SAS® Visual Forecasting 8.2 Time Series Packages

The correct bibliographic citation for this manual is as follows: SAS Institute Inc. 2017. *SAS® Visual Forecasting 8.2: Time Series Packages*. Cary, NC: SAS Institute Inc.

SAS® Visual Forecasting 8.2: Time Series Packages

SAS Institute Inc., SAS Campus Drive, Cary, NC 27513-2414

December 2017

Contents

Chapter 1

What's New in SAS Visual Forecasting 8.2 Time Series Packages

Contents

Overview

SAS Visual Forecasting 8.2 adds three new packages and adds enhancements to other packages.

New Packages

Singular Spectrum Analysis (SSA) Package

The new singular spectrum analysis (SSA) package enables you to perform SSA forecasting.

Time Series Motif (MTF) Package

The new time series motif (MTF) package enables you to search for patterns in time series.

Time Filter (TIMFIL) Package

The new time filter (TIMFIL) package enables you to filter time series vectors.

Package Enhancements

Automatic Time Series Modeling (ATSM) Package

- The new model combination list specification (COMBSPEC) object enables you to combine the forecasts that are generated from multiple model specifications into a single forecast.
- The new event database (EVENT) object enables you to generate dummy variables for predefined events and include them as independent variables in the model selection process.
- The new OUTSCORE object enables you to store forecast model score XML that is generated by a FORENG object into a CAS table.
- The new INSCORE object enables you to replay the contents of an OUTSCORE object.
- The new SCORE object enables you to generate forecasts by using a model that was previously generated by the FORENG object. The model source can be an instance of the FORENG object itself or an INSCORE object.

Time Series Model (TSM) Package

- The new unobserved component model specification (UCMSPEC) object enables you to specify UCM models for modeling and forecasting.
- The new intermittent demand model specification (IDMSPEC) object enables you to specify IDM models for modeling and forecasting.
- The new external model specification (EXMSPEC) object enables you to specify external models for modeling and forecasting.

Chapter 2
Introduction to Packages for the TSMODEL Procedure

Contents

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.

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 2.1 Packages for the TSMODEL Procedure

Package Name	Abbreviation	Description
Automatic time series analysis and forecasting	ATSM	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)
Time series motif discovery	MTF	Tools for the discovery of frequent patterns or repeated subsequences in time series
Simple forecast service	SFS	Tools for automatic forecasting of time series with a simple-to-use interface and using only ARIMA models
Singular spectrum analysis	SSA	Tools for decomposing a time series into additive components and categorizing those components based on the magnitudes of their contributions
Time filters	TIMFIL	Tools for filtering and aggregation of time series
Time frequency analysis	TFA	Tools for efficient analysis of time series in both time and frequency domains
Time series analysis	TSA	Tools for efficient statistical analysis of time series (transformations, decompositions, statistical tests for intermittency, seasonality, stationarity, and forecast bias, and so on)
Time series model	TSM	Tools for efficient time series modeling and forecasting
Utility	UTL	Tools for performing basic statistical computations on pairs of actual and predicted time series

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.

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 4, "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 OUTEST 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 Analysis and Forecasting Package

Contents

Overview: ATSM Package

The automatic time series analysis and forecasting (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, HPFENGINE, and HPFSELECT procedures in *SAS Forecast Server Procedures: User's Guide*. In addition, it is helpful to review Chapter 4, "The TSMODEL Procedure" (*SAS Visual Forecasting: Forecasting Procedures*), and Chapter 9, "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 3.1 Objects in the Automatic Time Series Modeling Package

Object	Description
Time Series Modeling and Forecasting Objects	
DIAGNOSE	Time series model generator
FORENG	Time series forecasting engine
SCORE	Time series forecast from model score
TSDF	Time series data frame
Objects for Controlling Model Identification	
COMBSPEC	Model combination list specification
DIAGSPEC	Time series model diagnosis control specification
EVENT	Time series event object
SELSPEC	Model selection list specification
Collector Objects (Note 1)	
OUTCOMP	Time series model components collector
OUTDIAG	Persistent diagnostic control specifications
OUTEST	Time series model parameter estimates collector
OUTFMSG	Persistent forecast model selection graph XML
OUTFOR	Time series model forecast collector
OUTINDEP	Time series model input series collector
OUTMODELINFO	Time series model information collector

Table 3.1 *continued*

Object	Description
OUTSCORE	Forecast score XML persistence
OUTSELECT	Time series model selection statistics collector
OUTSTAT	Time series model fit statistics collector
Repeater Objects (Note 2)	
INDIAG	Replay of time series diagnostic control specifications
INFMSG	Forecast model selection graph (FMSG) repeater
INEST	Automatic time series model parameter estimates repeater
INSCORE	Forecast model score repeater

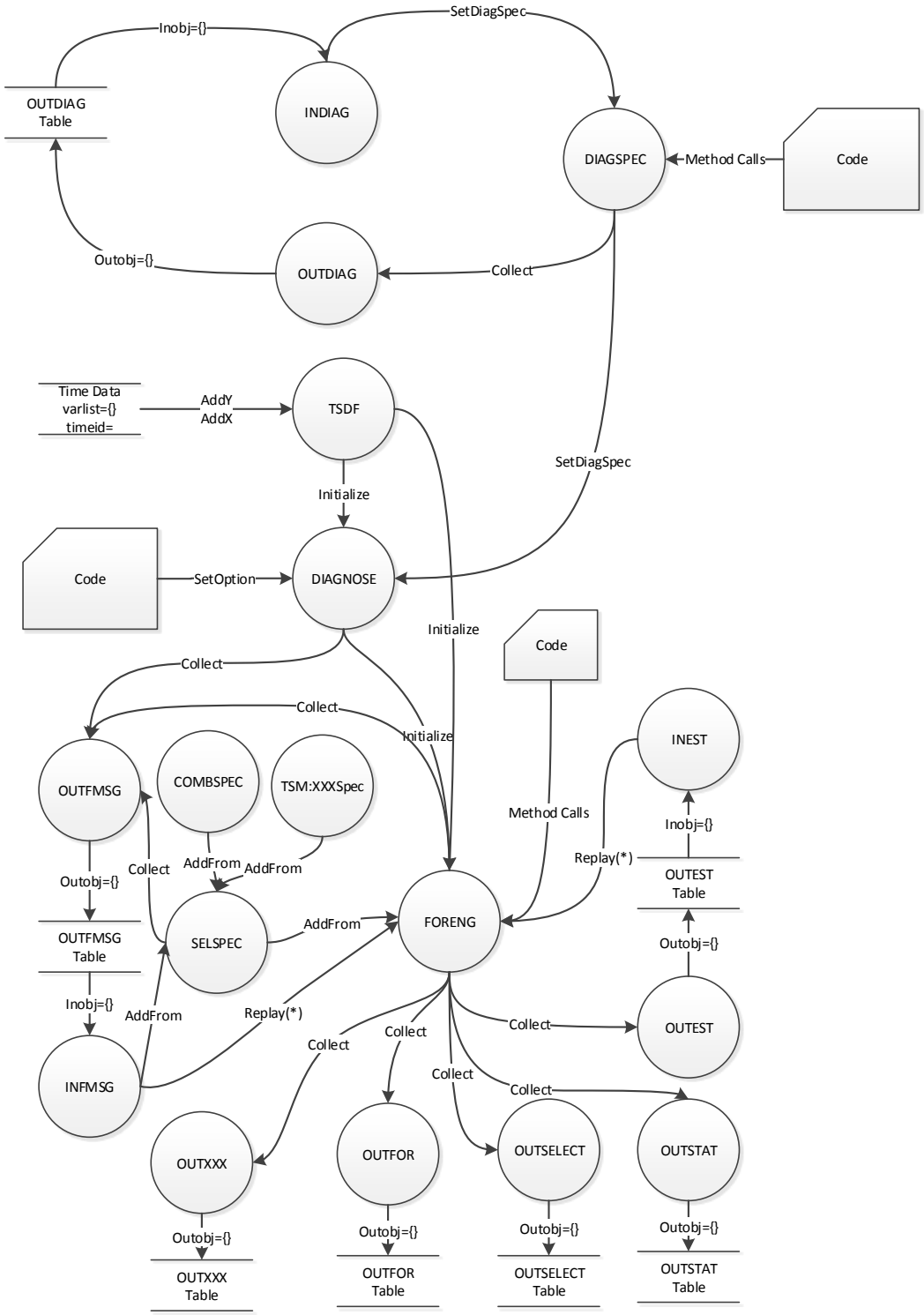
NOTE:

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:XXXSpec 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



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 3.2 Return Codes

Status	Meaning
< 0	An unrecoverable error occurred. No result was produced.
= 0	Unconditional success. The requested action completed and a normal result was produced.
> 0	Conditional success or warning. A result was produced subject to conditions.

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 3.3 Methods of the TSDF Object

Method	Description
AddEvent	Add series defined by event definition
AddSeries	Add ancillary series
AddX	Add X series
AddY	Add Y series
GetSeries	Get series
Initialize	Initialize TSDF instance
ntid	Return the observation count of the time ID variable
SetOption	Set named option

TSDF Synopsis

DECLARE OBJECT *obj* **(TSDF) ;**

Method syntax, in order of typical usage:

rc=*obj*.**Initialize** () ;
rc=*obj*.**AddY** (*YSeries* < ,'*Name*',*Value*>) ;
rc=*obj*.**AddX** (*XSeries* < ,'*Name*',*Value*>) ;
rc=*obj*.**AddEvent** (*EventObj, EventName* < ,'*Name*',*Value*>) ;
rc=*obj*.**AddSeries** (*Series* < ,'*Name*',*Value*>) ;
rc=*obj*.**SetOption** ('*Name*',*Value*) ;
rc=*obj*.**GetSeries** ('*Name*',*Result*) ;
ntids=*obj*.**ntid** () ;

TSDF Methods

TSDF.AddEvent Method

rc=*obj*.**AddEvent** (*EventObj, EventName* < ,'*Name*',*Value*>) ;

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. A numeric variable with this name is created for the TSDF instance. Only ARIMA and UCM models support events.

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.

TSDF.AddSeries Method

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 '*Name*',*Value* pair:

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

TSDF.AddX Method

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.

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.

TSDF.AddY Method

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 '*Name*',*Value* pair:

'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.

TSDF.GetSeries Method

rc=*obj*.**GetSeries (**'*Name*',*Result***) ;**

Retrieves the specified series by its name ('*Name*' or alias) from the TSDF instance and stores it in the specified numeric array (*Result*).

Input Arguments

You must specify the following input argument:

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

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*.**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*.**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.SetOption Method

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

Specifies an option for the TSDF instance.

Input Arguments

You must specify the following '*Name*' and a *Value* for it:

'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 3.4 Methods of the DIAGNOSE Object

Method	Description
Initialize	Initialize the DIAGNOSE object
nmodels	Get the number of model families diagnosed
Replay	Replay the diagnostic control specification
Run	Perform model generation
SetOption	Set named option
SetSpec	Set diagnostic controls

DIAGNOSE Synopsis

DECLARE OBJECT *obj* **(DIAGNOSE) ;**

Method syntax, in order of typical usage:

rc=*obj*.**Initialize (***TSDFObject* **) ;**
rc=*obj*.**SetSpec (***DIAGSPECObject***) ;**
rc=*obj*.**Replay (***INDIAGObject***) ;**
rc=*obj*.**Run () ;**
rc=*obj*.**SetOption (**'*Name*', *Value* < ,'*Name*', *Value* >**) ;**
nmodels=*obj*.**nmodels () ;**

DIAGNOSE Methods

DIAGNOSE.Initialize Method

rc=*obj*.**Initialize (***TSDFObject***) ;**

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

Input Arguments

You must specify the following input argument:

TSDFObject 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

nmodels=*obj*.**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=*obj*.**Replay (***INDIAGObject***) ;**

Restores a previously stored diagnostic control specification from the specified *INDIAGObject*. 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 63.

Input Arguments

You must specify the following input argument:

INDIAGObject specifies the INDIAG instance that defines the source of the diagnostic control specification to restore.

DIAGNOSE.Run Method

rc=*obj*.**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

Family	Description
ARIMAX	ARIMA model that includes predictors that use ARIMA-REG identification order
ESM	Seasonal and nonseasonal exponential smoothing models
IDM	Intermittent demand (IDM) model
REGARIMA	ARIMA model that includes predictors that use REG-ARIMA identification order
UCM	UCM model with predictors

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 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.

'CRITERION' 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 0.

'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.

DIAGNOSE.SetSpec Method

rc=*obj*.**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 3.6 Methods of the FORENG Object

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

FORENG Synopsis

DECLARE OBJECT *obj* **(FORENG) ;**

Method syntax, in order of typical usage:

rc=*obj*.**Initialize (***TSDFObject***) ;**
rc=*obj*.**Initialize (***DIAGNOSEObject***) ;**
rc=*obj*.**AddFrom (***specObject***) ;**
rc=*obj*.**Replay (***infmsgObject* < ,*inestObject*>**) ;**
rc=*obj*.**Run () ;**
rc=*obj*.**GetForecast (***Which, Result***) ;**
rc=*obj*.**GetXSeries (***WhichX, XSeries***) ;**
rc=*obj*.**SetOption (**'*Name*', *Value* < ,'*Name*', *Value*>**) ;**
model=*obj*.**model () ;**
nfor=*obj*.**nfor () ;**
criterion=*obj*.**criterion () ;**

FORENG Methods

FORENG.AddFrom Method

rc=*obj*.**AddFrom (***specObject***) ;**

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:

specObject 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

specObject	**Package**	**Description**
ARIMASPEC	TSM	Includes the ARIMA model specification
COMBSPEC	ATSM	Includes the combination model specification
ESMSPEC	TSM	Includes the exponential smoothing model specification
IDMSPEC	TSM	Includes the IDM model specification
SELSPEC	ATSM	Includes the model selection specification
TSM	TSM	Include the TSM model specification
UCMSPEC	TSM	Includes the UCM model specification

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 44 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*.**Initialize (***DIAGNOSEObject***) ;**

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*.**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*.**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*.**Replay (***infmsgObject* < ,*inestObject*>**) ;**

Restores a previously stored forecast model selection graph (FMSG) from the specified *infmsgObject*. This method establishes the context to allow for a model selection and forecasting run of the FORENG instance. Optionally, you can specify *inestObject* that supplies previously stored parameter estimates for the restored FMSG. The *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 81; for more information about storing FMSG model parameter estimates, see the section "OUTEST Object" on page 64.

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 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.

'CRITERION' 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*.

'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 1, inclusive, that specifies the holdout percentage. The default value is 0.

'HORIZON' takes a numeric *Value* that specifies the forecast horizon reference time.

'LEAD' takes a nonnegative integer *Value* that specifies the forecast lead. The default value is 12.

'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

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

SCORE Synopsis

DECLARE OBJECT *obj* **(SCORE) ;**

Method syntax, in order of typical usage:

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

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

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

rc=*obj*.**Initialize (***FORENGObject***) ;**

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

Input Arguments

You must specify the following input argument:

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

SCORE.Initialize Method

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 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

Family	Description
ARIMA	ARIMA/ARIMAX models
ESM	Exponential smoothing models
IDM	Intermittent demand models (Croston's model and average demand model)
UCM	Unobserved component models

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

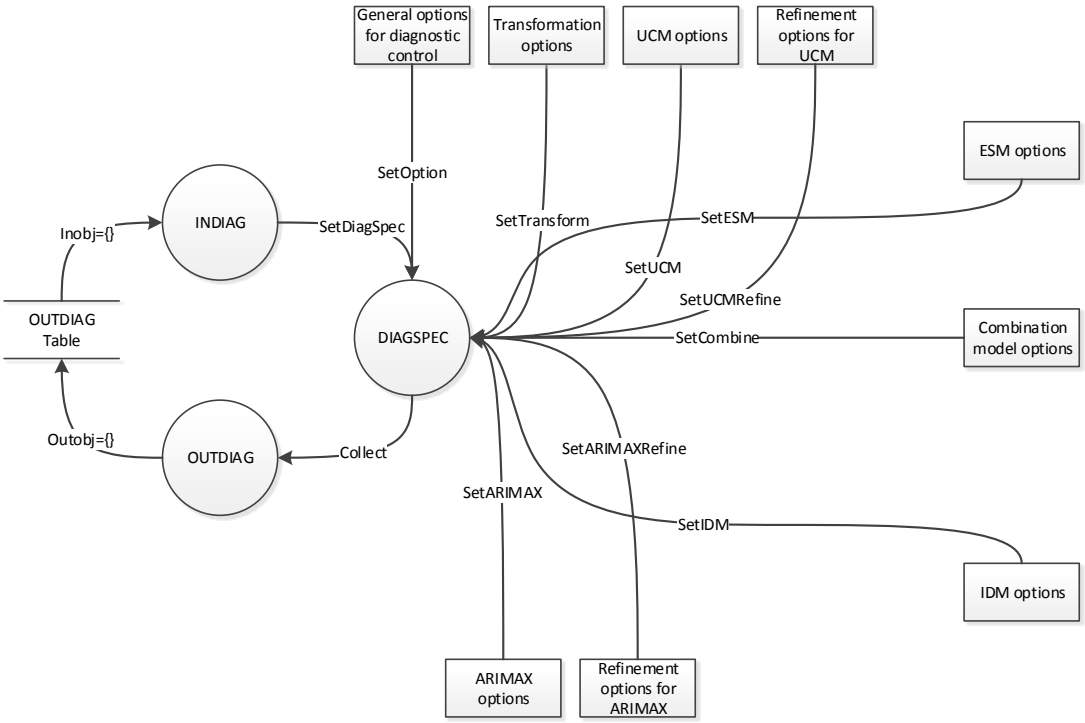
Table 3.10 Methods of the DIAGSPEC Object

Method	Description
Close	Close the DIAGSPEC object
Open	Open the DIAGSPEC object
SetARIMAX	Set ARIMA diagnostic options
SetARIMAXRefine	Set ARIMA parameter refinement options
SetCombine	Configure automatic combination options
SetESM	Set ESM diagnostic options
SetIDM	Set IDM diagnostic options
SetOption	Set overall control options
SetTransform	Set transform test options
SetTrend	Set trend test options
SetUCM	Set UCM diagnostic options
SetUCMRefine	Set UCM parameter refinement options

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.

Figure 3.2 DIAGSPEC Data Flow



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
- 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

rc=*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

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

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. 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*:

- **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.SetARIMAXRefine Method

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

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

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 ℓ_1 norm that involves the loss series.

LINF specifies that the objective is an ℓ_∞ 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 Σ_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 ℓ_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 ℓ_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 ℓ_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. The weights correspond to the order of specification of the model families. 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

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 HPFESMSPEC 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

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.

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*', *Value* < ,'*Name*',*Value*,... >) ;

Sets a named option for the DIAGSPEC instance.

Input Arguments

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

'CRITERION' 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 1 that specifies the percentage of AIC or SBC improvement between two candidate models. The default value is 0.001.

'INPUTMISSPCT' takes a numeric *Value* between 0 and 100 that specifies the 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, 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

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).

LOG requests logarithmic transformation.

LOGIT requests logistic transformation.

NONE does not apply a transformation.

SQRT specifies square-root transformation.

The default is NONE.

TRANSOPT 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

rc=*obj*.**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:

AUTO 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

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 HPFDIAGNOSE 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.

Table 3.11 Methods of the EVENT Object

Method	Description
Initialize	Initialize the EVENT instance

EVENT Synopsis

DECLARE OBJECT *obj* **(EVENT) ;**

Method syntax:

rc=*obj*.**Initialize () ;**

EVENT Methods

EVENT.Initialize Method

rc=*obj*.**Initialize () ;**

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

Arguments

There are no arguments associated with this method.

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

Object	Package	Value
ARIMASPEC	TSM	ARIMA model to include
COMBSPEC	ATSM	Forecast combination to include
DIAGNOSE	ATSM	Generated models to include
ESMSPEC	TSM	ESM model to include
FORENG	ATSM	FORENG current FMSG to include
IDMSPEC	TSM	IDM model to include
INFMSG	ATSM	Restored SELSPEC models to include
SELSPEC	ATSM	Subgraph to include
TSM	TSM	TSM model to include
UCMSPEC	TSM	UCM model to include

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.

Figure 3.3 SELSPEC Data Flow

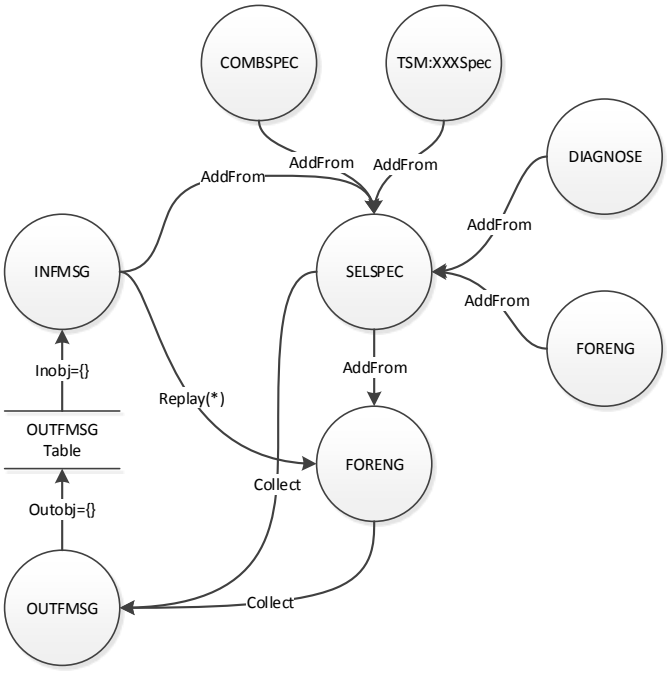


Table 3.13 summarizes the methods that are associated with the SELSPEC object.

Table 3.13 Methods of the SELSPEC Object

Method	Description
AddFrom	Add from the specified source to the SELSPEC object
Close	Close SELSPEC for editing
Open	Open SELSPEC for editing
SetDiagnose	Set diagnose options
SetOption	Set named option

SELSPEC Synopsis

DECLARE OBJECT *obj* **(SELSPEC) ;**

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 () ;**

SELSPEC Methods

SELSPEC.AddFrom Method

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 3.14 Model Objects for the SELSPEC.AddFrom Method

SourceObj	**Package**	**Value**
ARIMASPEC	TSM	Model from an ARIMASPEC object
COMBSPEC	ATSM	COMBSPEC object's DAG
DIAGNOSE	ATSM	DIAGNOSE object's generated models
ESMSPEC	TSM	Model from an ESMSPEC object
FORENG	ATSM	Current FORENG FMSG
IDMSPEC	TSM	Model from an IDMSPEC object
INFMSG	ATSM	INFMSG XML
SELSPEC	ATSM	SELSPEC object's DAG
TSM	TSM	Model from a TSM object
UCMSPEC	TSM	Model from an UCMSPEC object

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 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

Family	Package	Value
ARIMASPEC	TSM	ARIMA model to include
COMBSPEC	ATSM	Forecast combination to include
DIAGNOSE	ATSM	Generated models to include
ESMSPEC	TSM	ESM model to include
FORENG	ATSM	FORENG current FMSG to include
IDMSPEC	TSM	IDM model to include
INFMSG	ATSM	INFMSG FMSG to include
SELSPEC	ATSM	Subgraph to include
TSM	TSM	Object's model to include
UCMSPEC	TSM	UCM model to include

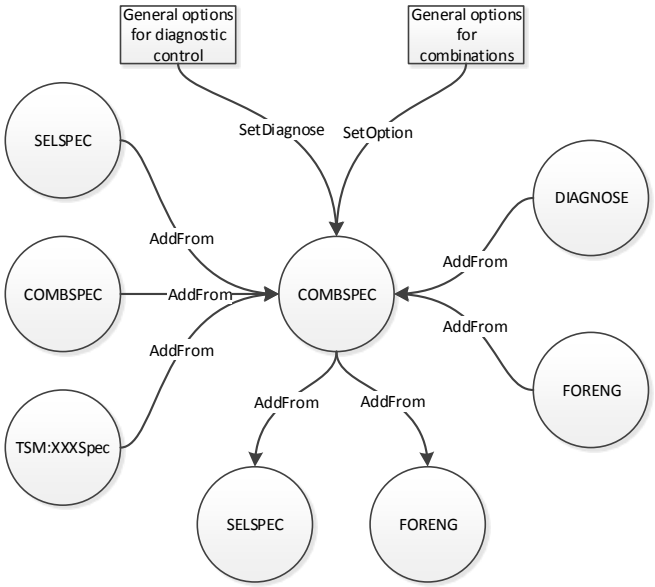
Table 3.16 summarizes the methods that associated with the COMBSPEC Object. TSM:XXXSpec refers to any of the ESMSPEC, ARIMASPEC, UCMSPEC, IDMSPEC, and TSM objects, which are part of the TSM package.

Table 3.16 Methods of the COMBSPEC Object

Method	Description
AddFrom	Add from the specified source to the COMBSPEC object
Close	Close COMBSPEC for editing
Open	Open COMBSPEC for editing
SetDiagnose	Set diagnose options
SetOption	Set combination process options

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 3.17 Model Objects for the COMBSPEC.AddFrom Method

Object	Package	Value
ARIMASPEC	TSM	Model from the ARIMASPEC object
COMBSPEC	ATSM	COMBSPEC object's DAG
DIAGNOSE	ATSM	DIAGNOSE object's generated models
ESMSPEC	TSM	Model from the ESMSPEC object
FORENG	ATSM	Current FORENG FMSG
IDMSPEC	TSM	Model from the IDMSPEC object
INFMSG	ATSM	INFMSG FMSG
SELSPEC	ATSM	SELSPEC object's DAG
TSM	TSM	Model from the TSM object
UCMSPEC	TSM	Model from the UCMSPEC object

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

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

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

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

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

Specifies 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 ℓ_1 norm that involves loss series.
- **LINF** specifies that the objective is an ℓ_∞ 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.

'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 Σ_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 ℓ_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 ℓ_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 ℓ_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, W1, 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 3.18 Contents of the OUTDIAG Object

Column	Type	Description
DIAGSPEC	String	Diagnostic specification XML document
NAME	String	Name of the dependent variable (can be missing)
SPECLEN	Numeric	Length of the diagnostic specification XML
SPECNAME	String	Name of the diagnostic specification

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

Table 3.19 Methods of the OUTDIAG Object

Method	Description
Collect	Collect diagnostic specification from the DIAGSPEC and DIAGNOSE objects
nrows	Query the number of rows in the OUTDIAG object

OUTDIAG Synopsis

DECLARE OBJECT *obj* **(OUTDIAG) ;**

Method syntax, in order of typical usage:

rc=*obj*.**Collect (***SourceObject* **) ;**
nrows=*obj*.**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.

Table 3.20 Contents of the OUTEST Object

Column	Type	Description
COMPMODEL	String	Component model name
COMPONENT	String	Component name within the component model
DSVAR	String	Corresponding variable name
EST	Numeric	Parameter estimate
FACTOR	Numeric	Factor number
LABEL	String	System-generated label for model
LAG	Numeric	Lag that is used
MODEL	String	Name of model in the selection list
MODELVAR	String	Symbol used in the model specification
NAME	String	Name of the dependent variable
PARM	String	Parameter name
PVALUE	Numeric	*p*-value for parameter estimate
SELECT	String	Name of the selection list
SHIFT	Numeric	Shift that is used
STDERR	Numeric	Parameter estimate standard error
TRANSFORM	String	Transform that is used for the dependent variable
TVALUE	Numeric	*t* statistic for parameter estimate
VARTYPE	String	Type of the variable (dependent or independent)

Table 3.21 summarizes the methods that are associated with the OUTEST Object.

Table 3.21 Methods of the OUTEST Object

Method	Description
Collect	Collect parameter estimates from the FORENG instance
nrows	Get the OUTEST row count

OUTEST Synopsis

DECLARE OBJECT *obj* **(OUTEST) ;**

Method syntax, in order of typical usage:

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

OUTEST Methods

OUTEST.Collect Method

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

Collects the parameter estimates 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 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.

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 3.22 shows the contents of the OUTFOR object.

Table 3.22 Contents of the OUTFOR Object

Column	Type	Description
NAME	String	Name of the dependent variable
TIMEID	Numeric	Uniform time ID values for series
ACTUAL	Numeric	Accumulated values of dependent variable
ERROR	Numeric	Residuals
LOWER	Numeric	Lower confidence limit
PREDICT	Numeric	Forecasts of dependent variable
STD	Numeric	Prediction standard error
UPPER	Numeric	Upper confidence limit

Table 3.23 summarizes the methods that are associated with the OUTFOR Object.

Table 3.23 Methods of the OUTFOR Object

Method	Description
Collect	Collect forecasts estimates from FORENG instance.
nrows	Get the OUTFOR row count.

OUTFOR Synopsis

DECLARE OBJECT *obj* **(OUTFOR) ;**

Method syntax, in order of typical usage:

rc=*obj*.**Collect (***FORENGObj* < ,*Region* > **) ;**
rc=*obj*.**nrows () ;**

OUTFOR Methods

OUTFOR.Collect Method

rc=*obj*.**Collect (***FORENGObj* < ,*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 3.24 shows the contents of the OUTCOMP object.

Table 3.24 Contents of the OUTCOMP Object

Column	Type	Description
ACTUAL	Numeric	Accumulated values of dependent variable
COMP	String	Name of the forecast component series
LOWER	Numeric	Lower confidence limit
NAME	String	Name of the dependent variable
PREDICT	Numeric	Forecasts of dependent variable
STD	Numeric	Prediction standard error
TIMEID	Numeric	Uniform time ID values for series
UPPER	Numeric	Upper confidence limit

Table 3.25 summarizes the methods that are associated with the OUTCOMP Object.

Table 3.25 Methods of the OUTCOMP Object

Method	Description
Collect	Collect model component series from the FORENG instance
nrows	Get the OUTCOMP row count

OUTCOMP Synopsis

DECLARE OBJECT *obj* **(OUTCOMP) ;**

Method syntax, in order of typical usage:

rc=*obj*.**Collect (***FORENGObj* < ,*Region* > **) ;**
rc=*obj*.**nrows () ;**

OUTCOMP Methods

OUTCOMP.Collect Method

rc=*obj*.**Collect (***FORENGObj* < ,*Region* > **) ;**

Collects the model component series 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 parameter estimates.

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.

OUTCOMP.nrows Method

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

Gets the current row count from the OUTCOMP instance.

Arguments

There are no arguments associated with this method.

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 3.26 shows the contents of the OUTINDEP object.

Table 3.26 Contents of the OUTINDEP Object

Column	Type	Description
NAME	String	Name of the dependent variable
TIMEID	Numeric	Uniform time ID values for series
XVAR	String	Name of the independent variable
X	Numeric	Independent (X) variable value

Table 3.27 summarizes the methods that are associated with the OUTINDEP Object.

Table 3.27 Methods of the OUTINDEP Object

Method	Description
Collect	Collect the independent variable series that are used in the forecast from the FORENG instance forecast object
nrows	Get the OUTINDEP row count

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* **) ;**

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.

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 3.28 shows the contents of the OUTMODELINFO object.

Table 3.28 Contents of the OUTMODELINFO Object

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

Table 3.29 summarizes the methods that are associated with the OUTMODELINFO Object.

Table 3.29 Methods of the OUTMODELINFO Object

Method	Description
Collect	Collect the selected model information from the FORENG instance *FORENGObj*
nrows	Get the current row count from the OUTMODELINFO instance

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 3.30 shows the contents of the OUTSELECT object.

Table 3.30 Contents of the OUTSELECT Object

Column	Type	Description
MODEL	String	Model specification name
NAME	String	Variable name
REGION	String	Region in which the statistics are calculated. Values in the _REGION_ variable include: **FIT** indicates that fit statistics were calculated over the fit region. **FORECAST** indicates that fit statistics were calculated over the forecast region.
SELECT	String	Name of model selection list to which _MODEL_ belongs
SELECTED	String	Indicates whether _MODEL_ was chosen to forecast the dependent series or used by the chosen forecast when the chosen forecast is a combined model. Values in the _SELECTED_ variable include: **NO** indicates that _MODEL_ is neither selected nor used. **YES** indicates that _MODEL_ is the primary model selected for the forecast. **USED** indicates that _MODEL_ is used by the primary model in producing the final forecast. **USED_SELECT** indicates that _MODEL_ is used by the primary model in the model selection region, but not in producing the final forecast.
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 percentage of standard deviation
GMAPPE	Numeric	Geometric mean absolute predictive percentage error

Table 3.30 *continued*

Column	Type	Description
GMRAE	Numeric	Geometric mean relative absolute error
GMASPE	Numeric	Geometric mean absolute symmetric percentage error
LABEL	String	Descriptive label for the variable name in _NAME_
MAE	Numeric	Mean absolute error
MAPE	Numeric	Mean absolute percentage error
MAPES	Numeric	Mean absolute error percentage of standard deviation
MAPPE	Numeric	Symmetric mean absolute predictive percentage error
MASE	Numeric	Mean absolute scaled error
MAXAPES	Numeric	Maximum absolute error percentage of standard deviation
MAXERR	Numeric	Maximum error
MAXPE	Numeric	Maximum percentage error
MAXPPE	Numeric	Maximum predictive percentage error
MAXRE	Numeric	Maximum relative error
MAXSPE	Numeric	Maximum symmetric percentage error
MDAPE	Numeric	Median absolute percentage error
MDAPES	Numeric	Median absolute error percentage of standard deviation
MDAPPE	Numeric	Median absolute predictive percentage error
MDASPE	Numeric	Median absolute symmetric percentage error
MDRAE	Numeric	Median relative absolute error
MPPE	Numeric	Mean predictive percentage error
ME	Numeric	Mean error
MINAPES	Numeric	Minimum absolute error percentage of standard deviation
MINERR	Numeric	Minimum error
MINPE	Numeric	Minimum percentage error
MINPPE	Numeric	Minimum predictive percentage error
MINRE	Numeric	Minimum relative error
MINSPE	Numeric	Minimum symmetric percentage error
MPE	Numeric	Mean percentage error
MRAE	Numeric	Mean relative absolute error

Table 3.30 *continued*

Column	Type	Description
MRE	Numeric	Mean relative error
MSE	Numeric	Mean square error
MSPE	Numeric	Mean symmetric percentage error
N	Numeric	Number of total observations
NMISSA	Numeric	Number of missing actual values
NMISSP	Numeric	Number of missing predicted values
NOBS	Numeric	Number of observations that were used
NPARMS	Numeric	Number of parameters
RMSE	Numeric	Root mean square error
RSQUARE	Numeric	R-square
RWRSQ	Numeric	Random walk R-square
SBC	Numeric	Schwarz Bayesian information criterion
SMAPE	Numeric	Symmetric mean absolute percentage error
SSE	Numeric	Sum of square error
SST	Numeric	Corrected total sum of squares
TSS	Numeric	Total sum of squares
UMSE	Numeric	Unbiased mean square error
URMSE	Numeric	Unbiased root mean square error

Table 3.31 summarizes the methods that are associated with the OUTSELECT Object.

Table 3.31 Methods of the OUTSELECT Object

Method	Description
Collect	Collect the model selection fit statistics from the FORENG instance *FORENGObj*
nrows	Get the current row count from the OUTSELECT instance

OUTSELECT Synopsis

DECLARE OBJECT *obj* **(OUTSELECT) ;**

Method syntax, in order of typical usage:

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

OUTSELECT Methods

OUTSELECT.Collect Method

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

Collects the model selection fit statistics 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 selection fit statistics.

OUTSELECT.nrows Method

rc=*obj*.**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 3.32 shows the contents of the OUTSTAT object.

Table 3.32 Contents of the OUTSTAT Object

Column	Type	Description
MODEL	String	Model specification name
NAME	String	Variable name
REGION	String	Region in which the statistics are calculated. Values in the _REGION_ variable include: **FIT** indicates that fit statistics were calculated over the fit region. **FORECAST** indicates that fit statistics were calculated over the forecast region.
SELECT	String	Name of model selection list to which _MODEL_ belongs
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 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 percentage of standard deviation
MAPPE	Numeric	Symmetric mean absolute predictive percentage error
MASE	Numeric	Mean absolute scaled error
MAXAPES	Numeric	Maximum absolute error percentage of standard deviation
MAXERR	Numeric	Maximum error
MAXPE	Numeric	Maximum percentage error
MAXPPE	Numeric	Maximum predictive percentage error

Table 3.32 *continued*

Column	Type	Description
MAXRE	Numeric	Maximum relative error
MAXSPE	Numeric	Maximum symmetric percentage error
MDAPE	Numeric	Median absolute percentage error
MDAPES	Numeric	Median absolute error percentage of standard deviation
MDAPPE	Numeric	Median absolute predictive percentage error
MDASPE	Numeric	Median absolute symmetric percentage error
MDRAE	Numeric	Median relative absolute error
MPPE	Numeric	Mean predictive percentage error
ME	Numeric	Mean error
MINAPES	Numeric	Minimum absolute error percentage of standard deviation
MINERR	Numeric	Minimum error
MINPE	Numeric	Minimum percentage error
MINPPE	Numeric	Minimum predictive percentage error
MINRE	Numeric	Minimum relative error
MINSPE	Numeric	Minimum symmetric percentage error
MPE	Numeric	Mean percentage error
MRAE	Numeric	Mean relative absolute error
MRE	Numeric	Mean relative error
MSE	Numeric	Mean square error
MSPE	Numeric	Mean symmetric percentage error
N	Numeric	Number of total observations
NMISSA	Numeric	Number of missing actual values
NMISSP	Numeric	Number of missing predicted values
NOBS	Numeric	Number of observations that were used
NPARMS	Numeric	Number of parameters
RMSE	Numeric	Root mean square error
RSQUARE	Numeric	R-square
RWRSQ	Numeric	Random walk R-square
SBC	Numeric	Schwarz Bayesian information criterion

Table 3.32 *continued*

Column	Type	Description
SMAPE	Numeric	Symmetric mean absolute percentage error
SSE	Numeric	Sum of square error
SST	Numeric	Corrected total sum of squares
TSS	Numeric	Total sum of squares
UMSE	Numeric	Unbiased mean square error
URMSE	Numeric	Unbiased root mean square error

Table 3.33 summarizes the methods that are associated with the OUTSTAT Object.

Table 3.33 Methods of the OUTSTAT Object

Method	Description
Collect	Collect the statistics of fit from the FORENG instance *FORENGObj*
nrows	Get the current row count from the OUTSTAT instance

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 (***FORENGObj* **) ;**

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 () ;**

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 3.34 shows the contents of the OUTFMSG object.

Table 3.34 Contents of the OUTFMSG Object

Column	Type	Description
FMSGPEC	String	FMSG specification XML document
NAME	String	Name of the dependent variable (can be missing)
SPECLEN	Numeric	Length of the XML specification for FMSG
SPECNAME	String	Name of the FMSG specification

Table 3.35 summarizes the methods that are associated with the OUTFMSG Object.

Table 3.35 Methods of the OUTFMSG Object

Method	Description
Collect	Collect forecast model selection graph for output to a CAS table
nrows	Query the OUTFMSG object for its current row count

OUTFMSG Synopsis

DECLARE OBJECT *obj* **(OUTFMSG) ;**

Method syntax, in order of typical usage:

rc=*obj*.**Collect (***SourceObject* **) ;**
rc=*obj*.**nrows () ;**

OUTFMSG Methods

OUTFMSG.Collect Method

rc=*obj*.**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*.**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 3.36 shows the contents of the OUTSCORE object.

Table 3.36 Contents of OUTSCORE Object

Column	Type	Description
NAME	String	Name of the dependent variable (might be missing)
SCORESPEC	String	Score specification XML document
SPECLEN	Numeric	Length of the score specification XML
SPECNAME	String	Name of the score specification

Table 3.37 summarizes the methods that are associated with the OUTSCORE object.

Table 3.37 Methods of the OUTSCORE Object

Method	Description
Collect	Collect forecast model score XML and store it in a CAS table
nrows	Query the OUTSCORE object for its current row count

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=*obj*.**Collect (***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=*obj*.**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 3.38 summarizes the methods that are associated with the INDIAG Object.

Table 3.38 Methods of the INDIAG Object

Method	Description
nrows	Query INDIAG object for its current row count

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 3.39 summarizes the methods that are associated with the INFMSG Object.

Table 3.39 Methods of the INFMSG Object

Method	Description
nrows	Return the number of rows in the INFMSG object. A returned missing value indicates that the INFMSG object has not been successfully configured.

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.

INEST 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 3.40 summarizes the methods that are associated with the INEST object.

Table 3.40 Methods of the INEST Object

Method	Description
nrows	Return the number of rows in the INEST object. A returned missing value indicates that the INEST object has not been successfully configured.

INEST Synopsis

DECLARE OBJECT *obj* **(INEST) ;**

Method syntax:

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

INEST Methods

INEST.nrows Method

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

Returns the number of rows in the INESTobject. A returned missing value indicates that the INEST 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 3.41 summarizes the methods that are associated with the INSCORE object.

Table 3.41 Methods of the INSCORE Object

Method	Description
nrows	Return the number of rows in the INSCORE object. A returned missing value indicates that the INSCORE object has not been successfully configured.

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 4, "The TSMODEL Procedure" (*SAS Visual Forecasting: Forecasting Procedures*).

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.

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

proc tsmodel 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.

Output 3.1.1 Information about the Selected Model (Partial Output)

Choice Between the ESM and ARIMA Models

MODEL	_SELECTED_	_LABEL_
ARIMAX	NO	ARIMA: AIR ~ P = 1 D = (1,12) NOINT
ESM	YES	Winters Method (Multiplicative)

Output 3.1.2 Parameter Estimates of the Selected Model (Partial Output)

Parameter Estimates for the Multiplicative Winters Model

COMPONENT	_EST_	_STDERR_	_TVALUE_	_PVALUE_
LEVEL	0.3073	0.0315	9.7448	1.8E-17
TREND	0.00100	0.00302	0.3307	0.7413
SEASON	0.8749	0.0777	11.2612	2.2E-21

Output 3.1.3 Forecasts Based on the Selected Model (Partial Output)

Forecasts Based On the Multiplicative Winters Model

DATE	PREDICT	STD	UPPER	LOWER
01JAN61	445.3	10.6910	466.3	424.3
01FEB61	418.1	11.1235	439.9	396.3
01MAR61	464.1	11.7900	487.2	441.0
01APR61	494.0	12.4171	518.4	469.7
01MAY61	505.0	12.9152	530.3	479.6
01JUN61	572.6	13.9646	600.0	545.2
01JUL61	662.7	15.3576	692.8	632.6
01AUG61	653.8	15.5579	684.3	623.3
01SEP61	545.9	14.4949	574.3	517.5
01OCT61	487.7	14.0959	515.3	460.1
01NOV61	415.3	13.5185	441.8	388.8
01DEC61	459.6	14.5119	488.0	431.2

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.

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

```
proc tsmodel 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 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 3.1.2 shows the parameter estimates of the selected model, and Output 3.1.3 shows the forecasts according to the selected model.

Output 3.2.1 General Model Description

Description of the Model

MODEL	_MODELTYPE_	_DEPTRANS_	_SEASONAL_	_TREND_	_EVENTS_	_OUTLIERS_
ARIMAX	ARIMA	NONE	1	1	1	0

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

COMPONENT	_EST_	_STDERR_	_TVALUE_	_PVALUE_
LEVEL	0.3073	0.0315	9.7448	1.8E-17
TREND	0.00100	0.00302	0.3307	0.7413
SEASON	0.8749	0.0777	11.2612	2.2E-21

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

DATE	PREDICT	STD	UPPER	LOWER
01JAN61	445.3	10.6910	466.3	424.3
01FEB61	418.1	11.1235	439.9	396.3
01MAR61	464.1	11.7900	487.2	441.0
01APR61	494.0	12.4171	518.4	469.7
01MAY61	505.0	12.9152	530.3	479.6
01JUN61	572.6	13.9646	600.0	545.2
01JUL61	662.7	15.3576	692.8	632.6
01AUG61	653.8	15.5579	684.3	623.3
01SEP61	545.9	14.4949	574.3	517.5
01OCT61	487.7	14.0959	515.3	460.1
01NOV61	415.3	13.5185	441.8	388.8
01DEC61	459.6	14.5119	488.0	431.2

References

Box, G. E. P., and Jenkins, G. M. (1976). *Time Series Analysis: Forecasting and Control*. Rev. ed. San Francisco: Holden-Day.

Chapter 4
Simple Forecast Service Package

Contents

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 4.1 summarizes the single object class in the SFS package.

Table 4.1 Object in the Simple Forecast Service Package

Object	Description
SFS	Automatically model and forecast univariate time series.

Common Argument Types

Table 4.2 defines the common argument types that are used in this chapter. The symbol *x* corresponds to the variable name.

Table 4.2 Common Argument Types

SAS Data Type	Declaration Syntax
String	LENGTH *x* **$***n***;**
Numeric	*x* or LENGTH *x 8***;**
Numeric array	ARRAY *x***[***n***]**/NOSYMBOLS**;**
Status	*x* or LENGTH *x 8***;**

Return Codes

Table 4.3 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 4.3 Return Codes

Status	Meaning
< 0	An unrecoverable error occurred. No result was produced.
= 0	Unconditional success. The requested action completed and a normal result was produced.
> 0	Conditional success or warning. A result was produced subject to conditions.

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 4, "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 4.4 summarizes the methods that are associated with the SFS object.

Table 4.4 Methods of the SFS Object

Method	Description
Initialize	Initialize or reset the SFS object for use
SetY	Specify the dependent time series
AddX	Add an independent time series
SetOption	Specify a computational option
Run	Automatically model and forecast the dependent variable
GetForecast	Retrieve a computed forecast series by name
nfor	Get the length (observation count) of the forecast series
criterion	Get the in-sample Bayesian information criterion (SBC) fit statistic for the selected time series model
model	Get the short name of the selected model that is used to produce the final forecast

SFS Synopsis

DECLARE OBJECT *obj* **(SFS) ;**

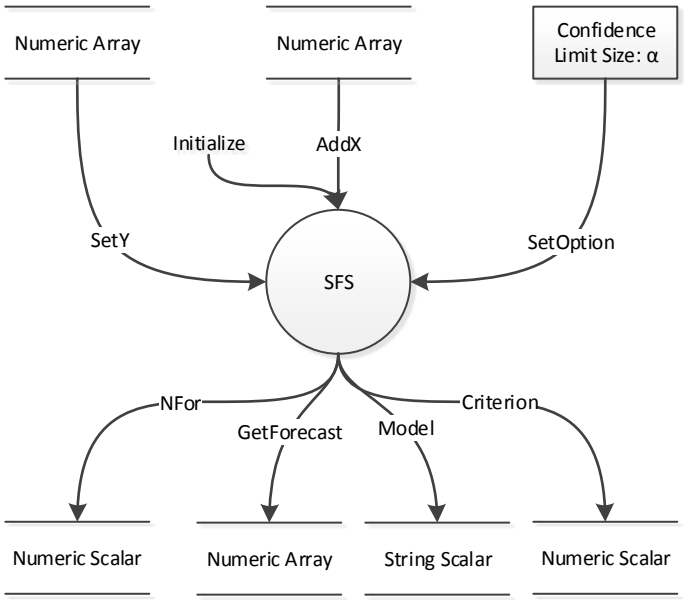
Each SFS object in your program is independently instantiated for each BY-group thread. Also, each SFS object in your program is automatically reset at the start and end of each BY group.

Method syntax, in order of typical usage:

rc=*obj*.**Initialize** () ;
rc=*obj*.**SetY** (*YSeries*) ;
rc=*obj*.**AddX** (*XSeries*) ;
rc=*obj*.**SetOption** ('*Name*',*Value*) ;
rc=*obj*.**Run** () ;
rc=*obj*.**GetForecast** (*Which,Result*) ;
rc=*obj*.**nfor** () ;
rc=*obj*.**criterion** () ;
rc=*obj*.**model** () ;

Figure 4.1 outlines the programmatic data flow through the SFS object; each arrow represents a different object method.

Figure 4.1 SFS Object Data Flow



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

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 a 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

rc=*obj*.**Initialize () ;**

Initializes or resets the SFS instance.

Arguments

There are no arguments associated with this method.

SFS.model Method

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 4.5 shows the model families that can be considered during the process.

Table 4.5 Model Families for the SFS.model Method

Family	Description
ARIMAX	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)
ESM	Exponential smoothing model
REGARIMA	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)

Arguments

There are no arguments associated with this method.

SFS.nfor Method

rc=*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=*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 4.6 shows the model families that can be considered during the process.

Table 4.6 Model Families for the SFS.Run Method

Family	Description
ARIMAX	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)
ESM	Exponential smoothing model
REGARIMA	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)

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 4, "The TSMODEL Procedure" (*SAS Visual Forecasting: Forecasting Procedures*).

Example 4.1: 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.

```
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 4.1.1.

Output 4.1.1 Summary of Time Series Processing for mycas.air

The TSMODEL Procedure

Summary of time series processing for AIR	
Number of analysis variables	1
Number of rows read	288
Number of groups read	1
Memory for group packages (KB)	10
Time to read groups (seconds)	0.0002150536
Time to load groups (seconds)	0.0040991306
Number of data threads	3
Thread BY group redundancy	3
Minimum time ID	JAN1949
Maximum time ID	DEC1960
Minimum time periods	144
Maximum time periods	144
Number of nodes run	1
Number of nodes with data	1
Number of nodes with groups	1
Number of threads budgeted	32
Minimum thread group count	1
Maximum thread group count	1
Minimum threads active	1
Maximum threads active	1
Average CPU utilization of nodes with data (%)	100
Minimum CPU utilization of nodes with data (%)	100
Maximum CPU utilization of nodes with data (%)	100
Number of groups processed by submitted code	1
Number of groups failing	0
Elapsed time to process groups (seconds)	0.0638198853
Number of array table rows produced	156
Number of scalar table rows produced	1

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 4.1.2.

```
proc print data=mycas.airmonos;
run;
```

Output 4.1.2 OUTSCALAR Table

Obs	_STATUS_	nfor	fitstat	rc
1	0	156	10.579085435	0

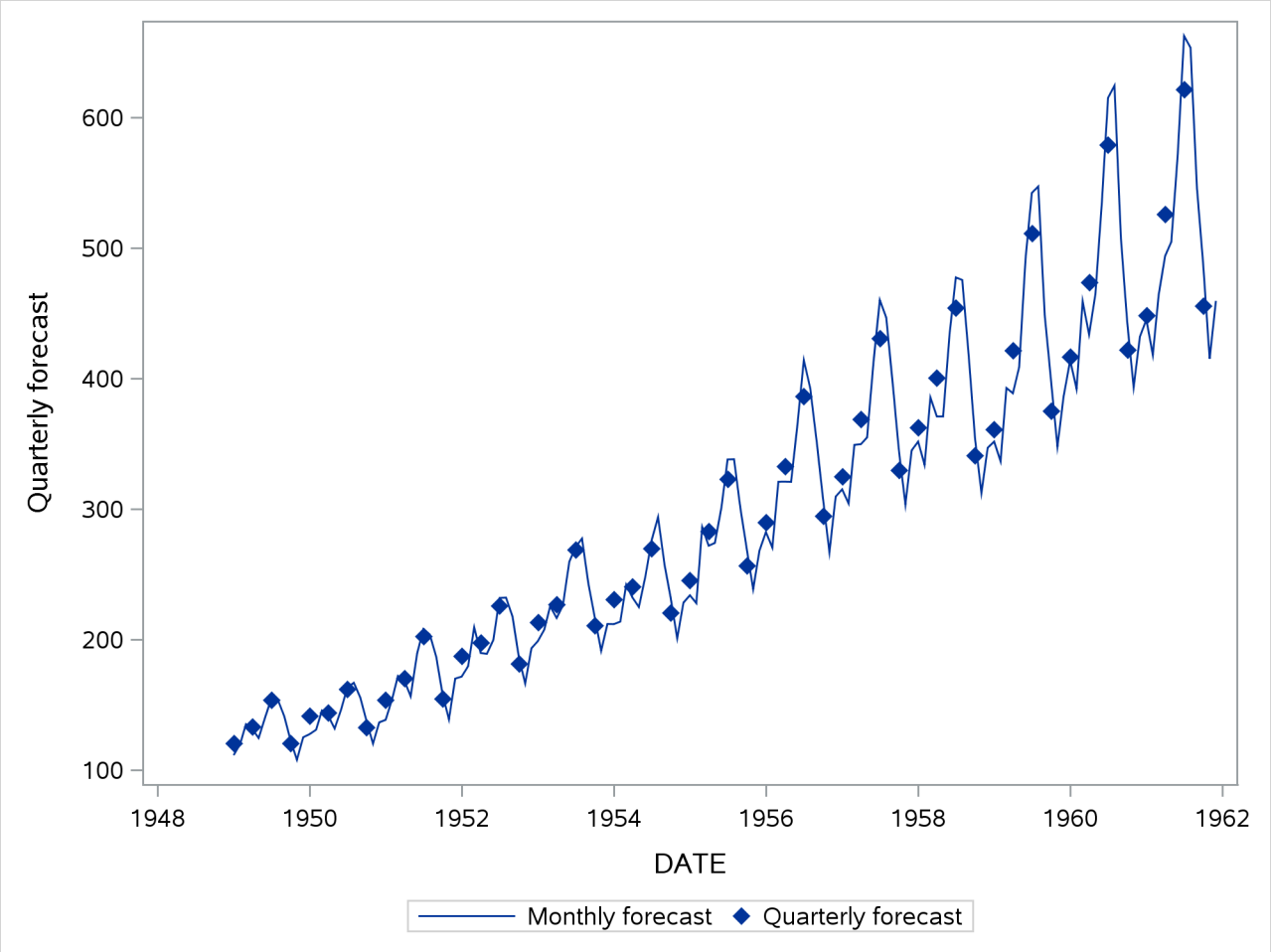
Suppose you want to forecast the quarterly average airline passenger data for a year into the future. The following simple modifications to the SAS code create the quarterly forecast for a four-quarter horizon:

```
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 4.1.3.

```
data airboth;
   label airmonfor='Monthly forecast';
   label airqtrfor='Quarterly forecast';
   merge mycas.airmonoa mycas.airqtroa;
   by date;
run;

proc sgplot data=airboth;
   series x=date y=airmonfor;
   scatter x=date y=airqtrfor/markerattrs=(symbol=DiamondFilled);
run;
```

Output 4.1.3 Monthly and Quarterly Airline Passenger Forecasts



Example 4.2: 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.

```
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.

```
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 4.2.1.

Output 4.2.1 Summary of Time Series Processing for mycas.pricedata

The TSMODEL Procedure

Summary of time series processing for PRICEDATA	
Number of analysis variables	3
Number of rows read	1020
Number of groups read	17
Memory for group packages (KB)	102
Time to read groups (seconds)	0.0015509129
Time to load groups (seconds)	0.0053670406
Number of data threads	3
Thread BY group redundancy	1.0588235294
Minimum time ID	JAN1998
Maximum time ID	DEC2002
Minimum time periods	60
Maximum time periods	60
Number of nodes run	1
Number of nodes with data	1
Number of nodes with groups	1
Number of threads budgeted	32
Minimum thread group count	1
Maximum thread group count	1
Minimum threads active	1
Maximum threads active	17
Average CPU utilization of nodes with data (%)	80.44
Minimum CPU utilization of nodes with data (%)	80.44
Maximum CPU utilization of nodes with data (%)	80.44
Number of groups processed by submitted code	17
Number of groups failing	0
Elapsed time to process groups (seconds)	0.0532197952
Number of array table rows produced	1224
Number of scalar table rows produced	17

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 4.2.2.

```
proc print data=mycas.saleos;
run;
```

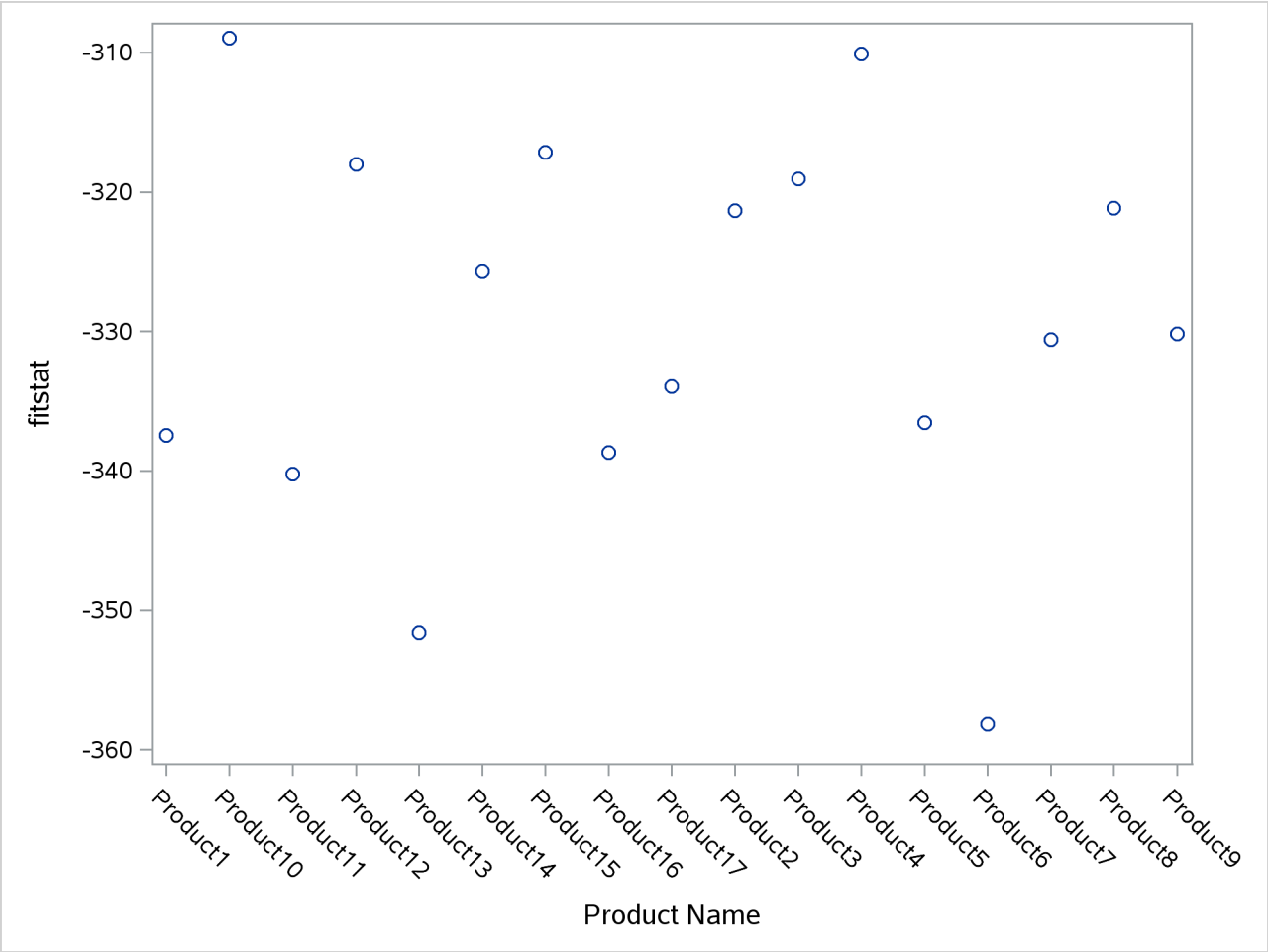
Output 4.2.2 OUTSCALAR Table

Obs	productName	_STATUS_	sbase	pbase	nfor	fitstat	rc	model
1	Product1	0	355	52.3	72	-337.4629078	0	REGARIMA
2	Product10	0	329	59	72	-308.9521995	0	REGARIMA
3	Product11	0	240	65.2	72	-340.2472948	0	REGARIMA
4	Product12	0	413	147	72	-318.0312143	0	REGARIMA
5	Product13	0	359	122	72	-351.6217156	0	REGARIMA
6	Product14	0	462	53	72	-325.7278047	0	REGARIMA
7	Product15	0	383	120.2	72	-317.1505661	0	REGARIMA
8	Product16	0	491	70.55	72	-338.6653664	0	REGARIMA
9	Product17	0	359	80.5	72	-333.9525131	0	REGARIMA
10	Product2	0	373	115	72	-321.3360261	0	REGARIMA
11	Product3	0	300	33.4	72	-319.0745186	0	REGARIMA
12	Product4	0	418	67.9	72	-310.1082484	0	REGARIMA
13	Product5	0	416	36	72	-336.5399506	0	REGARIMA
14	Product6	0	550	38.88	72	-358.1769437	0	REGARIMA
15	Product7	0	435	42	72	-330.6016099	0	REGARIMA
16	Product8	0	404	56.9	72	-321.1618287	0	REGARIMA
17	Product9	0	461	171.4	72	-330.1588485	0	REGARIMA

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 4.2.3.

```
proc sort data=mycas.saleos out=saleos;
   by productName;
run;
proc sgplot data=saleos;
   scatter x=productName y=fitstat;
run;
```

Output 4.2.3 Fit Statistics by Product



Example 4.3: 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 4.2. 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: CAS Procedure Programming Guide and 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 4.2, 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=mycas; 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 4.3.1.

Output 4.3.1 Summary of Time Series Processing for mycas.pricedata

Results from timeData.runTimeCode

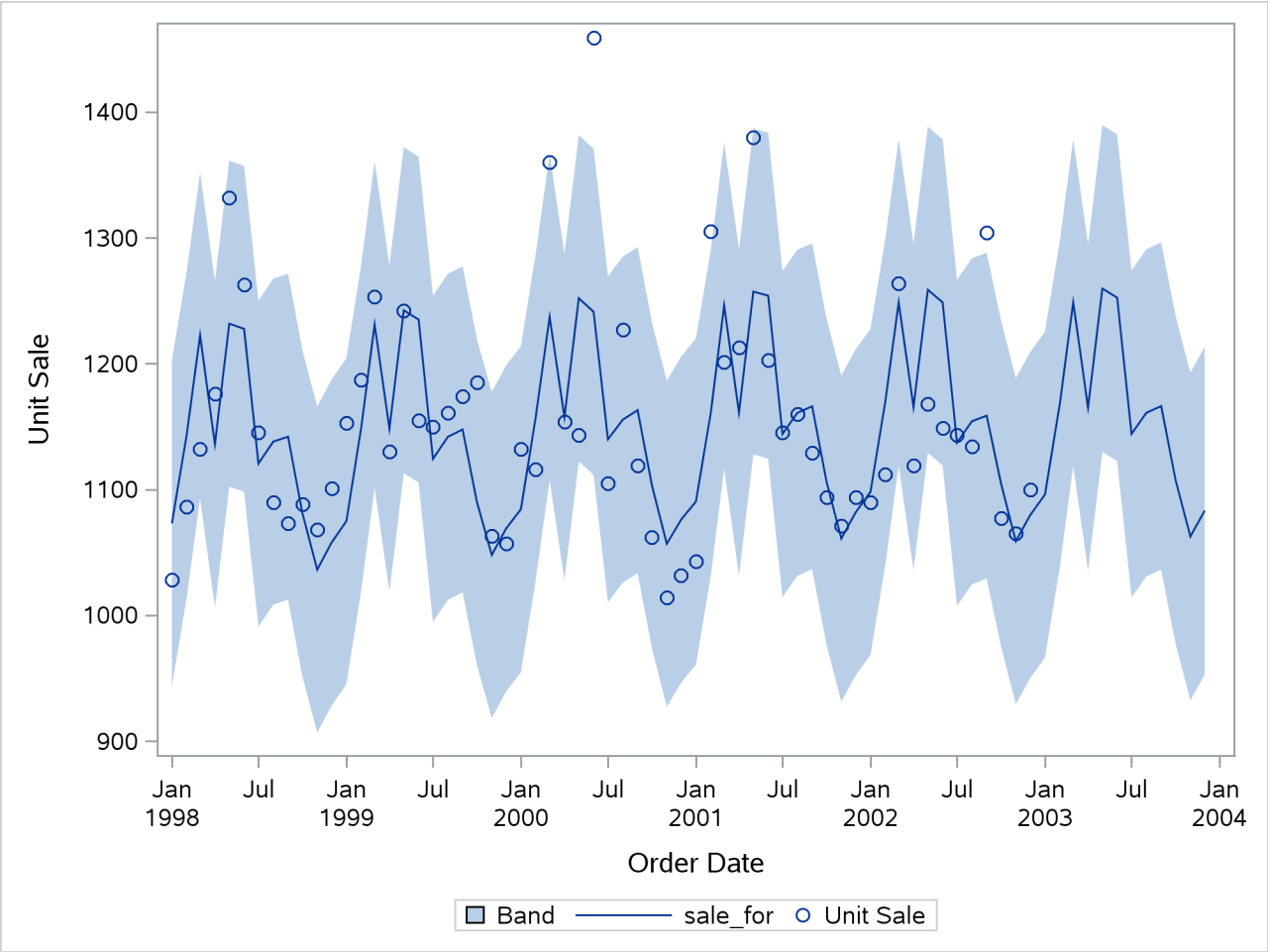
Summary of time series processing for PRICEDATA	
Number of analysis variables	1
Number of rows read	2040
Number of groups read	3
Memory for group packages (KB)	8
Time to read groups (seconds)	0.0019900799
Time to load groups (seconds)	0.006177187
Number of data threads	3
Thread BY group redundancy	1.3333333333
Minimum time ID	JAN1998
Maximum time ID	DEC2002
Minimum time periods	60
Maximum time periods	60
Number of nodes run	1
Number of nodes with data	1
Number of nodes with groups	1
Number of threads budgeted	32
Minimum thread group count	1
Maximum thread group count	1
Minimum threads active	1
Maximum threads active	3
Average CPU utilization of nodes with data (%)	71.04
Minimum CPU utilization of nodes with data (%)	71.04
Maximum CPU utilization of nodes with data (%)	71.04
Number of groups processed by submitted code	3
Number of groups failing	0
Elapsed time to process groups (seconds)	0.0085771084
Number of array table rows produced	216
Number of scalar table rows produced	3

The following SAS code generates a series plot of the forecasts and confidence limits for region 1.

```
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 4.3.2.

Output 4.3.2 Sales Forecasts for Region 1



Chapter 5
Singular Spectrum Analysis Package

Contents

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 Sasioca 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: Language Reference*.

SSA Package Summary

Table 5.1 summarizes the objects in the SSA package.

Table 5.1 Objects in the Singular Spectrum Analysis Package

Object	Description
SSA	Perform singular spectrum analysis of a time series
OUTSSA	Collect output from an SSA object
OUTSV	Collect singular values from an SSA object

Return Codes

Table 5.2 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 5.2 Return Codes

Status	Meaning
< 0	An unrecoverable error occurred. No result was produced.
= 0	Unconditional success. The requested action completed and a normal result was produced.
> 0	Conditional success or warning. A result was produced subject to conditions.

Upon returning a negative return 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 4, “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.

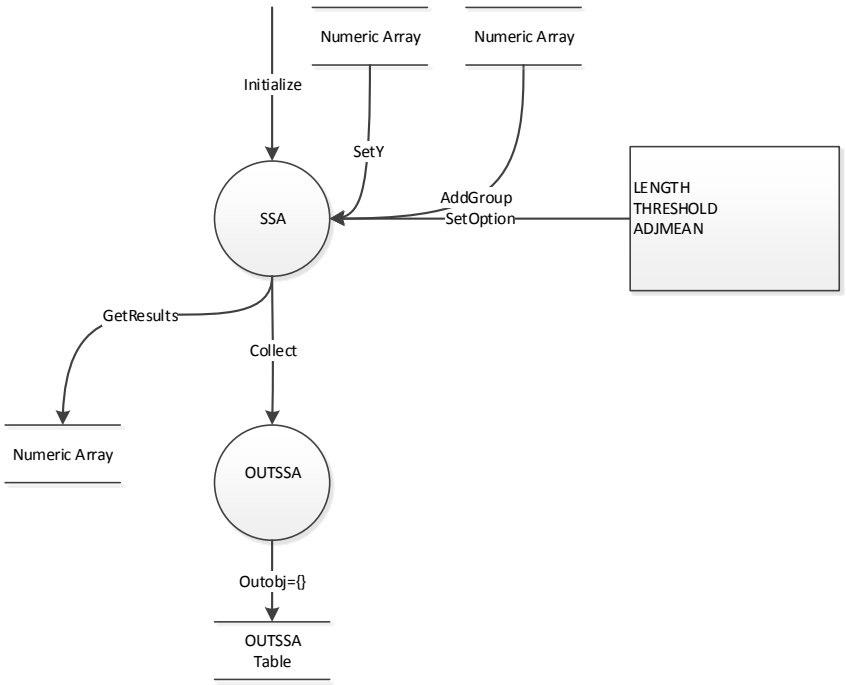
Table 5.3 summarizes the methods that are associated with the SSA object.

Table 5.3 Methods of the SSA Object

Method	Description
AddGroup	Add groupings manually
GetResult	Get the decomposition result
Initialize	Initialize an SSA instance
SetOption	Set a named option
SetY	Set the input time series
Run	Run the singular spectrum analysis

Figure 5.1 diagrams the methods of the SSA object.

Figure 5.1 SSA Data Flow



SSA Synopsis

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=*obj*.**AddGroup (***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=*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.

SSA.Initialize Method

rc=*obj*.**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=*obj*.**SetOption** ('*Name*',*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 *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* that specifies the window length to be used. The default is T/4, where T is the length of *YSeries*, 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* that 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

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.

OUTSSA Object

The OUTSSA object collects output from SSA object.

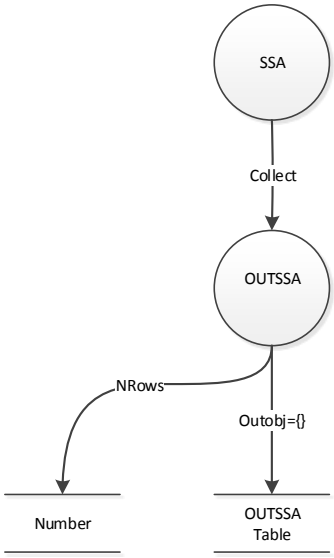
Table 5.4 summarizes the methods that are associated with the OUTSSA object.

Table 5.4 Methods of the OUTSSA Object

Method	Description
Collect	Collect the results of an SSA object
nrows	Get the current row count from the OUTSSA object

Figure 5.2 diagrams the methods of the OUTSSA object.

Figure 5.2 OUTSSA Data Flow



OUTSSA Synopsis

DECLARE OBJECT *obj* **(OUTSSA) ;**

Method syntax, in order of typical usage:

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

OUTSSA Methods

OUTSSA.Collect Method

rc=*obj*.**Collect () ;**

Collects the output of singular spectrum analysis from an SSA object and saves the results to a CAS table.

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.

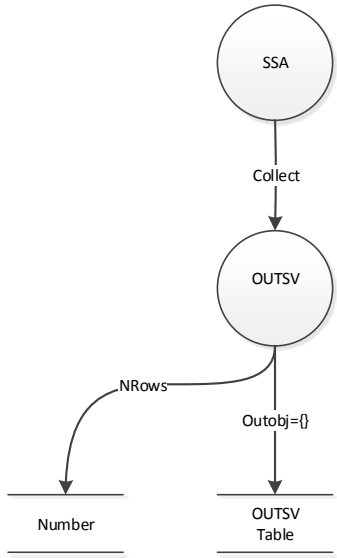
Table 5.5 summarizes the methods that are associated with the OUTSV object.

Table 5.5 Methods of the OUTSV Object

Method	Description
Collect	Collect the singular values of an SSA object
nrows	Get the current row count from the OUTSV object

Figure 5.3 diagrams the methods of the OUTSV object.

Figure 5.3 OUTSV Data Flow



OUTSV Synopsis

DECLARE OBJECT *obj* **(OUTSV) ;**

Method syntax, in order of typical usage:

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

OUTSV Methods

OUTSV.Collect Method

rc=*obj*.**Collect () ;**

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

nrows=*obj*.**nrows () ;**

Gets the current row count from the OUTSV 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, $2 \leq L < T/2$, singular spectrum analysis Golyandina, Nekrutkin, and Zhigljavsky (2001) decomposes the time series into spectral groupings by using the following steps:

Step1. Embedding

Using the time series, form a $K \times L$ trajectory matrix, $\mathbf{X}$, with elements

$$\mathbf{X} = \{x_{k,l}\}_{k=1,\ l=1}^{K,L}$$

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 $2 \leq L < T/2$.

Step2. Decomposition

Apply singular value decomposition to the trajectory matrix,

$$\mathbf{X} = \mathbf{UQV}$$

where $\mathbf{U}$ represents the $K \times L$ matrix that contains the left-hand-side (LHS) eigenvectors, $\mathbf{Q}$ represents the diagonal $L \times L$ matrix that contains the singular values, and $\mathbf{V}$ represents the $L \times L$ matrix that contains the right-hand-side (RHS) eigenvectors. Therefore,

$$\mathbf{X} = \sum_{l=1}^{L} \mathbf{X}^{(l)} = \sum_{l=1}^{L} \mathbf{u}_l q_l \mathbf{v}_l'$$

where $\mathbf{X}^{(l)}$ represents the $K \times L$ principal component matrix, $\mathbf{u}_l$ represents the $K \times 1$ left-hand-side (LHS) eigenvector, q_l represents the singular value, and $\mathbf{v}_l$ represents the $L \times 1$ right-hand-side (RHS) eigenvector that is associated with the lth window index.

Step3. Grouping

For each group index, $m = 1, \ldots, M$, define a group of window indices $I_m \subset \{1, \ldots, L\}$. Let the following equation represent the grouped trajectory matrix for group I_m:

$$\mathbf{X}_{I_m} = \sum_{l \in I_m} \mathbf{X}^{(l)} = \sum_{l \in I_m} \mathbf{u}_l q_l \mathbf{v}_l'$$

If groupings represent a spectral partition,

$$\bigcup_{m=1}^{M} I_m = \{1, \ldots, L\} \qquad \text{and} \qquad I_m \cap I_n = \emptyset \quad \text{for} \quad m \neq n$$

then according to the singular value decomposition theory,

$$\mathbf{X} = \sum_{m=1}^{M} \mathbf{X}_{I_m}$$

Step4. Averaging

For each group index, $m = 1, \ldots, M$, compute the diagonal average of $\mathbf{X}_{I_m}$,

$$\tilde{x}_t^{(m)} = \frac{1}{n_t} \sum_{l=s_t}^{e_t} x_{t-l+1,l}^{(m)}$$

where

$$\begin{array}{llll} s_t = 1, & e_t = t, & n_t = t & \text{for} \quad 1 \leq t < L \\ s_t = 1, & e_t = L, & n_t = L & \text{for} \quad L \leq t \leq T - L + 1 \\ s_t = T - t - 1, & e_t = L, & n_t = T - t + 1 & \text{for} \quad T - L + 1 < t \leq T \end{array}$$

If the groupings represent a spectral partition, then by definition

$$y_t = \sum_{m=1}^{M} \tilde{x}_t^{(m)}$$

Hence, singular spectrum analysis additively decomposes the original time series, y_t, into m component series $\tilde{x}_t^{(m)}$ 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:

$$\rho_{i,j}^{(w)} = \frac{\left(\tilde{x}_t^{(i)}, \tilde{x}_t^{(j)}\right)_w}{||\tilde{x}_t^{(i)}, \tilde{x}_t^{(i)}||_w ||\tilde{x}_t^{(j)}, \tilde{x}_t^{(j)}||_w}$$

where $\left(\tilde{x}_t^{(i)}, \tilde{x}_t^{(j)}\right)_w = \sum_{t-1}^{T} w_t \tilde{x}_t^{(i)} \tilde{x}_t^{(j)}$ and $w_t = \min(t, L, T - t)$.

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

$$\left(\sum_{l=1}^{l_\alpha - 1} q_l \Big/ \sum_{l=1}^{L} q_l \right) \geq \alpha \qquad 1 < l_\alpha \leq L$$

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 'THRESHOLDPCT' argument are based on the singular values that appear on the diagonal of $\mathbf{Q}$. Alternatively, each principal component's contribution to variation in the series can be quantified by using the squares of the singular values. An OUTSV object collects the singular values from an SSA object.

Automatic Grouping

Besides specifying the groups explicitly, you can also specify the value of AUTO for 'THRESHOLDPCT' argument in order to perform the automatic grouping. The group number is specified by the '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: $\tilde{x}_t^{(m)}$ for $m = 1, \ldots, L$.
3. The weighted correlations (w-correlations) between groups are computed: $\rho_{i,j}^{(m)}$.
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 $|\rho_{i,j}^{(m)}| \approx 1$ whenever $i, j \in I_m$.

Examples: SSA Package

Example 5.1: 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 5.1.2. Output 5.1.1 shows the values of each components.

```
proc tsmodel data=mycas.air outobj=(os=mycas.analytic (replace=YES));
   id date interval=month;
   var air;
   require ssa;
   submit;
      declare object s(ssa);
      declare object os(outssa);
      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 = os.Collect(s);
   endsubmit;
run;
```

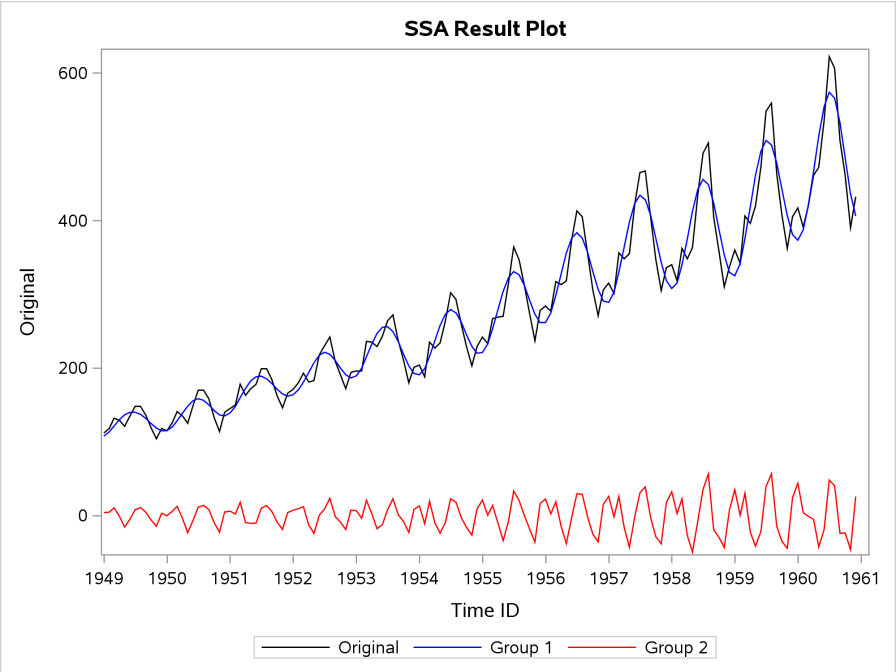
Output 5.1.1 SSA Results of AIR Data

SSA Result Table

Obs	_NAME_	DATE	ORIGINAL	GROUPSUM	GROUP1	GROUP2	GROUP3	GROUP4
1	AIR	JAN1949	112	112	107.86146454	4.1385354609	.	.
2	AIR	FEB1949	118	118	113.40770921	4.5922907916	.	.
3	AIR	MAR1949	132	132	121.50939907	10.490600933	.	.
4	AIR	APR1949	129	129	130.04589081	-1.045890813	.	.
5	AIR	MAY1949	121	121	136.62206663	-15.62206663	.	.
6	AIR	JUN1949	135	135	139.92109951	-4.921099508	.	.
7	AIR	JUL1949	148	148	140.0182405	7.9817594987	.	.
8	AIR	AUG1949	148	148	137.10648141	10.893518593	.	.
9	AIR	SEP1949	136	136	131.6108694	4.3891305963	.	.
10	AIR	OCT1949	119	119	124.86020336	-5.860203358	.	.
11	AIR	NOV1949	104	104	118.55785911	-14.55785911	.	.
12	AIR	DEC1949	118	118	114.83044425	3.1695557535	.	.
13	AIR	JAN1950	115	115	115.20424909	-0.20424909	.	.
14	AIR	FEB1950	126	126	120.16159642	5.8384035788	.	.
15	AIR	MAR1950	141	141	128.51838033	12.481619674	.	.
16	AIR	APR1950	135	135	138.69641074	-3.696410743	.	.
17	AIR	MAY1950	125	125	148.31164461	-23.31164461	.	.
18	AIR	JUN1950	149	149	155.63397881	-6.633978805	.	.
19	AIR	JUL1950	170	170	158.43892976	11.561070245	.	.
20	AIR	AUG1950	170	170	156.15977587	13.840224125	.	.
21	AIR	SEP1950	158	158	149.98327684	8.0167231632	.	.
22	AIR	OCT1950	133	133	142.3316826	-9.331682603	.	.

Obs	GROUP5	GROUP6	GROUP7	GROUP8	GROUP9	GROUP10
1	.	.	.	.	.	.
2	.	.	.	.	.	.
3	.	.	.	.	.	.
4	.	.	.	.	.	.
5	.	.	.	.	.	.
6	.	.	.	.	.	.
7	.	.	.	.	.	.
8	.	.	.	.	.	.
9	.	.	.	.	.	.
10	.	.	.	.	.	.
11	.	.	.	.	.	.
12	.	.	.	.	.	.
13	.	.	.	.	.	.
14	.	.	.	.	.	.
15	.	.	.	.	.	.
16	.	.	.	.	.	.
17	.	.	.	.	.	.
18	.	.	.	.	.	.
19	.	.	.	.	.	.
20	.	.	.	.	.	.
21	.	.	.	.	.	.
22	.	.	.	.	.	.

Output 5.1.2 SSA Results of AIR Data



Example 5.2: 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 5.2.3 shows the resulting groupings together with the original data. Output 5.2.1 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 5.2.2 shows the values of each components. Output 5.2.3 plots each component together with the original data.

```
proc tsmodel data=mycas.air outobj=(os=mycas.analytic (replace=YES)
                                    osv=mycas.sv (replace=YES));
   id date interval=month;
   var air;
   require ssa;
   submit;
      declare object s(ssa);
      declare object os(outssa);
      declare object osv(outsv);
      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);
   endsubmit;
run;
```

Output 5.2.1 Singular Values of AIR Data

Singular Values Table

Obs	_NAME_	SINGULARVALUE
1	AIR	11857.417652
2	AIR	1026.3522048
3	AIR	1024.2750479
4	AIR	532.52047187
5	AIR	528.49637245
6	AIR	230.9898283
7	AIR	222.84650302
8	AIR	184.1226536
9	AIR	180.98734598
10	AIR	151.90912581
11	AIR	150.75654992
12	AIR	63.028435941

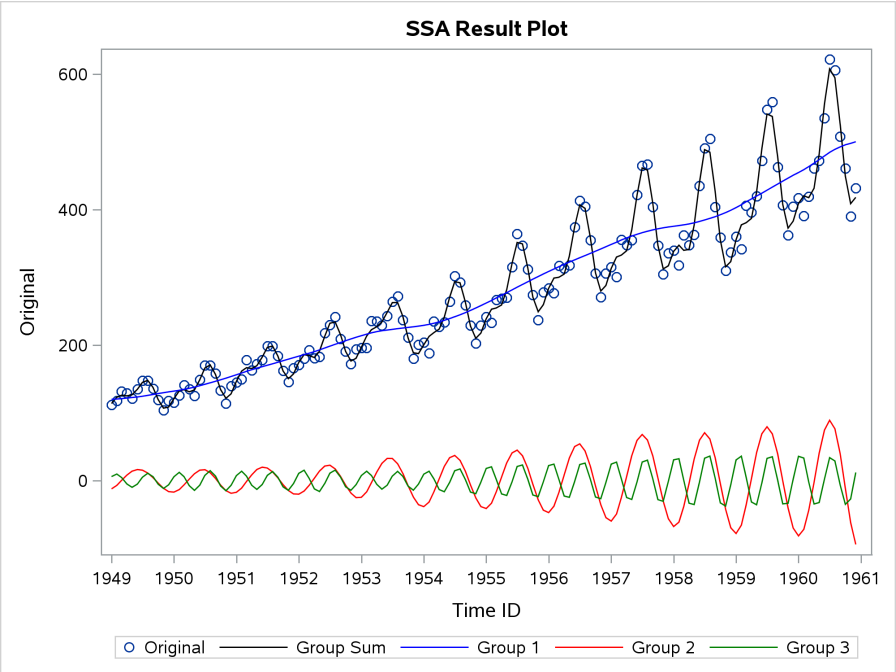
Output 5.2.2 SSA Results of AIR Data

SSA Result Table

Obs	_NAME_	DATE	ORIGINAL	GROUPSUM	GROUP1	GROUP2	GROUP3	GROUP4
1	AIR	JAN1949	112	113.8531969	120.04182977	-12.18036523	5.9917323634	.
2	AIR	FEB1949	118	123.16083396	120.45899832	-7.051289108	9.7531247537	.
3	AIR	MAR1949	132	126.12907055	121.08416438	0.425234683	4.6196714854	.
4	AIR	APR1949	129	125.12079135	121.86780976	8.1780810532	-4.925099464	.
5	AIR	MAY1949	121	126.75754898	122.67784605	13.944220575	-9.864517643	.
6	AIR	JUN1949	135	134.74542218	123.5194765	16.401623004	-5.175677331	.
7	AIR	JUL1949	148	145.1432656	124.57419013	15.444050369	5.1250250962	.
8	AIR	AUG1949	148	147.98972107	125.8433187	11.263162709	10.883239665	.
9	AIR	SEP1949	136	137.45174613	127.1963396	4.4145298018	5.8408767297	.
10	AIR	OCT1949	119	119.51466679	128.57222537	-3.712022009	-5.345536567	.
11	AIR	NOV1949	104	107.03417354	129.85660918	-11.29875007	-11.52368557	.
12	AIR	DEC1949	118	108.67419512	131.04978498	-16.21934073	-6.156249129	.
13	AIR	JAN1950	115	120.94095306	132.13206287	-16.92781378	5.7367039728	.
14	AIR	FEB1950	126	132.44156173	133.40313595	-13.24153953	12.279965308	.
15	AIR	MAR1950	141	134.39193046	134.78645215	-6.268071825	5.8735501288	.
16	AIR	APR1950	135	131.1623557	136.36388938	2.3325213673	-7.534055042	.
17	AIR	MAY1950	125	133.94606621	138.09377166	10.217872946	-14.3655784	.
18	AIR	JUN1950	149	148.90077318	140.12159096	15.512387846	-6.733205623	.
19	AIR	JUL1950	170	166.25791554	142.25166558	16.187264172	7.8189857801	.
20	AIR	AUG1950	170	170.83665644	144.42782619	11.731949689	14.676880566	.
21	AIR	SEP1950	158	156.81924614	146.65047846	3.3327983729	6.8359693047	.
22	AIR	OCT1950	133	134.32230039	148.90016965	-6.568487052	-8.009382209	.

Obs	GROUP5	GROUP6	GROUP7	GROUP8	GROUP9	GROUP10
1	.	.	.	.	.	.
2	.	.	.	.	.	.
3	.	.	.	.	.	.
4	.	.	.	.	.	.
5	.	.	.	.	.	.
6	.	.	.	.	.	.
7	.	.	.	.	.	.
8	.	.	.	.	.	.
9	.	.	.	.	.	.
10	.	.	.	.	.	.
11	.	.	.	.	.	.
12	.	.	.	.	.	.
13	.	.	.	.	.	.
14	.	.	.	.	.	.
15	.	.	.	.	.	.
16	.	.	.	.	.	.
17	.	.	.	.	.	.
18	.	.	.	.	.	.
19	.	.	.	.	.	.
20	.	.	.	.	.	.
21	.	.	.	.	.	.
22	.	.	.	.	.	.

Output 5.2.3 SSA Results of AIR Data



Example 5.3: Run a Singular Spectrum Analysis of a User-Selected Group

This example runs a singular spectrum analysis of a user-selected group. Output 5.3.2 shows the resulting groupings together with the original data. Output 5.3.1 shows the values of each components.

```
proc tsmodel data=mycas.air outobj=(os=mycas.analytic (replace=YES));
   id date interval=month;
   var air;
   require ssa;
   submit;
      array group1[3]/nosymbols;group1[1]=1;group1[2]=2;group1[3]=3;
      array group2[2]/nosymbols;group2[1]=4;group2[2]=5;
      array group3[2]/nosymbols;group3[1]=6;group3[2]=7;
      declare object s(ssa);
      declare object os(outssa);
      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;
```

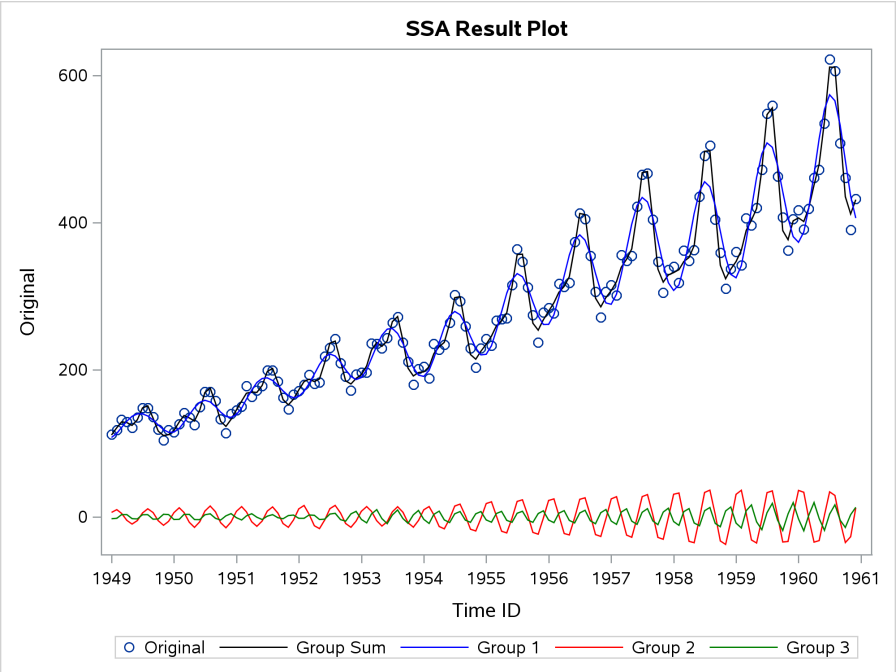
Output 5.3.1 SSA Results of AIR Data

SSA Result Table

Obs	_NAME_	DATE	ORIGINAL	GROUPSUM	GROUP1	GROUP2	GROUP3	GROUP4
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	140.0182405	5.1250250962	2.8489336817	.
8	AIR	AUG1949	148	150.88822784	137.10648141	10.883239665	2.8985067677	.
9	AIR	SEP1949	136	134.15922433	131.6108694	5.8408767297	-3.292521804	.
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	137.7085761	128.51838033	5.8735501288	3.3166456463	.
16	AIR	APR1950	135	134.46042538	138.69641074	-7.534055042	3.2980696802	.
17	AIR	MAY1950	125	130.2462494	148.31164461	-14.3655784	-3.699816806	.
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	129.910353	142.3316826	-8.009382209	-4.411947394	.

Obs	GROUP5	GROUP6	GROUP7	GROUP8	GROUP9	GROUP10
1	.	.	.	.	.	.
2	.	.	.	.	.	.
3	.	.	.	.	.	.
4	.	.	.	.	.	.
5	.	.	.	.	.	.
6	.	.	.	.	.	.
7	.	.	.	.	.	.
8	.	.	.	.	.	.
9	.	.	.	.	.	.
10	.	.	.	.	.	.
11	.	.	.	.	.	.
12	.	.	.	.	.	.
13	.	.	.	.	.	.
14	.	.	.	.	.	.
15	.	.	.	.	.	.
16	.	.	.	.	.	.
17	.	.	.	.	.	.
18	.	.	.	.	.	.
19	.	.	.	.	.	.
20	.	.	.	.	.	.
21	.	.	.	.	.	.
22	.	.	.	.	.	.

Output 5.3.2 SSA Results of AIR Data



References

Golyandina, N., Nekrutkin, V., and Zhigljavsky, A. (2001). *Analysis of Time Series Structure SSA and Related Techniques*. Boca Raton, FL: CRC Press.

Leonard, M., and Elsheimer, B. (2017). "Automatic Singular Spectrum Analysis and Forecasting." In *Proceedings of the SAS Global Forum 2017 Conference*. Cary, NC: SAS Institute Inc. https://support.sas.com/resources/papers/proceedings17/SAS0586-2017.pdf.

Leonard, M., John, M., Elsheimer, B., and Kessler, M. (2010). "An Introduction to Singular Spectrum Analysis with SAS/ETS Software." Paper presented at the SAS Global Forum 2010 Conference.

Chapter 6
Time Filters Package

Contents

Overview: Time Filters Package

The time filters package (TIMFIL) provides objects that enable you to perform various 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.

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: Language Reference*. For general information about CAS and CAS sessions, see *SAS Cloud Analytic Services: Fundamentals*.

Time Filters Package Summary

Table 6.1 summarizes the objects in the time filters package.

Table 6.1 Objects in the Time Filters Package

Object	Description
MRR	Computes the range and moving relative range for each observation in a time series
OUTMRR	Collects output from an MRR object

Return Codes

Table 6.2 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 6.2 Return Codes

Status	Meaning
< 0	An unrecoverable error occurred. No result was produced.
= 0	Unconditional success. The requested action completed and a normal result was produced.
> 0	Conditional success or warning. A result was produced subject to conditions.

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 tth element of the time series. The MRR object computes the range and moving relative range for X_t as

$$\mathrm{Range}_t = \mathrm{Range}(X_t, X_{t-1}, \ldots, X_{t-M+1})$$
$$\mathrm{MRR}_t = \frac{\mathrm{Range}_t}{\mathrm{Median}(\mathrm{Range}_t, \mathrm{Range}_{t-1}, \ldots, \mathrm{Range}_{t-K+1})}$$

where M is the window length for computing the range and K are the window lengths for computing the moving relative range.

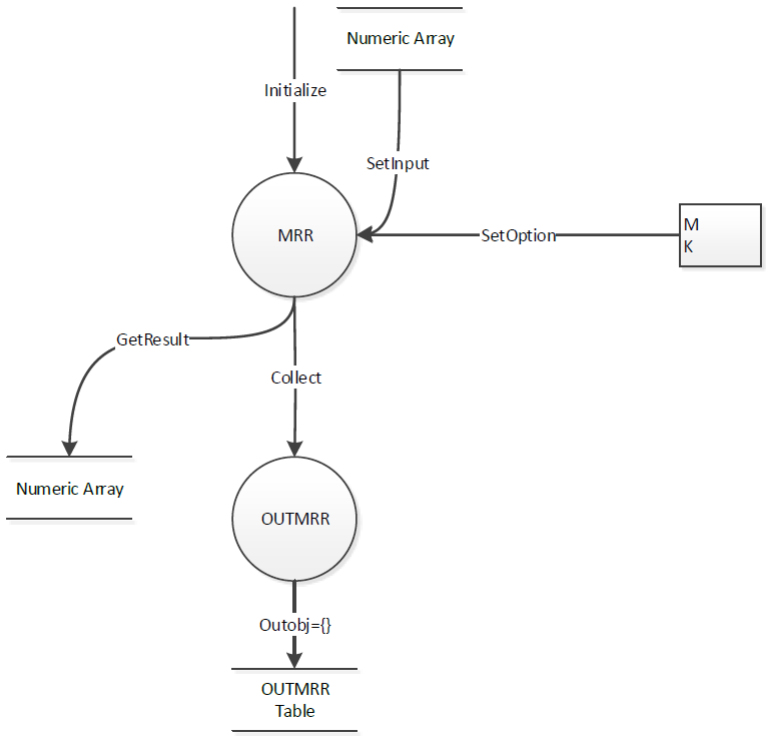
Table 6.3 summarizes the methods that are associated with the MRR object.

Table 6.3 Methods of the MRR Object

Object	Description
Getresult	Get the MRR result
Initialize	Initialize an MRR instance
Setinput	Set the input time series
Setoption	Set options for MRR analysis
Run	Run the MRR analysis

Figure 6.1 diagrams the methods of the MRR object.

Figure 6.1 MRR Data Flow



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=*obj*.**GetResult** **(***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=*obj*.**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.Setinput Method

rc=*obj*.**Setinput (***Value***) ;**

Adds a time series array (*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=*obj*.**Setoption ('***Name***', *Value*) ;**

Sets named options for the MRR instance.

Input Arguments

You must specify the following arguments as '*Name*', *Value* pairs:

'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.

'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.

MRR.Run Method

rc=*obj*.**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.

OUTMRR Object

The OUTMRR object collects output from the MRR object.

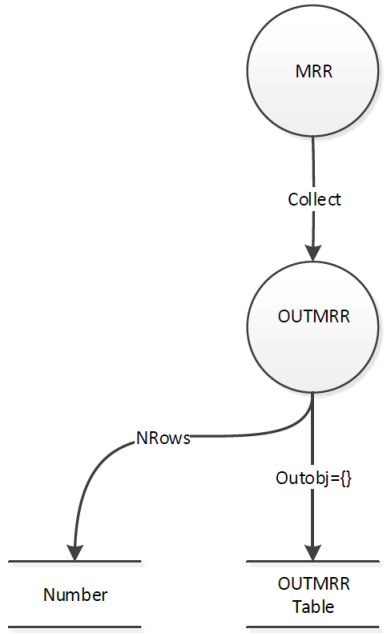
Table 6.4 summarizes the methods that are associated with the OUTMRR object.

Table 6.4 Methods of the OUTMRR Object

Method	Description
Collect	Collect the singular values of an MRR object
Nrows	Get the current row count from the OUTMRR object

Figure 6.2 diagrams the methods of the OUTMRR object.

Figure 6.2 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 6.5.

Table 6.5 CAS Table Collected with OUTMRR

Column	Type	Description
outrange	Numeric	The range value
outmrr	Numeric	The moving relative range value

Arguments

You must specify the following input argument:

obj specifies 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.

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:

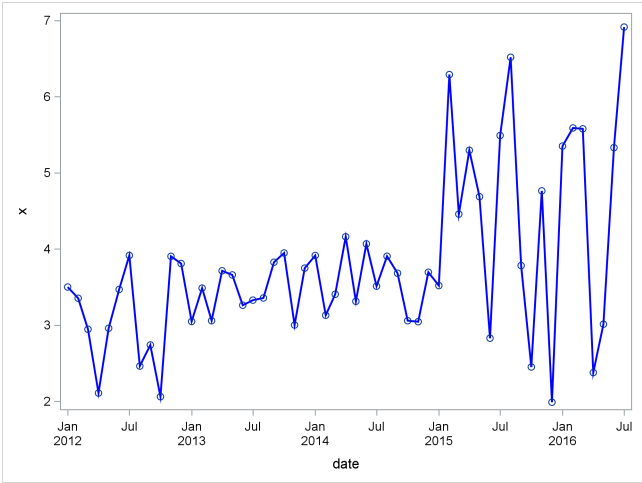
```
data input;
   input date : monyy7. x;
   format date monyy7.;
   datalines;
Jan-12 3.50137
Feb-12 3.35424
Mar-12 2.94675
Apr-12 2.10943

   ... more lines ...
```

The following statements plot the input time series.

```
proc sgplot data=input;
   series x=date y=x/lineattrs=(color=blue thickness=2) markers;
run;
```

Output 6.3 shows the results.

Figure 6.3 Plot of Input Data



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.

```
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:

```
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 sorts the mycas.outmrr data table and merges it with the mycas.input data table to create results.

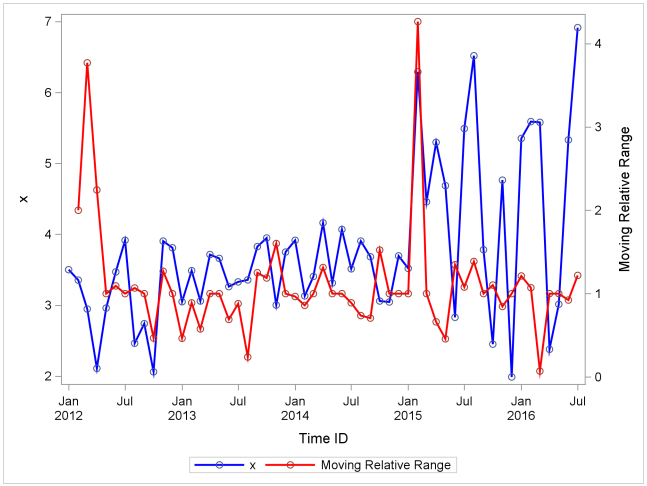
```
data outmrr;
   set mycas.outmrr;
run;

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 6.4 shows the results.

Figure 6.4 Plot of Input Data with MRR



Chapter 7
Time-Frequency Analysis Package

Contents

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 7.1 summarizes the classes in the TFA package.

Table 7.1 Classes in the Time-Frequency Analysis Package

TFA Class	Description
FFT	Computes the discrete Fourier transform of a real time series
FFTC	Computes the discrete Fourier transform of a complex time series
HILBERT	Computes the analytic signal that corresponds to a real time series
PWV	Computes the pseudo-Wigner-Ville distribution of a real time series
STFT	Computes the short-time Fourier transform of a real time series
WINDOW	Creates a window of a requested type and length

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 146.

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 $y = \begin{pmatrix} y[0] & y[1] & \dots & y[n-1] \end{pmatrix}$ and $s = \pm 1$, the output is $z = \begin{pmatrix} z[0] & z[1] & \dots & z[n-1] \end{pmatrix}$, where

$$z[t] = \sum_{k=0}^{n-1} y[k] \exp \frac{s 2\pi \mathbf{i}\, k t}{n},\ 0 \le t \le n-1$$

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 *sign* is 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 obj(OUTFFT) defines an object that is used to collect output from FFT instance.

OUTFFT.Collect Method

Usage: *rc* **=** *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 7.2.

Table 7.2 CAS Table Collected with OUTFFT

Column	Type	Description
X	Numeric	The real part of the discrete Fourier transform of input time series
Y	Numeric	The imaginary part of the discrete Fourier transform of input time series

OUTFFT.Nrows Attribute

Usage: *nrows* **=** *obj*.**nrows** *()* **;**

This attribute gets the current row count from the OUTFFT instance.

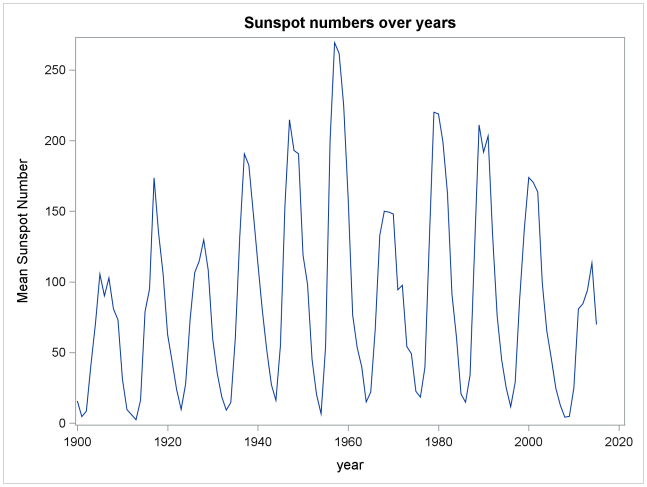
Example

The following statements read and plot the yearly sunspot count data since 1900. These statements assume that your CAS engine libref is named mycas, but you can substitute any appropriately defined CAS engine libref.

```
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
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;

proc sgplot data=mycas.sunspot;
   yaxis label="Mean Sunspot Number";
   title "Sunspot numbers over years";
   series x=Year y=x;
run;
```

Output 7.1 shows the results.

Figure 7.1 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;
```

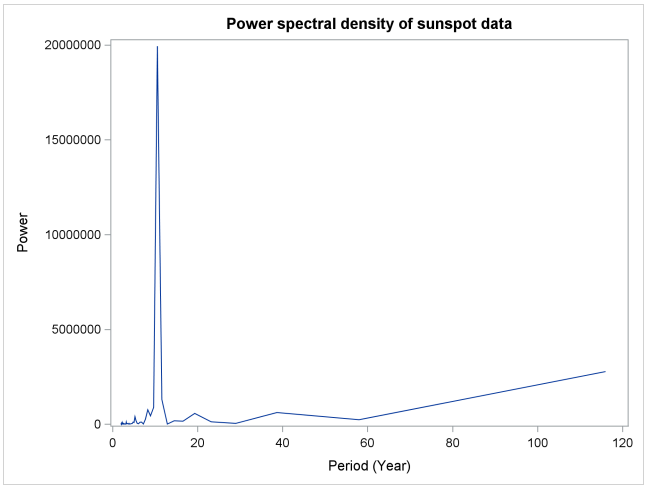
```
   declare object f(FFT);
   declare object of(OUTFFT);
   rc = f.Run("Y", x); if rc then stop;
   rc = of.Collect(f); if rc then stop;
   endsubmit;
run;
```

The following statements plot the power spectral density. You can see a spike around 10 years.

```
data mycas.power;
   keep Freq Period Power;
   set mycas.fft_x nobs=n;
   Power = x ** 2 + y ** 2;
   Freq = (_n_-1)/n; /* Unit Hz */
   if Freq > 0;
   Period = 1/Freq;
   output;
   if Freq > 0.5 then stop;
run;

proc sgplot data=mycas.power;
   title "Power spectral density of sunspot data";
   xaxis label="Period (Year)";
   series x=Period y=Power;
run;
```

Output 7.2 shows the results.

Figure 7.2 Power Spectral Density of Sunspot Data



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) ;**

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 $y = \begin{pmatrix} y[0] & y[1] & \ldots & y[n-1] \end{pmatrix}$ and $s = \pm 1$, the output is $z = \begin{pmatrix} z[0] & z[1] & \ldots & z[n-1] \end{pmatrix}$, where

$$z[t] = \sum_{k=0}^{n-1} y[k] \exp \frac{s 2\pi \mathbf{i}\, k t}{n},\ 0 \le t \le n-1$$

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* = *obj*.**Run** *('Name','Value['Name','Value,...])* **;**

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* = *obj*.**Collect** *('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 7.3.

Table 7.3 CAS Table Collected with OUTFFTC

Column	Type	Description
X	Numeric	The real part of the discrete Fourier transform of input time series
Y	Numeric	The imaginary part of the discrete Fourier transform of input time series

OUTFFTC.Nrows Attribute

Usage: *nrows* = *obj*.**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 x_re and x_im of the mycas.x table and then runs the class again on two input arrays that are named y_re and y_im:

```
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 */
   rc = f.Run("y_re", x_re, "y_im", x_im, "sign", 'forward'); if rc then stop;
   rc = of1.Collect(f); if rc then stop;
   /* Call FFTC on user defined input arrays y_re, y_im */
   array y_re[4]/nosymbols;
   y_re[1]=1; y_re[2]=2; y_re[3]=3; y_re[4]=4;
   array y_im[4]/nosymbols;
   y_im[1]=-1; y_im[2]=-2; y_im[3]=-3; y_im[4]=-4;
   rc = f.Run("y_re", y_re, "y_im", y_im, "sign", 'forward'); if rc then stop;
   rc = of2.Collect(f); if rc then stop;
   endsubmit;
run;
```

HILBERT Class

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: *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*.*Collect ('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 7.4.

Table 7.4 CAS Table Collected with OUTHILBERT

Column	Type	Description
X	Numeric	The real part of the discrete Fourier transform of input time series
Y	Numeric	The imaginary part of the discrete Fourier transform of input time series

OUTHILBERT.Nrows Attribute

Usage: *nrows* **=** *obj*.**nrows** *()* **;**

This attribute gets the current row count from the OUTHILBERT instance.

Example

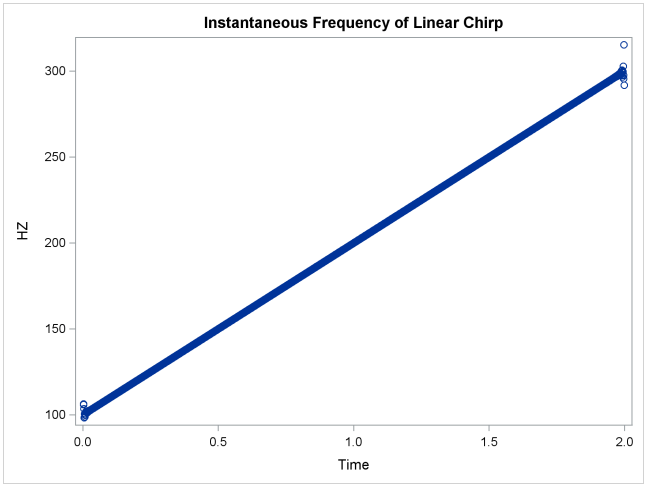
This example uses a time series that is obtained by sampling from a sinusoidal of the form $x(t) = A\cos(2\pi\theta(t) + \phi)$, where $\dot{\theta}(t) = \frac{d\theta(t)}{dt}$ is a linear function of time and is known as a linear chirp. For such a sinusoidal, it is reasonable to interpret $\dot{\theta}(t)$ 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:

```
/* 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*(f0*time + 0.5*(f1-f0)*time*time));
      i = i + 1;
      output;
   end;
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 7.3 shows the results.

Figure 7.3 Instantaneous Frequency of Linear Chirp



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 = 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 156) 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.

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 167. The default value of *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: *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 7.5.

Table 7.5 CAS Table Collected with OUTPWV

Column	Type	Description
Frequency	Numeric	The normalized frequency index
Time	Numeric	The time index
PWV	Numeric	The value of the pseudo-Wigner-Ville distribution

Note: The number of rows in the output table is *fftlen* × k, where

$$k = \begin{cases} \left\lfloor \dfrac{\text{series\_length} - 1}{\text{window\_length} - \textit{overlap}} \right\rfloor, & \text{when fade} = 1 \\ \left\lfloor \dfrac{\text{series\_length} - \textit{overlap}}{\text{window\_length} - \textit{overlap}} \right\rfloor + 1, & \text{when fade} = 0 \end{cases}$$

The frequency index starts from 0 and increments in multiples of $\dfrac{1}{2 \times \textit{fftlen}}$. The time index starts from 0 and increments in multiples of window_length – *overlap*.

OUTPWV.Nrows Attribute

Usage: *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:

```
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;
```

STFT Class

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 169.

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: *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 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 159).

- 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 7.6.

Table 7.6 CAS Table Collected with OUTSTFT

Column	Type	Description
Frequency	Numeric	The normalized frequency index
Time	Numeric	The time index
Power	Numeric	The power of the time series that corresponds to the current frequency and time index
Amplitude	Numeric	The amplitude of the time series that corresponds to the current frequency and time index
Phase	Numeric	The phase of the time series that corresponds to the current frequency and time index
Coef_re	Numeric	The real part of the Fourier coefficient
Coef_im	Numeric	The imaginary part of the Fourier coefficient

Note: The number of rows in the output table is *fftlen* $\times k$, where $k = \left\lfloor \frac{\text{series\_length} - \textit{overlap}}{\text{window\_length} - \textit{overlap}} \right\rfloor + 1$. The frequency index starts from 0 and increases in increments of $\frac{1}{\textit{fftlen}}$. The time index starts from 0 and increments in multiples of window_length – *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 $\sqrt{power}$, where *power* is the value stored in the third column. The phase is computed as $\mathrm{atan2}(y, x)$ 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:

```
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;
```

WINDOW Class

DECLARE OBJECT *w* (**WINDOW**) **;**

DECLARE OBJECT *ow* (**OUTWINDOW**) **;**

rc **=** *w***.Run** (*'Name',Value['Name',Value,...])* **;**

rc **=** *ow***.Collect** (*w*) **;**

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* **=** *obj***.Run** (*'Name',Value['Name',Value,...])* **;**

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

$$w[i] = 1 - \left| \frac{2i}{N-1} - 1 \right|, \; 0 \le i \le N-1$$

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

$$w[i] = 0.62 - 0.48 \left| \frac{i}{N-1} - 0.5 \right| - 0.38 \cos \frac{2\pi i}{N-1}, \; 0 \le i \le N-1$$

BLACKMAN specifies a Blackman window. For this window type, you do not need to specify any *params*. This window is defined as

$$w[i] = 0.42 - 0.5\cos\frac{2\pi i}{N-1} + 0.08\cos\frac{4\pi i}{N-1},\ 0 \le i \le N-1$$

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

$$\begin{aligned} w[i] =& 0.35875 - 0.48829\cos\frac{2\pi i}{N-1} + 0.14128\cos\frac{4\pi i}{N-1} \\ & - 0.01168\cos\frac{6\pi i}{N-1},\ 0 \le i \le N-1 \end{aligned}$$

BOHMAN specifies a Bohman window. For this window type, you do not need to specify any *params*. This window function is defined as

$$\begin{aligned} w[0] =& w[N-1] = 0 \\ w[i] =& \left(1 - \left|1 - \frac{2i}{N-1}\right|\right)\cos\left(\pi\left|1 - \frac{2i}{N-1}\right|\right) \\ & + \frac{1}{\pi}\sin\left(\pi\left|1 - \frac{2i}{N-1}\right|\right),\ \text{for } 1 \le i \le N-2 \end{aligned}$$

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 nth-degree Chebyshev polynomial, T_n, which is the unique polynomial such that $T_n(\cos\theta) = \cos n\theta$ for all values of θ. $T_n(x)$ can be computed as

$$T_n(x) = \begin{cases} \cos(n\ \cos^{-1} x), & |x| \le 1 \\ \cosh(n\ \operatorname{acosh}(x)), & x > 1 \\ (-1)^n T_n(-x), & x < -1 \end{cases}$$

For odd N (say, $N = 2M + 1$) with $M > 0$, the Chebyshev window of length N can be defined as

$$w[i] = c\left(1 + \frac{2}{T_{2M}(\beta)}\sum_{k=1}^{M} T_{2M}(\beta\cos\frac{k\pi}{N})\cos\frac{2\pi k(i-M)}{N}\right),\ 0 \le i \le N-1$$

where $\beta = \cosh(\operatorname{acosh}(10^{att/20})/(N-1))$ 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

$$w[i] = c \times \left\{\sum_{k=0}^{N-1}(-1)^k T_{N-1}(\beta\cos\frac{k\pi}{N})\cos\frac{\pi k(2i+1)}{N}\right\},\ 0 \le i \le N-1$$

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

$$w[i] = 0.21557895 - 0.41663158\cos\frac{2\pi i}{N-1} + 0.277263158\cos\frac{4\pi i}{N-1} - 0.083578947\cos\frac{6\pi i}{N-1} + 0.006947368\cos\frac{8\pi i}{N-1},\ 0 \le i \le N-1$$

GAUSSIAN specifies a Gaussian window. This window function needs one *param*: c, whose default value is 2.5. This window function is defined as

$$w[i] = \exp\left(-\frac{c^2}{2}\left(\frac{i - \frac{N-1}{2}}{\frac{N-1}{2}}\right)^2\right),\ 0 \le i \le N-1$$

HAMMING specifies a Hamming window. For this window type, you do not need to specify any *params*. This window function is defined as

$$w[i] = 0.54 - 0.46\cos\frac{2\pi i}{N-1},\ 0 \le i \le N-1$$

HANNING specifies a Hanning window. For this window type, you do not need to specify any *params*. This window function is defined as

$$w[i] = \frac{1}{2}\left(1 - \cos\frac{2\pi i}{N-1}\right),\ 0 \le i \le N-1$$

KAISER specifies a Kaiser window. This window function needs one *param*: β, whose default value is 0.5. This window function is defined as

$$w[i] = \frac{I_0\left(\beta\sqrt{1-(1-\frac{2i}{N-1})^2}\right)}{I_0(\beta)},\ 0 \le i \le N-1$$

where $I_0(\cdot)$ is the modified Bessel function of the first kind of order 0, which is defined as

$$I_0(x) = \sum_{m=0}^{\infty}\frac{\left(\frac{x}{2}\right)^{2m}}{m!^2}$$

PARZEN specifies a Parzen window. For this window type, you do not need to specify any *params*. This window function is defined as

$$w[i] = \begin{cases} 2\left(1 - \frac{|2i-(N-1)|}{N}\right)^3, & 0 \le i < \frac{N-1}{4} \\ 1 - 6\left(\frac{|2i-(N-1)|}{N}\right)^2 + 6\left(\frac{|2i-(N-1)|}{N}\right)^3, & \frac{N-1}{4} \le i \le \frac{N-1}{2} \\ W[N-i-1], & \frac{N-1}{2} < i \le N-1 \end{cases}$$

The last half of the window is defined by symmetry, which implies $w[i] = w[N-i-1]$ for $0 \le i \le N-1$.

RECTANGULAR specifies a rectangular window. For this window type, you do not need to specify any *params*. This window function is defined as

$$w[i] = 1,\ 0 \le i \le N-1$$

TUKEY specifies a Tukey window. This window function needs one *param*: α, whose default value is 0.5. Let $\epsilon = 10^{-12}$. If $\alpha \ge 1$, then a Hanning window is returned; if $\alpha \le \epsilon$, a rectangular window is returned. For $\epsilon < \alpha < 1$, this window function is defined as

$$w[i] = \begin{cases} \frac{1}{2}(1+\cos\left(\frac{\pi}{\alpha}(\frac{2i}{N-1}-\alpha)\right)), & 0 \le i < \alpha(N-1)/2 \\ 1, & \alpha(N-1)/2 \le i \le (N-1)/2 \\ w[N-i-1], & (N-1)/2 < i <= N-1 \end{cases}$$

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 154 and "STFT Class" on page 158.

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* = *obj*.**Collect** *('WINDOWObj')* **;**

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 7.7.

Table 7.7 CAS Table Collected with OUTWINDOW

Column	Type	Description
NAME	String	The type of the window
X	Numeric	The array of numbers that contain the window

OUTWINDOW.Nrows Attribute

Usage: *nrows* **=** *obj*.**nrows** *()* **;**

This attribute gets the current row count from the OUTWINDOW instance.

Example

The following DATA step creates a dummy data table:

```
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:

```
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 $y = \begin{pmatrix} y_0 & y_1 & \dots & y_{n-1} \end{pmatrix}^T$ of a time-series $x = \begin{pmatrix} x_0 & x_1 & \dots & x_{n-1} \end{pmatrix}^T$ can be obtained from the following matrix multiplication:

$$\begin{pmatrix} y_0 \\ y_1 \\ \dots \\ y_{n-1} \end{pmatrix} = \begin{pmatrix} 1 & 1 & 1 & \dots & 1 \\ 1 & \omega & \omega^2 & \dots & \omega^{n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega^{n-1} & \omega^{2(n-1)} & \dots & \omega^{(n-1)(n-1)} \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ \dots \\ x_{n-1} \end{pmatrix}$$

That is, $y = Wx$, where $\omega = \exp -\frac{2\pi \mathbf{i}}{n}$ 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 $z(t) = x(t) + \mathbf{i}\hat{x}(t)$, 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 $X(f) = \int x(t) \exp(-2\pi \mathbf{i} f t)\, dt$, the Hilbert transform is defined by inverting $X(f)$ over the positive frequencies (Cohen 1995, pg 30):

$$\hat{x}(t) = 2 \int_0^\infty X(f) \exp(2\pi i f t)\, df$$

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 $\mathrm{E} = \int_{-\infty}^{\infty} |x(t)|^2\, dt = \int_0^{2\pi} |X(f)|^2\, df$ 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 Δt and small Δf, 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 τ varies. So the Wigner-Ville distribution of a continuous, possibly complex-valued, time series $x(t)$ is given by $W_x(t, f)$:

$$\begin{aligned} W_x(t, f) &= \int_{-\infty}^{\infty} x(t + \tau/2)\overline{x(t - \tau/2)} \exp(-2\pi \mathbf{i} f \tau)\, d\tau \\ &= 2\int_{-\infty}^{\infty} x(t + \tau)\overline{x(t - \tau)} \exp(-4\pi \mathbf{i} f \tau)\, d\tau. \end{aligned}$$

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 $x(t)$ is given by

$$W_x(t, f) = \int w(\tau)x(t + \tau/2)\overline{x(t - \tau/2)} \exp(-2\pi \mathbf{i} f \tau)\, d\tau$$

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):

$$W_f(n, f) = 2 \sum_{k=-\infty}^{\infty} x[n + k]\overline{x[n - k]} \exp(-4\pi \mathbf{i} f k)$$

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 *window*=$(w[0], \ldots, w[2m])$, define

$$U_x(n, f) = \sum_{k=-m}^{m} w[m + k]x[n + k]\overline{x[n - k]} \exp(-2\pi \mathbf{i} k f)$$

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 series_length – 1 is replaced with 0.

Define the following:

- $S = \text{window\_length} - \textit{overlap}$
- Let

$$k = \begin{cases} \left\lfloor \dfrac{\text{series\_length} - 1}{S} \right\rfloor, \text{when } \textit{fade} = 1 \\ \left\lfloor \dfrac{\text{series\_length} - \textit{overlap}}{S} \right\rfloor, \text{when } \textit{fade} = 0 \end{cases}$$

- Let

$$c = \begin{cases} 0, \text{when } \textit{fade} = 0 \\ L, \text{when } \textit{fade} = 1 \end{cases}$$

  where the 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 $f = 0, \dfrac{1}{\textit{fftlen}}, \ldots, \dfrac{\textit{fftlen} - 1}{\textit{fftlen}}$.

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

$$S_x(n, f) = \sum_{k=0}^{m-1} x[n+k]w[k]\exp(-\mathrm{i}2\pi k f)$$

Let $k = \left\lfloor \frac{\text{series\_length} - \textit{overlap}}{\text{window\_length} - \textit{overlap}} \right\rfloor + 1$ and $S = \text{window\_length} - \textit{overlap}$. Then STFT consists of the computation of $S_x(n, f)$ for $n = 0, S, 2S, \ldots, kS$ and for $f = 0, \frac{1}{\textit{fftlen}}, \ldots, \frac{\textit{fftlen} - 1}{\textit{fftlen}}$.

References

Boashash, B. (1988). "Note on the Use of the Wigner Distribution for Time-Frequency Signal Analysis." *IEEE Transactions on Acoustics, Speech, and Signal Processing* 36:1518–1521.

Boashash, B. (1992a). "Estimating and Interpreting the Instantaneous Frequency of a Signal, Part 1: Fundamentals." *Proceedings of the IEEE* 80:520–538.

Boashash, B. (1992b). "Estimating and Interpreting the Instantaneous Frequency of a Signal, Part II: Algorithms and Applications." *Proceedings of the IEEE* 80:540–568.

Claasen, T. A. C. M., and Mecklenbräuker, W. F. G. (1980a). "The Wigner Distribution—a Tool for Time-Frequency Signal Analysis, Part I: Continuous Time Signals." *Philips Journal of Research* 35:217–250.

Claasen, T. A. C. M., and Mecklenbräuker, W. F. G. (1980b). "The Wigner Distribution—a Tool for Time-Frequency Signal Analysis, Part I: Discrete-Time Signals." *Philips Journal of Research* 35:276–300.

Cohen, L. (1995). *Time-Frequency Analysis*. Englewood Cliffs, NJ: Prentice Hall.

Debnath, L. (2002). "Recent Developments in the Wigner-Ville Distribution and Time-Frequency Signal Analysis." *Proceedings of the Indian National Science Academy, Part A* 68:35–56.

Harris, F. J. (1978). "On the Use of Windows for Harmonic Analysis with the Discrete Fourier Transform." *Proceedings of the IEEE* 66:51–83.

Marple, L., Jr. (1999). "Computing the Discrete-Time 'Analytic' Signal via FFT." *IEEE Transactions on Signal Processing* 47:2600–2603.

Van Loan, C. (1992). *Computational Frameworks for the Fast Fourier Transform.* Vol. 10 of Frontiers in Applied Mathematics. Philadelphia: SIAM.

Chapter 8
Time Series Analysis Package

Contents

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: Each function in this chapter has a prefix of "TSA."; however, the prefixes are omitted in descriptions for better readability. The mycas libref in the examples refers to Sasioca 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: Language Reference*.

Functional Summary

Table 8.1 summarizes the functions in the TSA package.

Table 8.1 Functions in Time Series Analysis Package

TSA Function	Description
ACCUMULATE	Accumulates a univariate time series to a particular frequency
ACCUMULATE2	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
ACF	Computes autocorrelation and autocovariance for a time series array
ARMAORDERS	Performs tests to tentatively identify the autoregressive and moving average orders of mixed autoregressive moving average models
CCF	Computes the cross-correlation and cross-covariance for two time series arrays
DPF Class Object	Performs count distribution analysis for time series
FREQ Class Object	Performs frequency analysis of a time series
INTERMITTENCYTEST	Tests for intermittency of a univariate time series
IACF	Computes the inverse autocorrelation for a time series array
MOVINGSUMMARY	Computes statistics for a set of values within a moving time window
PACF	Computes the partial autocorrelation for a time series array
SCALE	Scales a time series between the minimum value and the maximum value of the original time series
SEASONALDECOMP	Computes the seasonal indices of a univariate time series using classical decomposition
SEASONALINDICES	Computes the seasonal indices of a univariate time series by using regression seasonal dummies
SEASONTEST	Tests for seasonality of a univariate time series
SIMILARITY	Performs similarity analysis for time series
STATIONARITYTEST	Tests for stationarity of a univariate time series
TRANSFORM	Transforms time series according to the specified transformation type
UNBIASEDNESS	Tests whether a univariate time series is unbiased
WHITENOISE	Tests for white noise of a time series array

ACCUMULATE Function

rc = **TSA.ACCUMULATE** *(time, y, 'interval', id, z, <'accumulate'>, <'setmiss'>, <'zeromiss'>)* **;**

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:

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:

rc	**Termination Reason**
0	Success
< 0	Computational failure

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:

```
proc tsmodel 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', , );
   endsubmit;
run;
```

ACCUMULATE2 Function

rc = **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:

rc	Termination Reason
0	Success.
< 0	Computational failure.

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:

```
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 Function

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:

rc	Termination Reason
0	Success
< 0	Computational failure

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.

acflprob returns an array of 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:

```
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;
```

ARMAORDERS Function

rc **= TSA.ARMAORDERS** *(y, < dif >, < 'method' >, < p >, < q >, < perror >, porders, qorders)* **;**

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.

dif specifies either an array of positive integers or a positive integer that is used for differencing. The default value is 0.

'*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].

perror specifies the autoregressive orders used to estimate the error series for the MINIC method, where *perror* is an array of two nonnegative integers that define the minimum and maximum values.

By default, *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:

rc	**Termination Reason**
0	Success
< 0	Computational failure

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]/nosymbols; P[1]=0; P[2]=5;
   Array Q[2]/nosymbols; 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 Function

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 times 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:

rc	**Termination Reason**
0	Success
< 0	Computational failure

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:

```
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 ccfstd ccf2std ccfnorm ccfprob ccflprob;
   require tsa;
   submit;
   declare object TSA(tsa);
   rc=TSA.CCF(price, sale, 20, lags, df, ymu, xmu, ccov, ccf,
              ccfstd, ccf2std, ccfnorm, ccfprob, ccflprob);
   endsubmit;
run;
```

DPF Class 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.

Signature

```
declare object f(FREQ);
declare object of(OUTFREQ);
declare object dpf(DPF);
declare object odpe(OUTDPE);
declare object odprob(OUTDPROB);
rc = f.Initialize();
rc = f.SetY(claims);
rc = f.SetOption(<'name',value, 'name', value, ...>);
rc = f.Run();
rc = of.Collect(f);

rc = dpf.Initialize(f);
rc = dpf.SetOption(<'name',value, 'name', value, ...>);
rc = dpf.Run();
rc = odpe.Collect(dpf);
rc = odprob.Collect(dpf);
```

Required Input

You must specify a FREQ object through the Initialize() statement:

'*freq*' specifies a FREQ class object, which includes frequency analysis results.

Optional Specifications

You can optionally specify one or more of the following '*Name*', *Value* pairs, separated by commas, in the SetOption() statement:

'*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 *maxiter* is an integer.

'*select*' specifies the distribution selection criterion.

You can specify the following values within single quotation marks:

AIC specifies Akaike's information criterion.

BIC specifies the Bayesian information criterion.

LOGLIK specifies the log-likelihood.

The default value is LOGLIK.

'*method*' specifies candidate distribution to use in the analysis.

You can specify the following values within single quotation marks:

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.

POISSON specifies the Poisson distribution.

ZMBINOMIAL specifies the zero-modified binomial distribution.

ZMGEOMETRIC specifies the zero-modified geometric distribution.

ZMNEGBINOMIAL specifies the zero-modified negative binomial distribution.

ZMPOISSON specifies the zero-modified Poisson distribution.

The default distribution is BEST.

Returned Values

The Run() statement returns the following values:

rc returns one of the following scalar return codes:

rc	**Termination Reason**
0	Successful
< 0	Computational failure

Output Table Schema

Two collector objects, OUTDPE and OUTDPROB, collect the results of the count distribution analysis. Results can be output to tables using OUTOBJ=() statement.

OUTDPE collects parameter estimates information of selected count distribution. The output table contains the following columns:

Column Name	**Column Content**
NAME	Name of target variable
Distribution	Distribution
Parameter	Name of parameter
Estimate	Parameter estimate
StdErr	Standard error
Tvalue	*t* value
Probt	Approximate probability > |*t*|
Lower	Lower 95% of parameter estimate
Upper	Upper 95% of parameter estimate

OUTDPROB collects predictions using the count distribution that has the best fit. The output table contains the following columns:

Column Name	Column Content
NAME	Name of target variable
Index	Index
Value	Values
Observedzeros	Observed zeros
Expectedzeros	Expected zeros
Observed	Observed counts
Expected	Expected counts
Expectedlower	Expected lower confidence
Expectedupper	Expected upper confidence
Probability	Probability
Probabilitylower	Probability lower confidence
Probabilityupper	Probability upper confidence
Chisquare	Chi-square statistic
Chisquareprob	Chi-square probability
Chisquarelogprob	Chi-square log probability

Example

This example uses the TSMODEL procedure to analyze the frequency of the time series Air and then conducts a count distribution analysis:

```
proc tsmodel data=mycas.air outobj=(of=mycas.outfreq(replace=YES)
                                    odpe=mycas.dpe(replace=YES)
                                    odprob=mycas.dprob(replace=YES)
                                    );
   var air;
   id date interval=month;
   require tsa;
   submit;
    declare object f(FREQ);
    declare object of(OUTFREQ);
    declare object dpf(DPF);
    declare object odpe(OUTDPE);
    declare object odprob(OUTDPROB);
    rc = f.Initialize();
    rc = f.SetOption("SEASONALITY",12);
    rc = f.SetY(air);
    rc = f.Run();
    rc = of.Collect(f);

    rc = dpf.Initialize(f);
    rc = dpf.SetOption("SELECT","AIC","METHOD","POISSON");
    rc = dpf.Run();
    rc = odpe.Collect(dpf);
    rc = odprob.Collect(dpf);
   endsubmit;
run;
```

FREQ Class 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.

Signature

```
declare object f(FREQ);
declare object of(OUTFREQ);
rc = f.Initialize();
rc = f.SetY(y);
rc = f.SetOption('name',value);
rc = f.Run();
rc = f.GetResult('name',series);
rc = of.Collect(f);
```

Required Input

You must specify the following input series through the SetY() statement:

y specifies the times series array to analyze.

Optional Specification

You can optionally specify the following '*Name*', *Value* pair in the SetOption() statement:

'SEASONALITY' specifies the seasonality of the time series, where *Value* is an integer. The default *Value* is 0.

You can optionally retrieve the results series produced by the Run() statement and store them in specified numeric arrays by adding one or more GetResult() statements. All results series have the same length. You can specify one of the following '*Name*', *Series* pairs:

'VALUES' specifies a numeric array, *Series*, in which to store the results series that contains the unique values found in the input series.

'COUNTS' specifies a numeric array, *Series*, in which to store the results series that contains the number of appearances of each unique value in the input series.

'PCTS' specifies a numeric array, *Series*, in which to store the results series that contains the percentage of times that a unique value appears in the input series.

Returned Values

The Run() statement returns the following values:

rc returns one of the following scalar return codes:

rc	**Termination Reason**
0	Successful
1	Binary series
2	Nonnegative integer series
3	Integer series
4	Noninteger series
< 0	Computational failure

Output Table Schema

The collector object, OUTFREQ, collects the results of frequency function. Results can be output to a table using the OUTOBJ=() statement.

OUTFREQ collects the results of frequency function. The output table contains the following columns:

Column Name	**Column Content**
NAME	Name of target variable
Values	Array of series values
Counts	Array of frequency series counts
Percent	Percentage of total frequency

Example

This example uses the TSMODEL procedure to analyze the frequency of the time series Air:

```
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;
```

IACF Function

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:

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 *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.

iacflprob 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:

```
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, , , , );
   endsubmit;
run;
```

INTERMITTENCYTEST Function

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:

rc	**Termination Reason**
0	Time series is not intermittent
1	Time series is intermittent
< 0	Computational failure

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:

```
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 Function

rc = **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:

EWMA calculates the exponentially weighted moving average.

GMEAN calculates the moving geometric mean.

MAX calculates the maximum value.

MEAN calculates the moving average.

MED calculates the median value.

MIN calculates the minimum value.

PROD calculates the moving product.

RANGE calculates the maximum value minus minimum value.

SUM calculates the moving sum.

TVALUE calculates the standard deviation divided by mean.

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.

'*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.

'*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:

rc	**Termination Reason**
4	Warping limits relaxed
3	Expansion limits relaxed
2	Compression limits relaxed
1	Warping limits imposed
0	Success
< 0	Computational failure

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:

```
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:

```
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 Function

rc = **TSA.PACF** *(y, nlag, lags, df, < mu >, < pacf >, < pacfstd >, < pacf2std >, < pacfnorm >, < pacfprob >, < pacflprob >)* **;**

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.
pacflprob	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:

```
proc tsmodel 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 Function

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:

rc	**Termination Reason**
5	The input series is nearly constant
4	Missing values were found when the value 1 was specified for *nomiss*
3	One or more arguments are ignored
2	One or more arguments are set to the default value
1	The input series is all missing
–1	One or more arguments are not supported
–2	The minimum value of the transformed series is greater than its maximum value
–4	Extreme slope
–99	Bad arguments

x returns the transformed series.

Optional Returned Values

You can also specify the following argument to request an additional returned value:

nmiss returns the number of missing values that are generated.

Example

This example uses the TSMODEL procedure to scale the time series array Air between a minimum value of 0 and a maximum value of 100:

```
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 Function

rc = **TSA.SEASONALDECOMP** *(y, s, 'mode', <lambda>, <tcc>, <sic>, <sc>, <scstd>, <tcs>, <ic>, <sa>, <pcsa>, <tc>, <cc>)* **;**

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.

'*mode*' specifies the type of decomposition to be used to decompose the time series.

You can specify the following values within single quotation marks:

ADD | ADDITIVE specifies additive decomposition.

LOGADD | LOGADDITIVE specifies log-additive decomposition.

MULT | MULTIPLICATIVE specifies multiplicative decomposition.

MULTORADD specifies multiplicative or additive decomposition, depending on data.

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:

rc	**Termination Reason**
0	Success
< 0	Computational failure

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:

```
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 Function

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:

rc	**Termination Reason**
0	Success
< 0	Computational failure

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 Function

rc = **TSA.SEASONTEST** *(y, s, < dif >, < p >, < alpha >, < 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:

rc	**Termination Reason**
0	Time series is not seasonal
1	Time series is seasonal
< 0	Computational failure

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:

```
proc tsmodel 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 Function

rc = **TSA.SIMILARITY** *(x, y, <'type'>, <'scale'>, <expandpct>, <expandabs>, <compresspct>, <compressabs>, 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:

rc	**Termination Reason**
4	Warping limits relaxed
3	Expansion limits relaxed
2	Compression limits relaxed
1	Warping limits imposed
0	Success
< 0	Computational failure

measure returns the similarity measure.

Example

This example uses the TSMODEL procedure to compute the similarity of two time series arrays: x and y.

```
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 Function

rc = **TSA.STATIONARITYTEST** *(y, < dif >, < d >, < p >, < 'type' >, pvalue)* **;**

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.

dif specifies an array of positive integers or a positive integer that is used for differencing. The default value is 0.

d specifies the order of unit root ($d = 1, \ldots, 12$). If the *type* is SSM, then $d = 1$. The default value is 1.

p specifies the autoregressive order, where *p* must be a nonnegative integer. The default value is 5.

'*type*' specifies the type of test statistic used.

You can specify the following values within single quotation marks:

SSM specifies the studentized test statistic for the single mean (intercept) case.

STR specifies the studentized test statistic for the deterministic time trend case.

SZM specifies the studentized test statistic for the zero mean (no intercept) case. This value is allowed only when *d*=1.

The default value of *type* is SZM.

Returned Values

The STATIONARITYTEST function returns the following values:

rc returns one of the following scalar return codes:

rc	**Termination Reason**
0	Time series is stationary with the default significance level of 0.05
1	Time series is not stationary with the default significance level of 0.05
< 0	Computational failure

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:

```
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 Function

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: $x = \mathrm{sqrt}(y + c)$. 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 λ: $x = c^2 + (y^c - 1)/c$. 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:

rc	**Termination Reason**
0	Success
< 0	Computational failure

x returns the transformed series.

Example

This example uses the TSMODEL procedure to take the log transform of the time series array Air:

```
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 Function

rc = **TSA.UNBIASEDNESS** *(y, predict, < siglevel >, 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.

siglevel specifies the significance level.

Returned Values

The UNBIASEDNESS function returns the following values:

rc returns one of the following scalar return codes:

rc	**Termination Reason**
1	Biased predictions
0	Unbiased predictions
–1	Degree of freedom error
–2	Singular system
–3	Extreme value

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:

```
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 Function

rc **= 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:

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 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:

```
proc tsmodel 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;
```

Chapter 9
Time Series Model Package

Contents

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 9.1 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 9.1 Objects in the Time Series Model Package

Object	Description
Central Object (Note 1)	
TSM	Time series model object
Time Series Model Specification Objects (Note 2)	
ARIMASPEC	Autoregressive integrated moving average model specification object
ESMSPEC	Exponential smoothing model specification object
EXMSPEC	External model specification object
IDMSPEC	Intermittent demand model specification object
UCMSPEC	Unobserved component model (UCM) specification object

Table 9.1 *continued*

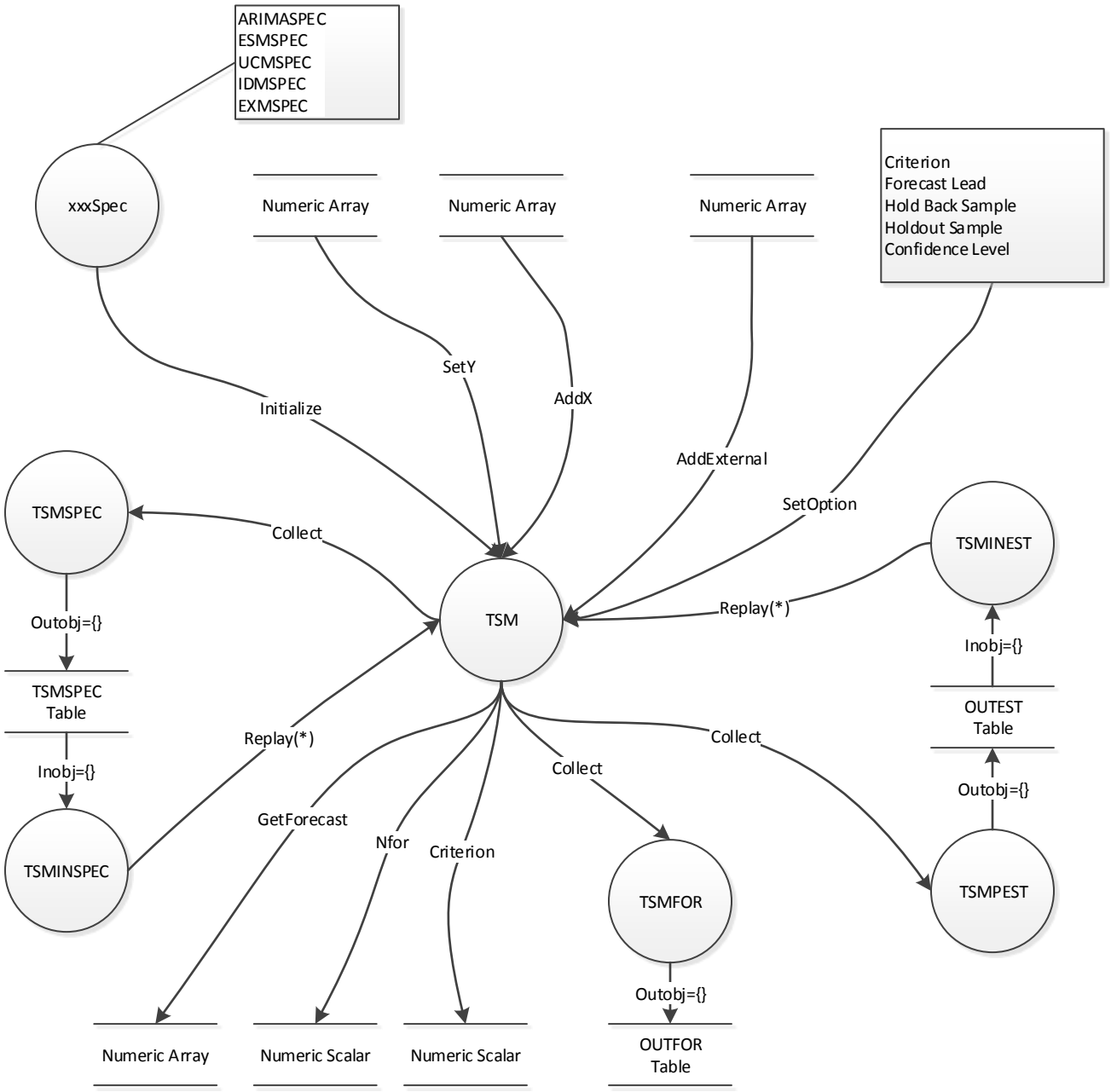
Object	Description
Time Series Model Collector Objects (Note 3)	
TSMFOR	Time series model forecast collector object
TSMPEST	Time series model parameter estimates collector object
TSMSPEC	Time series model specification collector object
Time Series Model Repeater Objects (Note 4)	
TSMINEST	Time series 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 time series model specification objects enable you to specify the characteristics of a time series model to be executed by a TSM object.
3. 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).
4. 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 9.1 diagrams the relationships among the objects in the TSM package.

Figure 9.1 TSM Object Data flow



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.

Step1: 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 Analysis and Forecasting 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.
2. **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 9.2 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 9.2 Return Codes

Status	Meaning
< 0	An unrecoverable error occurred. No result was produced.
= 0	Unconditional success. The requested action was completed, and a normal result was produced.
> 0	Conditional success or warning. A result was produced subject to conditions.

TSM Object

The TSM object generates forecasts of univariate time series.

Table 9.3 lists the time series model families that are supported.

Table 9.3 Model Families for the TSM Object

Family	Object	Description
ARIMA	ARIMASPEC	ARIMAX models
ESM	ESMSPEC	Exponential smoothing models
EXM	EXMSPEC	External model (external forecast)
IDM	IDMSPEC	Intermittent demand models (Croston's/average demand (ADEM))
UCM	UCMSPEC	Unobserved component models

Table 9.4 summarizes the methods that are associated with the TSM object.

Table 9.4 Methods of the TSM Object

Method	Description
AddX	Add an independent time series array (*XSeries*) for the TSM object
AddExternal	Add external forecast component series for the TSM object
criterion	Return the final forecast fit statistic
GetForecast	Get the forecast series
Initialize	Initialize the TSM object
nfor	Return the forecast series length
Replay	Replay the restored model and parameter estimates
Run	Run the TSM object
SetOption	Specify the named option for the TSM object
SetY	Specify the dependent time series array (*YSeries*) for the TSM object

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=*obj*.**Initialize** (< *ModelSpec* >) ;
rc=*obj*.**SetY** (*YSeries*) ;
rc=*obj*.**AddX** (*XSeries* < ,*Required,NoDiff,ModelSymbol* >) ;
rc=*obj*.**AddExternal** (*Series* < ,*Role* >) ;
rc=*obj*.**SetOption** ('*Name*', *Value* < ,'*Name*',*Value,...* >) ;
rc=*obj*.**Replay** (*TSMINSPECObj* < ,*TSMINESTObj* >) ;
rc=*obj*.**Run** () ;
rc=*obj*.**GetForecast** (*Which, Result*) ;
nfor=**obj.nfor** () ;
criterion=**obj.criterion** () ;

TSM Methods

TSM.AddExternal Method

rc=*obj*.**AddExternal** (*Series* < ,*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 236.

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

criterion=**obj.criterion () ;**

Returns the fit statistic value for the final forecast for the TSM object. The criterion is set via the 'CRITERION' argument in the TSM.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.

TSM.GetForecast Method

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=*obj*.**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=**obj.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=*obj*.**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=*obj*.**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

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

Specifies the named options for the TSM 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, 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 is specified with the TSM.SetY method) that ends at the last nonmissing observation. The default value is 0.

'LEAD' takes a nonnegative integer *Value* that specifies the forecast lead. The default value is 12.

TSM.SetY Method

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.

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 9.5 summarizes the methods that are associated with the ESMSPEC Object.

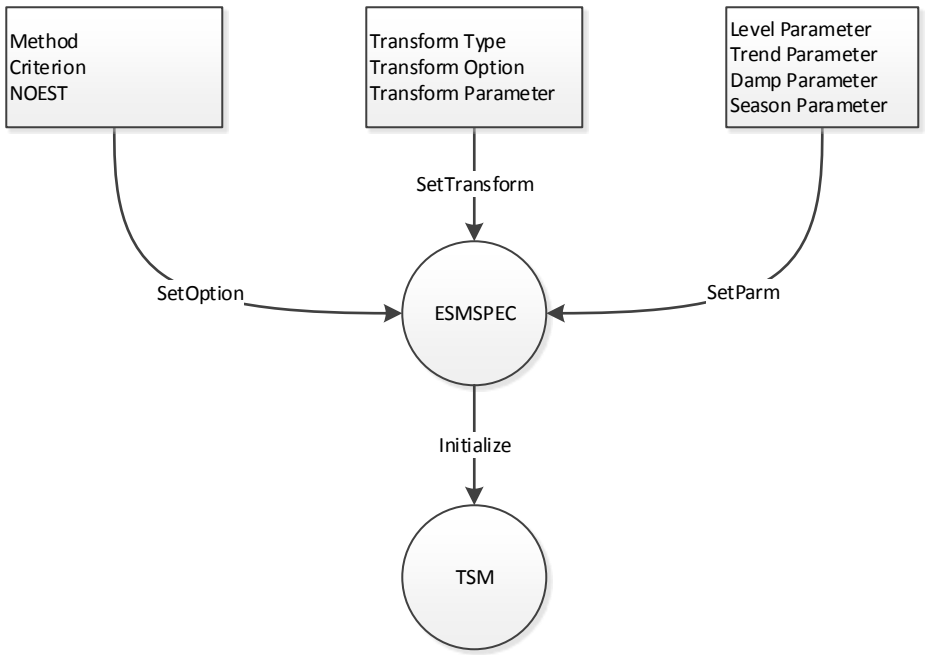
Table 9.5 Methods of the ESMSPEC Object

Method	Description
Close	Close ESM model specification
Open	Open ESM model specification
SetOption	Set option for ESM model
SetParm	Set parameters for ESM model
SetTransform	Set transform for ESM model

You can also store the XML representation of the ESMSPEC object in a CAS table. For more information, see the TSMSPEC object.

Figure 9.2 illustrates the data flow through the ESMSPEC object.

Figure 9.2 ESMSPEC Object Data Flow



ESMSPEC Synopsis

DECLARE OBJECT *obj* **(ESMSPEC) ;**

Method syntax, in order of typical usage:

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=*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=*obj*.**Open () ;**

Opens the ESMSPEC object.

Arguments

There are no arguments associated with this method.

ESMSPEC.SetOption Method

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

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 HPFDIAGNOSE 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.
- **MULTSEASONAL** 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 9.6 summarizes the methods that are associated with the TSM.ARIMASPEC method.

Table 9.6 Methods of the ARIMASPEC Object

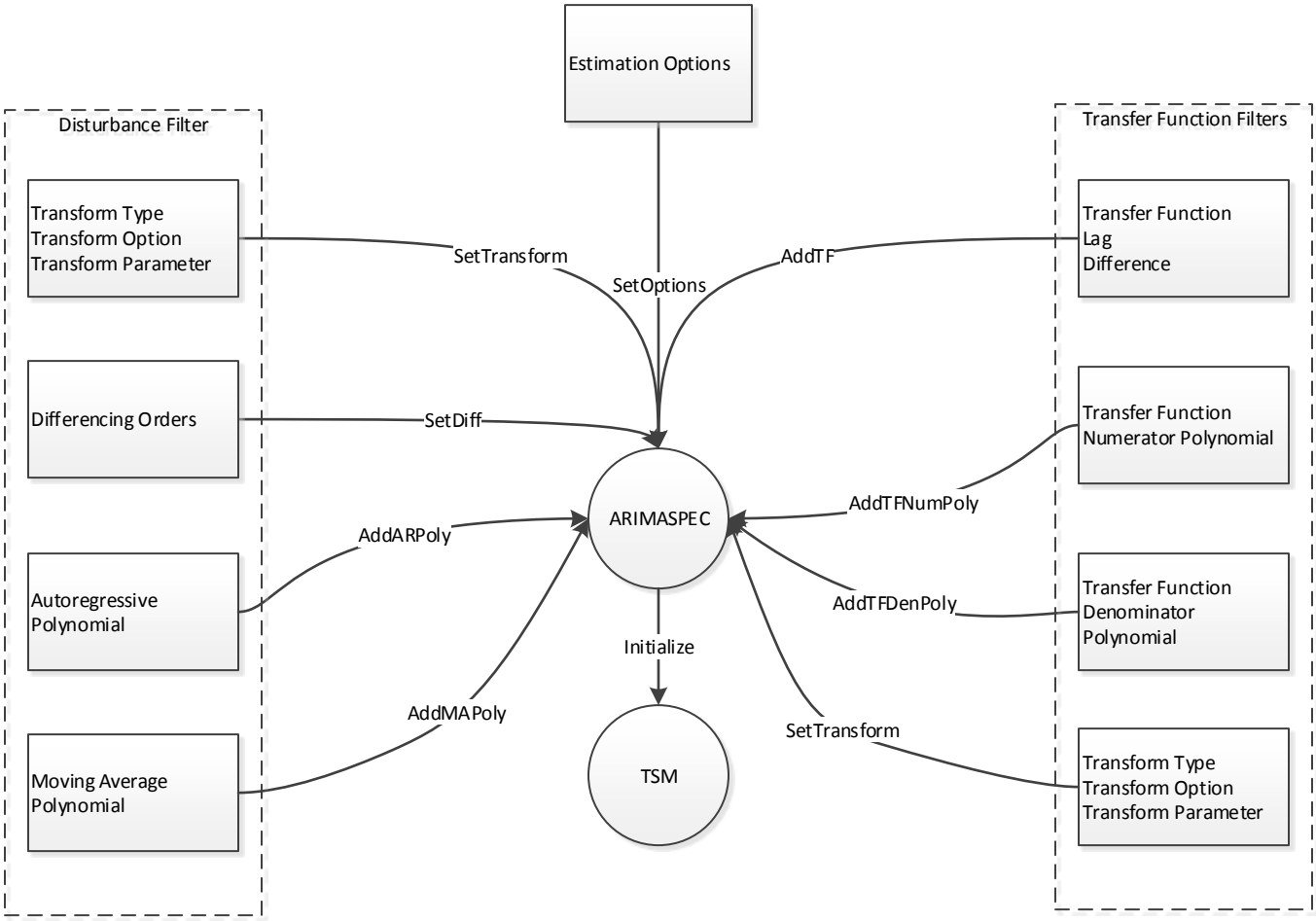
Method	Description
AddARPoly	Add an autoregressive polynomial factor to the ARIMA model
AddMAPoly	Add a moving average polynomial factor to the ARIMA model
AddTF	Add a transfer function to the ARIMA model
AddTFDenPoly	Add a transfer function denominator factors to the ARIMA model
AddTFNumPoly	Add a transfer function numerator factors to the ARIMA model
Close	Close the ARIMA model specification
Open	Open the ARIMA model specification
SetDiff	Add differencing to the ARIMA model
SetOption	Specify the ARIMA option
SetTFTransform	Specify the transfer function transform
SetTransform	Specify transform

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 9.3 illustrates the data flow through the ARIMASPEC object.

Figure 9.3 ARIMASPEC Object Data flow



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=*obj*.**AddMAPoly (***OrderArray* < ,*NOrder*,*Seasonal*,*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 the 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 *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 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 *OrderArray*.

ARIMASPEC.AddTF Method

rc=*obj*.**AddTF (***XName*, < *Delay*,*DifArray*,*NDiff* >**) ;**

Adds a transfer function to the ARIMA model for the specified *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.

DiffArray 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 *DiffArray* 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 *DiffArray* values to use. If not specified, then all the values in *DiffArray* are used.

ARIMASPEC.AddTFDenPoly Method

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

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=*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=*obj*.**Open () ;**

Initializes an empty ARIMASPEC object for configuration.

Arguments

There are no arguments associated with this method.

ARIMASPEC.SetDiff Method

rc=*obj*.**SetDiff (***DiffArray* < ,*NDiff* >**) ;**

Adds differencing to the ARIMA model.

Input Arguments

You must specify the following input argument:

DiffArray is a numeric array that specifies differencing orders, where each order must be an integer greater than or equal to 1. Values in *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:

NDiff takes a numeric value that specifies the number of *DiffArray* values to use. The default is to use all elements of *DiffArray*.

ARIMASPEC.SetOption Method

rc=*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.

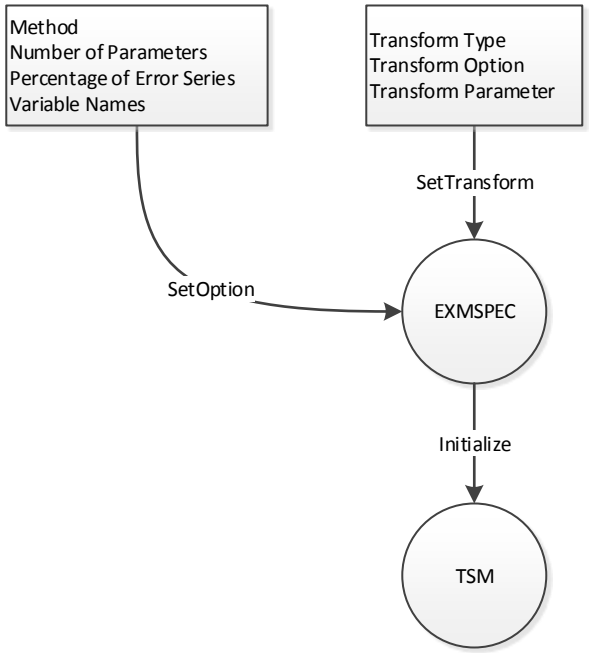
Table 9.7 summarizes the methods that are associated with the EXMSPEC object.

Table 9.7 Methods of the EXMSPEC Object

Method	Description
Open	Open EXM model specification
Close	Close EXM model specification
SetTransform	Set transform for EXM model specification
SetOption	Set options for EXM model specification

Figure 9.4 illustrates the data flow through the EXMSPEC object.

Figure 9.4 EXMSPEC Object Data Flow



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 ESMSPEC object, see the section "TSM Object" on page 217 or Chapter 3, "Automatic Time Series Analysis and Forecasting 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 253.
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 219.
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 Analysis and Forecasting Package."

EXMSPEC Synopsis

DECLARE OBJECT *obj* **(EXMSPEC) ;**

Method syntax, in order of typical usage:

rc=*obj*.**Open () ;**
rc=*obj*.**SetTransform (**'*Type*' < ,*Option*,*Parm*>**) ;**
rc=*obj*.**SetOption (**'*Name*', *Value* < ,'*Name*',*Value*,... >**) ;**
rc=*obj*.**Close () ;**

EXMSPEC Methods

EXMSPEC.Close Method

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

rc=*obj*.**Open () ;**

Initializes an EXMSPEC object for configuration.

Arguments

There are no arguments associated with this method.

EXMSPEC.SetOption Method

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.
'NLAGPCT'	takes a numeric *Value* that specifies the percentage of error series count for ACF computations.
'NPARMS'	takes a numeric *Value* that specifies the number of parameters for external forecast.
'PREDICT'	takes a string *Value* that specifies the variable name for PREDICT series.
'SIGMA'	takes a string *Value* that specifies the prediction error standard deviation.
'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:

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.

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 221 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 253.

Table 9.8 the methods that are associated with the IDMSPEC object.

Table 9.8 Methods of the IDMSPEC Object

Method	Description
Close	Close the IDM model specification
Open	Open the IDM model specification
SetMethod	Set component smoothing method
SetOption	Set options for IDM model specification
SetParm	Set component parameters
SetTransform	Set component transform

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=*obj*.**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 disabled (see the value NONE for the '*Type*' argument in the SetTransform method).

Arguments

There are no arguments associated with this method.

IDMSPEC.SetMethod Method

rc=*obj*.**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.

IDMSPEC.SetOption Method

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

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 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.

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 221. You can also store the XML representation of the UCM in a CAS table. For more information, see the section "TSMSPEC Object" on page 253. Note that the UCMSPEC object is equivalent to the HPFUCMSPEC procedure in *SAS Forecast Server Procedures: User's Guide*.

Table 9.9 lists the methods that are associated with the UCMSPEC object.

Table 9.9 Methods of the UCMSPEC Object

Method	Description
AddAutoreg	Add an autoregressive component to a UCM model
AddBlockSeason	Add a block-season component to a UCM model
AddComponent	Add a fundamental component to a UCM model
AddCycle	Add a cycle component to a UCM model
AddDeplag	Add a deplag component to a UCM model
AddInput	Add an input component to a UCM model
AddSeason	Add a seasonal component to a UCM model
Close	Close the UCM model specification
Open	Open the UCM model specification
SetTransform	Specify the transform

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,CoeffArray,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

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

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:

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 (***< Period, NoestPeriod,Rho,NoestRho,Variance, 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 *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 *NoestPeriod* is 1, a valid nonmissing *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 *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 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.AddDeplag Method

rc=*obj*.**AddDeplag (***LagArray, < NLag,CoeffArray,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 *LagArray* values to use. The default is all *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

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. If not specified, no transform is used.

TransParm takes a numeric value that specifies a transform parameter. This argument is used only for Box-Cox transformations.

UCMSPEC.AddSeason Method

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:

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.

'*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.

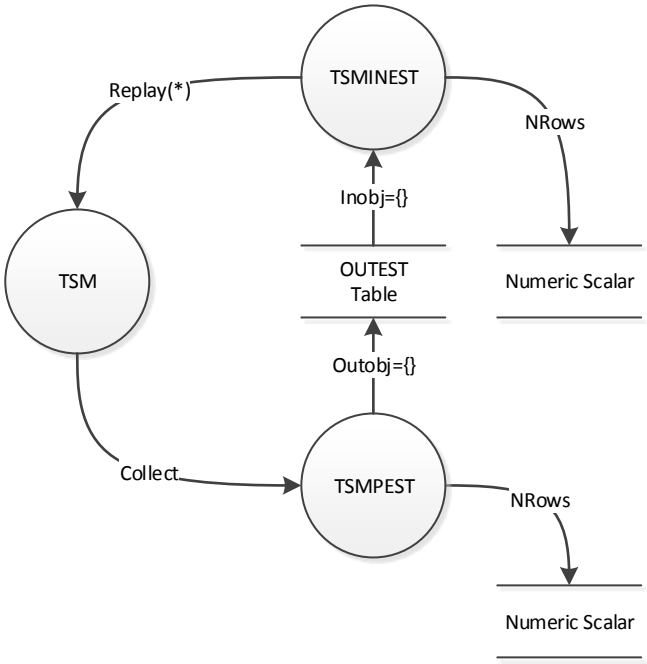
Table 9.10 summarizes the methods that are associated with the TSMPEST Object.

Table 9.10 Methods of the TSMPEST Object

Method	Description
Collect	Collect parameter estimates from TSM objects
nrows	Return the number of rows that are collected

Figure 9.5 illustrates the data flow through the TSMPEST object.

Figure 9.5 TSMPEST Object Data flow



TSMPEST Synopsis

DECLARE OBJECT *obj* **(TSMPEST) ;**

Method syntax, in order of typical usage:

rc=*obj*.**Collect (***TSMObj***) ;**
nrows=*obj*.**nrows () ;**

TSMPEST Methods

TSMPEST.Collect Method

rc=*obj*.**Collect (***TSMObj***) ;**

Collects time series model parameter estimates 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.

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.

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.

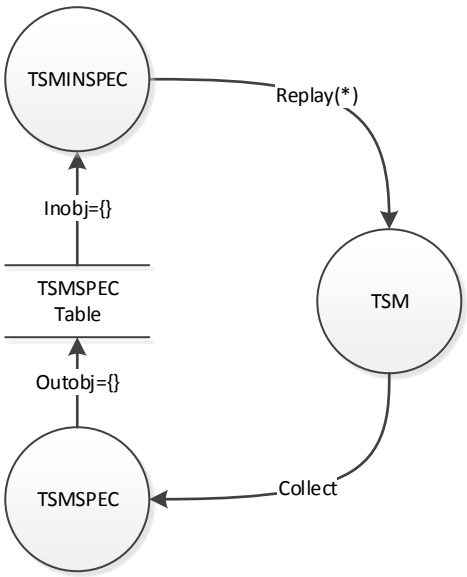
Table 9.11 summarizes the methods that are associated with the TSMSPEC Object.

Table 9.11 Methods of the TSMSPEC Object

Method	Description
Collect	Collect model specification from TSM objects
nrows	Return the number of rows that are collected

Figure 9.6 illustrates the data flow through the TSMSPEC object.

Figure 9.6 TSMSPEC Object Data flow



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 is used by HPFENGINE procedure for the data set that is specified in the TSMFOR= option.

Table 9.12 shows the contents of the TSMFOR object.

Table 9.12 Contents of the TSMFOR Object

Column	Type	Description
NAME	String	Name of the dependent variable
TIMEID	Numeric	Uniform time ID values for series
ACTUAL	Numeric	Accumulated values of dependent variable
ERROR	Numeric	Residuals
LOWER	Numeric	Lower confidence limit
PREDICT	Numeric	Forecasts of the dependent variable
STD	Numeric	Prediction standard error
UPPER	Numeric	Upper confidence limit

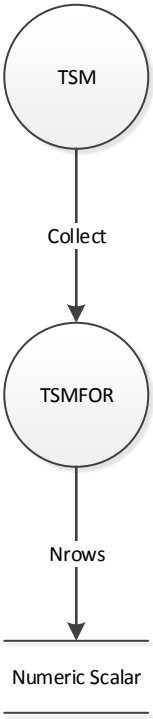
Table 9.13 summarizes the methods that are associated with the TSMFOR object.

Table 9.13 Methods of the TSMFOR Object

Method	Description
Collect	Collect forecasts estimates from the TSM object
nrows	Return the number of rows collected

Figure 9.7 illustrates the data flow through the TSMFOR object.

Figure 9.7 TSMFOR Object Data flow



TSMFOR Synopsis

DECLARE OBJECT *obj* **(***TSMFOR***) ;**

Method syntax:

rc=*obj*.**Collect (***ModelObj* < ,*Region* >**) ;**

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

TSMFOR Methods

TSMFOR.Collect Method

rc=*obj*.**Collect (***ModelObj* < ,*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.

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.

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

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.

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 9.3).

Table 9.14 summarizes the methods of the TSMINEST Object.

Table 9.14 Methods of the TSMINEST Object

Method	Description
nrows	Return number of rows in the TSMINEST object

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 9.15 summarizes the methods of the TSMINSPEC Object.

Table 9.15 Methods of the TSMINSPEC Object

Method	Description
nrows	Return number of rows in TSMINSPEC object

TSMINSPEC Synopsis

DECLARE OBJECT *obj* **(TSMINSPEC) ;**

Method syntax:

rc obj.**nrows () ;**

TSMINSPEC Methods

TSMINSPEC.nrows Method

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 4, "The TSMODEL Procedure" (*SAS Visual Forecasting: Forecasting Procedures*).

Example 9.1: 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) $\times$ $(0,1,1)_{12}$ 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;   **Store residuals of ESM and ARIMA models;
   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 9.1.1 shows the predictions, and Output 9.1.2 shows the parameter estimates for the airline model.

Output 9.1.1 Airline Model Predictions (Partial Output)

Airline Model Predictions

DATE	PREDICT	STD	UPPER	LOWER
JAN1961	450.4	16.9215	484.5	418.2
FEB1961	426.1	18.8590	464.2	390.3
MAR1961	480.1	24.0408	528.9	434.7
APR1961	492.8	27.2405	548.3	441.6
MAY1961	509.5	30.5863	572.0	452.1
JUN1961	584.2	37.6514	661.4	513.9
JUL1961	670.7	45.9957	765.3	585.1
AUG1961	668.2	48.4237	768.0	578.3
SEP1961	559.6	42.6271	647.7	480.7
OCT1961	498.3	39.7181	580.6	425.0
NOV1961	431.2	35.8240	505.5	365.2
DEC1961	478.9	41.3484	565.0	403.0

Output 9.1.2 Airline Model Parameter Estimates (Partial Output)

Airline Model Parameter Estimates

EST	_STDERR_	_TVALUE_	_PVALUE_
0.3773	0.0820	4.6033	9.828E-6
0.5724	0.0780	7.3361	2.17E-11

Output 9.1.3 shows the number of times the airline model residuals are smaller than the Winters model residuals.

Output 9.1.3 Number of Times the Airline Model Residuals Are Smaller Than the Winters Model Residuals

nbetter
71.0000

Example 9.2: 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 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 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 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:

```
proc tsmodel 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 9.2.1 shows the parameter estimates for the transfer function model.

Output 9.2.1 Parameter Estimates for the Transfer Function Model (Partial Output)

EST	_STDERR_	_TVALUE_	_PVALUE_
53.2630	0.1193	446.5	0
1.5329	0.0475	32.2472	6.25E-97
-0.6330	0.0501	-12.6434	2.27E-29
-0.5352	0.0748	-7.1534	7.21E-12
0.3760	0.1029	3.6553	0.000306
0.5189	0.1078	4.8124	2.425E-6
0.5484	0.0382	14.3499	1.72E-35

Example 9.3: 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 9.1 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.

```
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 9.3.1 shows the parameter estimates that are saved in mycas.airEst.

Output 9.3.1 Parameter Estimates for the Airline Model (Partial Output)

Airline Model Parameter Estimates

EST	_STDERR_	_TVALUE_	_PVALUE_
0.3773	0.0820	4.6033	9.828E-6
0.5724	0.0780	7.3361	2.17E-11

The following statements show how to forecast the airline series by using the previously saved model specification (mycas.airOSpec) and parameter estimates (mycas.airEst).

```
proc tsmodel 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);
      declare object airFor(tsmfor);

      *** Set up and run the TSM object ***;
      rc = airModel.Initialize();
      rc = airModel.SetY(Air);
      rc = airModel.SetOption('lead',12);
      rc = airModel.Replay(airSpec,airEst);
      rc = airModel.Run();

      *** Output the airline model forecasts ***;
      rc = airFor.Collect(airModel);
  endsubmit;
quit;
```

Output 9.3.2 shows the forecasts that are produced according to the fitted model.

Output 9.3.2 Replayed Forecasts (Partial Output)

Airline Model Predictions

DATE	PREDICT	STD	UPPER	LOWER
JAN1961	450.4	16.9215	484.5	418.2
FEB1961	426.1	18.8590	464.2	390.3
MAR1961	480.1	24.0408	528.9	434.7
APR1961	492.8	27.2405	548.3	441.6
MAY1961	509.5	30.5863	572.0	452.1
JUN1961	584.2	37.6514	661.4	513.9
JUL1961	670.7	45.9957	765.3	585.1
AUG1961	668.2	48.4237	768.0	578.3
SEP1961	559.6	42.6271	647.7	480.7
OCT1961	498.3	39.7181	580.6	425.0
NOV1961	431.2	35.8240	505.5	365.2
DEC1961	478.9	41.3484	565.0	403.0

References

Box, G. E. P., and Jenkins, G. M. (1976). *Time Series Analysis: Forecasting and Control*. Rev. ed. San Francisco: Holden-Day.

Chapter 10
Time Series Motif Discovery Package

Contents

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

Objects in the MTF package are divided into two categories: computational and collector. The objects are summarized in Table 10.1.

Table 10.1 Objects in Time Series Motif Discovery Package

Object	Type	Description
MTFBF	Computational	Brute-force method
MTFPM	Computational	Probabilistic method
MTFSCORE	Computational	Given a motif, search for its instances
MTFANOM	Computational	Subsequence anomaly detection
OUTMTF	Collector	Collect motif output
OUTMTFPM	Collector	Collect motif probabilistic output

Table 10.1 *continued*

Object	Type	Description
OUTMTFANOM	Collector	Collect anomaly detection output
OUTMTFSERIES	Collector	Collect motif series output
OUTMTFSCORE	Collector	Collect score output

Using the MTF Package

The following code provides an outline of how to use the motif package:

```
proc tsmodel 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:

- **Declare:** Create computational and collector objects by using the object declaration (DECLARE) statement.
- **Initialize:** Add a default model specification to the computational object.
- **Specify variables:** Specify time series variables by using SetX and SetY methods.
- **Specify option:** Specify model options and properties as appropriate by using the SetOption method.
- **Run:** Execute the model in the computational object to produce motifs and their instances.
- **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, *obj*, of type MTFBF. Upon declaration, the MTFBF object has a brute-force method model.

Table 10.2 summarizes the methods that are associated with the MTFBF object.

Table 10.2 Methods of the MTFBF Object

MTFBF Method	Description
Initialize	Initialize the MTFBF object
SetX	Specify a time series array for the MTFBF object
SetOption	Specify options for the MTFBF object
Run	Run the MTFBF object

MTFBF Synopsis

DECLARE OBJECT *obj* **(MTFBF) ;**

Method syntax, in order of typical usage:

rc=*obj*.**Initialize () ;**
rc=*obj*.**SetX (***XSeries* **) ;**
rc=*obj*.**SetOption (**'*Name*',*Value* < ,'*Name*',*Value* >**) ;**
rc=*obj*.**Run () ;**

MTFBF Methods

MTFBF.Initialize Method

rc **=** *obj*.**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

No input arguments are associated with this method.

MTFBF.Run Method

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

No input argument is associated with this method.

MTFBF.SetOption Method

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 that specifies the length of each motif instance. The *Value* should be greater than 2. The default value is 10.

'NMOTIF' takes a positive integer 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 value is YES.

'OVERLAP' takes a nonnegative numeric value that specifies the subsequence overlap allowance rate. The *Value* should be greater than or equal to 0 and less than 1. 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 $\times$ (1 – OVERLAP)).

MTFBF.SetX Method

rc = *obj*.**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 variable 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, *obj*, of type MTFPM. Upon declaration, the MTFPM object has a default probabilistic model.

Table 10.3 summarizes the methods that are associated with the MTFPM object.

Table 10.3 Methods of the MTFPM Object

MTFPM Method	Description
Initialize	Initialize the MTFPM object
SetX	Specify a time series array for the MTFPM object
SetOption	Specify options for the MTFPM object
Run	Run the MTFPM object

MTFPM Synopsis

DECLARE OBJECT *obj* **(MTFPM) ;**

Method syntax, in order of typical usage:

rc=*obj*.**Initialize () ;**
rc=*obj*.**SetX (***XSeries* **) ;**
rc=*obj*.**SetOption ('***Name***',***Value* < ,'*Name*',*Value* >**) ;**
rc=*obj*.**Run () ;**

MTFPM Methods

MTFPM.Initialize Method

rc **=** *obj*.**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

No input argument is associated with this method.

MTFPM.Run Method

rc=*obj*.**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

No input argument is associated with this method.

MTFPM.SetOption Method

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

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 that specifies the maximum number of iterations for the EM algorithm. The default value is 200 and should be less than $1,000,000$
'MOTIFLENGTH'	takes a positive integer that specifies the length of each motif instance. The *Value* should be greater than 2. The default value is 10.
'NBREAKPOINT'	takes a positive integer that specifies the number of quintile-based split points for symbolizing the time series (a Y-axis discretization). The *Value* should be between 1 and 20. The default value is 10.
'NMOTIF'	takes a positive integer that specifies the number of motif. The *Value* should not be greater than 20. The default value is 1.
'SEED'	takes a nonnegative integer that specifies a random number seed for the expectation and maximization (EM) algorithm, where $0 \leq Value \leq 9,999,999,999$. The default value is 0, which means a clock-generated seed.
'TOPK'	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 CUTOFFPROB argument, the first top instance of each motif is always output. The default value is 100.

MTFPM.SetX Method

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 variable 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 10.4 summarizes the methods that are associated with the MTFSCORE Object.

Table 10.4 Methods of the MTFSCORE Object

MTFSCORE Method	Description
Initialize	Initialize the MTFSCORE object
SetY	Specify a target time series array for the MTFSCORE object
SetX	Specify a time series array for the MTFSCORE object
SetOption	Specify options for the MTFSCORE object
Run	Run 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

No input argument is 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

No input argument is 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* that specifies the maximum distance between the specified target motif series and motif instances to be found. The *Value* should be greater than 0. The default value is 10, 000.

'MOTIFLENGTH' takes a positive integer that specifies the length of each motif instance. The *Value* should be greater than 2. 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 value is YES.

'OVERLAP' takes a nonnegative numeric *Value* that specifies the subsequence overlap allowance rate. The *Value* should be greater than or equal to 0 and less than 1. 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 × (1 – OVERLAP)).

'TOPK' takes a positive integer 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 variable 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 variable 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 10.5 summarizes the methods that are associated with the MTFANOM object.

Table 10.5 Methods of the MTFANOM Object

MTFANOM Method	Description
Initialize	Initialize the MTFANOM object
SetX	Specify the input time series array for the MTFANOM object
SetOption	Specify options for the MTFANOM object
Run	Run the MTFANOM object

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

No input arguments are associated with this method.

MTFANOM.Run Method

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.

Usage:

Input Arguments

No input arguments are associated with this method.

MTFANOM.SetOption Method

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* that specifies the length of anomaly subsequence. The *Value* should be greater than 2. 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 value is YES.

'OVERLAP' takes a numeric *Value* that specifies the subsequence overlap allowance rate. The *Value* should be between 0 and 1. The default value is 0, which means that no overlap is allowed. The actual offset value is calculated from: offset = round(LENGTH × (1 – 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

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 variable 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, *obj*, of OUTMTF.

Table 10.6 summarizes the methods that are associated with the OUTMTF object.

Table 10.6 Methods of the OUTMTF Object

OUTMTF Method	Description
Collect	Collects motifs and their instances from an MTFBF or MTFPM object

The CAS table schema that is used for storing the set of output variables is described in Table 10.7.

Table 10.7 Motif Output Variables

Column	Type	Description
Name	String	Name of the input time series
MotifID	Numeric	Motif identification number
StartPosition	Numeric	Starting time point of each motif instance of the motif
Distance	Numeric	Distance between each motif instance and the representative motif series

OUTMFT Synopsis

DECLARE OBJECT *obj* **(OUTMTF) ;**

Method syntax:

rc=*obj*.**Collect (***MTFObj* **) ;**

OUTMTF Methods

OUTMTF.Collect Method

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 10.8 summarizes the methods of the OUTMTFPM object:

Table 10.8 Methods of the OUTMTFPM Object

OUTMTFPM Method	Description
Collect	Collects motifs, their instance starting points, and their occurrence probabilities from an MTFPM object

The CAS table schema that is used for storing the set of output variables is described in Table 10.9.

Table 10.9 Probability Output Variables

Column	Type	Description
Name	String	Name of the input time series
MotifID	Numeric	Motif identification number
StartPosition	Numeric	Starting time point for each instance of the motif
ProbMotif	Numeric	Probability of the motif occurrence
ProbMotifStart	Numeric	Probability of each starting time point given the motif

OUTMFTPM 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 10.10 summarizes the methods of the OUTMTFANOM object.

Table 10.10 Methods of the OUTMTFANOM Object

OUTMTFANOM Method	Description
Collect	Collects anomaly subsequences and their starting points from an MTFANOM object

The CAS table schema that is used for storing the set of output variables is described in Table 10.11.

Table 10.11 Probability Output Variables

Column	Type	Description
Name	String	Name of the input time series
AnomalyRank	Numeric	Anomaly identification number
StartPosition	Numeric	Starting time point for each anomaly subsequence
Distance	Numeric	Distance instance used for the anomaly detection

OUTMFTANOM Synopsis

DECLARE OBJECT *obj* **(OUTMTFANOM) ;**

Method syntax:

rc=*obj*.**Collect (***MTFANOMObj* **) ;**

OUTMTFANOM Methods

OUTMTFANOM.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 (*MTFBFObj* or *MTFPMObj*) and stores them in a CAS table. The object declaration statement creates a new object, *obj*, of type OUTMTFSERIES.

Table 10.12 summarizes the methods of the OUTMTFSERIES object.

Table 10.12 Methods of the OUTMTFSERIES Object

OUTMTFSERIES Method	Description
Collect	Collects the representative motif series for each motif from an MTFBF or MTFPM object

The CAS table schema that is used for storing the set of output variables is described in Table 10.13.

Table 10.13 Output Variables

Column	Type	Description
Name	String	Name of the input time series
MotifID	Numeric	Motif identification
MotifSeries	Numeric	Motif representative series

OUTMTFSERIES Synopsis

DECLARE OBJECT *obj* **(***OUTMTFSERIES* **) ;**

Method syntax:

rc=*obj*.**Collect (***MTFObj* **) ;**

OUTMTFSERIES Methods

OUTMTFSERIES.Collect Method

rc = *obj*.**Collect (***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.

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, *obj*, of type OUTMTFSCORE.

Table 10.14 summarizes the methods that are associated with the OUTMTFSCORE object.

Table 10.14 Methods of the OUTMTFSCORE Object

OUTMTFSCORE Method	Description
Collect	Collects subsequences from MTFSCORE object

The CAS table schema that is used for storing the set of output variables is described in Table 10.15.

Table 10.15 Motif Score Output Variables

Column	Type	Description
Name	String	Name of the input time series
Rank	Numeric	Similarity rank of subsequences to the corresponding motif
StartPosition	Numeric	Starting time point for each found subsequence
Distance	Numeric	Distance between each found subsequence and the corresponding motif series

OUTMTFSCORE Synopsis

DECLARE OBJECT *obj* **(OUTMTFSCORE**) **) ;**

Method syntax:

rc=*obj*.**Collect (***MTFSCOREObj* **) ;**

OUTMTFSCORE Methods

OUTMTFSCORE.Collect Method

rc **=** *obj*.**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 $Y = \{y_t\}_{t=1,2,\ldots,T}$. If you set the motif length to m, you can get $(T - m + 1)$ subsequences from the time series. Define $S = \{S_i | i = 1, 2, \ldots, T - m + 1\}$ 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 $S_i = (y_{i1}, y_{i2}, \ldots, y_{im})$ and $S_j = (y_{j1}, y_{j2}, \ldots, y_{jm})$ 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:

$$\mathbf{D} = \{D_{ij}\}_{i,j=1,\ldots,(T-m+1)}$$

However, in practice, some potential trivial matches with a threshold of δ are ignored:

$$\mathbf{D} = \{D_{ij}\}_{i,j=1,\ldots,(T-m+1)}, \text{ and } j > i + \delta$$

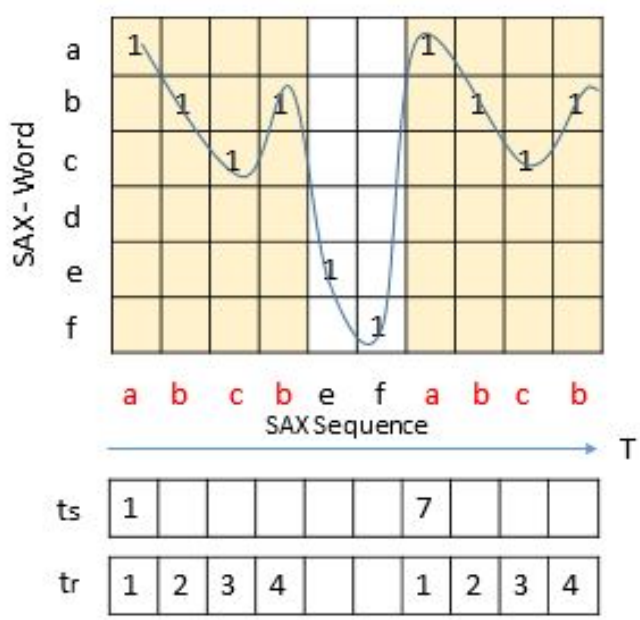
A commonly used δ value is the specified motif length m. The distances between subsequences $\{D_{ij}\}$ are sorted, and the smallest distance, $D_{i^*j^*}$, identifies a motif whose instances occur at i^* and j^* time points. So the subsequences S_{i^*} and S_{j^*} are the base instances of the motif.

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^* and j^*. For a particular i^*, you calculate $\{D_{i^*j}\}$ for $j = 1, \ldots, (T-m+1)$ and $j > i^* + \delta$; if $D_{i^*j} < (1+\tau) D_{i^*j^*}$, then S_j is an occurrence of the motif. Similarly, for a particular j^*, calculate $\{D_{ij^*}\}$ for $i = 1, \ldots, (T-m+1)$ and $i > j^* + \delta$; if $D_{ij^*} < (1+\tau) D_{i^*j^*}$, then S_i is an occurrence of the motif. When $\tau = 0$, the motif set includes only the selected pair of subsequences: S_{i^*} and S_{j^*}. When τ increases, the number of motif instances increases, but the accuracy decreases.

Motif Discovery, Probabilistic Model Method

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 10.1. 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 10.1 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.

Figure 10.1 A Probabilistic Model Framework Example



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.
- 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 Θ, 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

$$p(z, t_s|w, t_a) = \frac{p(z, t_s, w, t_a)}{p(w, t_a)}$$

$$\text{where } p(w, t_a) = \sum_{z=1}^{N_z} \sum_{t_s=1}^{T_s} p(z, t_s, w, t_a)$$

2. In the maximization step (M-step), the model parameters are updated according to the following posterior probability distributions:

$$p(z) \propto \sum_{t_s=1}^{T_s} \sum_{t_r=1}^{T_z-1} \sum_{w=1}^{N_w} n(w, t_s + t_r) p(z, t_s|w, t_s + t_r)$$

$$p(t_s|z) \propto \sum_{w=1}^{N_w} \sum_{t_r=1}^{T_z-1} n(w, t_s + t_r) p(z, t_s|w, t_s + t_r)$$

$$p(w|z) \propto \sum_{t_r=1}^{T_z-1} \sum_{t_s=1}^{T_s} n(w, t_s + t_r) p(z, t_s|w, t_s + t_r)$$

$$p(t_r|w, z) \propto \sum_{t_s=1}^{T_s} n(w, t_s + t_r) p(z, t_s|w, t_s + t_r)$$

With some random initial numbers of model parameters, the E-step and the M-step are alternately calculated until an optimum value of data log likelihood is reached. Therefore, for a particular motif length m, the two probability functions, $p(z)$ and $p(t_s|z)$, determine the motif and its occurrences. For more information, see Varadarajan, Emonet, and Odobez (2010a) and Varadarajan, Emonet, and Odobez (2010b).

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 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.

Example 10.1: Brute-Force Method

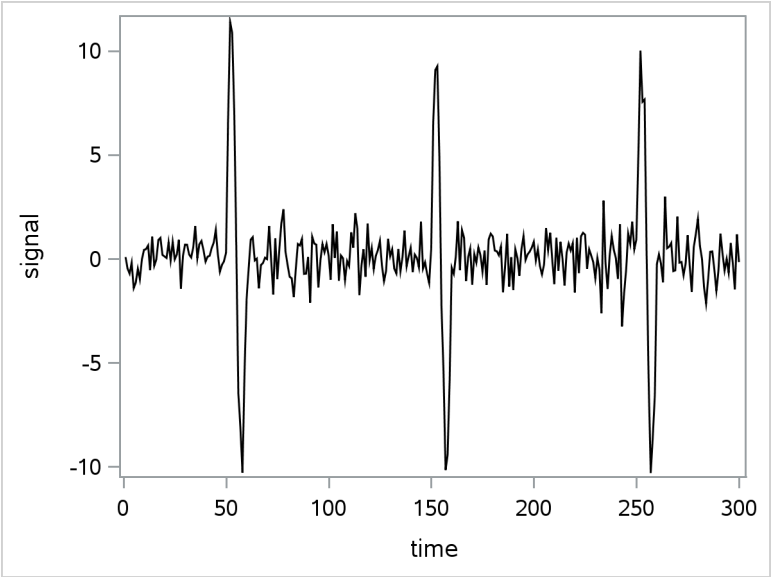
This example uses simulated data that are generated by the following SAS code as a simulated time series with a sine curve as a motif and background data that are from the standard normal distribution. The planted motif instances occur at times 50, 150, and 250. The length of the motif is 10. The length of the time series is 300.

```
%let motif_length = 10;
%let sequence_length = 300;
%let motif_position = (50,150,250);
%let n_motifs = 3;

data SimuData;
        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;
        output;
```

```
        end;
keep time signal;
run;
```

Output 10.1.1 shows the plot of simulated data, which appear to contain three big sine curves around times 50, 150, and 250.

Output 10.1.1 Motif Simulated Data Plot



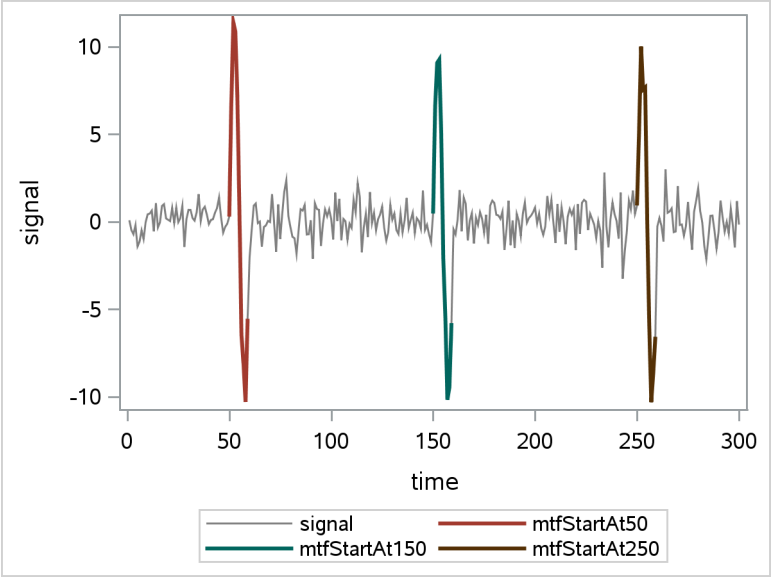
The brute-force method finds the exact three time points where the motif instances start, as shown at Output 10.1.3.

```
proc tsmodel data=mycas.SimuData
                  outobj=(of=mycas.outmotif
                          ofms = mycas.outmotifseries);
var signal;
id  time interval=day;
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 10.1.2 Motif List, Brute-Force Method

Obs	Variable Name	Motif ID	Start Position	Distance
1	signal	1	50	0.2740376908
2	signal	1	150	0.3228669954
3	signal	1	250	0.356297145

Output 10.1.3 Motif Plot, Brute-Force Method



Motif Representation from Motif Instances

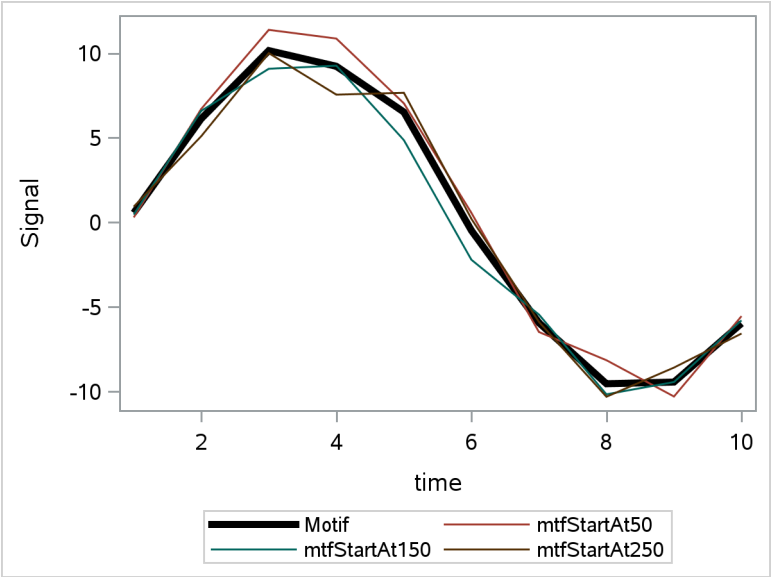
Output 10.1.4 shows that table that was collected by the OUTMTFSERIES collector. The table contains the motif representative series and could be used as a target sequence for motif scoring.

Output 10.1.4 Representative Motif Series

Obs	Variable Name	Motif ID	Motif Series
1	signal	1	0.5649936815
2	signal	1	6.1361722825
3	signal	1	10.158074349
4	signal	1	9.2295711121
5	signal	1	6.5244635657
6	signal	1	-0.474552533
7	signal	1	-5.931737613
8	signal	1	-9.537040246
9	signal	1	-9.434561978
10	signal	1	-5.965699933

Output 10.1.5 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.

Output 10.1.5 Motif and Its Instances Overlay Plot



Example 10.2: Motif Scoring

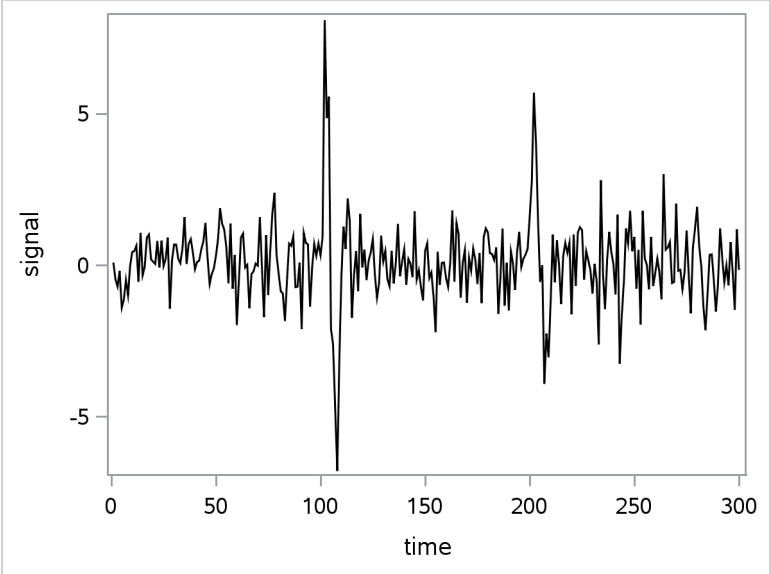
Given a target sequence, motif scoring tries to find subsequences that are most similar to the target sequence in a new time series. This example uses other simulated data, called Scoredata, and the motif representative series that was found in Example 10.1 as a target series. The following code generates score data that contain two noisy sine curves at time points 100 and 200 and searches the target sequence in Scoredata by using a moving window:

```
%let motif_length = 10;
%let sequence_length = 300;
%let motif_position = (100, 200);
%let n_motifs = 2;

data ScoreData;
  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=2*signal+5*sin((i-start[j])/&motif_length*(2*constant('pi')));
      end;
```

```
      end;
      output;
    end;
    keep time signal;
  run;
```

As you expect from the simulated score data, Output 10.2.1 shows two bumps near the 100 and 200 time points.

Output 10.2.1 Simulated Score Data Plot



The following code shows how to use the MTFSCORE and OUTMTFSCORE objects for motif scoring:

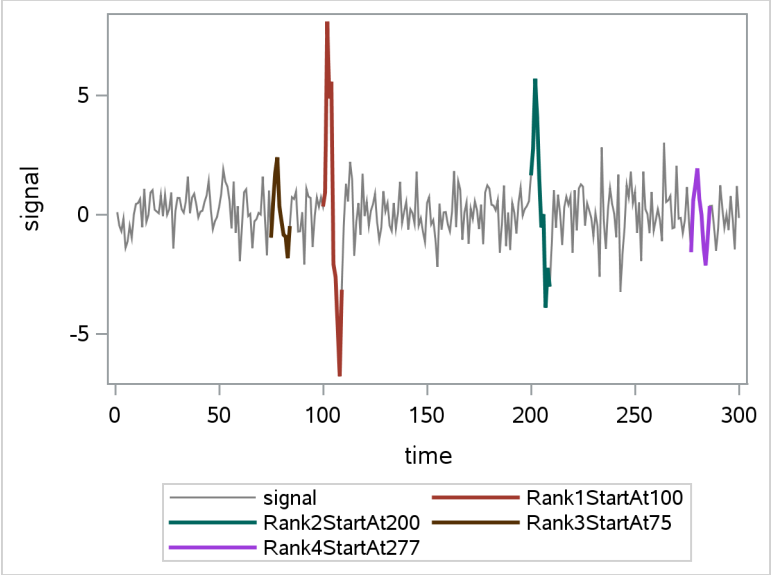
```
proc tsmodel data=mycas.scoredata
             outobj=(of=mycas.outscore
                    );
var signal motifseries;
id  time interval=day;
require mtf;
submit;
        declare object f(MTFSCORE);
        declare object of(OUTMTFSCORE);
        rc = f.Initialize();
        rc = f.SetX(signal);
        rc = f.SetY(motifseries);
        rc = f.SetOption("MOTIFLENGTH",10);
        rc = f.SetOption("NORMALIZE", "Y");
        rc = f.SetOption("MAXMOTIFDIST",2);
        rc = f.SetOption("TOPK", 10);
        rc = f.Run();
        rc = of.Collect(f);
endsubmit;
run;
```

The motif scoring finds that the motif instances start exactly at the 100 and 200 time points as the top two most matched subsequences. The top four results are shown in table form in Output 10.2.2 and in graph form in Output 10.2.3. The distances between the target sequence and the found motif instances are also shown in Output 10.2.2.

Output 10.2.2 Find Motif Instances Given a Target Motif Series

Obs	Variable Name	Motif Instance Rank	Start Position	Distance
1	signal	1	100	0.9693128906
2	signal	2	200	1.1172916593
3	signal	3	75	1.4502947816
4	signal	4	277	1.8895528747

Output 10.2.3 Score Output Plot



Example 10.3: Probabilistic Model Method

This example uses the same simulated data (the Simudata data set) that are used in Example 10.1. Therefore, you expect three motif instances of the sine curve motif at time points 50, 150, and 250. The following code requests four motif sets, each of which shows only the top five instances depending on the cutoff of the start probability:

```
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=obs;
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,
                         "CONVCRIT", 0.001,
                         "CUTOFFPROB", 0.1,
                         "MAXITER", 100,
                         "TOPK", 5,
                         "SEED", 12345
                       );
        rc = f.Run();
        rc = of.Collect(f);
        rc = ofpm.Collect(f);
        rc = ofms.Collect(f);
endsubmit;
print outlog;
run;
```

The probabilistic model produces a posterior probability table, which is shown in Output 10.3.1. The table shows four motif sets and their instances with start position probablities 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 start 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 10.3.1 Probabilistic Model: Motifs and Their Instances

Obs	Variable Name	Motif ID	Start Position	Motif Probability	Motif Start Probability
1	signal	1	256	0.298565969	0.1053295204
2	signal	2	291	0.2817893515	0.1052442557
3	signal	3	75	0.2546943832	0.077292251
4	signal	4	243	0.1649502963	0.1717459314
5	signal	4	46	0.1649502963	0.1345451164
6	signal	4	144	0.1649502963	0.111272976

Example 10.4: 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 101 and 201 time points, as in the following DATA step. The subsequence length is 10.

```
data SimuData2;
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:

```
proc tsmodel data=mycas.SimuData2  outobj=(of = mycas.outanomaly(replace=YES));
       var signal ;
       id time interval = obs;
       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;
```

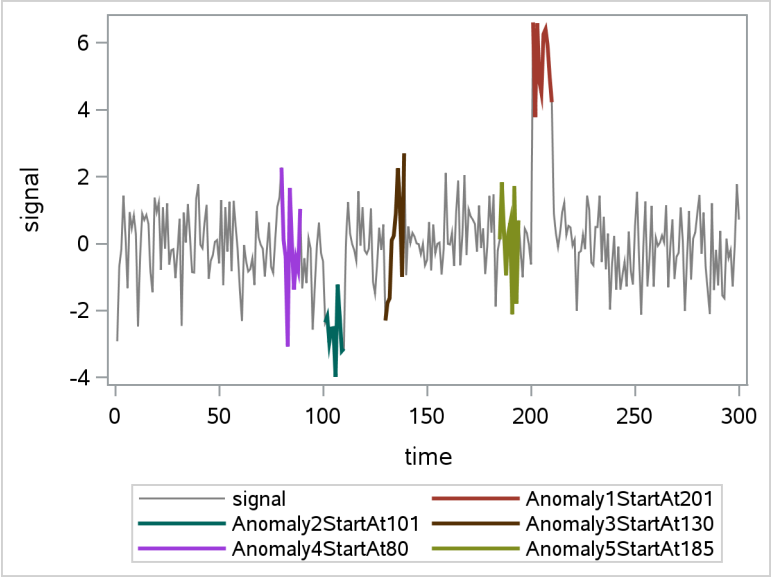
The table in Output 10.4.1 shows that the planted anomaly subsequences are discovered as the first and second anomaly sequences.

Output 10.4.1 Output of Subsequence Anomaly Detection

Obs	Variable Name	Anomaly Rank	Start Position	Distance
1	signal	1	201	15.446353266
2	signal	2	101	6.3363653471
3	signal	3	130	3.7449598719
4	signal	4	80	3.6118347881
5	signal	5	185	3.0896226704

The plot in Output 10.4.2 shows all the five anomaly subsequences; anomalies 1 and 2 show strong abnormality.

Output 10.4.2 Anomaly Subsequence Plot



References

Lin, J., Keogh, E., Lonardi, S., and Chiu, B. (2003). "A Symbolic Representation of Time Series, with Implications for Streaming Algorithms." In *Proceedings of the 8th ACM SIGMOD Workshop on Research Issues in Data Mining and Knowledge Discovery*. New York: ACM. https://dl.acm.org/citation.cfm?doid=882082.882086.

Varadarajan, J., Emonet, R., and Odobez, J.-M. (2010a). "Extracting Motifs from Time Series Generated by Concurrent Activities." NIPS-2010 Workshop on Learning and Planning from Batch Time Series Data, Whistler, British Columbia. http://publications.idiap.ch/index.php/publications/show/2589.

Varadarajan, J., Emonet, R., and Odobez, J.-M. (2010b). "Probabilistic Latent Sequential Motifs: Discovering Temporal Activity Patterns in Video Scenes." In *British Machine Vision Conference*, edited by F. Labrosse, R. Zwiggelaar, Y. Liu, and B. Tiddeman. Durham, UK: BMVA Press. http://www.bmva.org/bmvc/2010/conference/paper117/index.html.

Chapter 11
Utility Package

Contents

Overview

This document 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 11.1 summarizes the object classes that are contained in the UTL package.

Table 11.1 Classes in the Utility Package

UTL Function	Description
CLIMITS	Compute prediction standard errors and confidence limits for specified actual and predicted time series.
OUTNVP	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.
UTLSTAT	Collector object for computing forecast fit statistics for specified actual and predicted time series and storing those statistics in a CAS 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 11.2 defines the common argument types that are used in this chapter.

Table 11.2 Common Argument Types

SAS Data Type	Declaration Syntax
String	LENGTH *x* **$***n***;**
Numeric	*x* or LENGTH *x 8***;**
Numeric array	ARRAY *x***[***n***]**/NOSYMBOLS**;**
Status	*x* or LENGTH *x 8***;**

Status Code Convention

Table 11.3 shows the return code (rc) values that are used by all objects in this chapter.

Table 11.3 Status Codes

rc Value	Meaning
< 0	An unrecoverable error occurred. No result was produced.
= 0	Unconditional success. Requested action completed and normal result produced.
> 0	Conditional success or warning. A result was produced subject to conditions.

Upon returning a negative status code, most methods in the UTL package objects also write a message to the output log that explains the causes of the related failure. These messages provide extremely 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 4, "The TSMODEL Procedure" (*SAS Visual Forecasting: Forecasting Procedures*). Subsequent sections of this document describe the UTL package object classes and their use.

CLIMITS Object Summary

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.

CLIMITS Synopsis

The object declaration statement creates a new object, *obj*, of type CLIMITS:

DECLARE OBJECT *obj* (**CLIMITS**) **;**

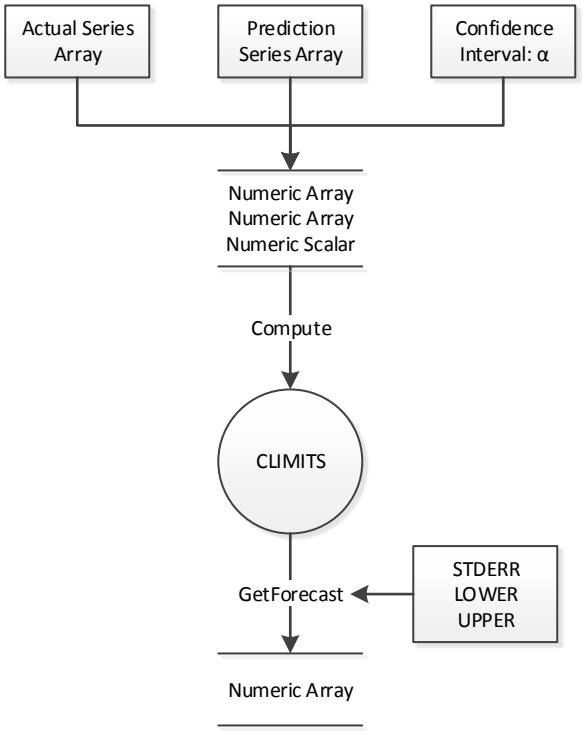
Table 11.4 summarizes the methods that are associated with the CLIMITS object.

Table 11.4 Methods of the CLIMITS Object

CLIMIT Method	Description
Compute	Compute the prediction standard errors and confidence limits for specified actual and predicted time series.
GetForecast	Retrieve a computed forecast series by name.

Figure 11.1 outlines the programmatic data flow through the CLIMITS object.

Figure 11.1 CLIMITS Object Data Flow



CLIMITS Methods

This section describes the methods of a CLIMITS object.

CLIMITS.Compute Method

The Compute method computes the prediction standard errors and confidence limits for specified actual and predicted time series.

Usage: *rc* **=** *obj*.**Compute** *(Actual, Predicted, < Alpha >)* **;**

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).

Parameters

[in] *Actual* Numeric array that corresponds to the actual time series.

[in] *Predicted* Numeric array that corresponds to the predicted time series.

[in] *Alpha* Optional numeric literal or variable that specifies the confidence level used to compute the lower and upper confidence limits. The default value is 0.05.

CLIMITS.GetForecast Method

The GetForecast method places the specified forecast series, *Which*, from the CLIMITS object into the specified numeric array, *Result*.

Usage: *rc* = *obj*.**GetForecast** *(Which, 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).

Parameters

[in] *Which* Character literal or string whose value specifies the forecast series to return. Table 11.5 shows the series types that are supported.

Table 11.5 'Which' Parameters for the CLIMITS.GetForecast Method

Which	Meaning
STDERR	Forecast standard error series
UPPER	Upper confidence limit series
LOWER	Lower confidence limit series

[out] *Result* 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 Summary

The OUTNVP collector object is a powerful tool that 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. The CAS table schema for the OUTNVP object contains the columns that are shown in Table 11.6.

Table 11.6 CAS Table Schema for the OUTNVP Collector Object

Column	Type	Description
NAME	String	Name of the variable
CALL	Numeric	Call count within the BY group
UTAG	Numeric	User-defined numeric tag
VIX	Numeric	Value index (1-based) for the row
VALUE	Numeric	Actual value for variable's row

OUTNVP Synopsis

The object declaration statement creates a new collector object, *obj*, of type OUTNVP:

DECLARE OBJECT *obj* (**OUTNVP**) **;**

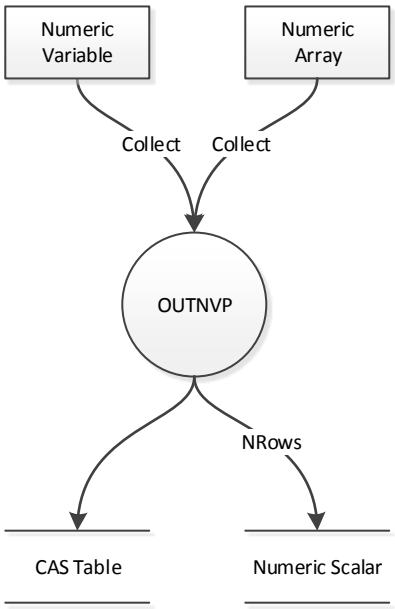
Table 11.7 summarizes the methods and attributes that are associated with the OUTNVP object.

Table 11.7 Methods of the OUTNVP Object

OUTNVP Method	Description
Collect	Collect value for numeric data type (scalar literal or variable or array variable).
nrows	Get the OUTNVP instance row count.

Figure 11.2 outlines the programmatic data flow through the OUTNVP object.

Figure 11.2 OUTNVP Object Data Flow



OUTNVP Methods

This section describes the methods of an OUTNVP collector object.

OUTNVP.Collect Method

The Collect method 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.

Usage: *rc* **=** *obj*.**Collect** *(Variable, < Utag >)* **;**

Parameters

[in] *Variable* Numeric scalar literal or variable or array variable to be collected into the OUTNVP table.

[in] *Utag* Optional numeric scalar literal or variable whose value is to be included in the _UTAG_ column of the OUTNVP table. The default value is a missing value.

OUTNVP.nrows Attribute

The Nrows attribute returns the number of rows that have been collected and stored in the CAS table.

Usage: *rc* **=** *obj***.nrows** *()* **;**

Parameters

There are no parameters associated with this method.

UTLSTAT Object Summary

The UTLSTAT collector object is used to conveniently compute 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. The table schema is shown in Table 11.8. For more information about the HPFENGINE procedure, see *SAS Forecast Server Procedures: User's Guide*.

Table 11.8 CAS Table Scheme for the UTLSTAT Collector Object

Column	Type	Description
NAME	String	Variable name of actual series
MODEL	String	Variable name of predicted series
DFE	Numeric	Degrees of freedom error
N	Numeric	Number of observations
NOBS	Numeric	Number of observations used
NMISSA	Numeric	Number of missing actuals
NMISSP	Numeric	Number of missing predicted values
NPARMS	Numeric	Number of model parameters
TSS	Numeric	Total sum of squares

Table 11.8 *continued*

Column	Type	Description
SST	Numeric	Corrected total sum of squares
SSE	Numeric	Sum of square error
MSE	Numeric	Mean square error
RMSE	Numeric	Root mean square error
UMSE	Numeric	Unbiased mean square error
URMSE	Numeric	Unbiased root mean square error
MAPE	Numeric	Mean absolute percentage of error
MAE	Numeric	Mean absolute error
RSQUARE	Numeric	R-square
ADJRSQ	Numeric	Adjusted R-square
AADJRSQ	Numeric	Amemiya's adjusted R-square
RWRSQ	Numeric	Random walk R-square
AIC	Numeric	Akaike's information criterion
AICC	Numeric	Finite sample corrected Akaike's information criterion
SBC	Numeric	Schwarz Bayesian information criterion
APC	Numeric	Amemiya's prediction criterion
MAXERR	Numeric	Maximum error
MINERR	Numeric	Minimum error
MAXPE	Numeric	Maximum percentage of error
MINPE	Numeric	Minimum percentage of error
ME	Numeric	Mean error
MPE	Numeric	Mean percentage of error
MDAPE	Numeric	Median absolute percentage of error
GMAPE	Numeric	Geometric mean absolute percentage of error
MINPPE	Numeric	Minimum predicted percentage of error
MAXPPE	Numeric	Maximum predicted percentage of error
MPPE	Numeric	Mean predicted percentage of error
MAPPE	Numeric	Mean absolute predicted percentage of error
MDAPPE	Numeric	Median absolute predicted percentage of error

Table 11.8 *continued*

Column	Type	Description
GMAPPE	Numeric	Geometric mean absolute predicted percentage of error
MINSPE	Numeric	Minimum symmetric percentage of error
MAXSPE	Numeric	Maximum symmetric percentage of error
MSPE	Numeric	Mean symmetric percentage of error
SMAPE	Numeric	Mean absolute symmetric percentage of error
MDASPE	Numeric	Median absolute symmetric percentage of error
GMASPE	Numeric	Geometric mean absolute symmetric percentage of error
MINRE	Numeric	Minimum relative error
MAXRE	Numeric	Maximum relative error
MRE	Numeric	Mean relative error
MRAE	Numeric	Mean relative absolute error
MDRAE	Numeric	Median relative absolute error
GMRAE	Numeric	Geometric mean relative absolute error
MASE	Numeric	Mean absolute scaled error
MINAPES	Numeric	Minimum absolute error percentage of standard deviation
MAXAPES	Numeric	Maximum absolute error percentage of standard deviation
MAPES	Numeric	Mean absolute error percentage of standard deviation
MDAPES	Numeric	Median absolute error percentage of standard deviation
GMAPES	Numeric	Geometric mean absolute error percentage of standard deviation

UTLSTAT Synopsis

The object declaration statement creates a new collector object, *obj*, of type UTLSTAT:

DECLARE OBJECT *obj* (**UTLSTAT**) **;**

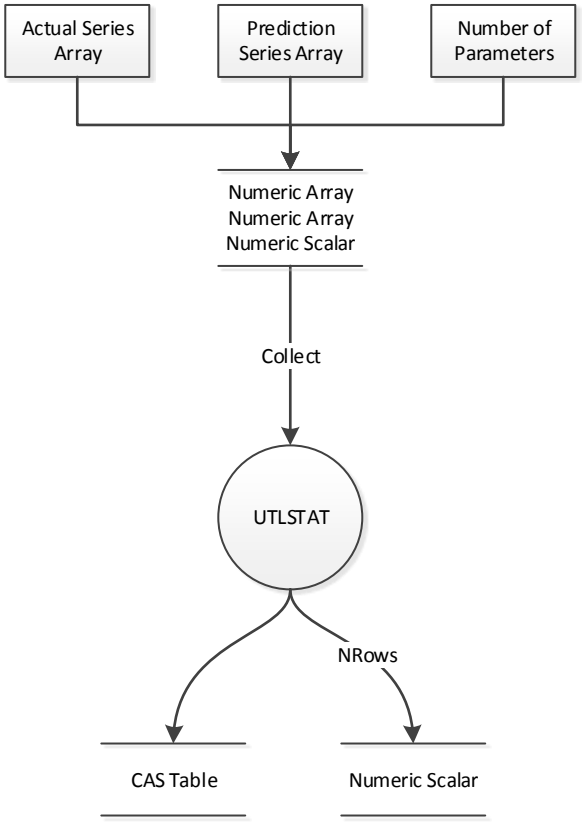
Table 11.9 summarizes the methods of the UTLSTAT object:

Table 11.9 Methods of the UTLSTAT Object

UTLSTAT Method	Description
Collect	Compute and collect forecast fit statistics for a specified ad hoc pair of actual and predicted time series.
nrows	Get the UTLSTAT instance row count.

Figure 11.3 outlines the programmatic data flow through the UTLSTAT object.

Figure 11.3 UTLSTAT Object Data Flow



UTLSTAT Methods

This section describes the methods of an UTLSTAT collector object.

UTLSTAT.Collect Method

The Collect method 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 11.8. 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.

Usage: *rc* **=** *obj*.**Collect** *(Actual, Predicted, < Nparms >)* **;**

Parameters

[in] *Actual* Numeric array that specifies the actual series.

[in] *Predicted* Numeric array that specifies the predicted series.

[in] *Nparms* Optional numeric scalar literal or variable that specifies the number of parameters used by the model that generated the predicted series. The default value is 0.

UTLSTAT.nrows Attribute

The Nrows attribute returns the number of rows that have been collected and stored in the CAS table.

Usage: *rc* **=** *obj*.**nrows** *()* **;**

Parameters

There are no parameters associated with this method.

Examples: UTL Package for the TSMODEL Procedure

Example 11.1: 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 11.8) 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.

```
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.

```
proc tsmodel 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;
var air;
require tsm utl;
submit;
    /* Declare the "Time Series Model" (TSM) object and perform fit */
    declare object esm(tsm);
    rc1 = esm.Initialize();
    if rc1 < 0 then do; stop; end;
    rc1 = esm.SetY(air);
    if rc1 < 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 11.1.1 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 11.1.1 Sample of the Forecast Fit Statistics Computed and Collected by the UTLSTAT Object

Obs	_NAME_	_MODEL_	NOBS	RMSE	MAPE	MAE	RSQUARE	AIC
1	AIR	predict	144	10.579085435	3.0845016398	8.0064787209	0.9921692375	679.35714621

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.outnvpmodel 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.outnvpmodel 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:

```
/* 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;

/* 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=outnvpmodel(obs=18);
run;
```

Output 11.1.2 Sample of the Six Ad Hoc Series Collected by the OUTNVP Object

Obs	_VAR_	_CALL_	_UTAG_	_VIX_	_VALUE_
1	AIR	1	0	1	112
2	predict	2	0	1	111.44700541
3	error	3	0	1	0.5529945856
4	stderr	4	0	1	10.691036547
5	lcl	5	0	1	90.492958825
6	ucl	6	0	1	132.401052
7	AIR	1	0	2	118
8	predict	2	0	2	119.94581467
9	error	3	0	2	-1.945814668
10	stderr	4	0	2	10.691036547
11	lcl	5	0	2	98.991768079
12	ucl	6	0	2	140.89986126
13	AIR	1	0	3	132
14	predict	2	0	3	135.17351431
15	error	3	0	3	-3.173514311
16	stderr	4	0	3	10.691036547
17	lcl	5	0	3	114.21946772
18	ucl	6	0	3	156.1275609

The sequence that is displayed in Output 11.1.2 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.outnvpmodel 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 11.2: 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 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 11.1 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:

```
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.

```
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:

```
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 11.1, 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;

/* 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 11.2.1 Sample of the Six Ad Hoc Series Collected by the OUTNVP Object

Obs	_VAR_	_CALL_	_UTAG_	_VIX_	_VALUE_
1	AIR	1	0	1	112
2	predict	2	0	1	117
3	error	3	0	1	5
4	stderr	4	0	1	5.0408057116
5	lcl	5	0	1	107.12020235
6	ucl	6	0	1	126.87979765
7	AIR	1	0	2	118
8	predict	2	0	2	124
9	error	3	0	2	6
10	stderr	4	0	2	5.0408057116
11	lcl	5	0	2	114.12020235
12	ucl	6	0	2	133.87979765
13	AIR	1	0	3	132
14	predict	2	0	3	135
15	error	3	0	3	3
16	stderr	4	0	3	5.0408057116
17	lcl	5	0	3	125.12020235
18	ucl	6	0	3	144.87979765

The sequence displayed in Output 11.2.1 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 11.2.1 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).

Subject Index

Syntax Index