**

```plaintext
rc = obj.nrows();
```

Returns the number of rows in the INEVENTBY object. A returned missing value indicates that the INEVENTBY object has not been successfully configured.

**Arguments**

There are no arguments associated with this method.
INSCORE Object

The INSCORE repeater object replays (via the OUTSCORE collector object) forecast model score specifications from a CAS table for use in a SCORE instance.

Table 3.48 summarizes the methods that are associated with the INSCORE object.

Table 3.48 Methods of the INSCORE Object

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>nrows</td>
<td>Return the number of rows in the INSCORE object. A returned missing value indicates that the INSCORE object has not been successfully configured.</td>
</tr>
</tbody>
</table>

INSCORE Synopsis

DECLARE OBJECT obj (INSCORE) ;

Method syntax:

rc=obj.nrows () ;

INSCORE Methods

INSCORE.nrows Method

rc=obj.nrows () ;

Returns the number of rows in the INSCORE object. A returned missing value indicates that the INSCORE object has not been successfully configured.

Arguments

There are no arguments associated with this method.
Examples: ATSM Package

Throughout this section, it is assumed that you have already started a CAS session and that the data tables that are used in this section are in mycas, a CAS library that you have necessary permissions to work with. This section assumes that you are familiar with the general workings of the TSMODEL procedure; for more information, see Chapter 11, “The TSMODEL Procedure” (SAS Visual Forecasting: Forecasting Procedures).

Using CAS Sessions and CAS Engine Librefs

SAS Cloud Analytic Services (CAS) is the analytic server and associated cloud services in SAS Viya. This section describes how to create a CAS session and set up a CAS engine libref that you can use to connect to the CAS session. It assumes that you have a CAS server already available; contact your system administrator if you need help starting and terminating a server. This CAS server is identified by specifying the host on which it runs and the port on which it listens for communications. To simplify your interactions with this CAS server, the host information and port information for the server are stored as SAS option values that are retrieved automatically whenever this CAS server needs to be accessed. You can examine the host and port values for the server at your site by using the following statements:

```sas
proc options option=(CASHOST CASPORT);
run;
```

In addition to starting a CAS server, your system administrator might also have created a CAS session and a CAS engine libref for your use. You can define your own sessions and CAS engine librefs that connect to the CAS server as shown in the following statements:

```sas
cas mysess;
libname mycas cas sessref=mysess;
```

The CAS statement creates the CAS session named mysess, and the LIBNAME statement creates the mycas CAS engine libref that you use to connect to this session. It is not necessary to explicitly name the CASHOST and CASPORT of the CAS server in the CAS statement, because these values are retrieved from the corresponding SAS option values.

If you have created the mysess session, you can terminate it by using the TERMINATE option in the CAS statement as follows:

```sas
cas mysess terminate;
```

For more information about the CAS statement and the LIBNAME statement, see SAS Cloud Analytic Services: User’s Guide. For general information about CAS and CAS sessions, see SAS Cloud Analytic Services: Fundamentals.
Example 3.1: Automatic Modeling and Forecasting of the Airline Series

The airline passenger data, given as Series G in Box and Jenkins (1976), have been used in time series analysis literature as an example of a nonstationary seasonal time series. For more information about ARIMA modeling of Series G, see “Example 7.2 Seasonal Model for the Airline Series” in the ARIMA procedure in *SAS/ETS User’s Guide*.

This example shows how you can use the objects in the ATSM package to automatically model the airline series. There is no one best way to do this. You can customize your modeling choices by appropriately configuring the DIAGSPEC and DIAGNOSE objects. After that, you can use a FORENG object to produce the analysis results based on the diagnostic choices that you have made for the DIAGNOSE object. The main steps in the program are as follows:

1. The PROC TSMODEL statement specifies the input data set (mycas.air) and a variety of output tables (mycas.airFor, mycas.airEst, and so on).
2. The ID statement specifies date as the time index variable, and the INTERVAL= option indicates that the data are monthly.
3. The VAR statement specifies the input data set variable, air, which contains the airline series.
4. The REQUIRE statement specifies the ATSM package, which is needed for the analysis.
5. The statements between the SUBMIT and ENDSUBMIT statements use the ATSM package objects to perform the actual analysis in your CAS session. These statements are grouped into four parts:
   - The first part creates a data frame, airData, which is a TSDF object that contains the necessary analysis variables. The variable roles—for example, whether target or input—and the default season length that is associated with the data frame are also assigned in this step.
   - In the second part, the model identification process is specified. This is done in two steps. First a DIAGSPEC object, airDiagSpec, is configured. It is then used to initialize a DIAGNOSE object, airDiag. The DIAGSPEC object in this example uses the default settings, which amounts to selecting the best fitting model from two model families: exponential smoothing models (ESMs) and ARIMAX models. As a result of setting the HOLDOUT parameter for the DIAGNOSE object to 12, the best fitting model is chosen within each family on the basis of the RMSE criterion (the default CRITERION choice) in the holdout region (the last 12 observations).
   - In the third part, a FORENG object is used to do the final model selection based on the DIAGNOSE object results and to produce forecasts. This FORENG object, airEng, is initialized by using the airDiag object that is created in the second part.
   - In the last part, various output tables are created by using collector objects of appropriate type.

```sas
data mycas.air;
  set sashelp.air;
run;
```
```plaintext
proc tsm model data=mycas.air
    outobj=(airFor=mycas.airFor airEst=mycas.airEst
              modInfo=mycas.modInfo airSelect=mycas.airSelect);
    id date interval=month;
    var air;
    require atsm;
    submit;

    declare object airData(tsdf);
    rc = airData.Initialize();
    rc = airData.AddY(air);
    rc = airData.SetOption('seasonality', 12);

    declare object airDiagSpec(diagspec);
    rc = airDiagSpec.Open();
    rc = airDiagSpec.SetESM();
    rc = airDiagSpec.SetARIMAX();
    rc = airDiagSpec.Close();

    declare object airDiag(diagnose);
    rc = airDiag.Initialize(airData);
    rc = airDiag.SetSpec(airDiagSpec);
    rc = airDiag.SetOption('holdout', 12);
    rc = airDiag.Run();

    declare object airEng(foreng);
    rc = airEng.Initialize(airDiag);
    rc = airEng.SetOption('lead', 12);
    rc = airEng.Run();

    declare object modInfo(outmodelinfo);
    rc = modInfo.Collect(airEng); 

    declare object airFor(outfor);
    rc = airFor.Collect(airEng);

    declare object airEst(outest);
    rc = airEst.Collect(airEng);

    declare object airSelect(outselect);
    rc = airSelect.Collect(airEng);

    endsubmit;
run;
```

Output 3.1.1 shows the results of final model selection step, Output 3.1.2 shows the parameter estimates of the selected model, and Output 3.1.3 shows the forecasts according to the selected model.
Example 3.2: Using an Event Object to Add Predefined Events to a Model

The following code is an example of using a predefined event in an ARIMAX model.

```sas
data mycas.air;
  set sashelp.air;
run;

proc tsmodel data=mycas.air
  outobj=(airFor=mycas.airForEx02 airEst=mycas.airEstEx02
        modInfo=mycas.modInfoEx02 airSelect=mycas.airSelect);
  id date interval=month;
  var air;
```
require atsm;
submit;

declare object ev1(event);
rc = ev1.Initialize();

declare object airData(tsdf);
rc = airData.Initialize();
rc = airData.AddY(air);
rc = airData.SetOption('seasonality', 12);
rc = airData.AddEvent(ev1, 'Easter');

declare object airDiagSpec(diagspec);
rc = airDiagSpec.Open();
rc = airDiagSpec.SetARIMAX();
rc = airDiagSpec.Close();

declare object airDiag(diagnose);
rc = airDiag.Initialize(airData);
rc = airDiag.SetSpec(airDiagSpec);
rc = airDiag.Run();

declare object airEng(foreng);
rc = airEng.Initialize(airDiag);
rc = airEng.SetOption('lead', 12);
rc = airEng.Run();

declare object modInfo(outmodelinfo);
rc = modInfo.Collect(airEng);

declare object airFor(outfor);
rc = airFor.Collect(airEng);

declare object airEst(outest);
rc = airEst.Collect(airEng);

declare object airSelect(outselect);
rc = airSelect.Collect(airEng);
endsubmit;
run;

Output 3.2.1 shows the general model description, an ARIMA model with 1 event and no outliers. Output 3.2.2 shows the parameter estimates of the selected model, and Output 3.2.3 shows the forecasts according to the selected model.

**Output 3.2.1** General Model Description

<table>
<thead>
<tr>
<th><em>MODEL</em></th>
<th><em>MODELTYPE</em></th>
<th><em>DEPTRANS</em></th>
<th><em>SEASONAL</em></th>
<th><em>TREND</em></th>
<th><em>EVENTS</em></th>
<th><em>OUTLIERS</em></th>
</tr>
</thead>
<tbody>
<tr>
<td>DIAG1_ARIMAX1</td>
<td>ARIMA</td>
<td>NONE</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
**Example 3.3: Performing Automatic Time Series Imputation**

Time series imputation is the process of replacing missing values in a time series with reasonable values that reflect the existing pattern in the available data (such as trend, seasonal variations, or long-term cyclical variations). It is a popular technique that is used across various domains of science. This example shows how you can do the following:

- Use a TSDF object to impute a time series that contains missing values.
- Retrieve the imputed values of the time series and store them in a CAS table.

The TSDF object uses the best candidate exponential smoothing model (ESMBEST) to impute a time series. For the purposes of this example, the following statements introduce two artificial missing values into the airline series (see Example 3.1 for more information about the airline series and the airline model). The modified series is stored in the mycas.airMiss CAS table. These statements assume that your CAS engine libref is named mycas, but you can substitute any appropriately defined CAS engine libref.

```plaintext
data mycas.airmiss;
  set sashelp.air;
  airmiss = air;
  if date = '01JUL1955'd then airmiss = .;
```

**Output 3.2.2** Parameter Estimates of the Selected Model (Partial Output)

**Parameter Estimates for the ARIMA (1 1 0)(0 1 0) Model with Easter**

<table>
<thead>
<tr>
<th>COMPONENT</th>
<th>EST</th>
<th>STDERR</th>
<th>TVALUE</th>
<th>PVALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>AR</td>
<td>-0.1890</td>
<td>0.0872</td>
<td>-2.1684</td>
<td>0.0320</td>
</tr>
<tr>
<td>SCALE</td>
<td>9.2536</td>
<td>2.6394</td>
<td>3.5060</td>
<td>0.000626</td>
</tr>
</tbody>
</table>

**Output 3.2.3** Forecasts Based on the Selected Model (Partial Output)

**Forecasts Based On the ARIMA (1 1 0)(0 1 0) Model with Easter**

<table>
<thead>
<tr>
<th>DATE</th>
<th>PREDICT</th>
<th>STD</th>
<th>UPPER</th>
<th>LOWER</th>
</tr>
</thead>
<tbody>
<tr>
<td>01JAN61</td>
<td>444.2</td>
<td>11.3538</td>
<td>466.4</td>
<td>421.9</td>
</tr>
<tr>
<td>01FEB61</td>
<td>418.2</td>
<td>14.6182</td>
<td>446.8</td>
<td>389.5</td>
</tr>
<tr>
<td>01MAR61</td>
<td>446.2</td>
<td>17.4960</td>
<td>480.5</td>
<td>411.9</td>
</tr>
<tr>
<td>01APR61</td>
<td>488.2</td>
<td>19.9263</td>
<td>527.2</td>
<td>449.1</td>
</tr>
<tr>
<td>01MAY61</td>
<td>499.2</td>
<td>22.0971</td>
<td>542.5</td>
<td>455.8</td>
</tr>
<tr>
<td>01JUN61</td>
<td>562.2</td>
<td>24.0719</td>
<td>609.3</td>
<td>515.0</td>
</tr>
<tr>
<td>01JUL61</td>
<td>649.2</td>
<td>25.8967</td>
<td>699.9</td>
<td>598.4</td>
</tr>
<tr>
<td>01AUG61</td>
<td>633.2</td>
<td>27.6011</td>
<td>687.3</td>
<td>579.1</td>
</tr>
<tr>
<td>01SEP61</td>
<td>535.2</td>
<td>29.2062</td>
<td>592.4</td>
<td>477.9</td>
</tr>
<tr>
<td>01OCT61</td>
<td>488.2</td>
<td>30.7276</td>
<td>548.4</td>
<td>427.9</td>
</tr>
<tr>
<td>01NOV61</td>
<td>417.2</td>
<td>32.1771</td>
<td>480.2</td>
<td>354.1</td>
</tr>
<tr>
<td>01DEC61</td>
<td>459.2</td>
<td>33.5641</td>
<td>524.9</td>
<td>393.4</td>
</tr>
</tbody>
</table>
The following statements plot the modified airline series along with the two artificial missing values that were introduced. The results are shown in Output 3.3.1.

```plaintext
proc sort data=mycas.airmiss out=airmiss;
  by date;
run;

proc sgplot data=airmiss;
  series x=date y = airmiss / break lineattrs=(color=blue thickness=3);
  series x=date y = air / lineattrs=(thickness=2 pattern=dot color=blue);
run;
```

In Output 3.3.1, the solid blue line represents all the nonmissing data values in the modified airline series. The dotted blue line between the years 1955 and 1956 represents the two actual values that were set to missing values.

**Output 3.3.1**  
Airline Passenger Time Series with Artificial Missing Values

The following statements fit the ESMBEST model to the modified airline series and store the imputed time series in the mycas.airimpute CAS table via the OUTARRAY statement:

```plaintext
proc tsmodel data=mycas.airmiss outarray=mycas.airimpute;
  id date interval=month;
  var air airmiss;
  outarray airimpute;
  require atsm;
```
submit;

*** Create a TSDF object to perform imputation ***;
declare object airData(tsdf);

*** Initialize the TSDF object with default options ***;
rc = airData.Initialize();
rc = airData.SetOption('seasonality', 12);

*** Add the airline time series to be imputed ***;
rc = airData.AddX(airmiss);

*** Retrieve the imputed airline time series ***;
rc = airData.GetSeries('airmiss',airimpute,'ADJUST','YES');
endsubmit;
quit;

The following statements plot the modified airline series along with its imputed version. The results are shown in Output 3.3.2.

proc sgplot data=mycas.airimpute;
  series x=date y = airmiss / break lineattrs=(thickness=3 color=blue);
  series x=date y = air / lineattrs=(thickness=2 pattern=dot color=blue);
  series x=date y = airimpute / lineattrs=(thickness=1 color=red);
  where year(date) >= 1954 and year(date) <= 1956;
run;

In Output 3.3.2, the thick blue line represents the modified airline series, and the thin red line represents its imputed version. The time axis range is restricted to the dates between the years 1954 and 1956 in order to facilitate the visualization. Notice how the thin red line (imputed values) closely mimics the dotted blue line (original actual values) over the region between the years 1955 and 1956 where the artificial missing values were introduced.
Output 3.3.2 Imputed Airline Passenger Time Series

References

Chapter 4
External Languages Package

Contents

Overview: EXTLANG Package .................................................. 122
External Languages Access Control Configuration .......................... 122
EXTLANG Package Summary .................................................. 130
Using the EXTLANG Package .................................................. 130
Return Codes ................................................................. 131
PYTHON2, PYTHON3, and R Objects ......................................... 131
Interpreter Object Synopsis .................................................. 134
Interpreter Object Methods .................................................. 134
OUTEXTLOG Object .......................................................... 143
OUTEXTLOG Synopsis ......................................................... 144
OUTEXTLOG Methods ......................................................... 144
OUTEXTVARSTATUS Object .................................................. 145
OUTEXTVARSTATUS Synopsis .............................................. 147
OUTEXTVARSTATUS Methods .............................................. 147
OUTEXTCODE Object ......................................................... 148
OUTEXTCODE Synopsis ....................................................... 148
OUTEXTCODE Methods ....................................................... 149
INEXTCODE Object ........................................................ 150
INEXTCODE Synopsis ......................................................... 150
INEXTCODE Methods ......................................................... 151
Examples: EXTLANG Package ................................................ 151
Using CAS Sessions and CAS Engine Librefs ............................... 151
Example 4.1: Calculate Moving Average of the Airline Series ............. 152
Example 4.2: Ensuring Code Integrity ....................................... 155
Example 4.3: BY-Group-Specific Code ..................................... 158
Common Pitfalls in Using the EXTLANG Package ......................... 162
References ................................................................. 163
Overview: EXTLANG Package

The external languages (EXTLANG) package provides objects that enable seamless integration of external-language programs into SAS environments. The EXTLANG package supports Python (versions 2.6.6–2.7.7 and 3.3 and higher) and R (versions 3.2.5 and higher). The objects in this package enable you to integrate any program that is written in these languages into your SAS environment, and they enable you to specify which variables should be shared between the two environments. In order to handle big data problems, the EXTLANG package is integrated into the SAS Cloud Analytic Services (CAS) system, so that external-language programs run in parallel. Furthermore, you can take advantage of the built-in mechanisms for parallel BY-group processing to parallelize programs that are written in these languages according to BY groups. Mechanisms are built into the EXTLANG package to help ensure compatibility with a wide range of site security policies. Administrators can enable or disable the EXTLANG package itself and can specify whether users can perform the following in their programs:

- specify the clusterwide path for the Python or R interpreter to use
- specify environment variables to be used in the running environment of the Python or R program
- specify the directory in which to store temporary data
- load external language source code either from disk or CAS tables
- execute inline external language source code

External Languages Access Control Configuration

Because the functionality of the EXTLANG package might conflict with your site’s security policies, the package includes mechanisms that enable CAS server administrators to control users’ access to installed external-language interpreters and usable storage areas. For example, administrators can do the following:

- specify whitelisted directories, which are directories that users can write to or that contain source code that users are allowed to load
- control whether users can specify paths to the executables for interpreting and compiling external-language programs
- control whether users can insert external-language source code directly into their SAS programs
- control whether users can specify environment variables for their external-language programs

By taking advantage of the hierarchical nature of XML, you can specify attributes as concisely as possible on a global, per-group, per-language, or per-group-per-language basis. This section describes the configuration file that is used for access control. The path to this XML-formatted file is specified via the CAS environment variable, SAS_EXTLANG_SETTINGS. For information about where to set environment variables for your particular CAS deployment, see the section “Environment Variables” in SAS Viya Administration: SAS Cloud
Analytic Services. By default, the EXTLANG package is disabled, all text attributes are set to an empty string, and all Boolean values are set to 'BLOCK'.

The XML schema of the configuration file is as follows.

You must specify the following tag:

**EXTERNAL**
contains attributes that control the default settings for nested tags. You can specify the following attributes in the EXTLANG tag:

*version*=*level*
specifies the XML document’s version number. This attribute exists to accommodate future changes to the XML definition.

*mode*="ALLOW" | 'ANARCHY' | 'BLOCK'
specifies the access mode for the EXTLANG package. Possible values for this attribute are:

'ALLOW' allows use of the EXTLANG package and initializes all Boolean properties to 'BLOCK'. (The properties can be explicitly overridden.)

'ANARCHY' allows use of the EXTLANG package and initializes all Boolean properties to 'ALLOW'. (The properties can be explicitly overridden.)

'BLOCK' disallows use of the EXTLANG package.

*allowAllUsers*="ALLOW" | 'BLOCK'
controls which users can run external-language programs. Possible values for this attribute are:

'ALLOW' allows all users to use the EXTLANG package. The administrator can still specify GROUP blocks to specify group-specific overrides. For users who are not specified in a GROUP block, the attributes from the DEFAULT tag are applied.

'BLOCK' allows only users specified in a GROUP block to use the EXTLANG package.

The following tags are all optional:

**DEFAULT**
enables you to specify global default settings. Any attributes that are specified in the DEFAULT block override the values that are initialized according to the *mode* attribute that is specified in the EXTLANG start tag. These settings cascade down to all LANGUAGE and GROUP blocks that are nested in the EXTLANG block. You can specify the following attributes in the DEFAULT tag:

*scratchDisk*=*location*
specifies a location on the file system in which to store temporary files that are used to enable support for external languages. This location must have enough space to store shared variables (that is, variables that are transferred between the SAS and external-language environments). Environment variables are expanded. By default, the system-defined default temporary directory is used.
diskWhiteList=paths
specifies one or more file system paths that users can use for the following:

- loading source code from files
- specifying as their scratchDisk
- specifying as options specific to certain objects. For example, the 'TEMPDIR' and 'EXECPATH' options of the PYTHON2, PYTHON3, and R objects of the EXTLANG package

You can specify multiple paths by separating them with the platform-specific path separator, which is “;” on Linux systems. Environment variables are expanded. The escape character is the “\”. A literal “\” can be inserted using “\\”. Attempting to push a file that is not inside a whitelisted directory or its subdirectories will result in an error. The diskWhiteList attribute does not affect the ability to insert external-language source code in other ways. For example, you can still insert code inline. Attempting to specify a path for the scratchDisk attribute that is not inside this directory will result in an error.

userSetScratchDisk='ALLOW' | 'BLOCK'
controls whether users are allowed to specify the path to the location that is specified in the scratchDisk attribute.

'ALLOW' allows users to specify the scratch disk.
'BLOCK' prevents users from specifying the scratch disk. A run time error will occur if a user program attempts to set the scratch disk.

By default, userSetScratchDisk='BLOCK'.

userSetEnv='ALLOW' | 'BLOCK' | 'UNDERSCORE'
controls whether users are allowed to specify environment variables to be passed to the external-language program via the AddEnvVariable method. Acceptable values are:

'ALLOW' allows users to specify environment variables whose names consist of a string of ASCII characters.
'BLOCK' does not allow users to specify environment variables.
'UNDERSCORE' allows users to specify environment variables whose name consists of an underscore followed by a string of ASCII characters.

By default, userSetEnv='BLOCK'.

userSetInterpreter='ALLOW' | 'BLOCK'
controls whether users can specify the external-language executable’s path.

'ALLOW' allows users to specify the path to the external-language interpreter.
'BLOCK' does not allow users to specify the path to the external-language interpreter. An error will occur if a user attempts to specify the path.

By default, userSetInterpreter='BLOCK'.
LANGUAGE

enables you to override default settings on a per-language basis. You can nest one or more LANGUAGE tags directly in the DEFAULT block or within a GROUP block. Any attribute specified in a LANGUAGE tag becomes the default value for that language (that is, those attributes cascade down for all GROUP blocks that do not specify them for that language). If a LANGUAGE tag is nested within a GROUP block, the specified attributes apply only to users within that group.

You must specify the following attribute in the LANGUAGE tag:

\texttt{name='PYTHON2' | 'PYTHON3' | 'R'}

specifies the name of the language being configured.

You can also specify the following optional attributes in the LANGUAGE tag. Unspecified attributes inherit the corresponding attribute value from the DEFAULT block, which inherits default values according to the \texttt{mode} attribute of the EXTLANG start tag:

\texttt{interpreter=\texttt{path}}

specifies the default path to the external language’s executable. This path will be used by all workers in the CAS cluster. Attempting to use objects of the EXTLANG package will fail if the \texttt{interpreter} and \texttt{userSetInterpreter} attributes are not specified.

\texttt{userInlineCode='ALLOW' | 'BLOCK'}

controls whether users can add external-language code from within their SAS programs.

‘ALLOW’ allows users to insert inline code.

‘BLOCK’ does not allow users to insert inline code.

By default, \texttt{userInlineCode='BLOCK'}.

\texttt{userSetEnv='ALLOW' | 'BLOCK' | 'UNDERSCORE'}

controls whether users can specify environment variables. Acceptable values are:

‘ALLOW’ allows users to specify environment variables whose names consist of a string of ASCII characters.

‘BLOCK’ does not allow users to specify environment variables.

‘UNDERSCORE’ allows users to specify environment variables whose name consists of an underscore followed by a string of ASCII characters.

By default, \texttt{userSetEnv='BLOCK'}.

\texttt{userSetInterpreter='ALLOW' | 'BLOCK'}

controls whether users can specify the path to this language’s executable.

‘ALLOW’ allows users to set the interpreter executable.

‘BLOCK’ does not allow users to set the interpreter executable. An error will occur if a user attempts to specify the path to the interpreter.
ENVIRONMENT

Enables you to set environment variables. You can nest one or more ENVIRONMENT tags within a LANGUAGE tag. Environment variables will be set in the external-language interpreter’s running environment. In order for environment variables to be passed, the value of the `userSetEnv` attribute of the enclosing LANGUAGE block must be ‘ALLOW’ or ‘UNDERCORE’. A run-time error will occur if `userSetEnv`='BLOCK' and a user attempts to set an environment variable or if `userSetEnv`='UNDERSCORE' and a user attempts to set an environment variable whose name does not begin with an underscore. Administrators can specify variables that begin with underscore regardless.

You must specify the following attributes in the ENVIRONMENT tag:

- `name=variableName` specifies the environment variable name, which must be a string of ASCII characters
- `value=variableValue` specifies the value of the variable, which must be a string of Unicode characters

GROUP

Specifies group-specific overrides. Every user must be specified in a GROUP block, unless `allowAllUsers='ALLOW'` in the EXTLANG tag. Users cannot belong to multiple groups. Any languages that do not contain a LANGUAGE block that is enclosed by the GROUP block inherit the default language attributes that are defined in the corresponding LANGUAGE block that is enclosed by the DEFAULT block. The default language attributes can themselves be implicitly defined according to the `mode` attribute of the EXTLANG start tag.

You must specify the following attribute in the GROUP tag:

- `name=groupName` is a string that identifies the group. If no `users` attribute is specified in the GROUP tag, the attributes that are defined in this GROUP block are applied to the operating system account whose user name is `groupName`.

You can also specify the following attributes of the GROUP start tag:

- `users=list` specifies a comma-delimited list of user names to which the settings that are defined in this GROUP block will apply. Each user must belong to only one group. The escape character is “\". A literal “\" can be inserted using “\".
- `scratchDisk=directory` specifies the temporary working directory for this group. By default, the location that is specified in the `scratchDisk=` attribute in the DEFAULT block is used.
- `diskWhiteList=paths` specifies a list of whitelisted paths. You can specify multiple paths by using the operating system’s native path separator (which is ‘:’ on Linux). If this attribute is specified, source code that the users of this group push from a file must reside under a path in this list. The location that is specified in the `scratchDisk=` attribute must also reside under a path in this list.
userSetScratchDisk='ALLOW' | 'BLOCK'
specifies whether the users in this group are allowed to specify the scratch disk.

'ALLOW' allows users in this group to set the \texttt{scratchDisk} location.

'BLOCK' does not allow users to set the \texttt{scratchDisk} location. An error will occur if a user attempts to specify the \texttt{scratchDisk} location.

Figure 4.1 shows a sample access control configuration file.
Figure 4.1 Sample Access Control File for the EXTLANG Package

<EXTLANG version="1.0" mode="ALLOW" allowAllUsers="BLOCK">
  <DEFAULT
    scratchDisk="/smalldisk/sas/scratch"
    diskWhitelist="/secure/sas/allowed_scripts:/whitelist"
    userSetScratchDisk="BLOCK"
    userSetEnv="BLOCK"
    userSetInterpreter="BLOCK">
    <LANGUAGE name="PYTHON2"
      userInlineCode="BLOCK"
      interpreter="/some/path/python2"
      userSetEnv="ALLOW"
      userSetInterpreter="BLOCK">
      <ENVIRONMENT name="PYTHONPATH" value="/some/path1:/some/path2"/>
      <ENVIRONMENT name="LDLIBRARYPATH" value="/some/ldpath1:/some/ldpath2"/>
    </LANGUAGE>
    <LANGUAGE name="PYTHON3"
      interpreter="/some/path/python3"
      userSetEnv="BLOCK"
      userSetInterpreter="BLOCK">
      <ENVIRONMENT name="PYTHONPATH" value="/some/path1:/some/path2"/>
      <ENVIRONMENT name="LDLIBRARYPATH" value="/some/ldpath1:/some/ldpath2"/>
    </LANGUAGE>
    <LANGUAGE name="R"
      interpreter="/some/path/Rscript"
      userSetEnv="BLOCK"
      userSetInterpreter="BLOCK">
      <ENVIRONMENT name="LDLIBRARYPATH" value="/some/ldpath1:/some/ldpath2"/>
    </LANGUAGE>
  </DEFAULT>
  <GROUP name="DanDLyons"
    scratchDisk="$HOME/scratch"/>
  <GROUP name="SassySean"
    userSetInterpreter="ALLOW"
    userSetEnv="UNDERSCORE">
    <LANGUAGE name="PYTHON3"
      interpreter="$HOME/anaconda/bin/python3.5"
      <ENVIRONMENT name="_ALGORITHM" value="BEST"/>
    </LANGUAGE>
  </GROUP>
  <GROUP name="sasUsers"
    users="Sam, Ada, Sergey"
    scratchDisk="/pan1"
    diskWhitelist="/home/$USER:/authorized/path"
    userSetScratchDisk="BLOCK"
    userReadDisk="BLOCK">
    <LANGUAGE name="PYTHON2"
      userInlineCode="ALLOW"
      userSetEnv="BLOCK"
      userSetInterpreter="BLOCK"/>
  </GROUP>
</EXTLANG>
The specific configuration that the XML text shown in Figure 4.1 defines is discussed in detail as follows:

- In the EXTLANG tag:
  - Specifying `mode='ALLOW'` enables the EXTLANG package and initializes all Boolean attributes to 'BLOCK'.
  - Specifying `allowAllUsers='BLOCK'` prohibits users whose user name does not appear in a GROUP from using the package.

- In the DEFAULT tag, restrictive settings initialized in the EXTLANG tag are overridden. Five settings are overridden in this block:
  - The `scratchDisk=` attribute is set to `/smalldisk/sas/scratch`. All temporary files that the EXTLANG package creates will go here; it must be large enough to accommodate all data sets that every user must work with.
  - The `diskWhitelist=` attribute specifies that users can push only code files that reside inside the directories `/smalldisk/sas/scratch` and `/whitelist`.
  - The `userSetScratchDisk`, `userSetEnv`, and `userSetInterpreter` attributes are set to 'BLOCK'; this is merely for readability since they were all initialized as such by `mode='ALLOW'` in the enclosing EXTLANG tag.

- In the LANGUAGE tags, default language settings are entered.
  - For PYTHON2, the interpreter path is specified and users are not allowed to change this setting in their program (unless the setting is overridden in a GROUP block). Users are also forbidden from setting environment variables. Default values are then given for 'PYTHONPATH' and 'LDLIBRARYPATH' environment variables.
  - The settings for PYTHON3 are the same as for PYTHON2 except the interpreter executable path.
  - A similar configuration is specified for R, but only the 'LDLIBRARYPATH' environment variable is set.

- GROUP tags are used to override default settings for users.
  - The first GROUP block does not have a `users=` attribute, so the settings it overrides will apply to user *DanDLyons*. All external-language programs run by *DanDLyons* will be stored in subdirectory `scratch` within *DanDLyons*’ home directory.
  - The next GROUP block does not have a `users` attribute either, so the settings it overrides apply only to user *SassySean*. This user is assigned a different default Python 3 interpreter, which `SassySean` can change programmatically because `userSetInterpreter='ALLOW'`. The default restriction on setting environment variables is also relaxed so that *SassySean* can set environment variables as long as they begin with an underscore.
  - The final GROUP tag specifies multiple users to which the specified attributes apply. These users’ temporary data will be stored in `/pan1` and they can push only scripts that are under their home directory. The `userInlineCode=` attribute is overridden to ‘ALLOW’, so these users can add code in their SAS program. The remainder of the settings are inconsequential because they match the defaults.
EXTLANG Package Summary

Table 4.1 summarizes the objects in the EXTLANG package. These objects enable you to load and execute arbitrary Python and R programs, obtain their return codes, check the status of shared variables, save code for later use, and replay the saved code.

<table>
<thead>
<tr>
<th>Object</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Objects for Interacting with External Language Interpreters</strong></td>
<td></td>
</tr>
<tr>
<td>PYTHON2</td>
<td>Interfaces with version 2 of the Python programming language</td>
</tr>
<tr>
<td>PYTHON3</td>
<td>Interfaces with version 3 of the Python programming language</td>
</tr>
<tr>
<td>R</td>
<td>Interfaces with the R programming language</td>
</tr>
<tr>
<td><strong>Objects for Outputting to CAS Tables</strong></td>
<td>Stores user-supplied external-language source code that is supplied via a PYTHON2, PYTHON3, or R object in a CAS table to “replay” it later</td>
</tr>
<tr>
<td>OUTEXTCODE</td>
<td></td>
</tr>
<tr>
<td>OUTEXTLOG</td>
<td>Stores execution and resource usage logs in a CAS table</td>
</tr>
<tr>
<td>OUTEXTVARSTATUS</td>
<td>Stores variables’ status information in a CAS table</td>
</tr>
<tr>
<td><strong>Objects for Inputting from CAS Tables</strong></td>
<td>Reads code from a CAS table and provides it to the external language interpreter for reuse on a per-BY-group basis</td>
</tr>
<tr>
<td>INEXTCODE</td>
<td></td>
</tr>
</tbody>
</table>

Using the EXTLANG Package

The following steps summarize how to use each object in the EXTLANG package. Subsequent sections describe the associated objects and steps in greater detail.

1. Create a PYTHON2, PYTHON3, or R object.
2. Add source code to the object.
3. Specify variables that should be shared between the SAS and external-language environments.
4. Run the external-language program.
5. Check the external-language program and variables’ return codes and status codes.
6. Retrieve the external program’s output and store it in a CAS table by using an OUTEXTLOG object.
7. Store the code in a CAS table by using an OUTEXTCODE object if the code will be run later.
Replay saved code by using an INEXTCODE object.

Return Codes

Table 4.2 shows the return code (rc in method statements) status values that are used in this package. These status code values are returned after a method that is associated with an object is called; they can help determine whether the method executed successfully.

<table>
<thead>
<tr>
<th>Status</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt; 0</td>
<td>An unrecoverable error occurred. No result was produced.</td>
</tr>
<tr>
<td>= 0</td>
<td>Unconditional success. The requested action completed and a normal result was produced.</td>
</tr>
<tr>
<td>&gt; 0</td>
<td>Conditional success or warning. A result was produced subject to conditions.</td>
</tr>
</tbody>
</table>

Upon returning a negative status code, most methods also write to the output log a message that explains the causes of the related failure. These messages provide useful information during the process of debugging a user program. In the TSMODEL procedure, the output log is stored in the CAS table that is specified in the OUTLOG= option in the PROC TSMODEL statement. For more information about how to enable and configure logging and about how to access the output log after an invocation of the TSMODEL procedure, see Chapter 11, “The TSMODEL Procedure” (SAS Visual Forecasting: Forecasting Procedures).

PYTHON2, PYTHON3, and R Objects

PYTHON2, PYTHON3, and R objects facilitate interaction with the corresponding external language. To simplify this chapter, these objects are called interpreter objects and are represented in syntax by InterpreterObject. All interpreter objects provide a common interface, whose methods are summarized in Table 4.3. Separate objects are available for PYTHON2 and PYTHON3 because identical code can produce different results in the two Python versions.
### Table 4.3 Methods of the Interpreter Objects (PYTHON2, PYTHON3, and R)

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AddEnvVariable</td>
<td>Registers an environmental variable for the external-language program’s execution environment</td>
</tr>
<tr>
<td>AddVariable</td>
<td>Specifies a variable to be shared between the SAS environment and the external-language program’s environment</td>
</tr>
<tr>
<td>GetExitCode</td>
<td>Returns the external-language program’s exit code from the last call to the Run method</td>
</tr>
<tr>
<td>GetRuntime</td>
<td>Returns the execution time (in seconds) of the last call to the Run method</td>
</tr>
<tr>
<td>GetVariableStatus</td>
<td>Provides status information that is related to transferring data back to the SAS environment</td>
</tr>
<tr>
<td>Initialize</td>
<td>Resets the interpreter object by clearing variables, options, and code</td>
</tr>
<tr>
<td>PushCodeFile</td>
<td>Reads external-language source code from a specified file</td>
</tr>
<tr>
<td>PushCodeFromTable</td>
<td>Reads external-language source code from a specified CAS Table</td>
</tr>
<tr>
<td>PushCodeLine</td>
<td>Reads external-language source code from a local variable or literal</td>
</tr>
<tr>
<td>RemoveEnvVariable</td>
<td>Removes a previously registered environmental variable</td>
</tr>
<tr>
<td>RemoveVariable</td>
<td>“Un-shares” a variable, so that it is no longer transferred between the SAS and external-language environments</td>
</tr>
<tr>
<td>Reset</td>
<td>Resets only a specified attribute of the object</td>
</tr>
<tr>
<td>Run</td>
<td>Executes the “pushed” source code, automatically handling all data exchanges between the SAS and external-language environments</td>
</tr>
<tr>
<td>SetOption</td>
<td>Configures various interpreter options</td>
</tr>
</tbody>
</table>

Figure 4.2 illustrates the data flow through the interpreter objects and their relationships with other components of the EXTLANG package.
Figure 4.2 PYTHON2, PYTHON3, and R Objects Data Flow
## Interpreter Object Synopsis

```plaintext
DECLARE OBJECT obj (PYTHON2) ;
DECLARE OBJECT obj (PYTHON3) ;
DECLARE OBJECT obj (R) ;
```

Method syntax, in order of typical usage:

```plaintext
rc=objc.Initialize () ;
rc=objc.PushCodeFile ('Path/To/FileName') ;
rc=objc.AddVariable (VarName <,'Name', Value, ...) ;
rc=objc.AddEnvVariable (Name, Value <,Name,Value, ... > ) ;
rc=objc.Run () ;
rc=objc.GetVariableStatus (VarName, 'WhichOption') ;
rc=objc.Reset ('What') ;
rc=objc.RemoveVariable (VarName ) ;
rc=objc.RemoveEnvVariable (VarName ) ;
rc=objc.PushCodeLine (Code Statements ) ;
rc=objc.Run () ;
```

## Interpreter Object Methods

### `InterpreterObject.AddEnvVariable Method`

```plaintext
rc=objc.AddEnvVariable (Name, Value <,Name,Value, ... > ) ;
```

Adds one or more environment variables to the external-language program’s execution environment. This method must be enabled by the CAS administrator in the external languages access control configuration by setting `userSetEnv = ALLOW`.

### Input Arguments

You must specify one or more `Name, Value` pairs as input arguments:

- **Name** specifies an ASCII character array that contains the environment variable’s name.
- **Value** is a string of Unicode characters or a numeric scalar that specifies the environment variable’s value. Numeric values are converted to a character array and missing values are converted to empty strings. You cannot pass in an array.

You can call this method multiple times to set multiple environment variables, or you can specify multiple `Name, Value` pairs in one call. For example, you can specify `AddEnvVariable(Name1, Value1, Name2, Value2, ...).`
InterpreterObject.AddVariable Method

```ruby
rc = obj.AddVariable (VarName <,'Name', Value, ...>);
```

Makes the specified variable (VarName) a shared variable—that is, a variable that is automatically transferred to (and optionally from) the external-language program. Shared variables will be available as global variables in the external-language program, where their names will be in uppercase. The following data types are supported:

- numeric scalars
- numeric arrays
- variable-length character strings
- arrays of variable-length strings

There are no restrictions on the number of dimensions for arrays. However, there are restrictions on the types of transformations that are allowed for read/write variables. (See the ARRAYRESIZE, STRINGRESIZE, and STRINGTRUNCATE arguments.) Numeric data are converted to 64-bit floating point values in Python and R. Returned numeric data from Python programs can be native Python integer or float data types, or NumPy integer or floating point data types (int32, int64, float32, float64). Returned numeric data from R programs can be of numeric or integer data type. If the external language is Python and the NumPy package is available, SAS arrays are converted to NumPy arrays in the Python program. Otherwise, SAS arrays are converted to Python lists in the Python program. Note that this could affect the consistency of your program because NumPy arrays and Python lists treat certain numerical operators differently. Consequently, either your program must check the data type of shared arrays or you must be certain whether NumPy is installed or not on all systems in which your program will run.

**Input Arguments**

You must specify the following input argument:

- **VarName** specifies the name of the SAS variable to be shared.

You can also specify the following 'Name', Value pairs:

- **'ALIAS'** takes a character string Value that specifies an alternate name for the variable in the external-language program with the same name that it uses in the SAS environment. However, whether an alias is specified or not, the variable’s name in the external-language program will be in uppercase (for example, “aIr” is changed to “AIR”).

- **'ARRAYRESIZE'** takes a character string Value that specifies whether a variable that is resized in the external-language program is transferred back. You can specify one of the following Values:
  - YES | Y | TRUE | ON allows you to resize this array in the external-language program. A resized array can have at most two dimensions. The AddVariable method fails if VarName is not an array.
NO | N | FALSE | OFF prohibits you from resizing this array in the external-language program. Array data will retain their original values if they are resized in the external-language program. The variable’s status flags will be set accordingly. See the GetVariableStatus method for the available flags.

The default Value is NO.

'READONLY' takes a character string Value that specifies whether to transfer the data back to the SAS environment after the external-language program completes. You can specify one of the following Values:

YES | Y | TRUE | ON does not transfer the data back to the SAS environment (that is, the data will not be affected by the external-language program).

NO | N | FALSE | OFF transfers the data back to the SAS environment upon returning from the external-language program. See status flags STRINGTRUNCATE, STRINGRESIZEFAIL, and ARRAYRESIZE in the GetVariableStatus method for restrictions that are related to the modification of shared variables.

The default Value is YES.

'STRINGRESIZE' takes a character string Value that specifies what happens when you resize a string in the external-language program. You can specify one of the following Values:

YES | Y | TRUE | ON transfers the contents of the string back to the SAS environment even if it is resized. If resizing fails, the STRINGRESIZEFAIL flag will be set to TRUE. You can retrieve the value of this flag via the GetVariableStatus method.

NO | N | FALSE | OFF specifies that the string should not be resized. The original string value will be retained if it is resized in the external-language program. The corresponding error will be recorded in the OUTEXTVARSTATUS table.

The default Value is NO.

'STRINGTRUNCATE' takes a character Value that specifies what happens when a string variable’s size differs when it returns from the external-language program. You can specify one of the following Values:

YES | Y | TRUE | ON truncates the string if it is larger than the original string when it is returned from the external-language program.

NO | N | FALSE | OFF retains the original value of the variable if its size increases in the external-language program.

The default Value is NO.
Interpreter Object Methods

**InterpreterObject.GetExitCode Method**

```plaintext
cr = obj.GetExitCode () ;
```

Returns the exit code that corresponds to the last call to the Run method.

**Arguments**
There are no arguments associated with this method.

**InterpreterObject.GetRuntime Method**

```plaintext
cr = obj.GetRuntime () ;
```

Returns the execution time that corresponds to the last call to the Run method, in seconds.

**Arguments**
There are no arguments associated with this method.

**InterpreterObject.GetVariableStatus Method**

```plaintext
cr = obj.GetVariableStatus ( VarName, 'StatusFlag' ) ;
```

Obtains the value of the specified 'StatusFlag' for the specified variable (VarName). This method applies only to read-write variables—that is, those that have a value of FALSE for the READONLY argument in the call to the AddVariable method. These parameters specify whether the variable was modified in the external-language program and, if so, whether the modification resulted in a change that could have compromised the variable’s data. A variable’s data might not be transferred back to the SAS environment because of unsupported changes in the external-language program. Consequently, you should look at applicable status flags for every shared variable after each call of the Run method to determine whether the data were not transferred back to the SAS environment because of an unsupported modification in the external-language program. If any status flag other than 'UPDATED' is set to 1, the data were not transferred back to the SAS environment and the original data were retained. You can also obtain a table of all variables’ status flags by using the OUTTEXTVARSTATUS object.

This method returns 0 if the specified 'StatusFlag' is not set, or 1 if it is set. A value less than 0 could be returned if an error occurred, such as if an invalid variable name was specified.

**Input Arguments**
You must specify the following input arguments:

- **VarName** specifies the name of the shared variable whose status you are querying. This must be the variable’s alias if you specified one in the call to the AddVariable method.
- **'StatusFlag'** takes a string that contains the status flag of interest. In each call of the GetVariableStatus method, you can specify only one of the following StatusFlags:

  - **'ARRAYRESIZEFAIL'** indicates whether array resizing failed. It will be set to 1 if the array failed to be resized and 0 otherwise. Resized arrays cannot exceed two dimensions. An array could also fail to be resized because of memory limitations.
'ARRAYUNRESIZABLE' indicates whether you resized an unresizable array. It will be set to 1 if the array was resized in the external-language program and the 'ARRAYRESIZE' argument of the AddVariable method was not set to TRUE. It will be set to 0 otherwise.

'DATATYPECHANGED' indicates whether the data type of the variable changed (1) or did not change (0) in the external-language program.

'DEELETED' indicates whether the variable was deleted in the external-language program (1) or was not deleted (0).

'GARbled' indicates whether the variable’s data were potentially garbled when they were transferred back to the SAS environment (1) or were not garbled (0). An array will become garbled if it was resized in the external-language program and the dimension lengths do not match.

'INVALIDSHAPE' indicates whether the returned array has an invalid shape, such as unequal dimension lengths or completely empty, (1) or not (0).

'STRINGRESIZEFAIL' indicates whether a string that was resized in the external-language program could not be resized in the SAS environment (1) or could be resized (0). A string can fail to be resized because of limited memory, unsupported length, and so on.

'STRINGTRUNCATED' indicates whether the string that was returned from the external-language program was truncated (1) or not (0).

'STRINGUNRESIZABLE' indicates whether a non-resizable string was resized in the external-language program (1) or not (0). A string is considered non-resizable if you do not set the 'STRINGRESIZE' parameter to TRUE in the call to the AddVariable method.

'UPDATED' indicates whether the variable was updated by the external-language program (1) or not (0). Note that a variable is considered 'UPDATED' even though it might not have changed in the external-language program. The interpreter object does not check variables’ contents for changes.

**InterpreterObject.Initialize Method**

```plaintext
rc=obj.Initialize () ;
```

Resets the interpreter object by clearing variables, options, and code.

**Arguments**
There are no arguments associated with this method.
**InterpreterObject.PushCodeFile Method**

```
rc = obj.PushCodeFile (PathToFile ) ;
```

Adds the contents of the specified file to the source code to be executed by the external-language interpreter. This method assumes that the source code text in the specified file is encoded in UTF-8. This method must be enabled by the CAS administrator by specifying a `diskWhiteList` in the DEFAULT or GROUP tag in the EXTLANG configuration file.

**Input Arguments**
You must specify the following input argument:

- **FilePath** specifies the path to the file that contains the external-language code to be executed.

**InterpreterObject.PushCodeFromTable Method**

```
rc = obj.PushCodeFromTable (INEXTCODE_Object, Name ) ;
```

Adds the contents of a row whose name is `Name` in the CAS table that is mapped to the `INEXTCODE_Object`. This method does not verify whether the contents of the table consist of valid source code for the interpreter object. This method must be enabled by the CAS administrator specifying `userInlineCode=TRUE` in the external languages access control configuration.

**Input Arguments**
You must specify the following input arguments:

- **INEXTCODE_Object** specifies the INEXTCODE object that contains the code to be executed.
- **Name** specifies the name of the row in the table that contains the external-language code.

**InterpreterObject.PushCodeLine Method**

```
rc = obj.PushCodeLine (code <, 'Name', Value>) ;
```

Specifies external-language code to be executed by the interpreter. This method must be enabled by the CAS administrator specifying `userInlineCode=TRUE` in the external languages access control configuration.

**Input Arguments**
You must specify the following input argument:

- **code** specifies a character string that contains external-language code to be added to the source code buffer. The source code in the buffer is run when the Run method is called.

You can also specify the following `Name`, `Value` pair:

- **NEWLINE** specifies whether to add a new-line character (`\n`) after the specified `code`.
  - **YES | Y | TRUE | ON** add the `\n` after the `code`.
  - **NO | N | FALSE | OFF** do not add the `\n` after `code`.

The default `Value` is YES.
Chapter 4: External Languages Package

**InterpreterObject.RemoveEnvVariable Method**

```plaintext
rc = obj.RemoveEnvVariable (VarName ) ;
```

Removes from the external-language program’s execution environment an environment variable that was previously added via the AddEnvVariable method. In future calls to the Run method, the variable will no longer be passed to the external-language environment.

You must specify the following input argument:

`VarName` specifies the name of the environment variable that is to be excluded from the environment.

**InterpreterObject.RemoveVariable Method**

```plaintext
rc = obj.RemoveVariable (VarName ) ;
```

Removes a variable that was previously added via the AddVariable method. Consequently, the variable will not be transferred between the SAS and external-language environments on future calls to the Run method.

You must specify the following input argument:

`VarName` specifies the name of the variable that should no longer be shared between environments.

**InterpreterObject.Reset Method**

```plaintext
rc = obj.Reset (What ) ;
```

Clears a specified attribute of the interpreter object.

You must specify the following input argument:

`What` specifies what to clear. You can specify one of the following:

- `'CODE'` clears all code that was added via the PushCodeLine, PushCodeFile, and PushCodeFromTable methods.
- `'ENVVARIABLES'` clears all environment variables that were added via the AddEnvVariable method.
- `'OPTIONS'` clears all options that were added via the SetOption method.
- `'VARIABLES'` clears all variables that were added via the AddVariable method.

**InterpreterObject.Run Method**

```plaintext
rc = obj.Run () ;
```

Runs the external-language program. All source code that you added via the PushCodeLine, PushCodeFile, and PushCodeFromTable methods will be combined in the order in which they were added into a single external-language program and run on CAS. All variables you specified via the AddVariable method will automatically be shared between the SAS environment and the external-language program. All variables’ status flags will be cleared before running. However, the external-language program will not run if any of the shared variables were compromised by a previous call to the Run method. Hence, those variables must be removed either by calling the RemoveVariable method or by calling the Initialize method before
calling the Run method. You can see the variable status flags via the GetVariableStatus method or the OUTFEXTVARSTATUS object. The return code that is returned by the Run method is the return code of the method call, not the return code of the external-language program. You must use the GetExitCode method to check the return code of the external-language program.

Data for shared variables for which the READONLY option was set to FALSE are transferred back to the SAS environment even if the external-language program did not exit with a return code of 0. This is done in case you are running a program for which failure is not necessarily catastrophic. Hence, you should verify both the exit code and the variable’s status flags after running the program.

If there is a syntax error in your code, the error text is stored in the OUTFEXTLOG object that is associated with the interpreter object. For languages (such as Python) that display a line number along with the syntax error message, the log text usually displays the line number in your original pushed source code. You will need to calculate the latter in the combined source code text if you used multiple PushCodeLine, PushCodeFromTable, or PushCodeFile method calls to generate the source code. If the EXTLANG package cannot determine the original line of the corresponding error in your source code, it displays the original syntax error message, which includes the line number in the autogenerated source code that was created internally by the EXTLANG package.

An example Python syntax error message is shown in Figure 4.3. You can see that the log message shows a syntax error at line 7 of your pushed code, as indicated by the file “<embedded program>”. Also shown is the original error message text that the Python interpreter produced. Languages that do not show line numbers show only the message text.

Figure 4.3 Example Log Output from a User’s Python Program That Contains a Syntax Error

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
SYNTAX ERROR IN EMBEDDED PROGRAM:
File "<embedded program>", Line 7
def foo
   ^
Syntax Error: Invalid Syntax
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

Similarly, if an unhandled error in your code causes the external-language interpreter to quit, the error message is stored in the OUTFEXTLOG object that is associated with the interpreter object. Languages (such as Python) that display the line number that is associated with the error generally display the line number that corresponds to your original pushed code. The EXTLANG package attempts to provide the line information from your original source code. In this case, it displays “<embedded file>” for the file name attribute of the traceback frame. If the EXTLANG package cannot determine the corresponding line numbers in your embedded program, it shows only the original stack trace that the external-language interpreter generated, so any line numbers shown correspond to the internally autogenerated code. An example error message from a Python program is shown in Figure 4.4. The log message shows that the error occurred in line 3 of a library named thelib, in a function named my_function, which was called from line 23 of your pushed code.
Figure 4.4  Example Log Output from a User's Python Program That Contains an Unhandled Exception

EXCEPTION OCCURRED IN EMBEDDED PROGRAM:
  MESSAGE: name 'out' is not defined
  LINE NUMBER: 3 in /path/to/lib/thelib.py
  TRACEBACK:
    (most recent call last):
    File "<embedded program>", line 23, in <module>
    my_function()
    File "/path/to/lib/thelib.py", line 3, in my_function
    print(out)

An example error message for an R program that contains a syntax error is shown in Figure 4.5 (“function f” is invalid R syntax). Line numbers are not displayed because R does not reliably include line numbers in its error messages. The special header is generally included only when there is a syntax error in the program. If a run-time error occurs, the log displays only the error that is generated by the R interpreter.

Figure 4.5  Example Log Output from a User's R Program That Contains a Syntax Error

ERROR IN EMBEDDED PROGRAM:
  MESSAGE:
    Error: unexpected symbol in:
      "print("0")
    function f"
    Execution halted

Arguments
There are no arguments associated with this method.

InterpreterObject.SetOption Method

rc=obj.SetOption ('Name', Value <,'Name', Value, ... ) ;

Sets one or more run-time options for subsequent calls to the Run method. You can specify one or more of the following 'Name', Value pairs:

'EXECPATH'  takes a string Value that specifies the path to the external-language interpreter’s executable. The path you provide must be accessible by all CAS workers if CAS is running in an MPP environment with more than one node. By default, Value is set to the executable path that is specified by the administrator. This argument can be specified only if the userSetInterpreter attribute of the LANGUAGE tag is set to ALLOW in the EXTLANG configuration file. An error is logged if you try to specify this argument when the CAS administrator has not set the userSetInterpreter attribute to 'ALLOW' in the external languages access control configuration.

'RUNID'  takes a string Value that specifies a unique ID to assign to the external-language program’s execution. The default Value is UNNAMED RUN.
OUTEXTLOG Object

The OUTEXTLOG collector object stores execution and resource usage logs in a CAS table. The execution log consists of all output that the external-language program sends to the operating system’s standard output and standard error streams. The resource usage log shows various resource utilization statistics.

Table 4.4 shows the contents of the OUTEXTLOG object.

<table>
<thead>
<tr>
<th>Column</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>LOGLEN</em></td>
<td>Numeric</td>
<td>Length of the log and resource utilization text</td>
</tr>
<tr>
<td><em>LOGTEXT</em></td>
<td>String</td>
<td>The output log and resource utilization text</td>
</tr>
<tr>
<td><em>LOGTYPE</em></td>
<td>String</td>
<td>The value that was specified in the WHAT argument of the Collect method: 'STATS', 'EXECUTION', or 'ALL'.</td>
</tr>
<tr>
<td><em>RUNID</em></td>
<td>VARCHAR</td>
<td>Run identification string</td>
</tr>
</tbody>
</table>

Table 4.5 summarizes the methods that are associated with the OUTEXTLOG object.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Collect</td>
<td>Collect the external-language program’s output and/or resource usage data</td>
</tr>
<tr>
<td>nrows</td>
<td>Return the number of rows that were collected</td>
</tr>
<tr>
<td>SetOption</td>
<td>Set a named option</td>
</tr>
</tbody>
</table>
OUTEXTLOG Synopsis

DECLARE OBJECT obj (OUTEXTLOG) ;

Method syntax, in order of typical usage:

\[ rc = obj.Collect (InterpreterObject < , 'What' > ) ; \]
\[ rc = obj.nrows () ; \]
\[ rc = obj.SetOption ( 'Name' , Value ) ; \]

OUTEXTLOG Methods

OUTEXTLOG.Collect Method

\[ rc = obj.Collect (InterpreterObject < , 'What' > ) ; \]

Collects the standard output and standard error streams from the specified Interpretation Object. The Collect method also collects resource usage data from the external-language program.

Input Arguments

You must specify the following input argument:

InterpreterObject specifies the interpreter object (PYTHON2, PYTHON3, or R) that holds the output and resource utilization text for the external-language program’s execution.

You can also specify the following argument:

'What' specifies which log to collect. Acceptable values are:

- STATS collects only the resource utilization statistics log.
- EXECUTION collects only the execution log, which consists of the standard output and error of your external-language program.
- ALL collects the execution log followed by the resource utilization log.

The default is ALL.

OUTEXTLOG.nrows Method

\[ nrows = obj.nrows () ; \]

Queries the OUTEXTLOG object for its current row count.

Arguments

There are no arguments associated with this method.
**OUTEXTLOG.SetOption Method**

```plaintext
rc = obj.SetOption ('Name', Value) ;
```

Sets a named option.

**Input Arguments**
You must specify the following ('Name', Value) pair:

- `'Name'` specifies the named option to set. You can specify the following:
  - **RUNID** takes a character string `Value` that specifies the Run ID to put in the collected table. The default `Value` is UNNAMED RUN.

---

**OUTEXTVARSTATUS Object**

The OUTEXTVARSTATUS collector object collects shared variables’ status flags and stores them in a CAS table. Shared variables include all variables that are added to an interpreter object via its `AddVariable` method. Their status is updated upon completion of the interpreter object’s Run method. A variable’s status indicates whether the variable was modified in the external-language program and, if so, whether the modification could have compromised the variable’s data. Table 4.6 shows the contents of the OUTEXTVARSTATUS object. For more information about the available status flags, see the `InterpreterObject.GetVariableStatus` Method.
### Table 4.6  Contents of the OUTEXTVARSTATUS Object

<table>
<thead>
<tr>
<th>Column</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>DATATYPE</em></td>
<td>String</td>
<td>Variable’s data type</td>
</tr>
<tr>
<td><em>NAME</em></td>
<td>String</td>
<td>Variable’s name</td>
</tr>
<tr>
<td><em>RUNID</em></td>
<td>VARCHAR</td>
<td>Run identification string</td>
</tr>
<tr>
<td>ARRAYRESIZEFAIL</td>
<td>Numeric</td>
<td>Specifies whether the array could not be resized</td>
</tr>
<tr>
<td>ARRAYUNRESIZABLE</td>
<td>Numeric</td>
<td>Specifies whether the array, for which the value of the</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ARRAYRESIZE argument is NO, was resized</td>
</tr>
<tr>
<td>DATATYPECHANGED</td>
<td>Numeric</td>
<td>Specifies whether the variable’s data type changed</td>
</tr>
<tr>
<td>DELETED</td>
<td>Numeric</td>
<td>Specifies whether the variable was deleted</td>
</tr>
<tr>
<td>GARBLED</td>
<td>Numeric</td>
<td>Specifies whether the array’s data was possibly garbled</td>
</tr>
<tr>
<td>INVALIDSHAPE</td>
<td>Numeric</td>
<td>Specifies whether the array’s shape became invalid</td>
</tr>
<tr>
<td>STRINGRESIZEFAIL</td>
<td>Numeric</td>
<td>Specifies whether the string could not be resized</td>
</tr>
<tr>
<td>STRINGTRUNCATED</td>
<td>Numeric</td>
<td>Specifies whether the string was truncated</td>
</tr>
<tr>
<td>STRINGUNRESIZABLE</td>
<td>Numeric</td>
<td>Specifies whether the non-resizable string was resized</td>
</tr>
<tr>
<td>UPDATED</td>
<td>Numeric</td>
<td>Specifies whether the variable’s modifications were</td>
</tr>
<tr>
<td></td>
<td></td>
<td>returned to the SAS environment</td>
</tr>
</tbody>
</table>

Table 4.7 summarizes the methods that are associated with the OUTEXTVARSTATUS object.

### Table 4.7  Methods of the OUTEXTVARSTATUS Object

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Collect</td>
<td>Collect the output and resource usage statistics</td>
</tr>
<tr>
<td>nrows</td>
<td>Return the number of rows collected</td>
</tr>
<tr>
<td>SetOption</td>
<td>Set named option</td>
</tr>
</tbody>
</table>
OUTEXTVARSTATUS Synopsis

DECLARE OBJECT obj (OUTEXTVARSTATUS) ;

Method syntax, in order of typical usage:

rc=obj.Collect (InterpreterObject) ;
rc=obj.nrows () ;
rc=obj.SetOption (‘Name’, Value ) ;

OUTEXTVARSTATUS Methods

OUTEXTVARSTATUS.Collect Method

rc=obj.Collect (InterpreterObject) ;

Collects the status flags of all the specified InterpreterObject’s writable shared variables and stores them in a CAS table. Writable variables are those for which the value of the READONLY argument in the AddVariable method is NO.

Input Arguments
You must specify the following input argument:

InterpreterObject specifies the interpreter object (PYTHON2, PYTHON3, or R) from which to get variables’ status.

OUTEXTVARSTATUS.nrows Method

nrows=obj.nrows () ;

Queries the OUTEXTVARSTATUS object for its current row count.

Arguments
There are no arguments associated with this method.

OUTEXTVARSTATUS.SetOption Method

rc=obj.SetOption (‘Name’, Value) ;

Sets a named option.

Input Arguments
You must specify the following (‘Name’, Value) pair:

‘Name’ a character string that specifies the named option to set. You can set the following:

‘RUNID’ takes a variable-length character string Value that specifies the Run ID to put in the collected table. The default is UNNAMED RUN.
OUTEXTCODE Object

The OUTEXTCODE collector object stores external-language source code in a CAS table. All code you have “pushed” between calls to the interpreter object’s Initialize or Reset('CODE') methods is stored. This includes code that was added via the interpreter object’s PushCodeLine, PushCodeFile, and PushCodeFromTable methods. This object enables you to create and store external-language source code that can be replayed later using an INEXTCODE object.

Table 4.8 shows the contents of the OUTEXTCODE Object.

<table>
<thead>
<tr>
<th>Column</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>CODELEN</em></td>
<td>Numeric</td>
<td>Specifies the number of characters in the source code</td>
</tr>
<tr>
<td><em>CODETEXT</em></td>
<td>String</td>
<td>Contains the source code text</td>
</tr>
<tr>
<td><em>RUNID</em></td>
<td>VARCHAR</td>
<td>Identifies the run</td>
</tr>
<tr>
<td><em>SOURCE</em></td>
<td>VARCHAR</td>
<td>Specifies whether the source code text was user-specified or internally generated from another object.</td>
</tr>
</tbody>
</table>

Table 4.9 summarizes the methods that are associated with the OUTEXTCODE object.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Collect</td>
<td>Collect the output and resource usage statistics</td>
</tr>
<tr>
<td>nrows</td>
<td>Return the number of rows collected</td>
</tr>
<tr>
<td>SetOption</td>
<td>Set named option</td>
</tr>
</tbody>
</table>

OUTEXTCODE Synopsis

DECLARE OBJECT obj (OUTEXTCODE) ;

Method syntax, in order of typical usage:

```plaintext
rc=obj.Collect (InterpreterObject) ;
rc=obj.nrows () ;
rc=obj.SetOption ('Name', Value ) ;
```
OUTEXTCODE Methods

OUTEXTCODE.Collect Method

```csharp
rc = obj.Collect (InterpreterObject) ;
```

Collects the source code from the specified `InterpreterObject` and stores it in a CAS table. The `InterpreterObject`'s source code includes all code you “pushed” via the interpreter object’s PushCodeFromTable, PushCodeFile, and PushCodeLine methods.

**Input Arguments**

You must specify the following input argument:

- `InterpreterObject` specifies the interpreter object (PYTHON2, PYTHON3, or R) from which to get the source code.

OUTEXTCODE.nrows Method

```csharp
nrows = obj.nrows () ;
```

Queries the OUTEXTCODE object for its current row count.

**Arguments**

There are no arguments associated with this method.

OUTEXTCODE.SetOption Method

```csharp
rc = obj.SetOption (‘Name’, Value ) ;
```

Sets a named option.

**Input Arguments**

You must specify the following as a (`Name`, `Value`) pair:

- `Name` a character string that specifies the named option to set. You can set the following:
  - `RUNID` takes a variable-length character array `Value` that specifies the Run ID to put in the collected table. The default is UNNAMED RUN.
INEXTCODE Object

The INEXTCODE repeater object loads source code from a CAS table. This object loads only the source code. You must still specify any execution-related parameters, such as shared variables.

Table 4.10 shows the INEXTCODE table schema that is produced by the OUTEXTCODE object.

<table>
<thead>
<tr>
<th>Column</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>CODELEN</em></td>
<td>Numeric</td>
<td>Specifies the number of characters in the source code</td>
</tr>
<tr>
<td><em>CODETEXT</em></td>
<td>String</td>
<td>Contains the source code text</td>
</tr>
<tr>
<td><em>RUNID</em></td>
<td>VARCHAR</td>
<td>Identifies the run</td>
</tr>
<tr>
<td><em>SOURCE</em></td>
<td>VARCHAR</td>
<td>Specifies whether the source code text was user-specified or internally generated from another object.</td>
</tr>
</tbody>
</table>

Table 4.11 summarizes the methods that are associated with the INEXTCODE object.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>nrows</td>
<td>Query INEXTCODE object for its current row count</td>
</tr>
</tbody>
</table>

INEXTCODE Synopsis

DECLARE OBJECT obj (INEXTCODE) ;

Method syntax:

rc=obj.nrows () ;
INEXTCODE Methods

INEXTCODE.nrows Method

```
rc=obj.nrows();
```

Queries the INEXTCODE object for its current row count. A returned missing value indicates that the INEXTCODE object has not been successfully configured.

**Arguments**
There are no arguments associated with this method.

Examples: EXTLANG Package

Throughout this section, it is assumed that you have already started a CAS session and that the data tables that are used in this section are in mycas, a CAS library that you have necessary permissions to work with. This section assumes that you are familiar with the general workings of the TSMODEL procedure; for more information, see Chapter 11, “The TSMODEL Procedure” (SAS Visual Forecasting: Forecasting Procedures). Working knowledge of Python and R is also assumed.

Using CAS Sessions and CAS Engine Librefs

SAS Cloud Analytic Services (CAS) is the analytic server and associated cloud services in SAS Viya. This section describes how to create a CAS session and set up a CAS engine libref that you can use to connect to the CAS session. It assumes that you have a CAS server already available; contact your system administrator if you need help starting and terminating a server. This CAS server is identified by specifying the host on which it runs and the port on which it listens for communications. To simplify your interactions with this CAS server, the host information and port information for the server are stored as SAS option values that are retrieved automatically whenever this CAS server needs to be accessed. You can examine the host and port values for the server at your site by using the following statements:

```
proc options option=(CASHOST CASPORT);
run;
```

In addition to starting a CAS server, your system administrator might also have created a CAS session and a CAS engine libref for your use. You can define your own sessions and CAS engine librefs that connect to the CAS server as shown in the following statements:

```
cas mysess;
libname mycas cas sessref=mysess;
```

The CAS statement creates the CAS session named mysess, and the LIBNAME statement creates the mycas CAS engine libref that you use to connect to this session. It is not necessary to explicitly name the CASHOST and CASPORT of the CAS server in the CAS statement, because these values are retrieved from the corresponding SAS option values.
If you have created the `mysess` session, you can terminate it by using the TERMINATE option in the CAS statement as follows:

```sas
cas mysess terminate;
```

For more information about the CAS statement and the LIBNAME statement, see *SAS Cloud Analytic Services: User’s Guide*. For general information about CAS and CAS sessions, see *SAS Cloud Analytic Services: Fundamentals*.

---

**Example 4.1: Calculate Moving Average of the Airline Series**

This example uses the airline passenger data, which are given as Series G in Box and Jenkins (1976) and have been used in time series analysis literature as an example of a nonstationary seasonal time series.

This example shows the basics of using the EXTLANG package to integrate Python and R code into your SAS program. It demonstrates how to use the PYTHON2, R, and OUTEXTLOG objects to run basic Python and R code that calculates the moving average of a univariate time series. It demonstrates the use of shared variables and how they can be transferred among the SAS, Python, and R environments. Comments in the code correspond to items in the explanation that follows the code.

```sas
/* Comment 1 */
data mycas.air;
  set sashelp.air;
run;

/* Comment 2 */
PROC TSMODEL DATA=mycas.air
  OUTARRAY=mycas.outarray
  OUTSCALAR=mycas.outscalar
  OUTOBJ=(pylog=mycas.pylog rlog=mycas.rlog);
/* Comment 3 */
ID date INTERVAL=MONTH;
/* Comment 4 */
VAR air;
/* Comment 5 */
OUTSCALAR runtime;
OUTARRAY pyair mavg r_mavg;
/* Comment 6 */
REQUIRE extlang;
SUBMIT;
/* Comment 7 */
/* PART 1 */
/* Copy input variable AIR into PYAIR */
do i=1 to _LENGTH_;  
  pyair[i] = air[i];
end;

/* PART 2 */
/* Define a PYTHON2 interpreter object to manipulate AIR data 
and use Python to calculate a moving average */
declare object py(PYTHON2);
```
Example 4.1: Calculate Moving Average of the Airline Series

```
rc = py.Initialize();
/* Create the script */
rc = py.PushCodeLine('import numpy as np');
/* Create window of size 12 and calculate the moving average. */
rc = py.PushCodeLine('w = np.ones(12)/12');
rc = py.PushCodeLine('nans = np.empty(11) ; nans[:] = np.nan');
/* SAS variable name is always UPPERCASE in external-language program */
rc = py.PushCodeLine('._air = np.concatenate((nans,AIR))');
rc = py.PushCodeLine('MAVG = np.convolve(_air, w, mode="valid")');
/* Multiply all values of PYAIR by 10 */
rc = py.PushCodeLine('AIR *= 10');
/* Specify shared variables */
rc = py.AddVariable(pyair,'ALIAS','air', 'READONLY', 'FALSE');
RC = PY.AddVariable(mavg, 'READONLY', 'FALSE');
/* Run */
rc = py.Run();
runtime = py.GetRuntime();
/* Store the execution and resource usage statistics logs */
declare object pylog(OUTEXTLOG);
rc = pylog.Collect(py, 'ALL');

/* PART 3 */
/* Define an R interpreter object to use R to calculate a moving average */
declare object r(R);
rc = r.Initialize();
/* Assemble the code */
/* Moving average with window size of 12 */
rc = r.PushCodeLine("ma <- function(x,w=12)
{filter(x,rep(1/w,w), sides=1)}\)
rc = r.PushCodeLine("R_MAVG <- ma(AIRX10)\)
/* Specify shared variables - change name of "AIR" to "AIRx10" since it's been multiplied by 10 */
rc = r.AddVariable(pyair,'ALIAS','AIRx10');
rc = r.AddVariable(r_mavg, 'READONLY', 'FALSE');
/* rc = r.SetOption('EXECPATH','/opt/R-3.5.1/bin/Rscript'); */
rc = r.Run();
runtime = r.GetRuntime();
/* Store the execution and resource usage statistics logs */
declare object rlog(OUTEXTLOG);
rc = rlog.Collect(r, 'ALL');
ENDSUBMIT;
RUN;
```

The example is structured as follows:

- **Comment 1:** The DATA step loads the airline passenger set from the sashelp library into the CAS library mycas.

- **Comment 2:** The PROC TSMODEL statement specifies the input data table (mycas.air), multiple output variables (mycas.outarray and mycas.outscalar), and multiple objects to store the external-language program’s output (mycas.pylog and mycas.rlog).
Comment 3: The ID statement specifies date as the time index variable, and the INTERVAL= option indicates that monthly data are used.

Comment 4: The VAR statement specifies the input data variable, air, which contains the airline series.

Comment 5: The OUTSCALAR and OUTARRAY statements specify the local variables: runtime stores the external-language program’s execution time; pyair stores the value of air after being multiplied by 10 in the Python program; mavg stores the moving average that is calculated in the Python program; and \( r_{\text{mavg}} \) stores the moving average that is calculated in the R program.

Comment 6: The REQUIRE statement specifies the EXTLANG package, which is needed for this example.

Comment 7: The statements between the SUBMIT and ENDSUBMIT statements use the EXTLANG package’s objects to run the Python and R programs in your CAS session. These statements are grouped into three parts:

- **PART 1:** Copy the contents of air to pyair.
- **PART 2:** Initialize, populate, and run the PYTHON2 object. All Python source code is entered via the PushCodeLine method for this simple Python program. Two shared variables (pyair and mavg) are specified using the AddVariable method: pyair brings the input data table into the Python program, where it will be modified; and mavg brings the moving average back to the SAS program. The call to the AddVariable method for pyair specifies that the alias “AIR” should be used to refer to pyair in the Python code. It is specified as a read/write variable, because its modified data will be used in the R program. You are not bound to a single external language. The execution logs of the Python program are stored in the OUTEXTLOG object pylog.

- **PART 3:** Initialize, populate, and run the R object. This section of code follows a pattern that is similar to the pattern in the Python section. It starts by initializing an R object, and then it pushes two lines of R code to enable the calculation of a moving average with a window size of 12. PYAIR uses the default access-mode (read-only) since it is not modified in the R program. Note that there is a commented line that demonstrates how to set the executable path, in case R is not installed in the search path or a specific executable is desired (or both). Be aware that the ability to set “EXECPATH” can be disabled by the CAS administrator. After the code is run, the execution time is stored in runtime. Then, the log data are stored in the OUTTEXTLOG object.

Output 4.1.1 shows the output for observations 24–48. As you can see, PYAIR was multiplied by 10. Since PYAIR was designated a read/write variable in the AddVariable method call, its updated values are brought back to the SAS environment and subsequently pushed to the R environment, where the updated values are used to calculate the moving average in R. Consequently, the values for the moving average in the R program (Column 6) are a factor of 10 larger than those of the Python program (Column 5). The log data stored in rlog and pylog can be seen in the listing file. Note that the transformations to air, mavg, and \( r_{\text{mavg}} \) did not modify the array size. Doing so would have resulted in their data not being transferred back to the SAS environment, because the value of the ‘ARRAYRESIZE’ argument was not YES.
Table 4.2.1 Moving Average Calculated in Python and R (Partial Output)

<table>
<thead>
<tr>
<th>Obs</th>
<th>DATE</th>
<th>AIR</th>
<th>pyair</th>
<th>mavg</th>
<th>r_mavg</th>
</tr>
</thead>
<tbody>
<tr>
<td>24</td>
<td>DEC1950</td>
<td>140</td>
<td>1400</td>
<td>139.6666667</td>
<td>1396.6666667</td>
</tr>
<tr>
<td>25</td>
<td>JAN1951</td>
<td>145</td>
<td>1450</td>
<td>142.1666667</td>
<td>1421.6666667</td>
</tr>
<tr>
<td>26</td>
<td>FEB1951</td>
<td>150</td>
<td>1500</td>
<td>144.1666667</td>
<td>1441.6666667</td>
</tr>
<tr>
<td>27</td>
<td>MAR1951</td>
<td>178</td>
<td>1780</td>
<td>147.25</td>
<td>1472.5</td>
</tr>
<tr>
<td>28</td>
<td>APR1951</td>
<td>163</td>
<td>1630</td>
<td>149.5833333</td>
<td>1495.8333333</td>
</tr>
<tr>
<td>29</td>
<td>MAY1951</td>
<td>172</td>
<td>1720</td>
<td>153.5</td>
<td>1535</td>
</tr>
<tr>
<td>30</td>
<td>JUN1951</td>
<td>178</td>
<td>1780</td>
<td>155.9166667</td>
<td>1559.1666667</td>
</tr>
<tr>
<td>31</td>
<td>JUL1951</td>
<td>199</td>
<td>1990</td>
<td>158.3333333</td>
<td>1583.3333333</td>
</tr>
<tr>
<td>32</td>
<td>AUG1951</td>
<td>199</td>
<td>1990</td>
<td>160.75</td>
<td>1607.5</td>
</tr>
<tr>
<td>33</td>
<td>SEP1951</td>
<td>184</td>
<td>1840</td>
<td>162.9166667</td>
<td>1629.1666667</td>
</tr>
<tr>
<td>34</td>
<td>OCT1951</td>
<td>162</td>
<td>1620</td>
<td>165.3333333</td>
<td>1653.3333333</td>
</tr>
<tr>
<td>35</td>
<td>NOV1951</td>
<td>146</td>
<td>1460</td>
<td>168</td>
<td>1680</td>
</tr>
<tr>
<td>36</td>
<td>DEC1951</td>
<td>166</td>
<td>1660</td>
<td>170.1666667</td>
<td>1701.6666667</td>
</tr>
<tr>
<td>37</td>
<td>JAN1952</td>
<td>171</td>
<td>1710</td>
<td>172.3333333</td>
<td>1723.3333333</td>
</tr>
<tr>
<td>38</td>
<td>FEB1952</td>
<td>180</td>
<td>1800</td>
<td>174.8333333</td>
<td>1748.3333333</td>
</tr>
<tr>
<td>39</td>
<td>MAR1952</td>
<td>193</td>
<td>1930</td>
<td>176.0833333</td>
<td>1760.8333333</td>
</tr>
<tr>
<td>40</td>
<td>APR1952</td>
<td>181</td>
<td>1810</td>
<td>177.5833333</td>
<td>1775.8333333</td>
</tr>
<tr>
<td>41</td>
<td>MAY1952</td>
<td>183</td>
<td>1830</td>
<td>178.5</td>
<td>1785</td>
</tr>
<tr>
<td>42</td>
<td>JUN1952</td>
<td>218</td>
<td>2180</td>
<td>181.8333333</td>
<td>1818.3333333</td>
</tr>
<tr>
<td>43</td>
<td>JUL1952</td>
<td>230</td>
<td>2300</td>
<td>184.4166667</td>
<td>1844.1666667</td>
</tr>
<tr>
<td>44</td>
<td>AUG1952</td>
<td>242</td>
<td>2420</td>
<td>188</td>
<td>1880</td>
</tr>
<tr>
<td>45</td>
<td>SEP1952</td>
<td>209</td>
<td>2090</td>
<td>190.0833333</td>
<td>1900.8333333</td>
</tr>
<tr>
<td>46</td>
<td>OCT1952</td>
<td>191</td>
<td>1910</td>
<td>192.5</td>
<td>1925</td>
</tr>
<tr>
<td>47</td>
<td>NOV1952</td>
<td>172</td>
<td>1720</td>
<td>194.6666667</td>
<td>1946.6666667</td>
</tr>
<tr>
<td>48</td>
<td>DEC1952</td>
<td>194</td>
<td>1940</td>
<td>197</td>
<td>1970</td>
</tr>
</tbody>
</table>

Example 4.2: Ensuring Code Integrity

This example uses the same airline passenger data that are used in Example 4.1. It demonstrates the behavior of interpreter objects when problematic code is input. Failures in external-language code will not cause SAS to fail, but shared variables’ data might not be successfully returned to SAS—either the data will not be returned or incorrect data can be returned if the external-language program fails. Hence, it is imperative that you verify the external-language program’s return code and writable variables’ status after each call to the Run method to ensure the integrity of your data processing. The correct procedure for verifying integrity of shared data is demonstrated in this example. Comments in the code correspond to items in the explanation that follows the code.

```/* Comment 1 */
data mycas.air;
  set sashelp.air;
run;```
/* Comment 2 */
PROC TSMODEL DATA=mycas.air OUTARRAY=mycas.outarray OUTSCALAR=mycas.outscalar
OUTOBJ=(pylog=mycas.pylog pyvarstatus=mycas.pyvarstatus pyvarstatusAfter=mycas.pyvarstatusAfter);
/* Comment 3 */
ID date INTERVAL=MONTH;
/* Comment 4 */
VAR air;
/* Comment 5 */
OUTSCALAR varUpdated exitStatus varUpdatedAfter exitStatusAfter;
OUTARRAY pyair;
/* Comment 6 */
REQUIRE extlang;
/* Comments 7a-d */
SUBMIT;
/* Comment 7a */
do i=1 to _LENGTH_;  
  pyair[i] = air[i];
end;
/* Comment 7b */
declare object py(PYTHON2);
rc = py.Initialize();
/* Create script with invalid code. Note that the variable name 
   should be 'MYAIR' */
rc = py.PushCodeLine('myAir *= 10') ;
rc = py.AddVariable(pyair, 'ALIAS','myAir', 'READONLY', 'FALSE') ;
/* Run */
rc = py.Run();
/* Comment 7c */
exitStatus = py.GetExitCode() ;
varUpdated = py.GetVariableStatus("MYAIR", "UPDATED") ;
/* Use an OUTTEXTVARSTATUS object to get all flags for pyair */
declare object pyvarstatus(OUTEXTVARSTATUS) ;
rc = pyvarstatus.Collect(py) ;
/* Comment 7d */
rc = py.Reset("CODE") ;
rc = py.PushCodeLine("MYAIR *= 10") ;
exitStatusAfter = py.Run() ;
varUpdatedAfter = py.GetVariableStatus("MYAIR", "UPDATED") ;
declare object pylog(OUTEXTLOG);
rc = pylog.Collect(py,'ALL');

declare object pyvarstatusAfter(OUTEXTVARSTATUS) ;
rc = pyvarstatusAfter.Collect(py) ;
ENDSUBMIT;
RUN;
Example 4.2: Ensuring Code Integrity

The example is structured as follows:

- **Comment 1:** The DATA step is the same as the one used in Example 4.1. It makes the airline passenger data available in CAS session mycas.

- **Comment 2:** The PROC TSMODEL statement specifies the input data table (mycas.air) and multiple output variables: array mycas.outarray, scalar mycas.outscalar, and various objects (pylog, pyvarstatus, and pyvarstatusAfter).

- **Comment 3:** The ID statement specifies date as the time index variable, and the INTERVAL= option indicates a monthly data interval.

- **Comment 4:** The VAR statement specifies the input data variable, air, which contains the airline series.

- **Comment 5:** The OUTSCALAR and OUTARRAY statements specify the local variables: varUpdated stores a variable’s UPDATED status flag after the first call to the Run method; exitStatus stores the exit status of the external-language program after the first call to the Run method; varUpdatedAfter stores the variable’s UPDATED status after the second call to the Run method; exitStatusAfter stores the external-language program’s exit code after the second call to the Run method. Separate variables are used for the two Run method calls so that they can all be displayed together after the program completes. The variable pyair contains a copy of the airline data, which will be passed to the Python program.

- **Comment 6:** The REQUIRE statement specifies the EXTLANG package, which is needed for this example.

- **Comment 7:** The statements between the SUBMIT and ENDSUBMIT statements use the EXTLANG package’s objects to run the Python program. The following actions are performed:
  
  - **Comment 7a:** The contents of the airline data are copied into pyair. This is done to compare the data table’s values before and after manipulating them in the Python program.
  
  - **Comment 7b:** The PYTHON2 object is created, a line of code is pushed, pyair is specified as a shared variable with alias MYAIR, and the interpreter is run. Note that the variable name is incorrect in the source code that is pushed via the PushCodeLine method—variables are always made uppercase in the external-language program, regardless of how they are specified in the SAS environment. Since Python variables are case-sensitive, myAir is an undefined symbol in the Python program.
  
  - **Comment 7c:** The exit code of the Python program is stored in exitStatus. The value of the UPDATED flag for PYAIR is stored in varUpdated. Note that the variable’s alias, MYAIR, is used as the argument to the GetVariableStatus method. If the variable has an alias and you do not use it, the GetVariableStatus method will fail. In addition, the collector object pyvarstatus stores the PYTHON2 object’s variable flags into a CAS table.
  
  - **Comment 7d:** The next section of code clears the previous source code, adds a new line of valid source code, and runs the code. This time, the Python program’s exit status is stored in exitStatusAfter and pyair’s UPDATED flag is stored in varUpdatedAfter. The table of py’s variable status flags is stored in pyvarstatusAfter.
Output 4.2.1 shows the outputs. The first table shows the values of the four status variables. Because variable data are transferred back to the SAS environment even if the external-language program fails and because the interpreter object does not check whether a variable was actually changed in the external-language program, varUpdated=1. Because of the typo in the source code (“myAir” instead of “MYAIR”), exitStatus=1. The source code that is used in the second call to the Run method is valid, so exitStatusAfter=0 and varUpdatedAfter=1.

The second table shows the contents of the OUTEXTVARSTATUS collector instance, pyvarstatus, which reflects the status of MYAIR after the first call to the Run method. Some of the flags are excluded in order to fit the table in one row. Only the UPDATED flag is set, as described in the previous paragraph. All the other flags’ values are 0, which is expected because the variable was not modified before the program exited abruptly.

The third table shows the same status flags for pyvarstatusAfter, which is populated after the second call to the Run method. As indicated in the first table, the UPDATED flag is 1. All other flags are 0 because myAir was not deleted, its data type did not change, and it was not resized. Also, the string-related status flags do not apply to myAir because it is numeric.

Output 4.2.1 External-Language Program and Variables’ Status

<table>
<thead>
<tr>
<th>Variable</th>
<th>status and Run method return codes</th>
</tr>
</thead>
<tbody>
<tr>
<td>varUpdated</td>
<td>exitStatus</td>
</tr>
<tr>
<td>1</td>
<td>63232</td>
</tr>
</tbody>
</table>

Variable status flags after calling problematic code (Partial Output)

<table>
<thead>
<tr>
<th><em>NAME</em></th>
<th>UPDATED</th>
<th>GARBLED</th>
<th>DATATYPECHANGED</th>
<th>ARRAYRESIZEFAIL</th>
<th>ARRAYUNRESIZABLE</th>
<th>INVALIDSHAPE</th>
</tr>
</thead>
<tbody>
<tr>
<td>myAir</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Variable status flags after calling good code (Partial Output)

<table>
<thead>
<tr>
<th><em>NAME</em></th>
<th>UPDATED</th>
<th>GARBLED</th>
<th>DATATYPECHANGED</th>
<th>ARRAYRESIZEFAIL</th>
<th>ARRAYUNRESIZABLE</th>
<th>INVALIDSHAPE</th>
</tr>
</thead>
<tbody>
<tr>
<td>myAir</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Example 4.3: BY-Group-Specific Code

This example demonstrates how different Python source code can be applied to different BY groups using environment variables. A synthetic data table is created to demonstrate this functionality, which requires that the CAS administrator has enabled the ability to add environment variables to the Python program’s running environment by setting the userSetEnv attribute to ALLOW in the external languages access control configuration. The example can be modified to use shared variables instead of environment variables if you are unable to set environment variables. The data consist of three copies of the sashelp.air data, with an additional variable called bykey. The value of bykey is set to '1' for the first copy of the data, '2' for the second copy, and '3' for the third copy. Each BY-group’s Python program performs two basic functions. The first function is calculating the average of its time series, using a methodology that is based on its group number. BY group 1 calculates the mean, BY group 2 calculates the median, and BY group 3 calculates the mode. This is accomplished by sending a different line of source code to each BY group. The other function of the Python program is to simply print the contents of an environment variable, which will be set according
Example 4.3: BY-Group-Specific Code

159

to the BY group. Because the SAS program runs within the TSMODEL procedure, each BY group will run in a different thread and a separate Python interpreter process will be invoked for each one. This example uses the PYTHON3 object in order to use functions in the statistics module that is introduced in Python 3.4. This program is structured as follows:

```sas
/* Comment 1 */
DATA mycas.air;
  SET sashelp.air;
  do bykey = 1 to 3;
    output;
  end;
RUN;

/* Comment 2 */
PROC TSMODEL DATA=mycas.air OUTARRAY=mycas.outarray
   OUTSCALAR=mycas.outscalar OUTOBJ=(pylog=mycas.pylog
                  pyvarstatus=mycas.pyvarstatus)
   OUTLOG=mycas.outlog nthreads=1 ;
/* Comment 3 */
ID date INTERVAL=MONTH ;
/* Comment 4 */
BY bykey;
/* Comment 5 */
VAR air;
OUTSCALAR avg exitStatus;
REQUIRE extlang;
PRINT outlog;
/* Comment 6 */
SUBMIT;
  /* Comment 6a */
  declare object py(PYTHON3);
  rc = py.Initialize();
  rc = py.PushCodeLine('import os') ;
  rc = py.PushCodeLine('import statistics as st') ;
  /* rc = py.PushCodeLine('import numpy as np') ; */
  if bykey = 1 then
    do ;
      rc = py.PushCodeLine('AVG=round(st.mean(AIR), 3)') ;
      rc = py.AddEnvVariable("METHOD", "mean") ;
    end ;
  if bykey = 2 then
    do ;
      rc = py.PushCodeLine('AVG=st.median(AIR)') ;
      rc = py.AddEnvVariable("METHOD", "median") ;
    end ;
  if bykey = 3 then
    do ;
      rc = py.PushCodeLine('AVG=st.mode(AIR)') ;
      rc = py.AddEnvVariable("METHOD", "mode") ;
    end ;
  /* Comment 6c */
  rc = py.PushCodeLine("print(os.environ['METHOD'])") ;
```
The program is structured as follows:

- **Comment 1:** The DATA step loads the sashelp.air data table and creates a new data table that consists of three copies of the data, each copy containing an additional variable bykey, which is set to 1 for the first copy, 2 for the second copy, and 3 for the final copy. Hence, there are 144 observations for which bykey='1', 144 for which bykey='2', and 144 for which bykey='3'. The resulting data are available in CAS as a data table named mycas.air.

- **Comment 2:** The PROC TSMODEL statement specifies the input data table (mycas.air) and multiple parameters: the table mycas.outscalar will hold the scalar variables, the table mycas.outarray will hold the array variables, the OUTOBJ parameter specifies the collector objects to be produced, and the OUTLOG parameter specifies the log collector.

- **Comment 3:** The ID statement specifies date as the time index variable, and the INTERVAL= option indicates a monthly data interval.

- **Comment 4:** The BY statement specifies that the time series are stratified according to bykey.

- **Comment 5:** The next few statements specify the variable to use, the output variables, and the required package.

- **Comment 6:** The statements between the SUBMIT and ENDSUBMIT statements contain the main logic for using Python to process the data:
  - **Comment 6a:** The PYTHON3 object is initialized.
  - **Comment 6b:** The BY-group-specific Python source code is created: First, the Python 3 statistics module is loaded. The next line of the Python source code is different for each BY group. BY group 1 calculates the mean of AIR, BY group 2 calculates the median of AIR, and BY group 3 calculates the mode of AIR. In addition to the code to calculate the average being pushed, the environment variable METHOD is set for each worker. It is set to “mean,” “median,” and “mode” for BY groups 1, 2, and 3, respectively.
  - **Comment 6c:** The next line of Python source code is the same for all BY groups—it prints the value of the METHOD environment variable.
  - **Comment 6d:** The remainder of the code is similar to the previous examples.

There are three versions of the Python code, as shown in Table 4.12.
### Table 4.12  Contents of the Python Program

<table>
<thead>
<tr>
<th>BY Group 1</th>
<th>BY Group 2</th>
<th>BY Group 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>import os</td>
<td>import os</td>
<td>import os</td>
</tr>
<tr>
<td>import statistics as st</td>
<td>import statistics as st</td>
<td>import statistics as st</td>
</tr>
<tr>
<td>AVG=round(st.mean(AIR), 2)</td>
<td>AVG=st.median(AIR)</td>
<td>AVG=st.mode(AIR)</td>
</tr>
<tr>
<td>print(os.environ['METHOD'])</td>
<td>print(os.environ['METHOD'])</td>
<td>print(os.environ['METHOD'])</td>
</tr>
</tbody>
</table>

Output 4.3.1 shows the outputs. The first table shows the values of the scalar variables. You can see how the value of `avg` is different for each BY group. The second table shows the log collector object’s values. The `_LOGTEXT_ column shows the standard output of the program, which corresponds to the type of average that is computed for each BY group. The `_LOGLEN_ column shows the length of the standard output, which corresponds to the number of characters in the `_LOGTEXT_ column plus one for the new line character added by Python’s print function.

**Output 4.3.1 Output Values**

**Scalar Variables from Example 3**

<table>
<thead>
<tr>
<th>Obs</th>
<th>bykey</th>
<th><em>STATUS</em></th>
<th>avg</th>
<th>exitStatus</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>280.299</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>0</td>
<td>265.5</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>0</td>
<td>229</td>
<td>0</td>
</tr>
</tbody>
</table>

**Log output from example 3**

<table>
<thead>
<tr>
<th>Obs</th>
<th>bykey</th>
<th><em>RUNID</em></th>
<th><em>EXITCODE</em></th>
<th><em>LOGTYPE</em></th>
<th><em>LOGLEN</em></th>
<th><em>LOGTEXT</em></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>UNNAMED RUN</td>
<td>EXECUTION</td>
<td>5</td>
<td>mean</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>UNNAMED RUN</td>
<td>EXECUTION</td>
<td>7</td>
<td>median</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>UNNAMED RUN</td>
<td>EXECUTION</td>
<td>5</td>
<td>mode</td>
<td></td>
</tr>
</tbody>
</table>
Common Pitfalls in Using the EXTLANG Package

This section outlines some issues you might encounter when you run external-language programs in your SAS scripts.

- **Case sensitivity**: As mentioned in the description of the AddVariable method, all variables are converted to uppercase before they are sent to the external-language program.

- **Shared variable size**: Unless you specify that shared variables can be resized, their size must be the same when they come back from the external-language program; otherwise, changes are not propagated back. You must account for settings within the SAS script that can alter the size of a variable. For example, specifying the LEAD=\(n\) option in the PROC TSMODEL statement will result in all array variables that are specified in outarrays being \(n\) time periods larger than the data set itself. Because the arrays are sized to include the historical and forecast regions, the external-language program must ensure that the final data of the array are sized accordingly.

- **Shared variable data type**: The EXTLANG package supports string arrays and scalars in addition to numeric arrays and scalars. NumPy arrays are also supported for Python programs. Be mindful of the data type that is returned by external-language functions you invoke. For example, many third-party Python libraries return Pandas DataFrame objects, which are not currently supported. Remember to always check the variable status flags after you call an interpreter object’s Run method to ensure that changes that are made to shared variables in the external-language program are propagated back.

- **Dynamically created variables in SAS**: A SAS program might dynamically create a variable, resulting in the wrong data being sent to an external-language program. For example, if you misspell the name of a variable you intended to pass to the AddVariable method, the external-language program will receive a dynamically created array of missing or not-a-number (NaN) values. For instance, in Example 1, replacing

  \[
  rc = \texttt{py.AddVariable(pyair,'ALIAS','air', 'READONLY', 'FALSE')}
  \]

  with

  \[
  rc = \texttt{py.AddVariable(myair,'ALIAS','air', 'READONLY', 'FALSE')}
  \]

  will not result in an error. The myair variable will be created as a numeric scalar if it has not been previously defined in the SAS program. Consequently, \(m_{avg}\) will also consist entirely of missing values.

- **Execution path assumptions**: Because your external-language code will be modified and copied to a temporary running path, it should not assume that dependencies will be available. Files that the code depends on should be specified using full path names, and the paths should be available to all workers in the cluster. Also, since only your pushed code is copied to the temporary working directory, if it depends on other modules, you must specify the absolute location of these dependencies (for example, by setting the PYTHONPATH environment variable for Python).
• Not checking outputs: Errors in external-language programs do not automatically cause your SAS program to fail. It behooves you to verify the exit code of your programs by calling the GetExitCode method after each call to the Run method. The variable status of all writable variables should also be checked via the GetVariableStatus method or by looking at the OUTVARSTATS table to ensure the variable status flag ‘UPDATED’ is set.

References

Chapter 5
Multivariate Singular Spectrum Analysis Package

Contents

Overview: MSSA Package ................................................................. 166
MSSA Package Summary ................................................................. 166
   Return Codes . ......................................................................... 166
MSSA Object .................................................................................. 167
   MSSA Synopsis .................................................................... 168
   MSSA Methods .................................................................... 169
OUTMSSA Object ........................................................................... 171
   OUTMSSA Synopsis .............................................................. 172
   OUTMSSA Methods .............................................................. 173
OUTMSV Object ............................................................................ 173
   OUTMSV Synopsis .............................................................. 174
   OUTMSV Methods .............................................................. 174
OUTMGROUPS Object ..................................................................... 175
   OUTMGROUPS Synopsis ......................................................... 176
   OUTMGROUPS Methods ......................................................... 176
OUTMWCORR Object ..................................................................... 177
   OUTMWCORR Synopsis ......................................................... 178
   OUTMWCORR Methods ......................................................... 178
Details ............................................................................................ 179
   Multivariate Singular Spectrum Analysis .................................. 179
Examples: MSSA Package ............................................................... 182
   Using CAS Sessions and CAS Engine Librefs .......................... 182
   Example 5.1: Run an Automatic Multivariate Singular Spectrum
   Analysis and Collect the Output .................................................. 183
References ....................................................................................... 190
Overview: MSSA Package

The multivariate singular spectrum analysis (MSSA) package contains a set of time series analysis functions that can be used as part of the programming statements in the TSMODEL procedure. This package provides a flexible way to analyze and decompose time series within the procedure.

**NOTE:** Each function in this chapter has a prefix of “MSSA.” However, the prefixes are omitted in descriptions for better readability. The mycas libref in the examples refers to the CAS library that is linked to a caslib. The mycas.air data table that is used in the examples refers to Sashelp.Air data. All the examples in this chapter assume that your CAS engine libref is named mycas, but you can substitute any appropriately defined CAS engine libref. For more information about CAS engine librefs, see *SAS Cloud Analytic Services: User’s Guide*.

MSSA Package Summary

Table 5.1 summarizes the objects in the MSSA package.

<table>
<thead>
<tr>
<th>Object</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MSSA</td>
<td>Perform multivariate singular spectrum analysis of a time series</td>
</tr>
<tr>
<td>OUTMGROUPS</td>
<td>Collect grouping results from an MSSA object</td>
</tr>
<tr>
<td>OUTMSSA</td>
<td>Collect output from an MSSA object</td>
</tr>
<tr>
<td>OUTMSV</td>
<td>Collect singular values from an MSSA object</td>
</tr>
<tr>
<td>OUTMWCORR</td>
<td>Collect absolute values of w-correlations from an MSSA object</td>
</tr>
</tbody>
</table>

Return Codes

Table 5.2 shows the return code (rc in method statements) status values that are used in the MSSA package. These status code values are returned after a method that is associated with an object is called; they can help determine whether the method executed successfully.
MSSA Object

Table 5.2  Return Codes

<table>
<thead>
<tr>
<th>Status</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt; 0</td>
<td>An unrecoverable error occurred. No result was produced.</td>
</tr>
<tr>
<td>= 0</td>
<td>Unconditional success. The requested action completed, and a normal result was produced.</td>
</tr>
<tr>
<td>&gt; 0</td>
<td>Conditional success or warning. A result was produced subject to conditions.</td>
</tr>
</tbody>
</table>

Upon returning a negative return code, most methods in the MSSA package objects also write a message to the output log that explains the causes of the related failure. These messages provide useful information during the process of debugging a user program. In the TSMODEL procedure, the output log is stored in the CAS table that is specified in the OUTLOG= option in the PROC TSMODEL statement. For more information about how to enable and configure logging and about how to access the output log after an invocation of the TSMODEL procedure, see Chapter 11, “The TSMODEL Procedure” (SAS Visual Forecasting: Forecasting Procedures).

MSSA Object

The MSSA object groups time series variables to be used as input for the other MSSA package objects.

Table 5.3 summarizes the methods that are associated with the MSSA object.

Table 5.3  Methods of the MSSA Object

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AddGroup</td>
<td>Add groupings manually</td>
</tr>
<tr>
<td>GetResult</td>
<td>Get the decomposition result</td>
</tr>
<tr>
<td>Initialize</td>
<td>Initialize an MSSA instance</td>
</tr>
<tr>
<td>SetOption</td>
<td>Set a named option</td>
</tr>
<tr>
<td>SetY</td>
<td>Set the input time series</td>
</tr>
<tr>
<td>Run</td>
<td>Run the multivariate singular spectrum analysis</td>
</tr>
</tbody>
</table>

Figure 5.1 diagrams the methods of the MSSA object.
MSSA Synopsis

DECLARE OBJECT obj (MSSA) ;

Method syntax, in order of typical usage:

   rc=obj.Initialize () ;
   rc=obj.SetY (YSeries) ;
   rc=obj.AddGroup (NumericArray) ;
   rc=obj.SetOption ('Name', Value) ;
   rc=obj.Run () ;
   rc=obj.GetResult (OutputArray, GroupNumber) ;
MSSA Methods

MSSA.AddGroup Method

rc = obj.AddGroup (NumericArray) ;

Adds a group manually for the MSSA instance. This method can be called multiple times. Each call adds a new group.

Input Arguments
You must specify the following input argument:

NumericArray specifies a numeric array of positive integers that indicates a group.

MSSA.GetResult Method

rc = obj.GetResult (OutputArray, GroupNumber) ;

Outputs the selected group component to an array.

Input Arguments
You must specify the following input arguments:

OutputArray specifies a dynamic array that is used to store the output group component.

GroupNumber specifies a positive integer that indicates which group to output.

MSSA.Initialize Method

rc = obj.Initialize () ;

Initializes an MSSA instance to an empty state. This method must be called before the time series arrays and other attributes are specified for the MSSA instance.

Arguments
There are no arguments associated with this method.

MSSA.Run Method

rc = obj.Run () ;

Runs the MSSA object to perform the MSSA analysis by using the dependent YSeries that has been specified for it. Upon successful completion, various results can be extracted from the MSSA object.

Arguments
There are no arguments associated with this method.
**MSSA.SetOption Method**

\[ rc = \text{obj.SetOption}('Name', \text{Value}) ; \]

Sets a named option for the MSSA instance.

**Input Arguments**
You can specify one of the following 'Names' and its associated Value:

- **'ADJUSTMEAN'**
  takes a string *Value* that specifies whether the series should be adjusted by its mean before the multivariate singular spectrum analysis is performed. You can specify one of the following *Values*:
  - TRUE | T | YES | Y  adjusts the mean before the multivariate singular spectrum analysis is performed.
  - FALSE | F | NO | N  does not adjust the mean before the multivariate singular spectrum analysis is performed.

- **'COMBINE'**
  takes a string *Value* that specifies how to calculate the combined w-correlations from the multiple w-correlations of every variables. You can specify one of the following *Values*:
  - MAX  uses the maximum value of the multiple w-correlation as the combined w-correlations.
  - MEAN  uses the mean value of the multiple w-correlation as the combined w-correlations.
  - MIN  uses the minimum value of the multiple w-correlation as the combined w-correlations.
  - RMS  uses the root mean square of the multiple w-correlation as the combined w-correlations.

- **'LENGTH'**
  takes a nonnegative integer *Value* less than the length of *YSeries* and specifies the window length to be used. The default is \( T/4 \), where \( T \) is the length of *YSeries* after the leading and ending missing values are trimmed; *YSeries* is specified in the MSSA.SetY method.

- **'METHOD'**
  takes a string *Value* that specifies the method of grouping. You can specify one of the following *Values*:
  - AUTO  uses automatic grouping.
  - GROUPS  specifies that the user select the grouping.
  - THRESHOLD  divides the MSSA components into two groups according to the cumulative percentage of their singular values.
  The default is THRESHOLD.

- **'NUMGROUPS'**
  takes a nonnegative integer *Value* less than or equal to 'LENGTH' and specifies the maximum number of groups to be retained when automatic grouping is used.
'THRESHOLDPCT' takes a numeric Value between 0 and 100 that specifies a percentage to be used to divide the MSSA components into two groups according to the cumulative percentage of their singular values. The default is 90.

'WCORRADJUSTMEAN' takes a string Value that specifies whether to adjust by its mean before w-correlations are calculated. You can specify one of the following Values:

TRUE | T | YES | Y adjusts by its mean before w-correlations are calculated.
FALSE | F | NO | N does not adjust by its mean before w-correlations are calculated.

'WCORRCUTOFF' takes a numeric Value between 0 and 100 that specifies the cutoff of w-correlations to use when the value of the 'METHOD' argument is AUTO. The default is 90.

**MSSA.SetY Method**

```
rc=obj.SetY (YSeries,YSeries<,YSeries >) ;
```

Adds multiple dependent time series arrays (YSeries) to the MSSA instance. Each run of this method overwrites the previously loaded dependent time series arrays.

**Input Arguments**

You must specify the following input argument:

YSeries specifies a numeric array that contains the dependent series for the MSSA instance.

---

**OUTMSSA Object**

The OUTMSSA object collects output from an MSSA object.

Table 5.4 summarizes the methods that are associated with the OUTMSSA object.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Collect</td>
<td>Collect the results of an MSSA object</td>
</tr>
<tr>
<td>nrows</td>
<td>Get the current row count from the OUTMSSA object</td>
</tr>
</tbody>
</table>

Figure 5.2 diagrams the methods of the OUTMSSA object.
OUTMSSA Synopsis

DECLARE OBJECT obj (OUTMSSA <('Name', Value )>) ;

Sets a named option for the MSSA instance.

Input Arguments
You can specify one of the following 'Names' and its associated Value:

'NGROUP' takes a nonnegative integer Value less than or equal to 100 and specifies the maximum number of groups that an OUTMSSA object can contain. The default is 100.

Method Syntax
In order of typical usage:

\[
rc=\text{obj.Collect}() ; \\
nrows=\text{obj.nrows}() ;
\]
OUTMSSA Methods

OUTMSSA.Collect Method

\[ rc = \text{obj}.\text{Collect}() ; \]

Collects the output of multivariate singular spectrum analysis from an MSSA object and saves the results to a CAS table.

**Arguments**
There are no arguments associated with this method.

OUTMSSA.nrows Attribute

\[ nrows = \text{obj}.\text{nrows}() ; \]

Gets the current row count from the OUTMSSA instance.

**Arguments**
There are no arguments associated with this method.

OUTMSV Object

The OUTMSV object collects output from an MSSA object.

Table 5.5 summarizes the methods that are associated with the OUTMSV object.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Collect</td>
<td>Collect the singular values of an MSSA object</td>
</tr>
<tr>
<td>nrows</td>
<td>Get the current row count from the OUTMSV object</td>
</tr>
</tbody>
</table>

Figure 5.3 diagrams the methods of the OUTMSV object.
OUTMSV Synopsis

```
DECLARE OBJECT obj (OUTMSV) ;
```

Method syntax, in order of typical usage:

```
rc=obj.Collect () ;
nrows=obj.nrows () ;
```

OUTMSV Methods

**OUTMSV.Collect Method**

```
rc=obj.Collect () ;
```

Collects the singular values from an MSSA object and saves the result to a CAS table.

**Arguments**
There are no arguments associated with this method.
OUTMSV.nrows Attribute

\[ \text{nrows} = \text{obj.nrows}() ; \]

Gets the current row count from the OUTMSV instance.

**Arguments**
There are no arguments associated with this method.

---

**OUTMGROUPS Object**

The OUTMGROUPS object collects output from an MSSA object.

Table 5.6 summarizes the methods that are associated with the OUTMGROUPS object.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Collect</td>
<td>Collect the grouping results of an MSSA object</td>
</tr>
<tr>
<td>nrows</td>
<td>Get the current row count from the OUTMGROUPS object</td>
</tr>
</tbody>
</table>

Figure 5.4 diagrams the methods of the OUTMGROUPS object.
OUTMGROUPS Synopsis

DECLARE OBJECT obj (OUTMGROUPS) ;

Method syntax, in order of typical usage:

```
rc=obj.Collect () ;
nrows=obj nrows () ;
```

OUTMGROUPS Methods

OUTMGROUPS.Collect Method

```
rc=obj.Collect () ;
```

Collects the grouping results from an MSSA object and saves the result to a CAS table.

**Arguments**

There are no arguments associated with this method.
**OUTMGROUPS.nrows Attribute**

```markdown
nrows = obj.nrows();
```

Gets the current row count from the OUTMGROUPS instance.

*Arguments*

There are no arguments associated with this method.

---

**OUTMWCORR Object**

The OUTMWCORR object collects output from an MSSA object.

Table 5.7 summarizes the methods that are associated with the OUTMWCORR object.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Collect</td>
<td>Collect the absolute values of the w-correlations of an MSSA object</td>
</tr>
<tr>
<td>nrows</td>
<td>Get the current row count from the OUTMWCORR object</td>
</tr>
</tbody>
</table>

Figure 5.5 diagrams the methods of the OUTMWCORR object.
OUTMWCORR Synopsis

DECLARE OBJECT obj (OUTMWCORR) ;

Method syntax, in order of typical usage:

rc=obj.Collect () ;
nrows=obj.nrows () ;

OUTMWCORR Methods

OUTMWCORR.Collect Method

rc=obj.Collect () ;

Collects the absolute values of the w-correlations from an MSSA object and saves the result to a CAS table.

Arguments

There are no arguments associated with this method.
OUTMWCORR.nrows Attribute

\[ \text{nrows} = \text{obj.nrows}() \]

Gets the current row count from the OUTMWCORR instance.

**Arguments**

There are no arguments associated with this method.

---

**Details**

**Multivariate Singular Spectrum Analysis**

Multivariate singular spectrum analysis (MSSA) is a technique for decomposing a time series into additive components and categorizing those components according to the magnitude of their contribution. MSSA uses a single parameter, the window length (specified in the 'LENGTH' argument in the MSSA.SetOption method), to quantify patterns in a time series without relying on prior information about the structure of the series. The window length represents the maximum lag that is considered in the analysis, and it corresponds to the dimensionality of the principal component analysis (PCA) on which MSSA is based. The components are combined into groups to categorize their roles in the MSSA decomposition.

**Main MSSA Steps**

Given a \( P \)-variate time series, \( Y_t = (y_t^{(1)}, \ldots, y_t^{(P)})', \) for \( t = 1, \ldots, T \), and a window length, \( 2 \leq L \leq T/2 \), multivariate singular spectrum analysis (Golyandina, Nekrutkin, and Zhigljavsky 2001) decomposes the time series into spectral groupings by using the following steps:

**Step 1. Embedding**

Using the time series, form a \( PK \times L \) trajectory matrix, \( X \), with elements

\[
X = \{x_{k,l}\}_{k=1, l=1}^{PK, L}
\]

such that \( x_{k+(p-1)K, l} = y_{k-l+1}^{(p)} \) for \( p = 1, \ldots, P; k = 1, \ldots, K; \) and \( l = 1, \ldots, L \), where \( K = T - L + 1 \). By definition \( L \leq K < T \), because \( 2 \leq L \leq T/2 \).

**Step 2. Decomposition**

Apply singular value decomposition to the trajectory matrix,

\[
X = UQV
\]

where \( U \) represents the \( PK \times L \) matrix that contains the left-hand-side (LHS) eigenvectors, \( Q \) represents the diagonal \( L \times L \) matrix that contains the singular values, and \( V \) represents the \( L \times L \) matrix that contains the right-hand-side (RHS) eigenvectors. Therefore,

\[
X = \sum_{l=1}^{L} X^{(l)} = \sum_{l=1}^{L} u_l q_l v_l'
\]
where \( X^{(l)} \) represents the \( PK \times L \) principal component matrix, \( u_l \) represents the \( PK \times 1 \) left-hand-side (LHS) eigenvector, \( q_l \) represents the singular value, and \( v_l \) represents the \( L \times 1 \) right-hand-side (RHS) eigenvector that is associated with the \( l \)th window index.

**Step 3. Grouping**

For each group index, \( m = 1, \ldots, M \), define a group of window indices \( I_m = \{1, \ldots, L\} \). Let the following equation represent the grouped trajectory matrix for group \( I_m \):

\[
X_{I_m} = \sum_{l \in I_m} X^{(l)} = \sum_{l \in I_m} u_l q_l v_l^T
\]

If groupings represent a spectral partition,

\[
\bigcup_{m=1}^{M} I_m = \{1, \ldots, L\} \quad \text{and} \quad I_m \cap I_n = \emptyset \quad \text{for} \quad m \neq n
\]

then according to the singular value decomposition theory,

\[
X = \sum_{m=1}^{M} X_{I_m}
\]

**Step 4. Averaging**

For each group index, \( m = 1, \ldots, M \), compute the diagonal average of \( X_{I_m} \),

\[
\tilde{x}^{(m)}_{t}(p) = \frac{1}{n_t} \sum_{l=s_t}^{e_t} x^{(m)}_{t-l+1+(p-1)K,l}
\]

where \( p \) denotes the \( p \)th variable, \( K = T - L + 1 \),

\[
s_t = 1, \quad e_t = t, \quad n_t = t \quad \text{for} \quad 1 \leq t < L
\]

\[
s_t = 1, \quad e_t = L, \quad n_t = L \quad \text{for} \quad L \leq t \leq T - L + 1
\]

\[
s_t = t - T + L, \quad e_t = L, \quad n_t = T - t + 1 \quad \text{for} \quad T - L + 1 < t \leq T
\]

If the groupings represent a spectral partition, then by definition

\[
y^{(p)}_t = \sum_{m=1}^{M} \tilde{x}^{(m)}_{t}(p)
\]

Hence, multivariate singular spectrum analysis additively decomposes the original time series, \( y^{(p)}_t \), into \( m \) component series \( \tilde{x}^{(m)}_{t}(p) \) for \( m = 1, \ldots, M \).
Computing w-Correlations

An important step in MSSA is specifying the groups $I_m \subset \{1, \ldots, L\}$ for $m = 1, \ldots, M$. In order to automate the MSSA grouping step, the weighted correlations (w-correlations) are computed separately for each of $P$ variables:

$$\rho_{i,j}(w)(p) = \frac{\left(\tilde{x}_t^{(i)}(p) \tilde{x}_t^{(j)}(p)\right)_w}{||\tilde{x}_t^{(i)}(p) \tilde{x}_t^{(j)}(p)||_w}$$

where $\left(\tilde{x}_t^{(i)}(p) \tilde{x}_t^{(j)}(p)\right)_w = \sum_{t=1}^{T} w_t \tilde{x}_t^{(i)}(p) \tilde{x}_t^{(j)}(p)$ and $w_t = \min(t, L - t + 1)$.

Specifying the Window Length

You can explicitly specify the maximum window length, $L \geq 2$, by using the 'LENGTH' argument in the MSSA.SetOption method. The window length is reduced on the basis of the time series length, $T$, to enforce the requirement that $2 \leq L \leq T/2$.

Specifying the Groups

The MSSA.AddGroup method explicitly specifies the composition and adds it to the groups, or you can use the 'THRESHOLD' argument after the 'METHOD' argument in the MSSA.SetOption method to implicitly specify the grouping. The 'THRESHOLD' argument is useful for removing noise or less dominant patterns from the time series.

Let $0 < \alpha < 100$ be the cumulative percentage singular value that is specified by the 'THRESHOLDPCT' argument. Then the last group, $I_M = \{l_\alpha, \ldots, L\}$, is determined by the smallest value such that

$$\left(\sum_{l=1}^{\alpha-1} q_l / \sum_{l=1}^{L} q_l\right) \geq \alpha \quad 1 < l_\alpha \leq L$$

Using this rule, the last group, $I_M$, describes the least dominant patterns in the time series, and the size of the last group is at least 1 and is less than the window length, $L \geq 2$.

The magnitudes of the principal components that the 'THRESHOLDPCT' argument selects are based on the singular values that appear on the diagonal of $Q$. Alternatively, each principal component’s contribution to variation in the series can be quantified by using the squares of the singular values. An OUTMSV object collects the singular values from an MSSA object.

Automatic Grouping

Besides specifying the groups explicitly, you can also specify the value of AUTO for the 'THRESHOLDPCT' argument in order to perform the automatic grouping. The group number is specified by the 'NUMGROUPS' argument in the MSSA.SetOption method. In this MSSA automatic grouping, the following steps are performed:

1. The maximal number of groups is initially assumed to be $M = L$. 
2. The groups are diagonally averaged as described previously: $\tilde{x}^{(m)(p)}_t$ for $m = 1, \ldots, L$.

3. The weighted correlations (w-correlations) between groups are computed for each of $P$ variables: $\rho^{(m)}_{i,j}(p)$, $p = 1, \ldots, P$. According to the user’s choice, the maximum, mean, root mean square, or minimum value of these multiple w-correlations is computed as the combined w-correlations, $\rho^{(m)}_{i,j}$.

4. The groups are selected on the basis of the combined w-correlations for which the absolute values are close to 1. More formally, $I_m \subset \{1, \ldots, L\}$ such that $|\rho^{(m)}_{i,j}| \approx 1$ whenever $i, j \in I_m$.

Examples: MSSA Package

Using CAS Sessions and CAS Engine Librefs

SAS Cloud Analytic Services (CAS) is the analytic server and associated cloud services in SAS Viya. This section describes how to create a CAS session and set up a CAS engine libref that you can use to connect to the CAS session. It assumes that you have a CAS server already available; contact your system administrator if you need help starting and terminating a server. This CAS server is identified by specifying the host on which it runs and the port on which it listens for communications. To simplify your interactions with this CAS server, the host information and port information for the server are stored as SAS option values that are retrieved automatically whenever this CAS server needs to be accessed. You can examine the host and port values for the server at your site by using the following statements:

```
proc options option=(CASHOST CASPORT);
run;
```

In addition to starting a CAS server, your system administrator might also have created a CAS session and a CAS engine libref for your use. You can define your own sessions and CAS engine librefs that connect to the CAS server as shown in the following statements:

```
cas mysess;
libname mycas cas sessref=mysess;
```

The CAS statement creates the CAS session named `mysess`, and the LIBNAME statement creates the `mycas` CAS engine libref that you use to connect to this session. It is not necessary to explicitly name the CASHOST and CASPORT of the CAS server in the CAS statement, because these values are retrieved from the corresponding SAS option values.

If you have created the `mysess` session, you can terminate it by using the TERMINATE option in the CAS statement as follows:

```
cas mysess terminate;
```

For more information about the CAS statement and the LIBNAME statement, see *SAS Cloud Analytic Services: User’s Guide*. For general information about CAS and CAS sessions, see *SAS Cloud Analytic Services: Fundamentals*. 
Example 5.1: Run an Automatic Multivariate Singular Spectrum Analysis and Collect the Output

This example runs a multivariate singular spectrum analysis in an MSSA object and uses an OUTMSSA object to collect the output. It extracts the first three important additive components automatically from the mycas.Usecon time series by specifying 'AUTO' in the 'METHOD' argument, the 'NUMGROUPS' argument in the MSSA.SetOption method, and 'MEAN' in the 'COMBINE' argument. Output 5.1.1 shows the singular value of each singular value component. A large singular value indicates that this singular value component captures a large portion of total variation of the original data. Output 5.1.2 shows the values of each component. Output 5.1.3 shows the resulting groupings. Output 5.1.4 shows the w-correlations that are used for autogrouping. Output 5.1.5 shows the resulting groupings together with the original data.

```proc tsmodel data=mycas.usecon outobj=(os=mycas.analytic (replace=YES)
                        osv=mycas.sv (replace=YES) osg=mycas.groups (replace=YES)
                        osw=mycas.wcorr (replace=YES));
id date interval=month;
var coal petrol;
require mssa;
submit;
declare object s(mssa);
declare object os(outmssa('NGROUP',3));
declare object osv(outmsv);
declare object osg(outmgroups);
declare object osw(outmwcorr);
rc = s.Initialize();
rc = s.SetY(coal, petrol);
rc = s.SetOption('METHOD','AUTO','COMBINE','MEAN',
                 'NUMGROUPS',3,'LENGTH',12);
rc = s.Run();
rc = os.Collect(s);
rc = osv.Collect(s);
rc = osg.Collect(s);
rc = osw.Collect(s);
endsubmit;
run;```
Output 5.1.1  Singular Values of USECON Data

Multivariate Singular Values Table

<table>
<thead>
<tr>
<th>Obs</th>
<th>SINGULARVALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3644724.5807</td>
</tr>
<tr>
<td>2</td>
<td>175473.02275</td>
</tr>
<tr>
<td>3</td>
<td>165827.26847</td>
</tr>
<tr>
<td>4</td>
<td>165354.20222</td>
</tr>
<tr>
<td>5</td>
<td>146255.13702</td>
</tr>
<tr>
<td>6</td>
<td>112996.06026</td>
</tr>
<tr>
<td>7</td>
<td>103653.52748</td>
</tr>
<tr>
<td>8</td>
<td>102329.66991</td>
</tr>
<tr>
<td>9</td>
<td>95290.185649</td>
</tr>
<tr>
<td>10</td>
<td>91561.742846</td>
</tr>
<tr>
<td>11</td>
<td>90248.528884</td>
</tr>
<tr>
<td>12</td>
<td>46000.147106</td>
</tr>
</tbody>
</table>
Example 5.1: Run an Automatic Multivariate Singular Spectrum Analysis and Collect the Output F 185

Output 5.1.2 MSSA Results of USECON Data

M S S A R e s u lt T a b le
O b s

_ V A R ID _

D A T E

O R IG IN A L

G R O U P S U M

G R O U P 1

G R O U P 2

G R O U P 3

1

1

_ N A M E _
C O A L

J A N 1 9 7 1

4 9 7 8 0

5 3 8 4 6 .3 0 6 9 0 1

4 5 4 3 3 .7 4 2 0 3 8

8 7 6 9 .1 7 1 7 1 2 6

-3 5 6 .6 0 6 8 4 9 9

2

1

C O A L

F E B 1 9 7 1

4 7 0 2 9

5 2 4 7 0 .8 8 8 6 8 9

4 5 4 9 4 .3 9 6 0 3

9 4 4 0 .4 6 1 6 4 0 3

-2 4 6 3 .9 6 8 9 8 2

3

1

C O A L

M A R 1 9 7 1

5 6 9 2 0

4 9 8 9 5 .8 1 6 7 1 9

4 5 6 1 2 .2 7 2 0 2 6

8 0 1 3 .0 7 0 8 8 6

-3 7 2 9 .5 2 6 1 9 3

4

1

C O A L

A P R 1 9 7 1

5 4 3 3 6

4 8 5 0 6 .7 4 7 8 7 7

4 5 6 4 3 .9 6 3 6 3 2

5 1 4 5 .6 0 0 8 5 8 2

-2 2 8 2 .8 1 6 6 1 3

5

1

C O A L

M A Y 1 9 7 1

5 0 4 4 2

4 9 8 7 8 .1 8 4 2 2 6

4 5 6 0 6 .3 4 6 1 3 9

2 2 5 0 .3 7 0 6 7 3 2

2 0 2 1 .4 6 7 4 1 4 3

6

1

C O A L

J U N 1 9 7 1

4 9 2 9 8

5 2 4 5 2 .8 9 5 6 7 8

4 5 6 2 8 .6 5 1 8 1

3 3 9 .6 2 1 1 9 3 2 9

6 4 8 4 .6 2 2 6 7 5 1

7

1

C O A L

J U L 1 9 7 1

3 9 5 3 7

5 2 0 6 5 .4 1 3 1 7 2

4 5 6 6 6 .7 0 8 0 1 6

-9 2 4 .7 4 7 8 4 6 7

7 3 2 3 .4 5 3 0 0 2 9

8

1

C O A L

A U G 1 9 7 1

5 6 1 8 5

4 7 0 4 2 .0 6 6 6 0 1

4 5 7 2 4 .6 8 9 5 1 1

-1 8 4 6 .1 9 5 7 6

3 1 6 3 .5 7 2 8 4 9 8

9

1

C O A L

S E P 1 9 7 1

5 4 4 4 9

3 9 0 0 9 .9 7 3 6 9 9

4 5 7 4 4 .1 3 7 8 6 7

-3 0 6 4 .1 8 3 6 2 4

-3 6 6 9 .9 8 0 5 4 4

1 0

1

C O A L

O C T 1 9 7 1

1 1 8 5 7

3 3 2 9 2 .6 9 8 9 3 7

4 5 7 3 0 .5 4 9 6 9 9

-4 0 3 4 .9 5 8 9 1 6

-8 4 0 2 .8 9 1 8 4 6

1 1

1

C O A L

N O V 1 9 7 1

2 6 3 2 7

3 4 4 3 0 .7 4 0 7 3 1

4 6 0 2 7 .9 7 6 3 1

-3 8 7 5 .2 4 1 5 1 5

-7 7 2 1 .9 9 4 0 6 4

1 2

1

C O A L

D E C 1 9 7 1

5 6 0 3 2

4 1 1 1 8 .7 0 4 8 7

4 6 4 4 7 .4 6 0 0 3 5

-2 7 4 4 .8 2 4 1 1 9

-2 5 8 3 .9 3 1 0 4 7

1 3

1

C O A L

J A N 1 9 7 2

4 9 6 8 0

4 8 8 9 1 .1 9 5 4 9 9

4 6 7 4 7 .7 2 8 5 3 5

-1 1 3 7 .8 0 0 7 8 7

3 2 8 1 .2 6 7 7 5 1 3

1 4

1

C O A L

F E B 1 9 7 2

4 9 1 1 2

5 3 7 5 3 .2 7 5 8 8 7

4 7 0 4 6 .2 4 9 6 4 9

2 1 2 .8 0 6 0 6 0 6 5

6 4 9 4 .2 2 0 1 7 7 1

1 5

1

C O A L

M A R 1 9 7 2

5 4 4 3 8

5 3 9 4 7 .0 5 3 3 9 9

4 7 3 0 7 .8 6 6 5 1

1 1 2 5 .0 6 0 2 2 7 9

5 5 1 4 .1 2 6 6 6 0 4

1 6

1

C O A L

A P R 1 9 7 2

4 9 8 1 4

5 0 9 9 2 .4 2 0 5 4 8

4 7 5 5 9 .3 3 0 7 2 8

1 6 3 6 .3 8 4 7 4 2

1 7 9 6 .7 0 5 0 7 8 1

1 7

1

C O A L

M A Y 1 9 7 2

5 2 8 7 9

4 8 0 2 9 .1 2 1 9 8 6

4 7 8 2 2 .4 0 2 7 9 2

1 9 8 4 .1 7 5 3 7 8 7

-1 7 7 7 .4 5 6 1 8 5

1 8

1

C O A L

J U N 1 9 7 2

5 0 0 8 3

4 6 8 6 0 .6 6 7 0 6 8

4 8 0 5 8 .8 3 4 5 7 8

2 0 7 9 .8 1 9 0 4 7 5

-3 2 7 7 .9 8 6 5 5 7

1 9

1

C O A L

J U L 1 9 7 2

4 0 9 6 4

4 7 5 1 9 .1 0 1 6 8 9

4 8 2 6 5 .9 2 7 2 9 9

1 7 8 5 .6 3 1 4 7 1 1

-2 5 3 2 .4 5 7 0 8 1

2 0

1

C O A L

A U G 1 9 7 2

5 2 1 6 9

4 9 2 0 0 .7 4 2 4 9 8

4 8 4 8 3 .0 2 7 3 3 1

1 2 8 6 .4 1 7 7 6 7 1

-5 6 8 .7 0 2 6

2 1

1

C O A L

S E P 1 9 7 2

4 9 3 7 4

5 0 4 3 8 .8 1 5 5 4 3

4 8 7 5 4 .1 2 0 0 1 2

5 9 1 .7 1 7 2 9 9 0 9

1 0 9 2 .9 7 8 2 3 1 8

2 2

1

C O A L

O C T 1 9 7 2

5 1 6 7 1

5 0 6 4 1 .5 4 1 0 3 8

4 9 0 5 3 .2 1 2 7 1 5

7 .9 2 4 2 9 1 4 8 7 4

1 5 8 0 .4 0 4 0 3 1 3

2 3

1

C O A L

N O V 1 9 7 2

5 0 2 9 7

4 9 5 3 9 .0 7 2 4 0 9

4 9 0 9 4 .2 9 8 0 8

-3 9 2 .7 1 2 1 7 5

8 3 7 .4 8 6 5 0 4 0 7

2 4

1

C O A L

D E C 1 9 7 2

4 4 9 0 4

4 7 9 0 0 .7 5 1 6 1 2

4 8 9 6 5 .5 4 5 5 6

-6 8 8 .5 6 2 2 4 2 2

-3 7 6 .2 3 1 7 0 5 6

2 5

1

C O A L

J A N 1 9 7 3

4 9 3 7 9

4 7 2 1 8 .5 0 0 8 2

4 8 9 4 0 .2 0 6 2 0 2

-7 9 6 .3 3 5 9 2 7 5

-9 2 5 .3 6 9 4 5 4 9

2 6

1

C O A L

F E B 1 9 7 3

4 5 8 9 3

4 7 7 7 8 .6 0 8 1 6 8

4 8 9 4 5 .7 1 1 8 4

-7 1 7 .3 1 3 6 2 3 1

-4 4 9 .7 9 0 0 4 9 3

2 7

1

C O A L

M A R 1 9 7 3

5 0 5 4 7

4 8 8 2 6 .5 9 6 5 1 3

4 9 0 0 1 .1 1 8 9 1 9

-5 2 8 .6 6 5 6 9 3 9

3 5 4 .1 4 3 2 8 8 0 3

2 8

1

C O A L

A P R 1 9 7 3

4 6 9 9 9

4 9 0 5 6 .9 9 3 6 6

4 9 0 8 6 .8 6 1 5 2 3

-5 1 5 .1 2 5 2 2 8 5

4 8 5 .2 5 7 3 6 5 8 4

2 9

1

C O A L

M A Y 1 9 7 3

5 1 4 2 0

4 8 4 7 8 .1 7 1 1

4 9 2 4 2 .0 4 0 5 3

-5 9 0 .1 8 5 3 7 3 3

-1 7 3 .6 8 4 0 5 6 8

3 0

1

C O A L

J U N 1 9 7 3

4 6 6 1 3

4 7 8 6 1 .0 1 4 6 7 9

4 9 4 4 9 .2 5 4 8 7 1

-6 8 6 .7 2 6 7 5 1

-9 0 1 .5 1 3 4 4 0 9

3 1

1

C O A L

J U L 1 9 7 3

4 3 8 0 1

4 8 3 3 6 .7 2 5 6 5 7

4 9 6 8 9 .3 7 1 5 9 9

-5 6 8 .5 8 7 2 5 9 7

-7 8 4 .0 5 8 6 8 2 1

3 2

1

C O A L

A U G 1 9 7 3

5 5 8 7 4

5 0 0 1 4 .5 2 3 9 8 7

4 9 9 4 7 .2 9 8 5 5 2

-1 6 5 .6 6 9 3 9 7

2 3 2 .8 9 4 8 3 2 1 2

3 3

1

C O A L

S E P 1 9 7 3

4 8 3 3 8

5 1 4 9 9 .5 6 9 8 5 4

5 0 1 4 9 .8 0 0 3 6 4

1 7 6 .4 5 4 2 1 5 0 9

1 1 7 3 .3 1 5 2 7 5 5

3 4

1

C O A L

O C T 1 9 7 3

5 4 3 8 0

5 1 9 8 5 .6 7 9 6 8 8

5 0 3 8 8 .1 7 3 0 5 9

4 1 9 .7 3 6 2 3 8 3 7

1 1 7 7 .7 7 0 3 9 0 8

3 5

1

C O A L

N O V 1 9 7 3

4 9 8 2 5

5 1 1 0 8 .1 5 9 0 2 5

5 0 6 4 8 .7 8 3 2 0 6

3 6 0 .7 4 8 8 8 0 6 7

9 8 .6 2 6 9 3 8 7 6 6

3 6

1

C O A L

D E C 1 9 7 3

4 8 6 6 8

5 0 1 0 2 .6 5 1 9 0 4

5 0 7 9 9 .5 0 7 8 3 7

3 4 2 .1 6 2 5 2 4 9 6

-1 0 3 9 .0 1 8 4 5 8

3 7

1

C O A L

J A N 1 9 7 4

5 3 5 3 0

5 0 3 0 0 .8 6 2 6 3

5 0 8 6 3 .6 5 0 5 3 6

6 0 7 .4 4 7 1 6 9 0 6

-1 1 7 0 .2 3 5 0 7 5

3 8

1

C O A L

F E B 1 9 7 4

4 9 8 5 1

5 1 6 2 0 .8 5 5 0 0 5

5 0 9 1 0 .4 0 0 7 3 8

1 0 2 9 .9 3 7 8 1 7 8

-3 1 9 .4 8 3 5 5 0 4

3 9

1

C O A L

M A R 1 9 7 4

5 1 0 2 7

5 2 9 1 6 .2 1 6 2 2 4

5 0 9 4 3 .4 4 7 2 0 8

1 4 0 8 .1 5 4 2 2 5 3

5 6 4 .6 1 4 7 8 9 9 2

4 0

1

C O A L

A P R 1 9 7 4

5 4 1 8 1

5 2 8 9 9 .4 5 9 6 3 4

5 0 9 8 4 .0 9 9 8 2 1

1 3 6 0 .9 1 5 3 5 4 5

5 5 4 .4 4 4 4 5 8 2 5


Output 5.1.3  MSSA Grouping Results of USECON Data

MSSA Grouping Result Table

<table>
<thead>
<tr>
<th>Obs</th>
<th>GROUPINDEX</th>
<th>WINDOWINDEX</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>
### Output 5.1.4 W-Correlations of USECON Data

**W-Correlations**

<table>
<thead>
<tr>
<th>Obs</th>
<th>WINDOWINDEX1</th>
<th>WINDOWINDEX2</th>
<th>WCORRELATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>1</td>
<td>0.041157187</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>1</td>
<td>0.001744158</td>
</tr>
<tr>
<td>5</td>
<td>3</td>
<td>2</td>
<td>0.2855078739</td>
</tr>
<tr>
<td>6</td>
<td>3</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>4</td>
<td>1</td>
<td>0.002844942</td>
</tr>
<tr>
<td>8</td>
<td>4</td>
<td>2</td>
<td>0.4123412264</td>
</tr>
<tr>
<td>9</td>
<td>4</td>
<td>3</td>
<td>0.9687978426</td>
</tr>
<tr>
<td>10</td>
<td>4</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>11</td>
<td>5</td>
<td>1</td>
<td>0.004164991</td>
</tr>
<tr>
<td>12</td>
<td>5</td>
<td>2</td>
<td>0.613523796</td>
</tr>
<tr>
<td>13</td>
<td>5</td>
<td>3</td>
<td>0.5988679266</td>
</tr>
<tr>
<td>14</td>
<td>5</td>
<td>4</td>
<td>0.6413749647</td>
</tr>
<tr>
<td>15</td>
<td>5</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>16</td>
<td>6</td>
<td>1</td>
<td>0.0156023585</td>
</tr>
<tr>
<td>17</td>
<td>6</td>
<td>2</td>
<td>0.1359003305</td>
</tr>
<tr>
<td>18</td>
<td>6</td>
<td>3</td>
<td>0.2134272082</td>
</tr>
<tr>
<td>19</td>
<td>6</td>
<td>4</td>
<td>0.2053113498</td>
</tr>
<tr>
<td>20</td>
<td>6</td>
<td>5</td>
<td>0.3360532565</td>
</tr>
<tr>
<td>21</td>
<td>6</td>
<td>6</td>
<td>1</td>
</tr>
<tr>
<td>22</td>
<td>7</td>
<td>1</td>
<td>0.0002100483</td>
</tr>
<tr>
<td>23</td>
<td>7</td>
<td>2</td>
<td>0.0061633848</td>
</tr>
<tr>
<td>24</td>
<td>7</td>
<td>3</td>
<td>0.0047968554</td>
</tr>
<tr>
<td>25</td>
<td>7</td>
<td>4</td>
<td>0.002755553</td>
</tr>
<tr>
<td>26</td>
<td>7</td>
<td>5</td>
<td>0.0118060393</td>
</tr>
<tr>
<td>27</td>
<td>7</td>
<td>6</td>
<td>0.4428822558</td>
</tr>
<tr>
<td>28</td>
<td>7</td>
<td>7</td>
<td>1</td>
</tr>
<tr>
<td>29</td>
<td>8</td>
<td>1</td>
<td>0.0041576553</td>
</tr>
<tr>
<td>30</td>
<td>8</td>
<td>2</td>
<td>0.0464131358</td>
</tr>
<tr>
<td>31</td>
<td>8</td>
<td>3</td>
<td>0.047054625</td>
</tr>
<tr>
<td>32</td>
<td>8</td>
<td>4</td>
<td>0.0447636033</td>
</tr>
<tr>
<td>33</td>
<td>8</td>
<td>5</td>
<td>0.1033255948</td>
</tr>
<tr>
<td>34</td>
<td>8</td>
<td>6</td>
<td>0.5852174367</td>
</tr>
<tr>
<td>35</td>
<td>8</td>
<td>7</td>
<td>0.6555004783</td>
</tr>
<tr>
<td>36</td>
<td>8</td>
<td>8</td>
<td>1</td>
</tr>
<tr>
<td>37</td>
<td>9</td>
<td>1</td>
<td>0.0005465487</td>
</tr>
<tr>
<td>38</td>
<td>9</td>
<td>2</td>
<td>0.0411587384</td>
</tr>
<tr>
<td>39</td>
<td>9</td>
<td>3</td>
<td>0.0313577521</td>
</tr>
<tr>
<td>40</td>
<td>9</td>
<td>4</td>
<td>0.0316409506</td>
</tr>
<tr>
<td>41</td>
<td>9</td>
<td>5</td>
<td>0.1219081602</td>
</tr>
<tr>
<td>42</td>
<td>9</td>
<td>6</td>
<td>0.5629249224</td>
</tr>
<tr>
<td>43</td>
<td>9</td>
<td>7</td>
<td>0.2514203112</td>
</tr>
<tr>
<td>44</td>
<td>9</td>
<td>8</td>
<td>0.7560323278</td>
</tr>
<tr>
<td>45</td>
<td>9</td>
<td>9</td>
<td>1</td>
</tr>
<tr>
<td>46</td>
<td>10</td>
<td>1</td>
<td>0.0007028068</td>
</tr>
<tr>
<td>47</td>
<td>10</td>
<td>2</td>
<td>0.0250848571</td>
</tr>
</tbody>
</table>
Output 5.1.4 continued

W-Correlations

<table>
<thead>
<tr>
<th>Obs</th>
<th>WINDOWINDEX1</th>
<th>WINDOWINDEX2</th>
<th>WCORRELATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>48</td>
<td>10</td>
<td>3</td>
<td>0.0189514167</td>
</tr>
<tr>
<td>49</td>
<td>10</td>
<td>4</td>
<td>0.0191684083</td>
</tr>
<tr>
<td>50</td>
<td>10</td>
<td>5</td>
<td>0.0654885974</td>
</tr>
<tr>
<td>51</td>
<td>10</td>
<td>6</td>
<td>0.3309037038</td>
</tr>
<tr>
<td>52</td>
<td>10</td>
<td>7</td>
<td>0.0871569467</td>
</tr>
<tr>
<td>53</td>
<td>10</td>
<td>8</td>
<td>0.4784545974</td>
</tr>
<tr>
<td>54</td>
<td>10</td>
<td>9</td>
<td>0.7575758074</td>
</tr>
<tr>
<td>55</td>
<td>10</td>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>56</td>
<td>11</td>
<td>1</td>
<td>0.0001080049</td>
</tr>
<tr>
<td>57</td>
<td>11</td>
<td>2</td>
<td>0.0057788375</td>
</tr>
<tr>
<td>58</td>
<td>11</td>
<td>3</td>
<td>0.008452176</td>
</tr>
<tr>
<td>59</td>
<td>11</td>
<td>4</td>
<td>0.0059815373</td>
</tr>
<tr>
<td>60</td>
<td>11</td>
<td>5</td>
<td>0.0399871407</td>
</tr>
<tr>
<td>61</td>
<td>11</td>
<td>6</td>
<td>0.2833975132</td>
</tr>
<tr>
<td>62</td>
<td>11</td>
<td>7</td>
<td>0.1843227865</td>
</tr>
<tr>
<td>63</td>
<td>11</td>
<td>8</td>
<td>0.5592000992</td>
</tr>
<tr>
<td>64</td>
<td>11</td>
<td>9</td>
<td>0.7323916605</td>
</tr>
<tr>
<td>65</td>
<td>11</td>
<td>10</td>
<td>0.9593463434</td>
</tr>
<tr>
<td>66</td>
<td>11</td>
<td>11</td>
<td>1</td>
</tr>
<tr>
<td>67</td>
<td>12</td>
<td>1</td>
<td>0.0012035352</td>
</tr>
<tr>
<td>68</td>
<td>12</td>
<td>2</td>
<td>0.0102994119</td>
</tr>
<tr>
<td>69</td>
<td>12</td>
<td>3</td>
<td>0.0120312844</td>
</tr>
<tr>
<td>70</td>
<td>12</td>
<td>4</td>
<td>0.0120056719</td>
</tr>
<tr>
<td>71</td>
<td>12</td>
<td>5</td>
<td>0.0161476133</td>
</tr>
<tr>
<td>72</td>
<td>12</td>
<td>6</td>
<td>0.0404754795</td>
</tr>
<tr>
<td>73</td>
<td>12</td>
<td>7</td>
<td>0.1393484251</td>
</tr>
<tr>
<td>74</td>
<td>12</td>
<td>8</td>
<td>0.0612492493</td>
</tr>
<tr>
<td>75</td>
<td>12</td>
<td>9</td>
<td>0.1055181719</td>
</tr>
<tr>
<td>76</td>
<td>12</td>
<td>10</td>
<td>0.0421279717</td>
</tr>
<tr>
<td>77</td>
<td>12</td>
<td>11</td>
<td>0.0606242811</td>
</tr>
<tr>
<td>78</td>
<td>12</td>
<td>12</td>
<td>1</td>
</tr>
</tbody>
</table>
Example 5.1: Run an Automatic Multivariate Singular Spectrum Analysis and Collect the Output

Output 5.1.5 MSSA Results of USECON Data

MSSA Result Plot of Coal

- Original
- Group 1
- Group 2
- Group 3
References


Chapter 6
Simple Forecast Service Package

Contents

Overview: SFS Package .......................................................... 191
SFS Package Summary ......................................................... 192
Common Argument Types ...................................................... 192
Return Codes ........................................................................ 192
SFS Object .......................................................................... 193
SFS Synopsis ....................................................................... 194
SFS Methods ....................................................................... 195
Examples: SFS Package ......................................................... 198
Using CAS Sessions and CAS Engine Librefs ....................... 199
Example 6.1: Simple Forecasting .......................................... 199
Example 6.2: Automatic Forecasting Using Predictor Series ... 203
Example 6.3: Using SFS with PROC CAS ............................... 207

Overview: SFS Package

This chapter describes the simple forecast service (SFS) package that can be used with the TSMODEL procedure. The SFS package provides a simple-to-use interface for automatically forecasting services for univariate time series. The SFS package implements a set of best-practice rules to define its behavior and uses the same underlying forecasting software as SAS Forecast Server and SAS Visual Analytics time series forecasting use. The package is called “simple” because only a few options are needed to control its behavior. More sophisticated time series forecasting services are available in other time series packages that are described in this book. If you want to exercise more control over the forecasting process, use one of those packages.

The SFS package is object-oriented (OO). To use the SFS package, you must declare instances of the object classes that are contained in the package. Declaring an object instance is the OO equivalent of declaring a program variable. As with simple program variables, the declaration assigns the instance a name of your choosing and a type, which is defined by the object’s class. Unlike simple program variables, the object instance requires a different syntax for interacting with it and offers different functions (methods) that are contextual to the object. Object instances hold information (data and results) over the lifetime of the instance. From a programming perspective, this property makes them very different from a function call, which generally is idempotent (a function operates on inputs and produces outputs that have no carryover effects from one call to the next). The object can offer very sophisticated capabilities with a simple-to-use interface.
SFS Package Summary

Table 6.1 summarizes the single object class in the SFS package.

<table>
<thead>
<tr>
<th>Object</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SFS</td>
<td>Automatically model and forecast univariate time series.</td>
</tr>
</tbody>
</table>

Common Argument Types

Table 6.2 defines the common argument types that are used in this chapter. The symbol $x$ corresponds to the variable name.

<table>
<thead>
<tr>
<th>SAS Data Type</th>
<th>Declaration Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>String</td>
<td>LENGTH $x$ $n;</td>
</tr>
<tr>
<td>Numeric</td>
<td>$x$ or LENGTH $x$ @;</td>
</tr>
<tr>
<td>Numeric array</td>
<td>ARRAY $x[n]/$NOSYMBOLS;</td>
</tr>
<tr>
<td>Status</td>
<td>$x$ or LENGTH $x$ @;</td>
</tr>
</tbody>
</table>

Return Codes

Table 6.3 shows the return code (rc in method statements) status values that are used in this package. These status code values are returned after a method that is associated with an object is called; they can help determine whether the method executed successfully.

<table>
<thead>
<tr>
<th>Status</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt; 0</td>
<td>An unrecoverable error occurred. No result was produced.</td>
</tr>
<tr>
<td>= 0</td>
<td>Unconditional success. The requested action completed and a normal result was produced.</td>
</tr>
<tr>
<td>&gt; 0</td>
<td>Conditional success or warning. A result was produced subject to conditions.</td>
</tr>
</tbody>
</table>

Upon returning a negative status code, most methods in the SFS package objects also write a message to the output log that explains the causes of the related failure. These messages provide useful information during
the process of debugging a user program. In the TSMODEL procedure, the output log is stored in the CAS table that is specified in the OUTLOG= option in the PROC TSMODEL statement. For more information about how to enable and configure logging and about how to access the output log after an invocation of the TSMODEL procedure, see Chapter 11, “The TSMODEL Procedure” (SAS Visual Forecasting: Forecasting Procedures).

SFS Object

The SFS object automatically models and forecasts univariate time series. SFS capabilities range from automatic exponential smoothing (auto-ESM) to automatic model generation and automatic forecasting. The behavior of the SFS instance is dynamic depending on whether independent variables (predictors) are included for consideration. When only a dependent variable (a Y series) is included, the SFS instance generates an auto-ESM forecast by selecting from among the possible ESM methods as a function of whether the Y series is seasonal or not. When independent variables are included, automatic model generation is performed by the use of two ARIMAX (autoregressive integrated moving average with explanatory variable) identification techniques, in addition to an auto-ESM model. As many as three candidate models might be generated. The candidate models are then evaluated for their in-sample Bayesian information criterion (SBC) fit statistic, and the best performing model is selected to forecast the dependent variable. If the selected model uses predictors, those predictor series are lead-extended by using auto-ESM methods to forecast their future values for use in generating the forecast of the dependent series.

Table 6.4 summarizes the methods that are associated with the SFS object.

Table 6.4 Methods of the SFS Object

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initialize</td>
<td>Initialize or reset the SFS object for use</td>
</tr>
<tr>
<td>SetY</td>
<td>Specify the dependent time series</td>
</tr>
<tr>
<td>AddX</td>
<td>Add an independent time series</td>
</tr>
<tr>
<td>SetOption</td>
<td>Specify a computational option</td>
</tr>
<tr>
<td>Run</td>
<td>Automatically model and forecast the dependent variable</td>
</tr>
<tr>
<td>GetForecast</td>
<td>Retrieve a computed forecast series by name</td>
</tr>
<tr>
<td>nfor</td>
<td>Get the length (observation count) of the forecast series</td>
</tr>
<tr>
<td>criterion</td>
<td>Get the in-sample Bayesian information criterion (SBC) fit statistic</td>
</tr>
<tr>
<td>model</td>
<td>Get the short name of the selected model that is used to produce the final forecast</td>
</tr>
</tbody>
</table>
SFS Synopsis

DECLARE OBJECT obj (SFS) ;

Each SFS object in your program is independently instantiated for each BY-group thread. Also, each SFS object in your program is automatically reset at the start and end of each BY group.

Method syntax, in order of typical usage:

\[
\begin{align*}
rc &= \text{obj}.\text{Initialize} () ; \\
rc &= \text{obj}.\text{SetY} (\text{YSeries}) ; \\
rc &= \text{obj}.\text{AddX} (\text{XSeries}) ; \\
rc &= \text{obj}.\text{SetOption} (\text{\textquote{Name}}, \text{Value}) ; \\
rc &= \text{obj}.\text{Run} () ; \\
rc &= \text{obj}.\text{GetForecast} (\text{Which}, \text{Result}) ; \\
rc &= \text{obj}.\text{nfor} () ; \\
rc &= \text{obj}.\text{criterion} () ; \\
rc &= \text{obj}.\text{model} () ;
\end{align*}
\]

Figure 6.1 outlines the programmatic data flow through the SFS object; each arrow represents a different object method.

The SFS object has the following calling sequence protocol:

1. You must call the SFS.Initialize method before calling any other SFS methods.

2. You must call the SFS.SetY method before calling the SFS.Run method. Each call to the SFS.SetY method before calling the SFS.Run method simply replaces the dependent series.
3. You can call the SFS.AddX method zero or more times prior to SFS.Run method. Each SFS.AddX method call includes the specified independent series to be considered during model generation. Independent variables are tracked by the array name that you specify in the SFS.AddX method call. Calling the SFS.AddX method repeatedly with the same array name replaces the variable’s series with the values from the most recent call.

4. You can call the SFS.SetOption method prior to the SFS.Run method to change mutable properties of the SFS instance. Properties in effect at the time of the SFS.Run method call are used.

5. After a successful SFS.Run method call, you can call the SFS.GetForecast method to retrieve the forecast series. Calls to the SFS.GetForecast method before calling the SFS.Run method or calls made after an unsuccessful SFS.Run method return missing values to your Result array.

6. After a successful SFS.Run method call, you can call the SFS.nfor, SFS.criterion, or SFS.model methods to retrieve their respective values. Calls to these methods before calling the SFS.Run method or calls made after an unsuccessful SFS.Run method return missing values.

---

SFS Methods

SFS.AddX Method

```
rc=obj.AddX (XSeries) ;
```

Adds an independent time series array (XSeries) for the SFS instance. Each call to the AddX method adds the specified independent variable to the SFS instance. This method can be called as many times as needed to specify all the independent variables that are needed for the time series model that is used to initialize the SFS instance.

**Input Arguments**

You must specify the following input argument:

- **XSeries** specifies a numeric array that contains an independent series for the SFS instance.

SFS.criterion Method

```
rc=obj.criterion () ;
```

Returns the fit statistic value for the final forecast for the SFS instance. The criterion is the in-sample Bayesian information criterion (SBC) fit statistic for the selected time series model. A missing value indicates that the SFS instance has not produced a successful forecast.

**Arguments**

There are no arguments associated with this method.
**SFS.GetForecast Method**

```c
rc = obj.GetForecast (Which, Result);
```

Gets the specified forecast series (`Which`) from the SFS instance and stores it into the specified numeric array (`Result`).

**Input Arguments**
You must specify the following input argument:

- `Which` is a case-insensitive character string that specifies the forecast series to return. You can specify one of the following values:
  - **ERROR** returns prediction errors.
  - **LOWER** returns a lower confidence limit series.
  - **STDERR** returns a prediction standard error series.
  - **PREDICT** returns a prediction series.
  - **UPPER** returns an upper confidence limit series.

**Output Arguments**
You must specify the following output argument:

- `Result` specifies a numeric array to receive the forecast series. If the array length is longer than the forecast series, the forecast series is padded with missing values.

**SFS.Initialize Method**

```c
rc = obj.Initialize();
```

Initializes or resets the SFS instance.

**Arguments**
There are no arguments associated with this method.

**SFS.model Method**

```c
rc = obj.model();
```

Returns the short name of the selected model that is used to produce the final forecast. A missing value (null string) indicates that the SFS instance has not produced a successful forecast.

**Table 6.5** shows the model families that can be considered during the process.
### Table 6.5  Model Families for the SFS.model Method

<table>
<thead>
<tr>
<th>Family</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARIMAX</td>
<td>ARIMA model that is generated by finding an ARIMA model for the error series first and then choosing significant inputs and events (ARIMA-REG order)</td>
</tr>
<tr>
<td>ESM</td>
<td>Exponential smoothing model</td>
</tr>
<tr>
<td>REGARIMA</td>
<td>ARIMA model that is generated by finding a regression model first and then deciding the autoregressive (AR) and moving average (MA) polynomial orders (REG-ARIMA order)</td>
</tr>
</tbody>
</table>

**Arguments**

There are no arguments associated with this method.

### SFS.nfor Method

\[ rc = \text{obj}.nfor(); \]

Returns the length (observation count) of the forecast series for the SFS instance. A missing value indicates that the SFS object has not produced a successful forecast.

**Arguments**

There are no arguments associated with this method.

### SFS.Run Method

\[ rc = \text{obj}.Run(); \]

Runs the SFS instance to automatically model and forecast the dependent variable. If any independent variables are specified, they are considered during model generation and are included in the candidate models if they affect forecasting the behavior of the dependent variable. Upon successful completion of this method, various results can be extracted from the SFS instance.

Table 6.6 shows the model families that can be considered during the process.

### Table 6.6  Model Families for the SFS.Run Method

<table>
<thead>
<tr>
<th>Family</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARIMAX</td>
<td>ARIMA model that is generated by finding an ARIMA model for the error series first and then choosing significant inputs and events (ARIMA-REG order)</td>
</tr>
<tr>
<td>ESM</td>
<td>Exponential smoothing model</td>
</tr>
<tr>
<td>REGARIMA</td>
<td>ARIMA model that is generated by finding a regression model first and then deciding the autoregressive (AR) and moving average (MA) polynomial orders (REG-ARIMA order)</td>
</tr>
</tbody>
</table>
Automatic model selection and forecasting follows by considering the performance of the candidate models based on their in-sample performance as measured by their Bayesian information criterion (SBC) values. ARIMAX and REGARIMA candidate models are generated only when independent variables are included in the analysis.

**Arguments**
There are no arguments associated with this method.

**SFS.SetOption Method**

```rc = obj.SetOption('Name', Value);
```

Specifies a named option for the SFS instance.

**Input Arguments**
You must specify the following `Name` and its associated `Value`:

- `'ALPHA'` takes a numeric `Value` between 0 and 1, exclusive, that specifies the significance level for forecast confidence bands. The default value is 0.05.

**SFS.SetY Method**

```rc = obj.SetY(YSeries);
```

Adds a dependent time series array (YSeries) for the SFS instance.

**Input Arguments**
You must specify the following input argument:

- `YSeries` specifies a numeric array that contains a dependent series for the SFS instance.

**Examples: SFS Package**

Throughout this section, it is assumed that you have already started a CAS session and that the data tables that are used in this section are in mycas, a CAS library that you have necessary permissions to work with. This section assumes that you are familiar with the general workings of the TSMODEL procedure; for more information, see Chapter 11, “The TSMODEL Procedure” (*SAS Visual Forecasting: Forecasting Procedures*).
Using CAS Sessions and CAS Engine Librefs

SAS Cloud Analytic Services (CAS) is the analytic server and associated cloud services in SAS Viya. This section describes how to create a CAS session and set up a CAS engine libref that you can use to connect to the CAS session. It assumes that you have a CAS server already available; contact your system administrator if you need help starting and terminating a server. This CAS server is identified by specifying the host on which it runs and the port on which it listens for communications. To simplify your interactions with this CAS server, the host information and port information for the server are stored as SAS option values that are retrieved automatically whenever this CAS server needs to be accessed. You can examine the host and port values for the server at your site by using the following statements:

```sas
proc options option=(CASHOST CASPORT);
run;
```

In addition to starting a CAS server, your system administrator might also have created a CAS session and a CAS engine libref for your use. You can define your own sessions and CAS engine librefs that connect to the CAS server as shown in the following statements:

```sas
cas mysess;
libname mycas cas sessref=mysess;
```

The CAS statement creates the CAS session named `mysess`, and the LIBNAME statement creates the `mycas` CAS engine libref that you use to connect to this session. It is not necessary to explicitly name the CASHOST and CASPORT of the CAS server in the CAS statement, because these values are retrieved from the corresponding SAS option values.

If you have created the `mysess` session, you can terminate it by using the TERMINATE option in the CAS statement as follows:

```sas
cas mysess terminate;
```

For more information about the CAS statement and the LIBNAME statement, see *SAS Cloud Analytic Services: User’s Guide*. For general information about CAS and CAS sessions, see *SAS Cloud Analytic Services: Fundamentals*.

Example 6.1: Simple Forecasting

This example shows how you can use the SFS object to generate a forecast by automatic exponential smoothing (auto-ESM). This example uses the familiar *Sashelp.Air* data set (in which the data are naturally recorded as monthly passenger counts), and it forecasts the data at two different frequencies: monthly level and quarterly level.

The following DATA step loads the *Sashelp.Air* data set onto the CAS server. This DATA step assumes that your CAS engine libref is named `mycas`, but you can substitute any appropriately defined CAS engine libref.

```sas
data mycas.Air (replace=yes);
set Sashelp.Air;
run;
```

The following SAS code uses the TSMODEL procedure to submit a program that uses the SFS object. The REQUIRE statement loads the SFS package and installs its classes (SFS class) so that the program can use the
package. Failure to include the REQUIRE statement would produce errors when the program is compiled. Air is the dependent variable to be forecast. Because no ACCUMULATE= option is specified in the ID or VAR statements, its default value of TOTAL is used, which accumulates observations within a time period as a total sum of the nonmissing values. The OUTSCALAR statement declares that Nfor, Fitstat, and Rc are numeric variables to be stored in the CAS table mycas.Airmonos, which is specified in the OUTSCALAR= option in the PROC TSMODEL statement. The OUTARRAY statement declares the length-conformant numeric array Airmonfor, which is written to the CAS table mycas.Airmonoa, as specified in the OUTARRAY= option in the PROC TSMODEL statement. You use the DECLARE OBJECT to define the SFS instance named esm. Method calls on the object instance use the dot notation. The calls esm.Initialize, esm.SetY, and so on are performed sequentially with a status check following each to ensure that the method call was successful. If any call fails, the program stops execution. Following a successful esm.Run call, attributes and forecasts are fetched from the esm object.

```
proc tsmodel data=mycas.air
   outarray=mycas.airmonoa(replace=yes)
   outscalar=mycas.airmonos(replace=yes)
   lead=12;
   id date interval=month;
   var air;
   outarray airmonfor;
   outscalar nfor fitstat rc;
   require sfs;
submit;
   declare object esm(sfs);
   rc = esm.Initialize();
   if rc < 0 then do;
       stop;
   end;
   rc = esm.SetY(air);
   if rc < 0 then do;
       stop;
   end;
   rc = esm.Run();
   if rc < 0 then do;
       stop;
   end;
   nfor = esm.nfor();
   fitstat = esm.criterion();
   rc = esm.GetForecast('predict',airmonfor);
   if rc < 0 then do;
       stop;
   end;
endsubmit;
quit;
```

When the program that you submit from PROC TSMODEL runs, it generates a summary of the processing that is performed in your CAS session, as shown in Output 6.1.1.
Output 6.1.1  Summary of Time Series Processing for mycas.air

The TSMODEL Procedure

<table>
<thead>
<tr>
<th>Summary of time series processing for AIR</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of analysis variables</td>
</tr>
<tr>
<td>Number of rows read</td>
</tr>
<tr>
<td>Number of groups read</td>
</tr>
<tr>
<td>Memory for group packages (KB)</td>
</tr>
<tr>
<td>Time to load groups (seconds)</td>
</tr>
<tr>
<td>Minimum time ID</td>
</tr>
<tr>
<td>Maximum time ID</td>
</tr>
<tr>
<td>Minimum time periods</td>
</tr>
<tr>
<td>Maximum time periods</td>
</tr>
<tr>
<td>Number of nodes run</td>
</tr>
<tr>
<td>Number of nodes with data</td>
</tr>
<tr>
<td>Number of nodes with groups</td>
</tr>
<tr>
<td>Number of threads budgeted</td>
</tr>
<tr>
<td>Minimum thread group count</td>
</tr>
<tr>
<td>Maximum thread group count</td>
</tr>
<tr>
<td>Minimum threads active</td>
</tr>
<tr>
<td>Maximum threads active</td>
</tr>
<tr>
<td>Number of groups processed by submitted code</td>
</tr>
<tr>
<td>Number of groups failing</td>
</tr>
<tr>
<td>Elapsed time to process groups (seconds)</td>
</tr>
<tr>
<td>Number of array table rows produced</td>
</tr>
<tr>
<td>Number of scalar table rows produced</td>
</tr>
</tbody>
</table>

When this code runs, the monthly time series for the column Air from the CAS table mycas.Air is formed by using the time ID variable Date. The program uses the SFS object esm to create the best ESM model for these data and then uses that model to forecast for 12 months. The best ESM model is determined by a statistical assessment of some properties of the dependent series (for example, seasonality). Then suitable ESM candidate smoothing methods (for example, SIMPLE, TREND, and DAMPTREND methods for nonseasonal data and WINTERS and MULTWINTERS methods for seasonal data) are used to select the model that has the best in-sample root mean square error (RMSE) fit statistic. The forecast is queried from the SFS object esm and stored in the array Airmonfor, which is then automatically saved to the CAS table mycas.Airmonoa by the use of the OUTARRAY statement and the OUTARRAY= option in the PROC TSMODEL statement. Scalar variables of interest from the execution of the program are also saved by using the OUTSCALAR statement and the OUTSCALAR= option in the PROC TSMODEL statement.

The following code prints the OUTSCALAR= table; the results are shown in Output 6.1.2.

```sas
proc print data=mycas.airmonos;
run;
```

Output 6.1.2  OUTSCALAR Table

<table>
<thead>
<tr>
<th>Obs</th>
<th><em>STATUS</em></th>
<th>nfor</th>
<th>fitstat</th>
<th>rc</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td>156</td>
<td>10.579085435</td>
<td>0</td>
</tr>
</tbody>
</table>
Suppose you want to forecast the quarterly average airline passenger data for a year into the future. The following simple modifications to the SAS code create the quarterly forecast for a four-quarter horizon:

```sas
proc tsmodel data=mycas.air
    outarray=mycas.airqtroa(replace=yes)
    outscalar=mycas.airqtros(replace=yes)
    lead=4;
    id date interval=qtr;
    var air/accumulate=avg;
    outarray airqtrfor;
    outscalar nfor fitstat rc;
    require sfs;
    submit;
    declare object esm(sfs);
    rc = esm.Initialize();
    if rc < 0 then do;
        stop;
    end;
    rc = esm.SetY(air);
    if rc < 0 then do;
        stop;
    end;
    rc = esm.Run();
    if rc < 0 then do;
        stop;
    end;
    nfor = esm.nfor();
    fitstat = esm.criterion();
    rc = esm.GetForecast('predict',airqtrfor);
    if rc < 0 then do;
        stop;
    end;
endsubmit;
quit;
```

The following SAS code demonstrates one of many ways that you might want to generate a plot of the two forecasts. This code uses the SGPLOT procedure to display the monthly forecasts as a time series and to overlay the quarterly forecasts as a scatter plot. The PROC SGPLOT results are shown in Output 6.1.3.

```sas
data airboth;
    label airmonfor='Monthly forecast';
    label airqtrfor='Quarterly forecast';
    merge mycas.airmonoa mycas.airqtroa;
    by date;
run;
```
Example 6.2: Automatic Forecasting Using Predictor Series

This example shows how you can use the SFS object to automatically model and forecast time series data that include the use of predictor series. The following DATA step loads the Sashelp.Pricedata data set onto the CAS server. This DATA step assumes that your CAS engine libref is named mycas, but you can substitute any appropriately defined CAS engine libref.

```sas
data mycas.Pricedata (replace=yes);
  set Sashelp.Pricedata;
run;
```

The example forecasts unit sales (Sale) over the BY groups that are defined by the distinct products (ProductName), considering possible predictor series for price and discount. However, it forecasts sales
by using the relative sales series (Relsale) as the dependent variable and relative price (Relprice) as a candidate predictor in place of the accumulated Sale and Price series. The example then rescales the Relsale forecasts back to the original domain for generating a sales forecast. For this particular example, there is no compelling reason to forecast in the domain of the indexed (relative) series, but it demonstrates the power of the TSMODEL procedure’s programming approach to devise custom treatments of time series processing problems that can be realized in a single pass of the data. Such techniques enable you to combine custom programming logic with the power of the SFS object to perform sophisticated automatic forecasting.

```plaintext
proc tsmodel data=mycas.pricedata
    outarray=mycas.saleoa(replace=yes)
    outscalar=mycas.saleos(replace=yes)
    lead=12;
    by productName;
    id date interval=month start='01jan1998'd end='01dec2002'd;
    var sale /accumulate=sum;
    var price discount /accumulate=avg;
    outarray relsale relprice predict;
    outscalar sbase pbase nfor fitstat rc model $32;
    require sfs;
    submit;
    declare object lasr(sfs);
    sbase=sale[1];
    pbase=price[1];
    do i=1 to _length_;  
        if sale[i] ne . then do;
            relsale[i] = sale[i]/sbase;
        end;
        if price[i] ne . then do;
            relprice[i]=price[i]/pbase;
        end;
    end;
    rc = lasr.Initialize();
    if rc < 0 then do;
        stop;
    end;
    rc = lasr.SetY(relsale);
    if rc < 0 then do;
        stop;
    end;
    rc = lasr.AddX(relprice);
    if rc < 0 then do;
        stop;
    end;
    rc = lasr.AddX(discount);
    if rc < 0 then do;
        stop;
    end;
    rc = lasr.Run();
```
if rc < 0 then do;
    stop;
end;

nfor = lasr.nfor();
fitstat = lasr.criterion();
model = lasr.model();

rc = lasr.GetForecast('predict',predict);
if rc < 0 then do;
    stop;
end;

do i=1 to _length_;  
    predict[i] = predict[i]*sbase;
end;
endsubmit;
quit;

When the program that you submit from PROC TSMODEL runs, it generates a summary of the processing that is performed in your CAS session, as shown in Output 6.2.1.

Output 6.2.1 Summary of Time Series Processing for mycas.pricedata

The TSMODEL Procedure

<table>
<thead>
<tr>
<th>Summary of time series processing for PRICEDATA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of analysis variables</td>
</tr>
<tr>
<td>Number of rows read</td>
</tr>
<tr>
<td>Number of groups read</td>
</tr>
<tr>
<td>Memory for group packages (KB)</td>
</tr>
<tr>
<td>Time to load groups (seconds)</td>
</tr>
<tr>
<td>Minimum time ID</td>
</tr>
<tr>
<td>Maximum time ID</td>
</tr>
<tr>
<td>Minimum time periods</td>
</tr>
<tr>
<td>Maximum time periods</td>
</tr>
<tr>
<td>Number of nodes run</td>
</tr>
<tr>
<td>Number of nodes with data</td>
</tr>
<tr>
<td>Number of nodes with groups</td>
</tr>
<tr>
<td>Number of threads budgeted</td>
</tr>
<tr>
<td>Minimum thread group count</td>
</tr>
<tr>
<td>Maximum thread group count</td>
</tr>
<tr>
<td>Minimum threads active</td>
</tr>
<tr>
<td>Maximum threads active</td>
</tr>
<tr>
<td>Number of groups processed by submitted code</td>
</tr>
<tr>
<td>Number of groups failing</td>
</tr>
<tr>
<td>Elapsed time to process groups (seconds)</td>
</tr>
<tr>
<td>Number of array table rows produced</td>
</tr>
<tr>
<td>Number of scalar table rows produced</td>
</tr>
</tbody>
</table>
The following SAS code prints the table that is specified in the OUTSCALAR= option in the PROC TSMODEL statement. The OUTSCALAR= table includes several variables that were captured from the program execution for each BY group in the Pricedata table. The PROC PRINT results are shown in Output 6.2.2.

```sas
proc print data=mycas.saleos;
run;
```

**Output 6.2.2 OUTSCALAR Table**

<table>
<thead>
<tr>
<th>Obs</th>
<th>productName</th>
<th><em>STATUS</em></th>
<th>sbase</th>
<th>pbase</th>
<th>nfor</th>
<th>fitstat</th>
<th>rc</th>
<th>model</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Product10</td>
<td>0</td>
<td>329</td>
<td>59</td>
<td>72</td>
<td>-308.9521995</td>
<td>0</td>
<td>REGARIMA</td>
</tr>
<tr>
<td>2</td>
<td>Product12</td>
<td>0</td>
<td>413</td>
<td>147</td>
<td>72</td>
<td>-318.0312143</td>
<td>0</td>
<td>REGARIMA</td>
</tr>
<tr>
<td>3</td>
<td>Product15</td>
<td>0</td>
<td>383</td>
<td>120.2</td>
<td>72</td>
<td>-317.1505661</td>
<td>0</td>
<td>REGARIMA</td>
</tr>
<tr>
<td>4</td>
<td>Product16</td>
<td>0</td>
<td>491</td>
<td>70.55</td>
<td>72</td>
<td>-338.6653664</td>
<td>0</td>
<td>REGARIMA</td>
</tr>
<tr>
<td>5</td>
<td>Product2</td>
<td>0</td>
<td>373</td>
<td>115</td>
<td>72</td>
<td>-321.3360261</td>
<td>0</td>
<td>REGARIMA</td>
</tr>
<tr>
<td>6</td>
<td>Product3</td>
<td>0</td>
<td>300</td>
<td>33.4</td>
<td>72</td>
<td>-319.0745186</td>
<td>0</td>
<td>REGARIMA</td>
</tr>
<tr>
<td>7</td>
<td>Product6</td>
<td>0</td>
<td>550</td>
<td>38.88</td>
<td>72</td>
<td>-358.1769437</td>
<td>0</td>
<td>REGARIMA</td>
</tr>
<tr>
<td>8</td>
<td>Product7</td>
<td>0</td>
<td>435</td>
<td>42</td>
<td>72</td>
<td>-330.6016099</td>
<td>0</td>
<td>REGARIMA</td>
</tr>
<tr>
<td>9</td>
<td>Product9</td>
<td>0</td>
<td>461</td>
<td>171.4</td>
<td>72</td>
<td>-330.1588485</td>
<td>0</td>
<td>REGARIMA</td>
</tr>
<tr>
<td>10</td>
<td>Product1</td>
<td>0</td>
<td>355</td>
<td>52.3</td>
<td>72</td>
<td>-337.4629078</td>
<td>0</td>
<td>REGARIMA</td>
</tr>
<tr>
<td>11</td>
<td>Product11</td>
<td>0</td>
<td>240</td>
<td>65.2</td>
<td>72</td>
<td>-340.2472948</td>
<td>0</td>
<td>REGARIMA</td>
</tr>
<tr>
<td>12</td>
<td>Product13</td>
<td>0</td>
<td>359</td>
<td>122</td>
<td>72</td>
<td>-351.6217156</td>
<td>0</td>
<td>REGARIMA</td>
</tr>
<tr>
<td>13</td>
<td>Product17</td>
<td>0</td>
<td>359</td>
<td>80.5</td>
<td>72</td>
<td>-333.9525131</td>
<td>0</td>
<td>REGARIMA</td>
</tr>
<tr>
<td>14</td>
<td>Product5</td>
<td>0</td>
<td>416</td>
<td>36</td>
<td>72</td>
<td>-336.5399506</td>
<td>0</td>
<td>REGARIMA</td>
</tr>
<tr>
<td>15</td>
<td>Product14</td>
<td>0</td>
<td>462</td>
<td>53</td>
<td>72</td>
<td>-325.7278047</td>
<td>0</td>
<td>REGARIMA</td>
</tr>
<tr>
<td>16</td>
<td>Product4</td>
<td>0</td>
<td>418</td>
<td>67.9</td>
<td>72</td>
<td>-310.1082484</td>
<td>0</td>
<td>REGARIMA</td>
</tr>
<tr>
<td>17</td>
<td>Product8</td>
<td>0</td>
<td>404</td>
<td>56.9</td>
<td>72</td>
<td>-321.1618287</td>
<td>0</td>
<td>REGARIMA</td>
</tr>
</tbody>
</table>

The following SAS code generates a scatter plot of the fit statistics for the various product groups on a single graph. Note that the X-axis ordering of ProductName values is forced to be the collating sequence by using the result of the PROC SORT step, whereas the order from the preceding PROC PRINT step is the natural order that is returned from the CAS table. When processing results are generated as output tables from CAS actions, it is often necessary to sort them in order to create a desired row set ordering. The PROC SGPLOT results are shown in Output 6.2.3.

```sas
proc sort data=mycas.saleos out=saleos;
    by productName;
run;
proc sgplot data=saleos;
    scatter x=productName y=fitstat;
run;
```
Example 6.3: Using SFS with PROC CAS

This example shows how you can use the SFS object with the CAS procedure to call the `timeData.runTimeCode` action. This example uses the same `Sashelp.Pricedata` data set as is used in Example 6.2. This example uses the auto-ESM mode of the SFS object to generate forecasts and confidence bands for average sales by region (Region).

The following SAS code shows how you can use PROC CAS to submit the program that uses the SFS object. Unlike PROC TSMODEL, PROC CAS does not have custom syntax and statements for the `timeData.runTimeCode` action. To use PROC CAS to call a CAS action, you must form the CAS action’s call as it is defined in the CAS action set. For more information about the `timeData` action set, see SAS Visual Forecasting: Programming Guide. For more information about PROC CAS, see SAS Cloud Analytic Services: CASL Reference. Even without detailed documentation, you can see the connections between the `timeData.runTimeCode` action’s arguments and the options and statements that are used in Example 6.2, which uses the TSMODEL procedure.
%macro cmpcode(yvar,pred,ucl,lcl);
declare object esm(sfs);
rc = esm.initialize();
if rc < 0 then do;
   stop;
end;
rc = esm.sety(&yvar);
if rc < 0 then do;
   stop;
end;
rc = esm.run();
if rc < 0 then do;
   stop;
end;
nfor = esm.nfor();
fitstat = esm.criterion();
rc = esm.getForecast('predict',&pred);
if rc < 0 then do;
   stop;
end;
rc = esm.getForecast('upper',&ucl);
if rc < 0 then do;
   stop;
end;
rc = esm.getForecast('lower',&lcl);
if rc < 0 then do;
   stop;
end;
%mend;

proc cas;
   cmpcode="%cmpcode(sale,sale_for,sale_ucl,sale_lcl)"
   session casref=mysess; run;
   timeData.runTimeCode / table={name="pricedata"
      groupby="region"
   } require={{pkg="sfs"}}
   series={{name="sale" acc="sum"}}
   timeid="date"
   interval="month"
   arrayOut={table={name="csaleoa" replace=true}
      arrays={"sale_for" "sale_ucl" "sale_lcl"}
   }
   scalarOut={table={name="csaleos" replace=true}
      scalars="rc" "nfor" "fitstat"
   }
   lead=12
   code=cmpcode;
run;
quit;

When the timeData action runs the program that you submit from PROC CAS, it generates a summary of the processing that is performed by your CAS session. That summary is shown in Output 6.3.1.
Output 6.3.1  Summary of Time Series Processing for mycas.pricedata

Results from timeData.runTimeCode

<table>
<thead>
<tr>
<th>Summary of time series processing for PRICEDATA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of analysis variables</td>
</tr>
<tr>
<td>Number of rows read</td>
</tr>
<tr>
<td>Number of groups read</td>
</tr>
<tr>
<td>Memory for group packages (KB)</td>
</tr>
<tr>
<td>Time to load groups (seconds)</td>
</tr>
<tr>
<td>Minimum time ID</td>
</tr>
<tr>
<td>Maximum time ID</td>
</tr>
<tr>
<td>Minimum time periods</td>
</tr>
<tr>
<td>Maximum time periods</td>
</tr>
<tr>
<td>Number of nodes run</td>
</tr>
<tr>
<td>Number of nodes with data</td>
</tr>
<tr>
<td>Number of nodes with groups</td>
</tr>
<tr>
<td>Number of threads budgeted</td>
</tr>
<tr>
<td>Minimum thread group count</td>
</tr>
<tr>
<td>Maximum thread group count</td>
</tr>
<tr>
<td>Minimum threads active</td>
</tr>
<tr>
<td>Maximum threads active</td>
</tr>
<tr>
<td>Number of groups processed by submitted code</td>
</tr>
<tr>
<td>Number of groups failing</td>
</tr>
<tr>
<td>Elapsed time to process groups (seconds)</td>
</tr>
<tr>
<td>Number of array table rows produced</td>
</tr>
<tr>
<td>Number of scalar table rows produced</td>
</tr>
</tbody>
</table>

The following SAS code generates a series plot of the forecasts and confidence limits for region 1.

```sas
data csaleoa;
  set mycas.csaleoa;
run;

proc sort data=csaleoa;
  by region date;
run;

proc sgplot data=csaleoa(where=(region=1));
  band x=date upper=sale_ucl lower=sale_lcl;
  series x=date y=sale_for;
  scatter x=date y=sale;
run;
```

PROC SGPLOT results are shown in Output 6.3.2.
Output 6.3.2 Sales Forecasts for Region 1
# Chapter 7

## Singular Spectrum Analysis Package

### Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Overview: SSA Package</td>
<td>212</td>
</tr>
<tr>
<td>SSA Package Summary</td>
<td>212</td>
</tr>
<tr>
<td>Return Codes</td>
<td>212</td>
</tr>
<tr>
<td>SSA Object</td>
<td>213</td>
</tr>
<tr>
<td>SSA Synopsis</td>
<td>214</td>
</tr>
<tr>
<td>SSA Methods</td>
<td>215</td>
</tr>
<tr>
<td>OUTGROUPS Object</td>
<td>217</td>
</tr>
<tr>
<td>OUTGROUPS Synopsis</td>
<td>218</td>
</tr>
<tr>
<td>OUTGROUPS Methods</td>
<td>218</td>
</tr>
<tr>
<td>OUTSSA Object</td>
<td>218</td>
</tr>
<tr>
<td>OUTSSA Synopsis</td>
<td>219</td>
</tr>
<tr>
<td>OUTSSA Methods</td>
<td>220</td>
</tr>
<tr>
<td>OUTSV Object</td>
<td>220</td>
</tr>
<tr>
<td>OUTSV Synopsis</td>
<td>221</td>
</tr>
<tr>
<td>OUTSV Methods</td>
<td>221</td>
</tr>
<tr>
<td>OUTWCORR Object</td>
<td>222</td>
</tr>
<tr>
<td>OUTWCORR Synopsis</td>
<td>223</td>
</tr>
<tr>
<td>OUTWCORR Methods</td>
<td>223</td>
</tr>
<tr>
<td>Details</td>
<td>223</td>
</tr>
<tr>
<td>Singular Spectrum Analysis</td>
<td></td>
</tr>
<tr>
<td>Examples: SSA Package</td>
<td>226</td>
</tr>
<tr>
<td>Using CAS Sessions and CAS Engine Librefs</td>
<td>226</td>
</tr>
<tr>
<td>Example 7.1: Singular Spectrum Analysis with Different Grouping Steps</td>
<td>227</td>
</tr>
<tr>
<td>Example 7.2: Run a Singular Spectrum Analysis and Collect the Output</td>
<td>229</td>
</tr>
<tr>
<td>Example 7.3: Run a Singular Spectrum Analysis of a User-Selected Group</td>
<td>235</td>
</tr>
<tr>
<td>References</td>
<td>237</td>
</tr>
</tbody>
</table>
Overview: SSA Package

The singular spectrum analysis (SSA) package contains a set of time series analysis functions that can be used as part of the programming statements in the TSMODEL procedure. This package provides a flexible way to analyze and decompose time series within the procedure.

**NOTE:** Each function in this chapter has a prefix of “SSA.”; however, the prefixes are omitted in descriptions for better readability. The mycas libref in the examples refers to CAS library that is linked to a caslib. The mycas.air data table that is used in the examples refers to Sashelp.Air data. All the examples in this chapter assume that your CAS engine libref is named mycas, but you can substitute any appropriately defined CAS engine libref. For more information about CAS engine librefs, see *SAS Cloud Analytic Services: User’s Guide*.

SSA Package Summary

Table 7.1 summarizes the objects in the SSA package.

**Table 7.1  Objects in the Singular Spectrum Analysis Package**

<table>
<thead>
<tr>
<th>Object</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SSA</td>
<td>Perform singular spectrum analysis of a time series</td>
</tr>
<tr>
<td>OUTGROUPS</td>
<td>Collect grouping results from an SSA object</td>
</tr>
<tr>
<td>OUTSSA</td>
<td>Collect output from an SSA object</td>
</tr>
<tr>
<td>OUTSV</td>
<td>Collect singular values from an SSA object</td>
</tr>
<tr>
<td>OUTWCORR</td>
<td>Collect absolute values of w-correlations from an SSA object</td>
</tr>
</tbody>
</table>

Return Codes

Table 7.2 shows the return code (rc in method statements) status values that are used in this package. These status code values are returned after a method that is associated with an object is called; they can help determine whether the method executed successfully.
### Table 7.2 Return Codes

<table>
<thead>
<tr>
<th>Status</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt; 0</td>
<td>An unrecoverable error occurred. No result was produced.</td>
</tr>
<tr>
<td>= 0</td>
<td>Unconditional success. The requested action completed and a normal result was produced.</td>
</tr>
<tr>
<td>&gt; 0</td>
<td>Conditional success or warning. A result was produced subject to conditions.</td>
</tr>
</tbody>
</table>

Upon returning a negative return code, most methods in the SSA package objects also write a message to the output log that explains the causes of the related failure. These messages provide useful information during the process of debugging a user program. In the TSMODEL procedure, the output log is stored in the CAS table that is specified in the OUTLOG= option in the PROC TSMODEL statement. For more information about how to enable and configure logging and about how to access the output log after an invocation of the TSMODEL procedure, see Chapter 11, “The TSMODEL Procedure” (*SAS Visual Forecasting: Forecasting Procedures*).

### SSA Object

The SSA object groups time series variables to be used as input for the other SSA package objects.

*Table 7.3* summarizes the methods that are associated with the SSA object.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AddGroup</td>
<td>Add groupings manually</td>
</tr>
<tr>
<td>GetResult</td>
<td>Get the decomposition result</td>
</tr>
<tr>
<td>Initialize</td>
<td>Initialize an SSA instance</td>
</tr>
<tr>
<td>SetOption</td>
<td>Set a named option</td>
</tr>
<tr>
<td>SetY</td>
<td>Set the input time series</td>
</tr>
<tr>
<td>Run</td>
<td>Run the singular spectrum analysis</td>
</tr>
</tbody>
</table>

*Figure 7.1* diagrams the methods of the SSA object.
Figure 7.1 SSA Data Flow

SSA Synopsis

```plaintext
DECLARE OBJECT obj (SSA) ;

Method syntax, in order of typical usage:

rc=obj.Initialize () ;
rc=obj.SetY (YSeries) ;
rc=obj.AddGroup (NumericArray) ;
rc=obj.SetOption ('Name', Value) ;
rc=obj.Run () ;
rc=obj.GetResult (OutputArray, GroupNumber) ;
```
SSA Methods

SSA.AddGroup Method

\[ rc = \text{obj}.\text{AddGroup} (\text{NumericArray}) ; \]

Adds a group manually for the SSA instance. This method can be called multiple times. Each call adds a new group.

**Input Arguments**

You must specify the following input argument:

- **NumericArray** specifies a numeric array of positive integers that indicates a group.

SSA.GetResult Method

\[ rc = \text{obj}.\text{GetResult} (\text{OutputArray},\text{GroupNumber}) ; \]

Outputs the selected group component to an array.

**Input Arguments**

You must specify the following input arguments:

- **OutputArray** specifies a dynamic array that is used to store the output group component.
- **GroupNumber** specifies a positive integer that indicates which group to output.

SSA.Initialize Method

\[ rc = \text{obj}.\text{Initialize} () ; \]

Initializes an SSA instance to an empty state. This method must be called before specifying the time series arrays and other attributes for the SSA instance.

**Arguments**

There are no arguments associated with this method.

SSA.SetOption Method

\[ rc = \text{obj}.\text{SetOption} ('\text{Name}', \text{Value}) ; \]

Sets a named option for the SSA instance.

**Input Arguments**

You can specify one of the following 'Names' and its associated Value:

- **'ADJUSTMEAN'** takes a string \text{Value} that specifies whether the series should be adjusted by its mean prior to performing the singular spectrum analysis. You can specify one of the following Values:
TRUE | T | YES | Y  adjusts the mean prior to performing singular spectrum analysis.
FALSE | F | NO | N  does not adjust the mean prior to performing singular spectrum analysis.

'LENGTH'  takes a nonnegative integer Value less than the length of YSeries and specifies the window length to be used. The default is T/4, where T is the length of YSeries after trimming leading and ending missing values, which is specified in the SSA.SetY method.

'METHOD'  takes a string Value that specifies the method of grouping. You can specify one of the following Values:

AUTO      uses automatic grouping.
GROUPS    specifies that the user selects the grouping.
THRESHOLD divides the SSA components into two groups based on the cumulative percentage of their singular values.

The default is THRESHOLD.

'NUMGROUPS'  takes a nonnegative integer Value less than or equal to 'LENGTH' and specifies the maximum number of groups to be retained when automatic grouping is used.

'THRESHOLDPCT'  takes a numeric Value between 0 and 100 that specifies a percentage to be used to divide the SSA components into two groups based on the cumulative percentage of their singular values. The default is 90.

'WCORRADJUSTMEAN'  takes a string Value that specifies whether to adjust by its mean prior to calculating w-correlations. You can specify one of the following Values:

TRUE | T | YES | Y  adjusts by its mean prior to calculating w-correlations.
FALSE | F | NO | N  does not adjust by its mean prior to calculating w-correlations.

'WCORRCUTOFF'  takes a numeric Value between 0 and 100 that specifies the cutoff of w-correlations used when the value of the 'METHOD' argument is AUTO. The default is 90.

SSA.SetY Method

rc=obj.SetY (YSeries) ;

Adds a dependent time series array (YSeries) to the SSA instance.

Input Arguments
You must specify the following input argument:

YSeries    specifies a numeric array that contains the dependent series for the SSA instance.
**SSA.Run Method**

```python
rc = obj.Run();
```

Runs the SSA object to perform the SSA analysis by using the dependent `YSeries` that has been specified for it. Upon successful completion, various results can be extracted from the SSA object.

**Arguments**

There are no arguments associated with this method.

---

**OUTGROUPS Object**

The OUTGROUPS object collects output from an SSA object.

Table 7.4 summarizes the methods that are associated with the OUTGROUPS object.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Collect</td>
<td>Collect the grouping results of an SSA object</td>
</tr>
<tr>
<td>nrows</td>
<td>Get the current row count from the OUTGROUPS object</td>
</tr>
</tbody>
</table>

Figure 7.2 diagrams the methods of the OUTGROUPS object.

**Figure 7.2** OUTGROUPS Data Flow
OUTGROUPS Synopsis

DECLARE OBJECT obj (OUTGROUPS) ;

Method syntax, in order of typical usage:

\[ rc = obj.\text{Collect} () ; \]
\[ nrows = obj.\text{nrows} () ; \]

OUTGROUPS Methods

OUTGROUPS.Collect Method

\[ rc = obj.\text{Collect} (\text{SSAObj}) ; \]

Collects the grouping results from an SSA object and saves the result to a CAS table.

Arguments
There are no arguments associated with this method.

OUTGROUPS.nrows Attribute

\[ nrows = obj.\text{nrows} () ; \]

Gets the current row count from the OUTGROUPS instance.

Arguments
There are no arguments associated with this method.

OUTSSA Object

The OUTSSA object collects output from SSA object.

Table 7.5 summarizes the methods that are associated with the OUTSSA object.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Collect</td>
<td>Collect the results of an SSA object</td>
</tr>
<tr>
<td>nrows</td>
<td>Get the current row count from the OUTSSA object</td>
</tr>
</tbody>
</table>

Figure 7.3 diagrams the methods of the OUTSSA object.
**OUTSSA Synopsis**

```
DECLARE OBJECT obj (OUTSSA <('Name', Value )> ) ;
```

**Input Arguments**
You can specify one of the following ‘Names’ and its associated Value:

‘NGROUP’ takes a nonnegative integer Value less than or equal to 100 and specifies the maximum number of groups that an OUTSSA object can contain. The default is 100.

**Method Syntax**
In order of typical usage:

```
rc=obj.Collect () ;
nrows=obj.nrows () ;
```
OUTSSA Methods

OUTSSA.Collect Method

\[ rc = obj.Collect (SSAObj) ; \]

Collects the output of singular spectrum analysis from an SSA object and saves the results to a CAS table. If the SSAObj contains more groups than NGROUP, only the first NGROUP groups will be collected. If the SSAObj contains less groups than NGROUP, the rest of the OUTSSAObj will be filled with missing values.

Arguments
There are no arguments associated with this method.

OUTSSA.nrows Attribute

\[ nrows = obj.nrows () ; \]

Gets the current row count from the OUTSSA instance.

Arguments
There are no arguments associated with this method.

OUTSV Object

The OUTSV object collects output from SSA object.

Table 7.6 summarizes the methods that are associated with the OUTSV object.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Collect</td>
<td>Collect the singular values of an SSA object</td>
</tr>
<tr>
<td>nrows</td>
<td>Get the current row count from the OUTSV object</td>
</tr>
</tbody>
</table>

Figure 7.4 diagrams the methods of the OUTSV object.
**OUTSV Synopsis**

DECLARE OBJECT obj (OUTSV) ;

Method syntax, in order of typical usage:

\[ rc = \text{obj}.\text{Collect}() ; \]
\[ \text{nrows} = \text{obj}.\text{nrows}() ; \]

**OUTSV Methods**

**OUTSV.Collect Method**

\[ rc = \text{obj}.\text{Collect}(\text{SSAObj}) ; \]

Collects the singular values from an SSA object and saves the result to a CAS table.

**Arguments**

There are no arguments associated with this method.

**OUTSV.nrows Attribute**

\[ \text{nrows} = \text{obj}.\text{nrows}() ; \]

Gets the current row count from the OUTSV instance.
**Arguments**
There are no arguments associated with this method.

---

**OUTWCORR Object**

The OUTWCORR object collects output from an SSA object.

Table 7.7 summarizes the methods that are associated with the OUTWCORR object.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Collect</td>
<td>Collect the absolute values of the w-correlations of an SSA object</td>
</tr>
<tr>
<td>nrows</td>
<td>Get the current row count from the OUTWCORR object</td>
</tr>
</tbody>
</table>

Figure 7.5 diagrams the methods of the OUTWCORR object.

**Figure 7.5** OUTWCORR Data Flow

![OUTWCORR Data Flow Diagram](image-url)
OUTWCORR Synopsis

DECLARE OBJECT obj (OUTWCORR) ;

Method syntax, in order of typical usage:

rc = obj.Collect () ;
nrows = obj.nrows () ;

OUTWCORR Methods

OUTWCORR.Collect Method

rc = obj.Collect (SSAObj ) ;

Collects the absolute values of the w-correlations from an SSA object and saves the result to a CAS table.

Arguments
There are no arguments associated with this method.

OUTWCORR.nrows Attribute

nrows = obj.nrows () ;

Gets the current row count from the OUTWCORR instance.

Arguments
There are no arguments associated with this method.

Details

Singular Spectrum Analysis

Singular spectrum analysis (SSA) is a technique for decomposing a time series into additive components and categorizing those components based on the magnitudes of their contributions. SSA uses a single parameter, the window length (specified in the 'LENGTH' argument in the SSA.SetOption method), to quantify patterns in a time series without relying on prior information about the structure of the series. The window length represents the maximum lag that is considered in the analysis, and it corresponds to the dimensionality of the principal components analysis (PCA) on which SSA is based. The components are combined into groups to categorize their roles in the SSA decomposition.
Main SSA Steps

Given a time series, \( y_t \), for \( t = 1, \ldots, T \), and a window length, \( 2 \leq L \leq T/2 \), singular spectrum analysis (Golyandina, Nekrutkin, and Zhigljavsky, 2001) decomposes the time series into spectral groupings by using the following steps:

**Step 1. Embedding**

Using the time series, form a \( K \times L \) trajectory matrix, \( X \), with elements

\[
X = \{x_{k,l}\}^{K,L}_{k=1, l=1}
\]

such that \( x_{k,l} = y_{k-l+1} \) for \( k = 1, \ldots, K \) and \( l = 1, \ldots, L \), where \( K = T - L + 1 \). By definition \( L \leq K < T \), because \( 2 \leq L \leq T/2 \).

**Step 2. Decomposition**

Apply singular value decomposition to the trajectory matrix,

\[
X = UQV
\]

where \( U \) represents the \( K \times L \) matrix that contains the left-hand-side (LHS) eigenvectors, \( Q \) represents the diagonal \( L \times L \) matrix that contains the singular values, and \( V \) represents the \( L \times L \) matrix that contains the right-hand-side (RHS) eigenvectors. Therefore,

\[
X = \sum_{l=1}^{L} X^{(l)} = \sum_{l=1}^{L} u_l q_l v_l'
\]

where \( X^{(l)} \) represents the \( K \times L \) principal component matrix, \( u_l \) represents the \( K \times 1 \) left-hand-side (LHS) eigenvector, \( q_l \) represents the singular value, and \( v_l \) represents the \( L \times 1 \) right-hand-side (RHS) eigenvector that is associated with the \( l \)th window index.

**Step 3. Grouping**

For each group index, \( m = 1, \ldots, M \), define a group of window indices \( I_m \subseteq \{1, \ldots, L\} \). Let the following equation represent the grouped trajectory matrix for group \( I_m \):

\[
X_{I_m} = \sum_{l \in I_m} X^{(l)} = \sum_{l \in I_m} u_l q_l v_l'
\]

If groupings represent a spectral partition,

\[
\bigcup_{m=1}^{M} I_m = \{1, \ldots, L\} \quad \text{and} \quad I_m \cap I_n = \emptyset \quad \text{for} \quad m \neq n
\]

then according to the singular value decomposition theory,

\[
X = \sum_{m=1}^{M} X_{I_m}
\]
Step 4. Averaging

For each group index, \( m = 1, \ldots, M \), compute the diagonal average of \( X_{I_m} \),

\[
\bar{x}^{(m)}_t = \frac{1}{n_t} \sum_{l=s_t}^{e_t} x^{(m)}_{t-l+1,l}
\]

where

\[
\begin{align*}
  s_t &= 1, & e_t &= t, & n_t &= t & \text{for} & & 1 \leq t < L \\
  s_t &= 1, & e_t &= L, & n_t &= L & \text{for} & & L \leq t \leq T - L + 1 \\
  s_t &= t - T + L, & e_t &= L, & n_t &= T - t + 1 & \text{for} & & T - L + 1 < t \leq T
\end{align*}
\]

If the groupings represent a spectral partition, then by definition

\[
y_t = \sum_{m=1}^{M} \bar{x}^{(m)}_t
\]

Hence, singular spectrum analysis additively decomposes the original time series, \( y_t \), into \( m \) component series \( \bar{x}^{(m)}_t \) for \( m = 1, \ldots, M \).

Computing w-Correlations

An important step in SSA is specifying the groups \( I_m \subset \{1, \ldots, L\} \) for \( m = 1, \ldots, M \). In order to automate the SSA grouping step, the weighted correlations (w-correlations) are computed:

\[
\rho_{i,j}(w) = \frac{\langle \bar{x}^{(i)}_t, \bar{x}^{(j)}_t \rangle_w}{\| \bar{x}^{(i)}_t \|_w \| \bar{x}^{(j)}_t \|_w}
\]

where \( \langle \bar{x}^{(i)}_t, \bar{x}^{(j)}_t \rangle_w = \sum_{t=1}^{T} w_t \bar{x}^{(i)}_t \bar{x}^{(j)}_t \) and \( w_t = \min(t, L, T - t + 1) \).

Specifying the Window Length

You can explicitly specify the maximum window length, \( 2 \leq L \leq 1000 \), by using the ‘LENGTH’ argument in the SSA.SetOption method. The window length is reduced based on the time series length, \( T \), to enforce the requirement that \( 2 \leq L \leq T/2 \).

Specifying the Groups

The SSA.AddGroup method explicitly specifies the composition and add it to the groups, or you can use the ‘THRESHOLD’ argument following the ‘METHOD’ argument in the SSA.SetOption method to implicitly specify the grouping. The ‘THRESHOLD’ argument is useful for removing noise or less dominant patterns from the time series.

Let \( 0 < \alpha < 100 \) be the cumulative percentage singular value that is specified by the ‘THRESHOLDPCT’ argument. Then the last group, \( I_M = \{l_\alpha, \ldots, L\} \), is determined by the smallest value such that

\[
\left( \sum_{l=1}^{L-1} q_l / \sum_{l=1}^{L} q_l \right) \geq \alpha \quad 1 < l_\alpha \leq L
\]
Using this rule, the last group, $I_M$, describes the least dominant patterns in the time series, and the size of the last group is at least one and is less than the window length, $L \geq 2$.

The magnitudes of the principal components that are selected by the \texttt{THRESHOLDPCT} argument are based on the singular values that appear on the diagonal of $Q$. Alternatively, each principal component’s contribution to variation in the series can be quantified by using the squares of the singular values. An \texttt{OUTSV} object collects the singular values from an SSA object.

**Automatic Grouping**

Besides specifying the groups explicitly, you can also specify the value of \texttt{AUTO} for \texttt{THRESHOLDPCT} argument in order to perform the automatic grouping. The group number is specified by the \texttt{NUMGROUPS} argument in the SSA.SetOption method. In this SSA automatic grouping, the following steps are performed:

1. The maximal number of groups is initially assumed to be $M = L$.
2. The groups are diagonally averaged as described previously: $\hat{\xi}_t^{(m)}$ for $m = 1, \ldots, L$.
3. The weighted correlations (w-correlations) between groups are computed: $\hat{\rho}_{i,j}^{(m)}$.
4. The groups are selected based on the w-correlations for which the absolute values are close to 1. More formally, $I_m \subset \{1, \ldots, L\}$ such that $|\hat{\rho}_{i,j}^{(m)}| \approx 1$ whenever $i, j \in I_m$.

---

### Examples: SSA Package

#### Using CAS Sessions and CAS Engine Librefs

SAS Cloud Analytic Services (CAS) is the analytic server and associated cloud services in SAS Viya. This section describes how to create a CAS session and set up a CAS engine libref that you can use to connect to the CAS session. It assumes that you have a CAS server already available; contact your system administrator if you need help starting and terminating a server. This CAS server is identified by specifying the host on which it runs and the port on which it listens for communications. To simplify your interactions with this CAS server, the host information and port information for the server are stored as SAS option values that are retrieved automatically whenever this CAS server needs to be accessed. You can examine the host and port values for the server at your site by using the following statements:

```sas
proc options option=(CASHOST CASPORT);
run;
```

In addition to starting a CAS server, your system administrator might also have created a CAS session and a CAS engine libref for your use. You can define your own sessions and CAS engine librefs that connect to the CAS server as shown in the following statements:

```sas
cas mysess;
libname mycas cas sessref=mysess;
```
The CAS statement creates the CAS session named mysess, and the LIBNAME statement creates the mycas CAS engine libref that you use to connect to this session. It is not necessary to explicitly name the CASHOST and CASPORT of the CAS server in the CAS statement, because these values are retrieved from the corresponding SAS option values.

If you have created the mysess session, you can terminate it by using the TERMINATE option in the CAS statement as follows:

```
cas mysess terminate;
```

For more information about the CAS statement and the LIBNAME statement, see SAS Cloud Analytic Services: User’s Guide. For general information about CAS and CAS sessions, see SAS Cloud Analytic Services: Fundamentals.

---

Example 7.1: Singular Spectrum Analysis with Different Grouping Steps

The following statements extract two additive components from the Sashelp.Air time series by using the SSA.SetOption method to request that the first component represent 80% of the variability in the series and to specify a window length of 12. The resulting groupings, which consist of the first three and remaining nine singular value components, are presented in Output 7.1.2. Output 7.1.1 shows the values of each components.

```
proc tsmodel data=mycas.air outarray=mycas.out1(replace=yes);
  id date interval=month;
  var air;
  outarray group1 group2;
  require ssa;
  submit;
   declare object s(ssa);
   rc = s.Initialize();
   rc = s.SetY(air);
   rc = s.SetOption('METHOD','THRESHOLD');
   rc = s.SetOption('LENGTH',12);
   rc = s.SetOption('THRESHOLDPCT',80);
   rc = s.Run();
   rc = s.GetResult(group1,1);
   rc = s.GetResult(group2,2);
  endsubmit;
run;
```
### Output 7.1.1  SSA Results of AIR Data (Partial Output)

#### SSA Results Table

<table>
<thead>
<tr>
<th>Obs</th>
<th>STATUS</th>
<th>SEASON</th>
<th>CYCLE</th>
<th>DATE</th>
<th>AIR</th>
<th>group1</th>
<th>group2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>JAN1949</td>
<td>112</td>
<td>107.86146454</td>
<td>4.1385354609</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>FEB1949</td>
<td>118</td>
<td>113.40770921</td>
<td>4.5922907916</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>MAR1949</td>
<td>132</td>
<td>121.50939907</td>
<td>10.490600933</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>APR1949</td>
<td>129</td>
<td>130.04589081</td>
<td>-1.045890813</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>0</td>
<td>5</td>
<td>MAY1949</td>
<td>121</td>
<td>136.62206663</td>
<td>-15.62206663</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>0</td>
<td>6</td>
<td>JUN1949</td>
<td>135</td>
<td>139.92109951</td>
<td>-4.921099508</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td>0</td>
<td>7</td>
<td>JUL1949</td>
<td>148</td>
<td>140.0182405</td>
<td>7.9817594087</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
<td>0</td>
<td>8</td>
<td>AUG1949</td>
<td>148</td>
<td>137.10648141</td>
<td>10.893518393</td>
</tr>
<tr>
<td>9</td>
<td>0</td>
<td>0</td>
<td>9</td>
<td>SEP1949</td>
<td>136</td>
<td>131.6108694</td>
<td>4.3891305963</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>0</td>
<td>10</td>
<td>OCT1949</td>
<td>119</td>
<td>124.86020336</td>
<td>-5.860203358</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>0</td>
<td>11</td>
<td>NOV1949</td>
<td>104</td>
<td>118.55785911</td>
<td>-14.55785911</td>
</tr>
<tr>
<td>12</td>
<td>0</td>
<td>0</td>
<td>12</td>
<td>DEC1949</td>
<td>118</td>
<td>114.83044425</td>
<td>3.169555735</td>
</tr>
<tr>
<td>13</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>JAN1950</td>
<td>115</td>
<td>115.20424909</td>
<td>-0.20424909</td>
</tr>
<tr>
<td>14</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>FEB1950</td>
<td>126</td>
<td>120.16159642</td>
<td>5.838403578</td>
</tr>
<tr>
<td>15</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>MAR1950</td>
<td>141</td>
<td>128.51838033</td>
<td>12.481619674</td>
</tr>
<tr>
<td>16</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>APR1950</td>
<td>135</td>
<td>138.69641074</td>
<td>-3.696410743</td>
</tr>
<tr>
<td>17</td>
<td>0</td>
<td>0</td>
<td>5</td>
<td>MAY1950</td>
<td>125</td>
<td>148.31164461</td>
<td>-23.31164461</td>
</tr>
<tr>
<td>18</td>
<td>0</td>
<td>0</td>
<td>6</td>
<td>JUN1950</td>
<td>149</td>
<td>155.63397881</td>
<td>-6.633978805</td>
</tr>
<tr>
<td>19</td>
<td>0</td>
<td>0</td>
<td>7</td>
<td>JUL1950</td>
<td>170</td>
<td>158.43892976</td>
<td>11.561072045</td>
</tr>
<tr>
<td>20</td>
<td>0</td>
<td>0</td>
<td>8</td>
<td>AUG1950</td>
<td>170</td>
<td>156.15977587</td>
<td>13.840224125</td>
</tr>
<tr>
<td>21</td>
<td>0</td>
<td>0</td>
<td>9</td>
<td>SEP1950</td>
<td>158</td>
<td>149.98327684</td>
<td>8.0167231632</td>
</tr>
<tr>
<td>22</td>
<td>0</td>
<td>0</td>
<td>10</td>
<td>OCT1950</td>
<td>133</td>
<td>142.3316862</td>
<td>-9.331686203</td>
</tr>
<tr>
<td>23</td>
<td>0</td>
<td>0</td>
<td>11</td>
<td>NOV1950</td>
<td>114</td>
<td>136.48277045</td>
<td>-22.48277045</td>
</tr>
<tr>
<td>24</td>
<td>0</td>
<td>0</td>
<td>12</td>
<td>DEC1950</td>
<td>140</td>
<td>135.12143886</td>
<td>4.8785611372</td>
</tr>
<tr>
<td>25</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>JAN1951</td>
<td>145</td>
<td>139.06040538</td>
<td>5.9395946176</td>
</tr>
<tr>
<td>26</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>FEB1951</td>
<td>150</td>
<td>147.82975674</td>
<td>2.1702432585</td>
</tr>
<tr>
<td>27</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>MAR1951</td>
<td>178</td>
<td>159.98626639</td>
<td>18.013733612</td>
</tr>
<tr>
<td>28</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>APR1951</td>
<td>163</td>
<td>172.36958802</td>
<td>-9.36958802</td>
</tr>
<tr>
<td>29</td>
<td>0</td>
<td>0</td>
<td>5</td>
<td>MAY1951</td>
<td>172</td>
<td>182.59462252</td>
<td>-10.59462252</td>
</tr>
<tr>
<td>30</td>
<td>0</td>
<td>0</td>
<td>6</td>
<td>JUN1951</td>
<td>178</td>
<td>188.07654364</td>
<td>-10.07654364</td>
</tr>
<tr>
<td>31</td>
<td>0</td>
<td>0</td>
<td>7</td>
<td>JUL1951</td>
<td>199</td>
<td>188.95327142</td>
<td>10.046728582</td>
</tr>
<tr>
<td>32</td>
<td>0</td>
<td>0</td>
<td>8</td>
<td>AUG1951</td>
<td>199</td>
<td>185.32511671</td>
<td>13.674883291</td>
</tr>
<tr>
<td>33</td>
<td>0</td>
<td>0</td>
<td>9</td>
<td>SEP1951</td>
<td>184</td>
<td>178.79247812</td>
<td>5.2075218797</td>
</tr>
<tr>
<td>34</td>
<td>0</td>
<td>0</td>
<td>10</td>
<td>OCT1951</td>
<td>162</td>
<td>171.14295364</td>
<td>-9.142953642</td>
</tr>
<tr>
<td>35</td>
<td>0</td>
<td>0</td>
<td>11</td>
<td>NOV1951</td>
<td>146</td>
<td>164.82980579</td>
<td>-18.82980579</td>
</tr>
<tr>
<td>36</td>
<td>0</td>
<td>0</td>
<td>12</td>
<td>DEC1951</td>
<td>166</td>
<td>161.88217886</td>
<td>4.1178211444</td>
</tr>
<tr>
<td>37</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>JAN1952</td>
<td>171</td>
<td>163.7659745</td>
<td>7.2340254973</td>
</tr>
<tr>
<td>38</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>FEB1952</td>
<td>180</td>
<td>170.54023775</td>
<td>9.4597622459</td>
</tr>
<tr>
<td>39</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>MAR1952</td>
<td>193</td>
<td>180.92835641</td>
<td>12.071643586</td>
</tr>
<tr>
<td>40</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>APR1952</td>
<td>181</td>
<td>193.94052435</td>
<td>-12.94052435</td>
</tr>
</tbody>
</table>
Example 7.2: Run a Singular Spectrum Analysis and Collect the Output

This example runs a singular spectrum analysis in an SSA object and uses an OUTSSA object to collect the output. It extracts the first three important additive components automatically from the mycas.Air time series by specifying AUTO for the 'METHOD' argument and specifying the 'NUMGROUPS' argument in the SSA.SetOption method. Output 7.2.5 shows the resulting groupings together with the original data. Output 7.2.1 shows the singular value of each singular value component. A large singular value indicates that this singular value component captures a large portion of total variation of the original data. Output 7.2.2 shows the values of each component. Output 7.2.3 shows the resulting groupings. Output 7.2.4 shows the w-correlations that are used for autogrouping. Output 7.2.5 plots each component together with the original data.

```
proc tsmodel data=mycas.air outobj=(os=mycas.analytic (replace=YES)
                               osv=mycas_sv (replace=YES) osg=mycas.groups (replace=YES)
                               osw=mycas.wcorr (replace=YES));
   id date interval=month;
   var air;
   require ssa;
```
submit;
declare object s(ssa);
declare object os(outssa('NGROUP',3));
declare object osv(outsv);
declare object osg(outgroups);
declare object osw(outwcorr);
rc = s.Initialize();
rc = s.SetY(air);
rc = s.SetOption('METHOD','AUTO');
rc = s.SetOption('NUMGROUPS',3);
rc = s.SetOption('LENGTH',12);
rc = s.Run();
rc = os.Collect(s);
rc = osv.Collect(s);
rc = osg.Collect(s);
rc = osw.Collect(s);
endsubmit;
run;

Output 7.2.1  Singular Values of AIR Data

<table>
<thead>
<tr>
<th>Obs</th>
<th>NAME</th>
<th>SINGULARVALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>AIR</td>
<td>11857.417652</td>
</tr>
<tr>
<td>2</td>
<td>AIR</td>
<td>1026.3522048</td>
</tr>
<tr>
<td>3</td>
<td>AIR</td>
<td>1024.2750479</td>
</tr>
<tr>
<td>4</td>
<td>AIR</td>
<td>532.52047187</td>
</tr>
<tr>
<td>5</td>
<td>AIR</td>
<td>528.49637245</td>
</tr>
<tr>
<td>6</td>
<td>AIR</td>
<td>230.9898283</td>
</tr>
<tr>
<td>7</td>
<td>AIR</td>
<td>222.84650302</td>
</tr>
<tr>
<td>8</td>
<td>AIR</td>
<td>184.1226536</td>
</tr>
<tr>
<td>9</td>
<td>AIR</td>
<td>180.98734598</td>
</tr>
<tr>
<td>10</td>
<td>AIR</td>
<td>151.90912581</td>
</tr>
<tr>
<td>11</td>
<td>AIR</td>
<td>150.75654992</td>
</tr>
<tr>
<td>12</td>
<td>AIR</td>
<td>63.028435941</td>
</tr>
</tbody>
</table>
### SSA Results Table

<table>
<thead>
<tr>
<th>Obs</th>
<th>NAME</th>
<th>DATE</th>
<th>ORIGINAL</th>
<th>GROUPSUM</th>
<th>GROUP1</th>
<th>GROUP2</th>
<th>GROUP3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>AIR</td>
<td>JAN1949</td>
<td>112.04892977</td>
<td>-12.18036523</td>
<td>5.991732634</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>AIR</td>
<td>FEB1949</td>
<td>123.6083396</td>
<td>-12.56899832</td>
<td>9.7531247537</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>AIR</td>
<td>MAR1949</td>
<td>132.126907055</td>
<td>9.81461438</td>
<td>4.52564683</td>
<td>6.619714854</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>AIR</td>
<td>APR1949</td>
<td>129.12512079135</td>
<td>12.186780976</td>
<td>8.1780810532</td>
<td>-4.925999464</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>AIR</td>
<td>MAY1949</td>
<td>121.0475754989</td>
<td>12.67784605</td>
<td>13.944202575</td>
<td>-9.864517643</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>AIR</td>
<td>JUN1949</td>
<td>135.057442218</td>
<td>123.5194765</td>
<td>16.401623004</td>
<td>-5.175677331</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>AIR</td>
<td>JUL1949</td>
<td>148.05742656</td>
<td>124.57419013</td>
<td>15.444050369</td>
<td>5.1250250962</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>AIR</td>
<td>AUG1949</td>
<td>148.0574972107</td>
<td>125.8433187</td>
<td>11.263162709</td>
<td>10.883239665</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>AIR</td>
<td>SEP1949</td>
<td>136.204714613</td>
<td>127.1963396</td>
<td>4.4145298018</td>
<td>5.8408767297</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>AIR</td>
<td>OCT1949</td>
<td>119.047566679</td>
<td>128.57222357</td>
<td>-3.712022009</td>
<td>-5.345536567</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>AIR</td>
<td>NOV1949</td>
<td>104.017734754</td>
<td>129.85660189</td>
<td>-11.29875007</td>
<td>-11.52368557</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>AIR</td>
<td>DEC1949</td>
<td>118.067419512</td>
<td>131.04978498</td>
<td>-16.21934073</td>
<td>-6.156249129</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>AIR</td>
<td>JAN1950</td>
<td>115.04905306</td>
<td>12.13206287</td>
<td>-16.92871378</td>
<td>5.7367039728</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>AIR</td>
<td>MAR1950</td>
<td>141.039319034</td>
<td>134.78645215</td>
<td>-6.268071825</td>
<td>5.873501288</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>AIR</td>
<td>APR1950</td>
<td>135.01623557</td>
<td>136.363889038</td>
<td>2.3325213673</td>
<td>-7.534055042</td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>AIR</td>
<td>MAY1950</td>
<td>125.0394606621</td>
<td>138.09377166</td>
<td>10.217827924</td>
<td>-14.3655784</td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>AIR</td>
<td>JUN1950</td>
<td>149.048977318</td>
<td>140.12159096</td>
<td>15.512387846</td>
<td>-6.733206523</td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>AIR</td>
<td>JUL1950</td>
<td>170.06675791554</td>
<td>142.25166558</td>
<td>16.187264172</td>
<td>7.8189857801</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>AIR</td>
<td>AUG1950</td>
<td>170.038656644</td>
<td>144.42782619</td>
<td>11.731949689</td>
<td>14.676808566</td>
<td></td>
</tr>
<tr>
<td>21</td>
<td>AIR</td>
<td>SEP1950</td>
<td>158.0561924614</td>
<td>146.65047846</td>
<td>3.332983729</td>
<td>6.8359693047</td>
<td></td>
</tr>
<tr>
<td>22</td>
<td>AIR</td>
<td>OCT1950</td>
<td>133.0342230293</td>
<td>148.90016995</td>
<td>-6.56847052</td>
<td>-8.009382209</td>
<td></td>
</tr>
<tr>
<td>23</td>
<td>AIR</td>
<td>NOV1950</td>
<td>114.0215774911</td>
<td>151.25501811</td>
<td>-14.77224766</td>
<td>-14.90527935</td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>AIR</td>
<td>DEC1950</td>
<td>140.02811942587</td>
<td>153.76538413</td>
<td>-16.84394527</td>
<td>-7.002012995</td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>AIR</td>
<td>JAN1951</td>
<td>145.04642961748</td>
<td>156.30547135</td>
<td>-17.24506597</td>
<td>7.3692120981</td>
<td></td>
</tr>
<tr>
<td>26</td>
<td>AIR</td>
<td>FEB1951</td>
<td>150.0167338156</td>
<td>158.81959382</td>
<td>-10.9893708</td>
<td>13.904904823</td>
<td></td>
</tr>
<tr>
<td>27</td>
<td>AIR</td>
<td>MAR1951</td>
<td>178.0667343386</td>
<td>161.37642638</td>
<td>-1.39016224</td>
<td>6.7481712971</td>
<td></td>
</tr>
<tr>
<td>28</td>
<td>AIR</td>
<td>APR1951</td>
<td>163.015088803</td>
<td>163.779565351</td>
<td>8.5900245141</td>
<td>-6.858707717</td>
<td></td>
</tr>
<tr>
<td>29</td>
<td>AIR</td>
<td>MAY1951</td>
<td>172.0696756125</td>
<td>166.11313897</td>
<td>16.481483547</td>
<td>-12.91897127</td>
<td></td>
</tr>
<tr>
<td>30</td>
<td>AIR</td>
<td>JUN1951</td>
<td>178.032056454</td>
<td>168.19630112</td>
<td>19.88024517</td>
<td>-5.773970908</td>
<td></td>
</tr>
<tr>
<td>31</td>
<td>AIR</td>
<td>JUL1951</td>
<td>199.01674693194</td>
<td>170.35370624</td>
<td>18.599565179</td>
<td>7.793660521</td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>AIR</td>
<td>AUG1951</td>
<td>199.076312471</td>
<td>172.52517521</td>
<td>12.803541503</td>
<td>13.438008005</td>
<td></td>
</tr>
<tr>
<td>33</td>
<td>AIR</td>
<td>SEP1951</td>
<td>184.0385104518</td>
<td>174.78392064</td>
<td>4.008576656</td>
<td>5.058670627</td>
<td></td>
</tr>
<tr>
<td>34</td>
<td>AIR</td>
<td>OCT1951</td>
<td>162.01619633078</td>
<td>177.03885054</td>
<td>-5.895869896</td>
<td>-9.178622864</td>
<td></td>
</tr>
<tr>
<td>35</td>
<td>AIR</td>
<td>NOV1951</td>
<td>146.0505429936</td>
<td>179.29554014</td>
<td>-14.46573436</td>
<td>-14.28450643</td>
<td></td>
</tr>
<tr>
<td>36</td>
<td>AIR</td>
<td>DEC1951</td>
<td>166.05747702782</td>
<td>181.51361867</td>
<td>-16.93143982</td>
<td>-4.40515039</td>
<td></td>
</tr>
<tr>
<td>37</td>
<td>AIR</td>
<td>JAN1952</td>
<td>171.04758914596</td>
<td>183.74792826</td>
<td>-19.98132376</td>
<td>10.823171454</td>
<td></td>
</tr>
<tr>
<td>38</td>
<td>AIR</td>
<td>FEB1952</td>
<td>180.0485142029</td>
<td>185.97629345</td>
<td>-15.4360557</td>
<td>15.373972533</td>
<td></td>
</tr>
<tr>
<td>39</td>
<td>AIR</td>
<td>MAR1952</td>
<td>193.01846307809</td>
<td>188.11021691</td>
<td>-7.181860498</td>
<td>3.6749176279</td>
<td></td>
</tr>
<tr>
<td>40</td>
<td>AIR</td>
<td>APR1952</td>
<td>181.07744639</td>
<td>190.3807623</td>
<td>3.5024461281</td>
<td>-12.16606076</td>
<td></td>
</tr>
</tbody>
</table>
Output 7.2.3  SSA Grouping Results of AIR Data

SSA Grouping Results Table

<table>
<thead>
<tr>
<th>Obs</th>
<th>GROUPINDEX</th>
<th>WINDOWINDEX</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>3</td>
<td>5</td>
</tr>
</tbody>
</table>
**Output 7.2.4** W-Correlations of AIR Data (Partial Output)

<table>
<thead>
<tr>
<th>Obs</th>
<th>WINDOWINDEX1</th>
<th>WINDOWINDEX2</th>
<th>WCORRELATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>1</td>
<td>0.0013287428</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>1</td>
<td>0.0011475977</td>
</tr>
<tr>
<td>5</td>
<td>3</td>
<td>2</td>
<td>0.999508341</td>
</tr>
<tr>
<td>6</td>
<td>3</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>4</td>
<td>1</td>
<td>0.000651801</td>
</tr>
<tr>
<td>8</td>
<td>4</td>
<td>2</td>
<td>0.0159495678</td>
</tr>
<tr>
<td>9</td>
<td>4</td>
<td>3</td>
<td>0.0125585197</td>
</tr>
<tr>
<td>10</td>
<td>4</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>11</td>
<td>5</td>
<td>1</td>
<td>0.000390683</td>
</tr>
<tr>
<td>12</td>
<td>5</td>
<td>2</td>
<td>0.0081218368</td>
</tr>
<tr>
<td>13</td>
<td>5</td>
<td>3</td>
<td>0.006555062</td>
</tr>
<tr>
<td>14</td>
<td>5</td>
<td>4</td>
<td>0.985943964</td>
</tr>
<tr>
<td>15</td>
<td>5</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>16</td>
<td>6</td>
<td>1</td>
<td>0.0008707427</td>
</tr>
<tr>
<td>17</td>
<td>6</td>
<td>2</td>
<td>0.0042687371</td>
</tr>
<tr>
<td>18</td>
<td>6</td>
<td>3</td>
<td>0.0038215045</td>
</tr>
<tr>
<td>19</td>
<td>6</td>
<td>4</td>
<td>0.0084034171</td>
</tr>
<tr>
<td>20</td>
<td>6</td>
<td>5</td>
<td>0.011530425</td>
</tr>
<tr>
<td>21</td>
<td>6</td>
<td>6</td>
<td>1</td>
</tr>
<tr>
<td>22</td>
<td>7</td>
<td>1</td>
<td>0.000433709</td>
</tr>
<tr>
<td>23</td>
<td>7</td>
<td>2</td>
<td>0.0007856255</td>
</tr>
<tr>
<td>24</td>
<td>7</td>
<td>3</td>
<td>0.0006613125</td>
</tr>
<tr>
<td>25</td>
<td>7</td>
<td>4</td>
<td>0.0048383318</td>
</tr>
<tr>
<td>26</td>
<td>7</td>
<td>5</td>
<td>0.0059614384</td>
</tr>
<tr>
<td>27</td>
<td>7</td>
<td>6</td>
<td>0.9876010328</td>
</tr>
<tr>
<td>28</td>
<td>7</td>
<td>7</td>
<td>1</td>
</tr>
<tr>
<td>29</td>
<td>8</td>
<td>1</td>
<td>0.0007109083</td>
</tr>
<tr>
<td>30</td>
<td>8</td>
<td>2</td>
<td>0.0038632385</td>
</tr>
<tr>
<td>31</td>
<td>8</td>
<td>3</td>
<td>0.0028463689</td>
</tr>
<tr>
<td>32</td>
<td>8</td>
<td>4</td>
<td>0.006213973</td>
</tr>
<tr>
<td>33</td>
<td>8</td>
<td>5</td>
<td>0.0021631625</td>
</tr>
<tr>
<td>34</td>
<td>8</td>
<td>6</td>
<td>0.0264797592</td>
</tr>
<tr>
<td>35</td>
<td>8</td>
<td>7</td>
<td>0.0555690426</td>
</tr>
<tr>
<td>36</td>
<td>8</td>
<td>8</td>
<td>1</td>
</tr>
<tr>
<td>37</td>
<td>9</td>
<td>1</td>
<td>0.0000574587</td>
</tr>
<tr>
<td>38</td>
<td>9</td>
<td>2</td>
<td>0.0007623599</td>
</tr>
<tr>
<td>39</td>
<td>9</td>
<td>3</td>
<td>0.0006672876</td>
</tr>
<tr>
<td>40</td>
<td>9</td>
<td>4</td>
<td>0.0003962508</td>
</tr>
</tbody>
</table>
Output 7.2.5 SSA Results of AIR Data

SSA Results Plot

Original

Time ID


Original Group Sum Group 1 Group 2 Group 3
Example 7.3: Run a Singular Spectrum Analysis of a User-Selected Group

This example runs a singular spectrum analysis of a user-selected group. Output 7.3.2 shows the resulting groupings together with the original data. Output 7.3.1 shows the values of each components.

```sas
proc tsmodel data=mycas.air outobj=(os=mycas.analytic (replace=YES));
id date interval=month;
var air;
require ssa;
submit;
array group3[2]/nosymbols;group3[1]=6;group3[2]=7;
declare object s(ssa);
declare object os(outssa('NGROUP',3));
rc = s.Initialize();
rc = s.SetY(air);
rc = s.SetOption('METHOD','GROUPS');
rc = s.SetOption('LENGTH',12);
rc = s.addgroup(group1);
rc = s.addgroup(group2);
rc = s.addgroup(group3);
rc = s.Run();
rc = os.Collect(s);
endsubmit;
run;
```
236 F Chapter 7: Singular Spectrum Analysis Package

Output 7.3.1 SSA Results of AIR Data (Partial Output)

SSA Results Table
Obs _NAME_

DATE ORIGINAL GROUPSUM

GROUP1

GROUP2

GROUP3

1 AIR

JAN1949

112 111.23641968 107.86146454 5.9917323634 -2.616777218

2 AIR

FEB1949

118 121.09754638 113.40770921 9.7531247537 -2.063287577

3 AIR

MAR1949

132 129.12383717 121.50939907 4.6196714854 2.9947666156

4 AIR

APR1949

129

128.07342 130.04589081 -4.925099464 2.9526286526

5 AIR

MAY1949

121

124.2807749 136.62206663 -9.864517643 -2.476774077

6 AIR

JUN1949

135 131.86362423 139.92109951 -5.175677331 -2.881797945

7 AIR

JUL1949

148 147.99219928

8 AIR

AUG1949

148 150.88822784 137.10648141 10.883239665 2.8985067677

9 AIR

SEP1949

136 134.15922433

10 AIR

OCT1949

119 116.53415987 124.86020336 -5.345536567 -2.980506919

11 AIR

NOV1949

104 110.46745746 118.55785911 -11.52368557 3.4332839177

12 AIR

DEC1949

118 111.69267624 114.83044425 -6.156249129 3.0184811272

13 AIR

JAN1950

115 117.25987801 115.20424909 5.7367039728 -3.681075048

14 AIR

FEB1950

126 128.99347656 120.16159642 12.279965308 -3.448085171

15 AIR

MAR1950

141

16 AIR

APR1950

135 134.46042538 138.69641074 -7.534055042 3.2980696802

17 AIR

MAY1950

125

18 AIR

JUN1950

149 144.99774617 155.63397881 -6.733205623 -3.903027013

19 AIR

JUL1950

170 169.05567533 158.43892976 7.8189857801 2.7977597902

20 AIR

AUG1950

170 174.92861238 156.15977587 14.676880566 4.0919559373

21 AIR

SEP1950

158 154.72195068 149.98327684 6.8359693047 -2.097295463

22 AIR

OCT1950

133

23 AIR

NOV1950

114 122.89135368 136.48277045 -14.90527935 1.3138625827

24 AIR

DEC1950

140 132.44955561 135.12143886 -7.002012995 4.3301297402

25 AIR

JAN1951

145 145.21484511 139.06040538 7.3692120981 -1.214772372

26 AIR

FEB1951

150 157.39777491 147.82975674 13.904094823 -4.336076655

27 AIR

MAR1951

178 168.02026692 159.98626639

28 AIR

APR1951

163 169.73251162 172.36958802 -6.858707717 4.2216313139

29 AIR

MAY1951

172

30 AIR

JUN1951

178 178.63382966 188.07654364 -5.773979098 -3.668734876

31 AIR

JUL1951

199 197.65140141 188.95327142

32 AIR

AUG1951

199 201.73993186 185.32511671 13.438008005 2.9768071435

33 AIR

SEP1951

184 182.60845062 178.79247812 5.0585670627 -1.242594561

34 AIR

OCT1951

162 159.32048484 171.14295364 -9.178622864 -2.643845933

35 AIR

NOV1951

146 152.26010009 164.82980579 -14.28450643 1.7148007332

36 AIR

DEC1951

166 159.86509052 161.88217886 -4.405151039 2.3880627022

37 AIR

JAN1952

171 172.55759638

38 AIR

FEB1952

180 183.86916496 170.54023775 15.373972533 -2.045045323

39 AIR

MAR1952

193 187.07098551 180.92835641 3.6749216729 2.4677074278

40 AIR

APR1952

181 183.83635302 193.94052435 -12.16606076 2.0618894194

140.0182405 5.1250250962 2.8489336817
131.6108694 5.8408767297 -3.292521804

137.7085761 128.51838033 5.8735501288 3.3166456463
130.2462494 148.31164461

129.910353

-14.3655784 -3.699816806

142.3316826 -8.009382209 -4.411947394

6.748117291 1.2858832421

169.0982717 182.59462252 -12.91897127 -0.577379548
7.793660521 0.9044694743

163.7659745 10.823171454 -2.031549576


Output 7.3.2 SSA Results of AIR Data

References


Chapter 8
Subspace Tracking Package

Contents

Overview: SST Package ............................................................... 239
SST Package Summary ............................................................ 240
  Return Codes ................................................................. 240
SST Object ........................................................................ 241
  SST Synopsis ............................................................... 243
  SST Methods ............................................................... 243
OUTSST Object ................................................................. 245
  OUTSST Synopsis ......................................................... 246
  OUTSST Methods ......................................................... 247
Details ............................................................................. 247
  Subspace Tracking ........................................................ 247
Examples: SST Package .......................................................... 248
  Using CAS Sessions and CAS Engine Librefs ......................... 248
  Example 8.1: Perform Subspace Tracking and Collect the Output 249
  Example 8.2: Detect the Subspace Change ............................ 253

Overview: SST Package

The subspace tracking (SST) package contains a set of time series analysis functions that can be used as part of the programming statements in the TSMODEL procedure. This package provides a flexible way to analyze and decompose time series within the procedure.

NOTE: Each function in this chapter has a prefix of “SST”; however, the prefixes are omitted in descriptions for better readability. The mycas libref in the examples refers to the CAS library that is linked to a caslib. All the examples in this chapter assume that your CAS engine libref is named mycas, but you can substitute any appropriately defined CAS engine libref. For more information about CAS engine librefs, see SAS Cloud Analytic Services: User’s Guide.
SST Package Summary

Table 8.1 summarizes the objects in the SST package.

<table>
<thead>
<tr>
<th>Object</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SST</td>
<td>Perform subspace tracking of multiple time series</td>
</tr>
<tr>
<td>OUTSST</td>
<td>Collect output from an SST object</td>
</tr>
</tbody>
</table>

Return Codes

Table 8.2 shows the return code (rc in method statements) status values that are used in this package. These values are returned after a method that is associated with an object is called; they can help determine whether the method executed successfully.

<table>
<thead>
<tr>
<th>Status</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt; 0</td>
<td>An unrecoverable error occurred. No result was produced.</td>
</tr>
<tr>
<td>= 0</td>
<td>Unconditional success. The requested action was completed and a normal result was produced.</td>
</tr>
<tr>
<td>&gt; 0</td>
<td>Conditional success or warning. A result was produced subject to conditions.</td>
</tr>
</tbody>
</table>

Upon returning a negative return code, most methods in the SST package objects also write a message to the output log that explains the causes of the related failure. These messages provide useful information for debugging a user program. In the TSMODEL procedure, the output log is stored in the CAS table that is specified in the OUTLOG= option in the PROC TSMODEL statement. For more information about how to enable and configure logging and about how to access the output log after an invocation of the TSMODEL procedure, see Chapter 11, “The TSMODEL Procedure” (SAS Visual Forecasting: Forecasting Procedures).
SST Object

The SST object tracks the principal subspace of multiple time series variables to be used as input for the other SST package objects.

Table 8.3 summarizes the methods that are associated with the SST object.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AddX</td>
<td>Add an input time series</td>
</tr>
<tr>
<td>Initialize</td>
<td>Initialize an SST instance</td>
</tr>
<tr>
<td>Run</td>
<td>Run the subspace tracking</td>
</tr>
<tr>
<td>SetOption</td>
<td>Set a named option</td>
</tr>
<tr>
<td>SetVars</td>
<td>Set the input time series</td>
</tr>
</tbody>
</table>

Figure 8.1 diagrams the methods of the SST object.
Figure 8.1: SST Data Flow

- **initialize**
  - SST
  - collect
  - OUTSST
  - outObj={}

- **SST**
  - Numeric Array
  - setVars
  - addX
  - setOption

- **OUTSST**
  - Table

- **OUTSST**

**Functions**:
- COVFORGETFACTOR
- EIGVALTOLCUMULATIVE
- MAXPRINCIPAL
- MEANFORGETFACTOR
SST Synopsis

DECLARE OBJECT obj (SST) ;

Method syntax, in order of typical usage:

\[
\begin{align*}
rc &= \text{obj}.\text{Initialize} () ; \\
rc &= \text{obj}.\text{SetVars} (\text{TimeSeries <,TimeSeries,…>} ) ; \\
rc &= \text{obj}.\text{AddX} (\text{TimeSeries}) ; \\
rc &= \text{obj}.\text{SetOption} (\text{‘Name’, Value}) ; \\
rc &= \text{obj}.\text{Run} () ; \\
\end{align*}
\]

SST Methods

SST.AddX Method

\[
rc = \text{obj}.\text{AddX} (\text{TimeSeries}) ;
\]

Adds a time series array (TimeSeries) for the SST instance. Each call to the AddX method adds the specified variable to the SST instance. You can call this method as many times as needed to specify the input variables for subspace tracking.

Input Arguments

You must specify the following input argument:

\[\text{TimeSeries} \quad \text{specifies a numeric array that contains a time series for the SST instance.}\]

SST.Initialize Method

\[
rc = \text{obj}.\text{Initialize} () ;
\]

Initializes an SST instance to an empty state. You must call this method before specifying the time series arrays and other attributes for the SST instance.

Arguments

There are no arguments associated with this method.

SST.SetOption Method

\[
rc = \text{obj}.\text{SetOption} (\text{‘Name’, Value}) ;
\]

Sets a named option for the SST instance.

Input Arguments

You can specify one of the following ‘Names’ and its associated Value:

- **‘COVFORGETFACTOR’**
  takes a numeric Value between 0 and 1 that specifies the parameter of the forgetting factor used to update the covariance matrix. The default is 0.5.
'EIGVALTOLCUMULATIVE' takes a positive numeric Value less than or equal to 1 that specifies the parameter of the threshold on the cumulative rate of eigenvalues. The default is 1.

'MAXPRINCIPAL' takes a positive integer Value less than or equal to the number of time series used for the subspace tracking. This specifies the maximum rank of the subspace for computing. The default is the number of input time series.

'MEANFORGETFACTOR' takes a numeric Value between 0 and 1 that specifies the parameter of the forgetting factor used to update the mean. The default is 0.1.

**SST.SetVars Method**

```cpp
rc = obj.SetVars (TimeSeries,<,TimeSeries,...>) ;
```

Adds multiple time series (`TimeSeries,<,TimeSeries,...>`) to the SST instance. You can use one call to the SetVars method to specify as many input variables as you need for subspace tracking.

**Input Arguments**

You must specify the following input argument:

*TimeSeries* specifies a numeric array that contains a time series for the SST instance.

**SST.Run Method**

```cpp
rc = obj.Run () ;
```

Runs the SST object to track the principal subspace of a sequence of observation vectors. Upon successful completion, various results can be extracted from the SST object.

**Arguments**

There are no arguments associated with this method.
OUTSST Object

The OUTSST object collects output from SST object.

Table 8.4 summarizes the methods that are associated with the OUTSST object.

Table 8.4  Methods of the OUTSST Object

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Collect</td>
<td>Collect the results of an SST object</td>
</tr>
<tr>
<td>nrows</td>
<td>Get the current row count from the OUTSST object</td>
</tr>
</tbody>
</table>

Figure 8.2 diagrams the methods of the OUTSST object.
Figure 8.2 OUTSST Data Flow

DECLARE OBJECT obj (OUTSST) ;

Method syntax, in order of typical usage:

rc=obj.Collect () ;
nrows=obj.nrows () ;
OUTSST Methods

OUTSST.Collect Method

```c
rc = obj.Collect();
```

Collects the residual and rank of each principal subspace from an SST object and saves the results to a CAS table.

**Arguments**

There are no arguments associated with this method.

OUTSST.nrows Attribute

```c
nrows = obj.nrows();
```

Gets the current row count from the OUTSST instance.

**Arguments**

There are no arguments associated with this method.

Details

Subspace Tracking

Suppose that the data contain a sequence of $n \times 1$ vectors: $x(t)$. Subspace tracking (SST) estimates the covariance matrix for each vector, $x(t)$, and then computes the first $p$ principal eigenvectors of the covariance matrix. For each iteration at time $t$, the covariance matrix, $C(t)$, is obtained by

$$
\mu(t) = (1 - \alpha)\mu(t - 1) + \alpha x(t)
$$

$$
C(t) = (1 - \beta)C(t - 1) + \beta(x(t) - \mu(t))(x(t) - \mu(t))^T
$$

where $\alpha$ and $\beta$ are the forgetting factors whose values are predetermined to be between zero and one, respectively. And the first $p$ principal eigenvectors, $W(t)$, can be obtained by the eigendecomposition of the covariance matrix.

Rank of the Principal Subspace

The principal subspace is defined as the span of the $p$ principal eigenvectors. Therefore, the rank of the principal subspace is $p$, which is usually less than $n$. And $p$ can be limited by 'EIGVALTOLCUMULATIVE' and 'MAXPRINCIPAL' in the SST.SetOption method. The ranks of each iteration are usually the same after a warm-up period. Therefore, the change in the rank values indicates that this observation might contain an outlier or be a sign of the subspace change. Because the length of the warm-up period can vary across application problems, experiments are required in order to determine the length.
Projection Angle of the Principal Subspace

The normalized projection angle of the subspace tracking method is calculated as

\[
\arccos \left( \frac{\| W(t)^T x(t) \|_2}{\| x(t) \|_2} \right) \left( \frac{2}{\pi} \right)
\]

After a warm-up period, the series of projection angles is relatively stable. A significant change in the projection angle can indicate a subspace change or the existence of an outlier in the observation.

Handling a Missing Value

If an observation contains a missing value at time \( t \), then the corresponding principal eigenvectors are taken from the previous iteration, \( W(t - 1) \). Therefore, the rank of the principal subspace is the same as the one from the previous iteration. The residual is set to the missing value.

Examples: SST Package

Using CAS Sessions and CAS Engine Librefs

SAS Cloud Analytic Services (CAS) is the analytic server and associated cloud services in SAS Viya. This section describes how to create a CAS session and set up a CAS engine libref that you can use to connect to the CAS session. It assumes that you have a CAS server already available; contact your system administrator if you need help starting and terminating a server. This CAS server is identified by specifying the host on which it runs and the port on which it listens for communications. To simplify your interactions with this CAS server, the host information and port information for the server are stored as SAS option values that are retrieved automatically whenever this CAS server needs to be accessed. You can examine the host and port values for the server at your site by using the following statements:

```
proc options option=(CASHOST CASPORT);
run;
```

In addition to starting a CAS server, your system administrator might also have created a CAS session and a CAS engine libref for your use. You can define your own sessions and CAS engine librefs that connect to the CAS server as shown in the following statements:

```
cas mysess;
libname mycas cas sessref=mysess;
```

The CAS statement creates the CAS session named `mysess`, and the LIBNAME statement creates the `mycas` CAS engine libref that you use to connect to this session. It is not necessary to explicitly name the CASHOST and CASPORT of the CAS server in the CAS statement, because these values are retrieved from the corresponding SAS option values.

If you have created the `mysess` session, you can terminate it by using the TERMINATE option in the CAS statement as follows:
Example 8.1: Perform Subspace Tracking and Collect the Output

This example performs subspace tracking in an SST object and uses an OUTSST object to collect the output. The following statements create a data table of a low-rank matrix whose rank is two:

```sas
%macro createData(dataname=, n=, m=, r=);
  data &dataname (keep = x:);
  retain id x:;
  array B[&r,&m]; /* dimensions: rank, number of variables */
  array A[&r]; /* dimension: rank */
  array x[&m] x1-x&m; /* dimension: number of variables */
  call streaminit(1);
  
  do row = 1 to &r;
    do col = 1 to &m;
      B[row,col] = rand("Normal");
    end;
  end;
  
  do obs = 1 to &n;
    do j = 1 to &r;
      A[j] = rand('Normal');
    end;
    id = obs;
    
    do k = 1 to &m;
      x[k] = 0;
      do j = 1 to &r;
        x[k] = x[k] + A[j]*B[j,k];
      end;
    end;
    output;
  end;
run;
%mend;

%createData(dataname=a, n=500, m=6, r=2);

data mycas.lmmat;
  id = _N_; set a;
run;
```

For more information about the CAS statement and the LIBNAME statement, see *SAS Cloud Analytic Services: User’s Guide*. For general information about CAS and CAS sessions, see *SAS Cloud Analytic Services: Fundamentals*. 
The following SAS code uses the TSMODEL procedure to submit a program that uses the SST object and the OUTSST object. The ID and VAR statements specify the id variable and input variables for the SST instance, respectively. The REQUIRE statement loads the SST package and installs its classes (SST class) so that the program can use the package. The DECLARE OBJECT statements define the SST instance named s and the OUTSST named os. Method calls of the object instance use the dot notation. The calls s.Initialize, s.SetVars, s.SetOption, and s.Run are performed sequentially, with a status check following each call to ensure that the method call was successful. If any call fails, the program stops execution. Following a successful s.Run, the call os.Collect fetches the results of the s object.

```sas
proc tsmodel data=mycas.lmmat
    outobj=(os=mycas.outputobj (replace=yes));
    id id interval=obs;
    var x1 x2 x3 x4 x5 x6;
require sst;
submit;
declare object s(sst);
declare object os(outsst);
    rc = s.Initialize();
    rc = s.SetVars(x1, x2, x3, x4, x5, x6);
    rc = s.SetOption('MAXPRINCIPAL', 2);
    rc = s.SetOption('COVFORGETFACTOR', 0.5);
    rc = s.SetOption('MEANFORGETFACTOR', 0);
    rc = s.Run();
    /* collect the output*/
    rc = os.Collect(s);
endsubmit;
run;
```

Output 8.1.1 shows the values of normalized projection angle and rank for each subspace. Output 8.1.2 plots the projection angles from Output 8.1.1. This shows that after a few warm-up iterations (two iterations), the resulting projection angles are very small (less than 1E–7).
## Output 8.1.1  SST Results (Partial Output)

### SST Results Table

<table>
<thead>
<tr>
<th>Obs id</th>
<th>PROJANGLE</th>
<th>RANK</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.4872969673</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>0.7749161402</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>9.4863738E-9</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>6</td>
<td>9.4863738E-9</td>
<td>2</td>
</tr>
<tr>
<td>7</td>
<td>1.3415759E-8</td>
<td>2</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>9</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>12</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>13</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>14</td>
<td>1.3415759E-8</td>
<td>2</td>
</tr>
<tr>
<td>15</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>16</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>17</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>18</td>
<td>1.8972748E-8</td>
<td>2</td>
</tr>
<tr>
<td>19</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>20</td>
<td>9.4863738E-9</td>
<td>2</td>
</tr>
<tr>
<td>21</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>22</td>
<td>1.8972748E-8</td>
<td>2</td>
</tr>
<tr>
<td>23</td>
<td>9.4863738E-9</td>
<td>2</td>
</tr>
<tr>
<td>24</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>25</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>26</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>27</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>28</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>29</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>30</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>31</td>
<td>1.6430881E-8</td>
<td>2</td>
</tr>
<tr>
<td>32</td>
<td>1.3415759E-8</td>
<td>2</td>
</tr>
<tr>
<td>33</td>
<td>1.3415759E-8</td>
<td>2</td>
</tr>
<tr>
<td>34</td>
<td>1.3415759E-8</td>
<td>2</td>
</tr>
<tr>
<td>35</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>36</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>37</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>38</td>
<td>2.3236775E-8</td>
<td>2</td>
</tr>
<tr>
<td>39</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>40</td>
<td>0</td>
<td>2</td>
</tr>
</tbody>
</table>
Output 8.1.2 SST Projection Angle Results

SST Projection Angle Plot

Projection Angle

Time ID
Example 8.2: Detect the Subspace Change

This example shows how to detect the subspace change by using the output.

The following statements create a data table that consists of two matrices of different rank. The first half of the data table consists of a matrix of rank two, and the second half consists of a matrix of rank four.

```sas
%createData(dataname=a,n=500,m=6,r=2);
%createData(dataname=b,n=500,m=6,r=4);

data mycas.lmmat;
  id = _N_;  
  set a b;
run;
```

As in the previous example, the following SAS code uses the TSMODEL procedure to submit a program that uses the SST object and the OUTSST object. Because the maximum rank of the data table is four, 'MAXPRINCIPAL' in the SST.SetOption method is set to 4.

```sas
proc tsmodel data=mycas.lmmat
  outobj=(os=mycas.outputobj (replace=yes));
  id id interval=obs;
  var x1 x2 x3 x4 x5 x6;
  require sst;
  submit;
    declare object s(sst);
    declare object os(outsst);
    rc = s.Initialize();
    rc = s.SetVars(x1, x2, x3, x4, x5, x6);
    rc = s.SetOption('MAXPRINCIPAL', 4);
    rc = s.SetOption('COVFORGETFACTOR', 0.5);
    rc = s.SetOption('MEANFORGETFACTOR', 0);
    rc = s.Run();
    /* collect the output*/
    rc = os.Collect(s);
  endsubmit;
run;
```

Output 8.2.1 shows the values of normalized projection angle and rank for observations between 481 and 520. Output 8.2.2 plots the projection angles from Output 8.2.1. The results table and graph show that the projection angle and residual change at the 501st observation. Then these values are stable after a few more iterations.
**Output 8.2.1** SST Results (Partial Output)

**SST Results Table**

<table>
<thead>
<tr>
<th>Obs</th>
<th>id</th>
<th>PROJANGLE</th>
<th>RANK</th>
</tr>
</thead>
<tbody>
<tr>
<td>481</td>
<td>481</td>
<td>2.1212177E-8</td>
<td>2</td>
</tr>
<tr>
<td>482</td>
<td>482</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>483</td>
<td>483</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>484</td>
<td>484</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>485</td>
<td>485</td>
<td>1.6430881E-8</td>
<td>2</td>
</tr>
<tr>
<td>486</td>
<td>486</td>
<td>1.3415759E-8</td>
<td>2</td>
</tr>
<tr>
<td>487</td>
<td>487</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>488</td>
<td>488</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>489</td>
<td>489</td>
<td>1.3415759E-8</td>
<td>2</td>
</tr>
<tr>
<td>490</td>
<td>490</td>
<td>9.4863738E-9</td>
<td>2</td>
</tr>
<tr>
<td>491</td>
<td>491</td>
<td>1.6430881E-8</td>
<td>2</td>
</tr>
<tr>
<td>492</td>
<td>492</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>493</td>
<td>493</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>494</td>
<td>494</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>495</td>
<td>495</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>496</td>
<td>496</td>
<td>9.4863738E-9</td>
<td>2</td>
</tr>
<tr>
<td>497</td>
<td>497</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>498</td>
<td>498</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>499</td>
<td>499</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>500</td>
<td>500</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>501</td>
<td>501</td>
<td>0.4387789436</td>
<td>3</td>
</tr>
<tr>
<td>502</td>
<td>502</td>
<td>0.0109340723</td>
<td>4</td>
</tr>
<tr>
<td>503</td>
<td>503</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>504</td>
<td>504</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>505</td>
<td>505</td>
<td>1.3415759E-8</td>
<td>4</td>
</tr>
<tr>
<td>506</td>
<td>506</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>507</td>
<td>507</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>508</td>
<td>508</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>509</td>
<td>509</td>
<td>9.4863738E-9</td>
<td>4</td>
</tr>
<tr>
<td>510</td>
<td>510</td>
<td>1.3415759E-8</td>
<td>4</td>
</tr>
<tr>
<td>511</td>
<td>511</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>512</td>
<td>512</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>513</td>
<td>513</td>
<td>2.3236775E-8</td>
<td>4</td>
</tr>
<tr>
<td>514</td>
<td>514</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>515</td>
<td>515</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>516</td>
<td>516</td>
<td>9.4863738E-9</td>
<td>4</td>
</tr>
<tr>
<td>517</td>
<td>517</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>518</td>
<td>518</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>519</td>
<td>519</td>
<td>1.6430881E-8</td>
<td>4</td>
</tr>
<tr>
<td>520</td>
<td>520</td>
<td>1.3415759E-8</td>
<td>4</td>
</tr>
</tbody>
</table>
Example 8.2: Detect the Subspace Change

**Output 8.2.2** SST Projection Angle Results

![SST Projection Angle Plot](image-url)
# Chapter 9
## Time Filters Package

## Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Overview: Time Filters Package</td>
<td>258</td>
</tr>
<tr>
<td>Digital Filtering Overview</td>
<td>258</td>
</tr>
<tr>
<td>Time Filters Package Summary</td>
<td>264</td>
</tr>
<tr>
<td>Return Codes</td>
<td>265</td>
</tr>
<tr>
<td>DFORDER Object</td>
<td>265</td>
</tr>
<tr>
<td>DFORDER Synopsis</td>
<td>269</td>
</tr>
<tr>
<td>DFORDER Methods</td>
<td>269</td>
</tr>
<tr>
<td>DFDESIGN Object</td>
<td>271</td>
</tr>
<tr>
<td>DFDESIGN Synopsis</td>
<td>272</td>
</tr>
<tr>
<td>DFDESIGN Methods</td>
<td>273</td>
</tr>
<tr>
<td>DFFILTER Object</td>
<td>274</td>
</tr>
<tr>
<td>DFFILTER Synopsis</td>
<td>275</td>
</tr>
<tr>
<td>DFFILTER Methods</td>
<td>276</td>
</tr>
<tr>
<td>DFSOSFILTER Object</td>
<td>277</td>
</tr>
<tr>
<td>DFSOSFILTER Synopsis</td>
<td>279</td>
</tr>
<tr>
<td>DFSOSFILTER Methods</td>
<td>279</td>
</tr>
<tr>
<td>DFCEPSTRUM Object</td>
<td>280</td>
</tr>
<tr>
<td>DFCEPSTRUM Synopsis</td>
<td>282</td>
</tr>
<tr>
<td>DFCEPSTRUM Methods</td>
<td>282</td>
</tr>
<tr>
<td>DFICEPSTRUM Object</td>
<td>284</td>
</tr>
<tr>
<td>DFICEPSTRUM Synopsis</td>
<td>285</td>
</tr>
<tr>
<td>DFICEPSTRUM Methods</td>
<td>286</td>
</tr>
<tr>
<td>MRR Object</td>
<td>287</td>
</tr>
<tr>
<td>MRR Synopsis</td>
<td>288</td>
</tr>
<tr>
<td>MRR Methods</td>
<td>289</td>
</tr>
<tr>
<td>OUTDFORDER Object</td>
<td>290</td>
</tr>
<tr>
<td>OUTDFORDER Synopsis</td>
<td>291</td>
</tr>
<tr>
<td>OUTDFORDER Methods</td>
<td>291</td>
</tr>
<tr>
<td>OUTDFDESIGN Object</td>
<td>292</td>
</tr>
<tr>
<td>OUTDFDESIGN Synopsis</td>
<td>293</td>
</tr>
<tr>
<td>OUTDFDESIGN Methods</td>
<td>293</td>
</tr>
<tr>
<td>OUTDFFILTER Object</td>
<td>295</td>
</tr>
<tr>
<td>OUTDFFILTER Synopsis</td>
<td>296</td>
</tr>
<tr>
<td>OUTDFFILTER Methods</td>
<td>296</td>
</tr>
<tr>
<td>OUTDFSOSFILTER Object</td>
<td>297</td>
</tr>
</tbody>
</table>
Overview: Time Filters Package

The time filters package (TIMFIL) provides objects that enable you to perform various types of filtering and aggregation on time series data. You can use these objects as part of the programming statements in the TSMODEL procedure in SAS Visual Forecasting.

Digital Filtering Overview

Information Representation in the Time Domain and the Frequency Domain

Digital filtering is an important part of digital signal processing (DSP). Signals usually originate as sensory data from the real world: seismic vibrations, visual images, sound waves, and so on. Digital signal processing uses various algorithms and techniques to manipulate these signals in digital form. DSP applications include image enhancement, speech recognition, data compression, and others.

To understand digital filtering, it is important to understand how a digital signal is represented in the time domain and frequency domain. In the time domain, a *sample* of a digital signal is defined as the signal’s amplitude at the point in time at which it occurs. Each sample in the signal indicates both what is happening at that instant (the event) and the level of the event. Time-domain representation is the waveform of a signal that is usually seen for real-world data.

In contrast, information that are represented in the frequency domain is more challenging to understand. A signal’s frequency-domain representation is related to the concept of using “trigonometric sums” (that is, sums of harmonically related sines and cosines or periodic complex exponentials) to describe periodic
Digital Filtering Overview

phenomena within a signal (Oppenheim, Willsky, and Nawab 1997). The most commonly used frequency-domain representation is the Fourier transform of a signal. The discrete-time Fourier transform of a digital signal $x(n)$ is defined as

$$X(e^{j\omega}) = \sum_{n=-\infty}^{\infty} x(n)e^{-j\omega n}$$

To understand the physical meaning of the Fourier transform, start with a simple periodic signal, the complex exponential signal $x(n)$,

$$x(n) = e^{j\omega_0 n}$$

where $x(n)$ is periodic with a fundamental period $N$ and $\omega_0 = \frac{2\pi}{N}$ is the fundamental frequency of the signal. The Fourier transform of $x(n)$, by definition, is a periodic impulse train.

$$X(e^{j\omega}) = \sum_{n=-\infty}^{\infty} e^{j\omega_0 n}e^{-j\omega n}$$

$$= \sum_{k=-\infty}^{\infty} 2\pi \delta(\omega - \omega_0 + 2\pi k) \quad (9.1)$$

where $\delta(x)$ is a unit impulse function, which is defined as

$$\delta(x) = \begin{cases} 1, & x = 0 \\ 0, & x \neq 0 \end{cases}$$

Figure 9.1 shows the Fourier transform of $x(n) = e^{j\omega_0 n}$.

**Figure 9.1** Fourier Transform of the Complex Exponential Signal

Because by definition the discrete-time Fourier transform $X(e^{j\omega})$ is periodic in $\omega$ with period $2\pi$, $X(e^{j\omega})$ for $\omega \in [0, 2\pi)$ is referred to mostly as the Fourier transform of a signal $x(n)$. The Fourier transform of $x(n) = e^{j\omega_0 n}$ when $\omega \in [0, 2\pi)$ is a single impulse in the frequency domain as shown in Figure 9.2.

**Figure 9.2** Fourier Transform of the Complex Exponential Signal in One Period
It has been shown that any real discrete-time signal $x(n)$ with length $N$ can be represented by a linear combination of complex exponential signals as

$$x(n) = \sum_{k=0}^{N-1} a_k e^{j\omega_0 n} = \sum_{k=0}^{N-1} a_k e^{j k (2\pi/N)n}$$

where $\omega_0 = \frac{2\pi}{N}$, and $a_k$ ($k = 0, 1, \ldots, N - 1$) are the Fourier series coefficients of the signal $x(n)$, which can be obtained by the following formulas (Oppenheim, Willsky, and Nawab 1997):

$$a_k = \frac{1}{N} \sum_{n=0}^{N-1} x_n e^{-j\omega_0 n} = \frac{1}{N} \sum_{n=0}^{N-1} x_n e^{-j k (2\pi/N)n}$$

Fourier transform has an additivity property, which means that addition in the time domain corresponds to addition in the frequency domain. So the Fourier transform of a real signal $x(n)$ is the sum of the Fourier transform of signals $a_k e^{j k (2\pi/N)n}$ ($k = 0, 2, \ldots, N - 1$). The Fourier transform of $x(n)$ is actually a series of impulses in the frequency domain as follows:

$$X(e^{j\omega}) = \sum_{k=0}^{N-1} 2\pi a_k \delta(\omega - \frac{2\pi k}{N}) = \sum_{k=0}^{N-1} 2\pi a_k \delta(\omega - k \omega_0)$$

Figure 9.3 is a graphical representation of these equations.

The Fourier transform, which is a frequency-domain representation of a signal, decomposes the signal into multiple components on harmonic frequencies ($0, \omega_0, 2\omega_0, \ldots$) similar to how a musical chord can be expressed as the frequencies (or pitches) of its constituent notes. With this representation, information is easier to extract or remove by its frequency range.
Introduction to Digital Filtering

In digital signal processing, the function of a filter is to remove unwanted parts of an input signal (such as random noise) or to extract useful parts of the signal (such as the components within a certain frequency range). A digital filter uses a digital processor (for example, a general purpose computer or a digital signal processing microprocessor) to perform numerical calculations on digital signals in order to reduce or enhance certain aspects of that signal. A digital filter is defined by its impulse response, \( h(n) \), which is the filter’s output when the input signal is a unit impulse signal \( \delta(n) \). The unit impulse signal \( \delta(n) \) is defined as follows:

\[
\delta(n) = \begin{cases} 
1, & n = 0 \\
0, & n \neq 0
\end{cases}
\]

Figure 9.4 illustrates the impulse response of a digital filter. The importance of the impulse response is that a linear time-invariant filter is fully characterized by its impulse response. The output of a linear time-invariant digital filter, \( y(n) \), is the convolution of its impulse response, \( h(n) \), with the input signal, \( x(n) \), as shown in Figure 9.5.

It is known that if a signal \( y(n) \) is the convolution of two signals \( h(n) \) and \( x(n) \) in the time domain and if the \( z \) transforms of these three signals are \( Y(z) \), \( H(z) \), and \( X(z) \), respectively, then \( Y(z) = H(z) \times X(z) \). The transfer function of a digital filter, \( H(z) \), is defined as the ratio between the output signal \( z \) transform and the input signal \( z \) transform:

\[
H(z) = \frac{Y(z)}{X(z)}
\]

A basic property of the \( z \) transform is that if \( z \) is evaluated on the unit circle \( (z = e^{j\omega}) \), the \( z \) transform of a signal is equivalent to its Fourier transform. The frequency response of a linear time-invariant digital filter is defined as the filter’s transfer function \( H(z) \) evaluated on the unit circle, \( H(e^{j\omega}) \). This definition implies that a filter’s frequency response is also the Fourier transform of the impulse response \( h(n) \) (Oppenheim and Schafer 2010). A filter’s frequency response and the Fourier transforms of the filter’s input and output satisfy the following equation:

\[
H(e^{j\omega}) = \frac{Y(e^{j\omega})}{X(e^{j\omega})}
\]
Chapter 9: Time Filters Package

The function of a digital filter can be better explained by its frequency response. The frequency response measures how much the magnitude and phase of the output signal have changed compared to those of the input signal in frequency domain. A digital filter is essentially a system that allows signals of only certain frequencies to pass while blocking all others. Depending on the regime of frequencies that a digital filter allows through (or not), it is characterized as lowpass, highpass, bandpass, or bandstop:

- A **lowpass filter** retains the low-frequency part of an input signal and blocks (attenuates or removes) the high-frequency part of the input.
- A **highpass filter** retains the high-frequency part of an input signal and blocks (attenuates or removes) the low-frequency part of the input.
- A **bandpass filter** allows the middle frequencies of an input signal to pass and blocks (attenuates or removes) other frequencies.
- A **bandstop filter** is the opposite of a bandpass filter. It blocks (attenuates or removes) the middle frequencies and retains other frequencies.

The shape of the magnitude of the frequency response determines which frequencies will pass. Figure 9.6 shows the magnitude of the frequency response for the four different types of filters.

**Figure 9.6** Magnitude of the Frequency Response for Digital Filters

![Magnitude of the Frequency Response for Digital Filters](image)

The term digital filtering encompasses both the design of a filter and the filtering of an input signal. The purpose of filter design is to obtain a desired filter transfer function $H(z)$. Evaluating $H(z)$ on the unit circle ($z = e^{j\omega}$) results in the frequency response of the filter—that is, how the filter will pass or block certain frequencies in the input signal. The causal linear time-invariant filter’s transfer function can be written as

$$
H(z) = \frac{Y(z)}{X(z)}
$$

$$
H(z) = \frac{b_0 + b_1z^{-1} + b_2z^{-2} + \ldots + b_Nz^{-N}}{a_0 + a_1z^{-1} + a_2z^{-2} + \ldots + a_Mz^{-M}}
$$
\[ H(z) = k \frac{(1-q_1 z^{-1})(1-q_2 z^{-1}) \ldots (1-q_N z^{-1})}{(1-p_1 z^{-1})(1-p_2 z^{-1}) \ldots (1-p_M z^{-1})} \]

where the order of the filter is the greater of \( N \) or \( M \); \( b_i (i = 0, 1, 2, \ldots, N) \) are the transfer function’s numerator polynomial coefficients; \( a_i (i = 0, 1, 2, \ldots, M) \) are the denominator polynomial coefficients; and \( q_i (i = 1, 2, \ldots, N) \), \( p_i (i = 1, 2, \ldots, M) \), and \( k \) are the zeros, poles, and gain of the transfer function, respectively.

In the time domain, the preceding transfer function is equivalent to the following relationship between the filter’s input signal \( x(n) \) and output \( y(n) \):

\[
y(n) = \frac{1}{a_0} (b_0 x[n] + b_1 x[n - 1] + b_2 x[n - 2] + \ldots + b_N x[n - N] \\
- a_1 y[n - 1] - a_2 y[n - 2] + \ldots + a_M y[n - M])
\]

Based on the transfer function structure, there are two categories of digital filters: the infinite impulse response (IIR) filter and the finite impulse response (FIR) filter. When \( a_0 = 1 \) and \( a_i = 0 \) for \( i = 1, 2, \ldots, M \), a digital filter is an FIR filter and the output \( y(n) \) depends only on the current and previous inputs:

\[
y(n) = b_0 x[n] + b_1 x[n - 1] + b_2 x[n - 2] + \ldots + b_N x[n - N]
\]

IIR filters, on the other hand, are recursive, with the output depending not only on the current and previous inputs, but also on previous outputs. The general form of an IIR filter is thus:

\[
\sum_{m=0}^{M} a_m y[n - m] = \sum_{k=0}^{N} b_k x[n - k]
\]

A digital filter can be designed as an IIR filter or an FIR filter. The advantage of IIR filters over FIR filters is that they typically meet a particular set of specifications with a much lower filter order than a corresponding FIR filter. The classical IIR filters include Butterworth, Chebyshev Types I and II, elliptic, and Bessel (Parks and Burrus 1987; Roy 2005; Constantinides 1970).

Once a digital filter is designed, it can be applied to an input signal in order to extract certain frequencies from the signal and remove others. This filtering process can be implemented directly (the direct form) or by using a second-order section form of the filter’s transfer function (the SOS form). Any transfer function \( H(z) \) can have a second-order sections representation as follows:

\[
H(z) = \prod_{k=1}^{N} H_k(z)
\]

\[
H(z) = \prod_{k=1}^{N} \frac{b_{0k} + b_{1k} z^{-1} + b_{2k} z^{-2}}{1 + a_{1k} z^{-1} + a_{2k} z^{-2}}
\]

By careful pairing of the poles and zeros and careful ordering of the sections in the cascade, it is possible to reduce quantization noise gain and avoid overflow in the fixed-point filter implementations. So the second-order section form of filtering is more numerically stable than the direct form of filtering, and is thus preferred for real-world digital filtering applications.
**Digital Filtering Using Time Filtering Objects**

The following steps show how you can use the objects in this package to implement a digital filtering process:

1. Use the `DFORDER` object to compute the minimum order required for a digital filter that satisfies your frequency-domain requirement.

2. Use the `DFDESIGN` to obtain the transfer function coefficients, zeros, poles, and gain of a digital filter that meets your design specification. Currently the only supported design is the digital Butterworth IIR filter.

3. Use either the `DFFILTER` object or the `DFSOSFILTER` object to apply the designed filter to the input signal. The `DFFILTER` implements a direct-form of a filter’s transfer function, and the `DFSOSFILTER` implements the filtering process with a second-order section form of a filter’s transfer function. The output of this step is a filtered signal that retains certain frequencies of the input signal and removes others.

---

**Time Filters Package Summary**

Table 9.1 summarizes the objects in the time filters package.

<table>
<thead>
<tr>
<th>Object Description</th>
<th>Object</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Objects for Digital Filtering</strong></td>
<td></td>
</tr>
<tr>
<td>Computes the minimum order of a digital filter that satisfies a specified requirement</td>
<td><code>DFORDER</code></td>
</tr>
<tr>
<td>Returns the transfer function coefficients of a digital filter that meets the design specification</td>
<td><code>DFDESIGN</code></td>
</tr>
<tr>
<td>Filters a time series by using a digital filter transfer function</td>
<td><code>DFFILTER</code></td>
</tr>
<tr>
<td>Filters a time series by using the second-order section form of a digital filter transfer function</td>
<td><code>DFSOSFILTER</code></td>
</tr>
<tr>
<td><strong>Objects for Cepstrum Analysis</strong></td>
<td></td>
</tr>
<tr>
<td>Computes the real cepstrum or complex cepstrum of a time series</td>
<td><code>DFCEPSTRUM</code></td>
</tr>
<tr>
<td>Computes the inverse complex cepstrum</td>
<td><code>DFICEPSTRUM</code></td>
</tr>
<tr>
<td><strong>Objects for Volatility Detection</strong></td>
<td></td>
</tr>
<tr>
<td>Computes the range and moving relative range for each observation in a time series</td>
<td><code>MRR</code></td>
</tr>
<tr>
<td><strong>Collector Objects</strong></td>
<td></td>
</tr>
<tr>
<td>Collects output from a <code>DFORDER</code> object</td>
<td><code>OUTDFORDER</code></td>
</tr>
<tr>
<td>Collects output from a <code>DFDESIGN</code> object</td>
<td><code>OUTDFDESIGN</code></td>
</tr>
<tr>
<td>Collects output from a <code>DFFILTER</code> object</td>
<td><code>OUTDFFILTER</code></td>
</tr>
<tr>
<td>Collects output from a <code>DFSOSFILTER</code> object</td>
<td><code>OUTDFSOSFILTER</code></td>
</tr>
</tbody>
</table>
Table 9.1  continued

<table>
<thead>
<tr>
<th>Object</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OUTDFCEPSTRUM</td>
<td>Collects output from a DFCEPSTRUM object</td>
</tr>
<tr>
<td>OUTDFICEPSTRUM</td>
<td>Collects output from a DFICEPSTRUM object</td>
</tr>
<tr>
<td>OUTMRR</td>
<td>Collects output from an MRR object</td>
</tr>
</tbody>
</table>

Return Codes

Table 9.2 shows the return code (rc in method statements) status values that are used in this package. These status code values are returned after a method that is associated with an object is called; they can help determine whether the method executed successfully.

Table 9.2  Return Codes

<table>
<thead>
<tr>
<th>Status</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt; 0</td>
<td>An unrecoverable error occurred. No result was produced.</td>
</tr>
<tr>
<td>= 0</td>
<td>Unconditional success. The requested action completed and a normal result was produced.</td>
</tr>
<tr>
<td>&gt; 0</td>
<td>Conditional success or warning. A result was produced subject to conditions.</td>
</tr>
</tbody>
</table>

DFORDER Object

The DFORDER object computes the minimum order required for a digital filter that satisfies a requirement that you specify. Digital filters are designed by specifying the frequency response of the filters in the frequency domain. The specification includes a digital filter’s passband and stopband edge frequencies, passband ripple, and stopband attenuation. Figure 9.7 through Figure 9.10 illustrate the filter specification for each type of digital filter.
Figure 9.7  Filter Specification for Lowpass Filter

Figure 9.8  Filter Specification for Highpass Filter

Figure 9.9  Filter Specification for Bandpass Filter
The digital filter passband and stopband edge frequencies are $\omega_p$ and $\omega_s$. For lowpass and highpass filters, you need to specify only one value of each edge frequency. However, for bandpass and bandstop filters, you need to specify two values of the passband edge frequencies ($\omega_{p1}$ and $\omega_{p2}$) and two values of the stopband edge frequencies ($\omega_{s1}$ and $\omega_{s2}$). The frequency range from the passband edge frequency to the stopband edge frequency is the transition band. The transition band has a frequency response that is unspecified.

The digital filter passband and stopband can contain oscillations known as ripples. $\delta_1$ is the magnitude of the passband ripple, which equals the maximum deviation from the unity magnitude. $\delta_2$ is the magnitude response of the stopband attenuation, which equals the maximum deviation from zero. The passband ripple ($R_p$) and stopband attenuation ($R_s$) are usually measured in decibels (dB), and are defined as

\[
\begin{align*}
\text{Passband ripple: } R_p &= -20\log_{10}(1 - \delta_1) \text{ (dB)} \\
\text{Stopband attenuation: } R_s &= -20\log_{10}(\delta_2) \text{ (dB)}
\end{align*}
\]

where the absolute value of the passband ripple $R_p$ must be less than the absolute value of the stopband attenuation $R_s$ — that is, \(\text{abs}(R_p) < \text{abs}(R_s)\).

The values of the passband and stopband edge frequencies are normalized values between 0 and 1, with 1 corresponding to the normalized Nyquist frequency (\(\pi\) rad/sample). When the passband and stopband edge frequencies are specified, the following expressions must be satisfied:

- Lowpass filter: $\omega_p < \omega_s$
- Highpass filter: $\omega_s < \omega_p$
- Bandpass filter: $\omega_{s1} < \omega_{p1} < \omega_{p2} < \omega_{s2}$
- Bandstop filter: $\omega_{p1} < \omega_{s1} < \omega_{s2} < \omega_{p2}$

The DFORDER object outputs the digital filter’s cutoff frequencies, which can be used by the DFDESIGN object to design the filter’s transfer function. A digital filter’s cutoff frequency is defined as the frequency at which the power of the frequency response reaches half of the unity power, or equivalently \(\frac{1}{2} \approx 0.707\) of the unity magnitude, approximately \(-20\log_{10}(0.707) = 3\) in dB. Because half power is about 3dB away from unity power, this frequency is often called the 3dB cutoff frequency.

Table 9.3 summarizes the methods that are associated with the DFORDER object.
Table 9.3  Methods of the DFORDER Object

<table>
<thead>
<tr>
<th>Object</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GetResults</td>
<td>Get the DFORDER result</td>
</tr>
<tr>
<td>Initialize</td>
<td>Initialize a DFORDER instance</td>
</tr>
<tr>
<td>Run</td>
<td>Run the DFORDER computation</td>
</tr>
<tr>
<td>SetOption</td>
<td>Specify options for DFORDER</td>
</tr>
</tbody>
</table>

Figure 9.11 diagrams the methods of the DFORDER object.

Figure 9.11  DFORDER Data Flow
DFORDER Synopsis

```c
DECLARE OBJECT obj (DFORDER) ;
```

Method syntax, in order of typical usage:

```c
rc=obj.Initialize () ;
rc=obj.SetOption ('Name', Value) ;
rc=obj.Run () ;
rc=obj.GetResults (OutputArray) ;
```

DFORDER Methods

**DFORDER.GetResults Method**
```
rc=obj.GetResults (OutputArray ) ;
```

Outputs the results to an array.

**Input Arguments**
You must specify the following input arguments:

- **OutputArray** specifies a dynamic array that is used to store the output group component.

**DFORDER.Initialize Method**
```
rc=obj.Initialize () ;
```

Initializes a DFORDER instance. This method must be called before other attributes of the DFORDER instance are specified.

**Arguments**
There are no arguments associated with this method.

**DFORDER.Run Method**
```
rc=obj.Run () ;
```

Runs the DFORDER object to compute the minimum order required for a digital filter that satisfies a user-specified requirement.

**Arguments**
There are no arguments associated with this method.
DFORDER.SetOption Method

```plaintext
rc = obj.SetOption ('Name', Value) ;
```

Specifies named options for the DFORDER instance.

**Input Arguments**

You must specify the following arguments as 'Name', Value pairs:

- **'FNAME'** takes a string Value that specifies the name of the desired digital filter. Currently only the 'BUTTERWORTH' filter is supported.

- **'FTYPE'** takes a string Value that specifies the type of the desired digital filter. It is one of the following four values, 'LOWPASS', 'HIGHPASS', 'BANDPASS' or 'BANDSTOP'.

- **'PASSBANDFREQ1'** and **'PASSBANDFREQ2'** take numerical values to specify the passband edge frequencies. For a lowpass or highpass filter, only 'PASSBANDFREQ1' is required. For a bandpass or bandstop filter, both 'PASSBANDFREQ1' and 'PASSBANDFREQ2' are required. The values of the passband edge frequencies are between 0 and 1, with 1 corresponding to the normalized Nyquist frequency (\(\pi\) rad/sample).

- **'STOPBANDFREQ1'** and **'STOPBANDFREQ2'** take numerical values to specify the stopband edge frequencies. For a lowpass or highpass filter, only 'STOPBANDFREQ1' is required. For a bandpass or bandstop filter, both 'STOPBANDFREQ1' and 'STOPBANDFREQ2' are required. The values of the stopband edge frequencies are between 0 and 1, with 1 corresponding to the normalized Nyquist frequency (\(\pi\) rad/sample).

- **'PASSBANDRIPPLE'** takes a numerical Value that specifies the passband ripple in dB.

- **'STOPBANDATTEN'** takes a numerical Value that specifies the stopband attenuation in dB. The absolute value of the passband ripple \(R_p\) must be less than the absolute value of the stopband attenuation \(R_s\)—that is, \(|R_p| < |R_s|\).
The DFDESIGN object returns the transfer function coefficients of a digital filter that meets the design specification. The specification includes a digital filter’s name, type, order, and low cutoff frequency (or low and high cutoff frequencies). The transfer function’s numerator polynomial coefficients, denominator polynomial coefficients, zeros, poles, and gain are returned.

A digital filter’s transfer function $H(z)$ is defined as

$$H(z) = \frac{Y(z)}{X(z)}$$

$$H(z) = \frac{b_0 + b_1 z^{-1} + b_2 z^{-2} + \ldots + b_N z^{-N}}{a_0 + a_1 z^{-1} + a_2 z^{-2} + \ldots + a_M z^{-M}}$$

$$H(z) = k \frac{(1 - q_1 z^{-1})(1 - q_2 z^{-1})\ldots(1 - q_N z^{-1})}{(1 - p_1 z^{-1})(1 - p_2 z^{-1})\ldots(1 - p_M z^{-1})}$$

where the order of the filter is the greater of $N$ or $M$; $b_i (i = 0, 1, 2, \ldots, N)$ are the transfer function’s numerator polynomial coefficients; $a_i (i = 0, 1, 2, \ldots, M)$ are the denominator polynomial coefficients; and $q_i (i = 1, 2, \ldots, N)$, $p_i (i = 1, 2, \ldots, M)$, and $k$ are the zeros, poles, and gain of the transfer function, respectively.

Table 9.4 summarizes the methods that are associated with the DFDESIGN object.

<table>
<thead>
<tr>
<th>Object</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GetResults</td>
<td>Get the DFDESIGN result</td>
</tr>
<tr>
<td>Initialize</td>
<td>Initialize a DFDESIGN instance</td>
</tr>
<tr>
<td>Run</td>
<td>Run the DFDESIGN computation</td>
</tr>
<tr>
<td>SetOption</td>
<td>Specify options for DFDESIGN</td>
</tr>
</tbody>
</table>

Figure 9.12 diagrams the methods of the DFDESIGN object.
**DFDESIGN Synopsis**

DECLARE OBJECT obj (DFDESIGN) ;

Method syntax, in order of typical usage:

\[
rc=obj.\text{Initialize} (obj) ;
rc=obj.\text{SetOption} ('Name', Value) ;
rc=obj.\text{Run} () ;
rc=obj.\text{GetResults} (OutputArray) ;
\]
DFDESIGN Methods

DFDESIGN.GetResults Method

```c
rc = obj.GetResults (OutputArray) ;
```

Outputs the results to an array.

**Input Arguments**
You must specify the following input arguments:

- `OutputArray` specifies a dynamic array that is used to store the output group component.

DFDESIGN.Initialize Method

```c
rc = obj.Initialize (obj) ;
```

Initializes a DFDESIGN instance. This method must be called before other attributes of the DFDESIGN instance are specified.

**Input Arguments**
You must specify the following input argument:

- `obj` specifies a DFORDER object. The DFDESIGN object reads the filter order, CUTOFFREQ1, and CUTOFFREQ2 (for bandpass and bandstop filters) from the input DFORDER object. You can override these values by using the `SetOption` method with the proper name and value.

DFDESIGN.Run Method

```c
rc = obj.Run () ;
```

Runs the DFDESIGN object to compute the transfer function coefficients of a digital filter that meets the design specification.

**Arguments**
There are no arguments associated with this method.

DFDESIGN.SetOption Method

```c
rc = obj.SetOption ('Name', Value) ;
```

Specifies named options for the DFDESIGN instance.

**Input Arguments**
You must specify the following arguments as `Name`, `Value` pairs:

- `'FNAME'` takes a string `Value` that specifies the name of the desired digital filter. Currently only the 'BUTTERWORTH' filter is supported.
'FTYPE' takes a string Value that specifies the type of the desired digital filter. Specify one of the following four values, 'LOWPASS', 'HIGHPASS', 'BANDPASS' or 'BANDSTOP'.

'FILTERORDER' takes a numerical Value that specifies the order of the digital filter, which is a positive integer scalar. The maximum supported filter order is 200.

'CUTOFFFREQ1' takes a numerical Value that specifies the low cutoff frequency. This argument is required by all types of digital filters. The value of the cutoff frequency is between 0 and 1, with 1 corresponding to the normalized Nyquist frequency ($\pi$ rad/sample).

'CUTOFFFREQ2' takes a numerical Value that specifies the high cutoff frequency. This argument is required only when you specify 'BANDPASS' or 'BANDSTOP' for the 'FTYPE' argument. The value of the cutoff frequency is between 0 and 1, with 1 corresponding to the normalized Nyquist frequency ($\pi$ rad/sample).

---

**DFFILTER Object**

The DFFILTER object filters a time series by using a digital filter transfer function. Before the DFFILTER object can run the filtering process, you need to initialize the object by using a DFDESIGN object that has the computed numerator and denominator coefficients, and the zeros, poles, and gain of a filter’s transfer function. When you use the DFFILTER object, you can choose to use either transfer function coefficients or zeros, poles, and gain to filter the data. The output signal has the same size as the input signal.

Table 9.5 summarizes the methods that are associated with the DFFILTER object.

<table>
<thead>
<tr>
<th>Object</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GetResults</td>
<td>Get the DFFILTER result</td>
</tr>
<tr>
<td>Initialize</td>
<td>Initialize a DFFILTER instance</td>
</tr>
<tr>
<td>Run</td>
<td>Run the DFFILTER computation</td>
</tr>
<tr>
<td>SetInput</td>
<td>Specify the input time series</td>
</tr>
<tr>
<td>SetOption</td>
<td>Specify options for DFFILTER</td>
</tr>
</tbody>
</table>

Figure 9.13 diagrams the methods of the DFFILTER object.
DFFILTER Synopsis

DECLARE OBJECT obj (DFFILTER) ;

Method syntax, in order of typical usage:

```
rc=obj.Initialize (obj) ;
rc=obj.SetInput (Value) ;
rc=obj.SetOption (‘Name’, Value) ;
rc=obj.Run () ;
rc=obj.GetResults (OutputArray) ;
```
DFFILTER Methods

DFFILTER.GetResults Method

```c
rc = obj.GetResults (OutputArray ) ;
```

Outputs the results to an array.

**Input Arguments**
You must specify the following input arguments:

- `OutputArray` specifies a dynamic array that is used to store the output group component.

DFFILTER.Initialize Method

```c
rc = obj.Initialize (obj) ;
```

Initializes a DFFILTER instance by using an instance of the DFDESIGN object. This method must be called before the input and parameters of the DFFILTER instance are specified.

**Input Arguments**
You must specify the following input argument:

- `obj` specifies a DFDESIGN object.

DFFILTER.Run Method

```c
rc = obj.Run () ;
```

Runs the DFFILTER object to filter the input time series.

**Arguments**
There are no arguments associated with this method.

DFFILTER.SetInput Method

```c
rc = obj.SetInput (Value) ;
```

Adds a time series array (Value) to the DFFILTER instance.

**Input Arguments**
You must specify the following input argument:

- `Value` specifies an array of input time series.
**DFSOSFILTER Object**

The DFSOSFILTER object filters a time series by using the second-order section form of a digital filter transfer function. You need to use a DFDESIGN object that has the computed numerator and denominator coefficients, and the zeros, poles, and gain of a filter’s transfer function, to initialize the DFSOSFILTER object before it can run the filtering process. When you use the DFSOSFILTER object, you can choose either transfer function coefficients or zeros, poles, and gain to filter the data. The output signal has the same size as the input signal.

When filtering using the second-order section form, the filter’s transfer function is a concatenation of a series of second-order section digital filters, with the following form:

\[
H(z) = \prod_{k=1}^{N} H_k(z) \\
H(z) = \prod_{k=1}^{N} \frac{b_{0k} + b_{1k}z^{-1} + b_{2k}z^{-2}}{1 + a_{1k}z^{-1} + a_{2k}z^{-2}}
\]

Filtering using the second-order section form with a DFSOSFILTER object is more numerically stable than the direct filtering with a DFFILTER object, especially when the filter order is high. So it is recommended that you use the DFSOSFILTER object to filter a time series.

Table 9.6 summarizes the methods that are associated with the DFSOSFILTER object.
Table 9.6  Methods of the DFSOSFILTER Object

<table>
<thead>
<tr>
<th>Object</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GetResults</td>
<td>Get the DFSOSFILTER result</td>
</tr>
<tr>
<td>Initialize</td>
<td>Initialize a DFSOSFILTER instance</td>
</tr>
<tr>
<td>Run</td>
<td>Run the DFSOSFILTER computation</td>
</tr>
<tr>
<td>SetInput</td>
<td>Specify the input time series</td>
</tr>
<tr>
<td>SetOption</td>
<td>Specify options for DFSOSFILTER</td>
</tr>
</tbody>
</table>

Figure 9.14 diagrams the methods of the DFSOSFILTER object.

Figure 9.14  DFSOSFILTER Data Flow
**DFSOSFILTER Synopsis**

```plaintext
DECLARE OBJECT obj (DFSOSFILTER) ;
```

Method syntax, in order of typical usage:

```plaintext
rc=obj.Initialize (obj) ;
rc=obj.SetInput (Value) ;
rc=obj.SetOption ('Name',Value) ;
rc=obj.Run () ;
rc=obj.GetResults (OutputArray) ;
```

---

**DFSOSFILTER Methods**

**DFSOSFILTER.GetResults Method**

```plaintext
rc=obj.GetResults (OutputArray ) ;
```

Outputs the results to an array.

**Input Arguments**

You must specify the following input arguments:

- **OutputArray** specifies a dynamic array that is used to store the output group component.

**DFSOSFILTER.Initialize Method**

```plaintext
rc=obj.Initialize (obj) ;
```

Initializes a DFSOSFILTER instance by using an instance of the DFDESIGN object. This method must be called before the input and parameters of the DFSOSFILTER instance are specified.

**Input Arguments**

You must specify the following input argument:

- **obj** specifies a DFDESIGN object.

**DFSOSFILTER.Run Method**

```plaintext
rc=obj.Run () ;
```

Runs the DFSOSFILTER object to filter the input time series.

**Arguments**

There are no arguments associated with this method.
DFSOSFILTER.SetInput Method

\[ rc = \text{obj}.\text{SetInput}(\text{Value}) \];

Adds a time series array (\text{Value}) to the DFSOSFILTER instance.

**Input Arguments**

You must specify the following input argument:

\text{Value} specifies an array of input time series.

DFSOSFILTER.SetOption Method

\[ rc = \text{obj}.\text{SetOption}('\text{Name}', \text{Value}) \];

Specifies named options for the DFSOSFILTER instance.

**Input Arguments**

You must specify the following arguments as 'Name', Value pairs:

- `FPARAMTYPE` takes a string Value that specifies the type of filter parameters that should be used for filtering. You can specify the following Values:
  - 'TRANSFER' uses the numerator and denominator coefficients of the filter transfer function.
  - 'ZPG' uses the zeros, poles, and gain of the filter transfer function.

DFCEPSTRUM Object

The DFCEPSTRUM object computes the real or complex cepstrum of a time series. The complex cepstrum of a time series \(x\) is defined as

\[
y = \frac{1}{2\pi} \int_{-\pi}^{\pi} \log|X(e^{j\omega})|e^{j\omega a} d\omega
\]

where \(X(e^{j\omega})\) is the Fourier transform of the input time series. The real cepstrum of the time series \(x\) is defined as

\[
y = \frac{1}{2\pi} \int_{-\pi}^{\pi} \log|X(e^{j\omega})|e^{j\omega a} d\omega
\]

where \(|X(e^{j\omega})|\) is the magnitude of the Fourier transform (Akay 1994).

Table 9.7 summarizes the methods that are associated with the DFCEPSTRUM object.
Table 9.7  Methods of the DFCEPSTRUM Object

<table>
<thead>
<tr>
<th>Object</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GetND</td>
<td>Get the number of delay samples for complex cepstrum</td>
</tr>
<tr>
<td>GetResults</td>
<td>Get the DFCEPSTRUM result</td>
</tr>
<tr>
<td>Initialize</td>
<td>Initialize a DFCEPSTRUM instance</td>
</tr>
<tr>
<td>Run</td>
<td>Run the DFCEPSTRUM computation</td>
</tr>
<tr>
<td>SetInput</td>
<td>Specify the input time series</td>
</tr>
<tr>
<td>SetOption</td>
<td>Specify options for DFCEPSTRUM</td>
</tr>
</tbody>
</table>

Figure 9.15 diagrams the methods of the DFCEPSTRUM object.

**Figure 9.15**  DFCEPSTRUM Data Flow

[Diagram of DFCEPSTRUM data flow showing the methods and their interactions]
**DFCEPSTRUM Synopsis**

```plaintext
DECLARE OBJECT obj (DFCEPSTRUM) ;

Method syntax, in order of typical usage:

```n
```
rc=obj.Initialize () ;
rc=obj.SetInput (Value) ;
rc=obj.SetOption ('Name',Value) ;
rc=obj.Run () ;
rc=obj.GetResults (OutputArray) ;
nd=obj.GetND () ;
```

**DFCEPSTRUM Methods**

**DFCEPSTRUM.GetND Method**

```plaintext
nd=obj.GetND () ;
```

Outputs the ND value, using the OUTSCALAR statement in the TSMODEL procedure. ND is the number of samples of (circular) delay that are added to the input before the complex cepstrum is computed.

**Input Arguments**
You must specify the following input arguments:

- **Output** specifies a variable that is used to store the number of delay samples for the complex cepstrum. For a real cepstrum, this output value is always 0.

**DFCEPSTRUM.GetResults Method**

```plaintext
rc=obj.GetResults (OutputArray ) ;
```

Outputs the results to an array.

**Input Arguments**
You must specify the following input arguments:

- **OutputArray** specifies a dynamic array that is used to store the output group component.

**DFCEPSTRUM.Initialize Method**

```plaintext
rc=obj.Initialize () ;
```

Initializes a DFCEPSTRUM instance. This method must be called before the input and parameters of the DFCEPSTRUM instance are specified.

**Input Arguments**
There are no arguments associated with this method.
**DFCEPSTRUM.Run Method**

```c
rc = obj.Run();
```

Runs the DFCEPSTRUM object to compute the cepstrum of the input time series.

**Arguments**

There are no arguments associated with this method.

**DFCEPSTRUM.SetInput Method**

```c
rc = obj.SetInput(Value);
```

Adds a time series array (`Value`) to the DFCEPSTRUM instance.

**Input Arguments**

You must specify the following input argument:

- **Value** specifies an array of input time series.

**DFCEPSTRUM.SetOption Method**

```c
rc = obj.SetOption('Name', Value);
```

Specifies named options for the DFCEPSTRUM instance.

**Input Arguments**

You must specify the following arguments as `Name`, `Value` pairs:

- **'CEPS_TYPE'** takes a numerical `Value` that specifies the type of cepstrum to be computed for the input time series. You can specify the following `Values`:
  
  0 specifies a real cepstrum.
  
  1 specifies complex cepstrum.
DFICEPSTRUM Object

The DFICEPSTRUM object computes the inverse complex cepstrum of a time series. Table 9.8 summarizes the methods that are associated with the DFICEPSTRUM object.

Table 9.8 Methods of the DFICEPSTRUM Object

<table>
<thead>
<tr>
<th>Object</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GetResults</td>
<td>Get the DFICEPSTRUM result</td>
</tr>
<tr>
<td>Initialize</td>
<td>Initialize a DFICEPSTRUM instance</td>
</tr>
<tr>
<td>Run</td>
<td>Run the DFICEPSTRUM computation</td>
</tr>
<tr>
<td>SetInput</td>
<td>Specify the input time series</td>
</tr>
<tr>
<td>SetOption</td>
<td>Specify options for DFICEPSTRUM</td>
</tr>
</tbody>
</table>

Figure 9.16 diagrams the methods of the DFICEPSTRUM object.
DFICEPSTRUM Synopsis

DECLARE OBJECT obj (DFICEPSTRUM) ;

Method syntax, in order of typical usage:

```c
rc = obj.Initialize () ;
rc = obj.SetInput (Value) ;
rc = obj.SetOption (‘Name’, Value) ;
rc = obj.Run () ;
rc = obj.GetResults (OutputArray) ;
```
DFICEPSTRUM Methods

DFICEPSTRUM.GetResults Method

\[ rc = \text{obj}.\text{GetResults}(\text{OutputArray}) ; \]

Outputs the results to an array.

**Input Arguments**
You must specify the following input arguments:

*OutputArray* specifies a dynamic array that is used to store the output group component.

DFICEPSTRUM.Initialize Method

\[ rc = \text{obj}.\text{Initialize}() ; \]

Initializes a DFICEPSTRUM instance. This method must be called before the input and parameters of the DFICEPSTRUM instance are specified.

**Input Arguments**
There are no arguments associated with this method.

DFICEPSTRUM.Run Method

\[ rc = \text{obj}.\text{Run}() ; \]

Runs the DFICEPSTRUM object to compute the inverse complex cepstrum of the input time series.

**Arguments**
There are no arguments associated with this method.

DFICEPSTRUM.SetInput Method

\[ rc = \text{obj}.\text{SetInput}(\text{Value}) ; \]

Adds a time series array (Value) to the DFICEPSTRUM instance.

**Input Arguments**
You must specify the following input argument:

*Value* specifies an array of input time series.
**DFICEPSTRUM.SetOption Method**

```plaintext
rc = obj.SetOption ('Name', Value);
```

Specifies named options for the DFICEPSTRUM instance.

**Input Arguments**
You must specify the following arguments as 'Name', 'Value' pairs:

- **'CEPS_TYPE'** takes a numerical Value that specifies the type of inverse cepstrum to be computed for the input time series. Only the inverse complex cepstrum is supported, so this value must be 1.
- **'ND'** takes a numerical Value that specifies the number of samples of delay to be removed when the inverse complex cepstrum is computed. This value is the output when DFCEPSTRUM is used to compute the complex cepstrum.

---

**MRR Object**

The MRR object computes the range and moving relative range (MRR) for each time series observation. The MRR provides a measure of volatility for nonstationary time series, when both the mean and variance of the series are changing over time. Let $X_t$ denote the $t$th element of the time series. The MRR object computes the range and moving relative range for $X_t$ as

\[
\text{Range}_t = \text{Range}(X_t, X_{t-1}, \ldots, X_{t-M+1})
\]

\[
\text{MRR}_t = \frac{\text{Range}_t}{\text{Median}(\text{Range}_t, \text{Range}_{t-1}, \ldots, \text{Range}_{t-K+1})}
\]

where $M$ is the window length for computing the range and $K$ is the window length for computing the moving relative range.

Table 9.9 summarizes the methods that are associated with the MRR object.

<table>
<thead>
<tr>
<th>Object</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Getresult</td>
<td>Get the MRR result</td>
</tr>
<tr>
<td>Initialize</td>
<td>Initialize an MRR instance</td>
</tr>
<tr>
<td>Run</td>
<td>Run the MRR analysis</td>
</tr>
<tr>
<td>SetInput</td>
<td>Set the input time series</td>
</tr>
<tr>
<td>SetOption</td>
<td>Set options for MRR analysis</td>
</tr>
</tbody>
</table>

Table 9.9 Methods of the MRR Object

Figure 9.17 diagrams the methods of the MRR object.
MRR Synopsis

DECLARE OBJECT obj (MRR) ;

Method syntax, in order of typical usage:

rc = obj.Initialize () ;
rc = obj.SetInput (Value) ;
rc = obj.SetOption ('Name', Value) ;
rc = obj.Run () ;
rc = obj.Getresult (OutputArray) ;
MRR Methods

MRR.GetResults Method

\[
rc = \text{obj}.\text{GetResult} (\text{OutputArray}) ;
\]
Outputs the analysis results to an array.

**Input Arguments**
You must specify the following input arguments:

**OutputArray** specifies a dynamic array that is used to store the output group component.

MRR.Initialize Method

\[
rc = \text{obj}.\text{Initialize} () ;
\]
Initializes an MRR instance to an empty state. This method must be called before specifying the time series arrays and other attributes for the MRR instance.

**Arguments**
There are no arguments associated with this method.

MRR.Run Method

\[
rc = \text{obj}.\text{Run} () ;
\]
Runs the MRR object to perform the MRR analysis on the input array of time series.

**Arguments**
There are no arguments associated with this method.

MRR.SetInput Method

\[
rc = \text{obj}.\text{SetInput} (\text{Value}) ;
\]
Adds a time series array (\text{Value}) to the MRR instance.

**Input Arguments**
You must specify the following input argument:

**Value** specifies an array of input time series.
**MRR.SetOption Method**

\[
rc = \text{obj.SetOption ('Name', Value)};
\]

Sets named options for the MRR instance.

**Input Arguments**
You must specify the following arguments as 'Name', Value pairs:

- `'K'` takes a numeric value that specifies the window length for computing the moving relative range. For a time series whose variance is changing quickly, specify a lower value of K.
- `'M'` takes a numeric Value that specifies the window length for computing the range. For a time series whose mean is changing quickly, specify a lower value of M.

---

**OUTDFORDER Object**

The OUTDFORDER object collects output from the DFORDER object.

Table 9.10 summarizes the methods that are associated with the OUTDFORDER object.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Collect</td>
<td>Collect the singular values of a DFORDER object</td>
</tr>
<tr>
<td>NRows</td>
<td>Get the current row count from the OUTDFORDER object</td>
</tr>
</tbody>
</table>

Figure 9.18 diagrams the methods of the OUTDFORDER object.
OUTDFORDER Synopsis

```plaintext
DECLARE OBJECT outobj (OUTDFORDER) ;

Method syntax in order of typical usage:

  rc=outobj.Collect () ;
  nrows=outobj.NRows () ;
```

OUTDFORDER Methods

OUTDFORDER.Collect Method

```plaintext
rc=outobj.Collect (obj) ;
```

Collects the output from a DFORDER object and saves the results to a CAS table whose schema is shown in Table 9.11.
Table 9.11  CAS Table Collected with OUTDFORDER

<table>
<thead>
<tr>
<th>Column</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>filterorder</td>
<td>Numeric</td>
<td>The minimum digital filter order required.</td>
</tr>
<tr>
<td>CUTOFFREQ1</td>
<td>Numeric</td>
<td>The low cutoff frequency of the filter. For a lowpass or highpass filter, only CUTOFFREQ1 is returned.</td>
</tr>
<tr>
<td>CUTOFFREQ2</td>
<td>Numeric</td>
<td>The high cutoff frequency of the filter. For a bandpass or bandstop filter, both CUTOFFREQ1 and CUTOFFREQ2 are returned.</td>
</tr>
</tbody>
</table>

**Input Arguments**

You must specify the following input argument:

`obj` specifies the DFORDER object.

**OUTDFORDER.NRows Method**

```nrows=outobj.NRows () ;```

Gets the current row count from the OUTDFORDER instance.

**Arguments**

There are no arguments associated with this method.

**OUTDFDESIGN Object**

The OUTDFDESIGN object collects output from the DFDESIGN object.

Table 9.12 summarizes the methods that are associated with the OUTDFDESIGN object.

Table 9.12  Methods of the OUTDFDESIGN Object

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Collect</td>
<td>Collect the singular values of a DFDESIGN object</td>
</tr>
<tr>
<td>NRows</td>
<td>Get the current row count from the OUTDFDESIGN object</td>
</tr>
</tbody>
</table>

Figure 9.19 diagrams the methods of the OUTDFDESIGN object.
OUTDFDESIGN Synopsis

DECLARE OBJECT outobj (OUTDFDESIGN) ;

Method syntax in order of typical usage:

\[ rc = \text{outobj}.\text{Collect}() ; \]
\[ nrows = \text{outobj}.\text{NRows}() ; \]

OUTDFDESIGN Methods

OUTDFDESIGN.Collect Method

\[ rc = \text{outobj}.\text{Collect}(\text{obj}) ; \]

Collects the filter design output from a DFDESIGN object and saves the results to a CAS table whose schema is shown in Table 9.13.
### Table 9.13  CAS Table Collected with OUTDFDESIGN

<table>
<thead>
<tr>
<th>Column</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>Numeric</td>
<td>Denominator polynomial coefficients vector of a filter’s transfer function</td>
</tr>
<tr>
<td>b</td>
<td>Numeric</td>
<td>Numerator polynomial coefficients vector of a filter’s transfer function</td>
</tr>
<tr>
<td>zeros_real</td>
<td>Numeric</td>
<td>Real part of the zeros of a filter’s transfer function</td>
</tr>
<tr>
<td>zeros_imag</td>
<td>Numeric</td>
<td>Imaginary part of the zeros of a filter’s transfer function</td>
</tr>
<tr>
<td>poles_real</td>
<td>Numeric</td>
<td>Real part of the poles of a filter’s transfer function</td>
</tr>
<tr>
<td>poles_imag</td>
<td>Numeric</td>
<td>Imaginary part of the poles of a filter’s transfer function</td>
</tr>
<tr>
<td>gain</td>
<td>Numeric</td>
<td>Gain of a filter’s transfer function</td>
</tr>
</tbody>
</table>

**Input Arguments**
You must specify the following input argument:

\[ \text{obj} \]

specifies the DFDESIGN object.

**OUTDFDESIGN.NRows Method**

\[ \text{nrows} = \text{outobj.NRows}() ; \]

Gets the current row count from the OUTDFDESIGN instance.

**Arguments**
There are no arguments associated with this method.
OUTDFFILTER Object

The OUTDFFILTER object collects output from the DFFILTER object.

Table 9.14 summarizes the methods that are associated with the OUTDFFILTER object.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Collect</td>
<td>Collect the output values of a DFFILTER object</td>
</tr>
<tr>
<td>NRows</td>
<td>Get the current row count from the OUTDFFILTER object</td>
</tr>
</tbody>
</table>

Figure 9.20 diagrams the methods of the OUTDFFILTER object.
OUTDFFILTER Synopsis

DECLARE OBJECT outobj (OUTDFFILTER) ;

Method syntax in order of typical usage:

\[ rc = \text{outobj.Collect}() ; \]
\[ nrows = \text{outobj.NRows}() ; \]

OUTDFFILTER Methods

OUTDFFILTER.Collect Method

\[ rc = \text{outobj.Collect}(\text{obj}) ; \]

Collects the output of signal filtering from a DFFILTER object and saves the results to a CAS table whose schema is shown in Table 9.15.

Table 9.15 CAS Table Collected with OUTDFFILTER

<table>
<thead>
<tr>
<th>Column</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>output</td>
<td>Numeric</td>
<td>The filtered signal, which has the same size as the input signal</td>
</tr>
</tbody>
</table>

Input Arguments
You must specify the following input argument:

\[ \text{obj} \] specifies the DFFILTER object.

OUTDFFILTER.NRows Method

\[ nrows = \text{outobj.NRows}() ; \]

Gets the current row count from the OUTDFFILTER instance.

Arguments
There are no arguments associated with this method.
OUTDFSOSFILTER Object

The OUTDFSOSFILTER object collects output from the DFSOSFILTER object.

Table 9.16 summarizes the methods that are associated with the OUTDFSOSFILTER object.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Collect</td>
<td>Collect the output values of a DFSOSFILTER object</td>
</tr>
<tr>
<td>NRows</td>
<td>Get the current row count from the OUTDFSOSFILTER object</td>
</tr>
</tbody>
</table>

Figure 9.21 diagrams the methods of the OUTDFSOSFILTER object.

Figure 9.21 OUTDFSOSFILTER Data Flow
OUTDFSOSFILTER Synopsis

DECLARE OBJECT outobj (OUTDFSOSFILTER) ;

Method syntax in order of typical usage:

\[
rc = \text{outobj}.\text{Collect} () ;
\]
\[
nrows = \text{outobj}.\text{NRows} () ;
\]

OUTDFSOSFILTER Methods

OUTDFSOSFILTER.Collect Method

\[
rc = \text{outobj}.\text{Collect} (obj) ;
\]
Collects the output of signal filtering from a DFSOSFILTER object and saves the results to a CAS table whose schema is shown in Table 9.17.

<table>
<thead>
<tr>
<th>Column</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>output</td>
<td>Numeric</td>
<td>The filtered signal, which has the same size as the input signal</td>
</tr>
</tbody>
</table>

Input Arguments
You must specify the following input argument:

\[
obj
\]
specifies the DFSOSFILTER object.

OUTDFSOSFILTER.NRows Method

\[
nrows = \text{outobj}.\text{NRows} () ;
\]
Gets the current row count from the OUTDFSOSFILTER instance.

Arguments
There are no arguments associated with this method.
OUTDFCEPSTRUM Object

The OUTDFCEPSTRUM object collects output from the DFCEPSTRUM object.

Table 9.18 summarizes the methods that are associated with the OUTDFCEPSTRUM object.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Collect</td>
<td>Collect the output values of a DFCEPSTRUM object</td>
</tr>
<tr>
<td>NRows</td>
<td>Get the current row count from the OUTDFCEPSTRUM object</td>
</tr>
</tbody>
</table>

Figure 9.22 diagrams the methods of the OUTDFCEPSTRUM object.
OUTDFCEPSTRUM Synopsis

DECLARE OBJECT outobj (OUTDFCEPSTRUM) ;

Method syntax in order of typical usage:

\[
rc = \text{outobj}.\text{Collect} () ; \\
nrows = \text{outobj}.\text{NRows} () ;
\]

OUTDFCEPSTRUM Methods

OUTDFCEPSTRUM.Collect Method

\[
rc = \text{outobj}.\text{Collect} (\text{obj}) ;
\]

Collects the computed cepstrum from a DFCEPSTRUM object and saves the results to a CAS table whose schema is shown in Table 9.19.

Table 9.19  CAS Table Collected with OUTDFCEPSTRUM

<table>
<thead>
<tr>
<th>Column</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>output</td>
<td>Numeric</td>
<td>The cepstrum of the input time series</td>
</tr>
</tbody>
</table>

Input Arguments
You must specify the following input argument:

\[
\text{obj} \quad \text{specifies the DFCEPSTRUM object.}
\]

OUTDFCEPSTRUM.NRows Method

\[
nrows = \text{outobj}.\text{NRows} () ;
\]

Gets the current row count from the OUTDFCEPSTRUM instance.

Arguments
There are no arguments associated with this method.
OUTDFICEPSTRUM Object

The OUTDFICEPSTRUM object collects output from the DFICEPSTRUM object. Table 9.20 summarizes the methods that are associated with the OUTDFICEPSTRUM object.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Collect</td>
<td>Collect the output values of a DFICEPSTRUM object</td>
</tr>
<tr>
<td>NRows</td>
<td>Get the current row count from the OUTDFICEPSTRUM object</td>
</tr>
</tbody>
</table>

Figure 9.23 diagrams the methods of the OUTDFICEPSTRUM object.

**Figure 9.23** OUTDFICEPSTRUM Data Flow
OUTDFICEPSTRUM Synopsis

DECLARE OBJECT outobj (OUTDFICEPSTRUM) ;

Method syntax in order of typical usage:

\[ rc = \text{outobj.Collect}() ; \]
\[ nrows = \text{outobj.NRows}() ; \]

OUTDFICEPSTRUM Methods

OUTDFICEPSTRUM.Collect Method

\[ rc = \text{outobj.Collect}(\text{obj}) ; \]

Collects the computed cepstrum from a DFICEPSTRUM object and saves the results to a CAS table whose schema is shown in Table 9.21.

<table>
<thead>
<tr>
<th>Column</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>output</td>
<td>Numeric</td>
<td>The inverse complex cepstrum of the input time series</td>
</tr>
</tbody>
</table>

Input Arguments

You must specify the following input argument:

\[ \text{obj} \]
specifies the DFICEPSTRUM object.

OUTDFICEPSTRUM.NRows Method

\[ nrows = \text{outobj.NRows}() ; \]

Gets the current row count from the OUTDFICEPSTRUM instance.

Arguments

There are no arguments associated with this method.
OUTMRR Object

The OUTMRR object collects output from the MRR object.

Table 9.22 summarizes the methods that are associated with the OUTMRR object.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Collect</td>
<td>Collect the singular values of an MRR object</td>
</tr>
<tr>
<td>NRows</td>
<td>Get the current row count from the OUTMRR object</td>
</tr>
</tbody>
</table>

Figure 9.24 diagrams the methods of the OUTMRR object.
Figure 9.24 OUTMRR Data Flow
OUTMRR Synopsis

DECLARE OBJECT outobj (OUTMRR) ;

Method syntax in order of typical usage:

rc = outobj.Collect () ;
nrows = outobj.NRows () ;

OUTMRR Methods

OUTMRR.Collect Method

rc = outobj.Collect (obj) ;

Collects the output of moving relative range analysis from an MRR object and saves the results to a CAS table whose schema is shown in Table 9.23.

<table>
<thead>
<tr>
<th>Column</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>outrange</td>
<td>Numeric</td>
<td>The range value</td>
</tr>
<tr>
<td>outmrr</td>
<td>Numeric</td>
<td>The moving relative range value</td>
</tr>
</tbody>
</table>

Input Arguments

You must specify the following input argument:

obj specifies the MRR object.

OUTMRR.NRows Method

nrows = outobj.NRows () ;

Gets the current row count from the OUTMRR instance.

Arguments

There are no arguments associated with this method.
Examples: Time Filters Package

Using CAS Sessions and CAS Engine Librefs

SAS Cloud Analytic Services (CAS) is the analytic server and associated cloud services in SAS Viya. This section describes how to create a CAS session and set up a CAS engine libref that you can use to connect to the CAS session. It assumes that you have a CAS server already available; contact your system administrator if you need help starting and terminating a server. This CAS server is identified by specifying the host on which it runs and the port on which it listens for communications. To simplify your interactions with this CAS server, the host information and port information for the server are stored as SAS option values that are retrieved automatically whenever this CAS server needs to be accessed. You can examine the host and port values for the server at your site by using the following statements:

```sas
proc options option=(CASHOST CASPORT);
run;
```

In addition to starting a CAS server, your system administrator might also have created a CAS session and a CAS engine libref for your use. You can define your own sessions and CAS engine librefs that connect to the CAS server as shown in the following statements:

```sas
cas mysess;
libname mycas cas sessref=mysess;
```

The CAS statement creates the CAS session named `mysess`, and the LIBNAME statement creates the `mycas` CAS engine libref that you use to connect to this session. It is not necessary to explicitly name the CASHOST and CASPORT of the CAS server in the CAS statement, because these values are retrieved from the corresponding SAS option values.

If you have created the `mysess` session, you can terminate it by using the TERMINATE option in the CAS statement as follows:

```sas
cas mysess terminate;
```

For more information about the CAS statement and the LIBNAME statement, see *SAS Cloud Analytic Services: User’s Guide*. For general information about CAS and CAS sessions, see *SAS Cloud Analytic Services: Fundamentals*. 
Example 9.1: DFORDER and DFDESIGN Examples

This example uses the DFORDER object to compute the minimum order of a digital filter that satisfies the specified requirement and also uses the DFDESIGN object to get the transfer function coefficients of a digital filter that meets the design specification. The following DATA step creates a data set named test:

```plaintext
data test;
  input i x1 x2 x3 y1 y2 y3 r;
datalines;
  1 3 2 4 2 3 2 1
  2 5 4 5 4 5 3 1
  3 3 3 4 6 4 5 1
  4 3 6 6 7 6 7 1
  5 3 5 5 3 5 7 1
  6 6 6 6 8 8 8 1
  7 3 8 5 9 9 8 1
  8 8 9 8 3 7 3 1
  9 6 7 6 8 4 9 1
 10 7 9 8 9 6 7 1;
run;
```

You can load the work.test data set into your CAS session by specifying your CAS engine libref in the following DATA step. This DATA step assumes that your CAS engine libref id named mycas, but you can substitute any appropriately defined CAS engine libref.

```plaintext
data mycas.test;
  set test;
run;
```

The following statements use the DFORDER object to compute the minimum order of a Butterworth bandpass filter, and then use the DFDESIGN object to design a desired bandpass filter. The DFORDER output results are stored in the mycas.outdforder data table, and the DFDESIGN results are stored in the mycas.outdfdesign data table.

```plaintext
data mycas.test;
  set test;
run;
```

The following statements use the DFORDER object to compute the minimum order of a Butterworth bandpass filter, and then use the DFDESIGN object to design a desired bandpass filter. The DFORDER output results are stored in the mycas.outdforder data table, and the DFDESIGN results are stored in the mycas.outdfdesign data table.

```plaintext
proc tsmodel data=mycas.test
  outobj=(o=mycas.outdforder(replace=YES)
    q=mycas.outdfdesign(replace=YES))
  outlog=mycas.outlog
  logcontrol=(error=keep warning=keep note=keep); 
  id i interval=seconds;
  var x1 ;
  require timfil utl;
submit;
  declare object w(DFORDER);
  declare object o(OUTDFORDER);
  rc = w.setOption("FNAME","Butterworth", "FTYPE", 'bandpass',
    "PASSBANDFREQ1", 0.25, "PASSBANDFREQ2", 0.5,
    "STOPBANDFREQ1", 0.125, "STOPBANDFREQ2", 0.625,
    "PASSBANDRIPPLE", 3, "STOPBANDATTEN", 40);
  rc = w.run();
```
rc = o.Collect(w);

declare object p(DFDESIGN);
declare object q(OUTDFDESIGN);

rc = p.initialize(w);
rc = p.setOption("FNAME", 'Butterworth', "FTYPE", 'bandpass',
                  "CUTOFFFREQ1", 0.25, "CUTOFFFREQ2", 0.375);
rc = p.run();
rc = q.Collect(p);
endsubmit;
print outlog;
run;

The following statements save the final results and logs in the data tables, work.outdforder, work.outdfdesign, and work.outlog:

data outdforder;
  set mycas.outdforder;
run;

data outdfdesign;
  set mycas.outdfdesign;
run;

data outlog;
  set mycas.outlog;
run;

Example 9.2: DFFILTER and DFSOSFILTER Examples

The following examples show how to use a DFDESIGN object to get the desired filter transfer function and then use the DFFILTER or DFSOSFILTER to filter an input signal. The example code uses a DFSOSFILTER object, but it can be replaced by a DFFILTER object and used the same way. Using a filter’s second-order section form (DFSOSFILTER) is more numerically stable, so it is recommended.

The following macro generates a signal whose shape is the sum of two sinusoidal signals (each with frequency $f_1$ and $f_2$) and whose sampling frequency is $samp\_freq$:

```
%macro create_input(f1=, f2=, samp_freq=);
data test(keep = i x1);
  pi=constant("pi");
  f1=&f1;
  f2=&f2;
  T=1/&samp_freq;
  do i=0 to 199;
    x1=sin(2*pi*f1*i*T)+sin(2*pi*f2*i*T);
    output;
  end;
run;
%mend create_input;
```
The following statement generates an input signal (which is the sum of two sinusoidal signals with frequency $f_1 = 20$ Hz and $f_2 = 80$ Hz) and the sampling frequency $samp\_freq = 800$ Hz. The Nyquist frequency of this signal is 400 Hz, and the normalized frequencies of the two sinusoids are $\frac{20}{400} = 0.05$ and $\frac{80}{400} = 0.2$.

\[
%\text{create\_input}(f1=20, f2=80, samp\_freq=800);
\]

The following statements plot the input signal:

\[
\begin{align*}
\text{proc sgplot data=}\text{test}; \\
\quad \text{series } x=i \text{ y}=x1/\text{lineattrs=(color=blue thickness=2)}; \\
\quad \text{run};
\end{align*}
\]

Output 9.2.1 shows the input signal.

![Output 9.2.1 Plot of Input Data for Lowpass Filtering](image)

You can load the work.test data set into your CAS session by specifying your CAS engine libref in the following DATA step. This DATA step assumes that your CAS engine libref id named mycas, but you can substitute any appropriately defined CAS engine libref.

\[
\text{data mycas.test;} \\
\quad \text{set test;} \\
\quad \text{run;}
\]

The following statements use a DFDESIGN object to design a lowpass filter whose normalized cutoff frequency is 0.125, then initialize a DFSOSFILTER object with the DFDESIGN output filter parameters, and finally lowpass-filter the input signal. Only the sinusoidal signal whose normalized frequency is 0.05 will be in the output; the other sinusoidal signal, whose normalized frequency is 0.2, will be filtered out.

\[
\begin{align*}
\text{proc tsmodel data=mycas.test outobj=(o=mycas.outdfdesign1(replace=YES) } \\
\quad p=mycas.outdfsosfilter1(replace=YES)) \\
\quad \text{outlog=mycas.outlog} \\
\quad \text{logcontrol=(error=keep warning=keep note=keep);} ; \\
\text{id i interval=seconds;} \\
\text{var x1 ;} \\
\text{require timfil utl;} \\
\text{submit;} \\
\quad \text{declare object v(DFDESIGN);} \\
\end{align*}
\]
Chapter 9: Time Filters Package

```
declare object o(OUTDFDESIGN);
array d[10,5] /noshapes;
rc=v.setOption("FNAME","Butterworth", "FTYPE","lowpass", "FILTERORDER", 9,
   "CUTOFFREQ1", 0.125);
rc= v.run();
rc=v.getresults(d);
rc = o.Collect(v);

declare object w(DFSOSFILTER);
declare object p(OUTDFSOSFILTER);
rc=w.initialize(v);
rc=w.setinput(x1);
rc= w.run();
rc = p.Collect(w);
endsubmit;
print outlog;
run;
```

The following statement saves the filtered signal in the data table work.outdfsosfilter1:

```
data outdfsosfilter1;
   set mycas.outdfsosfilter1;
run;
```

The following statements plot the filtered signal by the lowpass filter:

```
proc sgplot data=work.outdfsosfilter1;
   series x=i y=output/lineattrs=(color=blue thickness=2);
run;
```

**Output 9.2.2** shows the output signal.

![Plot of Output Data for Lowpass Filtering](image)

In the following highpass filtering example, first an input signal is generated; the input signal is the sum of two sinusoidal signals with frequency $f_1 = 20$ Hz and $f_2 = 80$ Hz and the sampling frequency $samp\_freq = 800$ Hz. The Nyquist frequency of this signal is 400 Hz, and the normalized frequencies of the two sinusoids are $\frac{20}{400} = 0.05$ and $\frac{80}{400} = 0.2$. 
Example 9.2: DFFILTER and DFSOSFILTER Examples

%create_input(f1=20, f2=80, samp_freq=800);

The following statements plot the input signal:

```plaintext
proc sgplot data=test;
    series x=i y=x1/lineattrs=(color=blue thickness=2);
run;
```

Output 9.2.3 shows the input signal.

### Output 9.2.3
Plot of Input Data for Highpass Filtering

You can load the work.test data set into your CAS session by specifying your CAS engine libref in the following DATA step. This DATA step assumes that your CAS engine libref id named mycas, but you can substitute any appropriately defined CAS engine libref.

```plaintext
data mycas.test;
    set test;
run;
```

The following statements use a DFDESIGN object to design a highpass filter whose normalized cutoff frequency is 0.125, then initialize a DFSOSFILTER object with the DFDESIGN output filter parameters, and finally highpass-filter the input signal. Only the sinusoidal signal, whose normalized frequency is 0.2 will be in the output; the other sinusoidal signal, whose normalized frequency is 0.05, will be filtered out.

```plaintext
proc tsmodel data=mycas.test outobj=(o=mycas.outdfdesign2(replace=YES)
p=mycas.outdfsosfilter2(replace=YES))
    outlog=mycas.outlog
    logcontrol=(error=keep warning=keep note=keep); 
    id i interval=seconds;
    var x1 ;
    require timfil utl;
    submit;
        declare object v(DFDESIGN);
        declare object o(OUTDFDESIGN);
        rc=v.setOption("FNAME","Butterworth", "FTYPE","highpass", "FILTERORDER", 11,
                       "CUTOFFREQ1", 0.125);
```
The following statements plot the filtered signal by the highpass filter:

```plaintext
proc sgplot data=work.outdfsosfilter2;
   series x=i y=output/lineattrs=(color=blue thickness=2);
run;
```

Output 9.2.4 shows the output signal.

In the following bandpass filtering example, first an input signal is generated; the input signal is the sum of two sinusoidal signals with frequency $f_1 = 20$ Hz and $f_2 = 200$ Hz and the sampling frequency $samp\_freq = 800$ Hz. The Nyquist frequency of this signal is 400 Hz, and the normalized frequencies of the two sinusoids are $\frac{20}{400} = 0.05$ and $\frac{200}{400} = 0.5$.

```plaintext
%create_input(f1=20, f2=200, samp_freq=800);
```

The following statements plot the input signal:
proc sgplot data=test;
  series x=i y=x1/lineattrs=(color=blue thickness=2);
run;

Output 9.2.5 shows the input signal.

You can load the work.test data set into your CAS session by specifying your CAS engine libref in the following DATA step. This DATA step assumes that your CAS engine libref id named mycas, but you can substitute any appropriately defined CAS engine libref.

data mycas.test;
  set test;
run;

The following statements use a DFDESIGN object to design a bandpass filter whose normalized cutoff frequency is 0.125 and 0.375, then initialize a DFSOSFILTER object by using the DFDESIGN output filter parameters, and finally bandpass-filter the input signal. Only the sinusoidal signal, whose normalized frequency is 0.2, will be in the output; the other sinusoidal signal, whose normalized frequency is 0.05, will be filtered out.

proc tsmodel data=mycas.test outobj=(o=mycas.outdfdesign3(replace=YES)
p=mycas.outdfsosfilter3(replace=YES)
outlog=mycas.outlog logcontrol=(error=keep warning=keep note=keep));
  id i interval=seconds;
  var x1;
  require timfil utl;
submit;
  declare object v(DFDESIGN);
  declare object o(OUTDFDESIGN);
  rc=v.setOption("FNAME","Butterworth", "FTYPE","bandpass", "FILTERORDER", 5,
    "CUTOFFREQ1", 0.125, "CUTOFFREQ2", 0.375);
  rc= v.run();
  rc = o.Collect(v);
  declare object w(DFSOSFILTER);
Chapter 9: Time Filters Package

```
declare object p(OUTDFSOSFILTER);
rc=w.initialize(v);
rc=w.setinput(x1);
rc = w.run();
rc = p.Collect(w);
endsubmit;
print outlog;
run;
```

data outdfsosfilter3;
set mycas.outdfsosfilter3;
run;

The following statements plot the filtered signal by the bandpass filter:

```
proc sgplot data=work.outdfsosfilter3;
    series x=i y=output/lineattrs=(color=blue thickness=2);
run;
```

Output 9.2.6 shows the output signal.

Output 9.2.6  Plot of Output Data for Bandpass Filtering

In the following bandstop filtering example, first an input signal is generated; the input signal is the sum of two sinusoidal signals with frequency \( f1 = 20 \) Hz and \( f2 = 100 \) Hz and the sampling frequency \( samp_freq = 800 \) Hz. The Nyquist frequency of this signal is 400 Hz, and the normalized frequencies of the two sinusoids are \( \frac{20}{400} = 0.05 \) and \( \frac{100}{400} = 0.25 \).

```
%create_input(f1=20, f2=100, samp_freq=800);
```

The following statements plot the original input signal:

```
proc sgplot data=test;
    series x=i y=x1/lineattrs=(color=blue thickness=2);
run;
```

Output 9.2.7 shows the input signal.
You can load the `work.test` data set into your CAS session by specifying your CAS engine libref in the following DATA step. This DATA step assumes that your CAS engine libref id named `mycas`, but you can substitute any appropriately defined CAS engine libref.

```cas
data mycas.test;
    set test;
run;
```

The following statements use a DFDESIGN object to design a bandstop filter that has a normalized cutoff frequency at 0.2 and 0.5, then initialize a DFSOSFILTER object by using the DFDESIGN output filter parameters, and finally bandstop-filter the input signal. Only the sinusoidal signal whose normalized frequency is 0.05 will be in the output, the other sinusoidal signal, whose normalized frequency is 0.25, will be filtered out.

```cas
proc tsmodel data=mycas.test outobj=(o=mycas.outdfdesign4(replace=YES) p=mycas.outdfsosfilter4(replace=YES))
    outlog=mycas.outlog
    logcontrol=(error=keep warning=keep note=keep); ;
    id i interval=seconds;
    var x1 ;
    require timfil utl;
    submit;
    declare object v(DFDESIGN);
    declare object o(OUTDFDESIGN);
    rc=v.setOption("FNAME",'Butterworth', "FTYPE","bandstop", "FILTERORDER", 8, "CUTOFFREQ1", 0.2,"CUTOFFREQ2",0.5);
    rc = v.run();
    rc = o.Collect(v);
    declare object w(DFSOSFILTER);
    declare object p(OUTDFSOSFILTER);
    rc=w.initialize(v);
    rc=w.setinput(x1);
    rc= w.run();
    rc = p.Collect(w);
```
Chapter 9: Time Filters Package

The following statements plot the filtered signal by the bandstop filter:

```plaintext
proc sgplot data=work.outdfsosfilter4;
    series x=i y=output/lineattrs=(color=blue thickness=2);
run;
```

Output 9.2.8 shows the output signal.

**Example 9.3: DFCEPSTRUM and DFICEPSTRUM Examples**

This example shows how to use the DFCEPSTRUM and DFICEPSTRUM objects to compute the complex cepstrum and inverse complex cepstrum.

The following macro generates a signal whose shape is the sum of two sinusoidal signals (whose frequencies are $f_1$ and $f_2$) and whose sampling frequency is $samp\_freq$:

```plaintext
%macro create_input(f1=, f2=, samp_freq=);
    data test(keep = i x1);
        pi=constant("pi");
        f1=&f1;
        f2=&f2;
        T=1/&samp_freq;
        do i=0 to 199;
            x1=sin(2*pi*f1*i*T)+sin(2*pi*f2*i*T);
            x1=x1**2+ranuni(23);
            output;
        end;
end;
```
Example 9.3: DFCEPSTRUM and DFICEPSTRUM Examples

run;
%mend create_input;

%create_input(f1=20, f2=80, samp_freq=800);

The following statements plot the input signal:

```sas
proc sgplot data=test;
   series x=i y=x1/lineattrs=(color=blue thickness=2);
run;
```

Output 9.3.1 shows the input signal.

![Output 9.3.1 Plot of Input Data for Cepstrum]

You can load the work.test data set into your CAS session by specifying your CAS engine libref in the following DATA step. This DATA step assumes that your CAS engine libref id named mycas, but you can substitute any appropriately defined CAS engine libref.

```sas
data mycas.test;
   set test;
run;
```

The following statements compute the complex cepstrum of the input signal and also the ND (number of delay) value:

```sas
proc tsmodel data=mycas.test outobj=(o=mycas.outdfcepstrum(replace=YES) )
   outscalar=mycas.parameter(replace=yes)
   outlog=mycas.outlog
   logcontrol=(error=keep warning=keep note=keep);
   id i interval=seconds;
   var x1;
   require timfil utl;
   outscalar nd;
submit;
   declare object w(DFCEPSTRUM);
   declare object o(OUTDFCEPSTRUM);
   array d[10,5] /nosymbols ;
   rc=w.initialize();
```
The following statements compute the inverse complex cepstrum of a signal, which is the complex cepstrum of the original input signal. The ND value is the output from the DFCEPSTRUM object. The original signal will be recovered after this operation.

```
proc tsmodel data=mycas.temp outobj=(q=mycas.outdficepstrum(replace=YES) )
outlog=mycas.outlog logcontrol=(error=keep warning=keep note=keep);
    id i interval=seconds;
    var x1 ;
    require timfil utl;
    submit;
        declare object p(DFICEPSTRUM);
        declare object q(OUTDFICEPSTRUM);
        array d[10,5] /nosymbols;
        rc=p.initialize();
        rc=p.setoption("ND", 4);
        rc=p.setinput(x1);
        rc=p.run();
        rc=q.collect(p);
    endsubmit;
    print outlog;
run;
```

```
data mycas.temp (keep=x1 i);
set mycas.outdficepstrum;
x1=output;
run;
```

Output 9.3.2 shows the output signal.
Example 9.4: MRR Example

This example runs a moving relative range analysis on an MRR object and uses an OUTMRR object to collect the output. The following DATA step creates a data set named input:

```plaintext
data input;
  input date : monyy7. x;
  format date monyy7.;
datalines;
Jan-12 3.50137
Feb-12 3.35424
Mar-12 2.94675
Apr-12 2.10943
... more lines ...
```

The following statements plot the input time series:

```plaintext
proc sgplot data=input;
  series x=date y=x/lineattrs=(color=blue thickness=2) markers;
run;
```

Output 9.4.1 shows the results.
You can load the `work.input` data set into your CAS session by specifying your CAS engine libref in the following DATA step. This DATA step assumes that your CAS engine libref id named `mycas`, but you can substitute any appropriately defined CAS engine libref.

```plaintext
data mycas.input;
   set input;
run;
```

The following statements call the MRR object to compute the moving relative range of the `mycas.input` data table and store the result in the `mycas.outmrr` data table:

```plaintext
proc tsmodel data=mycas.input outobj=(o=mycas.outmrr(replace=YES));
   id date interval=month;
   var x;
   require timfil;
submit;
   declare object w(MRR);
   declare object o(OUTMRR);
   rc = w.initialize();
   rc = w.setinput(x);
   rc = w.setOption("M",3,"K",3);
   rc = w.run();
   rc = o.Collect(w);
endsubmit;
run;
```

The following statements sort the `mycas.outmrr` data table and merge it with the `mycas.input` data table to create a data table named `result`:

```plaintext
data outmrr;
   set mycas.outmrr;
run;
```
proc sort data=outmrr;
  by date;
run;

data result;
  merge mycas.input(in=a) outmrr(in=b);
  by date;
  if a and b;
run;

The following statements plot the original time series with the MRR values:

proc sgplot data=result;
  series x=date y=x/lineattrs=(color=blue thickness=2) markers;
  series x=date y=outmrr/y2axis lineattrs=(color=red thickness=2) markers;
run;

Output 9.4.2 shows the results.

References


Chapter 10

Time-Frequency Analysis Package

Contents

Overview .................................................. 324
TFA Package Summary ..................................... 324
  Using CAS Sessions and CAS Engine Librefs .............. 325
FFT Class ................................................ 326
  FFT Object ........................................... 326
  FFT.Run Method ........................................ 326
  OUTFFT Object ......................................... 327
  OUTFFT.Collect Method ................................ 327
  OUTFFT.Nrows Attribute .............................. 327
  Example .............................................. 327
FFTC Class .............................................. 330
  FFTC Object .......................................... 331
  OUTFFTC Object ...................................... 331
  OUTFFTC.Collect Method ............................. 332
  OUTFFTC.Nrows Attribute ............................ 332
  Example .............................................. 332
HILBERT Class .......................................... 333
  HILBERT Object ...................................... 333
  HILBERT.Run Method .................................. 333
  OUTHILBERT Object .................................. 334
  OUTHILBERT.Collect Method ......................... 334
  OUTHILBERT.Nrows Attribute ......................... 334
  Example .............................................. 334
PWV Class .............................................. 336
  PWV Object .......................................... 336
  PWV.Run Method ...................................... 336
  OUTPWV Object ...................................... 338
  OUTPWV.Collect Method ............................. 338
  OUTPWV.Nrows Attribute ............................ 339
  Example .............................................. 339
STFT Class .............................................. 340
  STFT Object .......................................... 340
  STFT.Run Method ..................................... 340
  OUTSTFT Object ...................................... 341
  OUTSTFT.Nrows Attribute ............................ 342
Overview

Time-frequency analysis refers to techniques that analyze a time series in both time and frequency domains. The time-frequency analysis (TFA) package provides objects (organized in classes) that enable you to perform time-frequency analysis as part of the programming statements in the TSMODEL procedure in SAS Visual Forecasting.

TFA Package Summary

Table 10.1 summarizes the classes in the TFA package.

<table>
<thead>
<tr>
<th>TFA Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FFT</td>
<td>Computes the discrete Fourier transform of a real time series</td>
</tr>
<tr>
<td>FFTC</td>
<td>Computes the discrete Fourier transform of a complex time series</td>
</tr>
<tr>
<td>HILBERT</td>
<td>Computes the analytic signal that corresponds to a real time series</td>
</tr>
<tr>
<td>PWV</td>
<td>Computes the pseudo-Wigner-Ville distribution of a real time series</td>
</tr>
<tr>
<td>STFT</td>
<td>Computes the short-time Fourier transform of a real time series</td>
</tr>
<tr>
<td>WINDOW</td>
<td>Creates a window of a requested type and length</td>
</tr>
</tbody>
</table>
TFA classes can be run independently for each BY group by specifying the BY statement. It is necessary in most cases to also add the TRIMID=BOTH option in the ID statement. For an example, see the section “Example” on page 327.

---

**Using CAS Sessions and CAS Engine Librefs**

SAS Cloud Analytic Services (CAS) is the analytic server and associated cloud services in SAS Viya. This section describes how to create a CAS session and set up a CAS engine libref that you can use to connect to the CAS session. It assumes that you have a CAS server already available; contact your system administrator if you need help starting and terminating a server. This CAS server is identified by specifying the host on which it runs and the port on which it listens for communications. To simplify your interactions with this CAS server, the host information and port information for the server are stored as SAS option values that are retrieved automatically whenever this CAS server needs to be accessed. You can examine the host and port values for the server at your site by using the following statements:

```sas
proc options option=(CASHOST CASPORT);
run;
```

In addition to starting a CAS server, your system administrator might also have created a CAS session and a CAS engine libref for your use. You can define your own sessions and CAS engine librefs that connect to the CAS server as shown in the following statements:

```sas
cas mysess;
libname mycas cas sessref=mysess;
```

The CAS statement creates the CAS session named `mysess`, and the LIBNAME statement creates the `mycas` CAS engine libref that you use to connect to this session. It is not necessary to explicitly name the CASHOST and CASPORT of the CAS server in the CAS statement, because these values are retrieved from the corresponding SAS option values.

If you have created the `mysess` session, you can terminate it by using the TERMINATE option in the CAS statement as follows:

```sas
cas mysess terminate;
```

For more information about the CAS statement and the LIBNAME statement, see *SAS Cloud Analytic Services: User’s Guide*. For general information about CAS and CAS sessions, see *SAS Cloud Analytic Services: Fundamentals*. 

---
FFT Class

DECLARE OBJECT f (FFT) ;
DECLARE OBJECT of (OUTFFT) ;
rc = f.Run ('Name',Value[, 'Name',Value,...]) ;
rc = of.Collect (f) ;

The FFT class computes the discrete Fourier transform of a real time series. Given an input array \( y = (y[0], y[1], \ldots, y[n-1]) \) and \( s = \pm 1 \), the output is \( z = (z[0], z[1], \ldots, z[n-1]) \), where

\[
    z[t] = \sum_{k=0}^{n-1} y[k] \exp \left( \frac{s2\pi i kt}{n} \right), \quad 0 \leq t \leq n-1
\]

When \( s = -1 \), the output \( z \) is the forward discrete Fourier transform of \( y \); when \( s = 1 \), the output \( z \) is the backward discrete Fourier transform of \( y \).

FFT Object

DECLARE OBJECT obj(FFT) defines an object that is used to compute the discrete Fourier transform of a real time series.

FFT.Run Method

Usage: \( rc = obj.Run ('y',Value,[,'sign',Value]) ; \)

Required Arguments

You must specify the following argument as a 'Name', Value pair:

\( y \) specifies an array of real-valued time series. Any missing value in this array is replaced by 0 before computation.

Optional Arguments

You can also specify the following argument as a 'Name', Value pair:

\( sign \) specifies the sign of discrete Fourier transform. You can specify the following values within single or double quotation marks:

- 'FORWARD' calculates a forward discrete Fourier transform \( (s = -1) \).
- 'BACKWARD' calculates backward discrete Fourier transform with \( (s = 1) \).
The default value of \textit{sign} is \textsc{forward}. Forward and backward transforms are inverses to each other in the following sense: performing a backward transform on a time series of length \( n \) followed by a forward transform on the resulting series leads to \( n \) times the original series and vice-versa.

## OUTFFT Object

DECLARE OBJECT \texttt{obj(OUTFFT)} defines an object that is used to collect output from FFT instance.

## OUTFFT.Collect Method

Usage: \[
\textit{rc} = \texttt{obj.Collect('FFTObj')};
\]

This method collects the output of discrete Fourier transform from an FFT object and saves the result to a CAS table, whose schema is shown in Table 10.2.

<table>
<thead>
<tr>
<th>Column</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>Numeric</td>
<td>The real part of the discrete Fourier transform of input time series</td>
</tr>
<tr>
<td>Y</td>
<td>Numeric</td>
<td>The imaginary part of the discrete Fourier transform of input time series</td>
</tr>
</tbody>
</table>

## OUTFFT.Nrows Attribute

Usage: \[
\textit{nrows} = \texttt{obj.nrows()};
\]

This attribute gets the current row count from the OUTFFT instance.

## Example

The following statements read and plot the yearly sunspot count data since 1900. These statements assume that your CAS engine libref is named \texttt{mycas}, but you can substitute any appropriately defined CAS engine libref.

```plaintext
data mycas.sunspot;
  input x@@;
  year = _N_ + 1900 - 1;
  i = _N_;
datalines;
15.7 4.6 8.5 40.8 70.1 105.5 90.1 102.8 80.9 73.2 30.9 9.5 6.0 2.4
16.1 79.0 95.0 173.6 134.6 105.7 62.7 43.5 23.7 9.7 27.9 74.0 106.5
114.7 129.7 108.2 59.4 35.1 18.6 9.2 14.6 60.2 132.8 190.6 182.6
```
Chapter 10: Time-Frequency Analysis Package

148.0 113.0 79.2 50.8 27.1 16.1 55.3 154.3 214.7 193.0 190.7 118.9
98.3 45.0 20.1 6.6 54.2 200.7 269.3 261.7 225.1 159.0 76.4 53.4 39.9
15.0 22.0 66.8 132.9 150.0 149.4 148.0 94.4 97.6 54.1 49.2 22.5 18.4
39.3 131.0 220.1 218.9 198.9 162.4 91.0 60.5 20.6 14.8 33.9 123.0
211.1 191.8 203.3 133.0 76.1 44.9 25.1 11.6 28.9 88.3 136.3 173.9
170.4 163.6 99.3 65.3 45.8 24.7 12.6 4.2 4.8 24.9 80.8 84.5 94.0
113.3 69.8

; run;

data sunspot;
   set mycas.sunspot;
run;

proc sort data=sunspot;
   by year;
run;

proc sgplot data=sunspot;
   yaxis label="Mean Sunspot Number";
   title "Sunspot numbers over years";
   series x=Year y=x;
run;

Output 10.1 shows the results.

Figure 10.1 Sunspot Numbers over Years

The following statements call the FFT class to compute the forward discrete Fourier transform of the sunspot count data and store the result in the mycas.fft_x data set:

proc tsmodel data=mycas.sunspot outobj=(of=mycas.fft_x(replace=YES));
   var x;
   id i interval=second;
   require tfa;
   submit;
   declare object f(FFT);
The following example runs the FFT class on column x of the mycas.x table. The BY statement causes the FFT class to be run on both group 1 (g=1) and group 2 (g=2):
data mycas.x;
  input x g;
  i = _N_;
  datalines;
  1 1
  2 1
  3 1
  4 1
  -3 2
  1 2
  3 2
  4 2
  0 2;
run;

proc tsmodel data=mycas.x outobj=(of=mycas.z(replace=YES));
  var x;
  by g;
  id i interval=second trimid=both;
  require tfa;
  submit;
  declare object f(FFT);
  declare object of(OUTFFT);
  /* Call FFT on x column of SAS table mycas.x */
  rc = f.Run("y", x, "sign", 'forward'); if rc then stop;
  rc = of.Collect(f); if rc then stop;
  endsubmit;
run;

### FFTC Class

**DECLARE OBJECT f (FFTC) ;**

**DECLARE OBJECT of (OUTFFTC) ;**

```plaintext
rc = f.Run('Name',Value,['Name',Value,...]);
rc = of.Collect(f);
```

The FFTC class computes the discrete Fourier transform of a complex time series. Given an input array \( y = (y[0] \ y[1] \ldots y[n-1]) \) and \( s = \pm 1 \), the output is \( z = (z[0] \ z[1] \ldots z[n-1]) \), where

\[
    z[t] = \sum_{k=0}^{n-1} y[k] \exp \frac{2\pi i kt}{n}, \quad 0 \leq t \leq n - 1
\]

When \( s = -1 \), the output \( z \) is the *forward* discrete Fourier transform of \( y \); when \( s = 1 \), the output \( z \) is the *backward* discrete Fourier transform of \( y \).
**FFTC Object**

DECLARE OBJECT obj(FFTC) defines an object that is used to compute the discrete Fourier transform of a complex time series.

**FFTC.Run Method**

Usage:  
\[ rc = \text{obj}.\text{Run} (\{\text{Name},\text{Value},\ldots\}) \];

**Required Arguments**

You must specify the following arguments as 'Name', Value pairs and separated by a comma:

- \( y_{re} \) specifies an array of the real part of the input complex time series. Any missing value in this array is replaced with 0 before computation.
- \( y_{im} \) specifies an array of the imaginary part of the input complex time series. Any missing value in this array is replaced with 0 before computation.

**Optional Arguments**

You can also specify the following argument as a 'Name', Value pair:

- \( sign \) specifies the sign of the discrete Fourier transform. You can specify the following values within single or double quotation marks:
  
  - 'FORWARD' calculates a forward discrete Fourier Transform \( (s = -1) \).
  
  - 'BACKWARD' calculates a backward discrete Fourier Transform \( (s = 1) \).

  The default value of \( sign \) is FORWARD. The forward and backward transforms are inverses to each other in the following sense: performing the backward transform on a time series of length \( n \) followed by the forward transform on the resulting series leads to \( n \) times the original series and vice-versa.

**OUTFFTC Object**

DECLARE OBJECT obj(OUTFFTC) defines an object that is used to collect output from FFTC instance.
OUTFFTC.Collect Method

Usage: \( rc = \text{obj}.\text{Collect}(\text{"FFTCObj"}); \)

The method collects the output of discrete Fourier transform from an FFTC instance and saves the result to a CAS table, whose schema is shown in Table 10.3.

<table>
<thead>
<tr>
<th>Column</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>Numeric</td>
<td>The real part of the discrete Fourier transform of input time series</td>
</tr>
<tr>
<td>Y</td>
<td>Numeric</td>
<td>The imaginary part of the discrete Fourier transform of input time series</td>
</tr>
</tbody>
</table>

OUTFFTC.Nrows Attribute

Usage: \( n\text{rows} = \text{obj}.\text{nrows}(); \)

This attribute gets the current row count from the OUTFFTC instance.

Example

The following example runs the FFTC class once on the columns that are named \text{x\_re} and \text{x\_im} of the \text{mycas.x} table and then runs the class again on two input arrays that are named \text{y\_re} and \text{y\_im}:

```r
data mycas.x;
  input x_re x_im;
  i = _N_; datalines;
  1 -1
  2 -2
  3 -3
  4 -4;
run;

proc tsmodel data=mycas.x outobj=(of1=mycas.z1(replace=YES) of2=mycas.z2(replace=YES));
  var x_re x_im;
  id i interval=second;
  require tfa;
  submit;
  declare object f(FFTC);
  declare object of1(OUTFFTC);
  declare object of2(OUTFFTC);
  /* Call FFTC on x\_re, x\_im column of SAS table mycas.x */
```
HILBERT Class

```plaintext
DECLARE OBJECT h (HILBERT) ;
DECLARE OBJECT oh (OUTHILBERT) ;
rc = h.Run ('Name', Value[, 'Name', Value, ...]) ;
rc = oh.Collect (h) ;
```

The HILBERT class computes the analytic signal that corresponds to a real time series. This analytic signal is a complex time series whose real component is the input time series and whose imaginary component is the discrete Hilbert transform of the input time series.

HILBERT Object

DECLARE OBJECT obj(HILBERT) defines an object that is used to computes an analytic signal that corresponds to a real time series.

HILBERT.Run Method

Usage:  
```plaintext
rc = obj.Run ('Name', Value[, 'Name', Value, ...]) ;
```

Required Arguments

You must specify the following argument as a 'Name', Value pair:

- `y` specifies an array that represents an input time series. Any missing value is replaced with 0.
OUTHILBERT Object

DECLARE OBJECT obj(OUTHILBERT) defines an object that is used to collect output from a HILBERT instance.

OUTHILBERT.Collect Method

Usage: \( rc = obj.\text{Collect}('\text{HILBERTObj}') \);

This method collects the output of a discrete Hilbert transform from a HILBERT instance and saves the result to a CAS table, whose schema is shown in Table 10.4.

<table>
<thead>
<tr>
<th>Column</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>Numeric</td>
<td>The real part of the discrete Fourier transform of input time series</td>
</tr>
<tr>
<td>Y</td>
<td>Numeric</td>
<td>The imaginary part of the discrete Fourier transform of input time series</td>
</tr>
</tbody>
</table>

OUTHILBERT.Nrows Attribute

Usage: \( \text{nrows} = obj.\text{nrows}() \);

This attribute gets the current row count from the OUTHILBERT instance.

Example

This example uses a time series that is obtained by sampling from a sinusoidal of the form \( x(t) = A \cos(2\pi \theta(t) + \phi) \), where \( \dot{\theta}(t) = \frac{d\theta(t)}{dt} \) is a linear function of time and is known as a linear chirp. For such a sinusoidal, it is reasonable to interpret \( \dot{\theta}(t) \) as the “instantaneous frequency” of \( x(t) \), and the Hilbert transform can be used to extract the instantaneous frequency of the linear chirp (Boashash 1992a, b). The following example runs the HILBERT class to compute the instantaneous frequency of a linear chirp:

```plaintext
/* Generate a chirp with linear instantaneous frequency.
   The chirp is sampled at 1 kHz for 2 seconds. The instantaneous
   frequency is 100 at t = 0 and 200 Hz at t = 1. */

data mycas.chirp;
  retain pi %sysfunc(constant(pi));
  keep x i;
  f0 = 100;
  f1 = 200;
  Fs = 1000;
  i = 0;
  do time = 0 to (2-1/Fs) by 1/Fs;
```
\[
x = \cos(2\pi*(f_0*\text{time} + 0.5*(f_1-f_0)*\text{time}^2));
\]
\[
i = i + 1;
\]
\[
\text{output};
\]
\[
\text{end;}
\]
\[
\text{run;}
\]

/* Compute the Hilbert transform of the chirp data and use it compute the instantaneous frequency */
proc tsmodel data=mycas.chirp outobj=(oh=mycas.analytic (replace=YES)) outlog=mycas.log logControl=(warning=keep error=keep note=keep);
  var x;
  id i interval=second;
  require tfa;
  submit;
    declare object h(HILBERT);
    declare object oh(OUTHILBERT);
    rc = h.run('Y', x); if rc then stop;
    rc = oh.Collect(h);
  endsubmit;
run;

data mycas.analytic;
  set mycas.analytic;
  keep time instfreq;
  time = (_N_ - 1)*(1/1000);
  angle = atan2(Y, X);
  lag_angle = lag(angle);
  diff = angle - lag_angle;
  if diff < -constant('pi') then diff2 = diff + 2*constant('pi');
  else if diff > constant('pi') then diff2 = diff - 2*constant('pi');
  else diff2 = diff;
  instfreq = 1000/(2*constant('pi'))*diff2;
  if _N_ >= 3;
run;

proc sgplot data=mycas.analytic;
  title "Instantaneous Frequency of Linear Chirp";
  scatter x=time y=instfreq;
  xaxis label="Time";
  yaxis label="HZ";
run;

Output 10.3 shows the results.
PWV Class

DECLARE OBJECT p (PWV) ;
DECLARE OBJECT op (OUTPWV) ;
rc = p.Run ('Name', Value[, 'Name', Value,...]) ;
rc = op.Collect (p) ;

The PWV class computes the pseudo-Wigner-Ville distribution of a real time series.

PWV Object

DECLARE OBJECT obj(PWV) defines an object that is used to compute an analytic signal that corresponds to a real time series.

PWV.Run Method

Usage: rc = obj.Run ('Name', Value[, 'Name', Value,...]) ;

Required Arguments

You must specify the following arguments as 'Name', Value pairs and separated by a comma:

*y specifies an array of input time series. The length of this series is denoted in the rest of this section as series_length. Any missing value in this array is replaced with 0 before computation.
**window** specifies an array of numbers that contains the window values to be used for computation of the pseudo-Wigner-Ville distribution. The length of the window must be odd. The length of the window is denoted as window_length in the rest of this section.

### Optional Arguments

You can also specify the following arguments as ‘*Name*, Value’ pairs, separated by commas:

- **overlap** specifies the extent of overlap between two consecutive windows. The value of overlap must be an integer that is strictly less than window_length. The default value is \( \lfloor \frac{n}{2} \rfloor \), where \( n = \text{window_length} \).

- **fftlen** specifies the length of the vector on which the discrete Fourier transform is to be performed. The value of fftlen must be a positive integer and must be at least as large as window_length. It is recommended that fftlen be a power of two in order to speed up the computation. The default value is the larger of 256 and window_length.

- **center** specifies whether the input time series is centered. You can specify the following values for center:
  - 0 does not center the input time series.
  - 1 subtracts the mean of the entire series from each term of the input series before calculating the PWV distribution. The missing values of the input time series are excluded in the calculation of the mean.

  The default value of center is 0.

- **nthreads** specifies the number of threads to use. The value of nthreads must be a nonnegative integer and must not be larger than 128. This argument along with the value of k (see section “OUTPWV.Collect Method” on page 338) determines the number of threads that are used as follows:
  - If the value of nthreads is strictly positive, then the PWV object attempts to perform the computation using nthreads threads.
  - If the value of nthreads is larger than k, the computation uses only one thread.
  - If the value of nthreads is 0, then the number of threads that are used is equal to the number of available CPUs.

  The default value of nthreads is 1.

- **hilbert_tsf** specifies whether to replace the input signal by an analytical signal. You can specify the following values for hilbert_tsf:
  - 0 does not replace the input series.
  - 1 replaces the input series by its analytic signal before further computations if the value of center is 0; otherwise (the value of center is 1), replaces the original signal by the analytic signal that corresponds to the centered signal.

  The default value of hilbert_tsf is 1.
Chapter 10: Time-Frequency Analysis Package

\( \texttt{fade} \) determines the size and alignment of the output. For a description of the impact of this parameter on computation and output size, see the section “Pseudo-Wigner-Ville Distribution” on page 349. The default value of \( \texttt{fade} \) is 1.

### OUTPWV Object

DECLARE OBJECT obj(OUTPWV) defines an object that is used to collect output from a PWV instance.

### OUTPWV.Collect Method

Usage: \( \texttt{rc = obj.Collect ('PWVObj');} \)

This method collects the output of the pseudo-Wigner-Ville transform from a PWV instance and saves the result to a CAS table, whose schema is shown in Table 10.5.

<table>
<thead>
<tr>
<th>Column</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Frequency</td>
<td>Numeric</td>
<td>The normalized frequency index</td>
</tr>
<tr>
<td>Time</td>
<td>Numeric</td>
<td>The time index</td>
</tr>
<tr>
<td>PWV</td>
<td>Numeric</td>
<td>The value of the pseudo-Wigner-Ville distribution</td>
</tr>
</tbody>
</table>

Note: The number of rows in the output table is \( \texttt{fftlen \times k} \), where

\[
k = \begin{cases} 
\left\lfloor \frac{\text{series_length} - 1}{\text{window_length} - \text{overlap}} \right\rfloor & \text{, when } \texttt{fade} = 1 \\
\left\lfloor \frac{\text{series_length} - \text{overlap}}{\text{window_length} - \text{overlap}} \right\rfloor + 1 & \text{, when } \texttt{fade} = 0
\end{cases}
\]

The frequency index starts from 0 and increments in multiples of \( \frac{1}{2 \times \texttt{fftlen}} \). The time index starts from 0 and increments in multiples of \( \text{window_length} - \text{overlap} \).
OUTPWV.Nrows Attribute

Usage:  
\[
\text{nrows = obj.nrows ();}
\]

This attribute gets the current row count from the OUTPWV instance.

Example

The following example calls the PWV class on column y of the mycas.a data set:

```plaintext
data mycas.a;
  input y;
  i = _N_;  
datalines;
  3
  1
  4
  2
;
run;

proc tsmodel data=mycas.a outobj=(opwv=mycas.outpwv(replace=YES))
  outlog=mycas.log logControl=(warning=keep error=keep note=keep);
  id i interval=seconds;
  var y;
  require tfa;
  submit;
    declare object w(WINDOW);
    declare object pwv(PWV);
    declare object opwv(OUTPWV);
    rc = w.Run('name', 'hamming', 'length', 3); if rc then stop;
    array window[1]/nosymbols;
    rc = w.Save(window); if rc then stop;
    rc = pwv.Run('y', y, 'window', window, 'overlap', 2, 'fftlen', 4, 'fade', 1);
    if rc then stop;
    rc = opwv.Collect(pwv); if rc then stop;
  endsubmit;
run;
```
The STFT class computes the short-time Fourier transform of a real time series. The short-time Fourier transform is a time-frequency distribution; for more information, see the section “Short-Time Fourier Transform” on page 351.

**STFT Class**

```plaintext
DECLARE OBJECT s (STFT) ;
DECLARE OBJECT os (OUTSTFT) ;
rc = s.Run ('Name',Value,['Name',Value,...]) ;
rc = os.Collect (s) ;
```

The STFT class computes the short-time Fourier transform of a real time series. The short-time Fourier transform is a time-frequency distribution; for more information, see the section “Short-Time Fourier Transform” on page 351.

**STFT Object**

DECLARE OBJECT obj(STFT) defines an object that is used to compute the short-time Fourier transform of a real time series.

**STFT.Run Method**

Usage: 

```plaintext
rc = obj.Run ('Name',Value,['Name',Value,...]) ;
```

**Required Arguments**

You must specify the following argument as a 'Name', Value pair:

- `y` specifies an array of input time series. The length of this series is denoted in the rest of this section as `series_length`. Any missing value in this array is replaced with 0 before computation.

**Optional Arguments**

You can also specify the following arguments as 'Name', Value pairs, separated by commas:

- `window` specifies an array of numbers that contains the window values to be used for the short-time Fourier transform. The default is a Hanning window whose length is the lesser of 256 and the series length. The length of the window is denoted as `window_length` in the rest of this section.

- `overlap` specifies the extent of overlap between two consecutive windows, where `overlap` must be an integer that is strictly less than `window_length`. The default value is \( \lfloor \frac{n}{2} \rfloor \), where \( n = window_length \).

- `fftlen` specifies the length of the vector on which the discrete Fourier transformation is to be done, where `fftlen` must be a positive integer and must be at least as large as `window_length`. The default value is the larger of 256 and `window_length`. It is recommended that `fftlen` be a power of two in order to speed up the computation.
**center** specifies whether the input time series is centered. You can specify the following values for center:

- **0** does not perform centering.
- **1** performs centering by subtracting the mean of the entire series from each term of the input series before performing the short-time Fourier transform. Missing values are ignored during the computation of the mean.

The default value of center is 0.

**nthreads** specifies the number of threads to use, where nthreads must be a nonnegative integer and must not be larger than 128. The number of threads that are used is determined by the values of nthreads and k (see the section “Optional Arguments” on page 341).

- If the value of nthreads is strictly positive, then the STFT object attempts to perform the computation by using nthreads threads.
- If the value of nthreads is larger than k, the computation uses only one thread.
- If the value of nthreads is 0, then the number of threads that are used is equal to the number of available CPUs.

The default value of nthreads is 1.

---

**OUTSTFT Object**

DECLARE OBJECT obj(OUTSTFT) defines an object that is used to collect output from an STFT instance.

**OUTSTFT.Collect Method**

Usage:  
```
rc = obj.Collect('STFTObj');
```

The method collects the output of the short-time Fourier transform from an STFT instance and saves the result to a CAS table, whose schema is shown in Table 10.6.

**Table 10.6** CAS Table Collected with OUTSTFT

<table>
<thead>
<tr>
<th>Column</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Frequency</td>
<td>Numeric</td>
<td>The normalized frequency index</td>
</tr>
<tr>
<td>Time</td>
<td>Numeric</td>
<td>The time index</td>
</tr>
<tr>
<td>Power</td>
<td>Numeric</td>
<td>The power of the time series that corresponds to the current frequency and time index</td>
</tr>
<tr>
<td>Amplitude</td>
<td>Numeric</td>
<td>The amplitude of the time series that corresponds to the current frequency and time index</td>
</tr>
<tr>
<td>Phase</td>
<td>Numeric</td>
<td>The phase of the time series that corresponds to the current frequency and time index</td>
</tr>
<tr>
<td>Coef_re</td>
<td>Numeric</td>
<td>The real part of the Fourier coefficient</td>
</tr>
<tr>
<td>Coef_im</td>
<td>Numeric</td>
<td>The imaginary part of the Fourier coefficient</td>
</tr>
</tbody>
</table>
Note: The number of rows in the output table is \( \text{fftlen} \times k \), where \( k = \left\lfloor \frac{\text{series_length} - \text{overlap}}{\text{window_length} - \text{overlap}} \right\rfloor + 1 \).

The frequency index starts from 0 and increases in increments of \( \frac{1}{\text{fftlen}} \). The time index starts from 0 and increments in multiples of \( \text{window_length} - \text{overlap} \). The power is computed by \( x^2 + y^2 \), where \( x \) is the real part of the Fourier coefficient (the value stored in the sixth column) and \( y \) is the imaginary part of the Fourier-coefficient (the value stored in the seventh column). The amplitude is given by \( \sqrt{\text{power}} \), where \( \text{power} \) is the value stored in the third column. The phase is computed as \( \text{atan2}(y, x) \) where \( x \) is the real part of the time series (stored in the sixth column) and \( y \) is the imaginary part of the time-series (stored in the seventh column).

### OUTSTFT.Nrows Attribute

**Usage:**

```
nrows = obj.nrows () ;
```

This attribute gets the current row count from the OUTSTFT instance.

### Example

The following example calls the STFT class on column \( y \) of the mycas.a data table:

```plaintext
data mycas.a;
  input y;
  i = _N_;
datalines;
  3
  1
  4
  2
;
run;

proc tsmodel data=mycas.a outobj=(os=mycas.outstft(replace=YES))
  outlog=mycas.log logControl=(warning=keep error=keep note=keep);
  id i interval=seconds;
  var y;
  require TFA;
submit;
  declare object w(WINDOW);
  declare object s(STFT);
  declare object os(OUTSTFT);
  /*--- create a hamming window of size 3 ---*/
  rc = w.Run('name', 'hamming', 'length', 3); if rc then stop;
  array window[1]/nosymbols;
  rc = w.Save(window); if rc then stop;
  rc = s.Run('y', y, 'window', window, 'overlap', 2, 'fftlen', 4); if rc then stop;
  rc = os.Collect(s); if rc then stop;
endsubmit;
run;
```
The WINDOW class creates a window of a requested type and length. The WINDOW class is useful for smoothing spectra. For more information about the window functions that are implemented in the time-frequency analysis package, see Harris (1978).

**WINDOW Object**

DECLARE OBJECT obj(WINDOW) defines an object that is used to compute windows.

**WINDOW.Run Method**

Usage: \( rc = \text{obj.Run} \left( \text{\textquote{\textquote{Name},Value[,]\textquote{Name},Value,\ldots}} \right) \);

**Required Arguments**

You must specify the following argument as a 'Name, Value pair:

- **length** specifies the length of the requested window.

**Optional Arguments**

You can also specify the following arguments as 'Name, Value pairs, separated by a comma:

- **name** specifies the type of window. The default is a HANNING window.
  
  In the description of the following window functions, \( N \) denotes the length of the window, and the \( N \) values that define a window are given by \( w[0], \ldots, w[N - 1] \). Some windows need additional parameters.

  You can specify the following values of **name** within single or double quotation marks:

  - **BARTLETT** specifies a Bartlett window. For this window type, you do not need to specify any **params**. This window function is defined as
    
    \[ w[i] = 1 - \left| \frac{2i}{N - 1} - 1 \right|, \ 0 \leq i \leq N - 1 \]

  - **BARTLETT_HANN** specifies a Bartlett-Hann window. For this window type, you do not need to specify any **params**. This window function is defined as
    
    \[ w[i] = 0.62 - 0.48 \left| \frac{i}{N - 1} - 0.5 \right| - 0.38 \cos \left( \frac{2\pi i}{N - 1} \right), \ 0 \leq i \leq N - 1 \]
BLACKMAN specifies a Blackman window. For this window type, you do not need to specify any `params`. This window is defined as

\[
w[i] = 0.42 - 0.5 \cos \frac{2\pi i}{N - 1} + 0.08 \cos \frac{4\pi i}{N - 1}, \quad 0 \leq i \leq N - 1
\]

BLACKMAN_HARRIS specifies a Blackman-Harris window. For this window type, you do not need to specify any `params`. This window function is defined as

\[
w[i] = 0.35875 - 0.48829 \cos \frac{2\pi i}{N - 1} + 0.14128 \cos \frac{4\pi i}{N - 1} - 0.01168 \cos \frac{6\pi i}{N - 1}, \quad 0 \leq i \leq N - 1
\]

BOHMAN specifies a Bohman window. For this window type, you do not need to specify any `params`. This window function is defined as

\[
w[0] = w[N - 1] = 0
\]

\[
w[i] = \left(1 - \left|1 - \frac{2i}{N - 1}\right|\right) \cos \left(\pi \left|1 - \frac{2i}{N - 1}\right|\right)
\]

\[
+ \frac{1}{\pi} \sin \left(\pi \left|1 - \frac{2i}{N - 1}\right|\right), \quad 1 \leq i \leq N - 2
\]

CHEBYSHEV specifies a Chebyshev window. This window function needs one `param`: `att`, whose default value is 100. To define this window, you need to define the \(n\)th-degree Chebyshev polynomial, \(T_n\), which is the unique polynomial such that \(T_n(\cos \theta) = \cos n\theta\) for all values of \(\theta\). \(T_n(x)\) can be computed as

\[
T_n(x) = \begin{cases} 
\cos(n \cos^{-1} x), & |x| \leq 1 \\
\cosh(n \acosh(x)), & x > 1 \\
(-1)^n T_n(-x), & x < -1
\end{cases}
\]

For odd \(N\) (say, \(N = 2M + 1\)) with \(M > 0\), the Chebyshev window of length \(N\) can be defined as

\[
w[i] = c \left(1 + \frac{2}{T_{2M}(\beta)} \sum_{k=1}^{M} T_{2M}(\beta \cos \frac{k\pi}{N}) \cos \frac{2\pi k(i - M)}{N}\right), \quad 0 \leq i \leq N - 1
\]

where \(\beta = \cosh(\acosh(10^{\text{att}/20})/(N - 1))\) and \(c\) is chosen to make the largest term of \(w\) equal to 1.

For even \(N\), the Chebyshev window of length \(N\) can be defined as

\[
w[i] = c \times \left\{ \sum_{k=0}^{N-1} (-1)^k T_{N-1}(\beta \cos \frac{k\pi}{N}) \cos \frac{\pi k(2i + 1)}{N}\right\}, \quad 0 \leq i \leq N - 1
\]

where \(c\) is chosen to make the largest term of \(w\) equal to 1.
FLAT_TOP specifies a flat-top window. For this window type, you do not need to specify any params. This window is function defined as

\[ w[i] = 0.21557895 - 0.41663158 \cos \frac{2\pi i}{N-1} + 0.277263158 \cos \frac{4\pi i}{N-1} - 0.083578947 \cos \frac{6\pi i}{N-1} + 0.006947368 \cos \frac{8\pi i}{N-1}, \quad 0 \leq i \leq N - 1 \]

GAUSSIAN specifies a Gaussian window. This window function needs one param: \( c \), whose default value is 2.5. This window function is defined as

\[ w[i] = \exp\left(-\frac{c^2}{2} \left(\frac{i - \frac{N-1}{2}}{\frac{N-1}{2}}\right)^2\right), \quad 0 \leq i \leq N - 1 \]

HAMMING specifies a Hamming window. For this window type, you do not need to specify any params. This window function is defined as

\[ w[i] = 0.54 - 0.46 \cos \frac{2\pi i}{N-1}, \quad 0 \leq i \leq N - 1 \]

HANNING specifies a Hanning window. For this window type, you do not need to specify any params. This window function is defined as

\[ w[i] = \frac{1}{2} \left(1 - \cos \frac{2\pi i}{N-1}\right), \quad 0 \leq i \leq N - 1 \]

KAISER specifies a Kaiser window. This window function needs one param: \( \beta \), whose default value is 0.5. This window function is defined as

\[ w[i] = \frac{I_0\left(\beta \sqrt{1 - (1 - \frac{2i}{N-1})^2}\right)}{I_0(\beta)}, \quad 0 \leq i \leq N - 1 \]

where \( I_0(\cdot) \) is the modified Bessel function of the first kind of order 0, which is defined as

\[ I_0(x) = \sum_{m=0}^{\infty} \left(\frac{x}{2}\right)^{2m} \frac{1}{m!^2} \]

PARZEN specifies a Parzen window. For this window type, you do not need to specify any params. This window function is defined as

\[ w[i] = \begin{cases} 
2 \left(1 - \left|\frac{2i - (N-1)}{N}\right|\right)^3, & 0 \leq i < \frac{N-1}{4} \\
1 - 6 \left|\frac{2i - (N-1)}{N}\right|^2 + 6 \left(\frac{2i - (N-1)}{N}\right)^3, & \frac{N-1}{4} \leq i \leq \frac{N-1}{2} \\
W[N - i - 1], & \frac{N-1}{2} < i \leq N - 1
\end{cases} \]

The last half of the window is defined by symmetry, which implies \( w[i] = w[N - i - 1] \) for \( 0 \leq i \leq N - 1 \).
RECTANGULAR specifies a rectangular window. For this window type, you do not need to specify any params. This window function is defined as

\[ w[i] = 1, \ 0 \leq i \leq N - 1 \]

TUKEY specifies a Tukey window. This window function needs one param: \( \alpha \), whose default value is 0.5. Let \( \epsilon = 10^{-12} \). If \( \alpha \geq 1 \), then a Hanning window is returned; if \( \alpha \leq \epsilon \), a rectangular window is returned. For \( \epsilon < \alpha < 1 \), this window function is defined as

\[
w[i] = \begin{cases} 
\frac{1}{2} \left( 1 + \cos \left( \frac{\pi}{\alpha} \left( \frac{2i}{N-1} - \alpha \right) \right) \right), & 0 \leq i < \alpha(N - 1)/2 \\
1, & \alpha(N - 1)/2 \leq i \leq (N - 1)/2 \\
w[N - i - 1], & (N - 1)/2 < i \leq N - 1 
\end{cases}
\]

The last half of the window is defined by symmetry.

params specifies an array of numbers that specify parameters to be used for creating windows. Not all window types require parameters. When params is missing, the default value of params is used for any window type that needs a parameter.

---

#### WINDOW.Save Method

Usage: \( rc = obj.Save \ (window) \);

This method saves the result (window array) to a dynamic array. For examples, see the sections “PWV Class” on page 336 and “STFT Class” on page 340.

**Required Arguments**

You must specify the following argument:

- **window** specifies the TKCMP dynamic array to which to save the result.

---

#### OUTWINDOW Object

DECLARE OBJECT obj(OUTWINDOW) defines an object that is used to collect output from a WINDOW instance.
OUTWINDOW.Collect Method

Usage: \( rc = \textit{obj}.\textit{Collect} \left(\textit{\textquote{WINDOWObj}}\right) ; \)

This method collects an array of numbers that contain the window from a WINDOW instance and saves the result to a CAS table, whose schema is shown in Table 10.7.

<table>
<thead>
<tr>
<th>Column</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NAME</td>
<td>String</td>
<td>The type of the window</td>
</tr>
<tr>
<td>X</td>
<td>Numeric</td>
<td>The array of numbers that contain the window</td>
</tr>
</tbody>
</table>

OUTWINDOW.Nrows Attribute

Usage: \( \textit{nrows} = \textit{obj}.\textit{nrows} () ; \)

This attribute gets the current row count from the OUTWINDOW instance.

Example

The following DATA step creates a dummy data table:

```plaintext
data mycas.dummy;
dummy = 1;
i = _N_; run;
```

The following statements create a Hanning window of length 5 and output it to a CAS table:

```plaintext
proc tsmodel data=mycas.dummy outobj=(ow=mycas.outwindow(replace=YES))
   outlog=mycas.log logControl=(warning=keep error=keep note=keep);
   var dummy;
   id i interval=seconds;
   require tfa;
   submit;
   declare object w(WINDOW);
   declare object ow(OUTWINDOW);
   rc = w.Run("length", 5, "name", "hanning");
   rc = ow.Collect(w);
   endsubmit;
run;
```
Details

Discrete Fourier Transforms

The discrete Fourier transform \( y = (y_0 \ y_1 \ldots \ y_{n-1})^T \) of a time-series \( x = (x_0 \ x_1 \ldots \ x_{n-1})^T \) can be obtained from the following matrix multiplication:

\[
\begin{pmatrix}
y_0 \\
y_1 \\
\vdots \\
y_{n-1}
\end{pmatrix}
= 
\begin{pmatrix}
1 & 1 & 1 & \ldots & 1 \\
1 & \omega & \omega^2 & \ldots & \omega^{n-1} \\
\vdots & \vdots & \vdots & \ddots & \vdots \\
1 & \omega^{n-1} & \omega^{2(n-1)} & \ldots & \omega^{(n-1)(n-1)}
\end{pmatrix}
\begin{pmatrix}
x_0 \\
x_1 \\
\vdots \\
x_{n-1}
\end{pmatrix}
\]

That is, \( y = Wx \), where \( \omega = \exp(-\frac{2\pi i}{n}) \) and \( W = (\omega^{(i-1)(j-1)})_{i,j=0}^{n-1,n-1} \). The backward discrete transform of \( x \) can be obtained from the matrix vector product \( W^*x \), where \( W^* \) denotes the conjugate transpose of \( W \). \( W \) satisfies the relation \( WW^* = W^*W = nI \), where \( I \) is the identity matrix, thus explaining the inverse relation between the forward and the backward transforms. Even though naive computation of \( Wx \) takes \( \mathcal{O}(n^2) \) operations, the time-frequency analysis package implementation belongs to class of algorithms known as fast Fourier transforms, which exploit the structure of \( W \) to compute the discrete Fourier transform of \( x \) in \( \mathcal{O}(n \log n) \) operations. For more information, see Van Loan (1992).

Hilbert Transformation

The HILBERT function computes the analytic signal of the input. The analytic signal that corresponds to a continuous time series \( x(t) \) is the complex time series \( z(t) = x(t) + i\hat{x}(t) \), where \( \hat{x}(t) \) is the Hilbert transform of \( x(t) \). In many applications, replacing the original time series by its analytic transform produces better results (Marple 1999).

For a continuous time series \( x(t) \) with Fourier transform \( X(f) = \int x(t) \exp(-2\pi if t) \, dt \), the Hilbert transform is defined by inverting \( X(f) \) over the positive frequencies (Cohen 1995, pg 30):

\[
\hat{x}(t) = 2 \int_0^\infty X(f) \exp(2\pi if t) \, df
\]

The spectrum of \( \hat{x}(t) \) is identical to the spectrum of \( x(t) \) for positive frequencies, and the spectrum of \( \hat{x}(t) \) is 0 for negative frequencies.

The Hilbert transform of a discrete time series is similarly constructed as a time series whose discrete Fourier transform coincides with that of the input time series for positive spectra and vanishes otherwise. The time-frequency analysis package implementation is based on the method described in Marple (1999).
Time-Frequency Distributions

Time-frequency distributions are standard tools for studying a time series whose frequency behavior varies with time. The discussion here follows the treatment given in Cohen (1995).

An important concept in time frequency analysis is the “energy” of a signal. Let $x(t)$ be a continuous time series with Fourier transform $X(f)$. Then for well-behaved $x(t)$ and $X(f)$, you can consider $E = \int_{-\infty}^{\infty} |x(t)|^2 \, dt = \int_{0}^{2\pi} |X(f)|^2 \, df$ as the “total energy” in $x(t)$, and consequently you can interpret $\int_{t_0}^{t_1} |x(t)|^2 \, dt$ as the total energy of $x(t)$ between the time points $t_0$ and $t_1$. Similarly you can interpret $\int_{f_0}^{f_1} |X(f)|^2 \, df$ to be the energy of $x(t)$ between the frequencies $f_0$ and $f_1$. This also implies that $|x(t)|^2 = \lim_{h \to 0} \frac{1}{h} \int_{t}^{t+h} |x(u)|^2 \, du$ can be considered to be the instantaneous energy per unit time at time $t$ and $|X(f)|^2$ can be considered to be the instantaneous energy per unit frequency at $f$.

A time-frequency distribution of a time series $x(t)$ is a function $P_x(t, f)$ of the time index $t$ and frequency $f$ such that $P_x(t, f)$ is a measure of the intensity of energy of $x(t)$ at time $t$ and frequency $f$. That is, given a small $\Delta t$ and small $\Delta f$, you should be reasonably able to interpret $P_x(t, f) \Delta t \Delta f$ as the energy of $x(t)$ that can be attributed to $[x, x + \Delta t] \times [f, f + \Delta f]$.

Because energy is positive, you should expect that $P_x(t, f) \geq 0$ for all $t$ and $f$; this condition is known as positivity. Similarly you should expect $\int_{0}^{2\pi} P_x(t, f) \, df$ to yield the instantaneous energy at time $t$, so you should have $\int_{0}^{2\pi} P_x(t, f) \, df = |x(t)|^2$. Similarly you should expect $\int_{-\infty}^{\infty} P_x(t, f) \, dt = |X(f)|^2$. These two conditions are known as the marginal conditions. Other desirable properties of time-frequency distributions are discussed in Cohen (1995, chapter 6). However, it is not possible for a distribution to simultaneously satisfy positivity and the marginal conditions, and most time-frequency distributions that are used in practice satisfy these conditions only approximately.

The time-frequency analysis package implements the discrete versions of two widely used time-frequency distributions: the pseudo-Wigner-Ville distribution and the short-time Fourier transform. The continuous version of pseudo Wigner-Ville distribution satisfies the marginal property in the special case when it reduces to the Wigner-Ville distribution, but it does not satisfy the positivity condition. The short-time-Fourier transform satisfies the positivity condition, but it does not satisfy the marginal conditions.

Pseudo-Wigner-Ville Distribution

The pseudo-Wigner-Ville distribution is a generalization of the Wigner-Ville distribution. The Wigner-Ville distribution of a continuous time series $x(t)$ is obtained by computing the Fourier transform of $x(t + \tau/2)\overline{x(t - \tau/2)}$ for fixed $t$ as $\tau$ varies. So the Wigner-Ville distribution of a continuous, possibly complex-valued, time series $x(t)$ is given by $W_x(t, f)$:

$$W_x(t, f) = \int_{-\infty}^{\infty} x(t + \tau/2)\overline{x(t - \tau/2)} \exp(-2\pi i f \tau) \, d\tau$$

$$= 2 \int_{-\infty}^{\infty} x(t + \tau)\overline{x(t - \tau)} \exp(-4\pi i f \tau) \, d\tau.$$ 

It can be shown that the $W_x(t, f)$ is real even when $x(t)$ takes complex values.

The pseudo-Wigner-Ville distribution is a modification of Wigner-Ville distribution that is obtained by an additional term in the defining integral. The pseudo-Wigner-Ville distribution of a continuous time series
Chapter 10: Time-Frequency Analysis Package

\[ x(t) \text{ is given by} \]
\[ W_x(t, f) = \int w(\tau)x(t + \tau/2)x(t - \tau/2)\exp(-2\pi if\tau)\,d\tau \]

where \( w(\tau) \) is a window function.

The Wigner distribution of discrete time series \( x[k] \) is defined as follows (Claasen and Mecklenbräuker 1980b, a; Debnath 2002):

\[ W_f(n, f) = 2 \sum_{k=-\infty}^{\infty} x[n + k]\overline{x[n-k]}\exp(-4\pi ifk) \]

From the preceding formula, it follows that \( W_n(n, f/2) \) is the discrete-time Fourier transform of \( x[n + k]\overline{x[n-k]} \) and provides the basis for the computation here. This also explains why the normalized frequency in the PWV output varies from 0 to 1/2.

Given an input time series, \( x(t) \), the computation can be considered as the evaluation of a function \( U_x(n, f) \), which measures the value of the pseudo-Wigner-Ville distribution at time \( n \) and frequency \( f/2 \) for different values of \( n \) and \( f \). Now \( U_x(n, f) \) can be defined: given a possibly complex-valued time series \( x = (x[0], x[2], \ldots, x[L-1]) \) and a window of odd length \( 2m + 1 \), where \( \text{window}=(w[0], \ldots, w[2m]) \), define

\[ U_x(n, f) = \sum_{k=-m}^{m} w[m+k]x[n+k]\overline{x[n-k]}\exp(-2\pi ikf) \]

The preceding summation is performed with the following convention: any term in the summation for which both \( n + k \) and \( n - k \) do not lie between 0 and \( \text{series_length} - 1 \) is replaced with 0.

Define the following:

- \( S = \text{window_length} - \text{overlap} \)
- Let \( k = \left\{ \begin{array}{ll} \frac{\text{series_length} - 1}{S}, & \text{when } \text{fade} = 1 \\ \frac{\text{series_length} - \text{overlap}}{S}, & \text{when } \text{fade} = 0 \end{array} \right. \)
- Let \( c = \left\{ \begin{array}{ll} 0, & \text{when } \text{fade} = 0 \\ L, & \text{when } \text{fade} = 1 \end{array} \right. \)

where the \( \text{window_length} = 2L + 1 \).

The output of the PWV function consists of the evaluation of \( U_x(n, f) \) for \( n = c, c + S, c + 2S, \ldots, c + kS \) and for \( f = 0, \frac{1}{\text{fftlen}}, \ldots, \frac{\text{fftlen} - 1}{\text{fftlen}} \).
When `fade` is 0, the output has more observations, but some of the observations correspond to windows where only a portion of the data is available. When `fade` is 1, the output is restricted to windows in which all the data are used.

The pseudo-Wigner-Ville distribution has some undesirable properties. It displays annoying artifacts for multicomponent time series (Cohen 1995), and replacing the input with the analytic signal that corresponds to the input yields better results (Boashash 1988). For this reason, the TFA package provides the `hilbert_tsf` option, which replaces the original series with its analytic signal before computation, and the `center` option, which removes the mean from the series so that an overall mean effect does not show up in the output. Before any computation, the input series is first transformed depending on the value of the `hilbert_tsf` and `center` parameters: First the value of the `center` parameter is checked; if it is 1, then the input series is replaced by the centered series that is obtained by subtracting the series mean from each term of the series. If the value of the `hilbert_tsf` parameter is also 1, this possibly centered series is replaced by the analytic signal that corresponds to the centered input.

### Short-Time Fourier Transform

The short-time Fourier transform (STFT) computations consist of multiple “local” discrete Fourier transform computations. The input time series is divided into multiple contiguous blocks, and their discrete Fourier transforms are computed in succession. The use of window functions makes the spectra smooth.

Given a time series \(x[0], x[1], \ldots, x[L-1]\) and a window \(w[0], \ldots, w[m-1]\), the computation of STFT can be considered to be the evaluation of a function \(S_x(n, f)\), which measures the strength of the frequency \(f\) at time \(n\) for different values of \(n\) and \(f\), where \(S_x(n, f)\) is defined as

\[
S_x(n, f) = \sum_{k=0}^{m-1} x[n + k]w[k] \exp(-i2\pi kf)
\]

Let \(k = \left\lfloor \frac{\text{series length} - \text{overlap}}{\text{window length} - \text{overlap}} \right\rfloor + 1\) and \(S = \text{window length} - \text{overlap}\). Then STFT consists of the computation of \(S_x(n, f)\) for \(n = 0, S, 2S, \ldots, kS\) and for \(f = 0, \frac{1}{\text{fftlen}}, \frac{1}{\text{fftlen}} + 1\).

### References


# Chapter 11
## Time Series Analysis Package

<table>
<thead>
<tr>
<th>Contents</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Overview</td>
<td>354</td>
</tr>
<tr>
<td>Using CAS Sessions and CAS Engine Librefs</td>
<td>354</td>
</tr>
<tr>
<td>CORR Class</td>
<td>355</td>
</tr>
<tr>
<td>CORR Object</td>
<td>355</td>
</tr>
<tr>
<td>OUTCORR Object</td>
<td>359</td>
</tr>
<tr>
<td>CROSSCORR Class</td>
<td>362</td>
</tr>
<tr>
<td>CROSSCORR Object</td>
<td>362</td>
</tr>
<tr>
<td>OUTCROSSCORR Object</td>
<td>366</td>
</tr>
<tr>
<td>DECOMP Class</td>
<td>369</td>
</tr>
<tr>
<td>DECOMP Object</td>
<td>369</td>
</tr>
<tr>
<td>OUTDECOMP Object</td>
<td>373</td>
</tr>
<tr>
<td>OUTDECOMP2 Object</td>
<td>375</td>
</tr>
<tr>
<td>Details</td>
<td>376</td>
</tr>
<tr>
<td>DPF Class</td>
<td>378</td>
</tr>
<tr>
<td>DPF Object</td>
<td>378</td>
</tr>
<tr>
<td>OUTDPE Object</td>
<td>381</td>
</tr>
<tr>
<td>OUTDPROB Object</td>
<td>383</td>
</tr>
<tr>
<td>Example</td>
<td>384</td>
</tr>
<tr>
<td>FREQ Class</td>
<td>385</td>
</tr>
<tr>
<td>FREQ Object</td>
<td>385</td>
</tr>
<tr>
<td>OUTFREQ Object</td>
<td>389</td>
</tr>
<tr>
<td>Example</td>
<td>391</td>
</tr>
<tr>
<td>TSA Object</td>
<td>391</td>
</tr>
<tr>
<td>Functional Summary</td>
<td>391</td>
</tr>
<tr>
<td>ACCUMULATE Method</td>
<td>393</td>
</tr>
<tr>
<td>ACCUMULATE2 Method</td>
<td>395</td>
</tr>
<tr>
<td>ACF Method</td>
<td>397</td>
</tr>
<tr>
<td>ARMAORDERS Method</td>
<td>399</td>
</tr>
<tr>
<td>CCF Method</td>
<td>400</td>
</tr>
<tr>
<td>IACF Method</td>
<td>402</td>
</tr>
<tr>
<td>INTERMITTENCYTEST Method</td>
<td>403</td>
</tr>
<tr>
<td>MOVINGSUMMARY Method</td>
<td>405</td>
</tr>
<tr>
<td>PACF Method</td>
<td>408</td>
</tr>
<tr>
<td>SCALE Method</td>
<td>409</td>
</tr>
<tr>
<td>SEASONALDECOMP Method</td>
<td>411</td>
</tr>
</tbody>
</table>
Overview

The time series analysis (TSA) package contains a set of time series analysis functions that can be used as part of the programming statements in the TSMODEL procedure. This package provides a flexible way to analyze time series within the procedure.

**NOTE:** The mycas libref in the examples refers to the CAS library that is linked to a caslib. The mycas.air data table that is used in the examples refers to Sashelp.Air data. All the examples in this chapter assume that your CAS engine libref is named mycas, but you can substitute any appropriately defined CAS engine libref. For more information about CAS engine librefs, see *SAS Cloud Analytic Services: User’s Guide*.

Using CAS Sessions and CAS Engine Librefs

SAS Cloud Analytic Services (CAS) is the analytic server and associated cloud services in SAS Viya. This section describes how to create a CAS session and set up a CAS engine libref that you can use to connect to the CAS session. It assumes that you have a CAS server already available; contact your system administrator if you need help starting and terminating a server. This CAS server is identified by specifying the host on which it runs and the port on which it listens for communications. To simplify your interactions with this CAS server, the host information and port information for the server are stored as SAS option values that are retrieved automatically whenever this CAS server needs to be accessed. You can examine the host and port values for the server at your site by using the following statements:

```sas
proc options option=(CASHOST CASPORT);
run;
```

In addition to starting a CAS server, your system administrator might also have created a CAS session and a CAS engine libref for your use. You can define your own sessions and CAS engine librefs that connect to the CAS server as shown in the following statements:

```sas
cas mysess;
libname mycas cas sessref=mysess;
```

The CAS statement creates the CAS session named mysess, and the LIBNAME statement creates the mycas CAS engine libref that you use to connect to this session. It is not necessary to explicitly name the CASHOST and CASPORT of the CAS server in the CAS statement, because these values are retrieved from the corresponding SAS option values.
If you have created the `mysess` session, you can terminate it by using the `TERMINATE` option in the CAS statement as follows:

```plaintext
cas mysess terminate;
```

For more information about the CAS statement and the LIBNAME statement, see *SAS Cloud Analytic Services: User’s Guide*. For general information about CAS and CAS sessions, see *SAS Cloud Analytic Services: Fundamentals*.

---

**CORR Class**

**CORR Object**

The CORR class object functions perform the autocorrelation and autocovariance analysis of a time series array and output all the values and results.

Table 11.1 summarizes the methods that are associated with the CORR object.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GetResult</td>
<td>Get the analysis result</td>
</tr>
<tr>
<td>Initialize</td>
<td>Initialize the CORR object</td>
</tr>
<tr>
<td>Run</td>
<td>Run the analysis</td>
</tr>
<tr>
<td>SetNLags</td>
<td>Set the number of lags</td>
</tr>
<tr>
<td>SetNParms</td>
<td>Set the number of parameters</td>
</tr>
<tr>
<td>SetY</td>
<td>Set the input time series</td>
</tr>
</tbody>
</table>

Figure 11.1 diagrams the methods of the CORR object.
### CORR Synopsis

**DECLARE OBJECT obj (CORR) ;**

Method syntax, in order of typical usage:

```c
rc=obj.Initialize () ;
rc=obj.SetY (YSeries) ;
rc=obj.SetNLags (Value) ;
rc=obj.SetNParms (Value) ;
rc=obj.Run () ;
rc=obj.GetResult (’Name’,OutputArray) ;
```

### CORR Methods

**CORR.GetResult Method**

```c
rc=obj.GetResult (’Name’,OutputArray) ;
```

Outputs the selected results to an array.

**Input Arguments**  You must specify the following input arguments:
Name specifies a string that identifies the result series to return. You can specify only one of the following values within single quotation marks:

- `acov` returns an array of covariance estimates, with \( nlag + 1 \) entries.
- `acf` returns an array of autocorrelation estimates, with \( nlag + 1 \) entries.
- `acfstd` returns an array of standard errors, with \( nlag + 1 \) entries.
- `acf2std` returns an array of twice standard errors, with \( nlag + 1 \) entries.
- `acfnorm` returns an array of normalized autocorrelation, with \( nlag + 1 \) entries.
- `acfprob` returns an array of autocorrelation probabilities, with \( nlag + 1 \) entries.
- `acflprob` returns an array of autocorrelation log probabilities, with \( nlag + 1 \) entries.
- `pacf` returns an array of partial autocorrelation estimates, with \( nlag + 1 \) entries.
- `pacfstd` returns an array of partial autocorrelation standard errors, with \( nlag + 1 \) entries.
- `pacf2std` returns an array of twice standard errors, with \( nlag + 1 \) entries.
- `pacfnorm` returns an array of normalized partial autocorrelation, with \( nlag + 1 \) entries.
- `pacfprob` returns an array of partial autocorrelation probabilities, with \( nlag + 1 \) entries.
- `acflprob` returns an array of partial autocorrelation log probabilities, with \( nlag + 1 \) entries.
- `iacf` returns an array of inverse autocorrelation estimates, with \( nlag + 1 \) entries.
- `iacfstd` returns an array of inverse autocorrelation standard errors, with \( nlag + 1 \) entries.
- `iacf2std` returns an array of twice standard errors, with \( nlag + 1 \) entries.
- `iacfnorm` returns an array of normalized inverse autocorrelation, with \( nlag + 1 \) entries.
- `iacfprob` returns an array of inverse autocorrelation probabilities, with \( nlag + 1 \) entries.
- `iacflprob` returns an array of inverse autocorrelation log probabilities, with \( nlag + 1 \) entries.
- `wn` returns an array of Ljung-Box white noise tests, with \( nlag + 1 \) entries.
- `wnprob` returns white noise probabilities.
- `wnlprob` returns white noise log probabilities.

OutputArray specifies a numeric array to receive the result series.

**CORR.Initialize Method**

```cpp
rc = obj.Initialize();
```

Initializes a CORR object to an empty state. This method must be called before the time series arrays and other attributes are specified for the CORR object.

**Arguments** There are no arguments associated with this method.
**CORR.Run Method**

\[ \text{rc}=\text{obj}.\text{Run}() ; \]

Runs the CORR object to perform the autocorrelation and autocovariance analysis by using the time series array \( YSeries \) that has been specified for it. Upon successful completion, various results can be extracted from the CORR object.

**CORR.SetNLags Method**

\[ \text{rc}=\text{obj}.\text{SetNLags} (\text{Value}) ; \]

Sets a named option for the CORR object.

**Input Arguments**  You must specify the following input argument:

Value specifies the number of lags to use in the calculation. The default value is 24.

**CORR.SetNParms Method**

\[ \text{rc}=\text{obj}.\text{SetNParms} (\text{Value}) ; \]

Sets a named option for the CORR object.

**Input Arguments**  You must specify the following input argument:

Value specifies the degrees of freedom associated with the Ljung-Box statistics. The default value is 0.

**CORR.SetY Method**

\[ \text{rc}=\text{obj}.\text{SetY} (\text{YSeries}) ; \]

Specifies the time series array \( YSeries \) for the CORR object.

**Input Arguments**  You must specify the following input argument:

YSeries specifies a numeric array that contains the dependent series for the CORR object.

**Arguments**  There are no arguments associated with this method.
OUTCORR Object

The OUTCORR object collects output from a CORR object.

Table 11.2 summarizes the methods that are associated with the OUTCORR object.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Collect</td>
<td>Collect the results of a CORR object</td>
</tr>
<tr>
<td>nrows</td>
<td>Get the current row count from the OUTCORR object</td>
</tr>
</tbody>
</table>

Figure 11.2 diagrams the methods of the OUTCORR object.

OUTCORR Methods

*OUTCORR.Collect Method*

\[ rc=obj.Collect(); \]

Collects the results from a CORR object and saves the result to a CAS table.

**Arguments**  There are no arguments associated with this method.
**OUTCORR.nrows Attribute**

```csharp
nrows = obj.nrows();
```

Gets the current row count from the OUTCORR object.

**Arguments** There are no arguments associated with this method.

**Output Table Schema**

The collector object, OUTCORR, collects the results of a CORR object. OUTCORR collects the results of a CORR object. The output table contains the columns that are shown in Table 11.3.

**Table 11.3** OUTCORR Output Table Contents

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Column Content</th>
<th>Statistics Used to Calculate Content</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>NAME</em></td>
<td>Name of target variable</td>
<td>N/A</td>
</tr>
<tr>
<td>LAG</td>
<td>Array of time lags</td>
<td>( h \in {0, \ldots, H} )</td>
</tr>
<tr>
<td>N</td>
<td>Number of variance products</td>
<td>( N_h ), the number of observed products at lag ( h ), ignoring missing values</td>
</tr>
<tr>
<td>MU</td>
<td>Mean value</td>
<td>Mean value</td>
</tr>
<tr>
<td>ACOV</td>
<td>Array of covariance estimates, with ( nlag + 1 ) entries</td>
<td>( \hat{\gamma}(h) = \frac{1}{T} \sum_{t=h+1}^{T}(y_t - \bar{y})(y_{t-h} - \bar{y}) ), or ( \hat{\gamma}(h) = \frac{1}{N_h} \sum_{t=h+1}^{T}(y_t - \bar{y})(y_{t-h} - \bar{y}) ) when embedded missing values are present</td>
</tr>
<tr>
<td>ACF</td>
<td>Array of autocorrelation estimates, with ( nlag + 1 ) entries</td>
<td>( \hat{\rho}(h) = \hat{\gamma}(h)/\hat{\gamma}(0) )</td>
</tr>
<tr>
<td>ACFSTD</td>
<td>Array of standard errors, with ( nlag + 1 ) entries</td>
<td>( \text{Std}(\hat{\rho}(h)) = \sqrt{\frac{1}{T} \left( 1 + 2 \sum_{j=1}^{h-1} \hat{\rho}(j)^2 \right)} )</td>
</tr>
</tbody>
</table>
| ACF2STD     | Array of twice standard errors, with \( nlag + 1 \) entries | \( \text{Flag}(\hat{\rho}(h)) = \begin{cases} 
1 & \hat{\rho}(h) > 2\text{Std}(\hat{\rho}(h)) \\
0 & -2\text{Std}(\hat{\rho}(h)) < \hat{\rho}(h) < 2\text{Std}(\hat{\rho}(h)) \\
-1 & \hat{\rho}(h) < -2\text{Std}(\hat{\rho}(h)) 
\end{cases} \) |
| ACFNORM     | Array of normalized autocorrelation, with \( nlag + 1 \) entries | \( \text{Norm}(\hat{\rho}(h)) = \hat{\rho}(h)/\text{Std}(\hat{\rho}(h)) \) |
Table 11.3  continued

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Column Content</th>
<th>Statistics Used to Calculate Content</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACFPROB</td>
<td>Array of autocorrelation probabilities, with ( nlag + 1 ) entries</td>
<td>( \text{Prob}(\hat{\rho}(h)) = 2(1 - \Phi(</td>
</tr>
<tr>
<td>ACFLPROB</td>
<td>Array of autocorrelation log probabilities, with ( nlag + 1 ) entries</td>
<td>( \text{LogProb}(\hat{\rho}(h)) = -\log_{10}(\text{Prob}(\hat{\rho}(h))) )</td>
</tr>
<tr>
<td>PACF</td>
<td>Array of partial autocorrelation estimates, with ( nlag + 1 ) entries</td>
<td>( \hat{\phi}(h) = \Gamma_{(0,h-1)}{\gamma_j}^h_{j=1} )</td>
</tr>
<tr>
<td>PACFSTD</td>
<td>Array of partial autocorrelation standard errors, with ( nlag + 1 ) entries</td>
<td>( \text{Std}(\hat{\phi}(h)) = 1/\sqrt{N_0} )</td>
</tr>
<tr>
<td>PACF2STD</td>
<td>Array of twice standard errors, with ( nlag + 1 ) entries</td>
<td>( \text{Flag}(\hat{\phi}(h)) = \begin{cases} 1 &amp; \hat{\phi}(h) &gt; 2\text{Std}(\hat{\phi}(h)) \ 0 &amp; -2\text{Std}(\hat{\phi}(h)) &lt; \hat{\phi}(h) &lt; 2\text{Std}(\hat{\phi}(h)) \ -1 &amp; \hat{\phi}(h) &lt; -2\text{Std}(\hat{\phi}(h)) \end{cases} )</td>
</tr>
<tr>
<td>PACFNORM</td>
<td>Array of normalized partial autocorrelation, with ( nlag + 1 ) entries</td>
<td>( \text{Norm}(\hat{\rho}(h)) = \hat{\rho}(h)/\text{Std}(\hat{\rho}(h)) )</td>
</tr>
<tr>
<td>PACFPROB</td>
<td>Array of partial autocorrelation probabilities, with ( nlag + 1 ) entries</td>
<td>( \text{Prob}(\hat{\phi}(h)) = 2(1 - \Phi(</td>
</tr>
<tr>
<td>PACFLPROB</td>
<td>Array of partial autocorrelation log probabilities, with ( nlag + 1 ) entries</td>
<td>( \text{LogProb}(\hat{\phi}(h)) = -\log_{10}(\text{Prob}(\hat{\phi}(h))) )</td>
</tr>
<tr>
<td>IACF</td>
<td>Array of inverse autocorrelation estimates, with ( nlag + 1 ) entries</td>
<td>( \hat{\theta}(h) )</td>
</tr>
<tr>
<td>IACFSTD</td>
<td>Array of inverse autocorrelation standard errors, with ( nlag + 1 ) entries</td>
<td>( \text{Std}(\hat{\theta}(h)) = 1/\sqrt{N_0} )</td>
</tr>
<tr>
<td>IACF2STD</td>
<td>Array of twice standard errors, with ( nlag + 1 ) entries</td>
<td>( \text{Flag}(\hat{\theta}(h)) = \begin{cases} 1 &amp; \hat{\theta}(h) &gt; 2\text{Std}(\hat{\theta}(h)) \ 0 &amp; -2\text{Std}(\hat{\theta}(h)) &lt; \hat{\theta}(h) &lt; 2\text{Std}(\hat{\theta}(h)) \ -1 &amp; \hat{\theta}(h) &lt; -2\text{Std}(\hat{\theta}(h)) \end{cases} )</td>
</tr>
</tbody>
</table>
### Table 11.3 continued

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Column Content</th>
<th>Statistics Used to Calculate Content</th>
</tr>
</thead>
<tbody>
<tr>
<td>IACFNORM</td>
<td>Array of normalized inverse autocorrelation, with $nlag + 1$ entries</td>
<td>$\text{Norm}(\hat{\theta}(h)) = \hat{\theta}(h)/\text{Std}(\hat{\theta}(h))$</td>
</tr>
<tr>
<td>IACFPROB</td>
<td>Array of inverse autocorrelation probabilities, with $nlag + 1$ entries</td>
<td>$\text{Prob}(\hat{\theta}(h)) = 2 \left(1 - \Phi\left(</td>
</tr>
<tr>
<td>IACFLPROB</td>
<td>Array of inverse autocorrelation log probabilities, with $nlag + 1$ entries</td>
<td>$\text{LogProb}(\hat{\phi}(h)) = -\log_{10}(\text{Prob}(\hat{\phi}(h)))$</td>
</tr>
<tr>
<td>WN</td>
<td>Array of Ljung-Box white noise tests, with $nlag + 1$ entries</td>
<td>$Q(h) = T(T+2)\sum_{j=1}^{h} \rho(j)^2/(T-j)$</td>
</tr>
<tr>
<td>WNPBORB</td>
<td>White noise probabilities</td>
<td>$\text{Prob}(Q(h)) = \chi_{\max(1,h-p)}(Q(h))$</td>
</tr>
<tr>
<td>WNLPROB</td>
<td>White noise log probabilities</td>
<td>$\text{LogProb}(Q(h)) = -\log_{10}(\text{Prob}(Q(h)))$</td>
</tr>
</tbody>
</table>

---

**CROSSCORR Class**

**CROSSCORR Object**

The CROSSCORR class object functions perform the cross-correlation and cross-covariance analysis of two time series arrays and output all the values and results.

Table 11.4 summarizes the methods that are associated with the CROSSCORR object.
Table 11.4 Methods of the CROSSCORR Object

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GetResult</td>
<td>Get the analysis result</td>
</tr>
<tr>
<td>Initialize</td>
<td>Initialize the CROSSCORR object</td>
</tr>
<tr>
<td>Run</td>
<td>Run the analysis</td>
</tr>
<tr>
<td>SetNLags</td>
<td>Set the number of lags</td>
</tr>
<tr>
<td>SetX</td>
<td>Set the cross time series array</td>
</tr>
<tr>
<td>SetY</td>
<td>Set the primary time series array</td>
</tr>
</tbody>
</table>

Figure 11.3 diagrams the methods of the CROSSCORR object.
Figure 11.3 CROSSCORR Data Flow

CROSSCORR Synopsis

DECLARE OBJECT obj (CROSSCORR) ;

Method syntax, in order of typical usage:

rc=obj.Initialize () ;
rc=obj.SetY (YSeries) ;
rc=obj.SetX (XSeries) ;
rc=obj.SetNLags (Value) ;
rc=obj.Run () ;
rc=obj.GetResult ('Name',OutputArray) ;
CROSSCORR Methods

**CROSSCORR.GetResult Method**

```csharp
rc = obj.GetResult ('Name', OutputArray);
```

Outputs the selected results to an array.

**Input Arguments** You must specify the following input arguments:

- `Name` specifies a string that identifies the result series to return. You can specify the following values within single quotation marks:
  - `ccov` returns an array of cross-covariance estimates, with \(2 \times nlag + 1\) entries.
  - `ccf` returns an array of cross-correlation estimates, with \(2 \times nlag + 1\) entries.
  - `ccfstd` returns an array of standard errors, with \(2 \times nlag + 1\) entries.
  - `ccf2std` returns an array of double standard errors, with \(2 \times nlag + 1\) entries.
  - `ccfnorm` returns an array of normalized cross-correlation, with \(2 \times nlag + 1\) entries.
  - `ccfprob` returns an array of probabilities, with \(2 \times nlag + 1\) entries.
  - `ccflprob` returns an array of log probabilities, with \(2 \times nlag + 1\) entries.

- `OutputArray` specifies a numeric array to receive the result series.

**CROSSCORR.Initialize Method**

```csharp
rc = obj.Initialize();
```

Initializes an CROSSCORR object to an empty state. This method must be called before the time series arrays and other attributes are specified for the CROSSCORR object.

**Arguments** There are no arguments associated with this method.

**CROSSCORR.Run Method**

```csharp
rc = obj.Run();
```

Runs the CROSSCORR object to perform the cross-correlation and cross-covariance analysis by using the time series arrays `YSeries` and `XSeries` that have been specified for it. Upon successful completion, various results can be extracted from the CROSSCORR object.

**Arguments** There are no arguments associated with this method.

**CROSSCORR.SetNLags Method**

```csharp
rc = obj.SetNLags (Value);
```

Sets a named option for the CROSSCORR object.

**Input Arguments** You must specify the following input argument:
Value specifies the number of lags to use in the calculation. The default value is 24.

**CROSSCORR.SetX Method**

```
rc = obj.SetX (XSeries) ;
```

Specifies the cross time series array (XSeries) for the CROSSCORR object.

**Input Arguments** You must specify the following input argument:

- **XSeries** specifies a numeric array that contains the dependent series for the CROSSCORR object.

**CROSSCORR.SetY Method**

```
rc = obj.SetY (YSeries) ;
```

Specifies the primary time series array (YSeries) for the CROSSCORR object.

**Input Arguments** You must specify the following input argument:

- **YSeries** specifies a numeric array that contains the dependent series for the CROSSCORR object.

---

**OUTCROSSCORR Object**

The OUTCROSSCORR object collects output from a CROSSCORR object.

Table 11.5 summarizes the methods that are associated with the OUTCROSSCORR object.

**Table 11.5** Methods of the OUTCROSSCORR Object

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Collect</td>
<td>Collect the results of a CROSSCORR object</td>
</tr>
<tr>
<td>nrows</td>
<td>Get the current row count from the OUTCROSSCORR object</td>
</tr>
</tbody>
</table>

Figure 11.4 diagrams the methods of the OUTCROSSCORR object.
OUTCROSSCORR Methods

OUTCROSSCORR.Collect Method

\[ rc = obj\text{.Collect}() \]

Collects the results from a CROSSCORR object and saves the result to a CAS table.

**Arguments** There are no arguments associated with this method.

OUTCROSSCORR.nrows Attribute

\[ nrows = obj\text{.nrows}() \]

Gets the current row count from the OUTCROSSCORR object.

**Arguments** There are no arguments associated with this method.

Output Table Schema

The collector object, OUTCROSSCORR, collects the results of a CROSSCORR object. OUTCROSSCORR collects the results of a CROSSCORR object. The output table contains the columns that are shown in Table 11.6.
**Table 11.6** OUTCROSSCORR Output Table Contents

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Column Content</th>
<th>Statistics Used to Calculate Content</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>NAME</em></td>
<td>Variable name</td>
<td>N/A</td>
</tr>
<tr>
<td><em>CROSS</em></td>
<td>Cross variable name</td>
<td>N/A</td>
</tr>
<tr>
<td>LAG</td>
<td>Array of time lags</td>
<td>$h \in {-H, \ldots, 0, \ldots, H}$</td>
</tr>
<tr>
<td>N</td>
<td>Number of variance products</td>
<td>$N_h$, the number of observed products at lag $h$, ignoring missing values</td>
</tr>
<tr>
<td>MEAN</td>
<td>Mean of variable</td>
<td></td>
</tr>
<tr>
<td>CROSSMEAN</td>
<td>Mean of cross variable</td>
<td></td>
</tr>
<tr>
<td>CCOV</td>
<td>Array of cross-covariance estimates, with $2 \times nlag + 1$ entries</td>
<td>$\hat{\gamma}<em>{x,y}(h) =$ \begin{align*} \frac{1}{T} \sum</em>{t=h+1}^{T} (x_t - \bar{x})(y_{t-h} - \bar{y}) &amp; \quad 0 \leq h &lt; T \ \frac{1}{T} \sum_{t=</td>
</tr>
<tr>
<td>CCF</td>
<td>Array of cross-correlation estimates, with $2 \times nlag + 1$ entries</td>
<td>$\hat{\rho}<em>{x,y}(h) = \frac{\hat{\gamma}</em>{x,y}(h)}{\sqrt{\hat{\gamma}_x(0)\hat{\gamma}_y(0)}}$</td>
</tr>
<tr>
<td>CCFSTD</td>
<td>Array of cross-correlation standard errors, with $2 \times nlag + 1$ entries</td>
<td>$\text{Std}(\hat{\rho}_{x,y}(h)) = 1/\sqrt{N_0}$</td>
</tr>
<tr>
<td>CCF2STD</td>
<td>Array of indicators of sample cross-correlation outside 2 standard errors, with $2 \times nlag + 1$ entries</td>
<td>$\text{Flag}(\hat{\rho}<em>{x,y}(h)) =$ \begin{align*} 1 &amp; \quad \hat{\rho}</em>{x,y}(h) &gt; 2\text{Std}(\hat{\rho}<em>{x,y}(h)) \ 0 &amp; \quad -2\text{Std}(\hat{\rho}</em>{x,y}(h)) &lt; \hat{\rho}<em>{x,y}(h) &lt; 2\text{Std}(\hat{\rho}</em>{x,y}(h)) \ -1 &amp; \quad \hat{\rho}<em>{x,y}(h) &lt; -2\text{Std}(\hat{\rho}</em>{x,y}(h)) \end{align*}</td>
</tr>
<tr>
<td>CCFNORM</td>
<td>Array of normalized cross-correlations, with $2 \times nlag + 1$ entries</td>
<td>$\text{Norm}(\hat{\rho}<em>{x,y}(h)) = \frac{\hat{\rho}</em>{x,y}(h)}{\text{Std}(\hat{\rho}_{x,y}(h))}$</td>
</tr>
</tbody>
</table>
Table 11.6 continued

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Column Content</th>
<th>Statistics Used to Calculate Content</th>
</tr>
</thead>
<tbody>
<tr>
<td>CCFPROB</td>
<td>Array of cross-correlation probabilities, with $2 \times nlag + 1$ entries</td>
<td>$\text{Prob}(\hat{\rho}_{x,y}(h)) = 2 \left(1 - \Phi(</td>
</tr>
<tr>
<td>CCFLPROB</td>
<td>Array of cross-correlation log probabilities, with $2 \times nlag + 1$ entries</td>
<td>$\text{LogProb}(\hat{\rho}<em>{x,y}(h)) = -\log</em>{10}(\text{Prob}(\hat{\rho}_{x,y}(h)))$</td>
</tr>
</tbody>
</table>

DECOMP Class

DECOMP Object

The DECOMP class object functions perform the seasonal decomposition or analysis of the time series arrays and output all the values and results.

Table 11.7 summarizes the methods that are associated with the DECOMP object.

Table 11.7 Methods of the DECOMP Object

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GetResult</td>
<td>Get the analysis results</td>
</tr>
<tr>
<td>Initialize</td>
<td>Initialize the DECOMP object</td>
</tr>
<tr>
<td>Run</td>
<td>Run the analysis</td>
</tr>
<tr>
<td>SetLambda</td>
<td>Set the Hodrick-Prescott filter parameter for trend-cycle decomposition</td>
</tr>
<tr>
<td>SetMode</td>
<td>Set the type of decomposition</td>
</tr>
<tr>
<td>SetResults</td>
<td>Set the analysis results to be generated</td>
</tr>
<tr>
<td>SetSeasonality</td>
<td>Set the length of the seasonal cycles</td>
</tr>
<tr>
<td>SetY</td>
<td>Set the time series array</td>
</tr>
</tbody>
</table>

Figure 11.5 diagrams the methods of the DECOMP object.
**DECOMP Synopsis**

**DECLARE OBJECT** obj (DECOMP) ;

Method syntax, in order of typical usage:

\[ rc=obj.\text{Initialize} () ; \]
\[ rc=obj.\text{SetY} (\text{YSeries}) ; \]
\[ rc=obj.\text{SetMode} ('\text{Name}') ; \]
\[ rc=obj.\text{SetResults} ('\text{Name}<,\text{Name}',\ldots>) ; \]
\[ rc=obj.\text{SetSeasonality} (\text{Value}) ; \]
\[ rc=obj.\text{SetLambda} (\text{Value}) ; \]
\[ rc=obj.\text{Run} () ; \]
\[ rc=obj.\text{GetResult} ('\text{Name},\text{OutputArray}) ; \]
DECOMP Methods

**DECOMP.GetResult Method**

```c
rc = obj.GetResult ('Name', OutputArray);
```

Outputs the selected results to an array.

**Input Arguments** You must specify the following input arguments:

- **Name** specifies a string that identifies the result series to return. You can specify the following values within single quotation marks:
  - `tcc` returns the trend-cycle component.
  - `sic` returns the seasonal-irregular component.
  - `sc` returns the seasonal component.
  - `scstd` returns the seasonal component standard errors.
  - `tcs` returns the trend-cycle-seasonal component.
  - `ic` returns the irregular component.
  - `sa` returns the seasonally adjusted series.
  - `pcsa` returns the percentage of change in seasonally adjusted series.
  - `tc` returns the trend component.
  - `cc` returns the cycle component.

- **OutputArray** specifies a numeric array to receive the result series.

**DECOMP.Initialize Method**

```c
rc = obj.Initialize();
```

Initializes a DECOMP object to an empty state. This method must be called before the time series array and other attributes are specified for the DECOMP object.

**Arguments** There are no arguments associated with this method.

**DECOMP.Run Method**

```c
rc = obj.Run();
```

Runs the DECOMP object to perform the seasonal decomposition or analysis by using the time series array `YSeries` that has been specified for it. Upon successful completion, various results can be extracted from the DECOMP object.

**Arguments** There are no arguments associated with this method.

**DECOMP.SetLambda Method**

```c
rc = obj.SetLambda (Value);
```

Sets the Hodrick-Prescott filter parameter for trend-cycle decomposition.

**Input Arguments** You must specify the following input argument:
**Value** specifies the length of the seasonal cycle. The default value is 1.

**DECOMP.SetMode Method**
```
rc = obj.SetMode (Value) ;
```
Sets the type of decomposition to be used to decompose the time series.

**Input Arguments** You can specify one of the following *Values*:

- **ADD | ADDITIVE** specifies additive decomposition.
- **LOGADD | LOGADDITIVE** specifies log-additive decomposition.
- **MULT | MULTIPLICATIVE** specifies multiplicative decomposition.
- **MULTORADD** specifies multiplicative or additive decomposition, depending on the data.
- **PSEUDOADD | PSEUDOADDITIVE** specifies pseudo-additive decomposition.

**DECOMP.SetResults Method**
```
rc = obj.SetResults (’Name’, ’Name’, … ) ;
```
Set a group of results to be generated in the DECOMP object.

**Input Arguments** You must specify the following input arguments:

- **Name** specifies one or multiple strings that identify the result series to be generated. You can specify the following values within single quotation marks:
  - **tcc** specifies the trend-cycle component.
  - **sic** specifies the seasonal-irregular component.
  - **sc** specifies the seasonal component.
  - **scstd** specifies the seasonal component standard errors.
  - **tcs** specifies the trend-cycle-seasonal component.
  - **ic** specifies the irregular component.
  - **sa** specifies the seasonally adjusted series.
  - **pcsa** specifies the percentage of change in seasonally adjusted series.
  - **tc** specifies the trend component.
  - **cc** specifies the cycle component.

- **OutputArray** specifies a numeric array to receive the result series.

**DECOMP.SetSeasonality Method**
```
rc = obj.SetSeasonality (Value) ;
```
Sets the length of seasonal cycles for the DECOMP object.

**Input Arguments** You must specify the following input argument:
Value specifies the length of the seasonal cycle. The default value is 1.

**DECOMP.SetY Method**

```c
rc = obj.SetY (YSeries);
```

Specifies the time series array (YSeries) for the DECOMP object.

**Input Arguments** You must specify the following input argument:

YSeries specifies a numeric array that contains the dependent series for the DECOMP object.

---

**OUTDECOMP Object**

The OUTDECOMP object collects output from DECOMP object.

Table 11.8 summarizes the methods that are associated with the OUTDECOMP object.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Collect</td>
<td>Collect the results of a DECOMP object</td>
</tr>
<tr>
<td>nrows</td>
<td>Get the current row count from the OUTDECOMP object</td>
</tr>
</tbody>
</table>

Figure 11.6 diagrams the methods of the OUTDECOMP object.

**Figure 11.6** OUTDECOMP Data Flow
OUTDECOMP Methods

**OUTDECOMP.Collect Method**

```plaintext
rc = obj.Collect();
```

Collects the results from a DECOMP object and saves them to a CAS table.

**Arguments**

There are no arguments associated with this method.

**OUTDECOMP.nrows Attribute**

```plaintext
nrows = obj.nrows();
```

Gets the current row count from the OUTDECOMP object.

**Arguments**

There are no arguments associated with this method.

**Output Table Schema**

The collector object, OUTDECOMP, collects the results of a DECOMP object.

OUTDECOMP collects the results of a DECOMP object. The output table contains the columns that are shown in Table 11.9.

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Column Content</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>NAME</em></td>
<td>Variable name</td>
</tr>
<tr>
<td><em>MODE</em></td>
<td>Mode of decomposition</td>
</tr>
<tr>
<td><em>TIMEID</em></td>
<td>Time ID values</td>
</tr>
<tr>
<td><em>SEASON</em></td>
<td>Seasonal index</td>
</tr>
<tr>
<td>ORIGINAL</td>
<td>Original series</td>
</tr>
<tr>
<td>TCC</td>
<td>Trend-cycle component</td>
</tr>
<tr>
<td>SIC</td>
<td>Seasonal-irregular component</td>
</tr>
<tr>
<td>SC</td>
<td>Seasonal component</td>
</tr>
<tr>
<td>SCSTD</td>
<td>Seasonal component standard error</td>
</tr>
<tr>
<td>TCS</td>
<td>Trend-cycle-seasonal component</td>
</tr>
<tr>
<td>IC</td>
<td>Irregular component</td>
</tr>
<tr>
<td>SA</td>
<td>Seasonally adjusted series</td>
</tr>
<tr>
<td>PCSA</td>
<td>Percentage change in seasonally adjusted series</td>
</tr>
<tr>
<td>TC</td>
<td>Trend component</td>
</tr>
<tr>
<td>CC</td>
<td>Cycle component</td>
</tr>
</tbody>
</table>

For more information about each component, see Table 11.12.
OUTDECOMP2 Object

The OUTDECOMP2 object collects output from DECOMP object. In this object, the time periods are recorded as the column names instead of decomposition components.

Table 11.10 summarizes the methods that are associated with the OUTDECOMP2 object.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Collect</td>
<td>Collect the results of a DECOMP object</td>
</tr>
<tr>
<td>nrows</td>
<td>Get the current row count from the OUTDECOMP2 object</td>
</tr>
</tbody>
</table>

Figure 11.7 diagrams the methods of the OUTDECOMP2 object.

OUTDECOMP2 Methods

`OUTDECOMP2.Collect Method`

```plaintext
rc=obj.Collect () ;
```

Collects the results from a DECOMP object and saves them to a CAS table.

Arguments There are no arguments associated with this method.
**OUTDECOMP2.nrows Attribute**

```plaintext
nrows=obj.nrows () ;
```

Gets the current row count from the OUTDECOMP2 object.

**Arguments**

There are no arguments associated with this method.

**Output Table Schema**

The collector object, OUTDECOMP2, collects the results of a DECOMP object.

OUTDECOMP2 collects the results of a DECOMP object. The output table contains the columns that are shown in Table 11.11.

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Column Content</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>NAME</em></td>
<td>Variable name</td>
</tr>
<tr>
<td><em>MODE</em></td>
<td>Mode of decomposition</td>
</tr>
<tr>
<td>COMPONENT</td>
<td>Component Name</td>
</tr>
<tr>
<td>PERIOD</td>
<td>Period</td>
</tr>
</tbody>
</table>

**Details**

You can use the DECOMP object to perform seasonal decomposition (also called seasonal adjustment) on the working series. The DECOMP object uses classical decomposition. More complex seasonal decomposition or adjustment analysis can be performed by using the X11 or the X12 procedure of SAS/ETS.

The SetMode method determines the mode of the seasonal adjustment decomposition to be performed. There are four modes: multiplicative (MULT), additive (ADD), pseudo-additive (PSEUDOADD), and log-additive (LOGADD) decomposition (in which the components are exponentiated to the original metric). The default value for the SetMode method is MULTORADD, which uses multiplicative mode for series that are strictly positive, pseudo-additive mode for series that are nonnegative, and additive mode for series that are not nonnegative.

The SetLambda method specifies the Hodrick-Prescott filter parameter (Hodrick and Prescott 1980) for trend-cycle decomposition. The Hodrick-Prescott filter is used to decompose the trend-cycle component into the trend component and the cycle component in an additive fashion. The SetLambda method takes a single argument `Value`, which specifies the length of the seasonal cycle. A smaller `Value` assigns less significance to the cycle; that is, a value of 0 implies no cycle component. The default value is 1600.

Table 11.12 shows the notation and keywords that are associated with seasonal decomposition (adjustment) analysis.
### Table 11.12  Seasonal Adjustment Formulas

<table>
<thead>
<tr>
<th>Component</th>
<th>Input Argument</th>
<th>SetMode Value</th>
<th>Formula</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original series</td>
<td>original</td>
<td>MULT</td>
<td>( O_t = TC_t S_t I_t )</td>
</tr>
<tr>
<td></td>
<td>ADD</td>
<td></td>
<td>( O_t = TC_t + S_t + I_t )</td>
</tr>
<tr>
<td></td>
<td>LOGADD</td>
<td></td>
<td>( \log(O_t) = TC_t + S_t + I_t )</td>
</tr>
<tr>
<td></td>
<td>PSEUDOADD</td>
<td></td>
<td>( O_t = TC_t (S_t + I_t - 1) )</td>
</tr>
<tr>
<td>Trend-cycle component</td>
<td>tcc</td>
<td>MULT</td>
<td>Centered moving average of ( O_t )</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ADD</td>
<td>Centered moving average of ( O_t )</td>
</tr>
<tr>
<td></td>
<td></td>
<td>LOGADD</td>
<td>Centered moving average of ( \log(O_t) )</td>
</tr>
<tr>
<td></td>
<td></td>
<td>PSEUDOADD</td>
<td>Centered moving average of ( O_t )</td>
</tr>
<tr>
<td>Seasonal-irregular component</td>
<td>sic</td>
<td>MULT</td>
<td>( SI_t = S_t I_t = O_t/TC_t )</td>
</tr>
<tr>
<td></td>
<td>ADD</td>
<td></td>
<td>( SI_t = S_t + I_t = O_t - TC_t )</td>
</tr>
<tr>
<td></td>
<td>LOGADD</td>
<td></td>
<td>( SI_t = S_t + I_t = \log(O_t) - TC_t )</td>
</tr>
<tr>
<td></td>
<td>PSEUDOADD</td>
<td></td>
<td>( SI_t = S_t + I_t - 1 = O_t/TC_t )</td>
</tr>
<tr>
<td>Seasonal component</td>
<td>sc</td>
<td>MULT</td>
<td>Seasonal averages of ( SI_t )</td>
</tr>
<tr>
<td></td>
<td>ADD</td>
<td></td>
<td>Seasonal averages of ( SI_t )</td>
</tr>
<tr>
<td></td>
<td>LOGADD</td>
<td></td>
<td>Seasonal averages of ( SI_t )</td>
</tr>
<tr>
<td></td>
<td>PSEUDOADD</td>
<td></td>
<td>Seasonal averages of ( SI_t )</td>
</tr>
<tr>
<td>Irregular component</td>
<td>ic</td>
<td>MULT</td>
<td>( I_t = SI_t/S_t )</td>
</tr>
<tr>
<td></td>
<td>ADD</td>
<td></td>
<td>( I_t = SI_t - S_t )</td>
</tr>
<tr>
<td></td>
<td>LOGADD</td>
<td></td>
<td>( I_t = SI_t - S_t )</td>
</tr>
<tr>
<td></td>
<td>PSEUDOADD</td>
<td></td>
<td>( I_t = SI_t - S_t + 1 )</td>
</tr>
<tr>
<td>Trend-cycle-seasonal component</td>
<td>tcs</td>
<td>MULT</td>
<td>( TCS_t = TC_t S_t = O_t/I_t )</td>
</tr>
<tr>
<td></td>
<td>ADD</td>
<td></td>
<td>( TCS_t = TC_t + S_t = O_t - I_t )</td>
</tr>
<tr>
<td></td>
<td>LOGADD</td>
<td></td>
<td>( TCS_t = TC_t + S_t = O_t - I_t )</td>
</tr>
<tr>
<td></td>
<td>PSEUDOADD</td>
<td></td>
<td>( TCS_t = TC_t S_t )</td>
</tr>
<tr>
<td>Trend component</td>
<td>tc</td>
<td>MULT</td>
<td>( T_t = TC_t - C_t )</td>
</tr>
<tr>
<td></td>
<td>ADD</td>
<td></td>
<td>( T_t = TC_t - C_t )</td>
</tr>
<tr>
<td></td>
<td>LOGADD</td>
<td></td>
<td>( T_t = TC_t - C_t )</td>
</tr>
<tr>
<td></td>
<td>PSEUDOADD</td>
<td></td>
<td>( T_t = TC_t - C_t )</td>
</tr>
<tr>
<td>Cycle component</td>
<td>cc</td>
<td>MULT</td>
<td>( C_t = TC_t - T_t )</td>
</tr>
<tr>
<td></td>
<td>ADD</td>
<td></td>
<td>( C_t = TC_t - T_t )</td>
</tr>
<tr>
<td></td>
<td>LOGADD</td>
<td></td>
<td>( C_t = TC_t - T_t )</td>
</tr>
<tr>
<td></td>
<td>PSEUDOADD</td>
<td></td>
<td>( C_t = TC_t - T_t )</td>
</tr>
<tr>
<td>Seasonally adjusted series</td>
<td>sa</td>
<td>MULT</td>
<td>( SA_t = O_t/S_t = TC_t I_t )</td>
</tr>
<tr>
<td></td>
<td>ADD</td>
<td></td>
<td>( SA_t = O_t - S_t = TC_t + I_t )</td>
</tr>
<tr>
<td></td>
<td>LOGADD</td>
<td></td>
<td>( SA_t = O_t/\exp(S_t) = \exp(TC_t + I_t) )</td>
</tr>
<tr>
<td></td>
<td>PSEUDOADD</td>
<td></td>
<td>( SA_t = TC_t I_t )</td>
</tr>
</tbody>
</table>

When \( s \) is odd, the trend-cycle component is computed from the \( s \)-period-centered moving average as follows:

\[
TC_t = \sum_{k=-[s/2]}^{[s/2]} y_{t+k}/s
\]

When \( s \) is even, the trend-cycle component is computed from the \( s \)-period-centered moving average as
follows:

\[ T C_t = \sum_{k=-s/2}^{s/2-1} \frac{(y_{t+k} + y_{t+1+k})}{2s} \]

The seasonal component is obtained by averaging the seasonal-irregular component for each season,

\[ S_{k+j} = \sum_{t=k \mod s}^{T/s} \frac{SI_t}{T/s} \]

where \(0 \leq j \leq T/s\) and \(1 \leq k \leq s\). The seasonal components are normalized to sum to 1 (multiplicative) or 0 (additive).

**DPF Class**

**DPF Object**

The DPF class object functions take the output of the FREQ class object as input. The DPF class object performs count distribution analysis for a time series.

Table 11.13 summarizes the methods that are associated with the DPF object.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initialize</td>
<td>Initialize the DPF object</td>
</tr>
<tr>
<td>Run</td>
<td>Run the analysis</td>
</tr>
<tr>
<td>SetOption</td>
<td>Set the specifications</td>
</tr>
</tbody>
</table>

Figure 11.8 diagrams the methods of the DPF object.
**DPF Synopsis**

**DECLARE OBJECT** obj (DPF) ;

Method syntax, in order of typical usage:

\[ rc = obj.\text{Initialize} (\text{FREQObj}) ; \]
\[ rc = obj.\text{SetOption} (\text{`Name'}, \text{Value}) ; \]
\[ rc = obj.\text{Run} () ; \]
DPF Methods

**DPF.Initialize Method**

```plaintext
rc = obj.Initialize (FREQObj);
```

Specifies a FREQ object, which includes frequency analysis results. This method must be called before the time series arrays and other attributes are specified for the DPF object.

**Input Arguments**

You must specify a FREQ object, which includes frequency analysis results.

- **FREQObj** specifies the FREQ object as input.

---

**DPF.Run Method**

```plaintext
rc = obj.Run();
```

Runs the DPF object to perform the count distribution analysis. Upon successful completion, various results can be extracted from the DPF object.

---

**DPF.SetOption Method**

```plaintext
rc = obj.SetOption ('Name', Value);
```

Sets a named option for the DPF object.

**Input Arguments**

You can specify one of the following for 'Name' and its associated Value:

- **ALPHA** specifies the confidence level size, where alpha must be between 0 and 1. The default value is 0.05.
- **CONVERGE** specifies the convergence criterion.
- **MAXITER** specifies the maximum number of iterations, where Value is an integer.
- **'SELECT'** takes a string Value that specifies the distribution selection criterion. You can specify one of the following Values:
  - **AIC** specifies Akaike’s information criterion.
  - **BIC** specifies the Bayesian information criterion.
  - **LOGLIK** specifies the log-likelihood.

The default value is LOGLIK.

- **'METHOD'** takes a string Value that specifies the candidate distribution to use in the analysis. You can specify one of the following Values:
  - **BEST** specifies the best distribution, based on the value of the select argument.
  - **BINOMIAL** specifies the binomial distribution.
  - **GEOMETRIC** specifies the geometric distribution.
  - **NEGBINOMIAL** specifies the negative binomial distribution.
OUTDPE Object

The OUTDPE object collects output from a DPF object.

Table 11.14 summarizes the methods that are associated with the OUTDPE object.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Collect</td>
<td>Collect the results of a DPF object</td>
</tr>
<tr>
<td>nrows</td>
<td>Get the current row count from the OUTDPE object</td>
</tr>
</tbody>
</table>

Figure 11.9 diagrams the methods of the OUTDPF object.
OUTDPE Methods

**OUTDPE.Collect Method**

\[ rc = obj.Collect(); \]

Collects the results from a DPE object and saves the result to a CAS table.

**Arguments**  There are no arguments associated with this method.

**OUTDPE.nrows Attribute**

\[ nrows = obj.nrows(); \]

Gets the current row count from the OUTDPE object.

**Arguments**  There are no arguments associated with this method.

**Output Table Schema**

OUTDPE collects predictions using the count distribution that has the best fit. The output table contains the columns that are shown in Table 11.15.

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Column Content</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>NAME</em></td>
<td>Name of target variable</td>
</tr>
<tr>
<td>Distribution</td>
<td>Distribution</td>
</tr>
<tr>
<td>Parameter</td>
<td>Name of parameter</td>
</tr>
<tr>
<td>Estimate</td>
<td>Parameter estimate</td>
</tr>
<tr>
<td>StdErr</td>
<td>Standard error</td>
</tr>
<tr>
<td>Tvalue</td>
<td>t value</td>
</tr>
<tr>
<td>Probt</td>
<td>Approximate probability &gt;</td>
</tr>
<tr>
<td>Lower</td>
<td>Lower 95% of parameter estimate</td>
</tr>
<tr>
<td>Upper</td>
<td>Upper 95% of parameter estimate</td>
</tr>
</tbody>
</table>
OUTDPROB Object

The OUTDPROB object collects output from a DPF object.

Table 11.16 summarizes the methods that are associated with the OUTDPROB object.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Collect</td>
<td>Collect the results of a DPF object</td>
</tr>
<tr>
<td>nrows</td>
<td>Get the current row count from the OUTDPROB object</td>
</tr>
</tbody>
</table>

Figure 11.10 diagrams the methods of the OUTDPROB object.

**Figure 11.10** OUTDPROB Data Flow

OUTDPROB Methods

*OUTDPROB.Collect Method*

```c
rc = obj.Collect();
```

Collects the results from a DPE object and saves the result to a CAS table.

**Arguments** There are no arguments associated with this method.
**OUTDPROB.nrows Attribute**

```plaintext
nrows = obj.nrows();
```

Gets the current row count from the OUTDPROB object.

**Arguments**  There are no arguments associated with this method.

**Output Table Schema**

OUTDPROB collects predictions using the count distribution that has the best fit. The output table contains the columns that are shown in Table 11.17.

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Column Content</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>NAME</em></td>
<td>Name of target variable</td>
</tr>
<tr>
<td>Index</td>
<td>Index</td>
</tr>
<tr>
<td>Value</td>
<td>Values</td>
</tr>
<tr>
<td>Observedzeros</td>
<td>Observed zeros</td>
</tr>
<tr>
<td>Expectedzeros</td>
<td>Expected zeros</td>
</tr>
<tr>
<td>Observed</td>
<td>Observed counts</td>
</tr>
<tr>
<td>Expected</td>
<td>Expected counts</td>
</tr>
<tr>
<td>Expectedlower</td>
<td>Expected lower confidence</td>
</tr>
<tr>
<td>Expectedupper</td>
<td>Expected upper confidence</td>
</tr>
<tr>
<td>Probability</td>
<td>Probability</td>
</tr>
<tr>
<td>Probabilitylower</td>
<td>Probability lower confidence</td>
</tr>
<tr>
<td>Probabilityupper</td>
<td>Probability upper confidence</td>
</tr>
<tr>
<td>Chisquare</td>
<td>Chi-square statistic</td>
</tr>
<tr>
<td>Chisquareprob</td>
<td>Chi-square probability</td>
</tr>
<tr>
<td>Chisquarelogprob</td>
<td>Chi-square log probability</td>
</tr>
</tbody>
</table>

**Example**

This example uses the TSMODEL procedure to analyze the frequency of the time series Air and then conducts a count distribution analysis.

The following DATA step loads the Sashelp.Air data set onto the CAS server. This DATA step assumes that your CAS engine libref is named mycas, but you can substitute any appropriately defined CAS engine libref.

```plaintext
data mycas.air (replace=yes);
  set Sashelp.air;
run;
```
FREQ Object

The FREQ class object function analyzes the frequency of a time series and outputs all unique values and corresponding counts for the time series.

Table 11.18 summarizes the methods that are associated with the FREQ object.
Table 11.18 Methods of the FREQ Object

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GetResult</td>
<td>Get the analysis result</td>
</tr>
<tr>
<td>Initialize</td>
<td>Initialize the FREQ object</td>
</tr>
<tr>
<td>Nvalues</td>
<td>Return the number of unique values in the input series</td>
</tr>
<tr>
<td>Run</td>
<td>Run the analysis</td>
</tr>
<tr>
<td>SetOption</td>
<td>Set the specifications</td>
</tr>
<tr>
<td>SetY</td>
<td>Set the input time series</td>
</tr>
</tbody>
</table>

Figure 11.11 diagrams the methods of the FREQ object.
**FREQ Synopsis**

```c
DECLARE OBJECT obj (FREQ) ;
```

Method syntax, in order of typical usage:

```c
rc=obj.Initialize () ;
rc=obj.SetY (YSeries) ;
rc=obj.SetOption ('Name',Value) ;
rc=obj.Run () ;
rc=obj.GetResult ('Name',OutputArray) ;
rc=obj.Nvalues () ;
```
FREQ Methods

FREQ.GetResult Method

\[ rc = \text{obj}.\text{GetResult} (\text{'Name'}, \text{OutputArray}) ; \]

Outputs the selected results to an array.

**Input Arguments**
You must specify the following input arguments:

- **Name** specifies a string that identifies the result series to return. You can specify the following values within single quotation marks:
  - 'VALUES' retrieves a monotonically increasing list of the unique values that are found in the input series that is specified by using the SetY method.
  - 'COUNTS' retrieves the number of occurrences of each unique value.
  - 'PCTS' retrieves the percentage of total occurrences of each unique value.

- **OutputArray** specifies a numeric array to receive the result series.

FREQ.Initialize Method

\[ rc = \text{obj}.\text{Initialize} () ; \]

Initializes an FREQ object to an empty state. This method must be called before the time series arrays and other attributes are specified for the FREQ object.

**Arguments**
There are no arguments associated with this method.

FREQ.Nvalues Method

\[ rc = \text{obj}.\text{Nvalues} () ; \]

Returns the number of unique values in the input series to \( rc \). This number is also the length of all result series that are returned by the GetResult method. It is set to a missing value if the Run method did not complete successfully.

**Arguments**
There are no arguments associated with this method.

FREQ.Run Method

\[ rc = \text{obj}.\text{Run} () ; \]

Runs the FREQ object to perform the autocorrelation and autocovariance analysis by using the time series array \( Y\text{Series} \) that has been specified for it. Upon successful completion, various results can be extracted from the FREQ object.

FREQ.SetOption Method

\[ rc = \text{obj}.\text{SetOption} (\text{'Name'}, \text{Value}) ; \]

Sets a named option for the FREQ object.
**Input Arguments**
You can specify the following for 'Name' and its associated Value:

'SEASONALITY' specifies the seasonality of the time series, where Value is an integer. The default value is 0.

**FREQ.SetY Method**

\[ rc = \text{obj}.\text{SetY} (\text{YSeries}) \];

Specifies the time series array (YSeries) for the FREQ object.

**Input Arguments** You must specify the following input argument:

YSeries specifies the time series array to analyze.

---

**OUTFREQ Object**

The OUTFREQ object collects output from a FREQ object.

Table 11.19 summarizes the methods that are associated with the OUTFREQ object.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Collect</td>
<td>Collect the results of a FREQ object</td>
</tr>
<tr>
<td>nrows</td>
<td>Get the current row count from the OUTFREQ object</td>
</tr>
</tbody>
</table>

Figure 11.12 diagrams the methods of the OUTFREQ object.
OUTFREQ Methods

**OUTFREQ.Collect Method**

```rc = obj.Collect();```

Collects the results from a FREQ object and saves the result to a CAS table.

**Arguments**  There are no arguments associated with this method.

**OUTFREQ.nrows Attribute**

```nrows = obj.nrows();```

Gets the current row count from the OUTFREQ object.

**Arguments**  There are no arguments associated with this method.

**Output Table Schema**

The collector object, OUTFREQ, collects the results of frequency function.

OUTFREQ collects the results of frequency function. The output table contains the columns that are shown in Table 11.20.


**Example**

This example uses the TSMODEL procedure to analyze the frequency of the time series Air:

```plaintext
proc tsmodel data=mycas.air outobj=(of=mycas.outfreq(replace=YES));
  var air;
  id date interval=month;
  require tsa;
submit;
  declare object f(FREQ);
  declare object of(OUTFREQ);
  rc = f.Initialize();
  rc = f.SetOption("SEASONALITY",12);
  rc = f.SetY(air);
  rc = f.Run();
  rc = of.Collect(f);
endsubmit;
run;
```

**TSA Object**

**Functional Summary**

**NOTE:** Each function in this section has a prefix of “TSA.”; however, the prefixes are omitted in descriptions for better readability.

Table 11.21 summarizes the functions in the TSA package.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACCUMULATE</td>
<td>Accumulates a univariate time series to a particular frequency</td>
</tr>
<tr>
<td>ACCUMULATE2</td>
<td>Accumulates a high-frequency time series to a lower frequency and expands the lower-frequency series to have the same length as the high-frequency series</td>
</tr>
</tbody>
</table>

---

**Table 11.20**  OUTFREQ Output Table Contents

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Column Content</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>NAME</em></td>
<td>Name of target variable</td>
</tr>
<tr>
<td>Values</td>
<td>Array of series values</td>
</tr>
<tr>
<td>Counts</td>
<td>Array of frequency series counts</td>
</tr>
<tr>
<td>Percent</td>
<td>Percentage of total frequency</td>
</tr>
</tbody>
</table>
### Table 11.21 continued

<table>
<thead>
<tr>
<th>TSA Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACF</td>
<td>Computes autocorrelation and autocovariance for a time series array</td>
</tr>
<tr>
<td>ARMAORDERS</td>
<td>Performs tests to tentatively identify the autoregressive and moving average orders of mixed autoregressive moving average models</td>
</tr>
<tr>
<td>CCF</td>
<td>Computes the cross-correlation and cross-covariance for two time series arrays</td>
</tr>
<tr>
<td>DPF Class Object</td>
<td>Performs count distribution analysis for time series</td>
</tr>
<tr>
<td>FREQ Class Object</td>
<td>Performs frequency analysis of a time series</td>
</tr>
<tr>
<td>INTERMITTENCYTEST</td>
<td>Tests for intermittency of a univariate time series</td>
</tr>
<tr>
<td>IACF</td>
<td>Computes the inverse autocorrelation for a time series array</td>
</tr>
<tr>
<td>MOVINGSUMMARY</td>
<td>Computes statistics for a set of values within a moving time window</td>
</tr>
<tr>
<td>PACF</td>
<td>Computes the partial autocorrelation for a time series array</td>
</tr>
<tr>
<td>SCALE</td>
<td>Scales a time series between the minimum value and the maximum value of the original time series</td>
</tr>
<tr>
<td>SEASONALDECOMP</td>
<td>Computes the seasonal indices of a univariate time series using classical decomposition</td>
</tr>
<tr>
<td>SEASONALINDICES</td>
<td>Computes the seasonal indices of a univariate time series by using regression seasonal dummies</td>
</tr>
<tr>
<td>SEASONTEST</td>
<td>Tests for seasonality of a univariate time series</td>
</tr>
<tr>
<td>SIMILARITY</td>
<td>Performs similarity analysis for time series</td>
</tr>
<tr>
<td>STATIONARITYTEST</td>
<td>Tests for stationarity of a univariate time series</td>
</tr>
<tr>
<td>TRANSFORM</td>
<td>Transforms time series according to the specified transformation type</td>
</tr>
<tr>
<td>UNBIASEDNESS</td>
<td>Tests whether a univariate time series is unbiased</td>
</tr>
<tr>
<td>WHITENOISE</td>
<td>Tests for white noise of a time series array</td>
</tr>
</tbody>
</table>
ACCUMULATE Method

\[ rc = \text{TSA.ACCUMULATE} \left( \text{time}, y, \text{interval}, id, z, <\text{accumulate}>, <\text{setmiss}>, <\text{zeromiss}> \right) \]

The ACCUMULATE function accumulates a univariate time series to a particular frequency.

**Required Arguments**

You must specify the following arguments, separated by commas:

- **time** specifies the time ID array for the time series.
- **y** specifies the times series array to accumulate.
- **'interval'** specifies the time interval.

You can specify the following values within single quotation marks:

- **DAY** specifies a seasonal cycle of length 7.
- **MONTH** specifies a seasonal cycle of length 12.
- **QTR** specifies a seasonal cycle of length 4.

**Optional Arguments**

You can also specify the following arguments, separated by commas. If you want to use a default value for any of these arguments, enter a space for it.

- **'accumulate'** specifies the accumulation statistic.

You can specify the following values within single quotation marks:

- **AVERAGE | AVG** specifies the average of the values in the time series.
- **CSS** specifies the corrected sum of squares of the values in the time series.
- **FIRST** specifies the first value of the time series.
- **LAST** specifies the last value of the time series.
- **MAXIMUM | MAX** specifies the maximum value in the time series.
- **MEDIAN | MED** specifies the median of the values in the time series.
- **MINIMUM | MIN** specifies the minimum value in the time series.
- **N** specifies the number of nonmissing observations.
- **NMISS** specifies the number of missing observations.
- **NOBS** specifies the number of observations.
- **STDDEV | STD** specifies the standard deviation of the values in the time series.
- **TOTAL** specifies the total sum of the values in the time series.
- **USS** specifies the uncorrected sum of squares of the values in the time series.

The default is TOTAL.

- **'setmiss'** specifies the missing value interpretation.

You can specify the following values within single quotation marks:
 Chapter 11: Time Series Analysis Package

AVERAGE | AVG specifies the accumulated average value.
FIRST specifies the accumulated first nonmissing value.
MAXIMUM | MAX specifies the accumulated maximum value.
MEDIAN | MED specifies the accumulated median value.
MINIMUM | MIN specifies the accumulated minimum value.
MISSING specifies a missing value.
NEXT specifies the next period’s accumulated nonmissing value. Missing values at the end of the accumulated series remain missing.
PREVIOUS | PREV specifies the previous period’s accumulated nonmissing value. Missing values at the beginning of the accumulated series remain missing.

The default is MISSING.

'zeromiss' specifies the zero value interpretation.
You can specify the following values within single quotation marks:

BOTH sets both beginning and ending zeros to missing.
LEFT sets beginning zeros to missing.
NONE leaves beginning and ending zeros unchanged.
RIGHT sets ending zeros to missing.

The default is NONE.

Returned Values

The ACCUMULATE function returns the following values:

rc returns one of the following scalar return codes:

<table>
<thead>
<tr>
<th>rc</th>
<th>Termination Reason</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Success</td>
</tr>
<tr>
<td>&lt; 0</td>
<td>Computational failure</td>
</tr>
</tbody>
</table>

id returns the time ID array for the accumulated time series.

z returns the accumulated time series array.

Example

This example uses the TSMODEL procedure to accumulate a time series:
ACCUMULATE2 Method

```
proc tsmode data=mycas.air outarray=mycas.outarray;
  id date interval=month;
  var air;
  outarrays qtravg new_id;
  require tsa;
  submit;
    declare object TSA(tsa);
    rc=TSA.ACCUMULATE(date, air, 'QTR', new_id, qtravg, 'AVERAGE', , );
  endsymbol;
run;
```

**ACCUMULATE2 Method**

\[
rc = \text{TSA.ACCUMULATE2} (time, y, 'interval', id, z, <'accumulate'>, <'setmiss'>, <'zeromiss'>) ;
\]

The ACCUMULATE2 function accumulates a high-frequency time series to a lower frequency and expands
the lower-frequency time series to the same length as the high-frequency series.

**Required Arguments**

You must specify the following arguments, separated by commas:

- `time` specifies the time ID array for the time series.
- `y` specifies the times series array to accumulate.
- `'interval'` specifies the time interval.

You can specify the following values within single quotation marks:

- `DAY` specifies a seasonal cycle of length 7.
- `MONTH` specifies a seasonal cycle of length 12.
- `QTR` specifies a seasonal cycle of length 4.

**Optional Arguments**

You can also specify the following arguments, separated by commas. If you want to use a default value for
any of these arguments, enter a space for it.

- `'accumulate'` specifies the accumulation statistic.

You can specify the following values within single quotation marks:

- `AVERAGE | AVG` specifies the average of the values in the time series.
- `CSS` specifies the corrected sum of squares of the values in the time series.
- `FIRST` specifies the first value of the time series.
- `LAST` specifies the last value of the time series.
- `MAXIMUM | MAX` specifies the maximum value in the time series.
**MEDIAN | MED** specifies the median of the values in the time series.

**MINIMUM | MIN** specifies the minimum value in the time series.

**N** specifies the number of nonmissing observations.

**NMISS** specifies the number of missing observations.

**NOBS** specifies the number of observations.

**STDDEV | STD** specifies the standard deviation of the values in the time series.

**TOTAL** specifies the total sum of the values in the time series.

**USS** specifies the uncorrected sum of squares of the values in the time series.

The default is **TOTAL**.

'**setmiss**' specifies the missing value interpretation.

You can specify the following values within single quotation marks:

**AVERAGE | AVG** specifies the accumulated average value.

**FIRST** specifies the accumulated first nonmissing value.

**LAST** specifies the accumulated last nonmissing value.

**MAXIMUM | MAX** specifies the accumulated maximum value.

**MEDIAN | MED** specifies the accumulated median value.

**MINIMUM | MIN** specifies the accumulated minimum value.

**MISSING** specifies a missing value.

**NEXT** specifies the next period’s accumulated nonmissing value. Missing values at the end of the accumulated series remain missing.

**PREVIOUS | PREV** specifies the previous period’s accumulated nonmissing value. Missing values at the beginning of the accumulated series remain missing.

The default is **MISSING**.

'**zeromiss**' specifies the zero value interpretation.

You can specify the following values within single quotation marks:

**BOTH** sets both beginning and ending zeros to missing.

**LEFT** sets beginning zeros to missing.

**NONE** leaves beginning and ending zeros unchanged.

**RIGHT** sets ending zeros to missing.

The default is **NONE**.
Returned Values

The ACCUMULATE2 function returns the following values:

- \(rc\) returns one of the following scalar return codes:

<table>
<thead>
<tr>
<th>(rc)</th>
<th>Termination Reason</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Success.</td>
</tr>
<tr>
<td>&lt; 0</td>
<td>Computational failure.</td>
</tr>
</tbody>
</table>

- \(id\) returns the time ID array for the accumulated time series.
- \(z\) returns the accumulated time series array.

Example

This example uses the TSMODEL procedure to accumulate a monthly time series into a yearly time series:

```plaintext
proc tsmodel data=mycas.air outarray=mycas.outarray;
  id date interval=month;
  var air;
  outarrays yearavg_expand yearid_expand;
  require tsa;
  submit;
  declare object TSA(tsa);
  rc=TSA.ACCUMULATE2(date, air, 'YEAR', yearid_expand, yearavg_expand,
                     'AVERAGE', , );
  endsubmit;
run;
```

ACF Method

\[
rc = TSA.ACF (y, nlag, lags, df, < mu>, < acov>, < acf>, < acfstd>, < acf2std>, < acfnorm>, < acfprob>
< acflprob>);
\]

The ACF function computes autocorrelation and autocovariance for a time series array.

Required Arguments

You must specify the following arguments, separated by a comma:

- \(y\) specifies the times series array.
- \(nlag\) specifies the number of the lag to use in the calculation.
Returned Values

The ACF function returns the following values:

- \( rc \) returns one of the following scalar return codes:
  
<table>
<thead>
<tr>
<th>( rc )</th>
<th>Termination Reason</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Success</td>
</tr>
<tr>
<td>&lt; 0</td>
<td>Computational failure</td>
</tr>
</tbody>
</table>

- \( lags \) returns the number of the lags that were used in the calculation.
- \( df \) returns the number of observations used to compute \( acov \) and \( acf \).

Optional Returned Values

You can also specify the following arguments, separated by commas to request additional returned values. If you do not want the value to be returned, enter a space for it.

- \( mu \) returns the mean estimate.
- \( acov \) returns an array of covariance estimates, with \( nlag+1 \) entries.
- \( acf \) returns an array of autocorrelation estimates, with \( nlag+1 \) entries.
- \( acfstd \) returns an array of standard errors, with \( nlag+1 \) entries.
- \( acf2std \) returns an array of twice standard errors, with \( nlag+1 \) entries.
- \( acfnorm \) returns an array of normalized autocorrelation, with \( nlag+1 \) entries.
- \( acfprob \) returns an array of autocorrelation probabilities, with \( nlag+1 \) entries.
- \( acfprob \) returns an array of autocorrelation log probabilities, with \( nlag+1 \) entries.

Example

This example uses the TS MODEL procedure to compute the autocorrelation of lag 3 of the time series \( Air \):

```plaintext
proc tsmodel data=mycas.air outscalar=mycas.outscalars
  outarray=mycas.outarray;
  id date interval=month;
  var air;
  outscalars mu;
  outarrays acf acov lags df acfstd;
  require tsa;
  submit;
  declare object TSA(tsa);
  rc=TSA.ACF(air, 3, lags, df, mu, acov, acf, acfstd, , , , );
  endsubmit;
run;
```
The ARMAORDERS function performs tests to tentatively identify the autoregressive and moving average orders of mixed autoregressive moving average models.

**Required Arguments**

You must specify the following argument:

\[ y \]

specifies the times series array to test.

**Optional Arguments**

You can also specify the following arguments, separated by commas. If you want to use a default value for any of these arguments, enter a space for it.

\[ \text{dif} \]

specifies either an array of positive integers or a positive integer that is used for differencing. The default value is 0.

\[ \text{'method'} \]

specifies the method of tentative order selection. You can specify the following values within single quotation marks:

- **ESACF** specifies the extended sample autocorrelation function.
- **MINIC** specifies the minimum information criterion.
- **SCAN** specifies the squared canonical correlations.

The default value is MINIC.

\[ p \]

specifies the autoregressive order range, where \( p \) is an array of two nonnegative integers that define the minimum and maximum values.

By default, \( p \) is the array [0,5].

\[ q \]

specifies the moving average order range, where \( q \) is an array of two nonnegative integers that define the minimum and maximum values.

By default, \( q \) is the array [0,5].

\[ \text{perror} \]

specifies the autoregressive orders used to estimate the error series for the MINIC method, where \( \text{perror} \) is an array of two nonnegative integers that define the minimum and maximum values.

By default, \( \text{perror} \) is the array \([\max(p), \max(p)+\max(q)]\).
Returned Values

The ARMAORDERS function returns the following values:

- **rc** returns one of the following scalar return codes:
  
<table>
<thead>
<tr>
<th>rc</th>
<th>Termination Reason</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Success</td>
</tr>
<tr>
<td>&lt; 0</td>
<td>Computational failure</td>
</tr>
</tbody>
</table>

- **porders** returns the recommended autoregressive orders.

- **qorders** returns the recommended moving average orders.

Example

This example uses the TSMODEL procedure to tentatively identify the autoregressive and moving average orders for the time series **Air**:

```
proc tsmodel data=mycas.air outscalar=mycas.outscalars;
   id date interval=month;
   var air;
   outscalars porders qorders;
   require tsa;
   submit;
   declare object TSA(tsa);
   porders = 0;
   qorders = 0;
   Array P[2]/nospaces; P[1]=0; P[2]=5;
   Array Q[2]/nospaces; Q[1]=0; Q[2]=5;
   rc=TSA.ARMAORDERS(air, 0, 'SCAN', P, Q, , porders, qorders);
   rc=TSA.ARMAORDERS(air, 1, 'ESACF', P, Q, , porders, qorders);
   rc=TSA.ARMAORDERS(air, 1, 'MINIC', P, Q, , porders, qorders);
   endsubmit;
run;
```

CCF Method

- **rc** = TSA.CCF (y, x, nlag, lags, df, < ymu>, < xmu>, < ccov>, < ccf>, < ccfstd>, < ccf2std>, < ccfnorm>, < ccfprob>, < ccflprob>)

The CCF function computes the cross-correlation and cross-covariance for two time series arrays.
Required Arguments
You must specify the following arguments, separated by commas:

\[ y \] specifies one time series array.
\[ x \] specifies the other time series array.
\[ nlag \] specifies the number of the lag to compute.

Returned Values
The CCF function returns the following values:

\[ rc \] returns one of the following scalar return codes:

<table>
<thead>
<tr>
<th>rc</th>
<th>Termination Reason</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Success</td>
</tr>
<tr>
<td>&lt; 0</td>
<td>Computational failure</td>
</tr>
</tbody>
</table>

\[ lags \] returns an array of lags that were computed, with \( nlag + 1 \) entries.
\[ df \] returns an array of number of products for which to compute the cross-correlation, with \( nlag + 1 \) entries.

Optional Returned Values
You can also specify the following arguments, separated by commas to request additional returned values. If you do not want the value to be returned, enter a space for it.

\[ ymu \] returns the mean estimate of input time series \( y \).
\[ xmu \] returns the mean estimate of input time series \( x \).
\[ ccov \] returns an array of cross-covariance estimates, with \( 2 \times nlag + 1 \) entries.
\[ ccf \] returns an array of cross-correlation estimates, with \( 2 \times nlag + 1 \) entries.
\[ ccfstd \] returns an array of standard errors, with \( 2 \times nlag + 1 \) entries.
\[ ccf2std \] returns an array of double standard errors, with \( 2 \times nlag + 1 \) entries.
\[ ccfnorm \] returns an array of normalized cross-correlation, with \( 2 \times nlag + 1 \) entries.
\[ ccfprob \] returns an array of probabilities, with \( 2 \times nlag + 1 \) entries.
\[ ccflprob \] returns an array of log probabilities, with \( 2 \times nlag + 1 \) entries.
Example

This example uses the TSMODEL procedure to compute the cross-correlation and cross-covariance of two time series arrays (Price and Sale) with lag 20:

```
data mycas.pricedata (replace=yes);
  set Sashelp.pricedata;
run;

proc tsmodel data=mycas.pricedata outarray=mycas.ccf_array
  outscalar=mycas.ccf_scalar;
  id date interval=month;
  var price sale;
  by region line product;
  outscalars ymu xmu;
  outarrays lags df ccov ccf ccf2std ccfnorm ccfprob ccflprob;
  require tsa;
  submit;
    declare object TSA(tsa);
    rc=TSA.CCF(price, sale, 20, lags, df, ymu, xmu, ccov, ccf, ccf2std, ccfnorm, ccfprob, ccflprob);
  endsubmit;
run;
```

IACF Method

```
rc = TSA.IACF (y, nlag, lags, df, <mu>, <iacf>, <iacfstd>, <iacf2std>, <iacfnorm>, <iacfprob>, <iacflprob>);
```

The IACF function computes the inverse autocorrelation for a time series array.

Required Arguments

You must specify the following arguments, separated by a comma:

- **y** specifies the times series array.
- **nlag** specifies the number of the lag to use in the calculation.

Returned Values

The IACF function returns the following values:

- **rc** returns one of the following scalar return codes:

<table>
<thead>
<tr>
<th>rc</th>
<th>Termination Reason</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Success</td>
</tr>
<tr>
<td>&lt; 0</td>
<td>Computational failure</td>
</tr>
</tbody>
</table>

- **lags** returns the number of the lag that was used in the calculation.
\( df \) returns the number of observations used to compute \( iacf \).

**Optional Returned Values**

You can also specify the following arguments, separated by commas to request additional returned values. If you do not want the value to be returned, enter a space for it.

- \( mu \) returns the mean estimate.
- \( iacf \) returns an array of inverse autocorrelation estimates, with \( nlag+1 \) entries.
- \( iacfstd \) returns an array of inverse autocorrelation standard errors, with \( nlag+1 \) entries.
- \( iacf2std \) returns an array of twice standard errors, with \( nlag+1 \) entries.
- \( iacfnorm \) returns an array of normalized inverse autocorrelation, with \( nlag+1 \) entries.
- \( iacfprob \) returns an array of inverse autocorrelation probabilities, with \( nlag+1 \) entries.
- \( iacf1prob \) returns an array of inverse autocorrelation log probabilities, with \( nlag+1 \) entries.

**Example**

This example uses the TSMODEL procedure to compute the inverse autocorrelation of lag 3 of the time series \( Air \):

```plaintext
proc tsmodel data=mycas.air outscalar=mycas.outscalars
    outarray=mycas.outarray;
    id date interval=month;
    var air;
    outscalars mu;
    outarrays iacf lags df iacfstd;
    require tsa;
    submit;
    declare object TSA(tsa);
    rc=TSA.IACF(air, 3, lags, df, mu, iacf, iacfstd, , , , );
    endsheet;
run;
```

**INTERMITTENCYTEST Method**

\( rc = TSA.INTERMITTENCYTEST (y, base, threshold, med) \);

The INTERMITTENCYTEST function tests for intermittency of a univariate time series by computing the median of the length of contiguous constant periods (demand intervals).
Required Arguments

You must specify the following arguments, separated by commas:

- \( y \) specifies the times series array to test. The test is applied to the last 100 values.
- \( base \) specifies the base value to test. The value is typically 0.
- \( threshold \) specifies the threshold value for intermittency. The value is typically greater than 2.

Returned Values

The INTERMITTENCYTEST function returns the following values:

- \( rc \) returns one of the following scalar return codes:

<table>
<thead>
<tr>
<th>( rc )</th>
<th>Termination Reason</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Time series is not intermittent</td>
</tr>
<tr>
<td>1</td>
<td>Time series is intermittent</td>
</tr>
<tr>
<td>&lt; 0</td>
<td>Computational failure</td>
</tr>
</tbody>
</table>

- \( med \) returns the median length of the contiguous constant periods.

Example

This example uses the TSMODEL procedure to test the intermittency on the time series array Air:

```plaintext
proc tsmodel data=mycas.air outscalar=mycas.outscalars;
  id date interval=month;
  var air;
  outscalars intermittent;
  require tsa;
  submit;
  declare object TSA(tsa);
  intermittent=0;
  rc=TSA.INTERMITTENCYTEST(air, 0, 2, med);
  if rc>0 then intermittent= 1;
  endsubmit;
run;
```
MOVINGSUMMARY Method

\[
rc = \text{TSA.MOVINGSUMMARY} (y, 'method', k, <lead>, <w>, <'setmiss'>, <'abs'>, x, <p>, <nmiss>) ;
\]

The MOVINGSUMMARY function computes statistics for a set of values within a moving time window.

Required Arguments

You must specify the following arguments, separated by commas:

- \( y \) specifies the input time series array.
- \('method'\) specifies the statistic to calculate for each output array, \( x_t \), based on the elements of the \( y \) input array in the \( t \) window.
  
  You can specify the following methods within single quotation marks:

  - \text{EWMA} calculates the exponentially weighted moving average.
  - \text{GMEAN} calculates the moving geometric mean.
  - \text{MAX} calculates the maximum value.
  - \text{MEAN} calculates the moving average.
  - \text{MED} calculates the median value.
  - \text{MIN} calculates the minimum value.
  - \text{PROD} calculates the moving product.
  - \text{RANGE} calculates the maximum value minus minimum value.
  - \text{SUM} calculates the moving sum.
  - \text{TVALUE} calculates the standard deviation divided by mean.
  - \text{VAR} calculates the variance of the sample defined by the window around \( t \).

- \( k \) specifies the window size, where \( k \) is a positive integer. When the method is EWMA, \( k \) is set to 1 and defaults are used for all other arguments.

Optional Arguments

You can also specify the following arguments, separated by commas. If you want to use a default value for any of these arguments, enter a space for it.

- \( lead \) specifies the number of leading terms, where \( lead \) is a nonnegative integer less than \( k \). You can specify the following values:

  - 0 specifies the backward moving summary.
  - \( k/2 \) specifies the centered moving summary.
  - \( k-1 \) specifies the forward moving summary.

  The default value is 0. When the method is EWMA, \( lead \) is set to 0.
\( w \) specifies an array of weights that has \( k \) elements (a scalar when \( k=1 \)). This argument is required for the EWMA method, and it must be a scalar between 0 and 1, inclusive. This argument is optional for the MEAN, PROD, TVALUE, and VAR methods and is not supported for all other methods.

\( \text{'setmiss'} \) specifies how missing values are interpreted.

You can specify the following values within single quotation marks:

- **IGNORE** specifies that missing values have no effect on the summary.
- **MEAN** specifies that missing values are replaced with the mean of the remaining nonmissing values in the window. This value is supported only for the method SUM.
- **MISSING** specifies that if the input window contains a missing value, the output value is also missing.

The default value is IGNORE.

\( \text{'abs'} \) specifies how the series is transformed into nonnegative values prior to performing the moving summary.

You can specify the following values within single quotation marks:

- **OFF** specifies no modification. This value is not supported for the GMEAN method, because the geometric mean is undefined for negative values in the series.
- **ON** transforms each member of the series into its absolute value.
- **SQUARE** transforms each member of the series into its square

The default value is ON when the method is GMEAN and is OFF for all other methods.

**Returned Values**

The MOVINGSUMMARY function returns the following values:

- \( rc \) returns one of the following scalar return codes:

<table>
<thead>
<tr>
<th>rc</th>
<th>Termination Reason</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>Warping limits relaxed</td>
</tr>
<tr>
<td>3</td>
<td>Expansion limits relaxed</td>
</tr>
<tr>
<td>2</td>
<td>Compression limits relaxed</td>
</tr>
<tr>
<td>1</td>
<td>Warping limits imposed</td>
</tr>
<tr>
<td>0</td>
<td>Success</td>
</tr>
<tr>
<td>&lt; 0</td>
<td>Computational failure</td>
</tr>
</tbody>
</table>

- \( x \) returns the transformed series.
Optional Returned Values

You can also specify the following arguments, separated by commas to request additional returned values. If you do not want the value to be returned, enter a space for it.

- **p** returns an array in which element $t$ is the number of products that contributed to element $t$ of $x$. The $p$ argument is supported only when method is PROD or GMEAN.
- **nmiss** returns the number of missing values that are generated.

Examples

This example uses the TSMODEL procedure to compute the five-period moving average of the time series array Air:

```sas
proc tsmodel data=mycas.air outscalar=mycas.scalars outarray=mycas.arrays;
  id date interval=month;
  var air;
  outarrays x p;
  outscalars rc nmiss;
  require tsa;
  submit;
  declare object TSA(tsa);
  rc=TSA.MOVINGSUMMARY(air, 'MEAN' , 5, 0 , , , x, p, nmiss);
  endsubmit;
run;
```

This example uses the TSMODEL procedure to compute the five-period centered weighted moving product of the time series array Air:

```sas
proc tsmodel data=mycas.air outscalar=mycas.scalars outarray=mycas.arrays;
  id date interval=month;
  var air;
  outarrays x;
  outscalars rc;
  require tsa;
  submit;
  declare object TSA(tsa);
  Array w[5]/nosymbols; w[1]=0.3; w[2]=0.2; w[3]=0.25; w[4]=0.1; w[5]=0.15;
  rc=TSA.MOVINGSUMMARY(air, 'PROD' , 5, 2.5 , w, , , x, , );
  endsubmit;
run;
```
PACF Method

\[ rc = TSA.PACF( y, nlag, lags, df, < mu>, < pacf>, < pacfstd>, < pacf2std>, < pacfnorm>, < pacfprob>, < pacfprob>); \]

The PACF function computes the partial autocorrelation for a time series array.

Required Arguments

You must specify the following arguments, separated by a comma:

- \( y \) specifies the times series array.
- \( nlag \) specifies the number of the lag to use in the calculation.

Returned Values

The PACF function returns the following values:

- \( rc \) returns one of the following scalar return codes:
  - \( rc \) Termination Reason
    - 0 Success
    - < 0 Computational failure

- \( lags \) returns the number of the lag that was used in the calculation.
- \( df \) returns the number of observations used to compute \( pacf \).

Optional Returned Values

You can also specify the following arguments, separated by commas to request additional returned values. If you do not want the value to be returned, enter a space for it.

- \( mu \) returns the mean estimate.
- \( pacf \) returns an array of partial autocorrelation estimates, with \( nlag+1 \) entries.
- \( pacfstd \) returns an array of partial autocorrelation standard errors, with \( nlag+1 \) entries.
- \( pacf2std \) returns an array of twice standard errors, with \( nlag+1 \) entries.
- \( pacfnorm \) returns an array of normalized partial autocorrelation, with \( nlag+1 \) entries.
- \( pacfprob \) returns an array of partial autocorrelation probabilities, with \( nlag+1 \) entries.
- \( pacfprob \) returns an array of partial autocorrelation log probabilities, with \( nlag+1 \) entries.
Example

This example uses the TSMODEL procedure to compute the autocorrelation of lag 3 of the time series Air:

```sas
proc tsmode data=mycas.air outscalar=mycas.outscalars
   outarray=mycas.outarray;
   id date interval=month;
   var air;
   outscalars mu;
   outarrays pacf lags df pacfstd;
   require tsa;
   submit;
   declare object TSA(tsa);
   rc=TSA.PACF(air, 3, lags, df, mu, pacf, pacfstd, , , , );
   endsubmit;
run;
```

SCALE Method

```
rc = TSA.SCALE (y, min, max, nomiss, x, <nmiss>);
```

The SCALE function scales a time series between a specified minimum value and a specified maximum value.

Required Arguments

You must specify the following arguments, separated by commas:

- **y** specifies the input time series array.
- **min** specifies the minimum value in the output array.
- **max** specifies the maximum value in the output array.
- **nomiss** specifies how missing values are treated. You can specify the following values:
  - **0** allows missing values in the input array.
  - **1** does not allow missing values in the input array. If missing values exist, the output array $x_t$ becomes missing for all values of $t$.

The default is 0.

Returned Values

The SCALE function returns the following values:

- **rc** returns one of the following scalar return codes:
Chapter 11: Time Series Analysis Package

<table>
<thead>
<tr>
<th>rc</th>
<th>Termination Reason</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>The input series is nearly constant</td>
</tr>
<tr>
<td>4</td>
<td>Missing values were found when the value 1 was specified for nomiss</td>
</tr>
<tr>
<td>3</td>
<td>One or more arguments are ignored</td>
</tr>
<tr>
<td>2</td>
<td>One or more arguments are set to the default value</td>
</tr>
<tr>
<td>1</td>
<td>The input series is all missing</td>
</tr>
<tr>
<td>-1</td>
<td>One or more arguments are not supported</td>
</tr>
<tr>
<td>-2</td>
<td>The minimum value of the transformed series is greater than its maximum value</td>
</tr>
<tr>
<td>-4</td>
<td>Extreme slope</td>
</tr>
<tr>
<td>-99</td>
<td>Bad arguments</td>
</tr>
</tbody>
</table>

\( x \) returns the transformed series.

**Optional Returned Values**

You can also specify the following argument to request an additional returned value:

\( \text{nomiss} \) returns the number of missing values that are generated.

**Example**

This example uses the TSMODEL procedure to scale the time series array \( \text{Air} \) between a minimum value of 0 and a maximum value of 100:

```plaintext
proc tsmodel data=mycas.air outarray=mycas.scale_array;
    id date interval=month;
    var air;
    outarrays t1;
    require tsa;
    submit;
    declare object TSA(tsa);
    rc=TSA.SCALE(air, 0, 100, , t1, );
    endsubmit;
run;
```
SEASONALDECOMP Method

\[ rc = \text{TSA.SEASONALDECOMP} \left( y, s, \text{"mode"}, \lambda, \text{tcc}, \text{sic}, \text{sc}, \text{scstd}, \text{tcs}, \text{ic}, \text{sa}, \text{pcsa}, \text{tc}, \text{cc} \right) ; \]

The SEASONALDECOMP function computes the seasonal indices of a univariate time series by using classical decomposition.

**Required Arguments**

You must specify the following arguments, separated by commas:

- \( y \) specifies the times series array to decompose.
- \( s \) specifies the seasonality to test, where \( s \) must be either a positive integer or _SEASONALITY_, which is the length of the seasonal cycle as specified by the SEASONALITY= option in the PROC TSMODEL statement or implied by the INTERVAL= option in the ID statement.
- \text{"mode"} specifies the type of decomposition to be used to decompose the time series.

You can specify the following values within single quotation marks:

- \text{ADD | ADDITIVE} specifies additive decomposition.
- \text{LOGADD | LOGADDITIVE} specifies log-additive decomposition.
- \text{MULT | MULTIPLICATIVE} specifies multiplicative decomposition.
- \text{MULTORADD} specifies multiplicative or additive decomposition, depending on data.
- \text{PSEUDOADD | PSEUDOADDITIVE} specifies pseudo-additive decomposition.

**Optional Arguments**

You can also specify the following argument, separated by a comma from arguments that precede it. If you want to use a default value for this argument, enter a space for it.

- \( \lambda \) specifies the Hodrick-Prescott filter parameter for trend-cycle decomposition. The default value is 1,600. Filtering applies when the trend component or the cycle component is requested. If filtering is not specified, this option is ignored.

**Returned Values**

The SEASONALDECOMP function returns the following values:

- \( rc \) returns one of the following scalar return codes:

<table>
<thead>
<tr>
<th>( rc )</th>
<th>Termination Reason</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Success</td>
</tr>
<tr>
<td>&lt; 0</td>
<td>Computational failure</td>
</tr>
</tbody>
</table>
Optional Returned Values

You can also specify the following arguments, separated by commas to request additional returned values. If you do not want the value to be returned, enter a space for it.

- **tcc**: specifies the trend-cycle component.
- **sic**: specifies the seasonal-irregular component.
- **sc**: specifies the seasonal component.
- **scstd**: specifies the seasonal component standard errors.
- **tcs**: specifies the trend-cycle-seasonal component.
- **ic**: specifies the irregular component.
- **sa**: specifies the seasonally adjusted series.
- **pcsa**: specifies the percentage of change in seasonally adjusted series.
- **tc**: specifies the trend component.
- **cc**: specifies the cycle component.

Example

This example uses the TSMODEL procedure to compute the seasonal indices on the time series array Air:

```plaintext
proc tsmodel data=mycas.air outarray=mycas.outarray;
  id date interval=month;
  var air;
  outarrays ADJUSTED;
  require tsa;
  submit;
  declare object TSA(tsa);
  rc=TSA.SEASONALDECOMP(air, _SEASONALITY_, 'ADD', , , , , , , , ADJUSTED, , , );
  endsubmit;
run;
```

SEASONALINDICES Method

```plaintext
rc = TSA.SEASONALINDICES (y, s, <'mode'>, <'term'>, indices);
```

The SEASONALINDICES function computes the seasonal indices of a univariate time series by using regression seasonal dummies.

Required Arguments

You must specify the following arguments, separated by a comma:

- **y**: specifies the times series array.
- **s**: specifies the seasonality to test, where **s** must be either a positive integer or _SEASONALITY_, which is the length of the seasonal cycle as specified by the SEASONALITY= option in the PROC TSMODEL statement or implied by the INTERVAL= option in the ID statement.
Optional Arguments

You can also specify the following arguments, separated by commas. If you want to use a default value for any of these arguments, enter a space for it.

'mode' specifies the type of model to be used in the regression.
You can specify the following values within single quotation marks:

ADD | ADDITIVE uses an additive model.
MULT | MULTIPLICATIVE uses a multiplicative model.

The default method is ADD.

'term' specifies the type of terms to be used in the regression.
You can specify the following values within single quotation marks:

S uses only seasonal dummies terms.
SC uses only seasonal dummies and constant terms.
ST uses only seasonal dummies and trend terms.
STC uses seasonal dummies, trend, and constant terms.
STQ uses seasonal dummies, trend, and quadratic terms.
STQC uses seasonal dummies, trend, quadratic, and constant terms.

The default value is S. Quadratic values can be used only in the additive model.

Returned Values

The SEASONALINDICES function returns the following values:

rc returns one of the following scalar return codes:

<table>
<thead>
<tr>
<th>rc</th>
<th>Termination Reason</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Success</td>
</tr>
<tr>
<td>&lt; 0</td>
<td>Computational failure</td>
</tr>
</tbody>
</table>

indices returns an array of seasonal indices.

Example

This example uses the TSMODEL procedure to compute the seasonal indices of the time series Air:

```
proc tsmodel data=mycas.air outarray=mycas.outarray;
   id date interval=month;
   var air;
   outarrays sindices;
   require tsa;
   submit;
```
```
declare object TSA(tsa);
rc=TSA.SEASONALINDICES(air, _SEASONALITY_, 'ADD', 'STQC', sindices);
endsubmit;
run;
```

**SEASONTEST Method**

\[ rc = \text{TSA.SEASONTEST} (y, s, <\text{dif}>, <p>, <\text{alpha}>, <\text{aic}>) \]

The SEASONTEST function tests whether a univariate time series is seasonal by comparing two time series models: one seasonal and one nonseasonal.

**Required Arguments**

You must specify the following arguments, separated by a comma:

- `y` specifies the times series array to test.
- `s` specifies the seasonality to test, where `s` must be either a positive integer or `_SEASONALITY_`, which is the length of the seasonal cycle as specified by the `SEASONALITY=` option in the PROC TSMODEL statement or implied by the `INTERVAL=` option in the ID statement.

**Optional Arguments**

You can also specify the following arguments, separated by commas. If you want to use a default value for any of these arguments, enter a space for it.

- `dif` specifies an array of positive integers or a positive integer that is used for differencing. The default value is 0.
- `p` specifies the autoregressive order (0 or 1). The default value is 0.
- `alpha` specifies the significance level. The default value is 0.01.

**Returned Values**

The SEASONTEST function returns the following values:

- `rc` returns one of the following scalar return codes:

<table>
<thead>
<tr>
<th>rc</th>
<th>Termination Reason</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Time series is not seasonal</td>
</tr>
<tr>
<td>1</td>
<td>Time series is seasonal</td>
</tr>
<tr>
<td>&lt; 0</td>
<td>Computational failure</td>
</tr>
</tbody>
</table>
Optional Returned Values

You can also specify the following arguments, separated by commas to request additional returned values. If you do not want the value to be returned, enter a space for it.

\( aic \) returns an array of three values: Akaike’s information criterion (AIC) for the nonseasonal model, AIC for seasonal model, and the \( p \)-value for the \( F \) test.

Example

The following example uses the TSMODEL procedure to test the seasonality of the time series array Air:

```plaintext
proc tsmode data=mycas.air outscalar=mycas.outscalars
  outarray=mycas.outarray;
  id date interval=month;
  var air;
  outscalars seasonal;
  outarrays aic;
  require tsa;
  submit;
  declare object TSA(tsa);
  seasonal=0;
  rc=TSA.SEASONTEST(air, _SEASONALITY_, 0, 1, , aic); /*- no detrending -*/
  if rc>0 then seasonal= 1;
  rc=TSA.SEASONTEST(air, _SEASONALITY_, 1, 1, 0.05, ); /*- detrending -*/
  if rc>0 then seasonal= 1;
endsubmit;
run;
```

SIMILARITY Method

\( rc = TSA.SIMILARITY(x, y, \text{'type'}, \text{'scale'}, \text{expandpct}, \text{expandabs}, \text{compresspct}, \text{compressabs}, \text{measure}) ; \)

The SIMILARITY function analyzes the similarity between two time series.

Required Arguments

You must specify the following arguments, separated by a comma:

\( x \) specifies the input time series array to be compared to the target time series.
\( y \) specifies the target time series array to be compared to the input time series.
Optional Arguments

You can also specify the following arguments, separated by commas. If you want to use a default value for any of these arguments, enter a space for it.

'type' specifies the similarity measure.
You can specify the following values within single quotation marks:

- **ABSDEV** specifies the absolute deviation.
- **MABSDEV** specifies the mean absolute deviation.
- **MABSDEVINP** specifies the mean absolute deviation relative to the length of the input sequence.
- **MABSDEVMAX** specifies the mean absolute deviation relative to the maximum valid path length.
- **MABSDEVMIN** specifies the mean absolute deviation relative to the minimum valid path length.
- **MABSDEVTAR** specifies the mean absolute deviation relative to the length of the target sequence.
- **MSQRDEV** specifies the mean squared deviation.
- **MSQRDEVINP** specifies the mean squared deviation relative to the length of the input sequence.
- **MSQRDEVMAX** specifies the mean squared deviation relative to the maximum valid path length.
- **MSQRDEVMIN** specifies the mean squared deviation relative to the minimum valid path length.
- **MSQRDEVTAR** specifies the mean squared deviation relative to the length of the target sequence.
- **SQRDEV** specifies the squared deviation.

The default value is SQRDEV.

'scale' specifies how the working input sequence is scaled with respect to the working target sequence. Scaling is performed after normalization.
You can specify the following values within single quotation marks:

- **ABS** applies absolute scaling.
- **NONE** applies no scaling.
- **STD** applies standard scaling.

The default value is NONE.

expandpct specifies the warping expansion as a percentage of the length of the target sequence, where expandpct ranges from 0 to 100, 0 implies no compression, and 100 implies maximum allowable compression. The default value is 100.

expandabs specifies the absolute warping expansion, where expandabs is an integer that ranges from 0 to 10,000. The default is the maximum allowable absolute expansion.
**compresspct** specifies the warping compression as a percentage of the length of the target sequence, where **compresspct** ranges from 0 to 100, 0 implies no compression, and 100 implies maximum allowable compression. The default value is 100.

**compressabs** specifies the absolute warping compression, where **compressabs** is an integer that ranges from 0 to 10,000. The default is the maximum allowable absolute compression.

**Returned Values**

The **SIMILARITY** function returns the following values:

- **rc** returns one of the following scalar return codes:

<table>
<thead>
<tr>
<th>rc</th>
<th>Termination Reason</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>Warping limits relaxed</td>
</tr>
<tr>
<td>3</td>
<td>Expansion limits relaxed</td>
</tr>
<tr>
<td>2</td>
<td>Compression limits relaxed</td>
</tr>
<tr>
<td>1</td>
<td>Warping limits imposed</td>
</tr>
<tr>
<td>0</td>
<td>Success</td>
</tr>
<tr>
<td>&lt; 0</td>
<td>Computational failure</td>
</tr>
</tbody>
</table>

- **measure** returns the similarity measure.

**Example**

This example uses the **TSMODEL** procedure to compute the similarity of two time series arrays: x and y.

```plaintext
data test;
  input i x1 x2 x3 y1 y2 y3 r;
datalines;
  1 3 2 4 2 3 2 1
  2 5 4 5 4 5 3 1
  3 3 3 4 6 4 5 1
  4 3 6 6 7 6 7 1
  5 3 5 5 3 5 7 1
  6 6 6 6 8 8 8 1
  7 3 8 5 9 9 8 1
  8 8 9 8 3 7 3 1
  9 6 7 6 8 4 9 1
  10 7 9 8 9 6 7 1
;run;

data mycas.testsim;
  set test;
run;

proc tsmodel data=mycas.testsim
  outscalar=mycas.sim_scalar
  nthreads=1;
require tsa;
id i interval=day;
```
var x1 y1;
outscalars measure;
submit;
declare object TSA(tsa);
rc = TSA.SIMILARITY(x1, y1, 'absdev', 'NONE', , , , , measure);
endsubmit;
run;

---

**STATIONARITYTEST Method**

\[ rc = \text{TSA.STATIONARITYTEST} \left( y, \left< \text{dif}, \ldots, \text{d}, \text{p}, \text{type}, \text{pvalue} \right> \right) ; \]

The STATIONARITYTEST function tests for stationarity of a univariate time series.

**Required Arguments**

You must specify the following argument:

\[ y \]

specifies the times series array to test.

**Optional Arguments**

You can also specify the following arguments, separated by commas. If you want to use a default value for any of these arguments, enter a space for it.

\[ \text{dif} \]

specifies an array of positive integers or a positive integer that is used for differencing. The default value is 0.

\[ \text{d} \]

specifies the order of unit root \( (d = 1, \ldots, 12) \). If the \text{type} is SSM, then \( d = 1 \). The default value is 1.

\[ \text{p} \]

specifies the autoregressive order, where \( p \) must be a nonnegative integer. The default value is 5.

\[ \text{type} \]

specifies the type of test statistic used.

You can specify the following values within single quotation marks:

\[ \text{SSM} \]

specifies the studentized test statistic for the single mean (intercept) case.

\[ \text{STR} \]

specifies the studentized test statistic for the deterministic time trend case.

\[ \text{SZM} \]

specifies the studentized test statistic for the zero mean (no intercept) case. This value is allowed only when \( d \) has a value of 1.

The default value of \text{type} is SZM.
Returned Values

The STATIONARITYTEST function returns the following values:

\( rc \) returns one of the following scalar return codes:

<table>
<thead>
<tr>
<th>( rc )</th>
<th>Terminaison Reason</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Time series is stationary with the default significance level of 0.05</td>
</tr>
<tr>
<td>1</td>
<td>Time series is not stationary with the default significance level of 0.05</td>
</tr>
<tr>
<td>&lt; 0</td>
<td>Computational failure</td>
</tr>
</tbody>
</table>

\( pvalue \) returns the probability value associated with the test.

Example

This example uses the TSMODEL procedure to test the stationarity on the time series array Air:

```plaintext
proc tsmodel data=mycas.air outscalar=mycas.outscalars;
   id date interval=month;
   var air;
   outscalars stationary1 stationary2;
   require tsa;
   submit;
   declare object TSA(tsa);
   stationary1=1; stationary2=1;
   rc = TSA.STATIONARITYTEST(air,,,pvalue);
   *test with the default significant level=0.05;
   if rc =1 then stationary1 = 0;
   *test with significant level = 0.1;
   if pvalue > 0.1 then stationary2 = 0;
   endsubmit;
run;
```

TRANSFORM Method

\( rc = TSA.TRANSFORM (y, <'type'>, <inverse>, <c>, x) ; \)

The TRANSFORM function transforms a time series to another form.

Required Arguments

You must specify the following arguments:

\( y \) specifies an input time series array.
Optional Arguments

You can also specify the following arguments, separated by commas. If you want to use a default value for any of these arguments, enter a space for it.

- **type** specifies the type of transformation. You can specify the following values within single quotation marks:
  - LOG specifies logarithmic transformation.
  - SQRT specifies square root transformation.
  - LOGIT specifies logit transformation.
  - BOXCOX specifies Box-Cox transformation.
  - NONE requests that no transformation be performed.

  The default value is NONE.

- **inverse** specifies whether to perform an inverse transformation. You can specify the following values:
  - 0 does not perform an inverse transformation.
  - 1 returns the inverse of the specified transformation method.

  The default value is 0.

- **c** specifies a parameter to be used in the transformation. Its use depends on the transformation method as follows:
  - For log transformation, c is bias: \( x = \log(y + c) \). The default value is 0.
  - For square root transformation, c is bias: \( x = \sqrt{y + c} \). The default value is 0.
  - For logit transformation, c is scaling: \( x = \log(c \times y / (1 - (c \times y))) \). The default value is 1.
  - For the Box-Cox transformation, c is \( \lambda \): \( x = c^2 + (y^c - 1)/c \). If c is not specified, \( x = y \).

Returned Values

The TRANSFORM function returns the following values:

- **rc** returns one of the following scalar return codes:

<table>
<thead>
<tr>
<th>rc</th>
<th>Termination Reason</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Success</td>
</tr>
<tr>
<td>&lt; 0</td>
<td>Computational failure</td>
</tr>
</tbody>
</table>

- **x** returns the transformed series.
Example

This example uses the TSMODEL procedure to take the log transform of the time series array Air:

```sas
proc tsmodel data=mycas.air outarray=mycas.trans_array;
  id date interval=month;
  var air;
  outarrays t1;
  require tsa;
  submit;
    declare object TSA(tsa);
    rc=TSA.TRANSFORM(air, 'LOG', 0, 0, t1);
  endsubmit;
run;
```

UNBIASEDNESS Method

\[ rc = TSA.UNBIASEDNESS (y, predict, \textless siglevel\textgreater , intercept, scale, fvalue, pvalue) ; \]

The UNBIASEDNESS function tests whether a univariate time series is unbiased.

Required Arguments

You must specify the following arguments, separated by a comma:

- \( y \) specifies the input time series array.
- \( predict \) specifies an input array of predicted time series.

Optional Arguments

You can also specify the following argument. If you want to use a default value for this argument, enter a space for it.

- \( \text{siglevel} \) specifies the significance level.

Returned Values

The UNBIASEDNESS function returns the following values:

- \( rc \) returns one of the following scalar return codes:

<table>
<thead>
<tr>
<th>rc</th>
<th>Termination Reason</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Biased predictions</td>
</tr>
<tr>
<td>0</td>
<td>Unbiased predictions</td>
</tr>
<tr>
<td>-1</td>
<td>Degree of freedom error</td>
</tr>
<tr>
<td>-2</td>
<td>Singular system</td>
</tr>
<tr>
<td>-3</td>
<td>Extreme value</td>
</tr>
</tbody>
</table>
intercept returns the constant parameter.

scale returns the scale parameter.

fvalue returns the test statistic for the $F$ test.

pvalue returns the $p$-value for the $F$ test.

Example

This example uses the TSMODEL procedure to test whether the series Actual is unbiased:

```hpf
proc hpf data=sashelp.air out=_null_ outfor=outfor;
   id date interval=month;
   forecast air;
run;

proc reg data=outfor;
   model actual=predict;
   test intercept=0, predict=1;
run;
quit;
data mycas.outfor;
   set outfor;
run;

proc tsmodel data=mycas.outfor outscalar=mycas.bias_scalar outarray=mycas.bias_array;
   id date interval=month;
   var ACTUAL PREDICT;
   outscalars intercept scale fvalue pvalue;
   require tsa;
submit;
   declare object TSA(tsa);rc=TSA.UNBIASEDNESS(ACTUAL, PREDICT, 0.05, intercept, scale, fvalue, pvalue);
endsubmit;
run;
```

WHITENOISE Method

$$rc = \text{TSA.WHITENOISE} (y, nlag, lags, df, wn, < wnprob >, < wnlprob >);$$

The WHITENOISE function tests for white noise in a time series array.

Required Arguments

You must specify the following arguments, separated by a comma:

$y$ specifies the times series array to compute.

$nlag$ specifies the number of the lag to use in the calculation.
Returned Values

The WHITENOISE function returns the following values:

- **rc** returns one of the following scalar return codes:
  
<table>
<thead>
<tr>
<th>rc</th>
<th>Termination Reason</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Success</td>
</tr>
<tr>
<td>&lt; 0</td>
<td>Computational failure</td>
</tr>
</tbody>
</table>

- **lags** returns the number of the lag that was used in the calculation.
- **df** returns the number of observations that were used to test white noise.
- **wn** returns an array of Ljung-Box white noise tests, with \( nlag + 1 \) entries.

Optional Returned Values

You can also specify the following arguments, separated by commas to request additional returned values. If you do not want the value to be returned, enter a space for it.

- **wnprob** returns white noise probabilities.
- **wnlprob** returns white noise log probabilities.

Example

This example uses the TSMODEL procedure to perform the white noise test of lag 3 of the time series Air:

```plaintext
proc tsmode data=mycas.air outarray=mycas.outarray;
  id date interval=month;
  var air;
  outarrays lags df wn wnprob wnlprob;
  require tsa;
  submit;
  declare object TSA(tsa);
  rc=TSA.WHITENOISE(air, 3, lags, df, wn, wnprob, wnlprob);
  endsubmit;
run;
```

References

Chapter 12
Time Series Dimension Reduction Package

Contents

Overview: TDR Package .................................................. 426
TDR Package Summary .................................................. 426
Using the TDR Package .................................................. 426
TDR Object ................................................................. 427
TDR Synopsis ............................................................. 428
TDR Methods .............................................................. 429
OUTTDR Object ......................................................... 432
OUTTDR Synopsis ...................................................... 433
OUTTDR Method ........................................................ 433
Details: Dimension Reduction Methods ............................. 434
Discrete Fourier Transform ............................................. 434
Discrete Wavelet Transform ........................................... 435
Piecewise Aggregate Approximation ............................... 436
Random Projection ..................................................... 436
Symbolic Aggregate Approximation ................................ 437
Singular Value Decomposition ....................................... 437
Examples: TDR Package ............................................... 438
Using CAS Sessions and CAS Engine Librefs .................... 438
Example 12.1: A Simple Example with Piecewise Aggregate Approximation .......................... 439
Example 12.2: Dimension Reduction by Symbolic Aggregate Approximation ...................... 440
Example 12.3: Dimension Reduction by Discrete Fourier Transformation ........................... 441
Example 12.4: Dimension Reduction by Discrete Wavelet Transformation .......................... 442
Example 12.5: Dimension Reduction by Singular Value Decomposition ............................. 443
Example 12.6: Dimension Reduction by Random Projection ............................................. 447
References ................................................................. 449
Overview: TDR Package

TDR Package Summary

The time series dimension reduction (TDR) package contains functional objects that reduce the dimensionality of time series. The dimension of a time series is the number of time points in the time series. The resulting reduced-dimension time series keeps as much information as possible from the original series, so that you can take advantage of fewer dimensions to perform tasks such as similarity, classification, clustering, and so on.

Table 12.1 summarizes the objects in the TDR package.

<table>
<thead>
<tr>
<th>Object</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Computational Object</td>
<td></td>
</tr>
<tr>
<td>TDR</td>
<td>Various dimension reduction methods</td>
</tr>
<tr>
<td>Collector Object</td>
<td></td>
</tr>
<tr>
<td>OUTTDR</td>
<td>Collect reduced time series output</td>
</tr>
</tbody>
</table>

Using the TDR Package

The following code provides an outline of how to use the TDR package:

```plaintext
proc tsmodel data=InputDataSetName outobj=(of=OutDataSetName);
    var InputVarName;
    id TimeIDVarName;
    require tdr;
    submit;
    declare object f(a computational object);
    declare object of(a collector object);
    rc = f.Initialize();
    rc = f.SetInput(InputVarName);
    rc = f.SetOption("option1", option1_numeric_value,
                      "option2", "option2_char_value",
                      ...);
    rc = f.Run();
    rc = of.Collect(the declared computational Object);
    endsubmit;
run;
```
The basic execution pattern follows this sequence of operations:

1 **Declare**: Create computational and collector objects by using the object declaration statement.

2 **Initialize**: Add a default model specification to the computational object.

3 **Specify variables**: Specify time series variables by using the SetInput method.

4 **Specify option**: Specify model options and properties as appropriate by using the SetOption method.

5 **Run**: Execute the model in the computational object to produce reduced time series.

6 **Collect results**: Extract the results by using a collector object.

---

**TDR Object**

The TDR object executes a dimension reduction function by using one of the following techniques:

- piecewise aggregate approximation
- symbolic aggregate approximation
- discrete Fourier transformation
- discrete wavelet transformation
- random projection
- singular value decomposition

The object declaration statement creates a new object, `obj`, whose type is TDR. Upon declaration, the TDR object has a dimension reduction model.

Table 12.2 summarizes the methods that are associated with the TDR object.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initialize</td>
<td>Initialize the TDR object</td>
</tr>
<tr>
<td>Run</td>
<td>Run the TDR object</td>
</tr>
<tr>
<td>SetInput</td>
<td>Specify an input time series array for the TDR object</td>
</tr>
<tr>
<td>SetOption</td>
<td>Specify options for the TDR object</td>
</tr>
</tbody>
</table>

Figure 12.1 diagrams the methods of the TDR object.
**TDR Synopsis**

DECLARE OBJECT obj (TDR) ;

Method syntax, in order of typical usage:

```c
rc=obj.Initialize () ;
rc=obj.Run () ;
rc=obj.SetInput (InputSeries) ;
rc=obj.SetOption (‘Name’, Value <,’Name’,Value>) ;
```
TDR Methods

TDR.Initialize Method

\[
rc = \text{obj}.\text{Initialize} () ;
\]
Initializes a TDR object that has default parameters. This method must be called before you specify input time series and other attributes for the TDR object.

**Input Arguments**
There are no arguments associated with this method.

TDR.Run Method

\[
rc = \text{obj}.\text{Run} () ;
\]
Runs the TDR object to produce the reduced time series of the specified input time series. Upon successful completion, the distances and their path can be extracted from the TDR object.

**Input Arguments**
There are no arguments associated with this method.

TDR.SetInput Method

\[
rc = \text{obj}.\text{SetInput} (\text{InputSeries}) ;
\]
Specifies the input time series, \text{InputSeries}, for the TDR object.

**Input Arguments**
You must specify the following input argument:

\text{InputSeries} \quad \text{takes numeric array variables that specify the input time series for the TDR object.}

TDR.SetOption Method

\[
rc = \text{obj}.\text{SetOption} (\text{'Name', Value}<\text{'Name', Value}, \ldots>) ;
\]
Specifies options for the TDR object.

**Input Arguments**
You must specify at least one of the following \text{'Names'} and its associated \text{Value}:

\text{'METHOD'} \quad \text{takes a string \text{Value} that specifies a dimension reduction method. You can specify one of the following values:}

\begin{itemize}
  \item DFT \quad \text{uses discrete Fourier transformation.}
  \item DWT \quad \text{uses discrete wavelet transformation.}
  \item PAA \quad \text{uses piecewise aggregate approximation.}
  \item RP \quad \text{uses random projection.}
\end{itemize}
Chapter 12: Time Series Dimension Reduction Package

SAX uses symbolic aggregate approximation.
SVD uses singular value decomposition.

The default is PAA.

'MISSING' takes a string Value that specifies how missing values are interpreted in input time series. You can specify one of the following values:

MAX sets missing values to the maximum value of the series.
MEAN sets missing values to the mean value of the series.
MID sets missing values to the midrange value of the series. When the midrange value is not available, the mean value is used for imputation.
MIN sets missing values to the minimum value of the series.
MISSING sets missing values to missing.
NEXT sets missing values to the next nonmissing value of the missing value in the series. When the next value is not available, the mean value is used for imputation.
PREV sets missing values to the previous nonmissing value of the missing value in the series. When the previous value is not available, the mean value is used for imputation.
ZERO sets missing values to 0. This is useful for transaction data, because having no recorded data usually implies no activity.

When you specify both 'TRIM' and 'MISSING', 'TRIM' is applied first. The default is MEAN.

'NORMALIZE' takes a string Value that specifies whether the series should be normalized. You can specify one of the following values:

ABS normalizes all input sequences before their pairwise distance calculation by using the absolute normalization method.
STD normalizes all input and target sequences before their pairwise distance calculation by using the standard normalization method.
NONE suppresses normalization.

The default is NONE. The SAX method disables this option, and it always normalizes the data by using the STD option.

'PAASTAT' takes a string Value that specifies which statistic to use for piecewise aggregate approximation. You can specify one of the following values:

MAX uses the maximum value in each bin for aggregation.
MEAN uses the mean value in each bin for aggregation.
MIN uses the minimum value in each bin for aggregation.
SUM uses the sum value in each bin for aggregation.

The default is MEAN.

'SPARSITY' takes a numeric Value that specifies the rate of sparsity in a random projection matrix. The value should be a numeric value greater than or equal to 0 and less than 1. The default is 0, which means that the random projection matrix is dense.

'SEED' takes a nonnegative integer Value that specifies the seed to generate random numbers for a random projection matrix. The default is 0, which means that the seed is generated by the computer's internal clock.

'TRIM' takes a string Value that specifies how or whether missing values are trimmed from the time series. You can specify one of the following values:

LEFT trims beginning missing values.
RIGHT trims ending missing values.
BOTH trims both beginning and ending missing values.
NONE suppresses trimming of missing values.

The default is NONE. When you specify both 'TRIM' and 'MISSING', 'TRIM' is applied first. When at least one nonmissing observation in a multivariate time series exists at the same time point, the observations at that time point are not trimmed.

'NBREAKPOINT' takes a positive integer Value that specifies the number of break points for SAX symbols. It is one less than the number of symbols. The number must be between 1 and 255, inclusive. The default value is 10.

'NDIM' takes a positive integer Value that specifies the number of dimensions for the reduced time series. The value is also used as the number of bins for piecewise aggregate approximation. When the number of observations is less than the specified value, the number of observations becomes the 'NDIM' value. The default value is 10.
OUTTDR Object

The OUTTDR object collects the reduced time series from a TDR object and stores them in a CAS table for printing or archiving. The object declaration statement creates a new object, obj, whose type is OUTTDR.

Table 12.3 summarizes the method that is associated with the OUTTDR object.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Collect</td>
<td>Collect reduced time series from a TDR object</td>
</tr>
</tbody>
</table>

Figure 12.2 diagrams the methods of the OUTTDR object.

Figure 12.2 OUTTDR Data Flow

Table 12.4 shows the contents of the OUTTDR object.
**Table 12.4** Contents of the OUTTDR Object

<table>
<thead>
<tr>
<th>Column</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>String</td>
<td>Name of the input time series</td>
</tr>
<tr>
<td>ReducedSeries</td>
<td>Numeric</td>
<td>Reduced time series</td>
</tr>
<tr>
<td>TimeIndex</td>
<td>Numeric</td>
<td>Reduced time index</td>
</tr>
</tbody>
</table>

**OUTTDR Synopsis**

```plaintext
DECLARE OBJECT obj (OUTTDR) ;
```

Method syntax:

```plaintext
rc = obj.Collect (TDRObj) ;
```

**OUTTDR Method**

**OUTTDR.Collect Method**

```plaintext
rc = obj.Collect (TDRObj) ;
```

Retrieves the results of time series distance run actions from a TDR object, `TDRObj`, and stores them in a CAS table.

**Input Arguments**

You must specify the following input argument:

`TDRObj` takes an LCS object to use as the source of time series distance output.
Details: Dimension Reduction Methods

Suppose you want to reduce a time series $X$ that has $n$ observations, $x_1, x_2, \ldots, x_n$, to a resulting series $Y$ that has $d$ values, $y_1, y_2, \ldots, y_d$, where $1 \leq d \leq n$. The TDR package supports the methods that are described in the following subsections.

Discrete Fourier Transform

The discrete Fourier transform (DFT) decomposes data series into a sum of sine and cosine waves of different amplitudes and wavelengths. The Fourier transform decomposition of the series $x_t$ is

$$x_t = a_0 + \sum_{k=1}^{m-1} f_k[a_k \cos(\omega_k(t - 1)) + b_k \sin(\omega_k(t - 1))]$$

where

- $t$ is the time subscript, $t = 1, 2, \ldots, n$.
- $x_t$ are the time series data.
- $n$ is the number of observations in the time series.
- $m$ is the number of frequencies in the Fourier decomposition: $m = \frac{n}{2}$ if $n$ is even, $m = \frac{n-1}{2}$ if $n$ is odd.
- $k$ is the frequency subscript, $k = 0, 1, 2, \ldots, m - 1$.
- $a_0$ is the mean term: $a_0 = \bar{x}$.
- $a_k$ are the cosine coefficients.
- $b_k$ are the sine coefficients.
- $\omega_k$ are the Fourier frequencies: $\omega_k = \frac{2\pi k}{n}$.

If $n$ is even and $k = m - 1$, then $f_k = \frac{1}{2}$; otherwise, $f_k = 1$.

You can approximate the original series by using the first few Fourier coefficients of the series. The first $d$ coefficients compose the reduced series $Y$. For example, when $d = 5$, the reduced series $Y$ is \{a_0, a_1, b_1, a_2, b_2\}. If the STD normalization is used, the first element of the reduced series starts from $a_1$, so when $d = 5$, $Y$ becomes \{a_1, b_1, a_2, b_2, a_3\}. The TDR package uses the fast Fourier transform (FFT) that was developed by Cooley and Tukey (1965) and implemented by Singleton (1969). If $n$ is a power of 2, it uses a Chirp-Z algorithm similar to that proposed by Monro and Branch (1977).
Discrete Wavelet Transform

Wavelet transformation is widely used in many fields, such as signal processing and image analysis. It is very useful for compressing digital files and reducing image or signal noise. It also has a reversible characteristic, enabling the original series to be recovered easily after the transformation. The TDR package uses the Haar wavelet as follows.

The mother wavelet function $\psi(t)$ is

$$
\psi(t) = \begin{cases} 
1 & 0 \leq t < \frac{1}{2} \\
-1 & \frac{1}{2} \leq t < 1 \\
0 & \text{otherwise}
\end{cases}
$$

The scaling function $\phi(t)$ is

$$
\phi(t) = \begin{cases} 
1 & 0 \leq t < 1 \\
0 & \text{otherwise}
\end{cases}
$$

By translations and dilations of the functions $\psi(t)$ and $\phi(t)$, the families of wavelet and scaling functions are constructed as

$$
\psi_{j,k}(x) = 2^{j/2} \psi(2^j x - k)
$$

and

$$
\phi_{j,k}(x) = 2^{j/2} \phi(2^j x - k)
$$

where $j$ and $k$ are integers; the index $j$ defines the dilation or level, and the index $k$ defines the translation.

For a fixed $j_0$, discrete wavelet transformation decomposes a signal or function $f(x) \in L^2$ into a sum of the scale functions and wavelet functions,

$$
f(x) = \sum_k c_{k0}^j \phi_{j0,k}(x) + \sum_{j \geq j_0} \sum_k d_{k}^j \psi_{j,k}(x)
$$

where $c_{k0}^j$ and $d_{k}^j$ are known as the scaling coefficients and detail coefficients, respectively.

The original series can be reconstructed by adding or subtracting the detail coefficients from the lower-resolution representations. For time series dimension reduction, you keep $d$ coefficients to represent the original series. The first element in the reduced series is the scaling coefficient at $j_0$. You add the detail coefficients to the reduced series sequentially from the lowest-resolution, $j_0$ level until you get the $d$ elements in the reduced series.

Chan and Fu (1999) give you more details about time series indexing by using wavelets, and Ogden (1997) provides general information about wavelet analysis.
### Piecewise Aggregate Approximation

Keogh and Pazzani (2000) proposed the line segment method with a mean statistic; they call it piecewise aggregate approximation (PAA). This is the simplest dimension reduction method in time series. If you have a time dimension of size $n$, the method divides the time dimension into $d$ equal-size segments (or time intervals). After the segmentation, you can compute the sum, mean, or other aggregation statistic of each segment. For example, suppose you have a time series that contains 12 time points, $X = \{1, 2, 5, 7, 8, 5, 5, 7, 8, 2, 5, 3\}$, and you want a reduced time series that contains 3 time points. In this case, you can transform the original series into a reduced series, $Y = \{3.75, 6.25, 4.5\}$, that contains the mean statistic of each segment.

### Random Projection

Random projection (RP) is a technique that projects a set of points from a high-dimensional space to a randomly chosen low-dimensional subspace. Random projection approximately preserves pairwise distances with high probability. This idea of random projection has been influenced by the Johnson-Lindenstrauss lemma (Johnson and Lindenstrauss 1984). Let the matrix be $X_{mn}$, where $m$ is the number of time series and $n$ is the length of each time series. You want to reduce the matrix to $Y_{md}$, where $d$ is the resulting dimension number. Let $R_{nd} = (r_{ij})$ be a random matrix such that each entry $r_{ij}$, where $i = 1, 2, \ldots, n$ and $j = 1, 2, \ldots, d$, is chosen independently from the standard normal distribution, $N(0, 1)$. Then you obtain a reduced time series $Y$ of $X$ by using the projection matrix $R$ as follows:

$$Y_{md} = \frac{1}{\sqrt{d}} X_{mn} R_{nd}$$

$R_{nd}$ is a dense matrix, and $\frac{1}{\sqrt{d}}$ is a normalization factor.

You can construct a sparse random projection matrix by specifying the sparsity parameter, $s$. You generate the elements of the $R_{nd}$ with the sparse parameter as follows,

$$r_{ij} = \begin{cases} 1 & \text{with probability } \frac{1}{2}(1 - s) \\ 0 & \text{with probability } s \\ -1 & \text{with probability } \frac{1}{2}(1 - s) \end{cases}$$

where $0 < s < 1$. For example, if $s = 0.5$, half of the $R_{nd}$ elements are 0. For more information about the sparse random projection matrix, see Achlioptas (2003) and Li, Hastie, and Church (2006).
Symbolic Aggregate Approximation

The symbolic aggregate approximation (SAX) method is similar to the PAA method, but instead of assigning a basic statistic to each binned segment, it assigns a symbol to each binned segment. The critical values of symbol intervals are predefined from the standard normal distribution. For more information about SAX time series discretization, see Lin et al. (2003).

Singular Value Decomposition

Unlike other methods that reduce the dimensionality of each individual series directly on the basis of the series itself, singular value decomposition (SVD) treats the input data set together as a matrix in which each column contains a time point and each row contains a time series. Let the matrix be $X_{mn}$, where $m$ is the number of time series and $n$ is the length of each time series. You want to reduce the matrix to $Y_{md}$, where $d$ is the resulting dimension number. In other words, each time series $X(i) = \{x_{i1}, x_{i2}, \ldots, x_{in}\}$ in $X$ will be approximated by a series $Y(i) = \{y_{i1}, y_{i2}, \ldots, y_{id}\}$ in $Y$.

It is known that a matrix $X$ can be broken down into the product of three matrices: an orthogonal matrix $U$, a diagonal matrix $S$, and the transpose of an orthogonal matrix $V$. That is,

$$X_{mn} = U_{mm}S_{mn}V_{nn}^T$$

where $U^T U = I$ and $V^T V = I$.

The columns of $U$ are orthonormal eigenvectors of $XX^T$, the columns of $V$ are orthonormal eigenvectors of $X^TX$, and $S$ is a diagonal matrix that contains the square roots of eigenvalues from $U$ or $V$ in descending order.

You keep the top $d$ eigenvalues in $S$, and you keep the corresponding $d$ columns in $U$. By multiplying $U$ and $S$, you get the resulting matrix $Y$,

$$Y_{md} = U_{md}S_{dd}$$

where $S_{dd}$ contains only the top $d$ eigenvalues in $S$, and $U_{md}$ contains only the corresponding $d$ columns in $U$. 
Examples: TDR Package

In all the examples in this section, the functionality of the TDR package is illustrated using the TSModel procedure. This section assumes that you are familiar with the general workings of the TSModel procedure. For more information, see Chapter 11, “The TSModel Procedure” (SAS Visual Forecasting: Forecasting Procedures).

Using CAS Sessions and CAS Engine Librefs

SAS Cloud Analytic Services (CAS) is the analytic server and associated cloud services in SAS Viya. This section describes how to create a CAS session and set up a CAS engine libref that you can use to connect to the CAS session. It assumes that you have a CAS server already available; contact your system administrator if you need help starting and terminating a server. This CAS server is identified by specifying the host on which it runs and the port on which it listens for communications. To simplify your interactions with this CAS server, the host information and port information for the server are stored as SAS option values that are retrieved automatically whenever this CAS server needs to be accessed. You can examine the host and port values for the server at your site by using the following statements:

```sas
proc options option=(CASHOST CASPORT);
run;
```

In addition to starting a CAS server, your system administrator might also have created a CAS session and a CAS engine libref for your use. You can define your own sessions and CAS engine librefs that connect to the CAS server as shown in the following statements:

```sas
cas mysess;
libname mycas cas sessref=mysess;
```

The CAS statement creates the CAS session named mysess, and the LIBNAME statement creates the mycas CAS engine libref that you use to connect to this session. It is not necessary to explicitly name the CASHOST and CASPORT of the CAS server in the CAS statement, because these values are retrieved from the corresponding SAS option values.

If you have created the mysess session, you can terminate it by using the TERMINATE option in the CAS statement as follows:

```sas
cas mysess terminate;
```

For more information about the CAS statement and the LIBNAME statement, see SAS Cloud Analytic Services: User’s Guide. For general information about CAS and CAS sessions, see SAS Cloud Analytic Services: Fundamentals.
Example 12.1: A Simple Example with Piecewise Aggregate Approximation

This simple example illustrates how to use the TDR object to reduce the dimensionality of time series by using the piecewise aggregate approximation (PAA) method. The following statements generate a simple time series, \( x \), along with a time variable, Time:

```plaintext
data testdata;
  input Time Y @@;
datalines;
1 3 2 5 3 3 4 3 5 3 6 6 7 3 8 8 9 6 10 7
11 2 12 4 13 3 14 6 15 5 16 6 17 8 18 9 19 7 20 9
;run;
```

The following statements perform the dimension reduction by using the PAA method on the example data set. Specifying the value 10 for the 'NDIM' argument means that the length of reduced time series is 10. The value of 'NDIM' corresponds to the number of bins in the PAA. No normalization is imposed. SetInput specifies the input variable. The reduced time series is computed using the mean value in each bin because the 'MEAN' is specified in the 'PAASTAT' option. The OUTTDR output object is declared to retrieve the reduced time series.

```plaintext
proc tsmodel data=mycas.testdata nosummary outlog=mycas.outlog
  outobj=(of= mycas.outtdr1(replace=YES));
  var Y;
  id Time interval=obs;
  require tdr;
  submit;
    declare object f(TDR);
    declare object of(OUTTDR);
    rc = f.Initialize();
    rc = f.SetInput(Y);
    rc = f.SetOption('METHOD', 'PAA',
                     'PAASTAT', 'MEAN',
                     'NORMALIZE', 'NONE',
                     'NDIM', 10);
    rc = f.Run();if rc < 0 then stop;
    rc = of.Collect(f);if rc < 0 then stop;
  endsubmit;
  print outlog;
run;
```

```plaintext
proc print data=mycas.outtdr1 label;
run;
```

Output 12.1.1 shows the reduced time series of \( Y \) that are produced by the PAA method. Because each bin has two observations in the example, the first observation in the reduced time series is the average value of the first two observations in the original series.
## Example 12.2: Dimension Reduction by Symbolic Aggregate Approximation

This example shows how to produce a symbolic aggregate approximation (SAX) of a time series. In the following statements, the 'NBREAKPOINT' option determines how many characters will be used. For example, if the value of 'NBREAKPOINT' is 20, one of 21 unique characters is assigned to each segment. The SAX method in the TDR package outputs positive integer values instead of characters, but you can map them to characters.

```sas
proc tsmodeledata=mycas.testdata nosummary outlog=mycas.outlog
   outobj=(of= mycas.outtdr2(replace=YES));
var Y;
   id Time interval=obs;
   require tdr;
submit;
   declare object f(TDR);
   declare object of(OUTTDR);
   rc = f.Initialize();
   rc = f.SetInput(Y);
   rc = f.SetOption('METHOD', 'SAX',
                     'NBREAKPOINT', 20,
                     'NDIM', 10);
   rc = f.Run();if rc < 0 then stop;
   rc = of.Collect(f);if rc < 0 then stop;
endsubmit;
print outlog;
run;

proc format;
   value letter 1='A' 2='B' 3='C' 4='D' 5='E' 6='F'
      7='G' 8='H' 9='I' 10='J' 11='K' 12='L'
      13='M' 14='N' 15='O' 16='P' 17='Q' 18='R'
      19='S' 20='T' 21='U';
run;
proc print data=mycas.outtdr2 label;
   format timeindex best12. reducedseries letter.;
```
Example 12.3: Dimension Reduction by Discrete Fourier Transformation

This example shows how to get reduced time series by using the discrete Fourier transformation (DFT) method. As the following statements show, you need to specify the value 'DFT' for the 'METHOD' argument. The specified 'NDIM' value corresponds to the number of Fourier coefficients that you want to store.

```
proc tsmodel data=mycas.testdata nosummary outlog=mycas.outlog
   outobj=(of= mycas.outtdr3(replace=YES));
   var Y;
   id Time interval=obs;
   require tdr;
   submit;
      declare object f(TDR);
      declare object of(OUTTDR);
      rc = f.Initialize();
      rc = f.SetInput(Y);
      rc = f.SetOption('METHOD', 'DFT',
                       'NDIM', 10);
      rc = f.Run(); if rc < 0 then stop;
      rc = of.Collect(f); if rc < 0 then stop;
   endsubmit;
   print outlog;
run;

proc print data=mycas.outtdr3 label;
run;
```

Output 12.3.1 shows the reduced time series of $Y$ that are produced by the DFT method.
Output 12.3.1 Reduced Time Series by DFT

<table>
<thead>
<tr>
<th>Obs</th>
<th>Variable</th>
<th>Reduced Time Index</th>
<th>Reduced Series</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Y</td>
<td>1</td>
<td>5.3</td>
</tr>
<tr>
<td>2</td>
<td>Y</td>
<td>2</td>
<td>0.4413667022</td>
</tr>
<tr>
<td>3</td>
<td>Y</td>
<td>3</td>
<td>-1.079026584</td>
</tr>
<tr>
<td>4</td>
<td>Y</td>
<td>4</td>
<td>-0.430901699</td>
</tr>
<tr>
<td>5</td>
<td>Y</td>
<td>5</td>
<td>-2.014370027</td>
</tr>
<tr>
<td>6</td>
<td>Y</td>
<td>6</td>
<td>0.1929400812</td>
</tr>
<tr>
<td>7</td>
<td>Y</td>
<td>7</td>
<td>0.1118355278</td>
</tr>
<tr>
<td>8</td>
<td>Y</td>
<td>8</td>
<td>-0.580901699</td>
</tr>
<tr>
<td>9</td>
<td>Y</td>
<td>9</td>
<td>-0.321644081</td>
</tr>
<tr>
<td>10</td>
<td>Y</td>
<td>10</td>
<td>0.3</td>
</tr>
</tbody>
</table>

Example 12.4: Dimension Reduction by Discrete Wavelet Transformation

This example shows how to get reduced time series by using the discrete wavelet transformation (DWT) method. As the following statements show, you need to specify the value 'DWT' for the 'METHOD' argument. The specified 'NDIM' value corresponds to the number of wavelet coefficients that you want to store.

```plaintext
proc tsmodel data=mycas.testdata nosummary outlog=mycas.outlog
  outobj=(of= mycas.outtdr4(replace=YES));
var Y;
  id Time interval=obs;
  require tdr;
submit;
  declare object f(TDR);
  declare object of(OUTTDR);
  rc = f.Initialize();
  rc = f.SetInput(Y);
  rc = f.SetOption('METHOD', 'DWT',
                   'NDIM', 10);
  rc = f.Run();if rc < 0 then stop;
  rc = of.Collect(f);if rc < 0 then stop;
endsubmit;
print outlog;
run;
proc print data=mycas.outtdr4 label;
run;
```

Output 12.4.1 shows the reduced time series of Y that are produced by the DWT method.
Example 12.5: Dimension Reduction by Singular Value Decomposition

This example shows how to get reduced time series by using the singular value decomposition (SVD) method and how to use the reduced time series for time series clustering. The following statements generate a data set that contains 12 time series. The length of each time series is 100.

```sas
data testdata2;
call streaminit(1);
do time = 1 to 100;
x1 = 2+rand('uniform',-0.5,0.5);
x2 = x1+rand('uniform',-0.5,0.5);
x3 = x2+rand('uniform',-0.5,0.5);
y1 = 4*(1-time*1/100)+rand('Uniform',-0.5,0.5);
y2 = y1+rand('uniform',-0.5,0.5);
y3 = y2+rand('uniform',-0.5,0.5);
z1 = 4*time*1/100+rand('Uniform',-0.5,0.5);
z2 = z1+rand('uniform',-0.5,0.5);
z3 = z2+rand('uniform',-0.5,0.5);
s1 = 4+sin(time/10)+ rand('uniform',-0.5,0.5);
s2 = s1+rand('uniform',-0.5,0.5);
s3 = s2+rand('uniform',-0.5,0.5);
output;
end;
run;

data mycas.testdata2;
  set testdata2;
run;

proc sgplot data=testdata2;
yaxis label = " ";
xaxis label = "Time";
series x=Time y=x1;
series x=Time y=x2;
series x=Time y=x3;
```

Output 12.4.1 Reduced Time Series by DWT

<table>
<thead>
<tr>
<th>Obs</th>
<th>Variable Name</th>
<th>Reduced Time Index</th>
<th>Reduced Series</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Y</td>
<td>1</td>
<td>27.223611076</td>
</tr>
<tr>
<td>2</td>
<td>Y</td>
<td>2</td>
<td>-1.414213562</td>
</tr>
<tr>
<td>3</td>
<td>Y</td>
<td>3</td>
<td>-1.25</td>
</tr>
<tr>
<td>4</td>
<td>Y</td>
<td>4</td>
<td>3.25</td>
</tr>
<tr>
<td>5</td>
<td>Y</td>
<td>5</td>
<td>-2.121320344</td>
</tr>
<tr>
<td>6</td>
<td>Y</td>
<td>6</td>
<td>-0.353553391</td>
</tr>
<tr>
<td>7</td>
<td>Y</td>
<td>7</td>
<td>6.7175144213</td>
</tr>
<tr>
<td>8</td>
<td>Y</td>
<td>8</td>
<td>1</td>
</tr>
<tr>
<td>9</td>
<td>Y</td>
<td>9</td>
<td>-1</td>
</tr>
<tr>
<td>10</td>
<td>Y</td>
<td>10</td>
<td>3.5</td>
</tr>
</tbody>
</table>
Chapter 12: Time Series Dimension Reduction Package

series x=Time y=y1;
series x=Time y=y2;
series x=Time y=y3;
series x=Time y=z1;
series x=Time y=z2;
series x=Time y=z3;
series x=Time y=s1;
series x=Time y=s2;
series x=Time y=s3;
run;

Output 12.5.1 shows the overlaid plot of 12 time series. Each time series has one of four distinct patterns.

**Output 12.5.1** Data Plot of Multiple Series

As the following statements show, you need to specify the value 'SVD' for the 'METHOD' argument. The specified 'NDIM' value corresponds to the number of dimensions of the reduced time series that you want to store.

```
proc tsmodel data=mycas.testdata2 nosummary outlog=mycas.outlog
  outobj=(of= mycas.outtdr5(replace=YES));
  var x1 x2 x3 y1 y2 y3 z1 z2 z3 s1 s2 s3;
  id Time interval=obs;
  require tdr;
  submit;
    declare object f(TDR);
    declare object of(OUTTDR);
    rc = f.Initialize();
    rc = f.SetInput(x1,x2,x3,y1,y2,y3,z1,z2,z3,s1,s2,s3);
    rc = f.SetOption('METHOD', 'SVD',
                     'NDIM', 3);
    rc = f.Run();if rc < 0 then stop;
    rc = of.Collect(f);if rc < 0 then stop;
  endsubmit;
print outlog;
```
run;

proc print data=mycas.outtdr5(obs=10) label;
run;

Output 12.5.2 shows the reduced time series of \( x_1 \) and \( x_2 \) that are produced by the SVD method. The reduced versions of the remaining time series are not shown.

Output 12.5.2 Partial Output of Reduced Time Series by SVD

<table>
<thead>
<tr>
<th>Obs</th>
<th>Variable</th>
<th>Reduced Time Index</th>
<th>Reduced Series</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>x1</td>
<td>1</td>
<td>19.67576133</td>
</tr>
<tr>
<td>2</td>
<td>x1</td>
<td>2</td>
<td>0.481309514</td>
</tr>
<tr>
<td>3</td>
<td>x1</td>
<td>3</td>
<td>2.1470944887</td>
</tr>
<tr>
<td>4</td>
<td>x2</td>
<td>1</td>
<td>19.679006857</td>
</tr>
<tr>
<td>5</td>
<td>x2</td>
<td>2</td>
<td>0.2806842918</td>
</tr>
<tr>
<td>6</td>
<td>x2</td>
<td>3</td>
<td>2.8265031138</td>
</tr>
<tr>
<td>7</td>
<td>x3</td>
<td>1</td>
<td>19.800084441</td>
</tr>
<tr>
<td>8</td>
<td>x3</td>
<td>2</td>
<td>-0.437846019</td>
</tr>
<tr>
<td>9</td>
<td>x3</td>
<td>3</td>
<td>3.3628540206</td>
</tr>
<tr>
<td>10</td>
<td>y1</td>
<td>1</td>
<td>19.962853427</td>
</tr>
</tbody>
</table>

The following statements transpose the output table to create an input data set for time series clustering analysis:

```
proc sort data=mycas.outtdr5 out=sort_outtdr5;
   by name timeindex;
run;
proc transpose data=sort_outtdr5 out=mycas.touttdr5(drop=_NAME_ _LABEL_) prefix=Dimension;
   var ReducedSeries;
   by name;
run;
proc print data=mycas.touttdr5 label;
run;
```

Output 12.5.3 shows that the transposed reduced time series is the input data for \( k \)-means clustering.
Output 12.5.3  Transposed Reduced Time Series for Clustering

<table>
<thead>
<tr>
<th>Obs</th>
<th>Variable</th>
<th>Dimension1</th>
<th>Dimension2</th>
<th>Dimension3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>s1</td>
<td>42.393779821</td>
<td>-0.289028334</td>
<td>-2.583916668</td>
</tr>
<tr>
<td>2</td>
<td>x1</td>
<td>19.67576133</td>
<td>0.481309514</td>
<td>2.1470944887</td>
</tr>
<tr>
<td>3</td>
<td>y1</td>
<td>19.962853427</td>
<td>-10.94149987</td>
<td>1.9110878231</td>
</tr>
<tr>
<td>4</td>
<td>z1</td>
<td>19.647866178</td>
<td>11.839058771</td>
<td>1.5949007116</td>
</tr>
<tr>
<td>5</td>
<td>s2</td>
<td>42.41076634</td>
<td>-0.267395514</td>
<td>-3.42091502</td>
</tr>
<tr>
<td>6</td>
<td>x2</td>
<td>19.679006857</td>
<td>0.2806842918</td>
<td>2.8265031138</td>
</tr>
<tr>
<td>7</td>
<td>y2</td>
<td>20.290775548</td>
<td>-11.08568177</td>
<td>2.3456665931</td>
</tr>
<tr>
<td>8</td>
<td>z2</td>
<td>19.39717233</td>
<td>11.620176708</td>
<td>2.2268974967</td>
</tr>
<tr>
<td>9</td>
<td>s3</td>
<td>42.518740257</td>
<td>0.0043037047</td>
<td>-4.163839854</td>
</tr>
<tr>
<td>10</td>
<td>x3</td>
<td>19.80084441</td>
<td>-0.437846019</td>
<td>3.3628540206</td>
</tr>
<tr>
<td>11</td>
<td>y3</td>
<td>20.242092572</td>
<td>-11.25631305</td>
<td>2.7521518732</td>
</tr>
<tr>
<td>12</td>
<td>z3</td>
<td>19.464935358</td>
<td>11.830875816</td>
<td>2.6317151907</td>
</tr>
</tbody>
</table>

The following PROC KCLUS statements show how to cluster time series by using the table in Output 12.5.3:

```proc kclus data=mycas.touttdr5 maxclusters=4 seed =1234
   outstat(outiter)=svdclusoutstat;
   input Dimension1-Dimension3;
   score out=mycas.svdoutclus copyvars=(Name);
run;
```

```proc print data=mycas.svdoutclus label;
run;
```

Output 12.5.4 shows the results of clustering time series by using the reduced time series that are produced by SVD. All time series are classified correctly.

Output 12.5.4  Clustering the Reduced Time Series by SVD

<table>
<thead>
<tr>
<th>Obs</th>
<th>Variable</th>
<th><em>CLUSTER_ID</em></th>
<th><em>DISTANCE</em></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>s1</td>
<td>2</td>
<td>0.813829188</td>
</tr>
<tr>
<td>2</td>
<td>x1</td>
<td>3</td>
<td>5.9945220562</td>
</tr>
<tr>
<td>3</td>
<td>y1</td>
<td>3</td>
<td>5.4865358214</td>
</tr>
<tr>
<td>4</td>
<td>z1</td>
<td>1</td>
<td>3.476742E-13</td>
</tr>
<tr>
<td>5</td>
<td>s2</td>
<td>2</td>
<td>0.0940813699</td>
</tr>
<tr>
<td>6</td>
<td>x2</td>
<td>3</td>
<td>5.7861380542</td>
</tr>
<tr>
<td>7</td>
<td>y2</td>
<td>3</td>
<td>5.607342276</td>
</tr>
<tr>
<td>8</td>
<td>z2</td>
<td>4</td>
<td>0.2306851985</td>
</tr>
<tr>
<td>9</td>
<td>s3</td>
<td>2</td>
<td>0.8006345858</td>
</tr>
<tr>
<td>10</td>
<td>x3</td>
<td>3</td>
<td>5.120763587</td>
</tr>
<tr>
<td>11</td>
<td>y3</td>
<td>3</td>
<td>5.7741866146</td>
</tr>
<tr>
<td>12</td>
<td>z3</td>
<td>4</td>
<td>0.2306851985</td>
</tr>
</tbody>
</table>
Example 12.6: Dimension Reduction by Random Projection

This example shows how to get reduced time series by using the random projection (RP) method. As the following statements show, you need to specify the value 'RP' for the 'METHOD' argument. The specified 'NDIM' value corresponds to the number of random projection vectors. Each random projection vector produces a value for each time series.

```
proc tsmodel data=mycas.testdata2 nosummary outlog=mycas.outlog
   outobj=(of=mycas.outtdr6(replace=YES));
var x1 x2 x3 y1 y2 y3 z1 z2 z3 s1 s2 s3;
id Time interval=obs;
require tdr;
submit;
   declare object f(TDR);
   declare object of(OUTTDR);
   rc = f.Initialize();
   rc = f.SetInput(x1,x2,x3,y1,y2,y3,z1,z2,z3,s1,s2,s3);
   rc = f.SetOption('METHOD', 'RP',
                    'SPARSITY', 0,
                    'SEED', 12345,
                    'NDIM', 6);
   rc = f.Run();if rc < 0 then stop;
   rc = of.Collect(f);if rc < 0 then stop;
endsubmit;
print outlog;
run;
```

The following statements create an input data set for time series clustering analysis:

```
proc tsmodel data=mycas.testdata2 nosummary outlog=mycas.outlog
   outobj=(of=mycas.outtdr6(replace=YES));
var x1 x2 x3 y1 y2 y3 z1 z2 z3 s1 s2 s3;
id Time interval=obs;
require tdr;
submit;
   declare object f(TDR);
   declare object of(OUTTDR);
   rc = f.Initialize();
   rc = f.SetInput(x1,x2,x3,y1,y2,y3,z1,z2,z3,s1,s2,s3);
   rc = f.SetOption('METHOD', 'RP',
                    'SPARSITY', 0,
                    'SEED', 12345,
                    'NDIM', 6);
   rc = f.Run();if rc < 0 then stop;
   rc = of.Collect(f);if rc < 0 then stop;
endsubmit;
print outlog;
run;
```

Output 12.6.1 shows the reduced time series of x1 and x2 that are produced by the RP method. The reduced versions of the remaining time series are not shown.

**Output 12.6.1 Partial Output of Reduced Time Series by RP**

<table>
<thead>
<tr>
<th>Obs</th>
<th>Variable</th>
<th>Reduced Time Index</th>
<th>Reduced Series</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>x1</td>
<td>-4.114960863</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>x1</td>
<td>-8.641157866</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>x1</td>
<td>-3.396060033</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>x1</td>
<td>-6.767679813</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>x1</td>
<td>1.5157320591</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>x1</td>
<td>1.649125937</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>x2</td>
<td>-4.518731726</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>x2</td>
<td>-8.158655001</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>x2</td>
<td>-4.778592534</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>x2</td>
<td>-7.200128399</td>
<td></td>
</tr>
</tbody>
</table>

The following statements create an input data set for time series clustering analysis:
Chapter 12: Time Series Dimension Reduction Package

Output 12.6.2 shows the transposed reduced time series that are used for \(k\)-means clustering.

### Output 12.6.2 Transposed Reduced Time Series for Clustering

<table>
<thead>
<tr>
<th>Variable</th>
<th>Dimension1</th>
<th>Dimension2</th>
<th>Dimension3</th>
<th>Dimension4</th>
<th>Dimension5</th>
<th>Dimension6</th>
</tr>
</thead>
<tbody>
<tr>
<td>Obs 1</td>
<td>s1</td>
<td>-3.248612704</td>
<td>-14.34537101</td>
<td>-9.312168716</td>
<td>-10.96769766</td>
<td>4.1221754726</td>
</tr>
<tr>
<td>Obs 2</td>
<td>x1</td>
<td>-4.114960863</td>
<td>-8.641157866</td>
<td>-3.396060033</td>
<td>-6.767679813</td>
<td>1.5157320591</td>
</tr>
<tr>
<td>Obs 3</td>
<td>y1</td>
<td>-3.946354884</td>
<td>-4.651082639</td>
<td>-4.277991258</td>
<td>-5.65189505</td>
<td>7.528941301</td>
</tr>
<tr>
<td>Obs 4</td>
<td>z1</td>
<td>-3.858039822</td>
<td>-10.92709379</td>
<td>-1.678815558</td>
<td>-8.477544371</td>
<td>-6.755109634</td>
</tr>
<tr>
<td>Obs 5</td>
<td>s2</td>
<td>-2.552267034</td>
<td>-14.89083566</td>
<td>-8.508746897</td>
<td>-11.27120051</td>
<td>4.9257451354</td>
</tr>
<tr>
<td>Obs 6</td>
<td>x2</td>
<td>-4.518731726</td>
<td>-8.158655001</td>
<td>-4.778592534</td>
<td>-7.200128399</td>
<td>1.1187095209</td>
</tr>
<tr>
<td>Obs 7</td>
<td>y2</td>
<td>-4.51321947</td>
<td>-8.26128786</td>
<td>-4.523406661</td>
<td>-7.160853813</td>
<td>6.7963924602</td>
</tr>
<tr>
<td>Obs 8</td>
<td>z2</td>
<td>-4.732713032</td>
<td>-11.0060808</td>
<td>-3.374281961</td>
<td>-9.260933872</td>
<td>-5.826185963</td>
</tr>
<tr>
<td>Obs 9</td>
<td>s3</td>
<td>-2.20967166</td>
<td>-14.0960508</td>
<td>-8.036488751</td>
<td>-10.29408914</td>
<td>4.5728282535</td>
</tr>
<tr>
<td>Obs 10</td>
<td>x3</td>
<td>-5.820495758</td>
<td>-6.697714334</td>
<td>-5.440828963</td>
<td>-6.753109875</td>
<td>0.939532874</td>
</tr>
<tr>
<td>Obs 11</td>
<td>y3</td>
<td>-3.844506043</td>
<td>-6.493321268</td>
<td>-4.876326058</td>
<td>-7.920812278</td>
<td>7.2894082288</td>
</tr>
<tr>
<td>Obs 12</td>
<td>z3</td>
<td>-5.156132429</td>
<td>-11.73943054</td>
<td>0.8096536403</td>
<td>-10.78268926</td>
<td>-5.093425687</td>
</tr>
</tbody>
</table>

The following PROC KCLUS statements show how to cluster time series by using the table in Output 12.6.2:

```sas
proc kclus data=mycas.touttdr6 maxclusters=4 seed =12345
   outstat(outiter)=kclusOutstat1;
   input Dimension1-Dimension6;
   score out=mycas.rpoutclus copyvars=(Name);
run;
```

Output 12.6.3 shows the results of clustering time series by using the reduced time series produced by the RP method. All time series are classified correctly. Compared with the three reduced time series by the SVD method, the RP method uses the six reduced time series for clustering. However, it is known that the RP method is much faster than the SVD method in computation, especially for a high-dimensional data set.
## Output 12.6.3  Clustering the Reduced Time Series by RP

<table>
<thead>
<tr>
<th>Obs</th>
<th>Variable Name</th>
<th>CLUSTER_ID</th>
<th>DISTANCE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>s1</td>
<td>4</td>
<td>1.4822958094</td>
</tr>
<tr>
<td>2</td>
<td>x1</td>
<td>3</td>
<td>1.6081790118</td>
</tr>
<tr>
<td>3</td>
<td>y1</td>
<td>1</td>
<td>1.5005725167</td>
</tr>
<tr>
<td>4</td>
<td>z1</td>
<td>2</td>
<td>1.6416083677</td>
</tr>
<tr>
<td>5</td>
<td>s2</td>
<td>4</td>
<td>0.7612176843</td>
</tr>
<tr>
<td>6</td>
<td>x2</td>
<td>3</td>
<td>0.6513061665</td>
</tr>
<tr>
<td>7</td>
<td>y2</td>
<td>1</td>
<td>0.8212877255</td>
</tr>
<tr>
<td>8</td>
<td>z2</td>
<td>2</td>
<td>1.9939784946</td>
</tr>
<tr>
<td>9</td>
<td>s3</td>
<td>4</td>
<td>1.5847776027</td>
</tr>
<tr>
<td>10</td>
<td>x3</td>
<td>3</td>
<td>1.7973612771</td>
</tr>
<tr>
<td>11</td>
<td>y3</td>
<td>1</td>
<td>1.602918254</td>
</tr>
<tr>
<td>12</td>
<td>z3</td>
<td>2</td>
<td>2.829217192</td>
</tr>
</tbody>
</table>

### References


Chapter 13
Time Series Distance Measure Package

Contents

Overview: TSD Package .......................................................... 452
TSD Package Summary ......................................................... 452
Using the TSD Package .......................................................... 453
Common Options for the TSD Computational Objects ................. 454

DST Object ................................................................. 456
DST Synopsis ................................................................. 457
DST Methods ................................................................. 457

DTW Object ................................................................. 459
DTW Synopsis ................................................................. 459
DTW Methods ................................................................. 459

LCS Object ................................................................. 461
LCS Synopsis ................................................................. 462
LCS Methods ................................................................. 462

SAX Object ................................................................. 464
SAX Synopsis ................................................................. 464
SAX Methods ................................................................. 465

OUTTSD Object ........................................................... 466
OUTTSD Synopsis ........................................................... 467
OUTTSD Method ........................................................... 467

OUTTSDDTW Object ....................................................... 467
OUTTSDDTW Synopsis ....................................................... 468
OUTTSDDTW Method ....................................................... 468

OUTTSDLCS Object ........................................................ 468
OUTTSDLCS Synopsis ........................................................ 469
OUTTSDLCS Method ........................................................ 469

OUTTSDSAX Object ........................................................ 469
OUTTSDSAX Synopsis ........................................................ 470
OUTTSDSAX Method ........................................................ 470

Examples: TSD Package ...................................................... 471
Using CAS Sessions and CAS Engine Librefs .......................... 471
Example 13.1: A Simple Example with DTW ......................... 472
Example 13.2: Using Multiple Input and Target Time Series ..... 474
Example 13.3: Creating a Distance Matrix for Time Series Clustering ........................................ 476
Example 13.4: Multivariate Time Series Distance Measure .... 479
Example 13.5: Finding the Longest Common Subsequence and Distance ........................................ 481
Overview: TSD Package

The time series distance measure (TSD) package contains functional objects to measure the distance between two time series or among sequences in temporal data. If you have two ordered numeric sequences (input and target), a distance measure calculates the distance between the input and target sequences while taking into account the ordering of the data. Before you can compare the raw input and target timestamped data, the raw data must be accumulated in a time series format. After the input and target time series are formed, you can compare the two accumulated time series as two ordered numeric sequences.

Distance measures can be used to compare a single target sequence to many other input sequences. This type of comparison arises in time series search and retrieval or ranking. For example, starting with a single target sequence, you can find input sequences that are identical to or similar to the target. Search and retrieval and analogies are important for new product forecasting and analogous time series forecasting. These techniques are useful when a large number of historical time series are available.

Distance measures can also be used to compare a single input sequence to several other representative target sequences. This type of comparison arises in time series classification. For example, given a single input sequence, you can classify the input sequence by finding the most similar target sequence. This analysis can be repeated to classify large numbers of input sequences.

Distance measures can also be computed between several sequences to form a distance matrix. Clustering techniques can then be applied to the distance matrix. This situation arises in time series clustering.

TSD Package Summary

Table 13.1 summarizes the objects in the TSD package.

<table>
<thead>
<tr>
<th>Object</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DST</td>
<td>Various distance measure method</td>
</tr>
<tr>
<td>DTW</td>
<td>Dynamic time warping method</td>
</tr>
<tr>
<td>LCS</td>
<td>Longest common subsequence method</td>
</tr>
</tbody>
</table>
Table 13.1  continued

<table>
<thead>
<tr>
<th>Object</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SAX</td>
<td>Symbolic aggregate approximation method</td>
</tr>
</tbody>
</table>

**Collector Objects**

<table>
<thead>
<tr>
<th>Object</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OUTTSD</td>
<td>Collect distance measure output</td>
</tr>
<tr>
<td>OUTTSDDTW</td>
<td>Collect dynamic time warping output</td>
</tr>
<tr>
<td>OUTTSDLCS</td>
<td>Collect longest common subsequence output</td>
</tr>
<tr>
<td>OUTTSDSAX</td>
<td>Collect symbolic aggregate approximation output</td>
</tr>
</tbody>
</table>

---

**Using the TSD Package**

The following code provides an outline of how to use the TSD package:

```r
proc tsmodel data=InputDataSetName outobj=(of=OutDataSetName);
   var InputVarName, TargetVarName;
   id   TimeIDVarName;
   require tsd;
   submit;
      declare object f(a computational object);
      declare object of(a collector object);
      rc = f.Initialize();
      rc = f.SetTarget(TargetVarName);
      rc = f.SetInput(InputVarName);
      rc = f.SetOption("option1", option1_numeric_value,
                       "option2", "option2_char_value",
                       ...
                      );
      rc = f.Run();
      rc = of.Collect(the declared computational Object);
   endsubmit;
run;
```

The basic execution pattern follows this sequence of operations:

1. **Declare**: Create computational and collector objects by using the object declaration statement.
2. **Initialize**: Add a default model specification to the computational object.
3. **Specify variables**: Specify time series variables by using the SetTarget and SetInput methods.
4 Specify option: Specify model options and properties as appropriate by using the SetOption method.

5 Run: Execute the model in the computational object to produce distance measures and their instances.

6 Collect results: Extract the results by using a collector object.

---

Common Options for the TSD Computational Objects

The following common arguments in the SetOption method apply to all the TSD computational objects, which are the DST, DTW, LCS, and SAX objects:

\[
rc = obj.SetOption('Name', Value <, 'Name', Value, ... >) ;
\]

**Input Arguments**
You can specify one or more of the following 'Names' and the associated Value:

**'MISSING'**

takes a string Value that specifies how missing values are interpreted in input and target time series. You can specify one of the following values:

- MAX: sets missing values to the maximum value of the series.
- MEAN: sets missing values to the mean value of the series.
- MID: sets missing values to the midrange value of the series. When the midrange value is not available, the mean value is used for imputation.
- MIN: sets missing values to the minimum value of the series.
- MISSING: sets missing values to missing.
- NEXT: sets missing values to the nonmissing next value of the missing value in the series. When the next value is not available, the mean value is used for imputation.
- PREV: sets missing values to the nonmissing previous value of the missing value in the series. When the previous value is not available, the mean value is used for imputation.
- ZERO: sets missing values to 0. This is useful for transaction data, because having no recorded data usually implies no activity.

When you specify both 'TRIM' and 'MISSING', 'TRIM' is applied first. The default value is MISSING.

**'MTSMETHOD'**

takes a string Value that specifies how multivariate time series are handled in the distance calculation. You can specify one of the following values:

- DEFAULT: uses no dimension reduction techniques.
- PCA: uses the first principal components of the multivariate input and target time series as an input series and a target series, respectively, for the distance calculation.
Common Options for the TSD Computational Objects

'NORMALIZE'

takes a string \textit{Value} that specifies whether the series should be normalized. You can specify one of the following values:

\begin{itemize}
\item \textbf{ABS} normalizes all input and target sequences before their pairwise distance calculation by using the absolute normalization method.
\item \textbf{STD} normalizes all input and target sequences before their pairwise distance calculation by using the standard normalization method.
\item \textbf{NONE} suppresses normalization.
\end{itemize}

The default is NONE.

'SCALE'

takes a string \textit{Value} that specifies how or whether to apply the scaling method for the input sequence to the target sequence. You can specify one of the following values:

\begin{itemize}
\item \textbf{ABS} scales input time series by using the absolute normalization of target series.
\item \textbf{STD} scales input time series by using the standard normalization of target series.
\item \textbf{NONE} suppresses scaling.
\end{itemize}

The default is NONE.

'TRIM'

takes a string \textit{Value} that specifies how or whether missing values are trimmed from the time series. You can specify one of the following values:

\begin{itemize}
\item \textbf{LEFT} trims beginning missing values.
\item \textbf{RIGHT} trims ending missing values.
\item \textbf{BOTH} trims both beginning and ending missing values.
\item \textbf{NONE} suppresses trimming of missing values.
\end{itemize}

The default is NONE. When you specify both 'TRIM' and 'MISSING', 'TRIM' is applied first.
DST Object

The DST object executes a metric or distance function, which is a function that defines a distance between two time series. The metrics that the DST object provides are $L_0$, $L_1$, $L_2$, $L_\infty$, and cosine.

Consider time series $X = \{x_1, x_2, \ldots, x_T\}$ and $Y = \{y_1, y_2, \ldots, y_T\}$, which are two vectors. Then you define $L_p$ distances as

$$D_p(X, Y) = \sum_{t=1}^{T} (|x_t - y_t|^p)^{\frac{1}{p}}$$

$L_0$ distance is known as the Hamming distance,

$$D_0(X, Y) = \sum_{t=1}^{T} I(x_t = y_t)$$

where $I(x_t = y_t) = 1$ if $x_t = y_t$ and $I(x_t = y_t) = 0$ if $x_t \neq y_t$.

$L_1$ distance is known as the Manhattan distance:

$$D_1(X, Y) = \sum_{t=1}^{T} |x_t - y_t|$$

$L_2$ distance is the Euclidean distance between two vectors:

$$D_2(X, Y) = \sum_{t=1}^{T} (|x_t - y_t|^2)^{\frac{1}{2}}$$

$L_\infty$ distance is the maximum deviation along any one coordinate, which is defined as

$$D_\infty(X, Y) = \max_{t=1}^{T} |x_t - y_t|$$

Cosine distance measures the cosine of the angle between two vectors, which is defined as

$$D_{\text{cosine}}(X, Y) = 1 - \frac{\sum_{t=1}^{T} x_t y_t}{||X|| ||Y||}$$

where $||X|| = \sqrt{\sum_{t=1}^{T} x_t^2}$ and $||Y|| = \sqrt{\sum_{t=1}^{T} y_t^2}$.

The object declaration statement creates a new object, $obj$, whose type is DST. Upon declaration, the DST object has a longest common subsequence distance model.

Table 13.2 summarizes the methods that are associated with the DST object.
<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initialize</td>
<td>Initialize the DST object</td>
</tr>
<tr>
<td>Run</td>
<td>Run the DST object</td>
</tr>
<tr>
<td>SetInput</td>
<td>Specify an input time series array for the DST object</td>
</tr>
<tr>
<td>SetOption</td>
<td>Specify options for the DST object</td>
</tr>
<tr>
<td>SetTarget</td>
<td>Specify a target time series array for the DST object</td>
</tr>
</tbody>
</table>

**DST Synopsis**

```plaintext
DECLARE OBJECT obj (DST) ;
```

Method syntax, in order of typical usage:

```plaintext
rc = obj.Initialize () ;
rc = obj.Run () ;
rc = obj.SetInput (InputSeries) ;
rc = obj.SetOption ('Name', Value < , 'Name', Value >) ;
rc = obj.SetTarget (TargetSeries) ;
```

**DST Methods**

**DST.Initialize Method**

```plaintext
rc = obj.Initialize () ;
```

Initializes a DST object with default parameters. This method must be called before you specify input and target time series and other attributes for the DST object.

**Input Arguments**

There are no arguments associated with this method.

**DST.Run Method**

```plaintext
rc = obj.Run () ;
```

Runs the DST object to calculate the longest common subsequence distance between specified input and target time series. Upon successful completion, the distances and their path can be extracted from the DST object.

**Input Arguments**

There are no arguments associated with this method.
DST.SetOption Method

\[ rc = \text{obj.SetOption ('Name', Value <,'Name',Value,... }) ; \]

Specifies options for the DST object.

**Input Arguments**
You must specify at least one of the following 'Names' and its associated Value:

- **'METRIC'**
  - takes a string Value that specifies a metric as the distance measure. You can specify one of the following values:
    - **L0** uses the \( L_0 \) norm between two sequences as the distance measure.
    - **L1** uses the \( L_1 \) norm between two sequences as the distance measure.
    - **L2** uses the \( L_2 \) norm between two sequences as the distance measure.
    - **LINF** uses the \( L_\infty \) norm between two sequences as the distance measure.
    - **COSINE** uses the cosine between two sequences as the distance measure.

DST.SetInput Method

\[ rc = \text{obj.SetInput (InputSeries) ;} \]

Specifies the input time series, InputSeries, for the DST object.

**Input Arguments**
You must specify the following input argument:

- **InputSeries** takes numeric array variables that specify the input time series for the DST object.

DST.SetTarget Method

\[ rc = \text{obj.SetTarget (TargetSeries) ;} \]

Specifies the target time series, TargetSeries, for the DST object.

**Input Arguments**
You must specify the following input argument:

- **TargetSeries** takes numeric array variables that specify the target time series for the DST object.
DTW Object

The DTW object executes a dynamic time warping (DTW) distance calculation method for measuring time series distance. The object declaration statement creates a new object, *obj*, whose type is DTW. Upon declaration, the DTW object has a dynamic time warping distance model.

Table 13.3 summarizes the methods that are associated with the DTW object.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initialize</td>
<td>Initialize the DTW object</td>
</tr>
<tr>
<td>Run</td>
<td>Run the DTW object</td>
</tr>
<tr>
<td>SetInput</td>
<td>Specify an input time series array for the DTW object</td>
</tr>
<tr>
<td>SetOption</td>
<td>Specify options for the DTW object</td>
</tr>
<tr>
<td>SetTarget</td>
<td>Specify a target time series array for the DTW object</td>
</tr>
</tbody>
</table>

DTW Synopsis

```
DECLARE OBJECT obj (DTW) ;
```

Method syntax, in order of typical usage:

- `rc = obj.Initialize () ;`
- `rc = obj.Run () ;`
- `rc = obj.SetInput (InputSeries) ;`
- `rc = obj.SetOption ('Name', Value <, 'Name', Value>) ;`
- `rc = obj.SetTarget (TargetSeries) ;`

DTW Methods

**DTW.Initialize Method**

```
rc = obj.Initialize () ;
```

Initializes a DTW object with default parameters. This method must be called before you specify input and target time series and other attributes for the DTW object.

**Input Arguments**

There are no arguments associated with this method.


**DTW.Run Method**

```ruby
rc = obj.Run();
```

Runs the DTW object to calculate the dynamic time warping distance between specified input and target time series. Upon successful completion, the distances and their warping path can be extracted from the DTW object.

**Input Arguments**

There are no arguments associated with this method.

**DTW.SetOption Method**

```ruby
rc = obj.SetOption('Name', Value <,'Name', Value,..>);
```

Specifies options for the DTW object.

**Input Arguments**

You can specify one or more of the following 'Names' and the associated Value:

- **'INPUTWINPCT' | 'XWINPCT'**
  - takes a nonnegative numeric Value that specifies the warping window percentage for input time series. The default is 50.

- **'INPUTWINSIZE' | 'XWINSIZE'**
  - takes a positive integer Value that specifies the warping window size for input time series, which is a global constraint for finding the warping path. The default is 1,000. When the input series is longer than the target series, the minimum window size is the absolute value of the difference between the input and target series plus one.

- **'METRIC'**
  - takes a string Value that specifies the distance calculation. You can specify one of the following values:
    - ABSDEV uses absolute deviation.
    - MABSDEV uses mean absolute deviation.
    - MSQRDEV uses mean square deviation.
    - RSQRDEV uses root square deviation.
    - SQRDEV uses square deviation.
  - The default is RSQRDEV.

- **'OUTPATH'**
  - takes a string Value that specifies whether to output the warping path data. You can specify one of the following values:
    - YES | Y outputs the warping path. Because doing this requires the full cumulative distance matrix, the warping path is output only when the number of input time series and the number of target time series is the same.
    - NO | N does not output the warping path.
  - The default is NO.
'TARGETWINPCT' | 'YWINPCT' takes a nonnegative numeric Value that specifies the warping window percentage for target time series. The default is 50.

'TARGETWINSIZE' | 'YWINSIZE' takes a positive integer Value that specifies the warping window size for target time series, which is a global constraint for finding the warping path. The default is 1,000. When the target series is longer than the input series, the minimum window size is the absolute value of the difference between the input and target series plus one.

**DTW.SetInput Method**

```c
rc=obj.SetInput (InputSeries) ;
```

Specifies the input time series, \textit{InputSeries}, for the DTW object.

**Input Arguments**
You must specify the following input argument:

\textit{InputSeries} takes numeric array variables that specify the input time series for the DTW object.

**DTW.SetTarget Method**

```c
rc=obj.SetTarget (TargetSeries) ;
```

Specifies the target time series, \textit{TargetSeries}, for the DTW object.

**Input Arguments**
You must specify the following input argument:

\textit{TargetSeries} takes numeric array variables that specify the target time series for the DTW object.

**LCS Object**

The LCS object executes a longest common subsequence (LCS) finding method for measuring time series distance. After you find the LCS, you compute the distance measure on the basis of the lengths of the LCS, input series, and target series. Suppose you have two sequences, ACTAG and GATCA, and the longest common subsequence is ATA. The similarity measure between two sequences is 3, which is the length of the longest common subsequence. The distance measure is $1 - 2^{\frac{LCS length}{Input Series Length + Target Series Length}}$. Because the LCS object supports only numeric values, you need to use formats for character sequences. The object declaration statement creates a new object, \textit{obj}, whose type is LCS. Upon declaration, the LCS object has a longest common subsequence distance model.

Table 13.4 summarizes the methods that are associated with the LCS object.
## LCS Synopsis

`DECLARE OBJECT obj (LCS) ;`

Method syntax, in order of typical usage:

```c
rc=obj.Initialize () ;
rc=obj.Run () ;
rc=obj.SetInput (InputSeries) ;
rc=obj.SetOption ('Name',Value < ',',Name',Value>') ;
rc=obj.SetTarget (TargetSeries) ;
```

### LCS Methods

#### LCS.Initialize Method

```c
rc=obj.Initialize () ;
```

Initializes an LCS object with default parameters. This method must be called before you specify input and target time series and other attributes for the LCS object.

**Input Arguments**

There are no arguments associated with this method.

#### LCS.Run Method

```c
rc=obj.Run () ;
```

Runs the LCS object to calculate the longest common subsequence distance between specified input and target time series. Upon successful completion, the distances and their path can be extracted from the LCS object.

**Input Arguments**

There are no arguments associated with this method.
**LCS.SetOption Method**

\[ rc = obj.SetOption ('Name', Value <,'Name',Value,...>); \]

Specifies options for the LCS object.

**Input Arguments**
You can specify one or more of the following 'Names' and the associated Value:

- **'FUZZ'** takes a positive numeric Value that specifies the fuzz value for matching two points. If two points are not exactly the same, but the distance between them is less than the fuzz value, they are considered to be matched points. The default is 0.1.

- **'OUTPATH'** takes a string Value that specifies whether to output the longest common subsequence path data. You can specify one of the following values:
  - YES | Y outputs the path. Because doing this requires the full cumulative distance matrix, the path is output only when the number of input time series and the number of target time series is the same.
  - NO | N does not output the path.

The default is NO.

**LCS.SetInput Method**

\[ rc = obj.SetInput (InputSeries); \]

Specifies the input time series, InputSeries, for the LCS object.

**Input Arguments**
You must specify the following input argument:

- **InputSeries** takes numeric array variables that specify the input time series for the LCS object.

**LCS.SetTarget Method**

\[ rc = obj.SetTarget (TargetSeries); \]

Specifies the target time series, TargetSeries, for the LCS object.

**Input Arguments**
You must specify the following input argument:

- **TargetSeries** takes numeric array variables that specify the target time series for the LCS object.
**SAX Object**

The SAX object executes a symbolic aggregate approximation (SAX) method for measuring time series distance. The two symbolic sequences that are reduced by SAX are used to calculate the distance between two series. Two distance measures are used: LBDIST and MINDIST. LBDIST is a lower bounding approximation of the Euclidean distance between the original subsequences. MINDIST is the minimum distance between the original time series of two words from the symbolic representation. For more information about SAX time series discretization and the distance measures, see Lin et al. (2003).

Table 13.5 summarizes the methods that are associated with the SAX object.

**Table 13.5** Methods of the SAX Object

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initialize</td>
<td>Initialize the SAX object</td>
</tr>
<tr>
<td>Run</td>
<td>Run the SAX object</td>
</tr>
<tr>
<td>SetInput</td>
<td>Specify an input time series array for the SAX object</td>
</tr>
<tr>
<td>SetOption</td>
<td>Specify options for the SAX object</td>
</tr>
<tr>
<td>SetTarget</td>
<td>Specify a target time series array for the SAX object</td>
</tr>
</tbody>
</table>

**SAX Synopsis**

```
DECLARE OBJECT obj (SAX) ;
```

Method syntax, in order of typical usage:

```c
rc = obj.Initialize () ;
rc = obj.Run () ;
rc = obj.SetInput (InputSeries) ;
rc = obj.SetOption ('Name', Value < ,'Name', Value>) ;
rc = obj.SetTarget (TargetSeries) ;
```
SAX Methods

SAX.Initialize Method

```c
rc=obj.Initialize();
```

Initializes a SAX object with default parameters. This method must be called before you specify input and target time series and other attributes for the SAX object.

**Input Arguments**
There are no arguments associated with this method.

SAX.Run Method

```c
rc=obj.Run();
```

Runs the SAX object to calculate the distance between symbolic sequences of input and target time series. Upon successful completion, the distance and the SAX values can be extracted from the SAX object.

**Input Arguments**
There are no arguments associated with this method.

SAX.SetOption Method

```c
rc=obj.SetOption('Name', Value < , 'Name', Value, .. > );
```

Specifies options for the SAX object.

**Input Arguments**
You can specify one or more of the following 'Names' and the associated *Value*:

- **'DISTMEASURE'**
  takes a string *Value* that specifies a distance measure. You can specify one of the following values:

  - **LBDIST**
    uses the Euclidean distance by applying piecewise aggregate approximation.

  - **MINDIST**
    uses the minimum values between SAX representations of input and target time series for the distance calculation.

  The default is MINDIST.

- **'NBIN'**
  takes a positive integer *Value* that specifies the number of bins for piecewise aggregate approximation (PAA). The default value is 10.

- **'NBREAKPOINT'**
  takes a positive integer *Value* that specifies the number of break points for symbols. It is one less than the number of symbols. The number must be between 1 and 255, inclusive. The default value is 10.
SAX.SetInput Method

```c
rc=obj.SetInput (InputSeries) ;
```

Specifies the input time series, `InputSeries`, for the SAX object.

**Input Arguments**
You must specify the following input argument:

- `InputSeries` takes numeric array variables that specify the input time series for the SAX object.

SAX.SetTarget Method

```c
rc=obj.SetTarget (TargetSeries) ;
```

Specifies the target time series, `TargetSeries`, for the SAX object.

**Input Arguments**
You must specify the following input argument:

- `TargetSeries` takes numeric array variables that specify the target time series for the SAX object.

OUTTSD Object

The OUTTSD object collects distance measures from an distance computational object, and stores them in a CAS table for printing or archiving. The object declaration statement creates a new object, `obj`, whose type is OUTTSD.

Table 13.6 summarizes the method that is associated with the OUTTSD object.

**Table 13.6** Method of the OUTTSD Object

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Collect</td>
<td>Collect distance values from an computational object</td>
</tr>
</tbody>
</table>

Table 13.7 shows the contents of the OUTTSD object.

**Table 13.7** Contents of the OUTTSD Object

<table>
<thead>
<tr>
<th>Column</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Distance</td>
<td>Numeric</td>
<td>Distance between the input and target time series</td>
</tr>
<tr>
<td>InputSeries</td>
<td>String</td>
<td>Name of the input time series</td>
</tr>
<tr>
<td>TargetSeries</td>
<td>String</td>
<td>Name of the target time series</td>
</tr>
</tbody>
</table>
OUTTSD Synopsis

DECLARE OBJECT obj (OUTTSD) ;

Method syntax:

rc = obj.Collect (Obj) ;

OUTTSD Method

OUTTSD.Collect Method

rc = obj.Collect (Obj) ;

Retrieves the results of time series distance run actions from an computational object, Obj, and stores them in a CAS table.

**Input Arguments**

You must specify the following input argument:

*Obj* takes an computational object to use as the source of time series distance output.

OUTTSDDTW Object

The OUTTSDDTW collector object collects warping paths from a DTW object, DTWOBJ, and stores them in a CAS table for printing or archiving. The object declaration statement creates a new object, obj, whose type is OUTTSDDTW.

Table 13.8 summarizes the method of the OUTTSDDTW object.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Collect</td>
<td>Collect dynamic time warping statistics from a DTW object</td>
</tr>
</tbody>
</table>

Table 13.9 shows the contents of the OUTTSDDTW object.

<table>
<thead>
<tr>
<th>Column</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>InputPath</td>
<td>Numeric</td>
<td>Input warping path</td>
</tr>
<tr>
<td>SequenceIndex</td>
<td>Numeric</td>
<td>Sequence index</td>
</tr>
<tr>
<td>TargetPath</td>
<td>Numeric</td>
<td>Target warping path</td>
</tr>
</tbody>
</table>
OUTTSDDTW Synopsis

DECLARE OBJECT obj (OUTTSDDTW) ;

Method syntax:

\[ rc = obj.\text{Collect}(\text{DTWObj}) ; \]

OUTTSDDTW Method

OUTTSDDTW.Collect Method

\[ rc = obj.\text{Collect}(\text{DTWObj}) ; \]


**Input Arguments**

You must specify the following input argument:

- \text{DTWObj} takes a DTW object to use as the output source of a dynamic time warping distance calculation.

OUTTSDLCS Object

The OUTTSDLCS collector object collects longest common subsequence paths from an LCS object, \text{LCSObj}, and stores them in a CAS table for printing or archiving. The object declaration statement creates a new object, \text{obj}, whose type is OUTTSDLCS.

Table 13.10 summarizes the method of the OUTTSDLCS object.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Collect</td>
<td>Collect longest common subsequence statistics from an LCS object</td>
</tr>
</tbody>
</table>

Table 13.11 shows the contents of the OUTTSDLCS object.

<table>
<thead>
<tr>
<th>Column</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>InputPath</td>
<td>Numeric</td>
<td>Input warping path</td>
</tr>
<tr>
<td>SequenceIndex</td>
<td>Numeric</td>
<td>Sequence index</td>
</tr>
<tr>
<td>TargetPath</td>
<td>Numeric</td>
<td>Target warping path</td>
</tr>
</tbody>
</table>
OUTTSDLCS Synopsis

DECLARE OBJECT obj (OUTTSDLCS) ;

Method syntax:

\[ rc = obj.Collect (LCSObj) ; \]

OUTTSDLCS Method

OUTTSDLCS.Collect Method

\[ rc = obj.Collect (LCSObj) ; \]


Input Arguments

You must specify the following input argument:

LCSObj takes a LCS object to use as the output source of a longest common subsequence distance calculation.

OUTTSDSAX Object

The OUTTSDSAX collector object collects the statistics from piecewise aggregate approximation (PAA) and SAX representations of input and target time series from a SAX object, SAXObj, and stores them in a CAS table for printing or archiving. The object declaration statement creates a new object, obj, whose type is OUTTSDSAX.

Table 13.12 summarizes the method of the OUTTSDSAX object.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Collect</td>
<td>Collect PAA statistics and SAX values from a SAX object</td>
</tr>
</tbody>
</table>

Table 13.13 shows the contents of the OUTTSDSAX object.
Table 13.13  Contents of the OUTTSDSAX Object

<table>
<thead>
<tr>
<th>Column</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BinIndex</td>
<td>Numeric</td>
<td>Bin index</td>
</tr>
<tr>
<td>XBinValue</td>
<td>Numeric</td>
<td>Input series bin value</td>
</tr>
<tr>
<td>YBinValue</td>
<td>Numeric</td>
<td>Target series bin value</td>
</tr>
<tr>
<td>XSAXValue</td>
<td>Numeric</td>
<td>Input series SAX value</td>
</tr>
<tr>
<td>YSAXValue</td>
<td>Numeric</td>
<td>Target series SAX value</td>
</tr>
</tbody>
</table>

OUTTSDSAX Synopsis

DECLARE OBJECT obj (OUTTSDSAX) ;

Method syntax:

   rc = obj.Collect (SAXObj) ;

OUTTSDSAX Method

OUTTSDSAX.Collect Method

   rc = obj.Collect (SAXObj) ;

Retrieves PAA statistics and SAX values of a run method of a SAX object. The Collect method stores them for input and target time series.

**Input Arguments**
You must specify the following input argument:

   SAXObj        takes a SAX object to use as the output source of a SAX distance calculation.
Examples: TSD Package

Using CAS Sessions and CAS Engine Librefs

SAS Cloud Analytic Services (CAS) is the analytic server and associated cloud services in SAS Viya. This section describes how to create a CAS session and set up a CAS engine libref that you can use to connect to the CAS session. It assumes that you have a CAS server already available; contact your system administrator if you need help starting and terminating a server. This CAS server is identified by specifying the host on which it runs and the port on which it listens for communications. To simplify your interactions with this CAS server, the host information and port information for the server are stored as SAS option values that are retrieved automatically whenever this CAS server needs to be accessed. You can examine the host and port values for the server at your site by using the following statements:

    proc options option=(CASHOST CASPORT);
    run;

In addition to starting a CAS server, your system administrator might also have created a CAS session and a CAS engine libref for your use. You can define your own sessions and CAS engine librefs that connect to the CAS server as shown in the following statements:

    cas mysess;
    libname mycas cas sessref=mysess;

The CAS statement creates the CAS session named mysess, and the LIBNAME statement creates the mycas CAS engine libref that you use to connect to this session. It is not necessary to explicitly name the CASHOST and CASPORT of the CAS server in the CAS statement, because these values are retrieved from the corresponding SAS option values.

If you have created the mysess session, you can terminate it by using the TERMINATE option in the CAS statement as follows:

    cas mysess terminate;

For more information about the CAS statement and the LIBNAME statement, see *SAS Cloud Analytic Services: User’s Guide*. For general information about CAS and CAS sessions, see *SAS Cloud Analytic Services: Fundamentals*.

In all the examples in this section, the functionality of the TSD package is illustrated using the TSMODEL procedure. This section assumes that you are familiar with the general workings of the TSMODEL procedure. For more information, see Chapter 11, “The TSMODEL Procedure” (*SAS Visual Forecasting: Forecasting Procedures*).


Example 13.1: A Simple Example with DTW

This simple example illustrates how to use the DTW object to measure the distance between two time series. The following code generates two simple time series, x and y, along with a time variable, Time:

```plaintext
data testdata;
  input time y x;
datalines;
  1 2 3
  2 4 5
  3 6 3
  4 7 3
  5 3 3
  6 8 6
  7 9 3
  8 3 8
  9 10 7
 10 11 9;
run;
```

The following statements perform the dynamic time warping distance calculation on the example data set. Specifying the value 100 for the "XWINPCT" and "YWINPCT" arguments mean that no warping restrictions are imposed. SetInput specifies the input variable, and SetTarget specifies the target variable. The distance measure is computed using absolute deviation. The OUTTSDDTW output object is declared to retrieve the warping path by using the OUTPATH option.

```plaintext
proc tsmodel data=mycas.testdata nosummary outlog=mycas.outlog
  outobj=(of= mycas.outtsd(replace=YES)
          ofdtw= mycas.outdtw(replace=YES) );
  var x y;
id time interval=obs;
require tsd;
submit;
  declare object f(DTW);
  declare object of(OUTTSD);
  declare object ofdtw(OUTTSDDTW);
  rc = f.Initialize();
  rc = f.SetInput(x);
  rc = f.SetTarget(y);
  rc = f.SetOption(
    "XWINPCT", 100,
    "YWINPCT", 100,
    "METRIC", "ABSDEV",
    "NORMALIZE", "NONE",
    "OUTPATH","Y"
  );
  rc = f.Run();if rc < 0 then stop;
  rc = of.Collect(f);if rc < 0 then stop;
  rc = ofdtw.Collect(f);if rc < 0 then stop;
endsubmit;
print outlog;
```
Example 13.1: A Simple Example with DTW

Output 13.1.1 shows the distance measure between $x$ and $y$.

**Output 13.1.1** Distance Output Table

<table>
<thead>
<tr>
<th>Obs</th>
<th>Target Series</th>
<th>Input Series</th>
<th>Distance</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$y$</td>
<td>$x$</td>
<td>14</td>
</tr>
</tbody>
</table>

Output 13.1.2 shows the warping path between $x$ and $y$ when the dynamic time warping method is used.

**Output 13.1.2** Time Warping Path Table

<table>
<thead>
<tr>
<th>Obs</th>
<th>Sequence</th>
<th>Input Warping Path</th>
<th>Target Warping Path</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td>6</td>
<td>4</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>7</td>
<td>5</td>
</tr>
<tr>
<td>9</td>
<td>9</td>
<td>8</td>
<td>6</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>8</td>
<td>7</td>
</tr>
<tr>
<td>11</td>
<td>11</td>
<td>9</td>
<td>8</td>
</tr>
<tr>
<td>12</td>
<td>12</td>
<td>10</td>
<td>9</td>
</tr>
<tr>
<td>13</td>
<td>13</td>
<td>10</td>
<td>10</td>
</tr>
</tbody>
</table>

Output 13.1.3 shows the warping path plot between the input and target time series. The plot uses scaled series to show the warping path clearly.

**Output 13.1.3** Scaled Warping Path Plot
Example 13.2: Using Multiple Input and Target Time Series

This example shows distance measures that are calculated from all pairwise combinations between each input time series and each target time series, and it demonstrates that you can calculate the distance measure between two time series of different lengths. When you specify the value "BOTH" for the "TRIM" argument, the beginning and ending missing values are trimmed before the distance is calculated. The following code generates two target time series, $y_1$ and $y_2$, and three input time series, $x_1$, $x_2$, and $x_3$. After trimming on both sides, the length of two series, $y_2$ and $x_3$, is 8, whereas the length of the other time series is 10.

data mtestdata;
  input time y1 y2 x1 x2 x3;
datalines;
1 2 4 7 5 .
2 4 2 5 3 .
3 6 1 3 2 8
4 7 6 3 8 9
5 3 4 6 5 7
6 8 7 6 9 10
7 9 7 3 8 11
8 3 9 8 10 6
9 10 . 7 11 9
10 11 . 6 12 7
run;

Output 13.2.1 shows the overlaid plots of five time series.

Output 13.2.1 Data Plot of Multiple Series

In the following code, you also use the warping path restriction arguments, "XWINSIZE", "YWINSIZE", "XWINPCT", and "YWINPCT":
Example 13.2: Using Multiple Input and Target Time Series

```sas
proc tsmodel data=mycas.mtestdata nosummary outlog=mycas.outlog
   outobj=(of= mycas.outtsdm(replace=YES) );
var x1 x2 x3 y1 y2;
id time interval=obs;
require tsd;
submit;
   declare object f(DTW);
   declare object of(OUTTSD);
   rc = f.Initialize();
   rc = f.SetInput(x1,x2,x3);
   rc = f.SetTarget(y1,y2);
   rc = f.SetOption(
      "XWINSIZE", 5,
      "YWINSIZE", 5,
      "XWINPCT", 50,
      "YWINPCT", 50,
      "METRIC", "RSQRDEV",
      "SCALE", "NONE",
      "NORMALIZE", "NONE",
      "TRIM", "BOTH"
   );
   rc = f.Run();if rc < 0 then stop;
   rc = of.Collect(f);if rc < 0 then stop;
endsubmit;
print outlog;
run;
```

Output 13.2.2 shows the distance measures for all six pairs of target and input time series.

**Output 13.2.2 Distance Output Table**

<table>
<thead>
<tr>
<th>Obs</th>
<th>Target Series</th>
<th>Input Series</th>
<th>Distance</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>y1</td>
<td>x1</td>
<td>8.94427191</td>
</tr>
<tr>
<td>2</td>
<td>y1</td>
<td>x2</td>
<td>5.1961524227</td>
</tr>
<tr>
<td>3</td>
<td>y1</td>
<td>x3</td>
<td>10.392304845</td>
</tr>
<tr>
<td>4</td>
<td>y2</td>
<td>x1</td>
<td>5.0990195136</td>
</tr>
<tr>
<td>5</td>
<td>y2</td>
<td>x2</td>
<td>5.1961524227</td>
</tr>
<tr>
<td>6</td>
<td>y2</td>
<td>x3</td>
<td>11.704699911</td>
</tr>
</tbody>
</table>
Example 13.3: Creating a Distance Matrix for Time Series Clustering

This example illustrates how to use the distance measures in the TSD package for time series clustering. The data set Sashelp.appliance contains 24 variables that record sales histories, but only 9 variables are used in this example. The following statements create a distance measure table. Note that you do not need to set any input variables.

```plaintext
proc tsmodel data=mycas.appliance nosummary outlog=mycas.outlog
    outobj=(of= mycas.outtsddist(replace=YES) );
var units_1 units_2 units_3 units_4 units_5
    units_6 units_7 units_8 units_9;
id cycle interval=obs;
require tsd;
submit;
    declare object f(DTW);
    declare object of(OUTTSD);
    rc = f.Initialize();
    rc = f.SetTarget(units_1, units_2, units_3, units_4,units_5,
    units_6,units_7, units_8, units_9);
    rc = f.SetOption(
        "METRIC", "RSQRDEV",
        "NORMALIZE", "STD",
        "TRIM", "BOTH");
    rc = f.Run();if rc < 0 then stop;
    rc = of.Collect(f);if rc < 0 then stop;
endsubmit;
print outlog;
run;
```

Output 13.3.1 shows the distance table of all pairwise combinations of the nine variables.
### Output 13.3.1 Distance Measure Output Table

<table>
<thead>
<tr>
<th>Obs</th>
<th>Target Series</th>
<th>Input Series</th>
<th>Distance</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>units_1</td>
<td>units_2</td>
<td>3.1867997256</td>
</tr>
<tr>
<td>2</td>
<td>units_1</td>
<td>units_3</td>
<td>5.4261049089</td>
</tr>
<tr>
<td>3</td>
<td>units_1</td>
<td>units_4</td>
<td>5.3448896882</td>
</tr>
<tr>
<td>4</td>
<td>units_1</td>
<td>units_5</td>
<td>4.7587591444</td>
</tr>
<tr>
<td>5</td>
<td>units_1</td>
<td>units_6</td>
<td>3.281580897</td>
</tr>
<tr>
<td>6</td>
<td>units_1</td>
<td>units_7</td>
<td>5.8272838697</td>
</tr>
<tr>
<td>7</td>
<td>units_1</td>
<td>units_8</td>
<td>3.946612765</td>
</tr>
<tr>
<td>8</td>
<td>units_1</td>
<td>units_9</td>
<td>5.3967579915</td>
</tr>
<tr>
<td>9</td>
<td>units_2</td>
<td>units_3</td>
<td>2.7098623405</td>
</tr>
<tr>
<td>10</td>
<td>units_2</td>
<td>units_4</td>
<td>2.6092279348</td>
</tr>
<tr>
<td>11</td>
<td>units_2</td>
<td>units_5</td>
<td>7.1516619991</td>
</tr>
<tr>
<td>12</td>
<td>units_2</td>
<td>units_6</td>
<td>2.6729912798</td>
</tr>
<tr>
<td>13</td>
<td>units_2</td>
<td>units_7</td>
<td>2.7535963286</td>
</tr>
<tr>
<td>14</td>
<td>units_2</td>
<td>units_8</td>
<td>4.1979278904</td>
</tr>
<tr>
<td>15</td>
<td>units_2</td>
<td>units_9</td>
<td>2.6064555917</td>
</tr>
<tr>
<td>16</td>
<td>units_3</td>
<td>units_4</td>
<td>1.3259341185</td>
</tr>
<tr>
<td>17</td>
<td>units_3</td>
<td>units_5</td>
<td>4.6278649128</td>
</tr>
<tr>
<td>18</td>
<td>units_3</td>
<td>units_6</td>
<td>3.644427138</td>
</tr>
<tr>
<td>19</td>
<td>units_3</td>
<td>units_7</td>
<td>3.4234875664</td>
</tr>
<tr>
<td>20</td>
<td>units_3</td>
<td>units_8</td>
<td>4.4705311659</td>
</tr>
<tr>
<td>21</td>
<td>units_3</td>
<td>units_9</td>
<td>1.2218673048</td>
</tr>
<tr>
<td>22</td>
<td>units_4</td>
<td>units_5</td>
<td>5.7116306178</td>
</tr>
<tr>
<td>23</td>
<td>units_4</td>
<td>units_6</td>
<td>2.5117974018</td>
</tr>
<tr>
<td>24</td>
<td>units_4</td>
<td>units_7</td>
<td>2.9011466132</td>
</tr>
<tr>
<td>25</td>
<td>units_4</td>
<td>units_8</td>
<td>5.506460093</td>
</tr>
<tr>
<td>26</td>
<td>units_4</td>
<td>units_9</td>
<td>0.3947960542</td>
</tr>
<tr>
<td>27</td>
<td>units_5</td>
<td>units_6</td>
<td>6.284578749</td>
</tr>
<tr>
<td>28</td>
<td>units_5</td>
<td>units_7</td>
<td>8.4411483352</td>
</tr>
<tr>
<td>29</td>
<td>units_5</td>
<td>units_8</td>
<td>4.1938455874</td>
</tr>
<tr>
<td>30</td>
<td>units_5</td>
<td>units_9</td>
<td>5.8139637569</td>
</tr>
<tr>
<td>31</td>
<td>units_6</td>
<td>units_7</td>
<td>4.1612721338</td>
</tr>
<tr>
<td>32</td>
<td>units_6</td>
<td>units_8</td>
<td>4.6647748931</td>
</tr>
<tr>
<td>33</td>
<td>units_6</td>
<td>units_9</td>
<td>2.4735653503</td>
</tr>
<tr>
<td>34</td>
<td>units_7</td>
<td>units_8</td>
<td>7.4733111191</td>
</tr>
<tr>
<td>35</td>
<td>units_7</td>
<td>units_9</td>
<td>2.9104526164</td>
</tr>
<tr>
<td>36</td>
<td>units_8</td>
<td>units_9</td>
<td>5.5963294629</td>
</tr>
</tbody>
</table>

You can convert the output table in **Output 13.3.1** to a lower triangle distance matrix by using DATA step code. The distance matrix is shown in **Output 13.3.2**.
You can run the following statements with the table in Output 13.3.2 as input data for time series clustering:

```sas
ods graphics on;
proc cluster data=distmatrix(type=distance) method=ward
    plots=dendrogram(height=rsq);
    id targetseries;
run;
```

You can see the clustering results in the dendrogram shown in Output 13.3.3.
Example 13.4: Multivariate Time Series Distance Measure

The data for this example are generated from the accelerometer that is embedded in a smartphone while two activities are performed: drawing a circle and drawing a triangle. The data are collected by drawing a big circle and then drawing a big triangle in the air with the right hand, which is holding a smartphone. Each activity is performed for three seconds, and the data are recorded at 85 time points. Because the accelerometer generates three signals, from the X, Y, and Z axes, the data become a multivariate time series.

The following code shows a snapshot of the collected data. The set of variables with the prefix `crcl` comes from drawing the circle, and the set of variables with the prefix `trgl` comes from drawing the triangle. You can calculate the distance between the two activities, which measures how similar the two activities are to each other.

```r
data CircleAndTriangle;
input time crcl_x crcl_y crcl_z trgl_x trgl_y trgl_z @@;
cards;
1  8.281  -2.004  4.215  4.956  8.979  1.533
2  8.252  -2.044  4.763  5.419  9.236  0.954
3  8.224  -2.188  5.006  5.042  8.81   1.669
4  8.442  -2.052  4.827  4.77   8.849  1.696
5  8.412  -1.974  4.629  4.472  9.014  1.889
6  8.223  -2.04   4.571  3.877  8.766  1.934
7  8.303  -2.059  4.486  2.777  8.55   2.786
```

... more lines ...

The plot in Output 13.4.1 shows the signals that are generated from drawing a circle.

**Output 13.4.1** Activity Signal Plot of Drawing a Circle

The plot in Output 13.4.2 shows the signals that are generated from drawing a triangle.
Chapter 13: Time Series Distance Measure Package

Output 13.4.2 Activity Signal Plot of Drawing a Triangle

The following statements show you how to specify the multivariate distance measure model in the TSMODEL procedure. The number of variables that are specified by SetTarget should be equal to the number of variables that are specified by SetInput for multivariate analysis. Otherwise, the procedure treats this as a pairwise distance calculation task. As a default multivariate method, all distances over the multivariate dimension are summed at each time point.

```
proc tsmodel data=mycas.CircleAndTriangle nosummary outlog=mycas.outlog
   outobj=(of=mycas.outtsdmts(replace=YES));
var crcl_x crcl_y crcl_z trgl_x trgl_y trgl_z;
id time interval=obs;
require tsd;
submit;
declare object f(DTW);
declare object of(OUTTSD);
rc = f.Initialize();
rc = f.SetTarget(crcl_x, crcl_y, crcl_z);
rc = f.SetInput(trgl_x, trgl_y, trgl_z);
rc = f.SetOption("METRIC", "RSQRDEV");
rc = f.Run();if rc < 0 then stop;
rc = of.Collect(f);if rc < 0 then stop;
endsubmit;
print outlog;
run;
```

Output 13.4.3 shows the distance between the two activities, drawing a circle and drawing a triangle.

Output 13.4.3 Multivariate Distance Measure

<table>
<thead>
<tr>
<th>Obs</th>
<th>Target Series</th>
<th>Input Series</th>
<th>Distance</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Target</td>
<td>Input</td>
<td>94.387819283</td>
</tr>
</tbody>
</table>

The following statements show you how to specify the multivariate distance measure model in the TSMODEL procedure. The number of variables that are specified by SetTarget should be equal to the number of variables that are specified by SetInput for multivariate analysis. Otherwise, the procedure treats this as a pairwise distance calculation task. As a default multivariate method, all distances over the multivariate dimension are summed at each time point.

```
proc tsmodel data=mycas.CircleAndTriangle nosummary outlog=mycas.outlog
   outobj=(of=mycas.outtsdmts(replace=YES));
var crcl_x crcl_y crcl_z trgl_x trgl_y trgl_z;
id time interval=obs;
require tsd;
submit;
declare object f(DTW);
declare object of(OUTTSD);
rc = f.Initialize();
rc = f.SetTarget(crcl_x, crcl_y, crcl_z);
rc = f.SetInput(trgl_x, trgl_y, trgl_z);
rc = f.SetOption("METRIC", "RSQRDEV");
rc = f.Run();if rc < 0 then stop;
rc = of.Collect(f);if rc < 0 then stop;
endsubmit;
print outlog;
run;
```

Output 13.4.3 shows the distance between the two activities, drawing a circle and drawing a triangle.

Output 13.4.3 Multivariate Distance Measure

<table>
<thead>
<tr>
<th>Obs</th>
<th>Target Series</th>
<th>Input Series</th>
<th>Distance</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Target</td>
<td>Input</td>
<td>94.387819283</td>
</tr>
</tbody>
</table>
Another way to specify multivariate time series distance is to use the first principal component of each set of activity signals by specifying the value "PCA" for the "MTSMETHOD" argument. The following code shows the PCA option specification:

```sas
proc tsmodel data=mycas.CircleAndTriangle nosummary outlog=mycas.outlog
   outobj=(of= mycas.outtsdmtspca(replace=YES) );
var crcl_x crcl_y crcl_z trgl_x trgl_y trgl_z;
id time interval=obs;
require tsd;
submit;
   declare object f(DTW);
   declare object of(OUTTSD);
   rc = f.Initialize();
   rc = f.SetTarget(crcl_x, crcl_y, crcl_z);
   rc = f.SetInput(trgl_x, trgl_y, trgl_z);
   rc = f.SetOption("METRIC", "RSQRDEV",
      "MTSMETHOD", "PCA");
   rc = f.Run();if rc < 0 then stop;
   rc = of.Collect(f);if rc < 0 then stop;
endsubmit;
print outlog;
run;
```

Output 13.4.4 shows the distance between the two activities, drawing a circle and drawing a triangle, which is specified using their first principal components.

Output 13.4.4 Multivariate Distance Measure Using PCA

<table>
<thead>
<tr>
<th>Obs</th>
<th>Target</th>
<th>Series</th>
<th>Input</th>
<th>Distance</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Target</td>
<td>Input</td>
<td></td>
<td>24.257887358</td>
</tr>
</tbody>
</table>

Example 13.5: Finding the Longest Common Subsequence and Distance

This simple example illustrates how to use the LCS object to find the longest common subsequence and calculate the distance measure between two time series. The following code generates two simple time series, \( x \) and \( y \), along with a time variable, \( \text{Time} \):

```sas
proc format;
   value letter 1='A' 2='B' 3='C' 4='D' 5='E' 6='F'
      7='G' 8='H' 9='I' 10='J' 11='K' 12='L'
      13='M' 14='N' 15='O' 16='P' 17='Q' 18='R'
      19='S' 20='T' 21='U' 22='V' 23='W' 24='X'
      25='Y' 26='Z';
run;

data lcstestdata;
   input time x y;
   format time best12. x letter. y letter.;
datalines;
```
Output 13.5.1 shows the formatted x and y variables.

### Output 13.5.1 Formatted Input (x) and Target (y) Series

<table>
<thead>
<tr>
<th>Obs</th>
<th>time</th>
<th>x</th>
<th>y</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>S</td>
<td>K</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>B</td>
<td>L</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>A</td>
<td>S</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>J</td>
<td>E</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>D</td>
<td>I</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>L</td>
<td>A</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td>C</td>
<td>O</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>H</td>
<td>S</td>
</tr>
<tr>
<td>9</td>
<td>9</td>
<td>S</td>
<td>M</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>G</td>
<td>X</td>
</tr>
</tbody>
</table>

The following statements find the longest common subsequence and calculate the distance between two sequences in the example data set. SetInput specifies the input variable, and SetTarget specifies the target variable. The value 0.1 for the "FUZZ" argument means that if the distance between two points is less than the specified value, then they are considered the same value. The OUTTSD output object is declared to retrieve the distance measure. The OUTTSDLCS output object is declared to retrieve the longest common subsequence and its path by using the OUTPATH option.

```r
proc tsmodel data=mycas.lcstestdata nosummary outlog=mycas.outlog
  outobj=(of= mycas.outtsd(replace=YES)
          oflcs= mycas.outlcs(replace=YES) );
var x  y;
id time interval=obs;
require tsd;
submit;
  declare object f(LCS);
  declare object of(OUTTSD);
  declare object oflcs(OUTTSDLCS);
  rc = f.Initialize();
  rc = f.SetInput(x);
  rc = f.SetTarget(y);
  rc = f.SetOption("FUZZ", 0.1, "OUTPATH","Y");
  rc = f.Run();if rc < 0 then stop;
```
Example 13.6: Calculating the SAX Distance

This simple example illustrates how to use the SAX object to calculate the distance measure on the basis of the symbolic aggregate approximation between two time series. The following code generates two simple time series, \( x \) and \( y \), along with a time variable, \( \text{Time} \):

```sas
data saxtestdata;
  input time x y;
datalines;
  1 6.2 3.5
  2 7.1 4.7
  3 3.4 6.7
  4 3.9 7.8
  5 6.5 3.7
  6 5.6 2.7
  7 9.8 5.3
  8 8.2 4.5
  9 6.4 8.5
  10 7.1 9.5;
run;
```

The following statements calculate the SAX-based distance between two sequences in the example data set. SetInput specifies the input variable, and SetTarget specifies the target variable. Specifying the value 5 for the "NBIN" argument means that you divide the X axis into five bins for piecewise aggregate approximation.
(PAA), and specifying the value 10 for the "NBREAKPOINT" argument means that you use 11 symbols for the SAX representation. The OUTTSD output object is declared to retrieve the distance measure. The OUTTSDSAX output object is declared to retrieve the PAA values and the SAX representations for the input and target series.

```
proc tsmodel data=mycas.saxtestdata nosummary outlog=mycas.outlog
  outobj=(of= mycas.outtsd(replace=YES)
    ofsax= mycas.outsax(replace=YES) );
var x y;
id time interval=obs;
require tsd;
submit;
  declare object f(SAX);
  declare object of(OUTTSD);
  declare object ofsax(OUTTSDSAX);
  rc = f.Initialize();
  rc = f.SetInput(x);
  rc = f.SetTarget(y);
  rc = f.SetOption("DISTMEASURE", "MINDIST",
     "NBIN", 5,
     "NBREAKPOINT", 10 );
  rc = f.Run();if rc < 0 then stop;
  rc = of.Collect(f);if rc < 0 then stop;
  rc = ofsax.Collect(f);if rc < 0 then stop;
endsubmit;
print outlog;
run;
```

Output 13.6.1 shows the MINDIST measure between $x$ and $y$.

**Output 13.6.1** SAX Distance Output Table

<table>
<thead>
<tr>
<th>Obs</th>
<th>Target Series</th>
<th>Input Series</th>
<th>Distance</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>y</td>
<td>x</td>
<td>3.9156762458</td>
</tr>
</tbody>
</table>

Output 13.6.2 shows the PAA and SAX values for each bin. The SAX values are numeric, but they are formatted for display purposes.

**Output 13.6.2** PAA and SAX Values for Each Bin

<table>
<thead>
<tr>
<th>Obs</th>
<th>Bin Index</th>
<th>Input Series Bin Value</th>
<th>Target Series Bin Value</th>
<th>Input Series SAX Value</th>
<th>Target Series SAX Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0.1225208176</td>
<td>-0.687566898</td>
<td>E</td>
<td>I</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>-1.475576803</td>
<td>0.6745939376</td>
<td>K</td>
<td>C</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>-0.197098707</td>
<td>-1.076755708</td>
<td>G</td>
<td>J</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>1.3743639536</td>
<td>-0.341621289</td>
<td>A</td>
<td>G</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>0.1757907383</td>
<td>1.4313499573</td>
<td>E</td>
<td>A</td>
</tr>
</tbody>
</table>
Example 13.7: Distance Measures in the DST Object

This example illustrates how to use the DST object to calculate the various distance measures between two time series. The measures include $L_0$, $L_1$, $L_2$, $L_\infty$, and cosine. The following code generates two simple time series, $x$ and $y$, along with a time variable, time:

```sas
data dsttestdata;
    input time x1 y1;
datalines;
    1 3 2
    2 2 4
    3 3 6
    4 3 7
    5 4 3
    6 6 8
    7 3 9
    8 8 3
    9 6 7
    10 7 9;
run;

data dsttestdata;
    set dsttestdata;
    x2 = x1+rannor(122);
    x3 = x1+rannor(233);
    y2 = y1+rannor(565);
    y3 = y1+rannor(585);
run;
```

The following statements calculate the $L_1$ and cosine distance between two sequences in the example data set. SetInput specifies the input variable, and SetTarget specifies the target variable. You can set the distance measure by using the argument "METRIC". The OUTTSD output object is declared to retrieve the distance measure.

```sas
proc tsmodel data=mycas.dsttestdata nosummary outlog=mycas.outlog
    outobj=(of= mycas.outtsdL1(replace=YES));
    var x1 x2 x3 y1 y2 y3;
    id time interval=obs;
    require tsd;
    submit;
    declare object f(DST);
    declare object of(OUTTSD);
    rc = f.Initialize();
    rc = f.SetInput(x1,x2,x3,y2,y3);
    rc = f.SetTarget(y1);
    rc = f.SetOption("METRIC", "L1",
                   "NORMALIZE", "STD"
                     );
    rc = f.Run();if rc < 0 then stop;
    rc = of.Collect(f);if rc < 0 then stop;
```
Output 13.7.1 shows the $L_1$ distance measure between the target time series, $y_1$, and the input time series, $x_1, x_2, x_3, y_2,$ and $y_3$.

### Output 13.7.1 $L_1$ Distance Output Table

<table>
<thead>
<tr>
<th>Obs</th>
<th>Target Series</th>
<th>Input Series</th>
<th>Distance</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$y_1$</td>
<td>$x_1$</td>
<td>9.1696231252</td>
</tr>
<tr>
<td>2</td>
<td>$y_1$</td>
<td>$x_2$</td>
<td>7.9714606034</td>
</tr>
<tr>
<td>3</td>
<td>$y_1$</td>
<td>$x_3$</td>
<td>8.7674900719</td>
</tr>
<tr>
<td>4</td>
<td>$y_1$</td>
<td>$y_2$</td>
<td>2.6464676483</td>
</tr>
<tr>
<td>5</td>
<td>$y_1$</td>
<td>$y_3$</td>
<td>3.5114245119</td>
</tr>
</tbody>
</table>

```
proc tsmodel data=mycas.dsttestdata nosummary outlog=mycas.outlog
outobj=(of= mycas.outtsdCS(replace=YES));
var x1 x2 x3 y1 y2 y3;
id time interval=obs;
require tsd;
submit;
  declare object f(DST);
  declare object of(OUTTSD);
  rc = f.Initialize();
  rc = f.SetTarget(x1,x2,x3,y2,y3,y3);
  rc = f.SetOption("METRIC", "COSINE",
                  "NORMALIZE", "NONE" );
  rc = f.Run();if rc < 0 then stop;
  rc = of.Collect(f);if rc < 0 then stop;
endsubmit;
print outlog;
run;
```

Output 13.7.2 shows the cosine distance measure of all pairs among the nonstandardized target time series, $x_1, x_2, x_3, y_1, y_2,$ and $y_3$.

```
proc tsmodel data=mycas.dsttestdata nosummary outlog=mycas.outlog
outobj=(of= mycas.outtsdCS(replace=YES));
```
Output 13.7.2 $L_1$ Distance Output Table

<table>
<thead>
<tr>
<th>Obs</th>
<th>Target Series</th>
<th>Input Series</th>
<th>Distance</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>x1</td>
<td>x2</td>
<td>0.0228341374</td>
</tr>
<tr>
<td>2</td>
<td>x1</td>
<td>x3</td>
<td>0.0090383817</td>
</tr>
<tr>
<td>3</td>
<td>x1</td>
<td>y2</td>
<td>0.1468837029</td>
</tr>
<tr>
<td>4</td>
<td>x1</td>
<td>y3</td>
<td>0.1272766364</td>
</tr>
<tr>
<td>5</td>
<td>x1</td>
<td>y3</td>
<td>0.1272766364</td>
</tr>
<tr>
<td>6</td>
<td>x2</td>
<td>x3</td>
<td>0.0266434357</td>
</tr>
<tr>
<td>7</td>
<td>x2</td>
<td>y2</td>
<td>0.1226881719</td>
</tr>
<tr>
<td>8</td>
<td>x2</td>
<td>y3</td>
<td>0.1193424429</td>
</tr>
<tr>
<td>9</td>
<td>x2</td>
<td>y3</td>
<td>0.1193424429</td>
</tr>
<tr>
<td>10</td>
<td>x3</td>
<td>y2</td>
<td>0.1399516788</td>
</tr>
<tr>
<td>11</td>
<td>x3</td>
<td>y3</td>
<td>0.1397733572</td>
</tr>
<tr>
<td>12</td>
<td>x3</td>
<td>y3</td>
<td>0.1397733572</td>
</tr>
<tr>
<td>13</td>
<td>y2</td>
<td>y3</td>
<td>0.0192516359</td>
</tr>
<tr>
<td>14</td>
<td>y2</td>
<td>y3</td>
<td>0.0192516359</td>
</tr>
<tr>
<td>15</td>
<td>y3</td>
<td>y3</td>
<td>1.110223E-16</td>
</tr>
</tbody>
</table>

References


Chapter 14
Time Series Model Package

Contents

<table>
<thead>
<tr>
<th>Overview: TSM Package</th>
<th>490</th>
</tr>
</thead>
<tbody>
<tr>
<td>TSM Package Summary</td>
<td>490</td>
</tr>
<tr>
<td>Using the TSM Package</td>
<td>493</td>
</tr>
<tr>
<td>Return Codes</td>
<td>495</td>
</tr>
<tr>
<td>TSM Object</td>
<td>495</td>
</tr>
<tr>
<td>TSM Synopsis</td>
<td>497</td>
</tr>
<tr>
<td>TSM Methods</td>
<td>497</td>
</tr>
<tr>
<td>CFC Object</td>
<td>502</td>
</tr>
<tr>
<td>CFC Synopsis</td>
<td>504</td>
</tr>
<tr>
<td>CFC Methods</td>
<td>504</td>
</tr>
<tr>
<td>ESMSPEC Object</td>
<td>510</td>
</tr>
<tr>
<td>ESMSPEC Synopsis</td>
<td>511</td>
</tr>
<tr>
<td>ESMSPEC Methods</td>
<td>512</td>
</tr>
<tr>
<td>ARIMASPEC Object</td>
<td>515</td>
</tr>
<tr>
<td>ARIMASPEC Synopsis</td>
<td>517</td>
</tr>
<tr>
<td>ARIMASPEC Methods</td>
<td>517</td>
</tr>
<tr>
<td>EXMSPEC Object</td>
<td>523</td>
</tr>
<tr>
<td>EXMSPEC Synopsis</td>
<td>525</td>
</tr>
<tr>
<td>EXMSPEC Methods</td>
<td>525</td>
</tr>
<tr>
<td>IDMSPEC Object</td>
<td>527</td>
</tr>
<tr>
<td>IDMSPEC Synopsis</td>
<td>528</td>
</tr>
<tr>
<td>IDMSPEC Methods</td>
<td>528</td>
</tr>
<tr>
<td>UCMSPEC Object</td>
<td>532</td>
</tr>
<tr>
<td>UCMSPEC Synopsis</td>
<td>533</td>
</tr>
<tr>
<td>UCMSPEC Methods</td>
<td>534</td>
</tr>
<tr>
<td>TSMPEST Object</td>
<td>539</td>
</tr>
<tr>
<td>TSMPEST Synopsis</td>
<td>540</td>
</tr>
<tr>
<td>TSMPEST Methods</td>
<td>540</td>
</tr>
<tr>
<td>TSMSPEC Object</td>
<td>542</td>
</tr>
<tr>
<td>TSMSPEC Synopsis</td>
<td>543</td>
</tr>
<tr>
<td>TSMSPEC Methods</td>
<td>543</td>
</tr>
<tr>
<td>TSMFOR Object</td>
<td>543</td>
</tr>
<tr>
<td>TSMFOR Synopsis</td>
<td>545</td>
</tr>
<tr>
<td>TSMFOR Methods</td>
<td>546</td>
</tr>
<tr>
<td>TSMSTAT Object</td>
<td>547</td>
</tr>
</tbody>
</table>
Overview: TSM Package

The time series model (TSM) package contains a set of time series modeling objects that provide a flexible
way to model and forecast time series. For more information about the statistical methodology that underlies
this package, see relevant chapters in SAS/ETS User’s Guide and SAS Forecast Server Procedures: User’s
Guide.

TSM Package Summary

Table 14.1 lists the objects that are contained in the TSM package. These objects are designed to provide
access to various univariate time series model families.

<table>
<thead>
<tr>
<th>Table 14.1</th>
<th>Objects in the Time Series Model Package</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Object</strong></td>
<td><strong>Description</strong></td>
</tr>
<tr>
<td>Central Objects</td>
<td></td>
</tr>
<tr>
<td>TSM</td>
<td>Time series model object (Note 1)</td>
</tr>
<tr>
<td>CFC</td>
<td>Combine Forecasts object (Note 2)</td>
</tr>
<tr>
<td>Time Series Model Specification Objects (Note 3)</td>
<td></td>
</tr>
<tr>
<td>ARIMASPEC</td>
<td>Autoregressive integrated moving average model specification object</td>
</tr>
<tr>
<td>ESMSPEC</td>
<td>Exponential smoothing model specification object</td>
</tr>
</tbody>
</table>
**Table 14.1 continued**

<table>
<thead>
<tr>
<th>Object</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EXMSPEC</td>
<td>External model specification object</td>
</tr>
<tr>
<td>IDMSPEC</td>
<td>Intermittent demand model specification object</td>
</tr>
<tr>
<td>UCMSPEC</td>
<td>Unobserved component model (UCM) specification object</td>
</tr>
</tbody>
</table>

**Time Series Model Collector Objects (Note 4)**
- TSMFOR: Time series model forecast collector object
- TSMPEST: Time series model parameter estimates collector object
- TSMSPEC: Time series model specification collector object
- TSMSTAT: Time series model fit statistics collector object

**Time Series Model Repeater Objects (Note 5)**
- TSMINEST: Time series model input parameter estimates repeater object
- TSMINSPEC: Time series model input specification repeater object

**Note:**

1. The TSM object is the central hub that interacts with all other objects. It executes and encapsulates the computational services for all univariate time series models.
2. The Combine Forecasts (CFC) object allows you to perform forecast combinations.
3. The time series model specification objects enable you to specify the characteristics of a time series model to be executed by a TSM object.
4. The time series model collector objects operate on TSM objects to collect and save various results from the time series model execution (fit statistics, forecasts, parameter estimates, model specifications, and so on).
5. The time series model repeater objects act as conduits to replay collected model specifications and model parameter estimates as input to other TSM objects.

Figure 14.1 diagrams the relationships among the objects in the TSM package.
Chapter 14: Time Series Model Package

Figure 14.1 TSM Object Data Flow

- ARIMASPEC
- ESMSPEC
- UCMSPEC
- IDMSPEC
- EXMSPEC

xxxSpec

--

TSMSPEC

Inobj = {}

TSM

Initialize

Replay(*)

TSMSPEC

TSMFOR

TSMPEST

OUTEST

Table

TSMINESST

Outobj = {}

Inobj = {}

Replay(*)

GetForecast

Collect

OUTFOR

Table

TSMSTAT

OUTSTAT

Table

Outobj = {}

Collect

Criterion

Forecast Lead

Hold Back Sample

Holdout Sample

Confidence Level

---

xxxSpec

TSMSPEC

Table

TSMINSPEC

Inobj = {}

TSM

Initialize

Replay(*)

TSMSPEC

TSMFOR

TSMPEST

OUTEST

Table

TSMINESST

Outobj = {}

Inobj = {}

Replay(*)

GetForecast

Collect

Nfor

Criterion

OUTFOR

Table

TSMSTAT

OUTSTAT

Table

Outobj = {}

Collect
Using the TSM Package

The following steps provide a general outline of how to use each type of object in the TSM package. Subsequent sections describe each step in greater detail.

1. Configure a model specification object.
2. Use the TSM object with the model specification object to generate a forecast.
3. Use TSM collector objects to extract results and parameter estimates from the TSM object.
4. Use TSM repeater objects to replay collected parameters and specifications to another TSM object.

Step 1: Configure a Model Specification Object

Model specification objects define the time series model characteristics that you want to be applied to the TSM object, which then performs the model execution. Model specification objects use Open and Close methods to initialize and finalize model specifications. The basic execution pattern follows this sequence of operations:

1. **Declare**: Create the model specification object by using the object declaration statement. The object declaration assigns a default model specification to the specification objects.
2. **Open**: Initialize the specification object to a default state that is ready to accept configuration methods that shape the model and define its characteristics.
3. **Configure**: Use object-specific model specification methods to configure the model. For example, in an ESMSPEC, you might specify the smoothing method or model to be used, specify a functional transformation for the dependent variable to force the use of specific smoothing parameters, or specify bounds on the estimated parameters. For an ARIMASPEC, you might specify a set of autoregressive or moving average backshift operator polynomial factors, specify differencing operators, add simple and complex transfer functions, or specify a functional transformation for the dependent variable.
4. **Close**: Declare the model specification object to be complete and ready for use.
5. **Use**: Use the completed model specification object with a TSM object to directly perform the specified time series model fit and forecast computations. You can also use model specification objects with the automatic time series modeling (ATSM) package to include custom models into its automatic time series forecasting process. For more information about this use, see Chapter 3, “Automatic Time Series Modeling Package.”
Step2: Use a TSM Object with a Model Specification Object

The TSM object executes the time series model. The TSM object is configured with a time series model from one of the model specification objects. Then the TSM object applies that configuration to the time series data to produce a forecast. The basic execution pattern follows this sequence of operations:

1. **Declare**: Create the TSM object by using the object declaration statement.
2. **Initialize**: Add a model specification object to the TSM object.
3. **Specify variables**: Specify the dependent series (Y) and any independent series (X) variables.
4. **Specify options**: Specify other options and properties as appropriate.
5. **Run**: Execute the model in the TSM object to produce its forecast.
6. **Extract**: Extract the results by using collector objects.

Various properties (attributes) of the executed TSM object can be queried directly and saved into declared variables, or the results can be collected by the TSM collector objects for presentation and storage.

Step3: Use the TSM Collector Objects

Collector objects enable you to create a snapshot of results from TSM objects and store those results in CAS tables. Each collector object defines a table schema that is determined by the collector object’s design. The TSM collector objects follow a common pattern. The basic execution follows this sequence of operations:

1. **Declare**: Create the collector object using the object declaration statement.
2. **Collect**: Use the Collect method with a TSM object passed in as an argument to collects results from the TSM object. For example, the TSMPEST object collects parameter estimates from a TSM object, and the TSMSPEC object collects from a model specification’s XML. Rows that are collected are automatically appended to the collector’s associated CAS table at the end of each BY group, and the collector object’s saved row set is automatically reset. Rows that are added to the CAS table are qualified by the values of the corresponding values of the BY variable. This enables repeater objects to locate the rows that are relevant to each BY group and correctly replay that information into a TSM object. The **nrows** attribute returns the current row count in the collector. A missing value is returned if nothing has been collected. The data can now be used for reports or used by a repeater object on another model.

Step 4: Use the TSM Repeater Objects

Repeater objects read rows from a CAS table and convert their contents back into useful information that can be used by other TSM objects. This is the inverse function of the collector objects. Each repeater object defines a CAS table schema that is determined by its counterpart collector object’s design. Repeater objects must be associated with an existing CAS table that has the table schema that is required by the repeater object. The basic execution follows this sequence of events:

1. **Declare**: Create the repeater object by using the object declaration statement.
**Replay:** Use the TSM.Replay method on a TSM object with the repeater object passed in as an argument to execute a new time series model. The replayed TSM object generates the forecast data that was produced by the new time series model. Then the data can be collected by using a collector object as described in Step 3.

---

**Return Codes**

Table 14.2 shows the return code (designated by rc in method statements) status values that are used in this package. These status code values are returned after a method that is associated with an object is called; they can help determine whether the method executed successfully.

<table>
<thead>
<tr>
<th>Status</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt; 0</td>
<td>An unrecoverable error occurred. No result was produced.</td>
</tr>
<tr>
<td>= 0</td>
<td>Unconditional success. The requested action was completed, and a normal result was produced.</td>
</tr>
<tr>
<td>&gt; 0</td>
<td>Conditional success or warning. A result was produced subject to conditions.</td>
</tr>
</tbody>
</table>

---

**TSM Object**

The TSM object generates forecasts of univariate time series.

Table 14.3 lists the time series model families that are supported.

<table>
<thead>
<tr>
<th>Family</th>
<th>Object</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARIMA</td>
<td>ARIMASPEC</td>
<td>ARIMAX models</td>
</tr>
<tr>
<td>ESM</td>
<td>ESMSPEC</td>
<td>Exponential smoothing models</td>
</tr>
<tr>
<td>EXM</td>
<td>EXMSPEC</td>
<td>External model (external forecast)</td>
</tr>
<tr>
<td>IDM</td>
<td>IDMSPEC</td>
<td>Intermittent demand models (Croston’s/average demand (ADEM))</td>
</tr>
<tr>
<td>UCM</td>
<td>UCMSPEC</td>
<td>Unobserved component models</td>
</tr>
</tbody>
</table>

Table 14.4 summarizes the methods that are associated with the TSM object.
Table 14.4  Methods of the TSM Object

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AddX</td>
<td>Add an independent time series array (XSeries) for the TSM object</td>
</tr>
<tr>
<td>AddExternal</td>
<td>Add external forecast component series for the TSM object</td>
</tr>
<tr>
<td>criterion</td>
<td>Return the final fit statistic over a specified forecast region</td>
</tr>
<tr>
<td>GetForecast</td>
<td>Get the forecast series</td>
</tr>
<tr>
<td>Initialize</td>
<td>Initialize the TSM object</td>
</tr>
<tr>
<td>nfor</td>
<td>Return the forecast series length</td>
</tr>
<tr>
<td>Replay</td>
<td>Replay the restored model and parameter estimates</td>
</tr>
<tr>
<td>Run</td>
<td>Run the TSM object</td>
</tr>
<tr>
<td>SetOption</td>
<td>Specify the named option for the TSM object</td>
</tr>
<tr>
<td>SetY</td>
<td>Specify the dependent time series array (YSeries) for the TSM object</td>
</tr>
</tbody>
</table>

The basic execution pattern for using a TSM object follows this sequence of operations:

1 **Declare**: The object declaration statement creates a new TSM object.

2 **Initialize**: The TSM.Initialize method takes a model specification object as its argument and initializes
   the TSM object for that specified time series model. If no model specification object is provided, the
   TSM object is initialized as an exponential smoothing method (ESM) that uses the best suited exponential
   smoothing model. This is equivalent to initializing the TSM object with an ESMSPEC object that has
   default option values.

3 **SetY**: The TSM.SetY method defines the dependent time series for the TSM object.

4 **AddX**: The TSM.AddX method defines any independent time series for the TSM object. Each call defines
   one predictor series. Repeat as needed for each predictor series.

5 **SetOption**: The TSM.SetOption method specifies any options that affect the running of the model. Each
   call defines an option. Repeat as needed to specify all options that are required.

6 **Run**: The TSM.Run method uses its currently configured X and Y time series data to execute the time
   series model that is defined by the TSM object’s model specification. At completion, the model has
   estimated the parameters and produced a final forecast based on these parameters.
TSM Synopsis

DECLARE OBJECT obj (TSM) ;

Method syntax, in order of typical usage:

\[ rc = \text{obj}.\text{Initialize} (<\text{ModelSpec}>) ; \]
\[ rc = \text{obj}.\text{SetY} (\text{YSeries}) ; \]
\[ rc = \text{obj}.\text{AddX} (\text{XSeries} <,\text{Required},\text{NoDiff},\text{ModelSymbol}>) ; \]
\[ rc = \text{obj}.\text{AddExternal} (\text{Series} <,\text{Role}>) ; \]
\[ rc = \text{obj}.\text{SetOption} ('\text{Name}', \text{Value} <,'\text{Name}', \text{Value},...) >) ; \]
\[ rc = \text{obj}.\text{Replay} (\text{TSMINSPECObj} <,\text{TSMINESTObj}>) ; \]
\[ rc = \text{obj}.\text{Run} () ; \]
\[ rc = \text{obj}.\text{GetForecast} (\text{Which}, \text{Result}) ; \]
\[ nfor = \text{obj}\text{.nfor} () ; \]
\[ \text{criterion} = \text{obj}\text{.criterion} (\text{Region}) ; \]

TSM Methods

TSM.AddExternal Method

\[ rc = \text{obj}.\text{AddExternal} (\text{Series} <,\text{Role}>) ; \]

Adds a time series array, Series, for use in external model computations when the TSM object is initialized from an external model specification (EXMSPEC) object. Calling this method when the TSM object is not configured with an EXMSPEC results in an error return and no further action. For more information about external model support, see the section “EXMSPEC Object” on page 523.

No default role mapping is implied by the name of the Series variable that you specify in the method. Each call to the AddExternal method adds the specified series to the TSM object according to its role in the external model. That role association happens in one of the following ways:

1. You specify the Role argument in the method call. This takes precedence over any role mapping that is defined in the EXMSPEC that was used in the TSM.Initialize method. If you specify an invalid Role value, an error is generated.

2. You specify a Series variable that matches a role mapping in the EXMSPEC object that was used in the TSM.Initialize method. If you specify a Series variable that fails to match a role mapping in the EXMSPEC, an error is generated.

This method can be called as many times as needed to specify all of the external series that are required to run the external model. In all cases, if the series that you specify to add fails to resolve to a role in the EXMSPEC object, an error is generated without the series being included in the TSM object. Such failures do not cause subsequent TSM.AddExternal method calls to fail.
Input Arguments
You must specify the following input arguments:

**Series**
specifies a numeric array that contains an external forecast series for the TSM object.

**Role**
is a case-sensitive character string that specifies the role of the external forecast series in the external model. You can specify one of the following values:

- **ERROR** returns prediction errors.
- **LOWER** returns a lower confidence limit series.
- **STDERR** returns a prediction standard error series.
- **PREDICT** returns a prediction series.
- **UPPER** returns an upper confidence limit series.

TSM.AddX Method

```rc=obj.AddX (XSeries <,Required,NoDiff,ModelSymbol>) ;```

Adds an independent time series array (**XSeries**) for the TSM object. Each call of the TSM.AddX method adds the specified **XSeries** array variable to the TSM object. This method can be called as many times as needed to specify all the independent variables. By default, the name of the **XSeries** variable must match the name of an input symbol in the model specification that is used to configure the TSM object. You can specify a symbol name, **ModelSymbol**, to associate an **XSeries** array to an input symbol in the model specification. Only ARIMA models support **XSeries** predictors.

Input Arguments
You must specify the following input argument:

**XSeries** specifies a numeric array that contains an independent series for the TSM object.

You can also specify the following input arguments:

**Required** takes a Boolean value (0 or 1) that, when set to 1, specifies that the **XSeries** variable is required to be in the model. This might cause the model estimation to fail if the **XSeries** array variable is deemed inadmissible for inclusion into the underlying ARIMA model. The default value is 0.

**NoDiff** takes a Boolean value (0 or 1) that, when set to 1, specifies that the **XSeries** variable does not automatically follow the **XSeries** variable differencing. The default value is 0.

**ModelSymbol** takes a character variable that specifies the name of an input symbol in the model specification to be associated with the **XSeries** variable when the model is run. By default, the TSM.AddX method adds **XSeries** variables into an ARIMA model specification if they are not already referenced as an input symbol. Specification of **ModelSymbol** defines a way to include the **XSeries** variable if the ARIMASPEC object includes a symbol that matches the **ModelSymbol** argument.
**TSM.criterion Method**

```c
 criterion = obj.criterion (Region) ;
```

Returns the fit statistic value over the specified forecast Region for the TSM object. The criterion is set via the 'CRITERION' argument in the TSM.SetOption method. A missing value indicates that the TSM object instance has not produced a successful forecast.

**Input Arguments**
You must specify the following input argument:

- **Region** is a case-insensitive character string that specifies the forecast region over which the fit statistic is computed. You can specify one of the following values:
  - **BACK** returns the fit statistic over the time region that is subsequent to the FIT region and that did not contribute any data for estimating model parameters (that is, the model forecast region).
  - **FIT** returns the fit statistic over the time region that supplied observations for estimating model parameters (that is, the model fit region).

**TSM.GetForecast Method**

```c
 rc = obj.GetForecast (Which, Result) ;
```

Gets the specified forecast series (Which) from the TSM object and stores it in the specified numeric array (Result).

**Input Arguments**
You must specify the following input argument:

- **Which** is a case-insensitive character string that specifies the type of forecast series to return. You can specify one of the following values:
  - **ERROR** returns prediction errors.
  - **LOWER** returns a lower confidence limit series.
  - **STDERR** returns a prediction standard error series.
  - **PREDICT** returns a prediction series.
  - **UPPER** returns an upper confidence limit series.

**Output Arguments**
You must specify the following output argument:

- **Result** specifies a numeric array to receive the forecast series.
TSM.Initialize Method

\[ rc = \text{obj}.\text{Initialize (<ModelSpec>)} ; \]

Initializes a TSM object to use the specified ModelSpec. This method must be called before the time series arrays (XSeries and YSeries) and other attributes for the TSM object are specified. If no ModelSpec object is specified, the TSM object is initialized to use the default ESM specification. This is equivalent to initializing the TSM object with an ESMSPEC object that has default option values.

**Input Arguments**

You can specify the following input argument:

- **ModelSpec** specifies an optional name for a TSM model specification object that is used to configure the TSM object.

TSM.nfor Method

\[ nfor = \text{obj}.\text{nfor} () ; \]

Returns the length (observation count) of the forecast series for the TSM object. A missing value indicates that the TSM object has not produced a successful forecast.

**Arguments**

There are no arguments associated with this method.

TSM.Replay Method

\[ rc = \text{obj}.\text{Replay (TSMINSPECObj < ,TSMINESTObj>)} ; \]

Uses a previously saved model specification from a TSMINSPECObj as input to another TSM object. Optionally, restored parameter estimates from TSMINESTObj are applied to the model specification.

**Input Arguments**

You must specify the following input argument:

- **TSMINSPECObj** specifies the TSMINSPEC object to supply the model specification.

You can also specify the following input argument:

- **TSMINESTObj** specifies the TSMINEST object to supply the model’s parameter estimates.

TSM.Run Method

\[ rc = \text{obj}.\text{Run} () ; \]

Runs the TSM object to estimate and forecast the time series model by using the specified dependent (YSeries) and independent (XSeries) series. Upon successful completion, various results can be extracted from the TSM object.
**Arguments**

There are no arguments associated with this method.

**TSM.SetOption Method**

```plaintext
rc=obj.SetOption ('Name', Value <, 'Name', Value, ...>);
```

Specifies the named options for the TSM object. When you invoke this method, the previous forecast produced by the TSM.Run method is discarded. You must then rerun the TSM.Run method in order to produce a new forecast using the updated configuration options.

**Input Arguments**

You must specify at least one of the following 'Names' and its associated Value:

- **'ALPHA'** takes a numeric Value between 0 and 1, exclusive, that specifies the significance level for forecast confidence bands. The default value is 0.05.
- **'BACK'** takes a nonnegative integer Value that specifies the back region for model performance. If 'BACK'=n and the number of observations is T, then the first T - n observations are used to diagnose a series. The default is value 0.
- **'CRITERION'** takes a string Value that specifies the model selection criterion (statistic of fit) to be used to select from several candidate models. For a list of valid values, see the CRITERION= option in the HPFDIAGNOSE procedure in SAS Forecast Server Procedures: User’s Guide. The default is RMSE.
- **'HOLDOUT'** takes a nonnegative integer Value that specifies the holdout region for model selection. The holdout sample is a subset of dependent series (which you specify by using the TSM.SetY method) that ends at the last nonmissing observation. This option is relevant only when a TSM instance has been initialized via the Initialize method by using a model specification that requires model selection. Currently, this applies only to a model specification that is created by an ESMSPEC object and whose 'METHOD' option is set to either 'BEST', 'BESTN', or 'BESTS' via the ESMSPEC.SetOption method. The default value is 0.
- **'HORIZON'** takes a numeric Value that specifies the forecast horizon reference time. When set to a missing value, the forecast horizon reference time is automatically set as the first time period that follows the last nonmissing observation of the dependent series. The default value is a missing value.
- **'LEAD'** takes a nonnegative integer Value that specifies the forecast lead. The default value is the value specified in the PROC TSMODEL statement (LEAD= option). If LEAD= is not specified in the PROC TSMODEL statement, the default value of LEAD=0.

**TSM.SetY Method**

```plaintext
rc=obj.SetY (YSeries);
```

Specifies the dependent time series array (YSeries) for the TSM object.

**Input Arguments**

You must specify the following input argument:
YSeries specifies a numeric array that contains the dependent series for the TSM object.

CFC Object

The CFC object creates optimized combinations of time series model forecasts. The performance of combined model forecasts tends to improve over that of individual models. You can specify the model forecasts to be combined either as arbitrary time series arrays or as instances of the TSM object that have been previously run. Table 14.5 summarizes the methods that are associated with the CFC object.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AddModel</td>
<td>Add a forecast series from an instance of the TSM object</td>
</tr>
<tr>
<td>AddPredict</td>
<td>Add a time series array as a forecast series</td>
</tr>
<tr>
<td>criterion</td>
<td>Return the final fit statistic over a specified forecast region</td>
</tr>
<tr>
<td>GetForecast</td>
<td>Get the forecast series</td>
</tr>
<tr>
<td>Initialize</td>
<td>Initialize the CFC object</td>
</tr>
<tr>
<td>nfor</td>
<td>Return the forecast series length</td>
</tr>
<tr>
<td>Run</td>
<td>Run the CFC object to generate a combined forecast</td>
</tr>
<tr>
<td>SetOption</td>
<td>Specify the named option for the CFC object</td>
</tr>
<tr>
<td>SetY</td>
<td>Specify the dependent time series array (YSeries) for the CFC object</td>
</tr>
</tbody>
</table>

The basic execution pattern for using a CFC object follows this sequence of operations:

1 **Declare**: The object declaration statement creates a new instance of the CFC object.

2 **Initialize**: The CFC.Initialize method resets the object in preparation for a new run.

3 **SetY**: The CFC.SetY method defines the dependent time series for the CFC object.

4 **AddModel**: The CFC.AddModel method adds the forecast series that was produced by an instance of the TSM object. Each call adds one model forecast series to the combination. Repeat as needed for each model forecast series to be combined. Also, if you invoke this method prior to invoking the CFC.SetY method, then this method will use the dependent series of the TSM object as the dependent series of the CFC object.

5 **AddPredict**: The CFC.AddPredict method adds a forecast series to the combination that is specified as a numeric array. Each call adds one forecast series to the combination. Repeat as needed for each forecast series to be combined.
6 **SetOption**: The `CFC.SetOption` method specifies any options that affect the generation of the combined model forecast. Each call defines an option. Repeat as needed to specify all options that are required.

7 **Run**: The `CFC.Run` method uses its currently configured dependent and forecast series data to generate a combined forecast.

Figure 14.2 illustrates the data flow through the CFC object and its relationship with other components of the TSM package.
CFC Synopsis

DECLARE OBJECT obj (CFC) ;

Method syntax, in order of typical usage:

\[ rc = \text{obj}.\text{Initialize}() ; \]
\[ rc = \text{obj}.\text{SetY}(YSeries) ; \]
\[ rc = \text{obj}.\text{AddModel}(\text{ModelObj}, \text{ModelName}) ; \]
\[ rc = \text{obj}.\text{AddPredict}(\text{Predict}, \text{PredictSE}, \text{Name}, \text{Nparms}) ; \]
\[ rc = \text{obj}.\text{SetOption}(\text{Name}, \text{Name}, \text{Name}, \ldots) ; \]
\[ rc = \text{obj}.\text{Run}() ; \]
\[ rc = \text{obj}.\text{GetForecast}(\text{Which}, \text{Result}) ; \]
\[ \text{nfor} = \text{obj}.\text{nfor}() ; \]
\[ \text{criterion} = \text{obj}.\text{criterion}(\text{Region}) ; \]

CFC Methods

CFC.AddModel Method

\[ rc = \text{obj}.\text{AddModel}(\text{ModelObj}, \text{ModelName}) ; \]

Adds the forecast model series that were produced by an instance of the TSM object to the forecast combination process. This includes both the predicted and standard error series that were produced by the TSM object during the forecasting process. By default, the CFC object uses the name of the TSM object instance to identify the forecast component associated with its forecast series. However, you can optionally specify the forecast component via the \textit{ModelName} argument.

\textbf{Input Arguments}

You must specify the following input argument:

\textit{ModelObj} specifies the TSM object to use as a source of forecast model series.

You can also specify the following input argument:

\textit{ModelName} is a character string that specifies the name of the forecast component that is used to identify the model forecast series in the combination. The default is the name of the TSM object instance that is used as the source of forecast model series.

CFC.AddPredict Method

\[ rc = \text{obj}.\text{AddPredict}(\text{Predict}, \text{PredictSE}, \text{Name}, \text{Nparms}) ; \]

Adds a forecast series to be included in the forecast combination for the CFC object. This includes both the \textit{Predict} predicted series and the \textit{PredictSE} standard errors series as numeric arrays. You can optionally specify the name of the forecast component in the combination via the \textit{ModelName} argument. Each call to the AddPredict method adds the specified \textit{Predict} and \textit{PredictSE} series to the CFC object to be included in the combination process. The arrays that are used to specify the \textit{Predict} and \textit{PredictSE} series do not need to be unique, but the forecast component names that are specified in the \textit{ModelName} argument must be unique if you specify them. You can call the AddPredict method repeatedly to specify multiple forecast series.
**CFC Methods**

### Input Arguments

You must specify the following input arguments:

- **Predict** specifies a numeric array that contains the predicted series.
- **PredictSE** specifies a numeric array that contains the standard errors of the predicted series.

You can also specify the following input arguments:

- **ModelName** is a character string that specifies the name of the forecast component that is used to identify the model forecast series in the combination. The default is a unique name that is autogenerated by the CFC object.
- **Nparms** takes a numeric value that specifies the number of parameters used by the model that generated the forecast series. Valid values are integers greater than or equal to 0. The default value is 0.

### CFC.criterion Method

```c
 criterion = obj.criterion (Region) ;
```

Returns the fit statistic value over the specified forecast **Region** for the CFC object. The criterion is set via the ‘CRITERION’ argument in the CFC.SetOption method. A missing value indicates that the CFC object instance has not produced a successful forecast.

### Input Arguments

You must specify the following input argument:

- **Region** is a case-insensitive character string that specifies the forecast region over which the fit statistic is computed. You can specify one of the following values:
  - **BACK** returns the fit statistic over the time region that is subsequent to the FIT region and that did not contribute any data to the process of estimating combination weights for the specified model forecast series.
  - **FIT** returns the fit statistic over the time region that supplied observations for the process of estimating combination weights for the specified model forecast series.

### CFC.GetForecast Method

```c
 rc = obj.GetForecast (Which, Result) ;
```

Gets the specified forecast series (**Which**) from the CFC object and stores it in the specified numeric array (**Result**).

### Input Arguments

You must specify the following input argument:

- **Which** is a case-insensitive character string that specifies the type of forecast series to return. You can specify one of the following values:
Chapter 14: Time Series Model Package

- **ERROR** returns prediction errors.
- **LOWER** returns a lower confidence limit series.
- **STDERR** returns a prediction standard error series.
- **PREDICT** returns a prediction series.
- **UPPER** returns an upper confidence limit series.

**Output Arguments**
You must specify the following output argument:

**Result** specifies a numeric array to receive the forecast series.

**CFC.Initialize Method**

```plaintext
rc = obj.Initialize();
```

Initializes a CFC object by discarding a previous forecast and resetting the object’s configuration to the default values. This method must be called before any other attributes for the CFC object are specified.

**Arguments**
There are no arguments associated with this method.

**CFC.nfor Method**

```plaintext
nfor = obj.nfor();
```

Returns the length (observation count) of the forecast series for the CFC object. A missing value indicates that the CFC object has not produced a successful forecast.

**Arguments**
There are no arguments associated with this method.

**CFC.Run Method**

```plaintext
rc = obj.Run();
```

Runs the CFC object to perform the forecast combination process by using the dependent (YSeries) and candidate forecast series that have been specified for it. Upon successful completion, various results can be extracted from the CFC object.

**Arguments**
There are no arguments associated with this method.
CFC.SetOption Method

\[ rc = obj.SetOption('Name', Value <, 'Name', Value, ...>); \]

Specifies the named options for the CFC object. When you invoke this method, the previous forecast combination produced by the CFC.Run method is discarded. You must then rerun the CFC.Run method in order to produce a new forecast combination using the updated configuration options.

**Input Arguments**

You must specify at least one of the following 'Names' and its associated Value:

- **'BACK'** takes a nonnegative integer Value that specifies the back region for model performance. If 'BACK'=n and the number of observations is T, then the first \( T - n \) observations are used to diagnose a series. The default value is 0.

- **'CRITERION'** takes a string Value that specifies the fit statistic to be computed for the final forecast. The default is RMSE.

- **'ENCALPHA'** takes a numeric Value between 0 and 1, inclusive, that specifies the significance level of the encompassing test. The default value is 0.05.

- **'ENCTEST'** takes a string Value that specifies the encompassing test type. The encompassing test attempts to eliminate from consideration any forecasts that fail to add significant information to the final forecast. For a detailed description of the encompassing test, see the chapter “Forecast Combination Computational Details” in *SAS Forecast Server Procedures: User’s Guide*. You can specify the following Values:
  - **HLN** uses the Harvey-Leybourne-Newbold (HLN) test to estimate pairwise encompassing between candidate forecasts. Candidates are ranked from best to worst using the 'CRITERION' option. Iterating from best to worst, inferior candidates are tested together with the best of the untested candidates for retention in the combined set. The significance level for the test is given by the 'ENCALPHA' argument.
  - **NONE** performs no encompassing tests.
  - **OLS** uses an OLS-based regression test to estimate pairwise encompassing between candidate forecasts. Candidates are ranked from best to worst using the 'CRITERION' option. Iterating from best to worst, inferior candidates are tested together with the best of the untested candidates for retention in the combined set. The significance level for the test is given by the 'ENCALPHA' argument.

- **'HMISSPCT'** takes a numeric Value between 0 and 100 that specifies a threshold for the percentage of missing forecast values in the combination horizon. This threshold is used to exclude a candidate forecast from consideration in the final combination. By default, no horizon missing percentage test is performed on candidate forecasts. The forecast horizon is the region of time in which multistep forecasts are generated.

- **'LEAD'** takes a nonnegative integer Value that specifies the forecast lead. The default is the value that is specified in the PROC TSMODEL statement (LEAD= option). If the LEAD=
option is not specified in the PROC TSMODEL statement, the default value of LEAD=0 is used.

'MISSMODE' takes a string Value that specifies a method for treating missing values in the forecast combination. In a particular time slice across the combination ensemble, one or more combination contributors can have a missing value. This setting determines the treatment of those contributors in the final combination for such time indices. You can specify the following Values:

MISSING generates a missing combined forecast at each time index with one or more missing contributors.

RESCALE rescales the combination weights for the nonmissing contributors at each time index to sum to 1. You cannot specify RESCALE when the value of the 'WEIGHT' argument is OLS or NRLS.

The default value depends on the combination weight method that is specified by the 'WEIGHT' argument. RESCALE is the default for simple average, user-specified weights, ranked user weights, ranked weights, and RMSE weights. MISSING is the default for all other methods.

'MISSPCT' takes a numeric Value between 0 and 100 that specifies a threshold for the percentage of missing forecast values in the combination estimation region. This threshold is used to exclude a candidate forecast from consideration in the final combination. By default, no missing percentage test is performed on candidate forecasts.

'RANKING' takes a string Value that specifies the fit statistic to be used for ranking forecast candidates. This option is used in conjunction with the 'ENCTEST' and 'WEIGHT' (when Value is set to RANKWGT) arguments. For a list of valid values, see the CRITERION= option in the HPFDIAGNOSE procedure in SAS Forecast Server Procedures: User’s Guide. The default is RMSE.

'SEMODE' takes a string Value that specifies the method for computing the prediction error variance series. This series is used to compute the prediction standard error, which in turn is used to compute confidence bands on the combined forecast. You can specify the following Values:

DIAG computes the prediction error variance by assuming that the forecast errors at time $t$ are uncorrelated so that the simple diagonal form of $\Sigma_t$ is used.

ESTCORR computes the prediction error variance by using estimates of $\rho_{i,j,t}$, the sample cross-correlation between $e_{i,t}$ and $e_{j,t}$ over the time span $t = 1, \ldots, T$, where $t$ denotes the last time index of the actual series $y_t$. This Value implies that the error series $e_{i,t}$ and $e_{j,t}$ are assumed to be jointly stationary.

The default is DIAG.

'USERWEIGHTS' takes a numeric array Value that specifies the combination weights to be used when the value of the 'WEIGHT' argument is RANKWGT or USERDEF. If the specified weights do not sum to 1, then the weights are automatically normalized to sum to 1 and a warning message is issued. For more information, see the 'WEIGHT' argument.
'WEIGHT' takes a string Value that specifies the method for determining the combination weights that are used in the weighted average of the candidate model forecasts. You can specify the following Values:

AICC computes the combination weights on the basis of corrected AIC weights. By default, all AICC scored candidate forecasts are combined.

AVERAGE computes the simple average of the forecasts that are selected for combination.

ERLS computes the combination weights on the basis of a constrained least squares problem to minimize the $\ell_2$ norm of the combined forecast residuals subject to the constraint that the weights sum to 1.

LAD computes the weights on the basis of a least absolute deviations (LAD) measure of fit for the combined forecast. A linear program is formulated in which an objective function to be minimized is expressed in terms of the absolute values of a loss series subject to constraints that the weights sum to 1 and be nonnegative.

NERLS computes the combination weights on the basis of a constrained least squares problem to minimize the $\ell_2$ norm of the combined forecast residuals subject to the constraints that the weights sum to 1 and be nonnegative.

NRLS is equivalent to NERLS, except that the resulting combination weights are not constrained to summing to 1.

OLS computes the combination weights that result from the ordinary least squares problem to minimize the $\ell_2$ norm of the combined forecast residuals.

RANKWGT assigns weights by using the rank of the candidate forecasts at the time the combination is performed. You can optionally specify the weights in the 'USERWEIGHTS' argument, where the number of specified values must match the number of model forecasts that are specified in the current instance of the CFC object. The weights are assigned by ranking the candidate forecasts from best to worst on the basis of the fit statistic specified in the 'RANKING' argument. The best candidate uses the first weight, $W_1$, and so on. If you do not specify the weights in the 'USERWEIGHTS' argument, then the weight of each candidate forecast is set to the inverse value of its rank. The set of weights that is used is normalized to account for candidate forecasts that are omitted from the final combination.

RMSEWGT computes the combination weights on the basis of the RMSE statistic of fit for the forecast contributors. The weights are normalized to sum to 1.

USERDEF assigns weights by using the list of user-specified values. You must specify the weights in the 'USERWEIGHTS' argument, where the number of specified values must match the number of model forecasts that are specified in the current instance of the CFC object. The weights correspond to the order of specification of the model forecasts. The
set of weights that is used is normalized to account for candidates that are omitted from the final combination.

The default is AVERAGE.

CFC.SetY Method

\[ rc = \text{obj.SetY}(YSeries); \]

Specifies the dependent time series array (YSeries) for the CFC object.

Input Arguments
You must specify the following input argument:

YSeries specifies a numeric array that contains the dependent series for the CFC object.

ESMSPEC Object

The ESMSPEC object defines an exponential smoothing model specification for use with the TSM object. The basic execution pattern for defining an ESMSPEC follows this sequence of operations:

1 Declare: The object declaration statement creates a new ESMSPEC object. By default, it is an ESMSPEC with METHOD=BEST.

2 Open: The ESMSPEC.Open method initializes the default ESMSPEC object for a new configuration.

3 Configure: The various ESMSPEC.Set methods configure the ESMSPEC object.

4 Close: The ESMSPEC.Close method finalizes the ESMSPEC object.

5 Apply: Add the ESMSPEC to a TSM object using the TSM.Initialize method.

Table 14.6 summarizes the methods that are associated with the ESMSPEC object.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Close</td>
<td>Close ESM model specification</td>
</tr>
<tr>
<td>Open</td>
<td>Open ESM model specification</td>
</tr>
<tr>
<td>SetOption</td>
<td>Set option for ESM model</td>
</tr>
<tr>
<td>SetParm</td>
<td>Set parameters for ESM model</td>
</tr>
<tr>
<td>SetTransform</td>
<td>Set transform for ESM model</td>
</tr>
</tbody>
</table>
You can also store the XML representation of the ESMSPEC object in a CAS table. For more information, see the TSMSPEC object.

Figure 14.3 illustrates the data flow through the ESMSPEC object.

**Figure 14.3** ESMSPEC Object Data Flow

---

**ESMSPEC Synopsis**

```
DECLARE OBJECT obj (ESMSPEC) ;
```

Method syntax, in order of typical usage:

```c
rc=obj.Open () ;
rc=obj.SetTransform ('Type', 'Option', Parm) ;
rc=obj.SetParm ('CompName', Parm, <LRest,URest>) ;
rc=obj.SetOption ('Name', Value <;'Name', Value, >) ;
rc=obj.Close () ;
```
ESMSPEC Methods

ESMSPEC.Close Method

\[ rc = \text{obj.Close}() ; \]

Finalizes the ESMSPEC object to prepare the ESM model to be used in a TSM object or to be imported to a TSMSPEC object for printing or for storage in a model repository catalog.

**Arguments**
There are no arguments associated with this method.

ESMSPEC.Open Method

\[ rc = \text{obj.Open}() ; \]

Opens the ESMSPEC object.

**Arguments**
There are no arguments associated with this method.

ESMSPEC.SetOption Method

\[ rc = \text{obj.SetOption('Name', Value \ldots)} ; \]

Specifies ESM options.

**Input Arguments**
You must specify at least one of the following 'Names' and its associated Value:

- `'CRITERION'` takes a string Value that specifies the model selection criterion (statistic of fit) to be used to select from several candidate models. For a list of valid values, see the CRITERION= option in the HPDIAGNOSE procedure in *SAS Forecast Server Procedures: User’s Guide*. The default is RMSE.

- `'METHOD'` takes a string Value that specifies the ESM method. You can specify the following values:
  - `ADDWINTERS` requests the Winters additive method.
  - `BEST` requests the best candidate smoothing model among the SIMPLE, LINEAR, DAMPTREND, SEASONAL, ADDWINTERS, or WINTERS methods.
  - `BESTN` requests the best candidate nonseasonal smoothing model among the SIMPLE, LINEAR, or DAMPTREND methods.
  - `BESTS` requests the best candidate seasonal smoothing model among the SEASONAL, ADDWINTERS, or WINTERS methods.
  - `DAMPTREND` requests damped trend exponential smoothing.
  - `DOUBLE` requests second order exponential smoothing.
  - `LINEAR` requests linear (Holt) exponential smoothing.
MULT SEASONAL requests multiplicative seasonal exponential smoothing.
SEASONAL requests additive seasonal exponential smoothing.
SIMPLE requests simple (single) exponential smoothing.
WINTERS requests Winters multiplicative method.

The default is BEST.

'NOEST' takes a Boolean Value that indicates whether the smoothing model parameters are fixed values. By default, the smoothing model parameters are optimized. 'NOEST' requires all of the exponential smoothing model parameters to be explicitly specified via calls to the ESMSPEC.SetParm method. This argument is ignored if any of the model parameters is not specified.

ESMSPEC.SetParm Method

rc = obj.SetParm ('CompName', Parm, <LRest, URest>);

Specifies parameter values and restrictions for the specified ESM model component. Optional bounds, LRest and URest, can be specified to restrict the weight value for the specified ESM component during parameter optimization. Parameter values and restrictions must be in the range from –1 to 2.

Input Arguments
You must specify the following input arguments:

'CompName' specifies a character variable that indicates the ESM component context. You can specify the following values:

DAMP specifies that Parm is the initial value of a damping weight parameter.
LEVEL specifies that Parm is the initial value of a level weight parameter.
SEASON specifies that Parm is the initial value of a season weight parameter.
TREND specifies that Parm is the initial value of a trend weight parameter.

By default, if the initial value of a parameter is not specified, then the optimizer uses a grid search to find an appropriate initial value.

Parm takes a numeric value that specifies the smoothing component weight.

You can also specify the following input arguments:

LRest is a numeric variable that specifies a lower bound on the smoothing weight.
URest is a numeric variable that specifies an upper bound on the smoothing weight.
ESMSPEC.SetTransform Method

rc = obj.SetTransform (‘Type’ < , ’Option’, Parm >) ;

Specifies the functional transform, ‘Type’, to be used by the ESM model. Optional arguments Option and Parm offer greater control over the transform.

Input Arguments

You must specify the following input argument:

‘Type’ takes one of the following string values:

- **AUTO** automatically chooses between NONE and LOG based on model selection criteria.
- **BOXCOX(value)** requests Box-Cox transformation with a parameter value between –5 and 5. The default is BOXCOX(1).
- **LOG** requests logarithmic transformation.
- **LOGIT** requests logistic transformation.
- **NONE** does not apply a transformation.
- **SQRT** requests square-root transformation.

The default is NONE.

You can also specify the following input arguments:

‘Option’ takes one of the following string values that specifies prediction semantics for the inverse transform:

- **MEAN** requests that the inverse transform produce mean forecasts.
- **MEDIAN** requests that the inverse transform produce median forecasts.

The default is MEAN.

Parm takes a numeric value between –5 and 5 that specifies a control parameter. This parameter is allowed only for Box-Cox transforms.
**ARIMASPEC Object**

The ARIMASPEC object generates autoregressive integrated moving average (ARIMA) model specifications for use with a TSM object.

Table 14.7 summarizes the methods that are associated with the TSM.ARIMASPEC method.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AddARPoly</td>
<td>Add an autoregressive polynomial factor to the ARIMA model</td>
</tr>
<tr>
<td>AddMAPoly</td>
<td>Add a moving average polynomial factor to the ARIMA model</td>
</tr>
<tr>
<td>AddTF</td>
<td>Add a transfer function to the ARIMA model</td>
</tr>
<tr>
<td>AddTFDenPoly</td>
<td>Add a transfer function denominator factors to the ARIMA model</td>
</tr>
<tr>
<td>AddTFNumPoly</td>
<td>Add a transfer function numerator factors to the ARIMA model</td>
</tr>
<tr>
<td>Close</td>
<td>Close the ARIMA model specification</td>
</tr>
<tr>
<td>Open</td>
<td>Open the ARIMA model specification</td>
</tr>
<tr>
<td>SetDiff</td>
<td>Add differencing to the ARIMA model</td>
</tr>
<tr>
<td>SetOption</td>
<td>Specify the ARIMA option</td>
</tr>
<tr>
<td>SetTFTransform</td>
<td>Specify the transfer function transform</td>
</tr>
<tr>
<td>SetTransform</td>
<td>Specify transform</td>
</tr>
</tbody>
</table>

The basic execution pattern for using an ARIMASPEC object follows this sequence of operations:

1. **Declare**: The object declaration statement creates a new ARIMASPEC object. At creation, its default state is ARIMA(0,1,0) with intercept (random walk with drift).
2. **Open**: The ARIMASPEC.Open method readies the default ARIMASPEC object for new configuration.
3. **Configure**: The various ARIMASPEC.Set methods configure the ARIMASPEC object.
4. **Close**: The ARIMASPEC.Close method finalizes the ARIMASPEC object.
5. **Apply**: The TSM.Initialize method adds the ARIMASPEC to a TSM object.

You can also store the XML representation of the ARIMA model in a CAS table. For more information, see the TSMSPEC object. Figure 14.4 illustrates the data flow through the ARIMASPEC object.
You can include series transforms such as log or Box-Cox in the specification. Deferred seasonality wildcards for seasonal ARIMA polynomial factors and seasonal differencing lags are also supported. The following list summarizes where these features can be used. For more information, see the method descriptions.

- **ARIMASPEC.SetDiff** supports a seasonal wildcard lag in the DiffArray.

- **ARIMASPEC.AddARPoly** and **ARIMASPEC.AddMAPoly** support addition of seasonal ARIMA polynomial factors.

- **ARIMASPEC.AddTF** supports a seasonal wildcard lag in the DiffArray.

- **ARIMASPEC.AddTFNumPoly** and **ARIMASPEC.AddTFDenPoly** support addition of seasonal ARIMA polynomial factors for the transfer function numerator and denominator, respectively.

- **ARIMASPEC.SetTransform** applies transforms such as log, square root, logistic, or Box-Cox to the model.
When you specify ARIMA model coefficient values for any model components, you must specify them for all model components without regard to the value of the NOEST argument in the ARIMASPEC.SetOption method. Failing to specify a complete set of ARIMA model parameters results in an error when you call ARIMASPEC.Close, and the accumulated ARIMA model specification is reset.

ARIMASPEC Synopsis

DECLARE OBJECT obj (ARIMASPEC);

Method syntax, in order of typical usage:

\[ rc = obj.Open(); \]
\[ rc = obj.SetTransform ('Type' < 'Option', Parm>); \]
\[ rc = obj.SetDiff (DiffArray < NDiff>); \]
\[ rc = obj.AddARPoly (OrderArray < NOrder, Seasonal, CoeffArray>); \]
\[ rc = obj.AddMAPoly (OrderArray < NOrder, Seasonal, CoeffArray>); \]
\[ rc = obj.AddTF (XName, < Delay, DifArray, NDiff>); \]
\[ rc = obj.AddTFNumPoly (XName, NumArray < NNum, Seasonal, CoeffArray>); \]
\[ rc = obj.AddTFDenPoly (XName, DenArray < NDen, Seasonal, CoeffArray>); \]
\[ rc = obj.SetTFTransform (XName, 'Type', < Parm>); \]
\[ rc = obj.SetOption ('Name', Value < 'Name', Value, ...>); \]
\[ rc = obj.Close(); \]

ARIMASPEC Methods

ARIMASPEC.AddARPoly Method

\[ rc = obj.AddARPoly (OrderArray < NOrder, Seasonal, CoeffArray>); \]

Adds an autoregressive (AR) polynomial factor to ARIMA model. Additional AR polynomial factors can be added to the ARIMA model with subsequent calls to this method.

Input Arguments

You must specify the following input argument:

**OrderArray** is a numeric array that specifies AR polynomial lags. Valid values are integers greater than or equal to 1.

You can also specify the following input arguments:

**NOrder** takes a numeric variable that specifies the number of **OrderArray** values to use. By default, all **OrderArray** values are used.

**Seasonal** takes a Boolean value (0 or 1) that, when set to 1, specifies that the AR polynomial is seasonal. By default, the AR polynomial is not seasonal and the lags of the AR polynomial are simple.

**CoeffArray** is a numeric array that specifies the AR polynomial coefficients. If specified, this array must be of the same cardinality as the **OrderArray**.
ARIMASPEC.AddMAPoly Method

\[
rc = \text{obj}.\text{AddMAPoly} (\text{OrderArray} <,\text{NOrder},\text{Seasonal},\text{CoeffArray}>) ;
\]

Adds a moving average (MA) polynomial factor to ARIMA model. More MA polynomial factors can be added to the ARIMA model by subsequent calls to this method.

**Input Arguments**

You must specify the following input argument:

**OrderArray** is a numeric array that specifies MA polynomial lags. Valid values are integers greater than or equal to 1.

You can also specify the following input arguments:

**NOrder** takes a numeric variable that specifies the number of \text{OrderArray} values to use. By default, all \text{OrderArray} values are used.

**Seasonal** takes a Boolean value (0 or 1) that, when set to 1, specifies that the MA polynomial is seasonal. By default, the MA polynomial is not seasonal and the lags of the MA polynomial are simple.

**CoeffArray** is a numeric array that specifies the MA polynomial coefficients. If specified, this array must be of the same cardinality as the \text{OrderArray}.

ARIMASPEC.AddTF Method

\[
rc = \text{obj}.\text{AddTF} (\text{XName}, <\text{Delay},\text{DifArray},\text{NDiff}>) ;
\]

Adds a transfer function to the ARIMA model for the specified \text{XName} variable. This method adds the variable as a simple scale effect subject to any specified lag and differencing that might be applied.

**Input Arguments**

You must specify the following input argument:

**XName** is a character string that specifies the name of the X variable.

You can also specify the following input arguments:

**Delay** takes a numeric value that specifies the simple delay for the predictor. The default value is 0.

**DifArray** is a numeric array that specifies differencing orders for the X variable. Valid values are integers greater than or equal to 1. If not specified, then no differencing is applied. Values in \text{DifArray} are interpreted as follows:

- Negative values are not allowed and result in an error condition.
- Nonnegative values represent differencing orders.
- An .S missing value is interpreted to include seasonal difference order.
- Any other missing value is ignored.

**NDiff** takes a numeric value that specifies the number of \text{DifArray} values to use. If not specified, then all the values in \text{DifArray} are used.
ARIMASPEC.AddTFDenPoly Method

```plaintext
rc = obj.AddTFDenPoly (XName, DenArray <, NDen, Seasonal, CoeffArray>) ;
```

Adds a transfer function denominator polynomial factor for the specified `XName` variable. More polynomials can be added by subsequent calls to the ARIMASPEC.AddTFNumPoly method.

**Input Arguments**
You must specify the following input argument:

- **XName** is a character string that specifies the name of the X variable.
- **DenArray** is a numeric array that specifies denominator polynomial lags for the X variable. Valid values are integers greater than 0.

You can also specify the following input arguments:

- **NDen** takes a numeric value that specifies the number of `DenArray` values to use. Valid values are integers greater than 0.
- **Seasonal** takes a Boolean value (0 or 1) that, when set to 1, specifies that the denominator polynomial is seasonal. By default, the denominator polynomial is not seasonal and the lags of the denominator polynomial are simple.
- **CoeffArray** is a numeric array that specifies denominator polynomial coefficients.

ARIMASPEC.AddTFNumPoly Method

```plaintext
rc = obj.AddTFNumPoly (XName, NumArray <, NNum, Seasonal, CoeffArray>) ;
```

Adds a transfer function numerator polynomial factor for the specified `XName` variable. More polynomials can be added by subsequent calls to the ARIMASPEC.AddTFNumPoly method.

**Input Arguments**
You must specify the following input argument:

- **XName** is a character string that specifies the name of the X variable.
- **NumArray** is a numeric array that specifies numerator polynomial lags for the X variable. Valid values are integers greater than 0.

You can also specify the following input arguments:

- **NNum** takes a numeric value that specifies the number of `NumArray` values to use. By default, all `NumArray` values are used.
- **Seasonal** takes a Boolean value (0 or 1) that, when set to 1, specifies that the numerator polynomial is seasonal. By default, the numerator polynomial is not seasonal and the lags of the numerator polynomial are simple.
- **CoeffArray** is a numeric array that specifies numerator polynomial coefficients.
ARIMASPEC.Close Method

\[ rc = \text{obj.Close}(); \]

Finalizes the ARIMASPEC object to prepare the ARIMA model to be used in a TSM object or to be imported to a TSMSPEC object for printing or for storage in a model repository.

**Arguments**
There are no arguments associated with this method.

ARIMASPEC.Open Method

\[ rc = \text{obj.Open}(); \]

Initializes an empty ARIMASPEC object for configuration.

**Arguments**
There are no arguments associated with this method.

ARIMASPEC.SetDiff Method

\[ rc = \text{obj.SetDiff}(\text{DiffArray}, \text{NDiff}); \]

Adds differencing to the ARIMA model.

**Input Arguments**
You must specify the following input argument:

- \( \text{DiffArray} \)
  is a numeric array that specifies differencing orders, where each order must be an integer greater than or equal to 1. Values in \( \text{DiffArray} \) are interpreted as follows:
  - Negative values are not allowed and result in an error condition.
  - Nonnegative integer values represent differencing orders.
  - An \( .S \) missing value is interpreted to include seasonal difference order.
  - Any other missing value is ignored.

You can also specify the following input argument:

- \( \text{NDiff} \)
  takes a numeric value that specifies the number of \( \text{DiffArray} \) values to use. The default is to use all elements of \( \text{DiffArray} \).

ARIMASPEC.SetOption Method

\[ rc = \text{obj.SetOption}('Name', Value, 'Name', Value, ...); \]

Specifies ARIMA model options. Options are \( ('Name', Value) \) pairs where \( 'Name' \) is a case-insensitive character string and \( Value \) depends on the \( 'Name' \).
**Input Arguments**

You must specify at least one of the following *Names* and its associated *Value*:

- **'CONVERGE'**
  takes a numeric *Value* between 0 and 1, exclusive, that specifies the convergence criterion. Convergence is assumed when the largest change in the estimate for any parameter is less than the specified *Value*. If the absolute value of the parameter estimate is greater than 0.01, the relative change is used; otherwise, the absolute change in the estimate is used. The default value is 0.001.

- **'DELTA'**
  takes a numeric *Value* between 0 and 1, exclusive, that specifies the perturbation value for computing numerical derivatives. The default value is 0.001.

- **'MAXITER'**
  takes a positive integer *Value* that specifies the maximum number of iterations allowed. The default is 50.

- **'METHOD'**
  takes a string *Value* that specifies the estimation method to use. You can specify one of the following *Values*:

  - **CLS** specifies the conditional least squares method.
  - **ML** specifies the maximum likelihood method.
  - **ULS** specifies the unconditional least squares method.

  The default is CLS.

- **'MU'**
  takes a numeric *Value* that specifies a constant term for the ARIMA model. The default is 0.

- **'NOEST'**
  takes a Boolean *Value* (0 or 1) that, when set to 1, specifies that no estimation is performed. The default value is 0 (estimation is performed).

- **'NOINT'**
  takes a Boolean *Value* (0 or 1) that, when set to 1, specifies that no intercept is defined. This suppresses the fitting of a constant (or intercept) parameter in the model. That is, the value specified with MU value is omitted. The default value is 0 (intercept is defined).

- **'NOSTABLE'**
  takes a Boolean *Value* (0 or 1) that, when set to 1, requests that the autoregressive and moving average parameter estimates for the noise part of the model not be restricted to the stationary and invertible regions, respectively. The default is 0 (parameter estimates for the noise part of the model are restricted).

- **'SINGULAR'**
  takes a numeric *Value* between 0 and 1, exclusive, that specifies the criterion for checking singularity. If a pivot of a sweep operation is less than *Value*, the matrix is deemed singular. Sweep operations are performed on the Jacobian matrix during final estimation and on the covariance matrix when preliminary estimates are obtained. The default is 1E–7.

**ARIMASPEC.SetTFTransform Method**

```
rc = obj.SetTFTransform (XName, 'Type', <Parm>) ;
```

Specifies a functional transform for specified *XName* variable.
**Input Arguments**

You must specify the following input arguments:

- **XName** is a character string that specifies the name of the X variable.
- **'Type'** takes a string value that specifies the functional transform to use. You can specify the following values:
  - **BOXCOX(value)** requests Box-Cox transformation with a parameter `value` between –5 and 5. The default is `BOXCOX(1)`.
  - **LOG** requests logarithmic transformation.
  - **LOGIT** requests logistic transformation.
  - **NONE** does not apply a transformation.
  - **SQRT** specifies square-root transformation.

The default is **NONE**.

You can also specify the following input argument:

- **Parm** takes a numeric value between –5 and 5 that specifies a control argument. This parameter is allowed only for Box-Cox transforms.

**ARIMASPEC.SetTransform Method**

```
rc=obj.SetTransform ('Type', 'Option', Parm);
```

Specifies the functional transform `Type` to be used by the ARIMA model. Optional arguments `Option` and `Parm` offer greater control over the transform.

**Input Arguments**

You must specify the following input argument:

- **'Type'** takes a string value that specifies the functional transform to use. You can specify the following values:
  - **BOXCOX(value)** requests Box-Cox transformation with a parameter `value` between –5 and 5. The default is `BOXCOX(1)`.
  - **LOG** requests logarithmic transformation.
  - **LOGIT** requests logistic transformation.
  - **NONE** does not apply a transformation.
  - **SQRT** specifies square-root transformation.

The default is **NONE**.

You can also specify the following input arguments:
'Option' takes a string value that specifies prediction semantics for the inverse transform. You can specify the following values:

- **MEAN**: requests that the inverse transform produce mean forecasts.
- **MEDIAN**: requests that the inverse transform produce median forecasts.

The default is MEAN.

*Parm* takes a numeric value between –5 and 5 that specifies a control parameter. This parameter is allowed only for Box-Cox transforms.

---

**EXMSPEC Object**

The EXMSPEC object generates an external model (EXM) specification for use with the TSM object, or with the automatic time series modeling (ATSM) package for automatic forecasting services.

Table 14.8 summarizes the methods that are associated with the EXMSPEC object.

**Table 14.8  Methods of the EXMSPEC Object**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Open</td>
<td>Open EXM model specification</td>
</tr>
<tr>
<td>Close</td>
<td>Close EXM model specification</td>
</tr>
<tr>
<td>SetTransform</td>
<td>Set transform for EXM model specification</td>
</tr>
<tr>
<td>SetOption</td>
<td>Set options for EXM model specification</td>
</tr>
</tbody>
</table>

Figure 14.5 illustrates the data flow through the EXMSPEC object.
The basic execution pattern for using an EXMSPEC object follows this sequence of operations:

1 **Declare**: The object declaration statement creates a new EXMSPEC object.

2 **Open**: The EXMSPEC.Open method readies the default EXMSPEC object for new configuration.

3 **Configure**: The various EXMSPEC.Set methods configure the EXMSPEC object.

4 **Close**: The EXMSPEC.Close method finalizes the EXMSPEC object.

5 **Apply**: The EXMSPEC is applied to a TSM object using the TSM.Initialize method or is used with the ATSM package. For more information about using the EXMSPEC object, see the section “TSM Object” on page 495 or Chapter 3, “Automatic Time Series Modeling Package.”

**Note:**

1. You can also store the XML representation of the EXMSPEC to a CAS table. For more information, see the section “TSMSPEC Object” on page 542.

2. To run the EXMSPEC in a TSM object, time series variables for the forecast series roles must be added to the TSM object via the TSM.AddExternal method. For more information, see the section “TSM.AddExternal Method” on page 497.

3. To run the EXMSPEC in an ATSM:FORENG object, time series variables for the forecast series roles must be added to the ATSM:TSDF object that is used to specify the time series data for the
ATSM:FORENG object. When the EXMSPEC object is used with the ATSM package, more sophisticated persistence mechanisms are available. For more information, see Chapter 3, “Automatic Time Series Modeling Package.”

---

**EXMSPEC Synopsis**

```plaintext
DECLARE OBJECT obj (EXMSPEC) ;
```

Method syntax, in order of typical usage:

```plaintext
rc=obj.Open () ;
rc=obj.SetTransform ("Type" <,Option, Parm>) ;
rc=obj.SetOption ("Name", Value < , "Name", Value,... >) ;
rc=obj.Close () ;
```

---

**EXMSPEC Methods**

**EXMSPEC.Close Method**

```plaintext
rc=obj.Close () ;
```

Finalizes the EXM model in the EXMSPEC object. This prepares the EXM model for use in a TSM object or to be imported to a TSMSPEC object for printing or storage to a model repository catalog.

**Arguments**

There are no arguments associated with this method.

**EXMSPEC.Open Method**

```plaintext
rc=obj.Open () ;
```

Initializes an EXMSPEC object for configuration.

**Arguments**

There are no arguments associated with this method.

**EXMSPEC.SetOption Method**

```plaintext
rc=obj.SetOption ("Name", Value < , "Name", Value,... >) ;
```

Specifies the options for the EXMSPEC object.

**Input Arguments**

You must specify at least one of the following ‘Names’ and its associated Value:

- **'ERROR'** takes a string Value that specifies the variable name for ERROR series.
- **'LOWER'** takes a string Value that specifies the variable name for LOWER series.
- **'METHOD'** takes a string Value that specifies the method to approximate prediction STDERR series.

You can specify one of the following values:
ACF        Autocorrelation is used.
ERRORACF  Prediction error autocorrelation is used.
NONE      No prediction error autocorrelation is used.
PERFECT   Perfect autocorrelation is assumed.
WN        Prediction error autocorrelation is white noise.

The default is PERFECT.

'NLAGPCT'  takes a numeric Value between 0 and 100 that specifies the percentage of error series count for ACF computations. The default is 25 (25%).

'NPARMS'   takes a nonnegative integer Value that specifies the number of parameters for the external forecast. The default is 0.

'PREDICT'  takes a string Value that specifies the variable name for PREDICT series.

'SIGMA'    takes a numeric Value that specifies the prediction standard error for the external model. If left unspecified, then the prediction mean square error is computed from the prediction errors by using the 'NPARMS' argument. The default is missing value.

'STDERR'   takes a string Value that specifies the variable name for STDERR series.

'UPPER'    takes a string Value that specifies the variable name for UPPER series.

EXMSPEC.SetTransform Method

rc=obj.SetTransform ('Type' <,,Option,Parm>);

Specifies the functional transform to be used by the EXM model.

*Input Arguments*

You must specify the following input argument:

'Type'  takes one of the following string values:

BOXCOX(value)  requests Box-Cox transformation with a parameter value between –5 and 5. The default is BOXCOX(1).

LOG        requests logarithmic transformation.

LOGIT      requests logistic transformation.

NONE       does not apply a transformation.

SQRT       requests square-root transformation.

The default is NONE.

You can also specify the following input arguments:

'Option'   takes one of the following string values that specifies prediction semantics for the inverse transform:
**MEAN** requests that the inverse transform produce mean forecasts.
**MEDIAN** requests that the inverse transform produce median forecasts.

The default is MEAN.

*Parm* takes a numeric value between –5 and 5 that specifies a control parameter. This parameter is allowed only for Box-Cox transforms.

---

**IDMSPEC Object**

The IDMSPEC object generates intermittent demand models for use with the TSM object. You open the IDMSPEC object to begin defining a new IDM model. You call the IDMSPEC methods to configure the object with the desired settings to define the model characteristics of interest, and then you close the IDMSPEC object to ready it for use in a TSM object. For more information, see the section “TSM.Initialize Method” on page 500 method. You can also store the XML representation of the IDM to a CAS table. For more information, see the section “TSMSPEC Object” on page 542.

Table 14.9 the methods that are associated with the IDMSPEC object.

**Table 14.9** Methods of the IDMSPEC Object

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Close</td>
<td>Close the IDM model specification</td>
</tr>
<tr>
<td>Open</td>
<td>Open the IDM model specification</td>
</tr>
<tr>
<td>SetMethod</td>
<td>Set component smoothing method</td>
</tr>
<tr>
<td>SetOption</td>
<td>Set options for IDM model specification</td>
</tr>
<tr>
<td>SetParm</td>
<td>Set component parameters</td>
</tr>
<tr>
<td>SetTransform</td>
<td>Set component transform</td>
</tr>
</tbody>
</table>

Figure 14.6 illustrates the data flow through the IDMSPEC object.
**IDMSPEC Synopsis**

DECLARE OBJECT obj (IDMSPEC) ;

Method syntax, in order of typical usage:

```
rc=obj.Open () ;
rc=obj.SetOption (‘Name’, Value < ‘Name’,Value,... >) ;
rc=obj.SetTransform (‘IDMComp’, ‘Type’ < ‘Option’,Parm>) ;
rc=obj.SetMethod (‘IDMComp’, ‘Method’ ) ;
rc=obj.SetParm (‘IDMComp’, ‘ESMComp’,Parm < ,Noest,LRest, URest >) ;
rc=obj.Close () ;
```

**IDMSPEC Methods**

**IDMSPEC.Close Method**

```
rc=obj.Close () ;
```

Finalizes the IDM model in the IDMSPEC object and prepares the IDM model for use in a TSM object or to be imported to a TSMSPEC object for printing or storage to a model repository catalog.

**Arguments**

There are no arguments associated with this method.
**IDMSPEC.Open Method**

\[ rc=\text{obj}.\text{Open}(); \]

Opens the IDMSPEC object for configuration and initializes it with default options. The default options specify that the best IDM model from among the average demand and Croston’s model be selected based on the RMSE criterion (see the value BEST for the 'MODEL' argument in the SetOption method). The smoothing model for the individual components of both models is selected from among the best candidate nonseasonal smoothing method (see value BESTN for the 'Method' argument in the SetMethod method). In addition, by default, all component transformations are automatically chosen between NONE and LOG based on the model selection criterion (see the value AUTO for the 'Type' argument in the SetTransform method).

**Arguments**

There are no arguments associated with this method.

**IDMSPEC.SetMethod Method**

\[ rc=\text{obj}.\text{SetMethod}('IDMComp', 'Method'); \]

Sets a smoothing method for IDM model component.

**Input Arguments**

You must specify the following input arguments:

- **'IDMComp'** is a character string that specifies the IDM component model. You can specify one of the following values:
  - AVERAGE specifies the average demand model.
  - INTERVAL specifies demand interval model.
  - SIZE specifies demand size model.

- **'Method'** takes a string value that specifies the name of the smoothing method. You can specify one of the following values:
  - BESTN requests the best candidate nonseasonal smoothing model among the SIMPLE, LINEAR, or DAMPTREND methods.
  - DAMPTREND requests damped trend exponential smoothing.
  - LINEAR requests linear (Holt) exponential smoothing.
  - NONE disables the use of a smoothing model for the specified component.
  - SIMPLE requests simple (single) exponential smoothing.

The default is BESTN.
Chapter 14: Time Series Model Package

### IDMSPEC.SetOption Method

```plaintext
rc = obj.SetOption ('Name', Value <, 'Name', Value, ...>);
```

Specifies the options for the IDMSPEC object.

**Input Arguments**

You must specify at least one of the following `Names` and its associated `Value`:

- **'BASE'**
  - Takes a numeric `Value` that specifies the base demand level. A 'BASE' value that is specified as a missing numeric value is interpreted to infer the base demand automatically as the median value of the dependent series.

- **'CRITERION'**
  - Takes a string `Value` that specifies the model selection criterion (statistic of fit) to be used to select from several candidate models. For a list of valid values, see the CRITERION= option in the HPFDIAGNOSE procedure in *SAS Forecast Server Procedures: User’s Guide*. The default is RMSE.

- **'MODEL'**
  - Is a character string that specifies a mnemonic for the IDM model to use. You can specify the following `Values`:
    - **AVERAGE**
      - Uses the single smoothing model to fit the average demand component.
    - **BEST**
      - Requests the best IDM model from among the AVERAGE and CROSTON models.
    - **CROSTON**
      - Uses the two smoothing models to fit the demand interval component and the demand size component.

  The default is BEST.

### IDMSPEC.SetParm Method

```plaintext
rc = obj.SetParm ('IDMComp', 'ESMComp', Parm <, Noest, LRest, URest>);
```

Sets parameter value and restrictions for the ESM component in the IDM component model. The ESM method-specific bounds are employed to limit or filter the smoothing method weight values.

**Input Arguments**

You must specify the following input arguments:

- **'IDMComp'**
  - Is a character string that specifies the IDM component model. You can specify one of the following values:
    - **AVERAGE**
      - Specifies the average demand model.
    - **INTERVAL**
      - Specifies demand interval model.
    - **SIZE**
      - Specifies demand size model.

- **'ESMComp'**
  - Is a character string that specifies the ESM component of the IDM component. You can specify one of the following values:
DAMP specifies that Parm is the initial value of a damping weight parameter.
LEVEL specifies that Parm is the initial value of a level weight parameter.
SEASON specifies that Parm is the initial value of a season weight parameter.
TREND specifies that Parm is the initial value of a trend weight parameter.

Parm takes a numeric value that specifies the smoothing component weight.

You can also specify the following input arguments:

Noest takes a Boolean value (0 or 1) that, when set to 1, specifies that the Parm argument is fixed. The default value is 0 (initial).
LRest takes a numeric value between –1 and 2 that specifies a lower bound restriction on the ESM weight.
URest takes a numeric value between –1 and 2 that specifies an upper bound restriction on the ESM weight.

**IDMSPEC.SetTransform Method**

rc = obj.SetTransform ('IDMComp', 'Type < Option', Parm>);

Sets the functional transform to be used by the IDM model component.

**Input Arguments**

You must specify the following input arguments:

'IDMComp' is a character string that specifies the IDM component model. You can specify one of the following values:

AVERAGE specifies the average demand model.
INTERVAL specifies demand interval model.
SIZE specifies demand size model.

'Type' takes a string value that specifies the transform to use. You can specify one of the following values:

AUTO automatically chooses between NONE and LOG based on model selection criteria.
BOXCOX(value) requests Box-Cox transformation with a parameter value between –5 and 5. The default is BOXCOX(1).
LOG requests logarithmic transformation.
LOGIT requests logistic transformation.
NONE does not apply a transformation.
SQRT requests square-root transformation.

The default is AUTO.

You can also specify the following input arguments:
takes a string value that specifies prediction semantics for the inverse transform. You can specify the following values:

- **MEAN**: requests that the inverse transform produce mean forecasts.
- **MEDIAN**: requests that the inverse transform produce median forecasts.

The default is MEAN.

**Parm**

takes a numeric value between –5 and 5 that specifies a control parameter. This parameter is allowed only for Box-Cox transforms.

---

**UCMSPEC Object**

The unobserved component model (UCM) specification object generates UCM models for use with the TSM object. You open the UCMSPEC object to begin defining a new UCM model. You call the UCMSPEC methods to define the model characteristics, and then you close the UCMSPEC object to ready it for use in a TSM object. For more information, see the section “TSM.Initialize Method” on page 500. You can also store the XML representation of the UCM in a CAS table. For more information, see the section “TSMSPEC Object” on page 542. Note that the UCMSPEC object is equivalent to the HPFUCMSPEC procedure in *SAS Forecast Server Procedures: User’s Guide*.

Table 14.10 lists the methods that are associated with the UCMSPEC object.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AddAutoreg</td>
<td>Add an autoregressive component to a UCM model</td>
</tr>
<tr>
<td>AddBlockSeason</td>
<td>Add a block-season component to a UCM model</td>
</tr>
<tr>
<td>AddComponent</td>
<td>Add a fundamental component to a UCM model</td>
</tr>
<tr>
<td>AddCycle</td>
<td>Add a cycle component to a UCM model</td>
</tr>
<tr>
<td>AddDeplag</td>
<td>Add a deplag component to a UCM model</td>
</tr>
<tr>
<td>AddInput</td>
<td>Add an input component to a UCM model</td>
</tr>
<tr>
<td>AddSeason</td>
<td>Add a seasonal component to a UCM model</td>
</tr>
<tr>
<td>Close</td>
<td>Close the UCM model specification</td>
</tr>
<tr>
<td>Open</td>
<td>Open the UCM model specification</td>
</tr>
<tr>
<td>SetTransform</td>
<td>Specify the transform</td>
</tr>
</tbody>
</table>

Figure 14.7 illustrates the data flow through the UCMSPEC object.
UCMSPEC Synopsis

DECLARE OBJECT obj (UCMSPEC) ;

Method syntax, in order of typical usage:

rc = obj.Open () ;
rc = obj.AddAutoreg ( < Rho, NoestRho, Variance, Noest > ) ;
rc = obj.AddBlockSeason (BlockSize, NBlocks, < 'Type', Offset, Variance, Noest > ) ;
rc = obj.AddComponent ( Comp, < Variance, Noest > ) ;
rc = obj.AddCycle ( < Period, NoestPeriod, Rho, NoestRho, Variance, Noest > ) ;
rc = obj.AddDepLag (LagArray, < NLag, CoefArray, Noest > ) ;
rc = obj.AddInput (XName, < Delay, DiffArray, NDiff, Transform, TransParm > ) ;
rc = obj.AddSeason (Length, < 'Type', Variance, Noest > ) ;
rc = obj.SetTransform (XName, < Delay, DiffArray, NDiff, 'Type', TransParm > ) ;
rc = obj.Close () ;
UCMSPEC Methods

UCMSPEC.AddAutoreg Method

```plaintext
rc = obj.AddAutoreg (< Rho, NoestRho, Variance, Noest > ) ;
```

Adds an autoregressive component to a UCM model specification. You can add at most one autoregressive component to a UCM model specification.

**Input Arguments**
You can specify the following input arguments:

- **Rho**: takes a numeric value that specifies the starting value for the AR(1) coefficient during the parameter estimation process. The value of `Rho` must be in the interval \((-1, 1]\). The default is a missing value.
- **NoestRho**: takes a Boolean value that specifies whether the AR(1) coefficient is to be estimated (0) or fixed (1) at the specified starting value. If `NoestRho` is 1, a valid nonmissing `Rho` value must be specified. The default value is 0.
- **Variance**: specifies an initial value for the disturbance variance during the parameter estimation process. Any nonnegative value, including 0, is an acceptable starting value. The default is a missing value.
- **Noest**: takes a Boolean value that specifies whether the variance of the AR(1) noise process is to be estimated (0) or fixed (1). If `Noest` is 1, a valid nonmissing `Variance` value must be specified. The default value is 0.

UCMSPEC.AddBlockSeason Method

```plaintext
rc = obj.AddBlockSeason (BlockSize, NBlocks, < 'Type', Offset, Variance, Noest > ) ;
```

Adds a block-season component to a UCM model specification. You can use this method repeatedly to add more block-season components to a UCM model specification.

**Input Arguments**
You must specify the following input arguments:

- **BlockSize**: takes a numeric value that specifies the block size, where `BlockSize` can be any integer larger than or equal to 2. Typical examples of block sizes are 24 (which corresponds to the hours of the day when a day is used as a block in hourly data) or 60 (which corresponds to the minutes in an hour when an hour is used as a block in data that are recorded by minutes).
- **NBlocks**: takes a numeric value that specifies the number of blocks, where the `NBlocks` can be any integer greater than or equal to 2.

You can also specify the following input arguments:

- **'Type'**: is a character string that specifies the type of the seasonal component. You can specify one of the following values:
**UCMSPEC Methods**

**DUMMY** specifies a dummy type seasonal component.

**TRIG** specifies a trigonometric seasonal component.

The default is DUMMY.

**Offset** takes a numeric value that specifies the offset (the position of the first measurement within the block, if the first measurement is not at the start of a block). The Offset value must be an integer between 1 and BlockSize.

**Variance** specifies an initial value for the disturbance variance during the parameter estimation process. Any nonnegative value, including 0, is an acceptable starting value. The default is a missing value.

**Noest** takes a Boolean value that specifies whether the disturbance variance is to be estimated (0) or fixed (1). If Noest is 1, a valid nonmissing Variance value must be specified. The default value is 0.

**UCMSPEC.AddComponent Method**

\[ rc = obj.AddComponent(Comp, <Variance,Noest>) ; \]

Adds a fundamental component (such as LEVEL, SLOPE, or IRREGULAR) to a UCM model specification. This method must be called separately to add each of these fundamental components. A UCM model specification cannot contain more than one instance of these fundamental components.

**Input Arguments**

You must specify the following input argument:

**Comp** is a character string that specifies the UCM component model. You can specify one of the following values:

**IRREGULAR** includes an irregular component in the model that corresponds to the overall random error in the model.

**LEVEL** includes a level component in the model. The level component, either by itself or together with a slope component, forms the trend component. If the slope component is absent, the resulting trend is a random walk (RW). If the slope component is present, then a locally linear trend (LLT) is obtained.

**SLOPE** includes a slope component in the model. If you specify this value, you must call the method again with the value LEVEL for the Comp argument.

You can also specify the following input arguments:

**Variance** specifies an initial value for the disturbance variance during the parameter estimation process. Any nonnegative value, including 0, is an acceptable starting value. The default is a missing value.

**Noest** takes a Boolean value that specifies whether the disturbance variance is to be estimated (0) or fixed (1). If Noest is 1, a valid nonmissing Variance value must be specified. The default value is 0.
UCMSPEC.AddCycle Method

\[ rc = obj.AddCycle(<\text{Period}, \text{NoestPeriod}, \text{Rho}, \text{NoestRho}, \text{Variance}, \text{Noest}>) \];

Adds a cycle component to a UCM model specification. You can add multiple cycle components to a UCM model specification.

**Input Arguments**

You can specify the following input arguments:

- **Period**
  - Specifies a numeric value as a starting value for the cycle period during the parameter estimation process, where \text{Period} can be any number larger than or equal to 2. The default is a missing value.

- **NoestPeriod**
  - Takes a Boolean value that specifies whether the cycle period is to be estimated (0) or fixed (1) at the specified starting value. If \text{NoestPeriod} is 1, a valid nonmissing \text{Period} value must be specified. The default value is 0.

- **Rho**
  - Takes a numeric value that specifies the starting value for the AR(1) coefficient during the parameter estimation process. The value of \text{Rho} must be in the interval \((-1, 1]\). The default is a missing value.

- **NoestRho**
  - Takes a Boolean value that specifies whether the AR(1) coefficient is to be estimated (0) or fixed (1) at the specified starting value. If \text{NoestRho} is 1, a valid nonmissing \text{Rho} value must be specified. The default value is 0.

- **Variance**
  - Specifies an initial value for the disturbance variance during the parameter estimation process. Any nonnegative value, including 0, is an acceptable starting value. The default is a missing value.

- **Noest**
  - Takes a Boolean value that specifies whether the disturbance variance is to be estimated (0) or fixed (1). If \text{Noest} is 1, a valid nonmissing \text{Variance} value must be specified. The default value is 0.

UCMSPEC.AddDeplag Method

\[ rc = obj.AddDeplag(<\text{LagArray}, <\text{NLag}, \text{CoeffArray}, \text{Noest}>) \];

Adds a deplag component to a UCM model specification. You can add at most one deplag component to a UCM model specification.

**Input Arguments**

You must specify the following input argument:

- **LagArray**
  - Specifies an integer array that defines a model with specified lags of the dependent variable included as predictors.

You can also specify the following input arguments:

- **NLag**
  - Takes an integer that specifies the number of \text{LagArray} values to use. The default is all \text{LagArray} values.

- **CoeffArray**
  - Is a numeric array that specifies initial coefficients for the lagged dependent variables.
Noest takes a Boolean value that specifies whether lag coefficients are to be estimated (0) or fixed (1). If Noest is 1, a valid nonmissing Variance value must be specified. The default value is 0.

**UCMSPEC.AddInput Method**

```plaintext
rc = obj.AddInput (XName, <Delay,DiffArray,NDiff,Transform, TransParm>) ;
```

Adds an input (predictor variable) to a UCM model specification. You can add multiple inputs to a UCM model specification.

**Input Arguments**

You must specify the following input argument:

- **XName** is a character variable that specifies the name of the X-variable symbol.

You can also specify the following input arguments:

- **Delay** takes a nonnegative integer value that specifies the lag of the X-variable in the model. If not specified, the Delay is 0.
- **DiffArray** is numeric array of nonnegative integers that specifies differencing orders for the X-variable. If not specified, no differencing is applied to the predictor series.
- **NDiff** takes a numeric variable that specifies the number of DiffArray values to use. If not specified, the cardinality of DiffArray is used.
- **Transform** is character variable that specifies the name of a functional transform. You can specify one of the following values:
  - **BOXCOX(value)** requests Box-Cox transformation with a parameter value between –5 and 5. The default is BOXCOX(1).
  - **LOG** requests logarithmic transformation.
  - **LOGIT** requests logistic transformation.
  - **NONE** does not apply a transformation.
  - **SQRT** requests square-root transformation.

The default is NONE.

- **TransParm** takes a numeric value that specifies a transform parameter. This argument is used only for Box-Cox transformations.

**UCMSPEC.AddSeason Method**

```plaintext
rc = obj.AddSeason (Length, <'Type', Variance,Noest>) ;
```

Adds a seasonal component to a UCM model specification. You can add multiple seasonal component to a UCM model specification.
**Input Arguments**
You must specify the following input argument:

*Length* is a numeric variable that specifies the length of a seasonal cycle.

You can also specify the following input arguments:

*Type* is a character string that specifies the type of the seasonal component. You can specify one of the following values:

- **DUMMY** specifies a dummy type seasonal component.
- **TRIG** specifies a trigonometric seasonal component.

The default is DUMMY.

*Variance* specifies an initial value for the disturbance variance during the parameter estimation process. Any nonnegative value, including 0, is an acceptable starting value. The default is a missing value.

*Noest* takes a Boolean value that specifies whether the disturbance variance is to be estimated (0) or fixed (1). If *Noest* is 1, a valid nonmissing *Variance* value must be specified. The default value is 0.

**UCMSPEC.Close Method**

\[ rc = obj.Close(); \]

Finalizes the UCM model in the UCMSPEC object. This method prepares the UCM model for use in a TSM object or to be imported to a TSMSPEC object for printing or storage to a model repository catalog.

**Arguments**
There are no arguments associated with this method.

**UCMSPEC.Open Method**

\[ rc = obj.Open(); \]

Opens the UCMSPEC object for configuration. This method initializes an empty UCM model.

**Arguments**
There are no arguments associated with this method.

**UCMSPEC.SetTransform Method**

\[ rc = obj.SetTransform('Type < 'Option',Parm>); \]

Specifies the functional transform to be used by the UCM model.

**Input Arguments**
You must specify the following input argument:
'Type' takes a string value that specifies the transform to use. You can specify one of the following values:

- **BOXCOX(value)** requests Box-Cox transformation with a parameter `value` between –5 and 5. The default is BOXCOX(1).
- **LOG** requests logarithmic transformation.
- **LOGIT** requests logistic transformation.
- **NONE** does not apply a transformation.
- **SQRT** requests square-root transformation.

The default is NONE.

'Option' takes a string value that specifies prediction semantics for the inverse transform. You can specify the following values:

- **MEAN** requests that the inverse transform produce mean forecasts.
- **MEDIAN** requests that the inverse transform produce median forecasts.

The default is MEAN.

*Parm* takes a numeric value between –5 and 5 that specifies a control parameter for the functional transform. This argument is allowed only for Box-Cox transforms.

---

**TSMPEST Object**

The time series model parameter estimates (TSMPEST) collector object collects parameter estimates from a TSM object and stores them in a CAS table for printing or archiving or for use by a repeater object on another TSM object.

Table 14.11 summarizes the methods that are associated with the TSMPEST object.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Collect</td>
<td>Collect parameter estimates from TSM objects</td>
</tr>
<tr>
<td>nrows</td>
<td>Return the number of rows that are collected</td>
</tr>
<tr>
<td>SetOption</td>
<td>Set options for the TSMPEST collector object</td>
</tr>
</tbody>
</table>

Figure 14.8 illustrates the data flow through the TSMPEST object.
Figure 14.8  TSMPEST Object Data Flow

TSMPEST Synopsis

DECLARE OBJECT obj (TSMPEST) ;

Method syntax, in order of typical usage:

\[
rc = \text{obj}.\text{SetOption} ('Name', \text{Value} < 'Name', \text{Value}, \ldots >) ;
\]
\[
rc = \text{obj}.\text{Collect} (\text{TSMObj}) ;
\]
\[
nrows = \text{obj}.\text{nrows} () ;
\]

TSMPEST Methods

TSMPEST.Collect Method

\[
rc = \text{obj}.\text{Collect} (\text{TSMObj}) ;
\]

Collects time series model parameter estimates from a TSM object, \text{TSMObj}, and stores them in a CAS table.

Input Arguments
You must specify the following input argument:

\text{TSMObj} specifies the TSM object to use as the source of time series model parameter estimates.
**TSMPEST.nrows Method**

```
nrows=obj.nrows () ;
```

Returns the number of rows that have been collected and stored in the CAS table.

**Arguments**

There are no arguments associated with this method.

---

**TSMPEST.SetOption Method**

```
rc=obj.SetOption ('Name', Value <,'Name',Value,...> ) ;
```

Specifies the options for the TSMPEST object.

**Input Arguments**

You must specify at least one of the following 'Names' and its associated Value:

- **'MODELNAME'**
  - Takes a string Value that specifies the name of the model to be stored into the CAS table along with the parameter estimates of the model. By default, the model name is the name of the TSM object instance that is collected by the TSMPEST.Collect method.

- **'FORCEMODELNAME'**
  - Takes a boolean Value that specifies whether the model name should be stored into the CAS table even when the model failed to run. You can specify one of the following values:
    - **DISABLE|NO|N|FALSE|F|OFF**
      - Do not store the model name into the CAS table if the model failed to run.
    - **ENABLE|YES|Y|TRUE|T|ON**
      - Always store the model into the CAS table.

The default is DISABLE.
TSMSPEC Object

The time series model specification (TSMSPEC) collector object collects the model specification from a TSM object, TSMObj, and stores it in a CAS table for printing or archiving or for use by a repeater object on another TSM object.

Table 14.12 summarizes the methods that are associated with the TSMSPEC object.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Collect</td>
<td>Collect model specification from TSM objects</td>
</tr>
<tr>
<td>nrows</td>
<td>Return the number of rows that are collected</td>
</tr>
</tbody>
</table>

Figure 14.9 illustrates the data flow through the TSMSPEC object.
**TSMSPEC Synopsis**

```
DECLARE OBJECT obj (TSMSPEC) ;
```

Method syntax, in order of typical usage:

```
rc = obj.Collect (TSMObj) ;
nrows = obj.nrows () ;
```

---

**TSMSPEC Methods**

**TSMSPEC.Collect Method**

```
rc = obj.Collect (TSMObj) ;
```

Collects time series model specifications from a TSM object, `TSMObj`, and stores them in a CAS table.

**Input Arguments**

You must specify the following input argument:

`TSMObj` specifies the TSM object to use as the source of time series model parameter estimates.

---

**TSMSPEC.nrows Method**

```
nrows = obj.nrows () ;
```

Returns the number of rows that have been collected and stored in the CAS table.

**Arguments**

There are no arguments associated with this method.

---

**TSMFOR Object**

The TSMFOR object collects forecast series from a TSM object, `TSMObj`, and stores them in a CAS table. The CAS table schema that is used for storing the set of forecast series variables is compatible with the schema that the HPFENGINE procedure uses for the data set that is specified in the TSMFOR= option. Alternatively, the TSMFOR object can be configured to collect from a TSM object the smoothed (imputed) values for the dependent series and their associated standard errors and confidence limits.

Table 14.13 shows the contents of the TSMFOR object.
Table 14.13  Contents of the TSMFOR Object

<table>
<thead>
<tr>
<th>Column</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>NAME</em></td>
<td>String</td>
<td>Name of the dependent variable</td>
</tr>
<tr>
<td><em>MODEL</em></td>
<td>String</td>
<td>(Optional) Model name</td>
</tr>
<tr>
<td><em>TIMEID</em></td>
<td>Numeric</td>
<td>Uniform time ID values for series</td>
</tr>
<tr>
<td>ACTUAL</td>
<td>Numeric</td>
<td>Accumulated values of dependent variable</td>
</tr>
<tr>
<td>ERROR</td>
<td>Numeric</td>
<td>Residuals</td>
</tr>
<tr>
<td>LOWER</td>
<td>Numeric</td>
<td>Lower confidence limit</td>
</tr>
<tr>
<td>PREDICT</td>
<td>Numeric</td>
<td>Forecasts of the dependent variable</td>
</tr>
<tr>
<td>STD</td>
<td>Numeric</td>
<td>Prediction standard error</td>
</tr>
<tr>
<td>UPPER</td>
<td>Numeric</td>
<td>Upper confidence limit</td>
</tr>
</tbody>
</table>

Table 14.14 summarizes the methods that are associated with the TSMFOR object.

Table 14.14  Methods of the TSMFOR Object

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Collect</td>
<td>Collect forecasts estimates from the TSM object</td>
</tr>
<tr>
<td>nrows</td>
<td>Return the number of rows collected</td>
</tr>
<tr>
<td>SetOption</td>
<td>Specifies options for the TSMFOR instance</td>
</tr>
</tbody>
</table>

Figure 14.10 illustrates the data flow through the TSMFOR object.
TSMFOR Synopsis

DECLARE OBJECT obj (TSMFOR <('Name', Value)>) ;

Method syntax, in order of typical usage:

rc = obj.SetOption (Name, Value) ;
rc = obj.Collect (ModelObj < .Region >) ;
rc = obj.nrows () ;

Input Arguments
You can optionally specify one or more of the following 'Names' and its associated Value:

'MODELNAME'
takes a string Value that specifies whether to add a model name column to the output CAS table. The default model name is the TSM instance name. You can override the default model name via the 'MODELNAME' option of the SetOption method. You can specify one of the following Values:

YES includes a model name column in the output CAS table.
NO does not include a model name column in the output CAS table.

The default is NO.
takes a string Value that specifies whether to output the smoothed (imputed) dependent series values and their associated standard errors and confidence limits. If you set this option to 'YES', the PREDICT column holds the values of the smoothed dependent series instead of the forecast series. You can specify one of the following Values:

YES outputs the smoothed dependent series and its associated statistics from a TSM instance to a CAS table.

NO outputs the forecast series and its associated statistics from a TSM instance to a CAS table.

The default is NO.

---

**TSMFOR Methods**

**TSMFOR.Collect Method**

\[
rc = \text{obj.Collect}(\text{ModelObj}<.\text{Region}>)
\]

Collects the forecast series from a TSM object, ModelObj. An optional Region argument can be specified to indicate the forecast region to be collected. If you specify the 'SMOOTH' option in the declaration of the TSMFOR instance, then the smoothed (imputed) dependent series is collected instead.

**Input Arguments**

You must specify the following input argument:

ModelObj is a character string that specifies the name of the TSM object to use as the source of time series model forecasts.

You can also specify the following input argument:

Region specifies the time region over which to collect the forecast series. You can specify the following values for Region:

- **string** specifies the collection region. You can specify the following strings:
  - ALL collects over the entire time span of the available data.
  - FIT collects over the time region that supplied observations to estimate model parameters (that is, the model fit region).
  - FORECAST collects over the time region that is subsequent to the FIT region and that did not contribute any data to the model parameter estimation process (that is, the model forecast region).

The default is ALL.
**TSMSTAT Object**

The TSMSTAT object collects model statistics of fit from a TSM object, `TSMObj`, and stores them in a CAS table. This information is useful for evaluating how well the model fits the dependent series. The CAS table schema that is used for storing the fit statistics is compatible with the schema that the HPFENGINE procedure uses for its `OUTSTAT=` data set.

Table 14.15 shows the contents of the TSMSTAT object.

---

**numeric** is a two-valued numeric array, in which the first value specifies the starting time ID and the second value specifies the ending time ID of the time region over which the forecast series are to be collected. Either the starting time ID or the ending time ID can be a missing value. If both are missing values, then the default value ALL is used.

**TSMFOR.nrows Method**

```r
rc = obj.nrows();
```

Returns the number of rows that have been collected and stored in the CAS table.

**Arguments**

There are no arguments associated with this method.

**TSMFOR.SetOption Method**

```r
cr = obj.SetOption (‘Name’, Value);
```

Specifies the named options for the TSMFOR instance.

**Input Arguments**

You must specify the following `Name` and its associated `Value`:

- `'MODELNAME'` takes a string `Value` that is copied to the `_MODEL_` column of the output CAS table. This option is relevant only if you set the `MODELNAME` option to 'YES' in the declaration of the TSMFOR instance. The default value is the TSM instance name.

---

**TSMSTAT Object**

The TSMSTAT object collects model statistics of fit from a TSM object, `TSMObj`, and stores them in a CAS table. This information is useful for evaluating how well the model fits the dependent series. The CAS table schema that is used for storing the fit statistics is compatible with the schema that the HPFENGINE procedure uses for its `OUTSTAT=` data set.

Table 14.15 shows the contents of the TSMSTAT object.
### Table 14.15  Contents of the TSMSTAT Object

<table>
<thead>
<tr>
<th>Column</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>NAME</em></td>
<td>String</td>
<td>Name of the dependent variable</td>
</tr>
</tbody>
</table>
| _REGION_ | String | Region in which the fit statistics are calculated. You can specify the following values:  
  FIT indicates that fit statistics were calculated over the fit region.  
  FORECAST indicates that fit statistics were calculated over the forecast (back) region.  
  HOLDOUT indicates that fit statistics were calculated over the holdout region. |
<p>| <em>MODEL</em> | String | Model name                                                                   |
| AADJRSQ | Numeric | Amemiya’s adjusted R-square                                                  |
| ADJRSQ  | Numeric | Adjusted R-square                                                            |
| AIC     | Numeric | Akaike’s information criterion                                               |
| AICC    | Numeric | Finite sample corrected AIC                                                  |
| APC     | Numeric | Amemiya’s prediction criterion                                               |
| DFE     | Numeric | Degrees of freedom error                                                     |
| GMAPE   | Numeric | Geometric mean absolute percentage error                                     |
| GMAPES  | Numeric | Geometric mean absolute error as a percentage of standard deviation          |
| GMAPPE  | Numeric | Geometric mean absolute predictive percentage error                          |
| GMRAE   | Numeric | Geometric mean relative absolute error                                       |
| GMASPE  | Numeric | Geometric mean absolute symmetric percentage error                           |
| MAE     | Numeric | Mean absolute error                                                          |
| MAPE    | Numeric | Mean absolute percentage error                                                |
| MAPES   | Numeric | Mean absolute error as a percentage of standard deviation                    |
| MAPPE   | Numeric | Mean absolute predictive percentage error                                    |
| MASE    | Numeric | Mean absolute scaled error                                                   |
| MAXAPES | Numeric | Maximum absolute error as a percentage of standard deviation                 |
| MAXERR  | Numeric | Maximum error                                                                |
| MAXPE   | Numeric | Maximum percentage error                                                     |</p>
<table>
<thead>
<tr>
<th>Column</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MAXPPE</td>
<td>Numeric</td>
<td>Maximum predictive percentage error</td>
</tr>
<tr>
<td>MAXRE</td>
<td>Numeric</td>
<td>Maximum relative error</td>
</tr>
<tr>
<td>MAXSPE</td>
<td>Numeric</td>
<td>Maximum symmetric percentage error</td>
</tr>
<tr>
<td>MDAPE</td>
<td>Numeric</td>
<td>Median absolute percentage error</td>
</tr>
<tr>
<td>MDAPES</td>
<td>Numeric</td>
<td>Median absolute error as a percentage of standard deviation</td>
</tr>
<tr>
<td>MDAPPE</td>
<td>Numeric</td>
<td>Median absolute predictive percentage error</td>
</tr>
<tr>
<td>MDASPE</td>
<td>Numeric</td>
<td>Median absolute symmetric percentage error</td>
</tr>
<tr>
<td>MDRAE</td>
<td>Numeric</td>
<td>Median relative absolute error</td>
</tr>
<tr>
<td>ME</td>
<td>Numeric</td>
<td>Mean error</td>
</tr>
<tr>
<td>MINAPES</td>
<td>Numeric</td>
<td>Minimum absolute error as a percentage of standard deviation</td>
</tr>
<tr>
<td>MINERR</td>
<td>Numeric</td>
<td>Minimum error</td>
</tr>
<tr>
<td>MINPE</td>
<td>Numeric</td>
<td>Minimum percentage error</td>
</tr>
<tr>
<td>MINPPE</td>
<td>Numeric</td>
<td>Minimum predictive percentage error</td>
</tr>
<tr>
<td>MINRE</td>
<td>Numeric</td>
<td>Minimum relative error</td>
</tr>
<tr>
<td>MINSPE</td>
<td>Numeric</td>
<td>Minimum symmetric percentage error</td>
</tr>
<tr>
<td>MPE</td>
<td>Numeric</td>
<td>Mean percentage error</td>
</tr>
<tr>
<td>MPPE</td>
<td>Numeric</td>
<td>Mean predictive percentage error</td>
</tr>
<tr>
<td>MRAE</td>
<td>Numeric</td>
<td>Mean relative absolute error</td>
</tr>
<tr>
<td>MRE</td>
<td>Numeric</td>
<td>Mean relative error</td>
</tr>
<tr>
<td>MSE</td>
<td>Numeric</td>
<td>Mean square error</td>
</tr>
<tr>
<td>MSPE</td>
<td>Numeric</td>
<td>Mean symmetric percentage error</td>
</tr>
<tr>
<td>N</td>
<td>Numeric</td>
<td>Number of observations that were used</td>
</tr>
<tr>
<td>NMISSA</td>
<td>Numeric</td>
<td>Number of missing actual values</td>
</tr>
<tr>
<td>NMISSP</td>
<td>Numeric</td>
<td>Number of missing predicted values</td>
</tr>
<tr>
<td>NOBS</td>
<td>Numeric</td>
<td>Number of observations</td>
</tr>
<tr>
<td>NPARMS</td>
<td>Numeric</td>
<td>Number of parameters</td>
</tr>
<tr>
<td>RMSE</td>
<td>Numeric</td>
<td>Root mean square error</td>
</tr>
<tr>
<td>RSQUARE</td>
<td>Numeric</td>
<td>R-square</td>
</tr>
<tr>
<td>RWRSQ</td>
<td>Numeric</td>
<td>Random walk R-square</td>
</tr>
</tbody>
</table>
Table 14.15  continued

<table>
<thead>
<tr>
<th>Column</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SBC</td>
<td>Numeric</td>
<td>Schwarz Bayesian information criterion</td>
</tr>
<tr>
<td>SMAPE</td>
<td>Numeric</td>
<td>Symmetric mean absolute percentage error</td>
</tr>
<tr>
<td>SSE</td>
<td>Numeric</td>
<td>Sum of square error</td>
</tr>
<tr>
<td>SST</td>
<td>Numeric</td>
<td>Corrected total sum of squares</td>
</tr>
<tr>
<td>TSS</td>
<td>Numeric</td>
<td>Total sum of squares</td>
</tr>
<tr>
<td>UMSE</td>
<td>Numeric</td>
<td>Unbiased mean square error</td>
</tr>
<tr>
<td>URMSE</td>
<td>Numeric</td>
<td>Unbiased root mean square error</td>
</tr>
</tbody>
</table>

Table 14.16 summarizes the methods that are associated with the TSMSTAT object.

Table 14.16  Methods of the TSMSTAT Object

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Collect</td>
<td>Collect model statistics of fit from the TSM object</td>
</tr>
<tr>
<td>nrows</td>
<td>Return the number of rows collected</td>
</tr>
<tr>
<td>SetOption</td>
<td>Set options for the TSMSTAT collector object</td>
</tr>
</tbody>
</table>

Figure 14.11 illustrates the data flow through the TSMSTAT object.
TSMSTAT Synopsis

DECLARE OBJECT obj (TSMSTAT) ;

Method syntax, in order of typical usage:

   rc=obj.SetOption ('Name', Value < ',Name', Value, ... >) ;
   rc=obj.Collect (TSMObj) ;
   nrows=obj.nrows () ;

TSMSTAT Methods

TSMSTAT.Collect Method

   rc=obj.Collect (TSMObj) ;

Collects model statistics of fit from a TSM object, TSMObj, and stores them in a CAS table.

Input Arguments
You must specify the following input argument:

Figure 14.11  TSMSTAT Object Data Flow
**Chapter 14: Time Series Model Package**

*TSMObj* specifies the TSM object to use as the source of model fit statistics.

**TSMSTAT.nrows Method**

```plaintext
nrows = obj.nrows () ;
```

Returns the number of rows that have been collected and stored in the CAS table.

**Arguments**

There are no arguments associated with this method.

**TSMSTAT.SetOption Method**

```plaintext
rc = obj.SetOption ( 'Name', Value < 'Name', Value, ... >) ;
```

Specifies the options for the TSMSTAT object.

**Input Arguments**

You must specify at least one of the following *Names* and its associated *Value*:

- **'MODELNAME'**: takes a string *Value* that specifies the name of the model to be stored into the CAS table along with the parameter estimates of the model. By default, the model name is the name of the TSM object instance that is collected by the TSMSTAT.Collect method.

- **'FORCENAME'**: takes a boolean *Value* that specifies whether the model name should be stored into the CAS table even when the model failed to run. You can specify one of the following values:

  - **DISABLE|NO|N|FALSE|F|OFF**: Do not store the model name into the CAS table if the model failed to run.
  - **ENABLE|YES|Y|TRUE|T|ON**: Always store the model into the CAS table.

The default is DISABLE.
**TSMINEST Object**

The time series model input estimates (TSMINEST) repeater object imports parameter estimates from a CAS table for use in a TSM object. The TSMINEST table schema required for the parameter estimates is compatible with that used by the TSMPEST collector object. The TSMINEST object must be used in conjunction with a TSMINSPEC specification object and a TSM model object to replay a model by using its saved parameters (for an example, see Example 14.3).

Table 14.17 summarizes the methods of the TSMINEST object.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>nrows</td>
<td>Return number of rows in the TSMINEST object</td>
</tr>
</tbody>
</table>

### TSMINEST Synopsis

```
DECLARE OBJECT obj (TSMINEST) ;
```

Method syntax:

```
rc=obj.nrows () ;
```

### TSMINEST Methods

#### TSMINEST.nrows Method

```
rc=obj.nrows () ;
```

Returns the number of rows in the TSMINEST object, `obj`. A returned missing value indicates that the object has not been successfully configured.

**Arguments**

There are no arguments associated with this method.
**TSMINSPEC Object**

The time series model input specification (TSMINSPEC) repeater object imports model specifications from a CAS table for use in a TSM object. The TSMINSPEC table schema required for the model specification is compatible with that used by the TSMSPEC collector object.

Table 14.18 summarizes the methods of the TSMINSPEC object.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>nrows</td>
<td>Return number of rows in TSMINSPEC object</td>
</tr>
</tbody>
</table>

**TSMINSPEC Synopsis**

```plaintext
DECLARE OBJECT obj (TSMINSPEC) ;
```

Method syntax:

```plaintext
rc obj.nrows () ;
```

**TSMINSPEC Methods**

**TSMINSPEC.nrows Method**

```plaintext
rc=obj.nrows () ;
```

Returns the number of rows that have been collected and stored in the CAS table. A returned missing value indicates that the TSMINSPEC object has not been successfully configured.

**Arguments**

There are no arguments associated with this method.
Examples: TSM Package

Throughout this section it is assumed that you have already started a CAS session and the data tables that are used in this section are stored in mycas, a CAS library that you have necessary permissions to work with. This section assumes that you are familiar with the general workings of the TSMODEL procedure; for more information, see Chapter 11, “The TSMODEL Procedure” (*SAS Visual Forecasting: Forecasting Procedures*).

Using CAS Sessions and CAS Engine Librefs

SAS Cloud Analytic Services (CAS) is the analytic server and associated cloud services in SAS Viya. This section describes how to create a CAS session and set up a CAS engine libref that you can use to connect to the CAS session. It assumes that you have a CAS server already available; contact your system administrator if you need help starting and terminating a server. This CAS server is identified by specifying the host on which it runs and the port on which it listens for communications. To simplify your interactions with this CAS server, the host information and port information for the server are stored as SAS option values that are retrieved automatically whenever this CAS server needs to be accessed. You can examine the host and port values for the server at your site by using the following statements:

```
proc options option=(CASHOST CASPORT);
run;
```

In addition to starting a CAS server, your system administrator might also have created a CAS session and a CAS engine libref for your use. You can define your own sessions and CAS engine librefs that connect to the CAS server as shown in the following statements:

```
cas mysess;
libname mycas cas sessref=mysess;
```

The CAS statement creates the CAS session named *mysess*, and the LIBNAME statement creates the *mycas* CAS engine libref that you use to connect to this session. It is not necessary to explicitly name the CASHOST and CASPORT of the CAS server in the CAS statement, because these values are retrieved from the corresponding SAS option values.

If you have created the *mysess* session, you can terminate it by using the TERMINATE option in the CAS statement as follows:

```
cas mysess terminate;
```

For more information about the CAS statement and the LIBNAME statement, see *SAS Cloud Analytic Services: User’s Guide*. For general information about CAS and CAS sessions, see *SAS Cloud Analytic Services: Fundamentals*. 
Example 14.1: Fitting and Forecasting with ARIMA and ESM Models

The airline passenger data, given as Series G in Box and Jenkins (1976), have been used in time series analysis literature as an example of a nonstationary seasonal time series; for more information about ARIMA modeling of Series G, see “Example 7.2 Seasonal Model for the Airline Series” in the ARIMA chapter of SAS/ETS User’s Guide.

This example shows how you can use the objects in the TSM package to fit the airline model, ARIMA(0,1,1) × (0,1,1)\textsubscript{12} NOINT and to fit the Winters exponential smoothing model to the airline series. In particular, it shows how you can do the following:

1. Create ARIMA and ESM specifications by using the ARIMASPEC and ESMSPEC specification objects.
2. Use these specification objects to initialize the TSM model objects.
3. Use these TSM model objects to fit the airline model and Winters model to the airline series, and to forecast the series according to these models.
4. Postprocess the forecast results to compute an ad hoc statistic.
5. Output the computed results to CAS tables.

The broad outline of the code in this example is as follows:

1. The PROC TSMODEL statement specifies the input data set (mycas.air), a variety of output tables (mycas.airFor, mycas.airEst, and so on), and the forecast lead (12).
2. The ID statement specifies date as the time index variable, and the INTERVAL= option indicates that the data are monthly.
3. The VAR statement specifies the input data set variable, air, which contains the airline series.
4. The OUTARRAYS and OUTSCALARS statements declare some output arrays and scalars that are used to store the analysis results, which are subsequently saved as CAS tables.
5. The REQUIRE statement specifies the TSM package, which is needed for the analysis.
6. The statements between the SUBMIT and ENDSUBMIT statements use the TSM package objects to perform the actual analysis in your CAS session.
7. These statements are grouped in three sections:
   - The first section does the specification, fitting, and forecasting according to the airline model. The airSpec object contains the airline specification, and the airModel object is a TSM model object that is initialized by using airSpec. After the airModel is run, the parameter estimates are collected in airEst and the forecasts are collected in airFor. The model residuals are stored in the airErr array for later processing.
The second section does the specification, fitting, and forecasting according to the Winters exponential smoothing model. The esmSpec object contains the Winters specification, and the esmModel object is a TSM model object that is initialized by using esmSpec. The esmModel is run, and the model residuals are stored in the esmErr array for later processing.

The third section computes an ad hoc statistic called nbetter, which counts the number of times the airline model residuals are smaller (in absolute size) than the Winters model. This section is included to illustrate how you can write your own custom postprocessing code to analyze the results that are produced by the TSM objects.

```
proc tsmodel data=mycas.air
   outobj=(airFor=mycas.airFor airEst=mycas.airEst)
   outscalar=mycas.nbetter outarray=mycas.out lead=12;
   id date interval=month;
   var air;
   outarrays esmErr airErr;
   outscalars nbetter nfor;
   require tsm;
   submit;

   **Temporary work arrays used in ARIMA spec;
   array diff[2]/nosymbols;
   array ma[1]/nosymbols;

   *** Analysis based on the airline model ***;
   declare object airModel(tsm);
   declare object airSpec(arimaspec);

   **Set up the airline model spec:**;
   ** Model: log(air) ~ (0,1,1)(0,1,1)12 noint **;
   rc = airSpec.Open( );
   *** Specify differencing orders ***;
   diff[1] = 1;
   diff[2] = 12;
   rc = airSpec.SetDiff(diff,2);
   *** Specify moving average orders: q = (1)(12) ***;
   *** Use AddMAPoly twice for the two factors ***;
   ma[1] = 1;
   rc = airSpec.AddMAPoly(ma);
   ma[1] = 12;
   rc = airSpec.AddMAPoly(ma);
   *** Specify Noint ***;
   rc = airSpec.SetOption('noint',1);
   *** Specify the log transform ***;
   rc = airSpec.SetTransform('log');
   *** Done setting up the ARIMA model ***;
   rc = airSpec.Close( );

   *** Set up and run the airModel TSM object ***;
   rc = airModel.Initialize(airSpec);
   rc = airModel.SetY(Air);
   rc = airModel.SetOption('lead',12);
   rc = airModel.Run( );
```
*** Output the airline model forecasts and estimates ***;
declare object airFor(tsmfor);
declare object airEst(tsmpest);
rc = airFor.Collect(airModel);
rc = airEst.Collect(airModel);

*** Put the airline model residuals in airErr array; 
rc = airModel.getForecast('error',airErr);

*** Analysis based on ESM model ***;
declare object esmModel(tsm);
declare object esmSpec(esmspec);
rc = esmSpec.open( );
rc = esmSpec.SetOption('method', 'winters');
rc = esmSpec.close( );

*** Set up and run the TSM object ***;
rc = esmModel.Initialize(esmSpec);
rc = esmModel.SetY(Air);
rc = esmModel.SetOption('lead',12);
rc = esmModel.Run( );

*** Put the ESM model residuals in esmErr array; 
rc = esmModel.getForecast('error',esmErr);

*** Compute an ad hoc statistic based on airErr and esmErr arrays; 
nbetter = 0;
nfor = esmModel.nfor( );
do t=1 to nfor;
   if airErr[t] ^= . & esmErr[t] ^= . then do;
      if abs(airErr[t]) < abs(esmErr[t])
         then nbetter = nbetter + 1;
   end;
end;
endsubmit;
quit;

Output 14.1.1 shows the predictions, and Output 14.1.2 shows the parameter estimates for the airline model.
Output 14.1.1 Airline Model Predictions (Partial Output)

### Airline Model Predictions

<table>
<thead>
<tr>
<th>DATE</th>
<th>PREDICT</th>
<th>STD</th>
<th>UPPER</th>
<th>LOWER</th>
</tr>
</thead>
<tbody>
<tr>
<td>JAN1961</td>
<td>450.4</td>
<td>16.9215</td>
<td>484.5</td>
<td>418.2</td>
</tr>
<tr>
<td>FEB1961</td>
<td>426.1</td>
<td>18.8590</td>
<td>464.2</td>
<td>390.3</td>
</tr>
<tr>
<td>MAR1961</td>
<td>480.1</td>
<td>24.0408</td>
<td>528.9</td>
<td>434.7</td>
</tr>
<tr>
<td>APR1961</td>
<td>492.8</td>
<td>27.2405</td>
<td>548.3</td>
<td>441.6</td>
</tr>
<tr>
<td>MAY1961</td>
<td>509.5</td>
<td>30.5863</td>
<td>572.0</td>
<td>452.1</td>
</tr>
<tr>
<td>JUN1961</td>
<td>584.2</td>
<td>37.6514</td>
<td>661.4</td>
<td>513.9</td>
</tr>
<tr>
<td>JUL1961</td>
<td>670.7</td>
<td>45.9957</td>
<td>765.3</td>
<td>585.1</td>
</tr>
<tr>
<td>AUG1961</td>
<td>668.2</td>
<td>48.4237</td>
<td>768.0</td>
<td>578.3</td>
</tr>
<tr>
<td>SEP1961</td>
<td>559.6</td>
<td>42.6271</td>
<td>647.7</td>
<td>480.7</td>
</tr>
<tr>
<td>OCT1961</td>
<td>498.3</td>
<td>39.7181</td>
<td>580.6</td>
<td>425.0</td>
</tr>
<tr>
<td>NOV1961</td>
<td>431.2</td>
<td>35.8240</td>
<td>505.5</td>
<td>365.2</td>
</tr>
<tr>
<td>DEC1961</td>
<td>478.9</td>
<td>41.3484</td>
<td>565.0</td>
<td>403.0</td>
</tr>
</tbody>
</table>

Output 14.1.2 Airline Model Parameter Estimates (Partial Output)

### Airline Model Parameter Estimates

<table>
<thead>
<tr>
<th>EST</th>
<th>STDERR</th>
<th>TVALUE</th>
<th>PVALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.3773</td>
<td>0.0820</td>
<td>4.6033</td>
<td>9.828E-6</td>
</tr>
<tr>
<td>0.5724</td>
<td>0.0780</td>
<td>7.3361</td>
<td>2.17E-11</td>
</tr>
</tbody>
</table>

Output 14.1.3 shows the number of times the airline model residuals are smaller than the Winters model residuals.

Output 14.1.3 Number of Times the Airline Model Residuals Are Smaller Than the Winters Model Residuals

<table>
<thead>
<tr>
<th>nbetter</th>
</tr>
</thead>
<tbody>
<tr>
<td>71.0000</td>
</tr>
</tbody>
</table>

Example 14.2: Fitting a Transfer Function Model

This example uses the gas furnace data from Box and Jenkins (1976). The data, called Series J by Box and Jenkins, contain sequentially recorded measurements of two variables: \( x \), the input gas rate, and \( y \), the output \( \text{CO}_2 \). The data also include an index variable, time, which keeps track of the sequence number of each observation (essentially the row index). The TSMODEL procedure requires an ID variable that has a valid time interval associated with it to index the observations. In order to satisfy this requirement, the time variable is assigned as the time ID variable in the ID statement and its interval is specified as SECOND using the INTERVAL= option. The value INTERVAL=SECOND is one of the simplest interval types for sequential indexing. As shown in “Example 7.3 Model for Series J Data from Box and Jenkins” in the ARIMA chapter of *SAS/ETS User's Guide*, a reasonable ARIMAX model for \( y \) turns out to be \( y = \text{TFinput}(x) + \text{AR}(2) \), where TFinput\((x)\) is a transfer function term in \( x \) with a delay of 3, numerator polynomial of order 2, and
Chapter 14: Time Series Model Package

denominator polynomial of order 1, and where AR(2) is an error term of autoregressive order 2. The following statements show how to fit this model by using the objects in the TSM package:

```plaintext
proc tsm model data=mycas.seriesj
    outobj=(jEst=mycas.jEst);
    id time interval=second;
    var x y;
    require tsm;
    submit;

    *** Transfer function modeling for seriesJ ***;
    declare object jModel(tsm);
    declare object jSpec(arimaspec);
    declare object jEst(tsmpest);
    array num[2]/nosymbols;
    array den[1]/nosymbols;
    array ar[2]/nosymbols;

    *** Set up the transfer function model spec: ***;
    rc = jSpec.Open( );
    *** Specify AR orders: p = (1 2) ***;
    ar[1] = 1;
    ar[2] = 2;
    rc = jSpec.AddARPoly(ar);

    rc = jSpec.AddTF('x', 3); *delay=3;
    num[1] = 1;
    num[2] = 2;
    rc = jSpec.AddTFNumPoly('x', num);
    den[1] = 1;
    rc = jSpec.AddTFDenPoly('x', den);
    *** done setting up ARIMA model ***;
    rc = jSpec.Close( );

    *** Set up and run the TSM object ***;
    rc = jModel.Initialize(jSpec);
    rc = jModel.SetY(y);
    rc = jModel.AddX(x);
    rc = jModel.Run( );

    *** output gas furnace model forecasts and estimates ***;
    rc = jEst.Collect(jModel);
    endsubmit;
.quit;
```

Output 14.2.1 shows the parameter estimates for the transfer function model.
### Output 14.2.1 Parameter Estimates for the Transfer Function Model (Partial Output)

<table>
<thead>
<tr>
<th>EST</th>
<th>STDERR</th>
<th>TVALUE</th>
<th>PVALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>53.2630</td>
<td>0.1193</td>
<td>446.5</td>
<td>0</td>
</tr>
<tr>
<td>1.5329</td>
<td>0.0475</td>
<td>32.2472</td>
<td>6.25E-97</td>
</tr>
<tr>
<td>-0.6330</td>
<td>0.0501</td>
<td>-12.6434</td>
<td>2.27E-29</td>
</tr>
<tr>
<td>-0.5352</td>
<td>0.0748</td>
<td>-7.1534</td>
<td>7.21E-12</td>
</tr>
<tr>
<td>0.3760</td>
<td>0.1029</td>
<td>3.6553</td>
<td>0.000306</td>
</tr>
<tr>
<td>0.5189</td>
<td>0.1078</td>
<td>4.8124</td>
<td>2.425E-6</td>
</tr>
<tr>
<td>0.5484</td>
<td>0.0382</td>
<td>14.3499</td>
<td>1.72E-35</td>
</tr>
</tbody>
</table>

### Example 14.3: Replaying a Previously Fitted Model

In some cases, it is useful to save the model specification and parameter estimates that are computed during an analysis for later use. For example, you can use the saved model specification and parameter estimates to produce model forecasts at a later stage (possibly with new measurements appended to the original data). This example shows how you can do the following:

1. Save the model specification and parameter estimates for later use by using the Collect method of the TSMSPEC and TSMPest objects, respectively.

2. Reuse the previously saved model specification and parameter estimates to configure a TSM object by using the Replay method.

3. Produce the model forecasts by using this TSM object.

The following statements fit the airline model to the airline series (see Example 14.1 for more information about the airline series and the airline model). The model specification and parameter estimates are stored in CAS tables mycas.airOSpec and mycas.airEst, respectively.

```plaintext
proc tsmodel data=mycas.air
   outobj=(airEst=mycas.airEst airOSpec=mycas.airOSpec)
   id date interval=month;
   var air;
   require tsm;
submit;

   *** Analysis based on airline model ***;
   declare object airModel(tsm);
   declare object airSpec(arimaspec);
   declare object airEst(tsmpest);
   declare object airOSpec(tsmspec);

   array diff[2]/nosymbols;
   array ma[1]/nosymbols;

   *** Set up the airline model spec: ***;
   ** Model: log(air) ~ (0,1,1)(0,1,1)12 noint ***;
   rc = airSpec.Open();
```
*** Specify differencing orders ***;
  diff[1] = 1;
diff[2] = 12;
rc = airSpec.SetDiff(diff,2);
*** Specify moving average orders: q = (1)(12) ***;
*** Use AddMAPoly twice for the two factors ***;
  ma[1] = 1;
rc = airSpec.AddMAPoly(ma);
  ma[1] = 12;
rc = airSpec.AddMAPoly(ma);
*** Specify NOINT ***;
rc = airSpec.SetOption('noint',1);
*** Specify the log transform ***;
rc = airSpec.SetTransform('log');
*** Done setting up the ARIMA model ***;
rc = airSpec.Close();

*** Set up and run the TSM object ***;
rc = airModel.Initialize(airSpec);
rc = airModel.SetY(Air);
rc = airModel.SetOption('lead',12);
rc = airModel.Run();

*** Output airline model spec and estimates ***;
rc = airEst.Collect(airModel);
rc = airOSpec.Collect(airModel);
endsubmit;
quit;

Output 14.3.1 shows the parameter estimates that are saved in mycas.airEst.

**Output 14.3.1** Parameter Estimates for the Airline Model (Partial Output)

<table>
<thead>
<tr>
<th>Parameter Estimates for the Airline Model</th>
<th>EST</th>
<th>STDERR</th>
<th>TVALUE</th>
<th>PVALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.3773</td>
<td>0.0820</td>
<td>4.6033</td>
<td>9.828E-6</td>
<td></td>
</tr>
<tr>
<td>0.5724</td>
<td>0.0780</td>
<td>7.3361</td>
<td>2.17E-11</td>
<td></td>
</tr>
</tbody>
</table>

The following statements show how to forecast the airline series by using the previously saved model specification (mycas.airOSpec) and parameter estimates (mycas.airEst).

```plaintext
proc tsm model data=mycas.air
  outobj=(airFor=mycas.airFor)
  inobj=(airEst=mycas.airEst airSpec=mycas.airOSpec);
id date interval=month;
var air;
require tsm;
submit;

*** Analysis based on the airline model ***;
declare object airModel(tsm);
declare object airSpec(tsminspec);
declare object airEst(tsminest);
```
Example 14.4: Performing Time Series Imputation Using an ARIMA Model

Time series imputation is the process of replacing missing values in a time series with reasonable values that reflect the existing pattern in the available data (such as trend, seasonal variations, or long-term cyclical variations). It is a popular technique that is used across various domains of science. This example shows how you can do the following:

- Fit an ARIMA model to a time series that contains missing values by using a TSM object.
- Store the imputed values of the time series in a table.

For the purposes of this example, the following statements introduce two artificial missing values into the airline series (see Example 14.1 for more information about the airline series and the airline model). The modified series is stored in the mycas.airMiss CAS table. These statements assume that your CAS engine libref is named mycas, but you can substitute any appropriately defined CAS engine libref.
data mycas.airmiss;
    set mycas.air;
    airmiss = air;
    if date = '01JUL1955'd then airmiss = .;
    if date = '01AUG1955'd then airmiss = .;
run;

The following statements plot the modified airline series along with the two artificial missing values that were introduced. The results are shown in Output 14.4.1.

proc sort data=mycas.airmiss out=airmiss;
    by date;
run;

proc sgplot data=airmiss;
    series x=date y = airmiss / break lineattrs=(color=blue thickness=3);
    series x=date y = air / lineattrs=(thickness=2 pattern=dot color=blue);
run;

In Output 14.4.1, the solid blue line represents all the nonmissing data values in the modified airline series. The dotted blue line between the years 1955 and 1956 represents the two actual values that were set to missing values.

Output 14.4.1  Airline Passenger Time Series with Artificial Missing Values

The following statements fit the airline model to the modified airline series and store the imputed time series in the mycas.airImpute CAS table:
Example 14.4: Performing Time Series Imputation Using an ARIMA Model

```plaintext
proc tsmode data=mycas.airmiss outobj=(airImpute=mycas.airImpute) ;
   id date interval=month;
   var airmiss;
   require tsm;
submit;
   *** Analysis based on airline model ***;
   declare object airModel(tsm);
   declare object airSpec(arimaspec);
   declare object airImpute(tsmfor('SMOOTH','YES'));

   array diff[2]/nosymbols;
   array ma[1]/nosymbols;

   *** Set up the airline model spec: ***;
   ** Model: log(air) ~ (0,1,1)(0,1,1)12 noint ***;
   rc = airSpec.Open();
   *** Specify differencing orders ***;
   diff[1] = 1;
   diff[2] = 12;
   rc = airSpec.SetDiff(diff,2);
   *** Specify moving average orders: q = (1)(12) ***;
   *** Use AddMAPoly twice for the two factors ***;
   ma[1] = 1;
   rc = airSpec.AddMAPoly(ma);
   ma[1] = 12;
   rc = airSpec.AddMAPoly(ma);
   *** Specify NOINT ***;
   rc = airSpec.SetOption('noint',1);
   *** Specify the log transform ***;
   rc = airSpec.SetTransform('log');
   *** Done setting up the ARIMA model ***;
   rc = airSpec.Close();

   *** Set up and run the TSM object ***;
   rc = airModel.Initialize(airSpec);
   rc = airModel.SetY(Airmiss);
   rc = airModel.Run();

   *** Output the imputed airline time series ***;
   rc = airImpute.Collect(airModel);
endsubmit;
quit;
```

The following statements plot the modified airline series along with its imputed version. The results are shown in Output 14.4.2.

```plaintext
data mycas.airimpute;
   merge mycas.airimpute mycas.airmiss;
      by date;
run;
```
proc sort data=mycas.airimpute out=airimpute;
   by date;
run;

proc sgplot data=airimpute;
   series x=date y = actual / break lineattrs=(thickness=3 color=blue);
   series x=date y = air / lineattrs=(thickness=2 pattern=dot color=blue);
   series x=date y = predict / lineattrs=(thickness=1 color=red);
   where year(date) >= 1954 and year(date) <= 1956;
run;

In Output 14.4.2, the thick blue line represents the modified airline series, and the thin red line represents its imputed version. The time axis range is restricted to the dates between the years 1954 and 1956 in order to facilitate the visualization. Notice how the thin red line (imputed values) closely mimics the dotted blue line (original actual values) over the region between the years 1955 and 1956 where the artificial missing values were introduced.

Output 14.4.2  Imputed Airline Passenger Time Series
Example 14.5: Combining Forecasts

This example shows how you can use the objects in the TSM package to combine multiple model forecasts to generate an improved forecast. In particular, it shows how you can do the following:

1. Fit the airline model, \( \text{ARIMA}(0,1,1) \times (0,1,1)_{12} \text{ NOINT} \), to the airline series.

2. Fit the Winters exponential smoothing model to the airline series.

3. Use the CFC object to produce a combined forecast that is more accurate than the individual forecasts.

4. Output fit statistics for all models to a CAS table for later comparison.

```plaintext
proc tsmmodel data=mycas.air outobj=(fitstats=mycas.fitstats) lead=12;
    id date interval=month;
    var air;
    require tsm;
    submit;

    **Temporary work arrays used in ARIMA spec;
    array diff[2]/nosymbols;
    array ma[1]/nosymbols;

    *** Declare a collector object for fit statistics ***;
    declare object fitstats(tsmstat);

    *** Perform analysis based on airline model ***;
    declare object airModel(tsm);
    declare object airSpec(arimaspec);

    **Set up the airline model spec:**;
    ** Model: log(air) ~ (0,1,1)(0,1,1)_{12} noint *;**
    rc = airSpec.Open( );
    *** Specify differencing orders ***;
    diff[1] = 1;
    diff[2] = 12;
    rc = airSpec.SetDiff(diff,2);
    *** Specify moving average orders: q = (1)(12) ***;
    *** Use AddMAPoly twice for the two factors ***;
    ma[1] = 1;
    rc = airSpec.AddMAPoly(ma);
    ma[1] = 12;
    rc = airSpec.AddMAPoly(ma);
    *** Specify NOINT ***;
    rc = airSpec.SetOption('noint',1);
    *** Specify the log transform ***;
    rc = airSpec.SetTransform('log');
    *** Done setting up the ARIMA model ***;
    rc = airSpec.Close( );

    *** Set up and run the TSM object ***;
    rc = airModel.Initialize(airSpec);
```
rc = airModel.SetY(Air);
rc = airModel.SetOption('lead', 3);
rc = airModel.SetOption('back', 3);
rc = airModel.Run( );

*** Output airline model statistics of fit ***;
rc = fitstats.Collect(airModel);

*** Perform analysis based on ESM model ***;
declare object esmModel(tsm);
declare object esmSpec(esmspec);
rc = esmspec.Open( );
rc = esmspec.SetOption('method', 'winters');
rc = esmspec.Close( );

*** Set up and run the TSM object ***;
rc = esmModel.Initialize(esmspec);
rc = esmModel.SetY(Air);
rc = esmModel.SetOption('lead', 3);
rc = esmModel.SetOption('back', 3);
rc = esmModel.Run( );

*** Output ESM model statistics of fit ***;
rc = fitstats.Collect(esmModel);

declare object airCFC(cfc);
rc = airCFC.Initialize( );
rc = airCFC.SetY(Air);
rc = airCFC.AddModel(airModel);
rc = airCFC.AddModel(esmModel);
rc = airCFC.SetOption('weight', 'ols');
rc = airCFC.SetOption('lead', 3);
rc = airCFC.SetOption('back', 3);
rc = airCFC.Run( );

*** Output combined model statistics of fit ***;
rc = fitstats.Collect(airCFC);
endsubmit;
quit;

Output 14.5.1 shows the value of the root mean square error (RMSE) fit statistic for the airline, ESM, and forecast combination models over both the fit and forecast regions. Lower RMSE values indicate a more accurate model. Notice how the combined model forecast improves in accuracy over both of the independent forecasts in both the fit and forecast regions.
Output 14.5.1  Fit Statistics for the Airline, ESM, and Forecast Combination Models

**Root Mean Square Error (RMSE) Statistic**

<table>
<thead>
<tr>
<th>MODEL</th>
<th>REGION</th>
<th>RMSE</th>
</tr>
</thead>
<tbody>
<tr>
<td>airModel</td>
<td>FIT</td>
<td>10.7047</td>
</tr>
<tr>
<td>airModel</td>
<td>FORECAST</td>
<td>8.7732</td>
</tr>
<tr>
<td>esmModel</td>
<td>FIT</td>
<td>10.5826</td>
</tr>
<tr>
<td>esmModel</td>
<td>FORECAST</td>
<td>10.6018</td>
</tr>
<tr>
<td>airCFC</td>
<td>FIT</td>
<td>10.3719</td>
</tr>
<tr>
<td>airCFC</td>
<td>FORECAST</td>
<td>8.6551</td>
</tr>
</tbody>
</table>

References

## Chapter 15

Time Series Motif Discovery Package

### Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Overview: MTF Package</td>
<td>572</td>
</tr>
<tr>
<td>MTF Package Summary</td>
<td>572</td>
</tr>
<tr>
<td>Using the MTF Package</td>
<td>573</td>
</tr>
<tr>
<td>MTFBF Object</td>
<td>574</td>
</tr>
<tr>
<td>MTFBF Synopsis</td>
<td>574</td>
</tr>
<tr>
<td>MTFBF Methods</td>
<td>574</td>
</tr>
<tr>
<td>MTFPM Object</td>
<td>576</td>
</tr>
<tr>
<td>MTFPM Synopsis</td>
<td>576</td>
</tr>
<tr>
<td>MTFPM Methods</td>
<td>577</td>
</tr>
<tr>
<td>MTFSCORE Object</td>
<td>578</td>
</tr>
<tr>
<td>MTFSCORE Synopsis</td>
<td>579</td>
</tr>
<tr>
<td>MTFSCORE Methods</td>
<td>579</td>
</tr>
<tr>
<td>MTFANOM Object</td>
<td>581</td>
</tr>
<tr>
<td>MTFANOM Synopsis</td>
<td>581</td>
</tr>
<tr>
<td>MTFANOM Methods</td>
<td>581</td>
</tr>
<tr>
<td>OUTMTF Object</td>
<td>583</td>
</tr>
<tr>
<td>OUTMFT Synopsis</td>
<td>583</td>
</tr>
<tr>
<td>OUTMFT Methods</td>
<td>584</td>
</tr>
<tr>
<td>OUTMTFPM Object</td>
<td>584</td>
</tr>
<tr>
<td>OUTMTFPM Synopsis</td>
<td>585</td>
</tr>
<tr>
<td>OUTMTFPM Methods</td>
<td>585</td>
</tr>
<tr>
<td>OUTMTFANOM Object</td>
<td>585</td>
</tr>
<tr>
<td>OUTMTFANOM Synopsis</td>
<td>586</td>
</tr>
<tr>
<td>OUTMTFANOM Methods</td>
<td>586</td>
</tr>
<tr>
<td>OUTMTFSERIES Object</td>
<td>587</td>
</tr>
<tr>
<td>OUTMTFSERIES Synopsis</td>
<td>587</td>
</tr>
<tr>
<td>OUTMTFSERIES Methods</td>
<td>588</td>
</tr>
<tr>
<td>OUTMTFSCORE Object</td>
<td>588</td>
</tr>
<tr>
<td>OUTMTFSCORE Synopsis</td>
<td>589</td>
</tr>
<tr>
<td>OUTMTFSCORE Methods</td>
<td>589</td>
</tr>
<tr>
<td>Missing Values</td>
<td>589</td>
</tr>
<tr>
<td>Details</td>
<td>590</td>
</tr>
<tr>
<td>Motif Discovery, Brute-Force Method</td>
<td>590</td>
</tr>
<tr>
<td>Motif Discovery, Probabilistic Model Method</td>
<td>591</td>
</tr>
<tr>
<td>Motif-Based Subsequence Anomaly Detection</td>
<td>593</td>
</tr>
</tbody>
</table>
Overview: MTF Package

Time series motifs are frequent patterns or repeated subsequences in temporal data. Discovering motifs help you understand and interpret important characteristics of temporal data. These motifs are primitive shapes and implicit rules of time series data. Because motifs are extracted time series features, they can be used for time series association, classification, clustering, and anomaly detection. Motifs are especially useful for various internet of things (IoT) data analyses, including sequence matching from biomedical devices and recognition of activities or gestures from body-worn sensors. The time series motif package with PROC TSMODEL provides motif discovery functional objects that do the following:

- motif discovery by using a brute-force method
- motif discovery by using a probabilistic method based on a temporal topic model
- given a target motif, motif scoring that finds its motif instance occurrences in a new sequence
- motif-based subsequence anomaly detection

MTF Package Summary

Table 15.1 summarizes the objects in the MTF package.

<table>
<thead>
<tr>
<th>Object</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Computational Objects</strong></td>
<td></td>
</tr>
<tr>
<td>MTFBF</td>
<td>Brute-force method</td>
</tr>
<tr>
<td>MTFPM</td>
<td>Probabilistic method</td>
</tr>
<tr>
<td>MTFSCORE</td>
<td>Given a motif, search for its instances</td>
</tr>
<tr>
<td>MTFANOM</td>
<td>Subsequence anomaly detection</td>
</tr>
<tr>
<td><strong>Collector Objects</strong></td>
<td></td>
</tr>
<tr>
<td>OUTMTF</td>
<td>Collect motif output</td>
</tr>
</tbody>
</table>
Table 15.1  

<table>
<thead>
<tr>
<th>Object</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OUTMTFPM</td>
<td>Collect motif probabilistic output</td>
</tr>
<tr>
<td>OUTMTFANOM</td>
<td>Collect anomaly detection output</td>
</tr>
<tr>
<td>OUTMTFSERIES</td>
<td>Collect motif series output</td>
</tr>
<tr>
<td>OUTMTFSCORE</td>
<td>Collect score output</td>
</tr>
</tbody>
</table>

Using the MTF Package

The following code provides an outline of how to use the motif package:

```plaintext
proc tsmodeled data=InputDataSetName outobj=(of=OutDataSetName);
var InputVarName;
id TimeIDVarName;
require mtf;
submit;
    declare object f(a computational object);
    declare object of(a collector object);
    rc = f.Initialize();
    rc = f.SetX(InputVarName);
    rc = f.SetOption("option1", option1_numeric_value,
                     "option2", "option2_char_value",
                     ...);
    rc = f.Run();
    rc = of.Collect(the declared computational Object);
endsubmit;
run;
```

The basic execution pattern follows this sequence of operations:

1 **Declare**: Create computational and collector objects by using the object declaration (DECLARE) statement.

2 **Initialize**: Add a default model specification to the computational object.

3 **Specify variables**: Specify time series variables by using SetX and SetY methods.

4 **Specify option**: Specify model options and properties as appropriate by using the SetOption method.

5 **Run**: Execute the model in the computational object to produce motifs and their instances.

6 **Collect Results**: Extract the result by using a collector object.
MTFBF Object

The MTFBF object executes a brute-force method for motif discovery. The object declaration statement creates a new object, \( \textit{obj} \), of type MTFBF. Upon declaration, the MTFBF object has a brute-force method model.

Table 15.2 summarizes the methods that are associated with the MTFBF object.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initialize</td>
<td>Initialize the MTFBF object</td>
</tr>
<tr>
<td>SetX</td>
<td>Specify a time series array for the MTFBF object</td>
</tr>
<tr>
<td>SetOption</td>
<td>Specify options for the MTFBF object</td>
</tr>
<tr>
<td>Run</td>
<td>Run the MTFBF object</td>
</tr>
</tbody>
</table>

MTFBF Synopsis

DECLARE OBJECT \( \textit{obj} \) (MTFBF) ;

Method syntax, in order of typical usage:

\( \text{rc=} \textit{obj}.\text{Initialize} () ; \)
\( \text{rc=} \textit{obj}.\text{SetX} (X\text{Series}) ; \)
\( \text{rc=} \textit{obj}.\text{SetOption} ('\textit{Name}', 'Value<', 'Name', 'Value') ; \)
\( \text{rc=} \textit{obj}.\text{Run} () ; \)

MTFBF Methods

MTFBF.Initialize Method

\( \text{rc=} \textit{obj}.\text{Initialize} () ; \)

Initializes an MTFBF object with default parameters. This method must be called before specifying the time series \( X \) and other attributes for the MTFBF object.

Input Arguments
There are no arguments associated with this method.
**MTFBF.Run Method**

```c
rc=obj.Run();
```

Runs the MTFBF object to find motifs in the specified time series by using a brute-force method. Upon successful completion, motifs and their starting points can be extracted from the MTFBF object.

**Input Arguments**

There are no arguments associated with this method.

---

**MTFBF.SetOption Method**

```c
rc=obj.SetOption('Name',Value < ,'Name',Value,...);
```

Specifies options for the MTFBF object.

**Input Arguments**

You must specify one or more of the following 'Names' and its associated Value:

- **'DISTMARGIN'**
  takes a nonnegative numeric Value that specifies a distance margin factor to apply to searching for additional motif instances with a pair of motif instance candidates. The larger the Value you specify, the more motif instances that are produced. The default value is 0.01.

- **'MOTIFLENGTH'**
  takes a positive integer Value greater than 2 that specifies the length of each motif instance. The default value is 10.

- **'NMOTIF'**
  takes a positive integer Value that specifies the number of the motif. The Value should not be greater than 20. The default value is 1.

- **'NORMALIZE'**
  takes a string Value that specifies whether the series should be normalized. You can specify one of the following Values:

    - **YES|Y** normalizes all subsequences before their pairwise distance calculation.
    - **NO|N** does not normalize all subsequences before their pairwise distance calculation.

  The default is YES.

- **'OVERLAP'**
  takes a nonnegative numeric Value greater than or equal to 0 and less than 1 that specifies the subsequence overlap allowance rate. The default value is 0, which means that no overlap is allowed. A value of 1 means complete overlap, but is not allowed. The actual offset value is calculated from: offset=round(MOTIFLENGTH × (1 – OVERLAP)).
MTFBF.SetX Method

\[ rc = \text{obj}.\text{SetX}(XSeries); \]

Specifies the input time series, \( XSeries \), for the MTFBF object.

**Input Arguments**

You must specify the following input argument:

\( XSeries \) takes a numeric array that specifies an input time series for the MTFBF object.

---

**MTFPM Object**

The MTFPM computational object executes a probabilistic motif discovery method that is based on a temporal topic model. The object declaration statement creates a new object, \( \text{obj} \), of type MTFPM. Upon declaration, the MTFPM object has a default probabilistic model.

Table 15.3 summarizes the methods that are associated with the MTFPM object.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initialize</td>
<td>Initialize the MTFPM object</td>
</tr>
<tr>
<td>SetX</td>
<td>Specify a time series array for the MTFPM object</td>
</tr>
<tr>
<td>SetOption</td>
<td>Specify options for the MTFPM object</td>
</tr>
<tr>
<td>Run</td>
<td>Run the MTFPM object</td>
</tr>
</tbody>
</table>

---

**MTFPM Synopsis**

```
DECLARE OBJECT obj (MTFPM);
```

Method syntax, in order of typical usage:

\[ rc = \text{obj}.\text{Initialize}(); \]
\[ rc = \text{obj}.\text{SetX}(XSeries); \]
\[ rc = \text{obj}.\text{SetOption}('Name', Value < 'Name', Value>); \]
\[ rc = \text{obj}.\text{Run}(); \]
MTFPM Methods

**MTFPM.Initialize Method**

\[ rc = \text{obj}.\text{Initialize}() ; \]

Initializes an MTFPM object with default parameters. This method must be called before specifying the time series \( X \) and other attributes for the MTFPM object.

**Input Arguments**

There are no arguments associated with this method.

**MTFPM.Run Method**

\[ rc = \text{obj}.\text{Run}() ; \]

Runs the MTFPM object to find motifs at the specified time series by using a probabilistic model. Upon successful completion, motifs and their instance starting time points can be extracted from the MTFPM object.

**Input Arguments**

There are no arguments associated with this method.

**MTFPM.SetOption Method**

\[ rc = \text{obj}.\text{SetOption}(\text{'Name'}, \text{Value} < \text{'Name'}, \text{Value}, \ldots >) ; \]

Specifies options for the MTFPM object.

**Input Arguments**

You must specify one or more of the following ‘Names’ and its associated Value:

- **'CONVCRIT'**
  takes a positive numeric Value that specifies the convergence criterion for the EM algorithm. The default value is 0.001.

- **'CUTOFFPROB'**
  takes a numeric Value that specifies the minimum probability of starting time points to be included in each motif instance set. The default value is 0.

- **'MAXITER'**
  takes a positive integer Value less than 1,000,000 that specifies the maximum number of iterations for the EM algorithm. The default value is 200.

- **'MOTIFLENGTH'**
  takes a positive integer Value greater than 2 that specifies the length of each motif instance. The default value is 10.

- **'NBREAKPOINT'**
  takes a positive integer Value between 1 and 20 that specifies the number of quintile-based split points for symbolizing the time series (a Y-axis discretization). The default value is 10.

- **'NMOTIF'**
  takes a positive integer Value that specifies the number of motif. The Value should not be greater than 20. The default value is 1.

- **'SEED'**
  takes a nonnegative integer Value between 0 and 9,999,999,999, inclusive, that specifies a random number seed for the expectation and maximization (EM) algorithm. The default value is 0, which means a clock-generated seed.
takes a numeric nonnegative integer $k$, that requests that the top $k$ motif instances in each motif set be output. Regardless of the value of the CUTOFF-PROB argument, the first top instance of each motif is always output. The default value is 100.

MTFPM.SetX Method

```c
rc = obj.SetX (XSeries);
```

Specifies the input time series, $XSeries$, for the MTFPM object.

**Input Arguments**

You must specify the following input argument:

- $XSeries$ takes a numeric array that specifies an input time series for the MTFPM object.

MTFSCORE Object

The MTFSCORE object executes the scoring action that, given a motif sequence, finds motif instances in new sequences. The object declaration statement creates a new object, $obj$, of type MTFSCORE. Upon declaration, the MTFSCORE object has a default scoring model.

Table 15.4 summarizes the methods that are associated with the MTFSCORE object.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initialize</td>
<td>Initialize the MTFSCORE object</td>
</tr>
<tr>
<td>SetY</td>
<td>Specify a target time series array for the MTFSCORE object</td>
</tr>
<tr>
<td>SetX</td>
<td>Specify a time series array for the MTFSCORE object</td>
</tr>
<tr>
<td>SetOption</td>
<td>Specify options for the MTFSCORE object</td>
</tr>
<tr>
<td>Run</td>
<td>Run the MTFSCORE object</td>
</tr>
</tbody>
</table>

Table 15.4 Methods of the MTFSCORE Object
MTFSCORE Synopsis

DECLARE OBJECT obj (MTFSCORE) ;

Method syntax, in order of typical usage:

rc=obj.Initialize () ;
rc=obj.SetX (XSeries) ;
rc=obj.SetY (YSeries) ;
rc=obj.SetOption ('Name',Value <,'Name',Value>) ;
rc=obj.Run () ;

MTFSCORE Methods

MTFSCORE.Initialize Method

rc=obj.Initialize () ;

Initializes an MTFSCORE object with default parameters. This method must be called before specifying the time series X and Y, and other attributes for the MTFSCORE object.

Input Arguments
There are no arguments associated with this method.

MTFSCORE.Run Method

rc=obj.Run () ;

Runs the MTFSCORE object to find motif instances at the specified time series by measuring the distance between the specified target series and motif instance candidates. The candidates are all the subsequences that have the same motif length. Upon successful completion, the best matching instances with starting points can be extracted from the MTFSCORE object.

Input Arguments
There are no arguments associated with this method.

MTFSCORE.SetOption Method

rc=obj.SetOption ('Name', Value <,'Name',Value,...>) ;

Specifies options for the MTFSCORE object.

Input Arguments
You must specify one or more of the following 'Names' and its associated Value:

'MAXMOTIFDIST' takes a positive numeric Value greater than 0 that specifies the maximum distance between the specified target motif series and motif instances to be found. The default value is 10,000.

'MOTIFLENGTH' takes a positive integer Value greater than 2 that specifies the length of each motif instance. The default value is 10.
Chapter 15: Time Series Motif Discovery Package

'NORMALIZE' takes a string Value that specifies whether the series should be normalized. You can specify one of the following values:

YES|Y normalizes all subsequences before their pairwise distance calculation.

NO|N does not normalize all subsequences before their pairwise distance calculation.

The default is YES.

'OVERLAP' takes a nonnegative numeric Value greater than or equal to 0 and less than 1 that specifies the subsequence overlap allowance rate. The default value is 0, which means that no overlap is allowed. A value of 1 means the complete overlap, but is not allowed. The actual offset value is calculated from: offset = round(MOTIFLENGTH \times (1 – OVERLAP)).

'TOPK' takes a positive integer Value that specifies the maximum number of motif instances to be found. The default value is 100.

MTFSCORE.SetX Method

\[ rc = obj.SetX (XSeries) ; \]

Specifies the input time series, XSeries, for the MTFSCORE object.

**Input Arguments**

You must specify the following input argument:

XSeries takes a numeric array that specifies an input time series for the MTFSCORE object.

MTFSCORE.SetY Method

\[ rc = obj.SetY (YSeries) ; \]

Specifies the target time series, YSeries, for the MTFSCORE object.

**Input Arguments**

You must specify the following input argument:

YSeries takes a numeric array that specifies a target time series for the MTFSCORE object.
MTFANOM Object

The MTFANOM object executes a motif-based subsequence anomaly detection action, which finds anomaly subsequences in a specified input sequence. The object declaration statement creates a new object, obj, of type MTFANOM. Upon declaration, the MTFANOM object has a default anomaly detection model.

Table 15.5 summarizes the methods that are associated with the MTFANOM object.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initialize</td>
<td>Initialize the MTFANOM object</td>
</tr>
<tr>
<td>SetX</td>
<td>Specify the input time series array for the MTFANOM object</td>
</tr>
<tr>
<td>SetOption</td>
<td>Specify options for the MTFANOM object</td>
</tr>
<tr>
<td>Run</td>
<td>Run the MTFANOM object</td>
</tr>
</tbody>
</table>

MTFANOM Synopsis

DECLARE OBJECT obj (MTFANOM) ;

Method syntax, in order of typical usage:

```
rc=obj.Initialize () ;
rc=obj.SetX (XSeries ) ;
rc=obj.SetOption ('Name',Value < ,'Name',Value>) ;
rc=obj.Run () ;
```

MTFANOM Methods

MTFANOM.Initialize Method

```
rc=obj.Initialize () ;
```

Initializes an MTFANOM object with default parameters. This method must be called before specifying the time series X and other attributes for the MTFANOM object.

Input Arguments

There are no arguments associated with this method.
MTFANOM.Run Method

```plaintext
rc = obj.Run();
```

Runs the MTFANOM object to find anomaly subsequences from the specified time series. The candidates are all possible subsequences. Upon successful completion, the starting points of the top \( k \) anomaly subsequences can be extracted from the MTFOUTANOM object.

**Input Arguments**

There are no arguments associated with this method.

MTFANOM.SetOption Method

```plaintext
rc = obj.SetOption('Name', Value <, 'Name', Value, .. >);
```

Specifies options for the MTFANOM object.

**Input Arguments**

You must specify one or more of the following `Names` and its associated `Value`:

- **'LENGTH'** takes a numeric `Value` greater than 2 that specifies the length of anomaly subsequence. The default value is 10.
- **'NORMALIZE'** takes a string `Value` that specifies whether the series should be normalized. You can specify one of the following values:
  - **YES|Y** normalizes all subsequences before their pairwise distance calculation.
  - **NO|N** does not normalize all subsequences before their pairwise distance calculation.

  The default is YES.

- **'OVERLAP'** takes a numeric `Value` between 0 and 1 that specifies the subsequence overlap allowance rate. The default value is 0, which means that no overlap is allowed. The actual offset value is calculated from: \( \text{offset} = \text{round}(\text{LENGTH} \times (1 - \text{OVERLAP})) \).
- **'TOPK'** takes a numeric nonnegative integer `Value` that specifies the number of anomaly subsequences to be found. The default value is 100.

MTFANOM.SetX Method

```plaintext
rc = obj.SetX(XSeries);
```

Specifies the input time series, `XSeries`, for the MTFANOM object.

**Input Arguments**

You must specify the following input argument:

- **XSeries** takes a numeric array that specifies an input time series for the MTFANOM object.
OUTMTF Object

The OUTMTF object collects motifs and their instances from an MTFBF or MTFPM object and stores them in a CAS table for printing or archiving. The object declaration statement creates a new object, \( \textit{obj} \), of OUTMTF.

Table 15.6 summarizes the methods that are associated with the OUTMTF object.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Collect</td>
<td>Collects motifs and their instances from an MTFBF or MTFPM object</td>
</tr>
</tbody>
</table>

Table 15.7 shows the contents of the OUTMTF object.

<table>
<thead>
<tr>
<th>Column</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>String</td>
<td>Name of the input time series</td>
</tr>
<tr>
<td>MotifID</td>
<td>Numeric</td>
<td>Motif identification number</td>
</tr>
<tr>
<td>StartPosition</td>
<td>Numeric</td>
<td>Starting position of each motif instance of the motif</td>
</tr>
<tr>
<td>TimeID</td>
<td>Numeric</td>
<td>Starting time of each motif instance of the motif; this is the variable name</td>
</tr>
<tr>
<td></td>
<td></td>
<td>that is specified in the ID statement in the TSMODEL procedure</td>
</tr>
<tr>
<td>Distance</td>
<td>Numeric</td>
<td>Distance between each motif instance and the representative motif series</td>
</tr>
</tbody>
</table>

OUTMFT Synopsis

\[
\text{DECLARE OBJECT} \ \textit{obj} \ (\text{OUTMTF}) ;
\]

Method syntax:

\[
rc=\textit{obj}.Collect (\textit{MTFObj}) ;
\]
OUTMTF Methods

OUTMTF.Collect Method

```plaintext
rc = obj.Collect (MTFObj);
```

Retrieves the results of time series motif discovery run actions from an MTFBF or MTFPM object, `MTFObj`, and stores them in a CAS table.

**Input Arguments**

You must specify the following input argument:

- `MTFObj`: takes an MTFBF or MTFPM object to use as the source of time series motif output.

OUTMTFPM Object

The OUTMTFPM collector object collects the probabilities of motifs and their instance occurrences from an MTFPM object, `MTFPMObj`, and stores them in a CAS table. The object declaration statement creates a new object, `obj`, of type OUTMTFPM.

Table 15.8 summarizes the methods that are associated with the OUTMTFPM object.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Collect</td>
<td>Collects motifs, their instance starting points, and their occurrence probabilities from an MTFPM object</td>
</tr>
</tbody>
</table>

Table 15.9 shows the contents of the OUTMTFP object.

<table>
<thead>
<tr>
<th>Column</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>String</td>
<td>Name of the input time series</td>
</tr>
<tr>
<td>MotifID</td>
<td>Numeric</td>
<td>Motif identification number</td>
</tr>
<tr>
<td>StartPosition</td>
<td>Numeric</td>
<td>Starting position for each instance of the motif</td>
</tr>
<tr>
<td>TimeID</td>
<td>Numeric</td>
<td>Starting time for each instance of the motif; this is the variable name that is specified in the ID statement in the TSMODEL procedure</td>
</tr>
<tr>
<td>ProbMotif</td>
<td>Numeric</td>
<td>Probability of the motif occurrence</td>
</tr>
<tr>
<td>ProbMotifStart</td>
<td>Numeric</td>
<td>Probability of each starting time point given the motif</td>
</tr>
</tbody>
</table>
OUTMTFPM Synopsis

DECLARE OBJECT obj (OUTMTFPM) ;

Method syntax:

\[ rc = obj.Collect (MTFPMObj) ; \]

OUTMTFPM Methods

OUTMTFPM.Collect Method

\[ rc = obj.Collect (MTFPMObj) ; \]

Retrieves the probabilistic results of a run method of an MTFPM object. The Collect method stores motifs, their instance starting points, and their occurrence probabilities.

**Input Arguments**

You must specify the following input argument.

*MTFPMObj* takes an MTFPM object to use as the output source of a time series motif.

OUTMTFANOM Object

The OUTMTFANOM object collects the starting points of detected anomaly subsequences and their anomaly ranking from a Run method of an MTFANOM object, *MTFANOMObj*, and stores them in a CAS table. The object declaration statement creates a new object, *obj*, of type OUTMTFANOM.

Table 15.10 summarizes the methods that are associated with the OUTMTFANOM object.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Collect</td>
<td>Collects anomaly subsequences and their starting points from an MTFANOM object</td>
</tr>
</tbody>
</table>

Table 15.11 shows the contents of the OUTMTFANOM object.
### Table 15.11 Contents of the OUTMTFANOM Object

<table>
<thead>
<tr>
<th>Column</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>String</td>
<td>Name of the input time series</td>
</tr>
<tr>
<td>AnomalyRank</td>
<td>Numeric</td>
<td>Anomaly identification number</td>
</tr>
<tr>
<td>StartPosition</td>
<td>Numeric</td>
<td>Starting position for each anomaly subsequence</td>
</tr>
<tr>
<td>TimeID</td>
<td>Numeric</td>
<td>Starting time for each anomaly subsequence; this is the variable name that is specified in the ID statement in the TSMODEL procedure</td>
</tr>
<tr>
<td>Distance</td>
<td>Numeric</td>
<td>Distance instance used for the anomaly detection</td>
</tr>
</tbody>
</table>

---

**OUTMFTANOM Synopsis**

```
DECLARE OBJECT obj (OUTMFTANOM) ;
```

Method syntax:

```
rc=obj.Collect (MTFANOMObj) ;
```

---

**OUTMFTANOM Methods**

**OUTMFTANOM.Collect Method**

```
rc=obj.Collect (MTFANOMObj) ;
```

Retrieves the anomaly detection results of the Run method of an `MTFANOMObj`. The Collect method stores the anomaly rank ID, each anomaly instance starting position, and the distance used for the detection.

**Input Arguments**

You must specify the following input argument.

- `MTFANOMObj` takes an MTFANOM object to use as the source of the output of time series anomaly detection.
OUTMTFSERIES Object

The OUTMTFSERIES object collects the representative motif series from each motif instance set, which is obtained from a Run method of an MTFBF or MTFPM object (MTFBObj or MTFPMObj) and stores them in a CAS table. The object declaration statement creates a new object, obj, of type OUTMTFSERIES.

Table 15.12 summarizes the methods that are associated with the OUTMTFSERIES object.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Collect</td>
<td>Collects the representative motif series for each motif from an MTFBF or MTFPM object</td>
</tr>
</tbody>
</table>

Table 15.13 shows the contents of the OUTMTFSERIES object.

<table>
<thead>
<tr>
<th>Column</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>String</td>
<td>Name of the input time series</td>
</tr>
<tr>
<td>MotifID</td>
<td>Numeric</td>
<td>Motif identification</td>
</tr>
<tr>
<td>MotifSeries</td>
<td>Numeric</td>
<td>Motif representative series</td>
</tr>
</tbody>
</table>

OUTMTFSERIES Synopsis

DECLARE OBJECT obj (OUTMTFSERIES) ;

Method syntax:

rc=obj.Collect (MTFObj) ;
OUTMTFSERIES Methods

OUTMTFSERIES.Collect Method

\[
rc = \text{obj}.\text{Collect}(\text{MTFObj})
\]

Retrieves the representative motif series from a Run method of an MTFBF or MTFPM object.

**Input Arguments**

You must specify the following input argument.

\textit{MTFObj} takes an MTFBF or MTFPM object to use as the source of the output of a representative motif series.

OUTMTFSCORE Object

The OUTMTFSCORE object collects subsequences that are similar to the corresponding motif sequence from an MTFSCORE object and stores them in a CAS table for printing or archiving. The object declaration statement creates a new object, \textit{obj}, of type OUTMTFSCORE.

Table 15.14 summarizes the methods that are associated with the OUTMTFSCORE object.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Collect</td>
<td>Collects subsequences from MTFSCORE object</td>
</tr>
</tbody>
</table>

Table 15.15 shows the contents of the OUTMTFSCORE object.

<table>
<thead>
<tr>
<th>Column</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>String</td>
<td>Name of the input time series</td>
</tr>
<tr>
<td>Rank</td>
<td>Numeric</td>
<td>Similarity rank of subsequences to the corresponding motif</td>
</tr>
<tr>
<td>StartPosition</td>
<td>Numeric</td>
<td>Starting position for each found subsequence</td>
</tr>
<tr>
<td>TimeID</td>
<td>Numeric</td>
<td>Starting time for each found subsequence; this is the variable name that is specified in the ID statement in the TSMODEL procedure</td>
</tr>
<tr>
<td>Distance</td>
<td>Numeric</td>
<td>Distance between each found subsequence and the corresponding motif series</td>
</tr>
</tbody>
</table>
OUTMTFSCORE Synopsis

DECLARE OBJECT obj (OUTMTFSCORE) ;

Method syntax:

\[ rc = obj.\text{Collect} (MTFSCOREObj) ; \]

OUTMTFSCORE Methods

OUTMTFSCORE.Collect Method

\[ rc = obj.\text{Collect} (MTFSCOREObj) ; \]

Retrieves the results of score action from an MTFSCORE object, \( MTFSCOREObj \), and stores them in a CAS table.

**Input Arguments**

You must specify the following input argument.

\( MTFSCOREObj \) takes an MTFSCORE object to use as the source of scoring output.

Missing Values

The MTF package handles missing values as follows:

- For an input time series, the MTF package automatically replaces any missing value with the average value of the time series when the percentage of missing observations is less than 10%.

- For a target time series, the MTF package returns an error if there are any missing values within the specified motif length.
Details

Motif Discovery, Brute-Force Method

A brute-force method searches for motifs from all possible comparisons of subsequences. Brute-force methods are computationally expensive, but they are more accurate than other methods. This section explains a slightly modified brute-force method.

Define a time series \( Y = \{y_t\}_{t=1,2,...,T} \). If you set the motif length to \( m \), you can get \((T - m + 1)\) subsequences from the time series. Define \( S = \{S_i|i = 1,2,...,T - m + 1\} \) as the collection of the subsequences. Simply speaking, motif discovery is the process of finding the most similar subsequences from \( S \). When Euclidean distance is used as a similarity measure, the distance between \( S_i = (y_{i1},y_{i2},...,y_{im}) \) and \( S_j = (y_{j1},y_{j2},...,y_{jm}) \) is

\[
D_{ij} = \sqrt{\sum_{k=1}^{m} (y_{ik} - y_{jk})^2}
\]

So the distances for all pairs of subsequences are defined as a \( D \) matrix:

\[
D = \{D_{ij}\}_{i,j=1,...,(T-m+1)}
\]

However, in practice, some potential trivial matches with a threshold of \( \delta \) are ignored:

\[
D = \{D_{ij}\}_{i,j=1,...,(T-m+1), \text{ and } j > i + \delta}
\]

A commonly used \( \delta \) value is the specified motif length \( m \). The distances between subsequences \( \{D_{ij}\} \) are sorted, and the smallest distance, \( D_{i*}j* \), identifies a motif whose instances occur at \( i^\ast \) and \( j^\ast \) time points. So the subsequences \( S_{i^\ast} \) and \( S_{j^\ast} \) are the base instances of the motif.

Once you have a pair of subsequences as base instances, you further investigate some other motif instances by using a tolerance factor \( \tau > 0 \) at \( i^\ast \) and \( j^\ast \). For a particular \( i^\ast \), you calculate \( \{D_{i^\ast}j\} \) for \( j = 1,...,(T - m + 1) \) and \( j > i^\ast + \delta \); if \( D_{i^\ast}j < (1 + \tau)D_{i^\ast}j^\ast \), then \( S_j \) is an occurrence of the motif. Similarly, for a particular \( j^\ast \), calculate \( \{D_{ij^\ast}\} \) for \( i = 1,...,(T - m + 1) \) and \( i > j^\ast + \delta \); if \( D_{ij^\ast} < (1 + \tau)D_{i^\ast}j^\ast \), then \( S_i \) is an occurrence of the motif. When \( \tau = 0 \), the motif set includes only the selected pair of subsequences: \( S_{i^\ast} \) and \( S_{j^\ast} \). When \( \tau \) increases, the number of motif instances increases, but the accuracy decreases.
Define a time series, $Y = \{y_1, y_2, \ldots, y_T\}$ and convert it into a SAX (symbolic aggregate approximation) representation without using piecewise aggregate approximation (PAA). For more information about SAX time series discretization, see Lin et al. (2003). Suppose you have six quantile-based buckets $(a, b, c, d, e, f)$ of a standard normal distribution as shown at Figure 15.1. The value at each time point is converted to a bucket symbol such as $(a, b, c, d, e, f, a, b, c, d)$. The time series should be normalized before this mapping.

Figure 15.1 shows the setup for the probability that a SAX word $w$ appears at time point $t$, which is $p(w, t)$, $w \in \{a, b, c, d, e, f\}$. There is one motif $(a, b, c, b)$ for which two instances occur at time 1 and time 7.

The following formulas are a simplified version of Varadarajan, Emonet, and Odobez (2010b) that adapts a univariate time series into the temporal topic model. The notations are defined to set up the model:

- $z$ is a potential motif (topic).
- $p(z)$ is the probability that a latent motif $z$ appears in the time series.
- $w$ is a SAX word that appears in the motif $z$.
- $p(w|z)$ is the probability that a $w$ appears in the motif $z$.
- $t_s$ is the starting time, and $p(t_s|z)$ is the probability that a latent motif $z$ starts at time $t_s$. 

\[ \text{Figure 15.1 A Probabilistic Model Framework Example} \]
- $t_r$ is the relative time, and $p(t_r | z)$ is the probability that a $w$ appears at the time $t_r$ in the motif $z$.
- Define $t_a = t_s + t_r$ as the absolute time that a $w$ appears in the motif $z$.
- $n(w, t_a | z)$ is the number of times a $w$ at $z$ appears at a $t_a$ position (in other words, at a $t_r$ relative time of $z$). The count is always 1 for a univariate time series.

Then the joint probability distribution of $(w, t_a, z, t_s)$ is defined as follows:

$$p(w, t_a, z, t_s) = p(z) p(t_s | z) p(w | z) p(t_a - t_s | w, z)$$

Because the goal is to discover a motif and its starting time, you derive the log likelihood of the observed time series and estimate $\Theta$, which are the model parameters. In general, this could be done by maximizing the log likelihood, which is defined as

$$L(Y | \Theta) = \sum_{w=1}^{N_w} \sum_{t_a=1}^{T} n(w, t_a) \log \sum_{z=1}^{N_z} \sum_{t_s=1}^{T_s} p(w, t_a, z, t_s)$$

where $N_z$ is the number of motifs, $N_w$ is the number of all SAX words, and $T_s$ is the number of all possible time points at which the motif might start.

The preceding equation cannot be solved directly; the expectation-maximization (EM) algorithm approach is needed. The EM algorithm maximizes the expectation of the following complete log-likelihood equation instead of the preceding log-likelihood equation.

$$E[L] = \sum_{w=1}^{N_w} \sum_{t_a=1}^{T} \sum_{z=1}^{N_z} \sum_{t_s=1}^{T_s} n(w, t_a) p(z, t_s | w, t_a) \log p(w, t_a, z, t_s)$$

From the complete log-likelihood equation, the EM algorithm steps are derived as follows:

1. In the expectation step (E-step), the posterior distribution of $z$ and $t_s$ is obtained by

$$p(z, t_s | w, t_a) = \frac{p(z, t_s, w, t_a)}{p(w, t_a)}$$

where $p(w, t_a) = \sum_{z=1}^{N_z} \sum_{t_s=1}^{T_s} p(z, t_s, w, t_a)$

2. In the maximization step (M-step), the model parameters are updated according to the following posterior probability distributions:

$$p(z) \propto \sum_{t_s=1}^{T_s} \sum_{t_r=1}^{T_r} \sum_{w=1}^{N_w} n(w, t_s + t_r) p(z, t_s | w, t_s + t_r)$$

$$p(t_s | z) \propto \sum_{w=1}^{N_w} \sum_{t_r=1}^{T_r-1} n(w, t_s + t_r) p(z, t_s | w, t_s + t_r)$$
Motif-Based Subsequence Anomaly Detection

Motif-Based Subsequence Anomaly Detection

The anomaly subsequence is defined as the subsequence that has the largest distance to its nearest subsequence, excluding subsequences within the length of specified overlap. Define a time series \( Y = \{y_t\}_{t=1,2,\ldots,T} \). If you set the motif length to \( m \), you can get \((T - m + 1)\) subsequences from the time series. Define \( S \) as the collection of the subsequences, \( S = (S_1, S_2, \cdots, S(T-m+1)) \). If you consider only non-overlapping subsequences, the anomaly subsequence is searched by the following equation:

\[
\max_i \min_j [\text{Distance}(S_i, S_j)] |i - j| \geq m, i, j = 1, 2, \cdots, T - m + 1
\]

Examples: MTF Package

Throughout this section it is assumed that you have already started a CAS session and the data tables that are used in this section are in \textit{mycas}, a CAS library that you have necessary permissions to work with. In all the examples of this section, the functionality of the MTF package is illustrated using the TSMODEL procedure. This section assumes that you are familiar with the general workings of the TSMODEL procedure.

Using CAS Sessions and CAS Engine Librefs

SAS Cloud Analytic Services (CAS) is the analytic server and associated cloud services in SAS Viya. This section describes how to create a CAS session and set up a CAS engine libref that you can use to connect to the CAS session. It assumes that you have a CAS server already available; contact your system administrator if you need help starting and terminating a server. This CAS server is identified by specifying the host on which it runs and the port on which it listens for communications. To simplify your interactions with this CAS server, the host information and port information for the server are stored as SAS option values that are retrieved automatically whenever this CAS server needs to be accessed. You can examine the host and port values for the server at your site by using the following statements:
In addition to starting a CAS server, your system administrator might also have created a CAS session and a CAS engine libref for your use. You can define your own sessions and CAS engine librefs that connect to the CAS server as shown in the following statements:

```sas
proc options option=(CASHOST CASPORT);
rung;
```

The CAS statement creates the CAS session named `mysess`, and the LIBNAME statement creates the `mycas` CAS engine libref that you use to connect to this session. It is not necessary to explicitly name the CASHOST and CASPORT of the CAS server in the CAS statement, because these values are retrieved from the corresponding SAS option values.

If you have created the `mysess` session, you can terminate it by using the TERMINATE option in the CAS statement as follows:

```sas
cas mysess terminate;
```

For more information about the CAS statement and the LIBNAME statement, see SAS Cloud Analytic Services: User’s Guide. For general information about CAS and CAS sessions, see SAS Cloud Analytic Services: Fundamentals.

---

**Example 15.1: Brute-Force Method**

This example uses simulated data that are generated by the following SAS code as a simulated time series with a sine curve as a motif and background data that are from the standard normal distribution. The planted motif instances occur at positions 50, 150, and 250. Their starting times are 00:00:50, 00:02:30, and 00:04:10. The length of the motif is 10. The length of the time series is 300.

```sas
%let motif_length = 10;
%let sequence_length = 300;
%let motif_position = (50,150,250);
%let n_motifs = 3;

data SimuData;
  format time time8.0;
  array start {&n_motifs} &motif_position;
  array end {&n_motifs} &motif_position;
  call streaminit(123);
  do j = 1 to dim(start);
    end[j] = start[j] + &motif_length;
  end;
  do i = 1 to &sequence_length;
    time = i;
    signal = rand('NORMAL');
    do j = 1 to dim(start);
      if i >= start[j] and i<end[j] then do;
        signal = signal+10*sin((i-start[j])/&motif_length * (2*constant('pi')));
      end;
    end;
  end;
run;
```
Example 15.1: Brute-Force Method

Output 15.1.1 shows the plot of simulated data, which appear to contain three big sine curves around start positions 50, 150, and 250. Their corresponding starting times are 00:00:50, 00:02:30, and 00:04:10.

Output 15.1.1 Motif Simulated Data Plot

The brute-force method finds the exact three time points where the motif instances start, as shown at Output 15.1.3.

```plaintext
proc tsmodel data=mycas.SimuData
    outobj=(of=mycas.outmotif
        ofms = mycas.outmotifseries);
var signal;
id time interval=second format=time8.0;
require mtf;
submit;
declare object f(MTFBF);
declare object of(OUTMTF);
declare object ofms(OUTMTFSERIES);
rc = f.Initialize();
rc = f.SetX(signal);
rc = f.SetOption("NMOTIF", 1,
    "MOTIFLENGTH", 10,
    "NORMALIZE", "Y",
    "DISTMARGIN", 1 );
rc = f.Run();
rc = of.Collect(f);
```
rc = ofms.Collect(f);
endsubmit;
run;

**Output 15.1.2 Motif List, Brute-Force Method**

<table>
<thead>
<tr>
<th>Obs</th>
<th>Variable Name</th>
<th>Motif ID</th>
<th>Motif Start Position</th>
<th>Motif Start Time</th>
<th>Distance</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>signal</td>
<td>1</td>
<td>50</td>
<td>0:00:50</td>
<td>0.2740376908</td>
</tr>
<tr>
<td>2</td>
<td>signal</td>
<td>1</td>
<td>150</td>
<td>0:02:30</td>
<td>0.3228669954</td>
</tr>
<tr>
<td>3</td>
<td>signal</td>
<td>1</td>
<td>250</td>
<td>0:04:10</td>
<td>0.356297145</td>
</tr>
</tbody>
</table>

**Output 15.1.3 Motif Plot, Brute-Force Method**

Motif Representation from Motif Instances

Output 15.1.4 shows that table that was collected by the OUTMTFSERIES collector. The table contains the motif representative series and could be used as a target sequence for motif scoring.

**Output 15.1.4 Representative Motif Series**

<table>
<thead>
<tr>
<th>Obs</th>
<th>Variable Name</th>
<th>Motif ID</th>
<th>Motif Series</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>signal</td>
<td>1</td>
<td>0.5649936815</td>
</tr>
<tr>
<td>2</td>
<td>signal</td>
<td>1</td>
<td>6.1361722825</td>
</tr>
<tr>
<td>3</td>
<td>signal</td>
<td>1</td>
<td>10.158074349</td>
</tr>
<tr>
<td>4</td>
<td>signal</td>
<td>1</td>
<td>9.2295711121</td>
</tr>
<tr>
<td>5</td>
<td>signal</td>
<td>1</td>
<td>6.5244635657</td>
</tr>
<tr>
<td>6</td>
<td>signal</td>
<td>1</td>
<td>-0.474552533</td>
</tr>
<tr>
<td>7</td>
<td>signal</td>
<td>1</td>
<td>-5.931737613</td>
</tr>
<tr>
<td>8</td>
<td>signal</td>
<td>1</td>
<td>-5.965699933</td>
</tr>
<tr>
<td>9</td>
<td>signal</td>
<td>1</td>
<td>-5.965699933</td>
</tr>
<tr>
<td>10</td>
<td>signal</td>
<td>1</td>
<td>-5.965699933</td>
</tr>
</tbody>
</table>
Output 15.1.5 shows the motif representative series and its motif instances. The values at the representative series are average values of the three motif instance series at each time point.

\[
\text{data outmotifseries;}
\]
\[
\text{set mycas.outmotifseries;}
\]
\[
\text{run;}
\]

\textbf{Example 15.2: Motif Scoring}

Given a target sequence, motif scoring tries to find subsequences that are most similar to the target sequence in a new time series. This example uses other simulated data, called Scoredata, and the motif representative series that was found in Example 15.1 as a target series. The following code generates score data that contain two noisy sine curves at starting positions 100 (time 00:01:40) and 200 (time 00:03:20) and searches the target sequence in Scoredata by using a moving window:

\[
\%\text{let motif_length} = 10;
\]
\[
\%\text{let sequence_length} = 300;
\]
\[
\%\text{let motif_position} = (100, 200);
\]
\[
\%\text{let n_motifs} = 2;
\]

\textbf{data ScoreData;}
\[
\text{format time time8.0;}
\]
\[
\text{array start \{n_motifs\} \&motif_position;}
\]
\[
\text{array end \{n_motifs\} \&motif_position;}
\]
\[
\text{call streaminit(123);}
\]
\[
\text{do j = 1 to dim(start);}
\]
\[
\text{end[j] = start[j] + \&motif_length;}
\]
\[
\text{end;}
\]
\[
\text{do i = 1 to \&sequence_length;}
\]
time = i;
signal = rand('NORMAL');
do j = 1 to dim(start);
    if i >= start[j] and i<end[j] then do;
        signal=2*signal+5*sin((i-start[j])/&motif_length*(2*constant('pi')));
    end;
end;
output;
keep time signal;
run;

As you expect from the simulated score data, Output 15.2.1 shows two bumps near the starting positions 100 and 200.

The DATA steps in the following code merge the outmotifseries data set into the scoredata data set to create the score input data set.

```
data outmotifseries;
    format time time8.0;
    set outmotifseries;
    time = _N_;      
    keep time motifseries;
run;
data scoredata;
    merge scoredata outmotifseries;
    by time;
run;
data mycas.scoredata;
    set scoredata;
run;
```

The following code shows how to use the MTFSCORE and OUTMTFSCORE objects for motif scoring:
Example 15.2: Motif Scoring

The motif scoring finds that the motif instances start exactly at positions 100 and 200 to be the two topmost matched subsequences. The top four results are shown in table form in Output 15.2.2 and in graph form in Output 15.2.3. The distances between the target sequence and the found motif instances are also shown in Output 15.2.2.

Output 15.2.2  Find Motif Instances Given a Target Motif Series

<table>
<thead>
<tr>
<th>Obs</th>
<th>Variable Name</th>
<th>Motif Instance Rank</th>
<th>Motif Start Position</th>
<th>Motif Start Time</th>
<th>Distance</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>signal</td>
<td>1</td>
<td>100</td>
<td>0:01:40</td>
<td>0.9693128906</td>
</tr>
<tr>
<td>2</td>
<td>signal</td>
<td>2</td>
<td>200</td>
<td>0:03:20</td>
<td>1.1172916593</td>
</tr>
<tr>
<td>3</td>
<td>signal</td>
<td>3</td>
<td>75</td>
<td>0:01:15</td>
<td>1.4502947816</td>
</tr>
<tr>
<td>4</td>
<td>signal</td>
<td>4</td>
<td>277</td>
<td>0:04:37</td>
<td>1.8895528747</td>
</tr>
</tbody>
</table>
Example 15.3: Probabilistic Model Method

This example uses the same simulated data (the Simudata data set) that are used in Example 15.1. Therefore, you expect three motif instances of the sine curve motif at starting positions 50, 150, and 250. Their starting times are 00:00:50, 00:02:30, and 00:04:10. The following code requests four motif sets, each of which shows only the top five instances depending on the cutoff of the start probability:

```plaintext
proc tsmodel data=mycas.SimuData outlog = mycas.mylog
    outobj=( of=mycas.outmotif(replace=YES)
            ofpm=mycas.outmotifpm(replace=YES)
            ofms=mycas.outmotifseries(replace=YES));
var signal;
id time interval=second format=time8.0;;
require mtf;
submit;
declare object f(MTFPM);
declare object of(OUTMTF);
declare object ofpm(OUTMTFPM);
declare object ofms(OUTMTFSERIES);
rc = f.Initialize();
rc = f.SetX(signal);
rc = f.SetOption("NMOTIF", 4,
                   "MOTIFLENGTH", 10,
                   "NBREAKPOINT", 14,
                   "CONVRIT", 0.001,
                   "CUTOFFPROB", 0.1,
                   "MAXITER", 100,
                   "TOPK", 5,
                   "SEED", 12345
```
Example 15.4: Motif-Based Subsequence Anomaly Detection

The probabilistic model produces a posterior probability table, which is shown in Output 15.3.1. The table shows four motif sets and their instances with start position probabilities that are greater than the cutoff probability. The first instance for each motif is always shown in the table regardless of the cutoff value. When you look for motif instances that have the three largest start position probabilities, they all belong to the fourth motif and they have almost exactly the same starting positions (46, 144, and 243) as the simulated motif instances. The probabilistic model result varies depending on the initialization and model parameters. However, in general, the model first captures motifs that have many instances, but each instance has a small start position probability. Later the model obtains a small number of instances, but each instance has a larger start position probability. The latter condition is what you want for motif discovery. Usually, the number of motifs needs to be greater than or equal to 5 in order to capture the motif instances properly.

Output 15.3.1 Probabilistic Model: Motifs and Their Instances

<table>
<thead>
<tr>
<th>Obs</th>
<th>Variable</th>
<th>Motif ID</th>
<th>Motif Start Position</th>
<th>Motif Start Time</th>
<th>Motif Probability</th>
<th>Motif Start Probability</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>signal</td>
<td>1</td>
<td>256</td>
<td>0:04:16</td>
<td>0.298565969</td>
<td>0.1053295204</td>
</tr>
<tr>
<td>2</td>
<td>signal</td>
<td>2</td>
<td>291</td>
<td>0:04:51</td>
<td>0.2817893515</td>
<td>0.1052442557</td>
</tr>
<tr>
<td>3</td>
<td>signal</td>
<td>3</td>
<td>75</td>
<td>0:01:15</td>
<td>0.2546943832</td>
<td>0.0772922251</td>
</tr>
<tr>
<td>4</td>
<td>signal</td>
<td>4</td>
<td>243</td>
<td>0:04:03</td>
<td>0.1649502963</td>
<td>0.1717459314</td>
</tr>
<tr>
<td>5</td>
<td>signal</td>
<td>4</td>
<td>46</td>
<td>0:00:46</td>
<td>0.1649502963</td>
<td>0.1345415164</td>
</tr>
<tr>
<td>6</td>
<td>signal</td>
<td>4</td>
<td>144</td>
<td>0:02:24</td>
<td>0.1649502963</td>
<td>0.111272976</td>
</tr>
</tbody>
</table>

Example 15.4: Motif-Based Subsequence Anomaly Detection

This example shows a motif-based subsequence anomaly detection, which searches for the subsequences that have higher distances to any other subsequences. The example data are generated by using two anomaly subsequences being planted at the starting positions 101 (time 00:01:41) time and 201 (time 00:03:21), as in the following DATA step. The subsequence length is 10.

data SimuData2;
format time time8.0;
call streaminit(123);
do time=1 to 300;
   signal = rand('NORMAL');
   if (200 < time <= 210) then signal=signal+5;
   if (100 < time <= 110) then signal=signal-3;
output;
end;
run;
The following statements use PROC TS­MODE­L and the MTFANOM object to request the top five anomaly subsequences:

```plaintext
proc tsmodel data=mycas.SimuData2 outobj=(of = mycas.outanomaly(replace=YES));
var signal ;
id time interval = second format=time8.0;;
require mtf;
submit;
declare object f(MTFANOM);
declare object of(OUTMTFANOM);
rc = f.Initialize();
rc = f.SetX(signal);
rc = f.SetOption("LENGTH",10,
                "NORMALIZE","N",
                "TOPK", 5
                );
rc = f.run();
rc = of.Collect(f);
endsubmit;
run;
```

The table in Output 15.4.1 shows that the planted anomaly subsequences are discovered as the first and second anomaly sequences.

### Output 15.4.1 Output of Subsequence Anomaly Detection

<table>
<thead>
<tr>
<th>Obs</th>
<th>Variable Name</th>
<th>Anomaly Rank</th>
<th>Anomaly Start Position</th>
<th>Anomaly Start Time</th>
<th>Distance</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>signal</td>
<td>1</td>
<td>201</td>
<td>0:03:21</td>
<td>14.394201667</td>
</tr>
<tr>
<td>2</td>
<td>signal</td>
<td>2</td>
<td>101</td>
<td>0:01:41</td>
<td>8.5891954687</td>
</tr>
<tr>
<td>3</td>
<td>signal</td>
<td>3</td>
<td>236</td>
<td>0:03:56</td>
<td>3.7638112249</td>
</tr>
<tr>
<td>4</td>
<td>signal</td>
<td>4</td>
<td>275</td>
<td>0:04:35</td>
<td>3.0180637279</td>
</tr>
<tr>
<td>5</td>
<td>signal</td>
<td>5</td>
<td>51</td>
<td>0:00:51</td>
<td>2.8697607233</td>
</tr>
</tbody>
</table>

The plot in Output 15.4.2 shows all the five anomaly subsequences; anomalies 1 and 2 show strong abnormality.


Chapter 16
Utility Package

Contents

Overview: UTL Package ................................................................. 606
UTL Package Summary ............................................................... 606
Using the UTL Package .............................................................. 607
  Common Argument Types ....................................................... 607
  Return Codes ........................................................................ 608
CLLIMITS Object ................................................................. 608
  CLLIMITS Synopsis .................................................................. 609
  CLLIMITS Methods ................................................................. 610
OUTNVP Object ................................................................. 611
  OUTNVP Synopsis .................................................................. 612
  OUTNVP Methods ................................................................. 612
UTLSTAT Object ................................................................. 613
  UTLSTAT Synopsis ............................................................... 616
  UTLSTAT Methods ................................................................. 617
Examples: UTL Package ....................................................... 618
  Using CAS Sessions and CAS Engine Librefs ......................... 618
  Example 16.1: Collecting Forecast Fit Statistics and Ad Hoc Numeric Variables into CAS Tables ......................................................... 619
  Example 16.2: Computing Prediction Standard Errors and Confidence Limits for an Ad Hoc External Forecast ............................................... 622
Overview: UTL Package

This chapter describes the various utility object classes that are contained in the UTL package. The purpose of the UTL package is to provide a means for performing basic statistical computations on pairs of actual and predicted time series. The following types of computations are currently supported by the UTL package:

1. Computation of prediction standard errors and confidence limits for specified actual and predicted time series.
2. Computation and storing of model forecast fit statistics into CAS tables for specified actual and predicted time series.
3. Storing of ad hoc numeric variables that are defined in a user program into CAS tables.

The UTL package is object-oriented. To use the UTL package, you must declare instances of the object classes that are contained in the package. Declaring an object instance is the object-oriented equivalent of declaring a program variable. As with simple program variables, the declaration assigns the instance a name of your choosing and a type, which is defined by the object’s class. Unlike simple program variables, the object instance requires a different syntax for interacting with it and offers different functions (methods) that are contextual to the object. The object can offer very sophisticated capabilities with a simple-to-use interface.

UTL Package Summary

Table 16.1 summarizes the single object class in the SFS package.

<table>
<thead>
<tr>
<th>Object</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLIMITS</td>
<td>Compute prediction standard errors and confidence limits for specified actual and predicted time series.</td>
</tr>
<tr>
<td>OUTNVP</td>
<td>Ad hoc name, value pair collector object for storing numeric scalar or array variables that are found in a user program into a CAS table.</td>
</tr>
<tr>
<td>UTLSTAT</td>
<td>Collector object for computing forecast fit statistics for specified actual and predicted time series and storing those statistics in a CAS table.</td>
</tr>
</tbody>
</table>
Using the UTL Package

The objects in the UTL package are subdivided into two different categories:

1. Stateful computational objects (the CLIMITS object)
2. Collector objects (the OUTNVP and UTLSTAT objects)

Collector objects provide a mechanism to create a snapshot of results (either from stateful objects or from plain program variables) and store those results into CAS tables. Each collector object defines a CAS table schema that is determined by the collector object’s design. The collector objects in the UTL package (OUTNVP and UTLSTAT) follow a common method pattern. The basic execution follows this sequence of operations:

1. **Declare**: Create the collector object by using the object declaration statement.
2. **Collect**: Use the Collect method to store results into a CAS table. The input arguments of the Collect method are specific to the collector object. For example, the UTLSTAT collector object’s Collect method requires an actual and predicted time series as arguments. It then uses the specified series to compute forecast fit statistics and stores the results in a CAS table. In contrast, the OUTNVP collector object’s Collect method takes in an ad hoc numerical scalar or array variable from the user program and stores it into a CAS table. Rows that are collected are automatically appended to the collector’s associated CAS table at the end of each BY group, and the collector object’s saved row set is automatically reset. The Nrows attribute returns the current row count in the collector. A missing value is returned if nothing has been collected. The data, now stored in CAS tables, can then be used to produce reports or be used in further computations.

Common Argument Types

Table 16.2 defines the common argument types that are used in this chapter. The symbol $x$ corresponds to the variable name.

<table>
<thead>
<tr>
<th>SAS Data Type</th>
<th>Declaration Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>String</td>
<td>LENGTH $x$</td>
</tr>
<tr>
<td>Numeric</td>
<td>$x$ or LENGTH $x$</td>
</tr>
<tr>
<td>Numeric array</td>
<td>ARRAY $x$[n]/NOSYMBOLS;</td>
</tr>
<tr>
<td>Status</td>
<td>$x$ or LENGTH $x$</td>
</tr>
</tbody>
</table>
Return Codes

Table 16.3 shows the return code \((rc\) in method statements) status values that are used in this package. These status code values are returned after a method that is associated with an object is called; they can help determine whether the method executed successfully.

<table>
<thead>
<tr>
<th>Status</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt; 0</td>
<td>An unrecoverable error occurred. No result was produced.</td>
</tr>
<tr>
<td>= 0</td>
<td>Unconditional success. The requested action completed and a normal result was produced.</td>
</tr>
<tr>
<td>&gt; 0</td>
<td>Conditional success or warning. A result was produced subject to conditions.</td>
</tr>
</tbody>
</table>

Upon returning a negative status code, most methods in the SFS package objects also write a message to the output log that explains the causes of the related failure. These messages provide useful information during the process of debugging a user program. In the TSMODEL procedure, the output log is stored in the CAS table that is specified in the OUTLOG= option in the PROC TSMODEL statement. For more information about how to enable and configure logging and about how to access the output log after an invocation of the TSMODEL procedure, see Chapter 11, “The TSMODEL Procedure” (SAS Visual Forecasting: Forecasting Procedures).

CLIMITS Object

The CLIMITS object provides a mechanism for computing both the prediction standard errors and confidence limits of an external model (that is, a user-defined model) forecast, which is described by a pair of actual and predicted time series. The first stage of the computational process involves validating both input series. This validation is accomplished by ascertaining that the actual and predicted series have nonmissing observation values under at least one matching index. In addition, the predicted series is checked for the presence of extreme values. Next, the prediction standard errors are computed from the prediction errors (that is, the model residuals). Finally, the confidence limits are computed from the prediction standard errors. You can optionally supply the value of the confidence level that is used to compute the confidence limits. Note that the CLIMITS object retains all computed results in its internal memory. Individual forecast series can be queried via the GetForecast method, which is a method in the CLIMITS object.

Table 16.4 summarizes the methods that are associated with the UTL object.
Table 16.4 Methods of the CLIMITS Object

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compute</td>
<td>Compute the prediction standard errors and confidence limits for specified actual and predicted time series</td>
</tr>
<tr>
<td>GetForecast</td>
<td>Retrieve a computed forecast series by name</td>
</tr>
</tbody>
</table>

CLIMITS Synopsis

DECLARE OBJECT obj (CLIMITS) ;

Method syntax, in order of typical usage:

rc = obj.Compute (Actual, Predicted, <Alpha>) ;
rc = obj.GetForecast (Which, Result) ;

Figure 16.1 outlines the programmatic data flow through the CLIMITS object; each arrow represents a different object method.

Figure 16.1 CLIMITS Object Data Flow
CLIMITS Methods

CLIMITS.Compute Method

\[ \text{rc} = \text{obj}\.\text{Compute} (\text{Actual, Predicted, } <\text{Alpha}>); \]

Computes the prediction standard errors and confidence limits for specified actual and predicted time series. Both input series are validated by ascertaining the presence of nonmissing observation values under at least one matching index. Also, the predicted series is checked for the presence of extreme values. The computed forecast series are stored in the object’s internal memory and can be individually queried via the `GetForecast` method into a numeric array defined in the user program. A negative return code indicates that the validation of an input series failed (for example, the predicted series has extreme values or all missing values), you specified an out-of-range `Alpha` argument value, or a computational failure occurred (for example, out-of-memory error).

**Input Arguments**

You must specify the following input arguments:

- **Actual** specifies a numeric array that corresponds to the actual time series.
- **Predicted** specifies a numeric array that corresponds to the predicted time series.

You can also specify the following input argument:

- **Alpha** takes a numeric value between 0 and 1, exclusive, that specifies the significance level for forecast confidence bands. The default value is 0.05.

CLIMITS.GetForecast Method

\[ \text{rc} = \text{obj}\.\text{GetForecast} (\text{Which}, \text{Result}); \]

Places the specified forecast series (\textit{Which}) from the CLIMITS object into the specified numeric array (\textit{Result}). Forecast series have the same length as the predicted series that is supplied to the `Compute` method via its `Predicted` argument. The GetForecast method returns a negative status code if the `Compute` method returned a non zero value or if it was not yet executed (that is, no results exist to be queried).

**Input Arguments**

You must specify the following input argument:

- **Which** is a case-insensitive character string that specifies the type of forecast series to return. You can specify one of the following values:
  - **LOWER** returns a lower confidence limit series.
  - **STDERR** returns a prediction standard error series.
  - **UPPER** returns an upper confidence limit series.
**Output Arguments**

You must specify the following output argument:

(Result) specifies a numeric array to receive the forecast series. If the array length is longer than the requested series, it is padded with missing values.

---

**OUTNVP Object**

The OUTNVP object collects any ad hoc numeric variables that are defined in the user program into CAS tables. The OUTNVP collector object accepts any of the following numeric types: scalar literal, scalar variable, and array variable.

Table 16.5 shows the contents of the OUTNVP object.

<table>
<thead>
<tr>
<th>Column</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>NAME</em></td>
<td>String</td>
<td>Name of the dependent variable</td>
</tr>
<tr>
<td><em>CALL</em></td>
<td>Numeric</td>
<td>Call count within the BY group</td>
</tr>
<tr>
<td><em>UTAG</em></td>
<td>Numeric</td>
<td>User-defined numeric tag</td>
</tr>
<tr>
<td><em>VIX</em></td>
<td>Numeric</td>
<td>Value index (1-based) for the row</td>
</tr>
<tr>
<td><em>VALUE</em></td>
<td>Numeric</td>
<td>Actual value for variable’s row</td>
</tr>
</tbody>
</table>

Table 16.6 summarizes the methods that are associated with the OUTNVP object.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Collect</td>
<td>Collect value for numeric data type</td>
</tr>
<tr>
<td>nrows</td>
<td>Get the OUTNVP instance row count</td>
</tr>
</tbody>
</table>
OUTNVP Synopsis

DECLARE OBJECT obj (OUTNVP) ;

Method syntax, in order of typical usage:

\[
rc = \text{obj.Collect} \left( \text{Variable,} < \text{Utag} > \right) ;
\]

\[
nrows = \text{obj.nrows} () ;
\]

Figure 16.2 outlines the programmatic data flow through the OUTNVP object.

**Figure 16.2 OUTNVP Object Data Flow**

![](image)

OUTNVP Methods

OUTNVP.Collect Method

\[
rc = \text{obj.Collect} \left( \text{Variable,} < \text{Utag} > \right) ;
\]

Stores a numeric type, either scalar literal or variable or array variable in the OUTNVP table. When the Variable parameter is a numeric scalar literal or variable, this method collects a single row into the OUTNVP table. When the Variable parameter is a numeric array variable, this method collects a sequence of rows for the span of indices in the array. If the optional parameter Utag is specified, its value is included in the _UTAG_ column of each collected OUTNVP row. The name of the collected variable is also included in the _NAME_ column of each collected OUTNVP row. Similarly, the value of a counter that counts the number of calls to this method that are made within a BY group is also included in column _CALL_ of each OUTNVP row. A negative return value indicates that an error occurred while storing results into a CAS table.
Input Arguments
You must specify the following input arguments:

Variable specifies a numeric array to be collected into the OUTNVP table.

You can also specify the following input argument:

Utag takes a numeric value that is included in the _UTAG_ column of the OUTNVP table. The default value is a missing value.

OUTNVP.nrows Method

\[ \text{nrows} = \text{obj.nrows}() \]

Returns the number of rows that have been collected and stored in the CAS table.

Arguments
There are no arguments associated with this method.

UTLSTAT Object

The UTLSTAT object conveniently computes a number of forecast fit statistics for an ad hoc pair of user-specified actual and predicted time series. The computed forecast fit statistics are automatically stored in a CAS table. For each ad hoc pair of actual and predicted time series that is input into a UTLSTAT collector object, a single row of forecast fit statistics is added to the underlying CAS table. The CAS table schema that is used by the UTLSTAT object is compatible with the schema used by the HPFENGINE procedure for its OUTSTAT data set.

Table 16.7 shows the contents of the UTLSTAT object. For more information about the HPFENGINE procedure, see SAS Forecast Server Procedures: User’s Guide.

<table>
<thead>
<tr>
<th>Column</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>NAME</em></td>
<td>String</td>
<td>Variable name of actual series</td>
</tr>
<tr>
<td><em>MODEL</em></td>
<td>String</td>
<td>Variable name of predicted series</td>
</tr>
<tr>
<td>DFE</td>
<td>Numeric</td>
<td>Degrees of freedom error</td>
</tr>
<tr>
<td>N</td>
<td>Numeric</td>
<td>Number of observations</td>
</tr>
<tr>
<td>NOBS</td>
<td>Numeric</td>
<td>Number of observations used</td>
</tr>
<tr>
<td>NMISSA</td>
<td>Numeric</td>
<td>Number of missing actuals</td>
</tr>
<tr>
<td>NMISSP</td>
<td>Numeric</td>
<td>Number of missing predicted values</td>
</tr>
<tr>
<td>NPARMS</td>
<td>Numeric</td>
<td>Number of model parameters</td>
</tr>
<tr>
<td>Column</td>
<td>Type</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>--------</td>
<td>--------------------------------------------------</td>
</tr>
<tr>
<td>TSS</td>
<td>Numeric</td>
<td>Total sum of squares</td>
</tr>
<tr>
<td>SST</td>
<td>Numeric</td>
<td>Corrected total sum of squares</td>
</tr>
<tr>
<td>SSE</td>
<td>Numeric</td>
<td>Sum of square error</td>
</tr>
<tr>
<td>MSE</td>
<td>Numeric</td>
<td>Mean square error</td>
</tr>
<tr>
<td>RMSE</td>
<td>Numeric</td>
<td>Root mean square error</td>
</tr>
<tr>
<td>UMSE</td>
<td>Numeric</td>
<td>Unbiased mean square error</td>
</tr>
<tr>
<td>URMSE</td>
<td>Numeric</td>
<td>Unbiased root mean square error</td>
</tr>
<tr>
<td>MAPE</td>
<td>Numeric</td>
<td>Mean absolute percentage of error</td>
</tr>
<tr>
<td>MAE</td>
<td>Numeric</td>
<td>Mean absolute error</td>
</tr>
<tr>
<td>RSQUARE</td>
<td>Numeric</td>
<td>R-square</td>
</tr>
<tr>
<td>ADJRSQ</td>
<td>Numeric</td>
<td>Adjusted R-square</td>
</tr>
<tr>
<td>AADJRSQ</td>
<td>Numeric</td>
<td>Amemiya’s adjusted R-square</td>
</tr>
<tr>
<td>RWRSQ</td>
<td>Numeric</td>
<td>Random walk R-square</td>
</tr>
<tr>
<td>AIC</td>
<td>Numeric</td>
<td>Akaike’s information criterion</td>
</tr>
<tr>
<td>AICC</td>
<td>Numeric</td>
<td>Finite sample corrected Akaike’s information criterion</td>
</tr>
<tr>
<td>SBC</td>
<td>Numeric</td>
<td>Schwarz Bayesian information criterion</td>
</tr>
<tr>
<td>APC</td>
<td>Numeric</td>
<td>Amemiya’s prediction criterion</td>
</tr>
<tr>
<td>MAXERR</td>
<td>Numeric</td>
<td>Maximum error</td>
</tr>
<tr>
<td>MINERR</td>
<td>Numeric</td>
<td>Minimum error</td>
</tr>
<tr>
<td>MAXPE</td>
<td>Numeric</td>
<td>Maximum percentage of error</td>
</tr>
<tr>
<td>MINPE</td>
<td>Numeric</td>
<td>Minimum percentage of error</td>
</tr>
<tr>
<td>ME</td>
<td>Numeric</td>
<td>Mean error</td>
</tr>
<tr>
<td>MPE</td>
<td>Numeric</td>
<td>Mean percentage of error</td>
</tr>
<tr>
<td>MDAPE</td>
<td>Numeric</td>
<td>Median absolute percentage of error</td>
</tr>
<tr>
<td>GMAPE</td>
<td>Numeric</td>
<td>Geometric mean absolute percentage of error</td>
</tr>
<tr>
<td>MINPPE</td>
<td>Numeric</td>
<td>Minimum predicted percentage of error</td>
</tr>
<tr>
<td>MAXPPE</td>
<td>Numeric</td>
<td>Maximum predicted percentage of error</td>
</tr>
<tr>
<td>MPPE</td>
<td>Numeric</td>
<td>Mean predicted percentage of error</td>
</tr>
<tr>
<td>MAPPE</td>
<td>Numeric</td>
<td>Mean absolute predicted percentage of error</td>
</tr>
</tbody>
</table>
Table 16.7 continued

<table>
<thead>
<tr>
<th>Column</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MDAPPE</td>
<td>Numeric</td>
<td>Median absolute predicted percentage of error</td>
</tr>
<tr>
<td>GMAPPE</td>
<td>Numeric</td>
<td>Geometric mean absolute predicted percentage of error</td>
</tr>
<tr>
<td>MINSPE</td>
<td>Numeric</td>
<td>Minimum symmetric percentage of error</td>
</tr>
<tr>
<td>MAXSPE</td>
<td>Numeric</td>
<td>Maximum symmetric percentage of error</td>
</tr>
<tr>
<td>MSPE</td>
<td>Numeric</td>
<td>Mean symmetric percentage of error</td>
</tr>
<tr>
<td>SMAPE</td>
<td>Numeric</td>
<td>Mean absolute symmetric percentage of error</td>
</tr>
<tr>
<td>MDASPE</td>
<td>Numeric</td>
<td>Median absolute symmetric percentage of error</td>
</tr>
<tr>
<td>GMASPE</td>
<td>Numeric</td>
<td>Geometric mean absolute symmetric percentage of error</td>
</tr>
<tr>
<td>MINRE</td>
<td>Numeric</td>
<td>Minimum relative error</td>
</tr>
<tr>
<td>MAXRE</td>
<td>Numeric</td>
<td>Maximum relative error</td>
</tr>
<tr>
<td>MRE</td>
<td>Numeric</td>
<td>Mean relative error</td>
</tr>
<tr>
<td>MRAE</td>
<td>Numeric</td>
<td>Mean relative absolute error</td>
</tr>
<tr>
<td>MDRAE</td>
<td>Numeric</td>
<td>Median relative absolute error</td>
</tr>
<tr>
<td>GMRAE</td>
<td>Numeric</td>
<td>Geometric mean relative absolute error</td>
</tr>
<tr>
<td>MASE</td>
<td>Numeric</td>
<td>Mean absolute scaled error</td>
</tr>
<tr>
<td>MINAPES</td>
<td>Numeric</td>
<td>Minimum absolute error percentage of standard deviation</td>
</tr>
<tr>
<td>MAXAPES</td>
<td>Numeric</td>
<td>Maximum absolute error percentage of standard deviation</td>
</tr>
<tr>
<td>MAPES</td>
<td>Numeric</td>
<td>Mean absolute error percentage of standard deviation</td>
</tr>
<tr>
<td>MDAPES</td>
<td>Numeric</td>
<td>Median absolute error percentage of standard deviation</td>
</tr>
<tr>
<td>GMAPES</td>
<td>Numeric</td>
<td>Geometric mean absolute error percentage of standard deviation</td>
</tr>
</tbody>
</table>

Table 16.8 summarizes the methods that are associated with the UTLSTAT object.

Table 16.8 Methods of the UTLSTAT Object

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Collect</td>
<td>Compute and collect forecast fit statistics for a specified ad hoc pair of actual and predicted time series.</td>
</tr>
<tr>
<td>nrows</td>
<td>Get the UTLSTAT instance row count</td>
</tr>
</tbody>
</table>
UTLSTAT Synopsis

DECLARE OBJECT obj (UTLSTAT) ;

Method syntax, in order of typical usage:

rc = obj.Collect (Actual, Predicted, < Nparms >) ;
nrows = obj.nrows () ;

Figure 16.3 outlines the programmatic data flow through the UTLSTAT object.

Figure 16.3 UTLSTAT Object Data Flow
UTLSTAT Methods

UTLSTAT.Collect Method

    utlstatCollect rc=obj.Collect ;
    (Actual, Predicted, <Nparms>)

Computes and collects forecast fit statistics for an ad hoc pair of specified actual time series and predicted time series. Each call collects a single row into the UTLSTAT CAS table, which contains all the forecast fit statistics that are listed in Table 16.7. A negative return value indicates that an error occurred either during the computation of the forecast fit statistics or while storing results into a CAS table. The values of the forecast fit statistics computed by this method are sensitive to the value of optional parameter Nparms, which specifies the number of parameters that were used by the model that generated the predicted time series.

Input Arguments
You must specify the following input arguments:

Actual specifies a numeric array that corresponds to the actual time series.

Predicted specifies a numeric array that corresponds to the predicted time series.

You can also specify the following input argument:

Nparms takes a numeric value that specifies the number of parameters used by the model that generated the predicted series. The default value is 0.

UTLSTAT.nrows Method

    nrows=obj.nrows () ;

Returns the number of rows that have been collected and stored in the CAS table.

Arguments
There are no arguments associated with this method.
Examples: UTL Package

Throughout this section, it is assumed that you have already started a CAS session and that the data tables that are used in this section are in mycas, a CAS library that you have necessary permissions to work with. This section assumes that you are familiar with the general workings of the TSMODEL procedure; for more information, see Chapter 11, “The TSMODEL Procedure” (*SAS Visual Forecasting: Forecasting Procedures*).

Using CAS Sessions and CAS Engine Librefs

SAS Cloud Analytic Services (CAS) is the analytic server and associated cloud services in SAS Viya. This section describes how to create a CAS session and set up a CAS engine libref that you can use to connect to the CAS session. It assumes that you have a CAS server already available; contact your system administrator if you need help starting and terminating a server. This CAS server is identified by specifying the host on which it runs and the port on which it listens for communications. To simplify your interactions with this CAS server, the host information and port information for the server are stored as SAS option values that are retrieved automatically whenever this CAS server needs to be accessed. You can examine the host and port values for the server at your site by using the following statements:

```plaintext
proc options option=(CASHOST CASPORT);
run;
```

In addition to starting a CAS server, your system administrator might also have created a CAS session and a CAS engine libref for your use. You can define your own sessions and CAS engine librefs that connect to the CAS server as shown in the following statements:

```plaintext
cas mysess;
libname mycas cas sessref=mysess;
```

The CAS statement creates the CAS session named mysess, and the LIBNAME statement creates the mycas CAS engine libref that you use to connect to this session. It is not necessary to explicitly name the CASHOST and CASPORT of the CAS server in the CAS statement, because these values are retrieved from the corresponding SAS option values.

If you have created the mysess session, you can terminate it by using the TERMINATE option in the CAS statement as follows:

```plaintext
cas mysess terminate;
```

For more information about the CAS statement and the LIBNAME statement, see *SAS Cloud Analytic Services: User’s Guide*. For general information about CAS and CAS sessions, see *SAS Cloud Analytic Services: Fundamentals*.
Example 16.1: Collecting Forecast Fit Statistics and Ad Hoc Numeric Variables into CAS Tables

This example demonstrates the capabilities of the UTLSTAT and OUTNVP collector objects from the UTL package. The UTLSTAT collector object is used to compute and collect forecast fit statistics for a specified pair of actual and predicted time series. In contrast, the OUTNVP collector object is used to store ad hoc numeric variables that are found in the user-defined program into CAS tables. The example starts by using a DATA step to load a sample time series data set called Sashelp.Air into a CAS table. The TSMODEL procedure is then invoked and the Time Series Model (TSM) object (available in the TSM package) is used to generate a model for the Air series in the Sashelp.Air data set. Next, both the actual Air series and its predicted counterpart (generated by the TSM object) are input into the UTLSTAT object to compute and collect numerous forecast fit statistics (see Table 16.7) into a CAS table. Because this example processes only a single time series (the Air series) and a single BY group, a single row of output is stored in the UTLSTAT table. The example then uses the OUTNVP collector object to store into a CAS table various forecast series that are retrieved from the TSM object. Each forecast series is first queried into a numeric array via the GetForecast method in the TSM object. Each array is then input directly, one by one, into the OUTNVP collector object. Finally, some results are retrieved from the resulting UTLSTAT and OUTNVP CAS tables, sorted, and printed for further inspection.

The following DATA step loads the Sashelp.Air data set onto the CAS server. This DATA step assumes that your CAS engine libref is named mycas, but you can substitute any appropriately defined CAS engine libref.

```sas
data mycas.air (replace=yes);
  set Sashelp.Air;
run;
```

The following statements use the TSMODEL procedure to perform time series modeling on a single BY group. Because no ACCUMULATE= option is specified in the ID or VAR statements, its default value of TOTAL is used, which accumulates observations within a time period as a total sum of the nonmissing values.

```sas
proc tsmode data=mycas.air
  outarray = mycas.outarray (replace=yes)
  outscalar = mycas.outscalar (replace=yes)
  outobj=(
    utlstatobj = mycas.utlstat (replace=YES)
    outnvpmodelobj = mycas.outnvpmodel (replace=YES)
  )
  outlog = mycas.outlog (replace=yes)
  lead=12;
  id date interval=month start='01jan1949'd end='01dec1960'd;
  outarray predict error stderr lcl ucl;
  outscalar rc1 rc2 rc3 rc4 rc5 rc6 rc7 rc8;
  require tsm utl;
submit;
  /* Declare the "Time Series Model" (TSM) object and perform fit */
  declare object esm(tsm);
  rcl = esm.Initialize();
  if rcl < 0 then do; stop; end;
  rcl = esm.SetY(air);
  if rcl < 0 then do; stop; end;
```
rc1 = esm.Run();
if rc1 < 0 then do; stop; end;

/* Retrieve forecast series computed internally by the TSM object */
rc2 = esm.GetForecast('predict',predict); /*Predicted series*/
if rc2 < 0 then do; stop; end;
rc2 = esm.GetForecast('error',error); /*Forecast error series*/
if rc2 < 0 then do; stop; end;
rc2 = esm.GetForecast('stderr',stderr); /*Prediction std. errors series*/
if rc2 < 0 then do; stop; end;
rc2 = esm.GetForecast('lower',lcl); /*Lower conf. limits series*/
if rc2 < 0 then do; stop; end;
rc2 = esm.GetForecast('upper',ucl); /*Upper conf. limits series*/
if rc2 < 0 then do; stop; end;

/* Collect forecast series computed by the TSM object into a CAS table */
declare object outnvpmodelobj(outnvp);
rc3 = outnvpmodelobj.Collect(air,_SERIES_);
if rc3 < 0 then do; stop; end;
rc3 = outnvpmodelobj.Collect(predict,_SERIES_);
if rc3 < 0 then do; stop; end;
rc3 = outnvpmodelobj.Collect(error,_SERIES_);
if rc3 < 0 then do; stop; end;
rc3 = outnvpmodelobj.Collect(stderr,_SERIES_);
if rc3 < 0 then do; stop; end;
rc3 = outnvpmodelobj.Collect(lcl,_SERIES_);
if rc3 < 0 then do; stop; end;
rc3 = outnvpmodelobj.Collect(ucl,_SERIES_);
if rc3 < 0 then do; stop; end;

/* Compute and collect a vast number of forecast fit statistics */
declare object utlstatobj(utlstat);
rc4 = utlstatobj.Collect(air, predict);
if rc4 < 0 then do; stop; end;
endsubmit;
run;

You can use the PRINT procedure to display a small subset of the 55 different forecast fit statistics that are collected by the UTLSTAT object. The PRINT procedure can access CAS tables directly; thus, there is no need to retrieve the UTLSTAT table back from CAS and into a local data set prior to display.

/* Print a few forecast fit statistics for the single BY group */
proc print data=mycas.utlstat;
    var _NAME_ _MODEL_ NOBS RMSE MAPE MAE RSQUARE AIC;
run;

Output 16.1.1 shows that a single row of data was collected. This row corresponds to the forecast fit statistics that were collected for a single forecast (that is, one pair of actual and predicted series) within the single BY group that was processed by the TSMODEL procedure call.

**Output 16.1.1** Sample of the Forecast Fit Statistics Computed and Collected by the UTLSTAT Object

<table>
<thead>
<tr>
<th>Obs</th>
<th><em>NAME</em></th>
<th><em>MODEL</em></th>
<th>NOBS</th>
<th>RMSE</th>
<th>MAPE</th>
<th>MAE</th>
<th>RSQUARE</th>
<th>AIC</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>AIR</td>
<td>predict</td>
<td>144</td>
<td>10.579085435</td>
<td>3.0845016398</td>
<td>8.0064787209</td>
<td>0.9921692375</td>
<td>679.35714621</td>
</tr>
</tbody>
</table>
You can use the PRINT procedure again to display a small subset of observations from the following six different series that were collected by the OUTNVP object into the mycas.outnvptmodel CAS table:

- The actual series (that is, the Air series).
- The predicted series that was generated by the TSM object.
- The forecast error series that was generated by the TSM object.
- The prediction standard errors series that was generated by the TSM object.
- The lower confidence limits series that was generated by the TSM object.
- The upper confidence limits series that was generated by the TSM object.

You can print in sequence the values of the first three observations in each of these six series, for a total of 18 rows. To accomplish this, you must sort the mycas.outnvptmodel CAS table in a manner that sequentially aligns all rows that correspond to each unique observation index in all collected series (that is, all rows that correspond to the first observation in all six series, followed by all rows that correspond to the second observation in all six series, and so on). You can use the SORT procedure to simultaneously sort a CAS table and retrieve the results into a local data set as follows:

```sas
/* Sort OUTNVP table by "row index" and "BY group Collect() call count" */ /* (that is, _VIX_ and _CALL_ columns). Transfer the sorted table rows */ /* back from CAS and into a local data set. */ proc sort data=mycas.outnvptmodel out=outnvptmodel;
   by _VIX_ _CALL_
run;

/* Print the values of the first 18 rows, which correspond to the values */ /* of the first 3 observations in the six collected series. These 18 rows */ /* correspond to the condition "1 <= _VIX_ <= 3" in the CAS table called */ /* "MYCAS.OUTNVPMODEL". */ proc print data=outnvptmodel(obs=18);
run;
```
Chapter 16: Utility Package

Output 16.1.2 Sample of the Six Ad Hoc Series Collected by the OUTNVP Object

<table>
<thead>
<tr>
<th>Obs</th>
<th>VAR</th>
<th>CALL</th>
<th>UTAG</th>
<th>VIX</th>
<th>VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>AIR</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>112</td>
</tr>
<tr>
<td>2</td>
<td>predict</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>111.44700541</td>
</tr>
<tr>
<td>3</td>
<td>error</td>
<td>3</td>
<td>0</td>
<td>1</td>
<td>0.5529945856</td>
</tr>
<tr>
<td>4</td>
<td>stderr</td>
<td>4</td>
<td>0</td>
<td>1</td>
<td>10.691036547</td>
</tr>
<tr>
<td>5</td>
<td>lcl</td>
<td>5</td>
<td>0</td>
<td>1</td>
<td>90.492958825</td>
</tr>
<tr>
<td>6</td>
<td>ucl</td>
<td>6</td>
<td>0</td>
<td>1</td>
<td>132.401052</td>
</tr>
<tr>
<td>7</td>
<td>AIR</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>118</td>
</tr>
<tr>
<td>8</td>
<td>predict</td>
<td>2</td>
<td>0</td>
<td>2</td>
<td>119.94581467</td>
</tr>
<tr>
<td>9</td>
<td>error</td>
<td>3</td>
<td>0</td>
<td>2</td>
<td>-1.945814668</td>
</tr>
<tr>
<td>10</td>
<td>stderr</td>
<td>4</td>
<td>0</td>
<td>2</td>
<td>10.691036547</td>
</tr>
<tr>
<td>11</td>
<td>lcl</td>
<td>5</td>
<td>0</td>
<td>2</td>
<td>98.991768079</td>
</tr>
<tr>
<td>12</td>
<td>ucl</td>
<td>6</td>
<td>0</td>
<td>2</td>
<td>140.89986126</td>
</tr>
<tr>
<td>13</td>
<td>AIR</td>
<td>1</td>
<td>0</td>
<td>3</td>
<td>132</td>
</tr>
<tr>
<td>14</td>
<td>predict</td>
<td>2</td>
<td>0</td>
<td>3</td>
<td>135.17351431</td>
</tr>
<tr>
<td>15</td>
<td>error</td>
<td>3</td>
<td>0</td>
<td>3</td>
<td>-1.73514311</td>
</tr>
<tr>
<td>16</td>
<td>stderr</td>
<td>4</td>
<td>0</td>
<td>3</td>
<td>10.691036547</td>
</tr>
<tr>
<td>17</td>
<td>lcl</td>
<td>5</td>
<td>0</td>
<td>3</td>
<td>114.21946772</td>
</tr>
<tr>
<td>18</td>
<td>ucl</td>
<td>6</td>
<td>0</td>
<td>3</td>
<td>156.1275609</td>
</tr>
</tbody>
</table>

The sequence that is displayed in Output 16.1.2 was obtained by using the SORT procedure to sort the data in the OUTNVP table in increasing order of the columns _VIX_ (series row index) and _CALL_ (that is, the OUTNVP object’s Collect method call count within BY group). For example, rows 1–6 contain the values of the first observation in each of the six collected series, which correspond to the rows in the mycas.outnvpmode1 CAS table where column _VIX_ = 1: row 1 is the value of the first observation in the Air series (that is, column _VAR_ = 'AIR') and rows 2–6 are the values of the first observations in the Predict, Error, StdErr, LCL, and UCL forecast series that were retrieved from the TSM object via its GetForecast method. The same applies for the second observation in each of the six collected series as described by rows 7–12 (that is, rows where column _VIX_ = 2) and for the third observation in each of the six collected series as described by rows 13–18 (rows where column _VIX_ = 3).

Example 16.2: Computing Prediction Standard Errors and Confidence Limits for an Ad Hoc External Forecast

This example uses the TSMODEL procedure to compute the prediction standard errors and confidence limits of an ad hoc forecast that includes an actual and predicted time series. The example starts by using a DATA step to create a synthetic data set called ExternalModel. The synthetic data set contains three time series that make up an ad hoc external model (that is, a user-defined model) forecast: an actual series called Air, a simulated forecast error series called Error, and a simulated predicted series called Predict. This synthetic, ad hoc external forecast lacks the prediction standard errors and confidence limits, which will be computed by the CLIMITS object. Notice from the DATA step code that the actual time series is simply a copy of the Air series taken from the Sashelp.Air data set. The predicted time series is generated by simply adding a small amount of noise to the actual Air series. The added noise corresponds to random samples taken of the uniform distribution (which ranges from 0.0 to 1.0) multiplied by a factor of 10. Thus, the added noise samples range from 0.0 to 10.0 and have an expected value of 5.0. This expected value is important because
Example 16.2: Computing Prediction Standard Errors and Confidence Limits for an Ad Hoc Forecast

it corresponds to the average forecast error of the simulated predicted series, a value that over many samples should approximate the prediction standard errors that will be computed by the CLIMITS object for this synthetic, ad hoc external forecast. Next, the synthetic data set is uploaded to a CAS table and the TSMODEL procedure is invoked. The actual and predicted time series are input to the CLIMITS object’s Compute method to compute the prediction standard errors and confidence limits of the ad hoc external forecast. The CLIMITS object’s GetForecast method is then used to retrieve the resulting three forecast series into numeric arrays that are defined in the user program. Finally, the OUTNVP collector object collects a total of six series into a CAS table: the actual Air series, the simulated predicted and forecast error series, and the three forecast series that were computed by the CLIMITS object. This was similarly done in Example 16.1 to store into a CAS table the actual series plus the five forecast series that were retrieved from the TSM object via its own GetForecast method. Finally, some results are retrieved from CAS tables, sorted, and printed for further inspection:

The following DATA step creates a synthetic data set that contains actual and simulated predicted time series:

```plaintext
data ExternalModel (replace=yes);
set Sashelp.Air;     /* The actual series: Sashelp.Air */
error = floor(10*ranuni(246));  /* Simulated forecast error series */
predict = air + error;    /* Simulated predicted series */
run;
```

The following DATA step loads the ExternalModel data set onto the CAS server. This DATA step assumes that your CAS engine libref is named mycas, but you can substitute any appropriately defined CAS engine libref.

```plaintext
data mycas.ExternalModel;
set ExternalModel;
run;
```

The following statements use the TSMODEL procedure to compute the prediction standard errors and confidence limits of the ad hoc external forecast that resides in the synthetic data set:

```plaintext
proc tsmodel data = mycas.externalmodel
  outarray = mycas.outarray (replace=yes)
  outscalar = mycas.outscalar (replace=yes)
  outlog = mycas.outlog (replace=yes)
  outobj=( outnvpmodelobj = mycas.outnvpmodel (replace = YES) );
id date interval=month start='01jan1949'd end='01dec1960'd;
var air predict error;
outarrays stderr lcl ucl;
outscalar rc1 rc2 rc3;
require utl;
submit;
  /* Compute the prediction standard errors and confidence limits */
  declare object clim(CLIMITS);
  rc1 = clim.Compute(air,predict,0.05);  /* Confidence level: 0.05 */
  if rc1 < 0 then do; stop; end;
  /* Retrieve the forecast series stored internally in the CLIMITS object */
  rc2 = clim.GetForecast('stderr',stderr); /* Prediction sdt. errors series */
  if rc2 < 0 then do; stop; end;
  rc2 = clim.GetForecast('lower',lcl);    /* Lower conf. limits series */
  if rc2 < 0 then do; stop; end;
```
rc2 = clim.GetForecast('upper',ucl);  /* Upper conf. limits series */
if rc2 < 0 then do; stop; end;

/* Collect the actual, predicted, and forecast error series, in addition */
/* to the forecast series computed by the CLIMITS object into a CAS table*/
declare object outnvpmodelobj(outnvp);
   rc3 = outnvpmodelobj.Collect(air,_SERIES_);
   if rc3 < 0 then do; stop; end;
   rc3 = outnvpmodelobj.Collect(predict,_SERIES_);
   if rc3 < 0 then do; stop; end;
   rc3 = outnvpmodelobj.Collect(error,_SERIES_);
   if rc3 < 0 then do; stop; end;
   rc3 = outnvpmodelobj.Collect(stderr,_SERIES_);
   if rc3 < 0 then do; stop; end;
   rc3 = outnvpmodelobj.Collect(lcl,_SERIES_);
   if rc3 < 0 then do; stop; end;
   rc3 = outnvpmodelobj.Collect(ucl,_SERIES_);
   if rc3 < 0 then do; stop; end;
endsubmit;
run;

Following what was done in Example 16.1, you can use the PRINT procedure to display a small subset of observations from the following six different series that were collected by the OUTNVP object into the mycas.outnvpmodel CAS table:

- The actual series (Air series) of the ad hoc external forecast.
- The simulated predicted series of the ad hoc external forecast.
- The simulated forecast error series of the ad hoc external forecast.
- The prediction standard errors series that were computed by the CLIMITS object for the ad hoc external forecast.
- The lower confidence limits series that were computed by the CLIMITS object for the ad hoc external forecast.
- The upper confidence limits series that were computed by the CLIMITS object for the ad hoc external forecast.

You can print in sequence the values of the first three observations in each of these six series, for a total of 18 rows. To accomplish this, you must sort the mycas.outnvpmodel CAS table in a manner that sequentially aligns all rows that correspond to each unique observation index in all collected series (all rows that correspond to the first observation in all six series, followed by all rows that correspond to the second observation in all six series, and so on). You can use the SORT procedure to simultaneously sort a CAS table and retrieve the results into a local data set as follows:

/* Sort OUTNVP table by "row index" and "BY group Collect() call count" */
/* (that is, _VIX_ and _CALL_ columns). Transfer the sorted table rows */
/* back from CAS and into a local data set. */
proc sort data=mycas.outnvpmodel out=outnvpmodel;
   by _VIX_ _CALL_;
run;
Example 16.2: Computing Prediction Standard Errors and Confidence Limits for an Ad Hoc Forecast

/ * Print the value of the first 18 rows, which corresponds to the values */
/ * of the first 3 observations in the six collected series. These 18 rows */
/ * correspond to the condition "1 <= _VIX_ <= 3" in the CAS table called */
/ * "MYCAS.OUTNVPMODEL". */
proc print data=outnvpmodel(obs=18);
run;

Output 16.2.1 Sample of the Six Ad Hoc Series Collected by the OUTNVP Object

<table>
<thead>
<tr>
<th>Obs</th>
<th><em>VAR</em></th>
<th><em>CALL</em></th>
<th><em>UTAG</em></th>
<th><em>VIX</em></th>
<th><em>VALUE</em></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>AIR</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>112</td>
</tr>
<tr>
<td>2</td>
<td>predict</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>117</td>
</tr>
<tr>
<td>3</td>
<td>error</td>
<td>3</td>
<td>0</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>stderr</td>
<td>4</td>
<td>0</td>
<td>1</td>
<td>5.0408057116</td>
</tr>
<tr>
<td>5</td>
<td>lcl</td>
<td>5</td>
<td>0</td>
<td>1</td>
<td>107.12020235</td>
</tr>
<tr>
<td>6</td>
<td>ucl</td>
<td>6</td>
<td>0</td>
<td>1</td>
<td>126.87979765</td>
</tr>
<tr>
<td>7</td>
<td>AIR</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>118</td>
</tr>
<tr>
<td>8</td>
<td>predict</td>
<td>2</td>
<td>0</td>
<td>2</td>
<td>124</td>
</tr>
<tr>
<td>9</td>
<td>error</td>
<td>3</td>
<td>0</td>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td>10</td>
<td>stderr</td>
<td>4</td>
<td>0</td>
<td>2</td>
<td>5.0408057116</td>
</tr>
<tr>
<td>11</td>
<td>lcl</td>
<td>5</td>
<td>0</td>
<td>2</td>
<td>114.12020235</td>
</tr>
<tr>
<td>12</td>
<td>ucl</td>
<td>6</td>
<td>0</td>
<td>2</td>
<td>133.87979765</td>
</tr>
<tr>
<td>13</td>
<td>AIR</td>
<td>1</td>
<td>0</td>
<td>3</td>
<td>132</td>
</tr>
<tr>
<td>14</td>
<td>predict</td>
<td>2</td>
<td>0</td>
<td>3</td>
<td>135</td>
</tr>
<tr>
<td>15</td>
<td>error</td>
<td>3</td>
<td>0</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>16</td>
<td>stderr</td>
<td>4</td>
<td>0</td>
<td>3</td>
<td>5.0408057116</td>
</tr>
<tr>
<td>17</td>
<td>lcl</td>
<td>5</td>
<td>0</td>
<td>3</td>
<td>125.12020235</td>
</tr>
<tr>
<td>18</td>
<td>ucl</td>
<td>6</td>
<td>0</td>
<td>3</td>
<td>144.87979765</td>
</tr>
</tbody>
</table>

The sequence displayed in Output 16.2.1 was obtained by using the SORT procedure to sort the data in the OUTNVP table in increasing order of the columns _VIX_ (that is, the series row index) and _CALL_ (that is, the OUTNVP object’s Collect method call count within BY group). For example, rows 1–6 contain the values of the first observation in each of the six collected series, which correspond to the rows in the mycas.outnvpmodel CAS table where column _VIX_ = 1: row 1 is the value of the first observation in the actual Air series (that is, column _VAR_ = 'AIR'), rows 2–3 are the values of the first observations in the simulated Predict and Error series of the ad hoc external forecast, and rows 4–6 are the values of the first observations in the StdErr, LCL, and UCL forecast series that were computed by the CLIMITS object for the ad hoc external forecast and retrieved via its GetForecast method. The same applies for the second observation in each of the six collected series as described by rows 7–12 (that is, rows where column _VIX_ = 2) and for the third observation in each of the six collected series as described by rows 13–18 (rows where column _VIX_ = 3).

Notice also in Output 16.2.1 the value of 5.0408057116, which is reported for the first three observations of the prediction standard errors series, as shown by rows where column _VAR_ = 'stderr'. This reported value should be close to the expected value of the simulated forecast error of the synthetic external forecast (5.0).
Gain Greater Insight into Your SAS® Software with SAS Books.

Discover all that you need on your journey to knowledge and empowerment.

support.sas.com/bookstore for additional books and resources.