SAS® Visual Forecasting 8.1
Time Series Packages
# Contents

Chapter 1. Automatic Time Series Analysis and Forecasting Package for the TSMODEL Procedure 1  
Chapter 2. Simple Forecast Service Package for the TSMODEL Procedure 57  
Chapter 3. Time-Frequency Analysis Package for the TSMODEL Procedure 77  
Chapter 4. Time Series Analysis Package for the TSMODEL Procedure 105  
Chapter 5. Time Series Model Package for the TSMODEL Procedure 145  
Chapter 6. Utility Package for the TSMODEL Procedure 185  

Subject Index 205  

Syntax Index 207
Chapter 1
Automatic Time Series Analysis and Forecasting Package for the TSMODEL Procedure

Contents

Overview ........................................... 2
ATSM Package Summary ......................... 3
Status Codes ..................................... 5
TSDF Object Summary ............................. 6
TSDF Synopsis .................................. 6
TSDF Attributes ................................ 10
DIAGNOSE Object Summary ..................... 11
DIAGNOSE Synopsis ............................. 11
DIAGNOSE Attributes ........................... 14
FORENG Object Summary ....................... 14
FORENG Synopsis ............................... 14
FORENG Methods ............................... 15
FORENG Attributes ............................. 19
DIAGSPEC Object Summary ...................... 20
DIAGSPEC Synopsis ............................. 22
DIAGSPEC Methods ............................. 23
SELSPEC Object Summary ....................... 31
SELSPEC Synopsis ............................... 33
SELSPEC Methods ............................... 33
OUTDIAG Object Summary ...................... 36
OUTDIAG Synopsis ............................. 36
OUTDIAG Methods ............................. 36
OUTDIAG Attributes ........................... 37
OUTTEST Object Summary ...................... 37
OUTTEST Synopsis ............................. 38
OUTTEST Methods ............................. 39
OUTTEST Attributes ........................... 39
OUTFOR Object Summary ....................... 39
OUTFOR Synopsis ............................. 40
OUTFOR Methods ............................. 40
OUTFOR Attributes ........................... 41
OUTCOMP Object Summary ...................... 41
## Overview

This document describes the automatic time series analysis and forecasting (ATSM) package. The 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 relevant chapters—for example, chapters about the HPFDIAGNOSE and HPFENGINE procedures—in SAS Forecasting.
Server Procedures: User’s Guide. In addition, a review of the TSMODEL procedure in Chapter 2, “The TSMODEL Procedure” (SAS Visual Forecasting: Forecasting Procedures), and the TSM package in Chapter 5, “Time Series Model Package for the TSMODEL Procedure,” is also helpful. The ATSM package contains several objects, each of which is designed to carry out a particular task in the time series analysis process. The remainder of this section briefly explains the role of the main ATSM objects.

ATSM Package Summary

- The TSDF object is the time series data frame object.
- The DIAGNOSE object is the time series model generator object.
- The FORENG object is the time series forecasting engine.
- The following time series model specification objects enable you to control different aspects of the model identification process:
  - DIAGSPEC (time series model diagnosis control specification)
  - SELSPEC (custom model selection list specification)
- The following ATSM collector objects enable you to create a snapshot of results from the ATSM objects and save them to CAS tables. Many of these CAS tables 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.
  - OUTDIAG (persistent diagnostic control specifications)
  - OUTTEST (time series model parameter estimates collector)
  - OUTFOR (time series model forecast collector)
  - OUTCOMP (time series model components collector)
  - OUTINDEP (input variables in the model)
  - OUTMODELINFO (time series model information collector)
  - OUTSELECT (time series model selection statistics collector)
  - OUTSTAT (time series model fit statistics collector)
  - OUTFMSG (persistent forecast model selection graph XML)
- The following ATSM repeater objects enable you to restore rows from a CAS table to make them available for use by other ATSM objects. Each repeater object defines a CAS table that is determined by the repeater object’s design. 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.
  - INDIAG (replay time series diagnostic control specifications repeater)
  - INFMSG (forecast model selection graph (FMSG) repeater)
  - INEST (automatic time series model parameter estimates repeater)

Figure 1.1 diagrams the relationships between the different objects in the ATSM package.
Figure 1.1 ATSM Data Flow
Table 1.1 summarizes the objects in the ATSM package.

<table>
<thead>
<tr>
<th>Object</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TSDF</td>
<td>Time series data frame object</td>
</tr>
<tr>
<td>DIAGNOSE</td>
<td>Time series model generator object</td>
</tr>
<tr>
<td>FORENG</td>
<td>Time series automatic forecasting engine</td>
</tr>
<tr>
<td>DIAGSPEC</td>
<td>Time series model diagnosis control specification object</td>
</tr>
<tr>
<td>SELSPEC</td>
<td>Model selection list specification object</td>
</tr>
<tr>
<td>OUTDIAG</td>
<td>Persistent diagnostic control specifications</td>
</tr>
<tr>
<td>OUTTEST</td>
<td>Time series model parameter estimates collector object</td>
</tr>
<tr>
<td>OUTFOR</td>
<td>Time series model forecast collector object</td>
</tr>
<tr>
<td>OUTCOMP</td>
<td>Time series model components collector object</td>
</tr>
<tr>
<td>OUTINDEP</td>
<td>Time series model input series collector object</td>
</tr>
<tr>
<td>OUTMODELINFO</td>
<td>Time series model information collector object</td>
</tr>
<tr>
<td>OUTSELECT</td>
<td>Time series model selection statistics collector object</td>
</tr>
<tr>
<td>OUTSTAT</td>
<td>Time series model fit statistics collector object</td>
</tr>
<tr>
<td>OUTFMSG</td>
<td>Persistent forecast model selection graph XML</td>
</tr>
<tr>
<td>INDIAG</td>
<td>Replay time series diagnostic control specifications</td>
</tr>
<tr>
<td>INFMSG</td>
<td>Forecast model selection graph (FMSG) repeater object</td>
</tr>
<tr>
<td>INEST</td>
<td>Automatic time series model parameter estimates repeater object</td>
</tr>
</tbody>
</table>

**Status Codes**

Table 1.2 shows the status return codes (designated by `rc` in method usage statements) that are used in this package. These return codes are numeric values that are returned when a method associated with an object is called. These codes can help determine whether the method executed successfully.
### Table 1.2 Status Codes

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

### TSDF Object Summary

DECLARE OBJECT obj(TSDF) defines an object that is used to group time series variables to be used as input for the other ATSM package objects. As such, TSDF instances are time series data frames. Table 1.3 summarizes the methods that are associated with the TSDF object.

### Table 1.3 Methods of the TSDF Object

<table>
<thead>
<tr>
<th>TSDF Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initialize</td>
<td>Initialize TSDF instance</td>
</tr>
<tr>
<td>AddY</td>
<td>Add Y series</td>
</tr>
<tr>
<td>AddX</td>
<td>Add X series</td>
</tr>
<tr>
<td>AddSeries</td>
<td>Add ancillary series</td>
</tr>
<tr>
<td>SetOption</td>
<td>Set named option</td>
</tr>
<tr>
<td>GetSeries</td>
<td>Get series</td>
</tr>
<tr>
<td>NTid</td>
<td>Return the observation count of the time ID variable</td>
</tr>
</tbody>
</table>

### TSDF Synopsis

Object declaration statement:

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

Method syntax:

```plaintext
rc = obj.Initialize () ;
rc = obj.AddY (YSeries['Name',Value]) ;
rc = obj.AddX (XSeries['Name',Value]) ;
rc = obj.AddSeries (Series['Name',Value]) ;
```


```c
rc = obj.SetOption ("Name", Value [, "Name", Value]) ;
rc = obj.GetSeries ("Name", Result) ;
```

Attributes summary:

```c
ntids = obj.ntid () ;
```

**TSDF.Initialize Method**

**Usage:**
```c
rc = obj.Initialize () ;
```

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

**Parameters**
There are no parameters associated with this method.

**TSDF.AddY Method**

**Usage:**
```c
rc = obj.AddY (YSeries[, "Name", Value]) ;
```

Add a dependent time series array (Y) for the TSDF instance. Optional 'Name', Value pairs can be specified in TSDF.AddY.

**Parameters**
- [in] YSeries Numeric array used to specify the dependent series for the TSDF instance.

<table>
<thead>
<tr>
<th>Table 1.4 Allowed Argument for 'Name', Value Pairs</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Name</strong></td>
</tr>
<tr>
<td>REPMISS</td>
</tr>
</tbody>
</table>

**TSDF.AddX Method**

**Usage:**
```c
rc = obj.AddX (XSeries[, "Name", Value]) ;
```

Add an independent time series array (X) for the TSDF instance. Each call to the AddX method adds the specified X variable to the TSDF instance. This method can be called as many times as needed to specify all the independent variables that are needed. 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 the user-defined time series model (see optional 'Name', Value parameters). Only ARIMA and UCM models support predictors.

**Parameters**
- [in] XSeries Numeric array used to specify an independent series for the TSDF instance.
Table 1.5  Allowed Arguments for 'Name', Value Pairs

<table>
<thead>
<tr>
<th>'Name'</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EXTEND</td>
<td>String</td>
<td>Specifies an independent series forecast method for extension or replacement</td>
</tr>
<tr>
<td>REQUIRED</td>
<td>String</td>
<td>Specifies a variable selection hint for model diagnosis (the default value is 'NO')</td>
</tr>
<tr>
<td>KEEP</td>
<td>Boolean</td>
<td>Keeps a variable if it is referenced in a model (the default value is 0)</td>
</tr>
<tr>
<td>NODIFF</td>
<td>Boolean</td>
<td>Specifies that the variable does not automatically follow Y differencing (the default value is 0)</td>
</tr>
<tr>
<td>CONTROL</td>
<td>Boolean</td>
<td>Specifies that the variable is controllable in forecast functions (the default value is 0)</td>
</tr>
<tr>
<td>REPMISS</td>
<td>Boolean</td>
<td>Replaces embedded missing values with forecasts (the default value is 1)</td>
</tr>
<tr>
<td>ALIAS</td>
<td>String</td>
<td>Specifies an alias for XSeries in lieu of the array name</td>
</tr>
</tbody>
</table>

The EXTEND option controls the generation of future values for the independent variable. The following values are allowed for the EXTEND option:

<table>
<thead>
<tr>
<th>EXTEND</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NONE</td>
<td>Uses as specified</td>
</tr>
<tr>
<td>AVERAGE</td>
<td>Uses the average of historical values</td>
</tr>
<tr>
<td>FIRST</td>
<td>Uses the first nonmissing value</td>
</tr>
<tr>
<td>LAST</td>
<td>Uses the last nonmissing value</td>
</tr>
<tr>
<td>MIN</td>
<td>Uses the minimum value</td>
</tr>
<tr>
<td>MAX</td>
<td>Uses the maximum value</td>
</tr>
<tr>
<td>MEDIAN</td>
<td>Uses the median value</td>
</tr>
<tr>
<td>STOCHASTIC</td>
<td>Uses the ESMBEST model (default)</td>
</tr>
</tbody>
</table>

The REQUIRED option provides a hint about the importance of the independent variable when the TSDF is used in the DIAGNOSE object. For more information, see the HPFDIAGNOSE procedure documentation. The following values are valid for the REQUIRED option:
You can use any method for forecasting the independent series and supply all nonmissing values for it. If you supply the nonmissing values, it does not matter how you specify the EXTEND option; the independent series is always used as specified.

- The REPMISS option controls whether embedded missing values in the historical region of the independent series are replaced via the EXTEND method.

- The KEEP option forces the inclusion of the independent variable when it is referenced from a model in the FORENG object during model selection. When KEEP=0, independent variables that might cause a model fit to fail are selectively removed in the FORENG model selection process. When KEEP=1, the independent variable is always kept and a model fit failure excludes the model from consideration during model selection. The KEEP option functions the same as the REQUIRED= option for PROC HPFENGINE.

### Table 1.8

<table>
<thead>
<tr>
<th>'Name'</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALIAS</td>
<td>String</td>
<td>Name of the series in the TSDF (the default is the argument name)</td>
</tr>
</tbody>
</table>
**TSDF.GetSeries Method**

Usage: \( rc = \text{obj}.\text{GetSeries}('Name',\text{Result}) ; \)

Retrieve the specified series by its name ('Name' or alias) from the TSDF instance into the specified numeric array (Result).

**Parameters**
- [in] 'Name' Character string whose value specifies the variable to return
- [out] Result Numeric array to receive the variable’s series

**TSDF Attributes**

**TSDF.ntid Method**

Usage: \( \text{ntids} = \text{obj}.\text{ntid}() ; \)

Return 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.

**Parameters**
There are no parameters associated with this method.

**TSDF.SetOption Method**

Usage: \( rc = \text{obj}.\text{SetOption}('Name',\text{Value} [, 'Name', \text{Value}]) ; \)

Set named option for the TSDF instance. 'Name' is a character variable. The type of Value depends on the value of 'Name'.

**Parameters**
- [in] 'Name' Character variable that names the option to set.
- [in] Value Specifies the value for the named option.

<table>
<thead>
<tr>
<th>'Name'</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SEASONALITY</td>
<td>Numeric</td>
<td>Seasonal cycle length (a positive integer)</td>
</tr>
</tbody>
</table>
DIAGNOSE Object Summary

DECLARE OBJECT obj(DIAGNOSE) defines a object that is used to generate (diagnose) 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.

DIAGNOSE Synopsis

Object declaration statement:

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

Method syntax:

```plaintext
rc = obj.Initialize ( TSDFObject) ;
rc = obj.SetSpec ( DIAGSPECObject ) ;
rc = obj.Replay ( INDIAGObject) ;
rc = obj.Run ( ) ;
rc = obj.SetOption ( 'Name', Value [,'Name', Value]) ;
rc = obj.GetModel ( 'Family', ModelXML ) ;
```

Attribute summary:

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

Table 1.10 summarizes the methods associated with the DIAGNOSE Object.

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

Usage: \( rc = obj.Initialize(TSDFObject) \);

Initialize a DIAGNOSE instance and specify the time series data for the DIAGNOSE instance.

**Parameters**

[in] **TSDFObject** 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.SetSpec Method

Usage: \( rc = obj.SetSpec(DIAGSPECObject) \);

Set diagnostic control options for the DIAGNOSE instance. Modified control settings in the DIAGSPEC object (DIAGSPECObject) are copied into the control settings for the DIAGNOSE instance for use by the next DIAGNOSE.Run method call.

**Parameters**

[in] **DIAGSPECObject** The DIAGSPEC instance that defines the diagnostic options to be used by the DIAGNOSE instance.

DIAGNOSE.Replay Method

Usage: \( rc = obj.Replay(INDIAGObject) \);

Restore a previously persisted diagnostic control specification from the specified INDIAG instance INDIAGObject. This method establishes the diagnostic control settings in a model identification run of the DIAGNOSE instance. For more information about persisting diagnostic control specifications, see the section “OUTDIAG Object Summary” on page 36.

**Parameters**

[in] **INDIAGObject** The INDIAG instance that specifies the source of the diagnostic control specification to restore.

DIAGNOSE.Run Method

Usage: \( rc = obj.Run() \);

Run model diagnosis for the time series data frame that is specified for the DIAGNOSE instance. **Table 1.11** shows the model families that might be considered during the diagnostic process.

<table>
<thead>
<tr>
<th>Family</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ESM</td>
<td>Seasonal and nonseasonal exponential smoothing models</td>
</tr>
<tr>
<td>ARIMAX</td>
<td>ARIMA model with predictors using ARIMA-REG order</td>
</tr>
</tbody>
</table>
Table 1.11 continued

<table>
<thead>
<tr>
<th>Family</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>REGARIMA</td>
<td>ARIMA model with predictors using REG-ARIMA order</td>
</tr>
<tr>
<td>UCM</td>
<td>UCM model with predictors</td>
</tr>
<tr>
<td>IDM</td>
<td>Intermittent demand (IDM) model</td>
</tr>
</tbody>
</table>

The default families considered include ESM and ARIMAX.

**Parameters**
There are no parameters associated with this method.

**DIAGNOSE.SetOption Method**

Usage: \[ rc = obj.SetOption ('Name', Value [, 'Name', Value]) ; \]

Set named option for the DIAGNOSE instance. ‘Name’ is a character variable. The type of Value depends on the value of ‘Name’.

**Parameters**
- [in] ‘Name’ Character variable that names the option to set.
- [in] Value Specifies the value for the named option.

<table>
<thead>
<tr>
<th>'Name'</th>
<th>Value Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CRITERION</td>
<td>String</td>
<td>Selection statistic name</td>
</tr>
<tr>
<td>BACK</td>
<td>Numeric</td>
<td>Back region for model performance (a nonnegative integer)</td>
</tr>
<tr>
<td>HOLDOUT</td>
<td>Numeric</td>
<td>Holdout region for model selection (a nonnegative integer)</td>
</tr>
</tbody>
</table>

**DIAGNOSE.GetModel Method**

Usage: \[ rc = obj.GetModel ('Family', ModelXML ) ; \]

Fetch the model information by its family (identification technique) from the DIAGNOSE instance.

**Parameters**
- [in] 'Family' Character string whose value specifies the model information to return.
- [out] ModelXML Character variable to receive the XML for the model.
Table 1.13  Allowed Arguments for 'Family'

<table>
<thead>
<tr>
<th>Family</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ESM</td>
<td>Generated ESM model</td>
</tr>
<tr>
<td>ARIMAX</td>
<td>Generated ARIMA model using ARIMA-REG order</td>
</tr>
<tr>
<td>REGARIMA</td>
<td>Generated ARIMA model using REG-ARIMA order</td>
</tr>
<tr>
<td>UCM</td>
<td>Generated UCM model</td>
</tr>
<tr>
<td>IDM</td>
<td>Generated IDM model</td>
</tr>
</tbody>
</table>

If no model is generated for the specified 'Family', a zero-length string is returned. If the specified ModelXML character variable is too short to receive the XML string, a failure status is returned.

**DIAGNOSE Attributes**

**DIAGNOSE.nmodels Method**

Usage:  

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

Return the number of models that are generated from the DIAGNOSE.Run() method. A missing value indicates that a Run() method has not been successfully completed since the last DIAGNOSE.SetData() method.

**Parameters**

There are no parameters associated with this method.

**FORENG Object Summary**

DECLARE OBJECT obj(FORENG) defines an object that is used to automate 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.

**FORENG Synopsis**

Object declaration statement:

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

Method syntax:

```plaintext
rc = obj.Initialize ( TSDFObj ect) ;
rc = obj.Initialize ( DIAGNOSEObject) ;
```
FORENG Methods

FORENG.Initialize Method

Usage: \( rc = \text{obj}.\text{Initialize} (\text{TSDFObject}) \);  

Initialize 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 ESMBEST.
Chapter 1: Automatic Time Series Analysis and Forecasting Package for the TSMODEL Procedure

Parameters
[in] TSDFObject The TSDF object that holds the time series data to be forecast by the FORENG instance.

FORENG.Initialize Method
Usage: \( rc = \text{obj}.\text{Initialize} \left( \text{DIAGNOSEObject} \right) \);
Initialize the FORENG instance to use the generated models and time series data frame from the specified DIAGNOSE instance.

Parameters
[in] DIAGNOSEObject The DIAGNOSE object that holds the time series data and generated model information that are used to initialize the FORENG instance.

FORENG.AddFrom Method
Usage: \( rc = \text{obj}.\text{AddFrom} \left( \text{specObject} \right) \);
Add models from a source instance in the FORENG object’s model selection graph. Models can be included from the objects shown in Table 1.15.

Table 1.15 Model Objects for the FORENG.Run Method

<table>
<thead>
<tr>
<th>Object</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SELSPEC</td>
<td>Includes the model selection specification</td>
</tr>
<tr>
<td>ESMSPEC</td>
<td>Includes the exponential smoothing model specification</td>
</tr>
<tr>
<td>UCMSPEC</td>
<td>Includes the UCM model specification</td>
</tr>
<tr>
<td>IDMSPEC</td>
<td>Includes the IDM model specification</td>
</tr>
<tr>
<td>ARIMASPEC</td>
<td>Includes the ARIMA model specification</td>
</tr>
</tbody>
</table>

Note:

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

2. Calling FORENG.AddFrom subsequent to a FORENG instance results in a replay that augments the restored model selection graph with the models from the SpecObject and force TASK=SELECT for the subsequent FORENG.Run method.

Parameters
[in] specObject source of models to add.
FORENG.Replay Method

Usage:  \( rc = \text{obj}.\text{Replay} (\text{infmsgObject} [, \text{inestObject}]) \);

Restore a previously persisted forecast model selection graph (FMSG) from the specified INFMSG instance. This establishes the context to allow for a model selection and forecasting run of the FORENG instance. Optionally, you can specify an INEST instance that supplies previously persisted parameter estimates for the restored FMSG. This 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 for PROC HPFENGINE. If you restore a previously saved FMSG, you can call the FORENG.AddFrom method to augment the restored model set with other models as desired. 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 instance. For more information about persisting FMSG specifications, see the section “OUTFMSG Object Summary” on page 48; for more information about persisting FMSG model parameter estimates, see the section “OUTEST Object Summary” on page 37.

Parameters

- \( \text{inestObject} \)  
  optional argument that specifies source of the model parameter estimates to restore. This both subsets the FMSG model set to those 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.Run method. You can change this by using the FORENG.SetOption method.

FORENG.Run Method

Usage:  \( rc = \text{obj}.\text{Run} () \);

Run 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.

Parameters

There are no parameters associated with this method.

FORENG.GetForecast Method

Usage:  \( rc = \text{obj}.\text{GetForecast} (\text{Which}, \text{Result}) \);

Get the specified forecast series (\( \text{Which} \)) from the FORENG instance and store it in the specified numeric array (\( \text{Result} \)). \( \text{Which} \) is a case-insensitive character string.
Parameters
[in] Which Character string whose value specifies the forecast series to return
[out] Result Numeric array to receive the forecast series

Table 1.16 Allowed Arguments for Which

<table>
<thead>
<tr>
<th>Which</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>PREDICT</td>
<td>Prediction series</td>
</tr>
<tr>
<td>ERROR</td>
<td>Prediction errors</td>
</tr>
<tr>
<td>STDERR</td>
<td>Prediction standard error series</td>
</tr>
<tr>
<td>UPPER</td>
<td>Upper confidence limit series</td>
</tr>
<tr>
<td>LOWER</td>
<td>Lower confidence limit series</td>
</tr>
</tbody>
</table>

FORENG.GetXSeries Method

Usage: \( rc = \text{obj. GetXSeries (WhichX, XSeries)}; \)

Get the specified X series (WhichX) from the FORENG instance and store it in the specified numeric array (XSeries). WhichX is a case-insensitive character string. 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.

Parameters
[in] WhichX Character string that specifies the name of the X series to return
[out] XSeries Numeric array to receive the X series

FORENG.SetOption Method

Usage: \( rc = \text{obj.SetOption (‘Name’, Value [, ‘Name’, Value])}; \)

Set the named option for the FORENG instance. ‘Name’ is a character variable. The type of Value depends on the value of ‘Name’.

Parameters
[in] ‘Name’ Character variable that names the option to set.
[in] Value Specifies the value for the named option.
### Table 1.17  Allowed Arguments for 'Name', Value Pairs

<table>
<thead>
<tr>
<th>'Name'</th>
<th>Value Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALPHA</td>
<td>Numeric</td>
<td>Significance level for forecast confidence bands (0 &lt; Value &lt; 1)</td>
</tr>
<tr>
<td>CRITERION</td>
<td>String</td>
<td>Fit statistic mnemonic. For more information, see the CRITERION= option in the PROC HPFDIAGNOSE statement in the HPFDIAGNOSE chapter in SAS Forecast Server Procedures: User’s Guide</td>
</tr>
<tr>
<td>LEAD</td>
<td>Numeric</td>
<td>Forecast lead (a nonnegative integer)</td>
</tr>
<tr>
<td>BACK</td>
<td>Numeric</td>
<td>Back region (a nonnegative integer)</td>
</tr>
<tr>
<td>HOLDOUT</td>
<td>Numeric</td>
<td>Holdout region (a nonnegative integer)</td>
</tr>
<tr>
<td>HOLDOUTPCT</td>
<td>Numeric</td>
<td>Holdout percentage (0 ≤ Value ≤ 1.0)</td>
</tr>
<tr>
<td>TASK</td>
<td>String</td>
<td>Run task (SELECT, FORECAST, FIT, UPDATE)</td>
</tr>
<tr>
<td>HORIZON</td>
<td>Numeric</td>
<td>Forecast horizon reference time</td>
</tr>
</tbody>
</table>

### FORENG Attributes

**FORENG.nfor Method**

**Usage:**

```
nfor = obj.nfor () ;
```

Return 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.

**FORENG.criterion Method**

**Usage:**

```
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.

**Parameters**

There are no parameters associated with this method.
**DIAGSPEC Object Summary**

DECLARE OBJECT obj(DIAGSPEC) defines an object that is used as a control 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 1.18 lists the time series model families that are supported.

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

Once configured, the DIAGSPEC object can be used to set the search settings of the DIAGNOSE object (DIAGNOSEObject.SetSpec( DIAGSPECObject)). In turn, this DIAGNOSE object searches for appropriate models for the target series in the data frame that initialized the DIAGNOSE object. Figure 1.2 diagrams the different methods of the DIAGSPEC object.
Figure 1.2 DIAGSPEC Data Flow

DIAGSPEC
SetDiagSpec
Collect
General options for diagnostic control
Transformation Options
UCM Options
Refinement options for UCM
SetOption
SetTransform
SetESM
SetUCMRefine
SetCombine
SetARIMAXRefine
SetIDM
SetCombine
ARIMX Options
Refinement options for ARIMAX
ESM Options
Combination Model Options
IDM Options

INobj={}
Outobj={}

OUTDIAG Table

OUTobj={}

INobj={}
Outobj={}

OUTDIAG

DIAGSPEC
DIAGSPEC Synopsis

Object declaration statement:

```
DECLARE OBJECT obj (DIAGSPEC) ;
```

Method syntax:

```
rc = obj.Open ( ) ;
rc = obj.Close ( ) ;
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.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,'Name',Value,...]) ;
```

Table 1.19 summarizes the methods that are associated with the DIAGSPEC Object.

<table>
<thead>
<tr>
<th>DIAGSPEC Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Open</td>
<td>Open the DIAGSPEC object</td>
</tr>
<tr>
<td>Close</td>
<td>Close the DIAGSPEC object</td>
</tr>
<tr>
<td>SetOption</td>
<td>Set overall control options</td>
</tr>
<tr>
<td>SetTransform</td>
<td>Set transform test options</td>
</tr>
<tr>
<td>SetTrend</td>
<td>Set trend test options</td>
</tr>
<tr>
<td>SetESM</td>
<td>Set ESM diagnostic options</td>
</tr>
<tr>
<td>SetIDM</td>
<td>Set IDM diagnostic options</td>
</tr>
<tr>
<td>SetARIMAX</td>
<td>Set ARIMA diagnostic options</td>
</tr>
<tr>
<td>SetARIMAXRefine</td>
<td>Set ARIMA parameter refinement options</td>
</tr>
<tr>
<td>SetUCM</td>
<td>Set UCM diagnostic options</td>
</tr>
<tr>
<td>SetUCMRefine</td>
<td>Set UCM parameter refinement options</td>
</tr>
</tbody>
</table>
The following remarks apply to 'Name', Value pairs in the various DIAGSPEC methods:

1. 'Name' values are always case-insensitive.

2. Value 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 affect enablement of the diagnostic tests for that model family with the associated default diagnostic control options. These property methods differ from the others that require at least one 'Name'/Value pair.

Methods that can be called with no arguments include:

- SetESM
- SetIDM
- SetARIMAX
- SetARIMAXRefine
- SetUCM
- SetUCMRefine

In all of 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.Open Method**

Usage: `rc = obj.Open();`

Open the DIAGSPEC instance for configuration. This initializes an empty diagnostic specification.

**Parameters**

There are no parameters associated with this method.

**DIAGSPEC.Close Method**

Usage: `rc = obj.Close();`

Finalize the diagnostic settings in the DIAGSPEC instance. This prepares DIAGSPEC to be used in a DIAGNOSE instance or to be imported to a DIAGSPEC instance for printing or storage to a specification repository catalog.
Parameters
There are no parameters associated with this method.

DIAGSPEC.SetOption Method

Usage:
\[
rc = obj.SetOption (\text{'Name'}, Value[, \text{'Name'}, Value,...]) ;
\]
Set named option for the DIAGSPEC instance. \text{'Name'} is a character variable. The type of Value depends on the value of \text{'Name'}.

Parameters
- \text{[in] 'Name'} Character variable that names the option to set.
- \text{[in] Value} Specifies the value for the named option.

Table 1.20  Allowed Arguments for 'Name', Value Pairs

<table>
<thead>
<tr>
<th>Name</th>
<th>Value Type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIGLEVEL</td>
<td>Numeric</td>
<td>Significance level. Must be between 0 and 1.</td>
</tr>
<tr>
<td>CRITERION</td>
<td>String</td>
<td>Fit statistic mnemonic. For more information, see the CRITERION= option in the PROC HPFDIAGNOSE statement in the HPFDIAGNOSE chapter in SAS Forecast Server Procedures: User’s Guide. The default is RMSE.</td>
</tr>
<tr>
<td>INPUTMISSPCT</td>
<td>Numeric</td>
<td>Input missing percentage. Must be between 0 and 1.</td>
</tr>
<tr>
<td>ENTRYPCT</td>
<td>Numeric</td>
<td>AIC or SBC improvement percentage. Must be between 0 and 1.</td>
</tr>
<tr>
<td>PREFILTER</td>
<td>String</td>
<td>Prefilter control (YES, MISSING, EXTREME, or BOTH). The default is YES.</td>
</tr>
<tr>
<td>DELAYINPUT</td>
<td>Numeric</td>
<td>Input variable lag. Must be a nonnegative value.</td>
</tr>
<tr>
<td>DELAYEVENT</td>
<td>Numeric</td>
<td>Event variable lag. Must be a nonnegative value.</td>
</tr>
<tr>
<td>TESTINPUT</td>
<td>String</td>
<td>Test input control (NONE, TRANSFORM, TREND, or BOTH. The default is NONE.</td>
</tr>
<tr>
<td>SELECTINPUT</td>
<td>String</td>
<td>Select input control (SELECT or ALL). The default is SELECT.</td>
</tr>
<tr>
<td>SELECTINPUT</td>
<td>Numeric</td>
<td>Best N input variables. Must be a value no less than 1.</td>
</tr>
</tbody>
</table>
DIAGSPEC.SetARIMAX Method

Usage: \( \texttt{rc} = \texttt{obj.SetARIMAX (\textquoteleft Name, Value[, Name, Value, ...])} ; \)

Set control options for ARIMAX model diagnostics to be performed. \textquoteleft Name\textquoteright{} is a character variable. The type of \textit{Value} depends on the value of \textquoteleft Name\textquoteright{}.

**Parameters**

[in] \textquoteleft Name\textquoteright{} Character variable that names the option to set.

[in] \textit{Value} Specifies the value for the named option.

<table>
<thead>
<tr>
<th>Name</th>
<th>Value Type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIGLEVEL</td>
<td>Numeric</td>
<td>Significance level. A number between 0 and 1.</td>
</tr>
<tr>
<td>PERROR</td>
<td>Array[2]</td>
<td>AR order for MINIC error. ( \text{PERROR}[1]=\text{Min}, \text{PERROR}[2]=\text{Max} )</td>
</tr>
<tr>
<td>P</td>
<td>Array[2]</td>
<td>Nonseasonal AR order range. ( \text{P}[1]=\text{Min}, \text{P}[2]=\text{Max} )</td>
</tr>
<tr>
<td>Q</td>
<td>Array[2]</td>
<td>Nonseasonal MA order range. ( \text{Q}[1]=\text{Min}, \text{Q}[2]=\text{Max} )</td>
</tr>
<tr>
<td>PS</td>
<td>Array[2]</td>
<td>Seasonal AR order range. ( \text{PS}[1]=\text{Min}, \text{PS}[2]=\text{Max} )</td>
</tr>
<tr>
<td>QS</td>
<td>Array[2]</td>
<td>Seasonal MA order range. ( \text{QS}[1]=\text{Min}, \text{QS}[2]=\text{Max} )</td>
</tr>
<tr>
<td>XNUM</td>
<td>Array[2]</td>
<td>Transfer function numerator order range. ( \text{XNUM}[1]=\text{Min}, \text{XNUM}[2]=\text{Max} )</td>
</tr>
<tr>
<td>XDEN</td>
<td>Array[2]</td>
<td>Transfer function denominator order range. ( \text{XDEN}[1]=\text{Min}, \text{XDEN}[2]=\text{Max} )</td>
</tr>
<tr>
<td>CRITERION</td>
<td>String</td>
<td>Identification criterion (AIC or SBC). The default is SBC.</td>
</tr>
<tr>
<td>ESTMETHOD</td>
<td>String</td>
<td>ARIMA estimation method (ULS, CLS, or ML). The default is CLS.</td>
</tr>
<tr>
<td>METHOD</td>
<td>String</td>
<td>Tentative ARMA orders (MINIC, SCAN, or ESACF). The default is MINIC.</td>
</tr>
<tr>
<td>IDENTIFY</td>
<td>String</td>
<td>Identification order (ARIMA, REG, or BOTH). The default is ARIMA.</td>
</tr>
<tr>
<td>NOINT</td>
<td>Numeric</td>
<td>Suppress constant term ( (0 \mid 1) ). The default is 0 or no constant term.</td>
</tr>
</tbody>
</table>

AIC is Akaike’s information criterion, SBC is the Schwarz Bayesian information criterion, MINIC is the minimum information criterion, SCAN is the smallest canonical correlation analysis, and ESACF is the extended sample autocorrelation function.

DIAGSPEC.SetARIMAXRefine Method

Set ARIMA parameter refinement options

Usage: \( \texttt{rc} = \texttt{obj.SetARIMAXRefine (\textquoteleft Name, Value[, Name, Value, ...])} ; \)

Set control options for ARIMAX model parameter refinement. \textquoteleft Name\textquoteright{} is a character variable. The type of \textit{Value} depends on the value of \textquoteleft Name\textquoteright{}. 

Table 1.21  Allowed Arguments for 'Name', Value Pairs
Chapter 1: Automatic Time Series Analysis and Forecasting Package for the TSMODEL Procedure

**Parameters**

- **[in] 'Name'**  Character variable that names the option to set.
- **[in] Value**  Specifies the value for the named option.

**Table 1.22  Allowed Arguments for 'Name', Value Pairs**

<table>
<thead>
<tr>
<th>'Name'</th>
<th>Value Type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIGLEVEL</td>
<td>Numeric</td>
<td>Significance level. A number between 0 and 1.</td>
</tr>
<tr>
<td>ORDER</td>
<td>String</td>
<td>Order of diagnosing model components. The default is INPUT.</td>
</tr>
</tbody>
</table>

Table 1.23 shows the valid ORDER argument values.

**Table 1.23  Allowed Arguments for ORDER**

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARMA:INPUT</td>
<td>Test ARMA coefficients before predictor coefficients</td>
</tr>
<tr>
<td>INPUT:ARMA</td>
<td>Test predictor coefficients before ARMA coefficients</td>
</tr>
</tbody>
</table>

A SetARIMAXRefine method call with no arguments enables ARIMA model diagnosis and sets default options for ARIMA refinement.

**DIAGSPEC.SetESM Method**

**Usage:**  
```
rc = obj.SetESM(['Name',Value,'Name',Value,...]);
```

Set control options for ESM testing to be performed. 'Name' is a character variable. The type of Value depends on the value of 'Name'. A SetESM method call with no arguments enables ESM diagnosis with the default options.

**Parameters**

- **[in] 'Name'**  Character variable that names the option to set.
- **[in] Value**  Specifies the value for the named option.

**Table 1.24  Allowed Arguments for 'Name', Value Pairs**

<table>
<thead>
<tr>
<th>'Name'</th>
<th>Value Type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIGLEVEL</td>
<td>Numeric</td>
<td>Significance level. Must be between 0 and 1.</td>
</tr>
<tr>
<td>METHOD</td>
<td>String</td>
<td>ESM method. The default is BEST.</td>
</tr>
</tbody>
</table>

Table 1.25 shows the valid values for Method.
### Table 1.25 Valid Values for METHOD

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIMPLE</td>
<td>Simple (single) exponential smoothing</td>
</tr>
<tr>
<td>DOUBLE</td>
<td>Double (Brown) exponential smoothing</td>
</tr>
<tr>
<td>LINEAR</td>
<td>Linear (Holt) exponential smoothing</td>
</tr>
<tr>
<td>DAMPTREND</td>
<td>Damped trend exponential smoothing</td>
</tr>
<tr>
<td>ADDSEASONAL</td>
<td>Additive seasonal exponential smoothing</td>
</tr>
<tr>
<td>MULTSEASONAL</td>
<td>Multiplicative seasonal exponential smoothing</td>
</tr>
<tr>
<td>SEASONAL</td>
<td>Same as ADDSEASONAL</td>
</tr>
<tr>
<td>WINTERS</td>
<td>Winters multiplicative method</td>
</tr>
<tr>
<td>ADDWINTERS</td>
<td>Winters additive method</td>
</tr>
<tr>
<td>BEST</td>
<td>Best candidate smoothing model (SIMPLE, LINEAR, DAMPTREND), (ADDSEASONAL, ADDWINTERS, or WINTERS)</td>
</tr>
<tr>
<td>BESTN</td>
<td>Best candidate nonseasonal smoothing model (SIMPLE, LINEAR, or DAMPTREND)</td>
</tr>
<tr>
<td>BESTS</td>
<td>Best candidate seasonal smoothing model (ADDSEASONAL, ADDWINTERS, or WINTERS)</td>
</tr>
</tbody>
</table>

### DIAGSPEC.SetIDM Method

Set IDM diagnostic options

Usage: \[ rc = obj.SetIDM ([\'Name\', Value,\'Name\', Value,...]) ; \]

Set control options for IDM testing to be performed. 'Name' is a character variable. The type of Value depends on the value of 'Name'. The following table lists the allowed options:

**Parameters**
- [in] 'Name' Character variable that names the option to set.
- [in] Value Specifies the value for the named option.

### Table 1.26 Allowed Arguments for 'Name', Value Pairs

<table>
<thead>
<tr>
<th>'Name'</th>
<th>Value Type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>INTERMITTENT</td>
<td>Numeric</td>
<td>Intermittency threshold (the default is 2)</td>
</tr>
<tr>
<td>BASE</td>
<td>Numeric</td>
<td>Base value for demand series (the default is AUTO)</td>
</tr>
<tr>
<td>'Name'</td>
<td>Value Type</td>
<td>Value</td>
</tr>
<tr>
<td>----------</td>
<td>------------</td>
<td>------------------------------------------------------------</td>
</tr>
<tr>
<td>METHOD</td>
<td>String</td>
<td>IDM method (CROSTON, AVERAGE, or BEST; the default is BEST)</td>
</tr>
<tr>
<td>TRANSFORM</td>
<td>Numeric</td>
<td>Transform (same values as SetTransform)</td>
</tr>
<tr>
<td>TRANSPARM</td>
<td>Numeric</td>
<td>Transform parameter (Box-Cox only)</td>
</tr>
</tbody>
</table>

A SetIDM method call with no parameters enables IDM diagnosis with the default options.

**DIAGSPEC.SetTransform Method**

Usage: \( rc = \text{obj.SetTransform} ([ 'Name', \text{Value}, 'Name', \text{Value}, ... ]) ; \)

Set control options for functional transform testing to be performed. 'Name' is a character variable. The type of \( \text{Value} \) depends on the value of 'Name'.

**Parameters**

[in] 'Name'  Character variable that names the option to set.

[in] Value  Specifies the value for the named option.

**Table 1.27  Allowed Arguments for 'Name', Value Pairs**

<table>
<thead>
<tr>
<th>'Name'</th>
<th>Value Type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIGLEVEL</td>
<td>Numeric</td>
<td>Significance level (0 ≤ ( \text{Value} ) ≤ 1.0)</td>
</tr>
<tr>
<td>P</td>
<td>Numeric</td>
<td>AR order for log transform test</td>
</tr>
<tr>
<td>TRANSFORM</td>
<td>String</td>
<td>Transform to use (the default is NONE)</td>
</tr>
<tr>
<td>TRANSPARM</td>
<td>Numeric</td>
<td>Transform parameter (Box-Cox only)</td>
</tr>
<tr>
<td>TRANSOPT</td>
<td>String</td>
<td>Inverse forecasts (MEAN or MEDIAN; the default is MEAN)</td>
</tr>
</tbody>
</table>

**DIAGSPEC.SetTrend Method**

Usage: \( rc = \text{obj.SetTrend} ([ 'Name', \text{Value}, 'Name', \text{Value}, ... ]) ; \)

Set control options for trend testing to be performed. 'Name' is a character variable. The type of \( \text{Value} \) depends on the value of 'Name'.

**Parameters**

[in] 'Name'  Character variable that names the option to set.

[in] Value  Specifies the value for the named option.
Table 1.28  Valid Values for 'Name', Value Pairs

<table>
<thead>
<tr>
<th>'Name'</th>
<th>Value Type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIGLEVEL</td>
<td>Numeric</td>
<td>Significance level (0 ≤ Value ≤ 1.0)</td>
</tr>
<tr>
<td>P</td>
<td>Numeric</td>
<td>AR order for ADF-seasonality tests.</td>
</tr>
<tr>
<td>DIFF</td>
<td>String</td>
<td>Simple differencing (NONE or AUTO)</td>
</tr>
<tr>
<td>DIFFN</td>
<td>Numeric</td>
<td>Forced simple differencing order (DIFF=n)</td>
</tr>
<tr>
<td>DIFFRAN</td>
<td>Numeric</td>
<td>Tests simple differencing range (DIFF=(0:n))</td>
</tr>
<tr>
<td>SDIFF</td>
<td>String</td>
<td>Seasonal differencing (NONE or AUTO)</td>
</tr>
<tr>
<td>SDIFFN</td>
<td>Numeric</td>
<td>Forced seasonal differencing (SDIFF=n)</td>
</tr>
</tbody>
</table>

DIAGSPEC.SetUCM Method

Usage: \( rc = \text{obj.SetUCM} ([\text{'Name'}, \text{Value}, \text{'Name'}, \text{Value}, ...]) \);

Set control options for UCM model diagnostics to be performed.'Name' is a character variable. The type of Value depends on the value of 'Name'.

**Parameters**

[in] 'Name'  Character variable that names the option to set.

[in] Value  Specifies the value for the named option.

Table 1.29  Allowed Arguments for 'Name', Value Pairs

<table>
<thead>
<tr>
<th>'Name'</th>
<th>Value Type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIGLEVEL</td>
<td>Numeric</td>
<td>Significance level (0 ≤ Value ≤ 1.0)</td>
</tr>
<tr>
<td>IRREGULAR</td>
<td>Numeric</td>
<td>Test IRREGULAR component (0 or 1, default=1)</td>
</tr>
<tr>
<td>LEVEL</td>
<td>Numeric</td>
<td>Test LEVEL component (0 or 1; the default is 1)</td>
</tr>
<tr>
<td>SLOPE</td>
<td>Numeric</td>
<td>Test SLOPE component (0 or 1; the default is 1)</td>
</tr>
<tr>
<td>SEASON</td>
<td>Numeric</td>
<td>Test SEASON component (0 or 1; the default is 0)</td>
</tr>
<tr>
<td>CYCLE</td>
<td>Numeric</td>
<td>Test CYCLE component (0 or 1; the default is 0)</td>
</tr>
<tr>
<td>AUTOREG</td>
<td>Numeric</td>
<td>Test AUTOREG component (0 or 1; the default is 0)</td>
</tr>
<tr>
<td>DEPLAG</td>
<td>Numeric</td>
<td>Test DEPLAG component (0 or 1; the default is 0)</td>
</tr>
<tr>
<td>ALL</td>
<td>Numeric</td>
<td>Include all components (0 or 1; the default is 0)</td>
</tr>
</tbody>
</table>
A SetUCM method call with no parameters enables UCM diagnosis with the default options, which includes defaults for UCM refinement.

**DIAGSPEC.SetUCMRefine Method**

Usage: 
```
rc = obj.SetUCMRefine (["Name",Value,"Name",Value,...]) ;
```

Set control options for UCM model diagnostics to be performed. `Name` is a character variable. The type of `Value` depends on the value of `Name`.

**Parameters**

- [in] 'Name' Character variable that names the option to set.
- [in] Value Specifies the value for the named option.

**Table 1.30 Allowed Arguments for 'Name', Value Pairs**

<table>
<thead>
<tr>
<th>'Name'</th>
<th>Value Type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIGLEVEL</td>
<td>Numeric</td>
<td>Significance level (0 ≤ Value ≤ 1.0)</td>
</tr>
<tr>
<td>ORDER</td>
<td>String</td>
<td>Order of subsetting UCM components (the default is INPUT)</td>
</tr>
</tbody>
</table>

A SetUCM method call with no specified parameters enables UCM diagnosis with the default options. This includes defaults for UCM refinement. A SetUCMRefine method call with no specified parameters enables UCM diagnosis and sets default options for UCM refinement.
DELCARE OBJECT obj(SELSPEC) defines an object that is used to define and manipulate forecast model selection list (MSL) objects. The MSL defines a model-based selection strategy for forecasting a dependent variable given a historical information set (TSDF). The MSL is a directed acyclic graph (DAG) of time series models, and model selection 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 these sources:

<table>
<thead>
<tr>
<th>Object</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>DIAGNOSE</td>
<td>Generated models to include</td>
</tr>
<tr>
<td>SELSPEC</td>
<td>Subgraph to include</td>
</tr>
<tr>
<td>INFMSG</td>
<td>Restored SELSPEC models to include</td>
</tr>
<tr>
<td>ESMSPEC</td>
<td>ESM model to include</td>
</tr>
<tr>
<td>ARIMASPEC</td>
<td>ARIMA model to include</td>
</tr>
<tr>
<td>UCMSPEC</td>
<td>UCM model to include</td>
</tr>
<tr>
<td>IDMSPEC</td>
<td>IDM model to include</td>
</tr>
</tbody>
</table>

ESMSPEC, ARIMASPEC, UCMSPEC, and ICMSPEC objects are found in the TSM package.

Figure 1.3 diagrams the different methods of the SELSPEC object.
Figure 1.3 SELSPEC Data Flow
**SELSPEC Synopsis**

Object declaration statement:

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

Method syntax:

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

Table 1.32 summarizes the methods that are associated with the SELSPEC Object.

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

**SELSPEC Methods**

**SELSPEC.AddFrom Method**

Usage: 

```
rc = obj.AddFrom (SourceObj [,ListIndex]) ;
```

Add 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.

**Parameters**

- `[in] SourceObj` Source for models to be added to the SELSPEC instance
- `[in] ListIndex` The index in the SELSPEC instance list where the model DAG from the `SourceObj` is to be inserted. If this parameter is not specified, the DAG from the `SourceObj` is inserted at the next available list index.
Table 1.33  Allowed Arguments for SourceObj

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

SELSPEC.Close Method

Usage: 

\[
rc = obj\textunderscore Close();
\]

Close 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 includes a specification name for which no XML is defined, then a consistency check failure occurs.

Parameters
There are no parameters associated with this method.

SELSPEC.Open Method

Usage: 

\[
rc = obj\textunderscore Open(nspecs);
\]

Open SELSPEC object for construction of a model selection list.

Parameters
\[
[in] \quad nspecs \quad \text{Numeric variable that specifies the number of model specification slots to be defined in the FMSG list.}
\]

SELSPEC.SetDiagnose Method

Usage: 

\[
rc = obj\textunderscore SetDiagnose(\textquote{Name}, \textquote{Value});
\]

Set options for selection diagnostics.
**Parameters**

- **[in] 'Name'**
  Character string that specifies the option name.

- **[in] Value**
  Variable that specifies a value of the option to be set. The type of Value depends on the option 'Name'.

### Table 1.34  Allowed Arguments for 'Name', Value Pairs

<table>
<thead>
<tr>
<th>'Name'</th>
<th>Value Type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>IDMBASE</td>
<td>Numeric</td>
<td>IDM base value (Missing==&gt;Automatic)</td>
</tr>
<tr>
<td>INTERMITTENT</td>
<td>Numeric</td>
<td>Intermittency threshold (0 ≤ Value )</td>
</tr>
<tr>
<td>SEASONTEST</td>
<td>Numeric</td>
<td>Filter models using seasonality test (0 or 1)</td>
</tr>
<tr>
<td>LOGTEST</td>
<td>Numeric</td>
<td>Filter models using log test (0 or 1)</td>
</tr>
</tbody>
</table>

**SELSPEC.SetOption Method**

Usage:  

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

Set named options for SELSPEC object.

**Parameters**

- **[in] 'Name'**
  Character string that specifies the option name.

- **[in] Value**
  Variable that specifies a value of the option to be set. The type of Value depends on the option 'Name'.

### Table 1.35  Allowed Arguments for 'Name', Value Pairs

<table>
<thead>
<tr>
<th>'Name'</th>
<th>Value Type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALPHA</td>
<td>Numeric</td>
<td>Alpha value (0 ≤ Value ≤ 1.0)</td>
</tr>
<tr>
<td>CRITERION</td>
<td>String</td>
<td>Fit statistic mnemonic. For more information, see the CRITERION= option in the PROC HPFDIAGNOSE statement in the HPFDIAGNOSE chapter in SAS Forecast Server Procedures: User’s Guide</td>
</tr>
<tr>
<td>HOLDOUT</td>
<td>Numeric</td>
<td>Holdout region (a positive integer)</td>
</tr>
<tr>
<td>HOLDOUTPCT</td>
<td>Numeric</td>
<td>Holdout percentage (0 ≤ Value ≤ 1.0)</td>
</tr>
<tr>
<td>CHOOSE</td>
<td>String</td>
<td>Preselect model with this name</td>
</tr>
</tbody>
</table>
OUTDIAG Object Summary

DECLARE OBJECT obj(OUTDIAG) defines an object that is used to persist diagnostic control specifications.

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

OUTDIAG Synopsis

Object declaration statement:

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

Method syntax:

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

Attributes summary:

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

Table 1.37 summarizes the methods that are associated with the OUTDIAG Object.

<table>
<thead>
<tr>
<th>OUTDIAG Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Collect</td>
<td>Collect diagnostic specification from OUTDIAG object</td>
</tr>
<tr>
<td>nrows</td>
<td>Number of rows in OUTDIAG object</td>
</tr>
</tbody>
</table>

OUTDIAG Methods

OUTDIAG.Collect Method

Usage: ```plaintext
rc = obj.Collect (SourceObject) ;
```

Collect the diagnostic control specification from source object.
Parameters

[in] **SourceObject**  The instance to be used as the source of the diagnostic control option specification to be persisted.

**Table 1.38**  Allowed Arguments for SourceObject

<table>
<thead>
<tr>
<th>SourceObject</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DIAGNOSE</td>
<td>Renders DIAGSPEC XML from DIAGNOSE instance</td>
</tr>
<tr>
<td>DIAGSPEC</td>
<td>Renders DIAGSPEC XML from DIAGSPEC instance</td>
</tr>
</tbody>
</table>

---

**OUTDIAG Attributes**

**OUTDIAG.nrows Method**

Usage:

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

Query the OUTDIAG object for its current row count.

---

**OUTEST Object Summary**

DECLARE OBJECT obj(OUTEST) defines an object that is used to collect parameter estimates from a FORENG instance. The CAS table schema that is used for storing the parameter estimates is compatible with that used by PROC HPFENGINE for its OUTEST= data set.

<table>
<thead>
<tr>
<th>Column</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>NAME</em></td>
<td>String</td>
<td>Name of the dependent variable</td>
</tr>
<tr>
<td><em>SELECT</em></td>
<td>Numeric</td>
<td>Name of the selection list</td>
</tr>
<tr>
<td><em>MODEL</em></td>
<td>Numeric</td>
<td>Name of model in the selection list</td>
</tr>
<tr>
<td><em>MODELVAR</em></td>
<td>String</td>
<td>Symbol used in the model specification</td>
</tr>
<tr>
<td><em>DSVAR</em></td>
<td>String</td>
<td>Corresponding variable name</td>
</tr>
<tr>
<td><em>VARTYPE</em></td>
<td>String</td>
<td>Type of the variable (dependent or independent)</td>
</tr>
<tr>
<td><em>TRANSFORM</em></td>
<td>String</td>
<td>Transform that is used for the dependent variable</td>
</tr>
<tr>
<td><em>COMPONENT</em></td>
<td>String</td>
<td>Component name within the component model</td>
</tr>
<tr>
<td><em>COMPMODEL</em></td>
<td>String</td>
<td>Component model name</td>
</tr>
<tr>
<td><em>PARM</em></td>
<td>String</td>
<td>Parameter name</td>
</tr>
</tbody>
</table>
Table 1.39  continued

<table>
<thead>
<tr>
<th>Column</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>FACTOR</em></td>
<td>Numeric</td>
<td>Factor number</td>
</tr>
<tr>
<td><em>LAG</em></td>
<td>Numeric</td>
<td>Lag that is used</td>
</tr>
<tr>
<td><em>SHIFT</em></td>
<td>Numeric</td>
<td>Shift that is used</td>
</tr>
<tr>
<td><em>EST</em></td>
<td>Numeric</td>
<td>Parameter estimate</td>
</tr>
<tr>
<td><em>STDERR</em></td>
<td>Numeric</td>
<td>Parameter estimate standard error</td>
</tr>
<tr>
<td><em>TVALUE</em></td>
<td>Numeric</td>
<td>t statistic for parameter estimate</td>
</tr>
<tr>
<td><em>PVALUE</em></td>
<td>Numeric</td>
<td>p-value for parameter estimate</td>
</tr>
<tr>
<td><em>LABEL</em></td>
<td>String</td>
<td>System-generated label for model</td>
</tr>
</tbody>
</table>

OUTEST Synopsis

Object declaration statement:

```
DECLARE OBJECT obj (OUTEST) ;
```

Method syntax:

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

Attributes summary:

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

Table 1.40 summarizes the methods that are associated with the OUTEST Object.

Table 1.40  Methods of the OUTEST Object

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

OUTTEST Methods

OUTTEST.Collect Method

Usage: \[ rc = \textit{obj}.\texttt{Collect} (\textit{FORENGObj}) ; \]

Collect the parameter estimates from the FORENG instance \textit{FORENGObj}.

Parameters

\begin{itemize}
  \item [\textcolor{blue}{\texttt{in}}] \textit{FORENGObj} specifies the FORENG object instance to use as the source of time series model parameter estimates.
\end{itemize}

OUTTEST Attributes

OUTTEST.nrows Method

Usage: \[ \texttt{nrows} = \textit{obj}.\texttt{nrows} () ; \]

Get the current row count from the OUTTEST instance.

Parameters

There are no parameters associated with this method.

OUTFOR Object Summary

DECLARE OBJECT \texttt{obj(OUTFOR)} defines an object that is used to collect forecast series from a FORENG instance. The CAS table schema that is used for storing the set of forecast series variables is compatible with that used by PROC HPFENGINE for its OUTFOR= data set.

<table>
<thead>
<tr>
<th>Table 1.41 Contents of the OUTFOR Object</th>
</tr>
</thead>
<tbody>
<tr>
<td>Column</td>
</tr>
<tr>
<td>--------------</td>
</tr>
<tr>
<td><em>NAME</em></td>
</tr>
<tr>
<td><em>TIMEID</em></td>
</tr>
<tr>
<td><em>ACTUAL</em></td>
</tr>
<tr>
<td><em>PREDICT</em></td>
</tr>
<tr>
<td><em>ERROR</em></td>
</tr>
<tr>
<td><em>STD</em></td>
</tr>
<tr>
<td><em>UPPER</em></td>
</tr>
<tr>
<td><em>LOWER</em></td>
</tr>
</tbody>
</table>
OUTFOR Synopsis

Object declaration statement:

```
DECLARE OBJECT obj (OUTFOR) ;
```

Method syntax:

```
rc = obj.Collect (FORENGObj[,Region]) ;
```

Attributes summary:

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

Table 1.42 summarizes the methods that are associated with the OUTFOR Object.

**Table 1.42  Methods of the OUTFOR Object**

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

OUTFOR Methods

OUTFOR.Collect Method

Usage:  

```
rc = obj.Collect (FORENGObj[,Region]) ;
```

Collect the forecast series from the FORENG instance forecast object. An optional `Region` parameter can be specified to indicate the forecast region to be collected.

**Parameters**

- [in] `FORENGObj`  
  The FORENG object instance to use as the source of time series model parameter estimates.
- [in] `Region`  
  (Optional) The time region to be collected (the default is ALL).

**Table 1.43  Allowed Arguments for Region**

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>String</td>
<td>Collection region(ALL, FIT, or HORIZON; the default is ALL)</td>
</tr>
</tbody>
</table>
OUTFOR Attributes

OUTFOR.nrows Method

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

Get the current row count from the OUTFOR instance.

Parameters
There are no parameters associated with this method.

OUTCOMP Object Summary

DECLARE OBJECT obj(OUTCOMP) defines an object that is used to collect 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 that used by PROC HPFENGINE for its OUTCOMP= data set.

Table 1.44 Contents of the OUTCOMP Object

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

OUTCOMP Synopsis

Object declaration statement:

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

Method syntax:

```plaintext
rc = obj.Collect (FORENGObj[,Region]) ;
```
Attributes summary:

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

Table 1.45 summarizes the methods that are associated with the OUTCOMP Object.

**Table 1.45**  Methods of the OUTCOMP Object

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

### OUTCOMP Methods

**OUTCOMP.Collect Method**

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

Collect the model component series from the FORENG instance \text{FORENGObj}. An optional \text{Region} parameter can be specified to indicate the component series region to be collected.

**Parameters**

- **\text{FORENGObj}** (in) The FORENG object instance to use as the source of time series model component series.
- **\text{Region}** (Optional) The time region to be collected (the default is ALL).

**Table 1.46**  Allowed Arguments for Region

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>String</td>
<td>Collection region (ALL, FIT, or HORIZON; the default is ALL)</td>
</tr>
</tbody>
</table>

### OUTCOMP Attributes

**OUTCOMP.nrows Method**

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

Get the current row count from the OUTCOMP instance.
Parameters
There are no parameters associated with this method.

OUTINDEP Object Summary

DECLARE OBJECT obj(OUTINDEP) defines an object that is used to collect 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 that used by PROC HPFENGINE for its OUTINDEP= data set.

Table 1.47 Contents of the OUTINDEP Object

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

OUTINDEP Synopsis

Object declaration statement:

```
DECLARE OBJECT obj (OUTINDEP) ;
```

Method syntax:

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

Attributes summary:

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

Table 1.48 summarizes the methods that are associated with the OUTINDEP Object.

Table 1.48 Methods of the OUTINDEP Object

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

OUTINDEP.Collect Method

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

Collect the independent variable series that are used in the forecast from the FORENG instance \text{FORENGObj}.

Parameters

[in] \text{FORENGObj} The FORENG object instance to use as the source of time series model independent variable series.

OUTINDEP Attributes

OUTINDEP.nrows Method

Usage: \( rc = obj.\text{nrows} () \);

Get the current row count from the OUTINDEP instance.

Parameters

There are no parameters associated with this method.

OUTMODELINFO Object Summary

DECLARE OBJECT obj(OUTMODELINFO) defines an object that is used to collect characteristics of the selected model from a FORENG instance. The CAS table schema that is used for storing model information is compatible with that used by PROC HPFENGINE for its OUTMODELINFO= data set.

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

### OUTMODELINFO Synopsis

Object declaration statement:

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

Method syntax:

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

Attributes summary:

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

Table 1.50 summarizes the methods that are associated with the OUTMODELINFO Object.

### Table 1.50  Methods of the OUTMODELINFO Object

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

### OUTMODELINFO Methods

**OUTMODELINFO.Collect Method**

Usage:  

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

Collect the model selection information from the FORENG instance `FORENGObj`.

**Parameters**

- `[in]` `FORENGObj` specifies the FORENG object instance to use as the source of time series model selection fit statistics.
OUTMODELINFO Attributes

OUTMODELINFO.nrows Method

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

Get the current row count from the OUTMODELINFO instance.

Parameters
There are no parameters associated with this method.

OUTSELECT Object Summary

DECLARE OBJECT obj(OUTSELECT) defines an object that is used to collect model selection statistics from a FORENG instance. The CAS table schema that is used for storing the fit statistics is compatible with that used by PROC HPFENGINE for its OUTSELECT= data set.

OUTSELECT Synopsis

Object declaration statement:

\[
\text{DECLARE OBJECT } \text{obj(OUTSELECT)} ;
\]

Method syntax:

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

Attributes summary:

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

Table 1.51 summarizes the methods associated with the OUTSELECT Object.

<table>
<thead>
<tr>
<th>OUTSELECT Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Collect</td>
<td>Collect the model selection fit statistics from the FORENG instance ( \text{FORENGObj} ).</td>
</tr>
<tr>
<td>nrows</td>
<td>Get the current row count from the OUTSELECT instance.</td>
</tr>
</tbody>
</table>
OUTSELECT Methods

OUTSELECT.Collect Method

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

Collect the model selection fit statistics from the FORENG instance \text{FORENGObj}.

Parameters

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

OUTSELECT Attributes

OUTSELECT.nrows Method

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

Get the current row count from the OUTSELECT instance.

Parameters

There are no parameters associated with this method.

OUTSTAT Object Summary

DECLARE OBJECT obj(OUTSTAT) defines an object that is used to collect parameter estimates from a FORENG instance, print them, and store them. The CAS table schema used for storing the parameter estimates is compatible with that used by PROC HPFENGINE for its OUTSTAT= data set.

OUTSTAT Synopsis

Object declaration statement:

    DECLARE OBJECT obj (OUTSTAT) ;

Method syntax:

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

Attributes summary:

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

Table 1.52 summarizes the methods that are associated with the OUTSTAT Object.
**Table 1.52** Methods of OUTSTAT Object

<table>
<thead>
<tr>
<th>OUTSTAT Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Collect</td>
<td>Collect the parameter estimates from the FORENG instance FORENGObj.</td>
</tr>
<tr>
<td>nrows</td>
<td>Get the current row count from the OUTSTAT instance.</td>
</tr>
</tbody>
</table>

**OUTSTAT Methods**

**OUTSTAT.Collect Method**

Usage: `rc = obj.Collect (FORENGObj) ;`

Collect the parameter estimates from the FORENG instance FORENGObj.

**Parameters**

[in] FORENGObj The FORENG object instance to use as the source of time series model parameter estimates.

**OUTSTAT Attributes**

**OUTSTAT.nrows Method**

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

Get the current row count from the OUTSTAT instance.

**Parameters**

There are no parameters associated with this method.

**OUTFMSG Object Summary**

DECLARE OBJECT obj(OUTFMSG) defines an object that is used to persist forecast model selection graph (FMSG) XML to a CAS table. OUTFMSG is a PROC TSMODEL collector object.

**Table 1.53** Contents of the OUTFMSG Object

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

Object declaration statement:

```
DECLARE OBJECT obj (OUTFMSG) ;
```

Method syntax:
```
rc = obj.Collect (SourceObject) ;
```

Attributes summary:
```
rc = obj.nrows () ;
```

Table 1.54 summarizes the methods that are associated with the OUTFMSG Object.

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

OUTFMSG Methods

OUTFMSG.Collect Method

Usage:  
```
rc = obj.Collect (SourceObject) ;
```

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

Parameters
- **SourceObject** [in]  The instance to be used as the source of the diagnostic control option specification to be persisted.
Table 1.55  Allowed Arguments for SourceObject

<table>
<thead>
<tr>
<th>SourceObject</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SELSPEC</td>
<td>FMSG XML generated from specification object content</td>
</tr>
<tr>
<td>DIAGNOSE</td>
<td>FMSG XML generated from DIAGNOSE object model XML</td>
</tr>
<tr>
<td>FORENG</td>
<td>FMSG XML generated from FORENG FMSG that is used to forecast</td>
</tr>
</tbody>
</table>

OUTFMSG Attributes

OUTFMSG.nrows Method

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

Query OUTFMSG object for its current row count.

Parameters
There are no parameters associated with this method.

INDIAG Object Summary

DECLARE OBJECT obj(INDIAG) defines a repeater object that is used to replay diagnostic control specifications that have been persisted to a CAS table via the OUTDIAG collector object.

INDIAG Synopsis

Object declaration statement:

```
DECLARE OBJECT obj(INDIAG) ;
```

Attributes summary:

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

Table 1.56 summarizes the methods that are associated with the INDIAG Object.

Table 1.56  Methods of the INDIAG Object

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

INDIAG.nrows Method

Usage: \[ rc = obj.nrows() ; \]
Query INDIAG object for its current row count.

Parameters
There are no parameters associated with this method.

INFMSG Object Summary

DECLARE OBJECT obj(INFMSG) defines a repeater object that is used to replay forecast model selection graph (FMSG) XML specifications from a CAS table for use in a FORENG instance.

INFMSG Synopsis

Object declaration statement:

\[ \text{DECLARE OBJECT } obj \text{ (INFMSG) ;} \]
Attributes summary:

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

Table 1.57 summarizes the methods that are associated with the INFMSG Object.

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

INFMSG Attributes

INFMSG.nrows Method

Usage: \[ rc = obj.nrows() ; \]
Return the number of rows in the INFMSG object. A returned missing value indicates that the INFMSG object has not been successfully configured.
Parameters
There are no parameters associated with this method.

INEST Object Summary

DECLARE OBJECT obj(INEST) defines a repeater object that is used to replay parameter estimates from CAS table for use in a FORENG instance.

INEST Synopsis

Object declaration statement:

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

Attributes summary:

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

Table 1.58 summarizes the methods that are associated with the INEST object.

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

INEST Attributes

INEST.nrows Method

Usage:

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

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

Parameters
There are no parameters 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. In all the examples of this section, the functionality of the TSM package is illustrated using the TSMODEL procedure. This section assumes that you are familiar with the general workings of the TSMODEL procedure; for more information, see Chapter 2, “The TSMODEL Procedure” (SAS Visual Forecasting: Forecasting Procedures).

Example 1.1: 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 chapter of 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 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 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 the collector objects of appropriate type.

```plaintext
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;
quit;
```

Output 1.1.1 shows the results of final model selection step, Output 1.1.2 shows the parameter estimates of the selected model, and Output 1.1.3 shows the forecasts according to the selected model:
**Output 1.1.1** Information about the Selected Model (Partial Output)

**Choice Between the ESM and ARIMA Models**

<table>
<thead>
<tr>
<th>MODEL</th>
<th>SELECTED</th>
<th>LABEL</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARIMAX</td>
<td>NO</td>
<td>ARIMA: AIR ~ P = 1 D = (1,12) NOINT</td>
</tr>
<tr>
<td>ESM</td>
<td>YES</td>
<td>Winters Method (Multiplicative)</td>
</tr>
</tbody>
</table>

**Output 1.1.2** Parameter Estimates of the Selected Model (Partial Output)

**Parameter Estimates for the Multiplicative Winters Model**

<table>
<thead>
<tr>
<th>COMPONENT</th>
<th>EST</th>
<th>STDERR</th>
<th>TVALUE</th>
<th>PVALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>LEVEL</td>
<td>0.3073</td>
<td>0.0315</td>
<td>9.7448</td>
<td>1.8E-17</td>
</tr>
<tr>
<td>TREND</td>
<td>0.00100</td>
<td>0.00302</td>
<td>0.3307</td>
<td>0.7413</td>
</tr>
<tr>
<td>SEASON</td>
<td>0.8749</td>
<td>0.0777</td>
<td>11.2612</td>
<td>2.2E-21</td>
</tr>
</tbody>
</table>

**Output 1.1.3** Forecasts Based on the Selected Model (Partial Output)

**Forecasts Based On the Multiplicative Winters Model**

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

**References**

Chapter 2
Simple Forecast Service Package for the TSMODEL Procedure

Overview

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. It 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 2.1 summarizes the single object class that is contained in the SFS package.

<table>
<thead>
<tr>
<th>SFS Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SFS</td>
<td>Automatically model and forecast univariate time series.</td>
</tr>
</tbody>
</table>

Common Argument Types

Table 2.2 defines the common argument types that are used in this chapter.

<table>
<thead>
<tr>
<th>SAS Data Type</th>
<th>Declaration Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>String</td>
<td>LENGTH x $n$;</td>
</tr>
<tr>
<td>Numeric</td>
<td>x or LENGTH x @;</td>
</tr>
<tr>
<td>Numeric array</td>
<td>ARRAY x[n]/NOSYMBOLS;</td>
</tr>
<tr>
<td>Status</td>
<td>x or LENGTH x @;</td>
</tr>
</tbody>
</table>

Status Code Convention

Table 2.3 shows the return code (rc) values that are used by all objects in this chapter.

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

Upon returning a negative status code, most methods in the SFS package objects also write a message to the output log that explains the causes of the related failure. These messages provide 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 2, “The TSMODEL Procedure” (SAS Visual Forecasting: Forecasting Procedures).
**SFS Object Summary**

The SFS object is used to automatically model and forecast 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 autoregressive integrated moving average with explanatory variable (ARIMAX) 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.

**SFS Synopsis**

The object declaration statement creates a new object, `obj`, of type SFS:

```
DECLARE OBJECT obj ( SFS ) ;
```

Each SFS object in your program is independently instantiated by the `timeData` action for each BY-group thread. Each SFS object in your program is automatically reset by the `timeData` action at the start and end of each BY group. Table 2.4 summarizes the methods that are associated with the SFS object.

**Table 2.4** Methods of the SFS Object

<table>
<thead>
<tr>
<th>SFS Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initialize</td>
<td>Initialize or reset the SFS object for use.</td>
</tr>
<tr>
<td>SetY</td>
<td>Specify the dependent time series.</td>
</tr>
<tr>
<td>AddX</td>
<td>Add an independent time series.</td>
</tr>
<tr>
<td>SetOption</td>
<td>Specify a computational option via a <code>Name, Value</code> pair.</td>
</tr>
<tr>
<td>Run</td>
<td>Automatically model and forecast the dependent variable.</td>
</tr>
<tr>
<td>GetForecast</td>
<td>Retrieve a computed forecast series by name.</td>
</tr>
<tr>
<td>nfor</td>
<td>Get the length (observation count) of the forecast series.</td>
</tr>
<tr>
<td>criterion</td>
<td>Get the in-sample Bayesian information criterion (SBC) fit statistic for the selected time series model.</td>
</tr>
<tr>
<td>model</td>
<td>Get the short name of the selected model that is used to produce the final forecast.</td>
</tr>
</tbody>
</table>
The SFS object has the following calling sequence protocol:

1. You must call SFS.Initialize() before calling any other SFS methods or attributes.

2. You must call SFS.SetY() before calling SFS.Run(). Each call to SFS.SetY() before calling SFS.Run() simply replaces the dependent series.

3. You can call SFS.AddX() zero or more times prior to SFS.Run(). Each SFS.AddX() call includes the specified independent series for consideration during model generation. Independent variables are tracked by the array name that you specify in the SFS.AddX() method call. Calling SFS.AddX() repeatedly with the same array name simply replaces the variable’s series with the values from the most recent call.

4. You can call SFS.SetOption() prior to SFS.Run() to change mutable properties of the SFS instance. Mutable properties in effect at the time of the SFS.Run() call are used.

5. After a successful SFS.Run() call, you can call SFS.GetForecast() to retrieve the forecast series. Calls to SFS.GetForecast() before calling SFS.Run() or calls made after an unsuccessful SFS.Run() return missing values to your Result array.

6. After a successful SFS.Run, you can call the attribute methods to retrieve their respective values. Calls to the attribute methods before calling SFS.Run() or calls made after an unsuccessful SFS.Run() return missing values.

Figure 2.1 outlines the programmatic data flow through the SFS object; each arrow represents a different object method.

Figure 2.1 SFS Object Data Flow
SFS Methods

This section describes the methods of a SFS object.

SFS.Initialize Method

The Initialize method initializes or resets the SFS instance for use.

Usage:

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

Parameters

There are no parameters associated with this method.

SFS.SetY Method

The SetY method specifies the dependent time series array (YSeries) for the SFS instance.

Usage:

```
rc = obj.SetY(YSeries);
```

Parameters

[in] YSeries Numeric array used to specify the dependent series for the SFS instance.

SFS.AddX Method

The AddX method adds an independent time series array (XSeries) for the SFS instance.

Usage:

```
rc = obj.AddX(XSeries);
```

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.

Parameters

[in] XSeries Numeric array used to specify an independent series for the SFS instance.

SFS.SetOption Method

The SetOption method specifies computational options for the SFS instance as Name, Value pairs.

Usage:

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

More than one Name, Value pair can be specified in a single call. Name is a character literal or variable. The type of Value depends on the value of Name. Table 2.5 shows the allowed options.
Table 2.5 Parameters for the SFS.SetOption Method

<table>
<thead>
<tr>
<th>Name</th>
<th>Value Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALPHA</td>
<td>Numeric</td>
<td>Significance level for confidence bands (0 &lt; Value &lt; 1)</td>
</tr>
</tbody>
</table>

Parameters

[in] Name Character literal or variable that names the option.

[in] Value Specifies the value for the named option.

SFS.Run Method

The Run method runs the SFS instance to automatically model and forecast the dependent variable.

Usage: \( rc = \text{obj.Run}() \);

If any independent variables are specified, they are considered during model generation and are included in the candidate models if they are causal to forecast the behavior of the dependent variable. Upon successful completion, various results can be extracted from the SFS instance. Up to three candidate models are generated automatically and might include the models shown in Table 2.6.

Table 2.6 Models Generated by the SFS.Run Method

<table>
<thead>
<tr>
<th>Model</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ESM</td>
<td>Exponential smoothing model</td>
</tr>
<tr>
<td>ARIMAX</td>
<td>ARIMA model generated by finding an ARIMA model for the error series first and then choosing significant inputs and events (ARIMA-REG order)</td>
</tr>
<tr>
<td>REGARIMA</td>
<td>ARIMA model generated by finding a regression model first and then deciding the autoregressive (AR) and moving average (MA) polynomial orders (REG-ARIMA order)</td>
</tr>
</tbody>
</table>

Automatic model selection and forecasting follows by considering the performance of the candidate models based on their in-sample performance as measured by their Bayesian information criterion (SBC) values. ARIMA candidate models are generated only when independent variables are included in the analysis.

Parameters

There are no parameters associated with this method.
**SFS.GetForecast Method**

The GetForecast method fetches the specified forecast series (*Which*) from the SFS instance and writes it into the specified numeric array (*Result*). *Which* is a case-insensitive character string.

Usage: \( rc = obj.GetForecast(\text{Which}, \text{Result}) \);

**Parameters**

- **[in] Which** Character literal or string whose value specifies the forecast series to return. Table 2.7 shows the series types that are supported.

<table>
<thead>
<tr>
<th>Which</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PREDICT</td>
<td>Forecast series</td>
</tr>
<tr>
<td>ERROR</td>
<td>Error series (forecast minus actual)</td>
</tr>
<tr>
<td>STDERR</td>
<td>Forecast standard error series</td>
</tr>
<tr>
<td>UPPER</td>
<td>Upper confidence limit series</td>
</tr>
<tr>
<td>LOWER</td>
<td>Lower confidence limit series</td>
</tr>
</tbody>
</table>

- **[out] Result** Numeric array in which to write the forecast series. If the array length is longer than the forecast series, it is padded with missing values.

**SFS.nfor Attribute**

The nfor attribute returns the length (observation count) of the forecast series for the SFS instance.

Usage: \( rc = obj.nfor() \);

A missing value indicates that the SFS object has not produced a successful forecast.

**Parameters**

There are no parameters associated with this method.

**SFS.criterion Attribute**

The criterion attribute returns the fit statistic value for the final forecast for the SFS instance.

Usage: \( rc = obj.criterion() \);

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.
Parameters

There are no parameters associated with this method.

SFS.model Attribute

The model attribute returns the short name of the selected model that is used to produce the final forecast.

Usage: \( rc = \text{obj}.\text{model}(); \)

A missing value (null string) indicates that the SFS instance has not produced a successful forecast. Table 2.8 shows the possible returned values.

<table>
<thead>
<tr>
<th>Model</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ESM</td>
<td>Exponential smoothing model</td>
</tr>
<tr>
<td>ARIMAX</td>
<td>ARIMA model generated by finding an ARIMA model for the error series first and then choosing significant inputs and events (ARIMA-REG order)</td>
</tr>
<tr>
<td>REGARIMA</td>
<td>ARIMA model generated by finding a regression model first and then deciding the autoregressive (AR) and moving average (MA) polynomial orders (REG-ARIMA order)</td>
</tr>
</tbody>
</table>

Parameters

There are no parameters associated with this method.

Examples: SFS Package for the TSMODEL Procedure

Example 2.1: 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, which is forecast at two different frequencies. The data are naturally recorded as monthly passenger counts. They are forecast at the monthly level and the 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.

```plaintext
data mycas.Air (replace=yes);
  set Sashelp.Air;
run;
```
Example 2.1: Simple Forecasting

The following SAS code uses the TSMODEL procedure to submit the 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 compilation errors when the program is submitted to the timeData.runTimeCode action. 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.Airmons, 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, which is 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.

```sas
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 `timeData` action runs the program that you submit from PROC TSMODEL, it generates a summary of the processing that is performed in your CAS session, as shown in Output 2.1.1.

**Output 2.1.1** Summary of Time Series Processing for mycas.air

**The TSMODEL Procedure**

<table>
<thead>
<tr>
<th>Summary of time series processing for AIR</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of analysis variables</td>
</tr>
<tr>
<td>Number of rows read</td>
</tr>
<tr>
<td>Number of groups read</td>
</tr>
<tr>
<td>Memory for group packages (KB)</td>
</tr>
<tr>
<td>Time to read groups (seconds)</td>
</tr>
<tr>
<td>Time to load groups (seconds)</td>
</tr>
<tr>
<td>Number of data threads</td>
</tr>
<tr>
<td>Thread BY group redundancy</td>
</tr>
<tr>
<td>Minimum time ID</td>
</tr>
<tr>
<td>Maximum time ID</td>
</tr>
<tr>
<td>Minimum time periods</td>
</tr>
<tr>
<td>Maximum time periods</td>
</tr>
<tr>
<td>Number of nodes run</td>
</tr>
<tr>
<td>Number of nodes with data</td>
</tr>
<tr>
<td>Number of nodes with groups</td>
</tr>
<tr>
<td>Number of threads budgeted</td>
</tr>
<tr>
<td>Minimum thread group count</td>
</tr>
<tr>
<td>Maximum thread group count</td>
</tr>
<tr>
<td>Minimum threads active</td>
</tr>
<tr>
<td>Maximum threads active</td>
</tr>
<tr>
<td>Average CPU utilization of nodes with data (%)</td>
</tr>
<tr>
<td>Minimum CPU utilization of nodes with data (%)</td>
</tr>
<tr>
<td>Maximum CPU utilization of nodes with data (%)</td>
</tr>
<tr>
<td>Number of groups processed by submitted code</td>
</tr>
<tr>
<td>Number of groups failing</td>
</tr>
<tr>
<td>Elapsed time to process groups (seconds)</td>
</tr>
<tr>
<td>Number of array table rows produced</td>
</tr>
<tr>
<td>Number of scalar table rows produced</td>
</tr>
</tbody>
</table>

When this code runs, the `timeData.runTimeCode` action forms the monthly time series for the column Air from the CAS table mycas.Air using the time ID variable Date. The program uses the SFS object `esm` to create the best ESM model for this 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 `esm` instance into the array `Airmonfor`, which is then automatically saved to the CAS table `mycas.Airmoana` 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 the use of the OUTSCALAR statement and the OUTSCALAR= option in the PROC TSMODEL statement.
Example 2.1: Simple Forecasting

The following code prints the OUTSCALAR= table from the preceding `timeData` action call. The PROC PRINT results are shown in Output 2.1.2.

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

**Output 2.1.2 OUTSCALAR Table**

<table>
<thead>
<tr>
<th>Obs <em>STATUS</em></th>
<th>nfor</th>
<th>fitstat</th>
<th>rc</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>156</td>
<td>10.579085435</td>
</tr>
</tbody>
</table>

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

```sas
proc tsmodel data=mycas.air
   outarray=mycas.airqtroa(replace=yes)
   outscalar=mycas.airqtros(replace=yes)
   lead=4;
   id date interval=qtr;
   var air/accumulate=avg;
   outarray airqtrfor;
   outscalar nfor fitstat rc;
   require sfs;
   submit;
   declare object esm(sfs);
   rc = esm.Initialize();
   if rc < 0 then do;
     stop;
   end;
   rc = esm.SetY(air);
   if rc < 0 then do;
     stop;
   end;
   rc = esm.Run();
   if rc < 0 then do;
     stop;
   end;
   nfor = esm.nfor();
   fitstat = esm.criterion();
   rc = esm.GetForecast('predict',airqtrfor);
   if rc < 0 then do;
     stop;
   end;
endsubmit;
quit;
```
The following SAS code demonstrates one of many ways that you might want to generate a plot of the two forecasts. This code uses the SGPLOT procedure to display the monthly forecasts as a time series and to overlay the quarterly forecasts as a scatter plot. The PROC SGPLOT results are shown in Output 2.1.3.

```sas
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 2.1.3** Monthly and Quarterly Airline Passenger Forecasts

![Output 2.1.3 Monthly and Quarterly Airline Passenger Forecasts](chart)
Example 2.2: Automatic Forecasting Using Predictor Series

This example shows how you can use the SFS object to automatically model and forecast time series data that include the use of predictor series. The following DATA step loads the Sashelp.Pricedata data set onto the CAS server. This DATA step assumes that your CAS engine libref is named mycas, but you can substitute any appropriately defined CAS engine libref.

```sas
data mycas.Pricedata (replace=yes);
set Sashelp.Pricedata;
run;
```

The example forecasts unit sales (Sale) over the BY groups that are defined by the distinct products (ProductName), considering possible predictor series for price and discount. However, it forecasts sales by using the relative sales series (Relsale) as the dependent variable and relative price (Relprice) as a candidate predictor in place of the accumulated Sale and Price series. The example then rescales the RelSale forecasts back to the original domain for generating a sales forecast. For this particular example, there is no compelling reason to forecast in the domain of the indexed (relative) series, but it demonstrates the power of the timeData action’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 allow for combining custom programming logic with the power of the SFS object to perform sophisticated automatic forecasting.

```sas
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;
rc = lasr.Initialize();
if rc < 0 then do;
   stop;
end;
rc = lasr.SetY(relsale);
if rc < 0 then do;
```

Chapter 2: Simple Forecast Service Package for the TSMODEL Procedure

```plaintext
    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 `timeData` action runs the program that you submit from PROC TSMODEL, it generates a summary of the processing that is performed in your CAS session, as shown in Output 2.2.1.
Output 2.2.1  Summary of Time Series Processing for mycas.pricedata

The TSIMODEL Procedure

<table>
<thead>
<tr>
<th>Summary of time series processing for PRICEDATA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of analysis variables</td>
</tr>
<tr>
<td>Number of rows read</td>
</tr>
<tr>
<td>Number of groups read</td>
</tr>
<tr>
<td>Memory for group packages (KB)</td>
</tr>
<tr>
<td>Time to read groups (seconds)</td>
</tr>
<tr>
<td>Time to load groups (seconds)</td>
</tr>
<tr>
<td>Number of data threads</td>
</tr>
<tr>
<td>Thread BY group redundancy</td>
</tr>
<tr>
<td>Minimum time ID</td>
</tr>
<tr>
<td>Maximum time ID</td>
</tr>
<tr>
<td>Minimum time periods</td>
</tr>
<tr>
<td>Maximum time periods</td>
</tr>
<tr>
<td>Number of nodes run</td>
</tr>
<tr>
<td>Number of nodes with data</td>
</tr>
<tr>
<td>Number of nodes with groups</td>
</tr>
<tr>
<td>Number of threads budgeted</td>
</tr>
<tr>
<td>Minimum thread group count</td>
</tr>
<tr>
<td>Maximum thread group count</td>
</tr>
<tr>
<td>Minimum threads active</td>
</tr>
<tr>
<td>Maximum threads active</td>
</tr>
<tr>
<td>Average CPU utilization of nodes with data (%)</td>
</tr>
<tr>
<td>Minimum CPU utilization of nodes with data (%)</td>
</tr>
<tr>
<td>Maximum CPU utilization of nodes with data (%)</td>
</tr>
<tr>
<td>Number of groups processed by submitted code</td>
</tr>
<tr>
<td>Number of groups failing</td>
</tr>
<tr>
<td>Elapsed time to process groups (seconds)</td>
</tr>
<tr>
<td>Number of array table rows produced</td>
</tr>
<tr>
<td>Number of scalar table rows produced</td>
</tr>
</tbody>
</table>

The following SAS code prints the table that is specified in the OUTSCALAR= option in the PROC TSIMODEL 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 2.2.2.

```sas
proc print data=mycas.saleos; run;
```
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 2.2.3.

```
proc sort data=mycas.saleos out=saleos;
   by productName;
run;
proc sgplot data=saleos;
   scatter x=productName y=fitstat;
run;
```
Example 2.2: Automatic Forecasting Using Predictor Series

**Output 2.2.3** Fit statistics by Product

![Graph showing fit statistics by Product](image-url)
Example 2.3: 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 2.2. 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 2.2, which uses the TSMODEL procedure.

```sas
%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"}
```
Example 2.3: Using SFS with PROC CAS

```r
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 2.3.1.

**Output 2.3.1** Summary of Time Series Processing for mycas.pricedata

**Results from timeData.runTimeCode**

<table>
<thead>
<tr>
<th>Summary of time series processing for PRICEDATA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of analysis variables</td>
</tr>
<tr>
<td>Number of rows read</td>
</tr>
<tr>
<td>Number of groups read</td>
</tr>
<tr>
<td>Memory for group packages (KB)</td>
</tr>
<tr>
<td>Time to read groups (seconds)</td>
</tr>
<tr>
<td>Time to load groups (seconds)</td>
</tr>
<tr>
<td>Number of data threads</td>
</tr>
<tr>
<td>Thread BY group redundancy</td>
</tr>
<tr>
<td>Minimum time ID</td>
</tr>
<tr>
<td>Maximum time ID</td>
</tr>
<tr>
<td>Minimum time periods</td>
</tr>
<tr>
<td>Maximum time periods</td>
</tr>
<tr>
<td>Number of nodes run</td>
</tr>
<tr>
<td>Number of nodes with data</td>
</tr>
<tr>
<td>Number of nodes with groups</td>
</tr>
<tr>
<td>Number of threads budgeted</td>
</tr>
<tr>
<td>Minimum thread group count</td>
</tr>
<tr>
<td>Maximum thread group count</td>
</tr>
<tr>
<td>Minimum threads active</td>
</tr>
<tr>
<td>Minimum threads active</td>
</tr>
<tr>
<td>Average CPU utilization of nodes with data (%)</td>
</tr>
<tr>
<td>Minimum CPU utilization of nodes with data (%)</td>
</tr>
<tr>
<td>Maximum CPU utilization of nodes with data (%)</td>
</tr>
<tr>
<td>Number of groups processed by submitted code</td>
</tr>
<tr>
<td>Number of groups failing</td>
</tr>
<tr>
<td>Elapsed time to process groups (seconds)</td>
</tr>
<tr>
<td>Number of array table rows produced</td>
</tr>
<tr>
<td>Number of scalar table rows produced</td>
</tr>
</tbody>
</table>
The following SAS code generates a series plot of the forecasts and confidence limits for region 1.

```sas
data csaleoa;
  set mycas.csaleoa;
run;
proc sort data=csaleoa;
  by region date;
run;
proc sgplot data=csaleoa(where=(region=1));
  band x=date upper=sale_ucl lower=sale_lcl;
  series x=date y=sale_for;
  scatter x=date y=sale;
run;
```

PROC SGPOIT results are shown in Output 2.3.2

**Output 2.3.2** Sales Forecasts for Region 1
Chapter 3
Time-Frequency Analysis Package for the TSMODEL Procedure

Contents

Overview ................................................................. 78
TFA Package Summary ............................................. 78
FFT Class ............................................................... 79
   FFT Object ......................................................... 79
   FFT.Run Method .................................................. 79
   OUTFFT Object .................................................... 80
   OUTFFT.Collect Method ....................................... 80
   OUTFFT.Nrows Attribute ..................................... 80
   Example ........................................................... 80
FFTC Class ........................................................... 83
   FFTC Object ....................................................... 83
   OUTFFTC Object ................................................ 84
   OUTFFTC.Collect Method ..................................... 84
   OUTFFTC.Nrows Attribute .................................. 85
   Example .......................................................... 85
HILBERT Class ......................................................... 85
   HILBERT Object ................................................ 86
   HILBERT.Run Method .......................................... 86
   OUTHILBERT Object .......................................... 86
   OUTHILBERT.Collect Method ............................... 86
   OUTHILBERT.Nrows Attribute ............................. 87
   Example .......................................................... 87
PWV Class ............................................................. 88
   PWV Object ....................................................... 89
   PWV.Run Method ............................................... 89
   OUTPWV Object ................................................. 90
   OUTPWV.Collect Method ..................................... 90
   OUTPWV.Nrows Attribute ................................... 91
   Example .......................................................... 91
STFT Class ............................................................ 92
   STFT Object ....................................................... 92
   STFT.Run Method .............................................. 92
   OUTSTFT Object ............................................... 93
   OUTSTFT.Nrows Attribute ................................... 94
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 3.1 summarizes the classes in the TFA package.

<table>
<thead>
<tr>
<th>TFA Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FFT</td>
<td>Computes the discrete Fourier transform of a real time series</td>
</tr>
<tr>
<td>FFTC</td>
<td>Computes the discrete Fourier transform of a complex time series</td>
</tr>
<tr>
<td>HILBERT</td>
<td>Computes the analytic signal that corresponds to a real time series</td>
</tr>
<tr>
<td>PWV</td>
<td>Computes the pseudo-Wigner-Ville distribution of a real time series</td>
</tr>
<tr>
<td>STFT</td>
<td>Computes the short-time Fourier transform of a real time series</td>
</tr>
<tr>
<td>WINDOW</td>
<td>Creates a window of a requested type and length</td>
</tr>
</tbody>
</table>
TFA classes can be run independently for each BY group by specifying the BY statement. It is necessary in most cases to also add the TRIMID=BOTH option in the ID statement. For an example, see the section “Example” on page 80.

**FFT Class**

```plaintext
DECLARE OBJECT f (FFT) ;
DECLARE OBJECT of (OUTFFT) ;
rc = f.Run ('Name', Value['Name', Value,...]) ;
rc = of.Collect (f) ;
```

The FFT class computes the discrete Fourier transform of a real time series. Given an input array \( y = (y[0], y[1], \ldots, y[n-1]) \) and \( s = \pm 1 \), the output is \( z = (z[0], z[1], \ldots, z[n-1]) \), where

\[
z[t] = \sum_{k=0}^{n-1} y[k] \exp\left(\frac{2\pi i k t}{n}\right), \quad 0 \leq t \leq n - 1
\]

When \( s = -1 \), the output \( z \) is the *forward* discrete Fourier transform of \( y \); when \( s = 1 \), the output \( z \) is the *backward* discrete Fourier transform of \( y \).

**FFT Object**

DECLARE OBJECT obj(FFT) defines an object that is used to compute the discrete Fourier transform of a real time series.

**FFT.Run Method**

Usage: \( rc = \text{obj.Run ('y', Value['sign', Value])} ; \)

**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 3.2.

<table>
<thead>
<tr>
<th>Column</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>Numeric</td>
<td>The real part of the discrete Fourier transform of input time series</td>
</tr>
<tr>
<td>Y</td>
<td>Numeric</td>
<td>The imaginary part of the discrete Fourier transform of input time series</td>
</tr>
</tbody>
</table>

---

**OUTFFT.Nrows Attribute**

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

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

Output 3.1 shows the results.

Figure 3.1 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:

```plaintext
proc tsmodel data=mycas.sunspot outobj=(of=mycas.fft_x(replace=YES));
  var x;
  id i interval=second;
  require tfa;
submit;
  declare object f(FFT);
```
The following statements plot the power spectral density. You can see a spike around 10 years.

```sas
data mycas.power;
  keep Freq Period Power;
  set mycasfft_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;
```

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

Output 3.2 shows the results.

![Figure 3.2 Power Spectral Density of Sunspot Data](image)

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):

```sas
data x;
  input x g;
datalines;
1 1
2 1
3 1
4 1
```
```
data mycas.x;
  set x;
  i = _N_;
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 (OUTFFT) ;**

\[
rc = f.Run('Name', Value['Name', Value, ...]) ;
\]

\[
rc = of.Collect (f) ;
\]

The FFTC class computes the discrete Fourier transform of a complex time series. Given an input array \( y = (y[0], y[1], \ldots, y[n-1]) \) and \( s = \pm 1 \), the output is \( z = (z[0], z[1], \ldots, z[n-1]) \), where

\[
z[t] = \sum_{k=0}^{n-1} y[k] \exp \left( \frac{s2\pi i k t}{n} \right), \quad 0 \leq t \leq n - 1
\]

When \( s = -1 \), the output \( z \) is the *forward* discrete Fourier transform of \( y \); when \( s = 1 \), the output \( z \) is the *backward* discrete Fourier transform of \( y \).

### FFTC Object

**DECLARE OBJECT obj(FFTC) defines an object that is used to compute the discrete Fourier transform of a complex time series.**
FFTC.Run Method

Usage:  \[ rc = \text{obj}.\text{Run}(\text{"Name", Value[,"Name", Value, ...]}); \]

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:

- \( \text{sign} \) 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 \( \text{sign} \) is FORWARD. The forward and backward transforms are inverses to each other in the following sense: performing the backward transform on a time series of length \( n \) followed by the forward transform on the resulting series leads to \( n \) times the original series and vice-versa.

OUTFFTC Object

DECLARE OBJECT obj(OUTFFTC) defines an object that is used to collect output from FFTC instance.

OUTFFTC.Collect Method

Usage:  \[ rc = \text{obj}.\text{Collect}(\text{"FFTCObj"}); \]

The method collects the output of discrete Fourier transform from an FFTC instance and saves the result to a CAS table, whose schema is shown in Table 3.3.

<table>
<thead>
<tr>
<th>Table 3.3</th>
<th>CAS Table Collected with OUTFFTC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Column</td>
<td>Type</td>
</tr>
<tr>
<td>X</td>
<td>Numeric</td>
</tr>
<tr>
<td>Y</td>
<td>Numeric</td>
</tr>
</tbody>
</table>
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:

```sas
data mycas.x;
  input x_re x_im;
i = _N_;
datalines;
1 -1
2 -2
3 -3
4 -4
;
```

```sas
proc tseries 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;
  array y_im[4]/nosymbols;
  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

```sas
DECLARE OBJECT h (HILBERT) ;
DECLARE OBJECT oh (OUTHILBERT) ;
```
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: 

```printf
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: 

```printf
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 3.4.

<table>
<thead>
<tr>
<th>Column</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>Numeric</td>
<td>The real part of the discrete Fourier transform of input time series</td>
</tr>
<tr>
<td>Y</td>
<td>Numeric</td>
<td>The imaginary part of the discrete Fourier transform of input time series</td>
</tr>
</tbody>
</table>
OUTHILBERT.Nrows Attribute

Usage: \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 Hz at t = 0 and 200 Hz at t = 1. */
data mycas.chirp;
  retain pi %sysfunc(constant(pi));
  keep x;
  f0 = 100;
  f1 = 200;
  Fs = 1000;
  do time = 0 to (2-1/Fs) by 1/Fs;
    x = cos(2*pi*(f0*time + 0.5*(f1-f0)*time*time));
    output;
  end;

/* Compute the Hilbert transform of the chirp data and use it compute the instantaneous frequency */
proc tsmodel data=mycas.chirp outobj=(ch=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 3.3 shows the results.

Figure 3.3  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: \( rc = \text{obj.Run}('Name',Value[,,'Name',Value,...]) \);

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.

\( \text{window} \) 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:

\( \text{overlap} \) specifies the extent of overlap between two consecutive windows. The value of overlap must be a nonnegative integer that is strictly less than window_length. The default value is \( \lfloor \frac{n}{2} \rfloor \), where \( n = \text{window_length} \).

\( \text{fftlen} \) 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.

\( \text{center} \) 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.

\( \text{nthreads} \) 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 90) 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 `n threads` 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 101. 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 3.5.

<table>
<thead>
<tr>
<th>Column</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Frequency</td>
<td>Numeric</td>
<td>The normalized frequency index</td>
</tr>
<tr>
<td>Time</td>
<td>Numeric</td>
<td>The time index</td>
</tr>
<tr>
<td>PWV</td>
<td>Numeric</td>
<td>The value of the pseudo-Wigner-Ville distribution</td>
</tr>
</tbody>
</table>
Note: The number of rows in the output table is \( \text{fftlen} \times k \), where

\[
k = \begin{cases} 
\frac{\text{series_length} - 1}{\text{window_length} - \text{overlap}}, & \text{when } \text{fade} = 1 \\
\frac{\text{series_length} - \text{overlap}}{\text{window_length} - \text{overlap}} + 1, & \text{when } \text{fade} = 0
\end{cases}
\]

The frequency index starts from 0 and increments in multiples of \( \frac{1}{2 \times \text{fftlen}} \). The time index starts from 0 and increments in multiples of \( \text{window_length} - \text{overlap} \).

**OUTPWV.Nrows Attribute**

**Usage:**

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

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

**Example**

The following example calls the PWV class on column \( y \) of the mycas.a data set:

```plaintext
data mycas.a;
  input y;
i = _N_;
datalines;
  3
  1
  4
  2;

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];
  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;
```
Chapter 3: Time-Frequency Analysis Package for the TSMODEL Procedure

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 103.

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. 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.
overlap specifies the extent of overlap between two consecutive windows, where overlap must be a nonnegative number that is strictly less than the window length. The default value is $\frac{n}{2}$, where $n$ is the length of the window.
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 the window length. The default value is the larger of 256 and the 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 93).

- 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 3.6.

**Table 3.6  CAS Table Collected with OUTSTFT**

<table>
<thead>
<tr>
<th>Column</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Frequency</td>
<td>Numeric</td>
<td>The normalized frequency index</td>
</tr>
<tr>
<td>Time</td>
<td>Numeric</td>
<td>The time index</td>
</tr>
<tr>
<td>Power</td>
<td>Numeric</td>
<td>The power of the time series that corresponds to the current frequency and time index</td>
</tr>
<tr>
<td>Amplitude</td>
<td>Numeric</td>
<td>The amplitude of the time series that corresponds to the current frequency and time index</td>
</tr>
<tr>
<td>Phase</td>
<td>Numeric</td>
<td>The phase of the time series that corresponds to the current frequency and time index</td>
</tr>
<tr>
<td>Coef_re</td>
<td>Numeric</td>
<td>The real part of the Fourier coefficient</td>
</tr>
<tr>
<td>Coef_im</td>
<td>Numeric</td>
<td>The imaginary part of the Fourier coefficient</td>
</tr>
</tbody>
</table>
Note: The number of rows in the output table is $fftlen \times k$, where $k = \frac{\text{series_length} - \text{overlap}}{\text{window_length} - \text{overlap}} + 1$.

The frequency index starts from 0 and increases in increments of $\frac{1}{fftlen}$. The time index starts from 0 and increases in multiples of $\text{window_length} - \text{overlap}$. The power is computed by $x^2 + y^2$, where $x$ is the real part of the Fourier coefficient (the value stored in the sixth column) and $y$ is the imaginary part of the Fourier-coefficient (the value stored in the seventh column). The amplitude is given by $\sqrt{\text{power}}$, where $\text{power}$ is the value stored in the third column. The phase is computed as $\text{atan2}(y, x)$ where $x$ is the real part of the time series (stored in the sixth column) and $y$ is the imaginary part of the time-series (stored in the seventh column).

### OUTSTFT.Nrows Attribute

**Usage:**

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

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

### Example

The following example calls the STFT class on column $y$ of the `mycas.a` data table:

```plaintext
data mycas.a;
   input y;
   i = _N_;
datalines;
3
1
4
2;

data mycas.a outobj=(os=mycas.outstft(replace=YES))
   outlog=mycas.log logControl=(warning=keep error=keep note=keep);
id i interval=seconds;
var y;
require TFA;
submit;
declare object w(WINDOW);
declare object s(STFT);
declare object os(OUTSTFT);
/**** create a hamming window of size 3 ****/
rc = w.Run('name', 'hamming', 'length', 3); if rc then stop;
array window[1]/nosymbols;
rc = w.Save(window); if rc then stop;
rc = s.Run('y', y, 'window', window, 'overlap', 2, 'fftlen', 4); if rc then stop;
rc = os.Collect(s); if rc then stop;
endsubmit;
run;
```
The WINDOW class creates a window of a requested type and length. The WINDOW class is useful for smoothing spectra. For more information about the window functions that are implemented in the time-frequency analysis package, see Harris (1978).

**WINDOW Object**

DECLARE OBJECT obj(WINDOW) defines an object that is used to compute windows.

**WINDOW.Run Method**

Usage: 

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

**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

\[
    w[i] = 1 - \left| \frac{2i}{N-1} - 1 \right|, \quad 0 \leq i \leq N-1
\]

- **BARTLETT_HANN** specifies a Bartlett-Hann window. For this window type, you do not need to specify any params. This window function is defined as

\[
    w[i] = 0.62 - 0.48\left| \frac{i}{N-1} - 0.5 \right| - 0.38 \cos \frac{2\pi i}{N-1}, \quad 0 \leq i \leq N-1
\]
BLACKMAN specifies a Blackman window. For this window type, you do not need to specify any params. This window is defined as

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

BLACKMAN_HARRIS specifies a Blackman-Harris window. For this window type, you do not need to specify any params. This window function is defined as

$$w[i] = 0.35875 - 0.48829 \cos \frac{2\pi i}{N-1} + 0.14128 \cos \frac{4\pi i}{N-1} - 0.01168 \cos \frac{6\pi i}{N-1}, \hspace{1em} 0 \leq i \leq N-1$$

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

$$w[0] = w[N - 1] = 0$$

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

$$\hspace{1em} + \frac{1}{\pi} \sin \left(\pi \left|1 - \frac{2i}{N-1}\right|\right), \hspace{1em} \text{for} \hspace{1em} 1 \leq i \leq N - 2$$

CHEBYSHEV specifies a Chebyshev window. This window function needs one param: att, whose default value is 100. To define this window, you need to define the nth-degree Chebyshev polynomial, $T_n$, which is the unique polynomial such that $T_n(\cos \theta) = \cos n\theta$ for all values of $\theta$. $T_n(x)$ can be computed as

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

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

$$w[i] = c \left(1 + \frac{2}{T_2M(\beta)} \sum_{k=1}^{M} T_{2M}(\beta \cos \frac{k\pi}{N}) \cos \frac{2\pi k(i - M)}{N}\right), \hspace{1em} 0 \leq i \leq N-1$$

where $\beta = \cosh(\text{acosh}(10^{\text{att}/20})/(N - 1))$ and $c$ is chosen to make the largest term of $w$ equal to 1.

For even $N$, the Chebyshev window of length $N$ can be defined as

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

where $c$ is chosen to make the largest term of $w$ equal to 1.
**FLAT_TOP**

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

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

**GAUSSIAN**

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

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

**HAMMING**

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

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

**HANNING**

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

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

**KAISER**

specifies a Kaiser window. This window function needs one *param*: \( \beta \), whose default value is 0.5. This window function is defined as

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

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

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

**PARZEN**

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

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

The last half of the window is defined by symmetry, which implies \( w[i] = w[N - i - 1] \) for \( 0 \leq i \leq N - 1 \).
RECTANGULAR specifies a rectangular window. For this window type, you do not need to specify any params. This window function is defined as

\[ w[i] = 1, \quad 0 \leq i \leq N - 1 \]

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

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

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: \( rc = obj.\text{Save}(\text{window}) \);

This method saves the result (window array) to a TKCMP dynamic array. For examples, see the sections “PWV Class” on page 88 and “STFT Class” on page 92.

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: \( rc = obj.\text{Collect}('WINDOWObj') \);
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 3.7.

<table>
<thead>
<tr>
<th>Column</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NAME</td>
<td>String</td>
<td>The type of the window</td>
</tr>
<tr>
<td>X</td>
<td>Numeric</td>
<td>The array of numbers that contain the window</td>
</tr>
</tbody>
</table>

### OUTWINDOW.Nrows Attribute

**Usage:**

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

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

### Example

The following DATA step creates a dummy data table:

```r
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:

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

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

That is, $y = Wx$, where $\omega = \exp(-2\pi i/n)$ and $W = (\omega^{(i-1)(j-1)})_{i,j=0}^{n-1}$. 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 $O(n^2)$ 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 $O(n \log n)$ operations. For more information, see Van Loan (1992).

Hilbert Transformation

The HILBERT function computes the analytic signal of the input. The analytic signal that corresponds to a continuous time series $x(t)$ is the complex time series $z(t) = x(t) + i\hat{x}(t)$, where $\hat{x}(t)$ is the Hilbert transform of $x(t)$. In many applications, replacing the original time series by its analytic transform produces better results (Marple 1999).

For a continuous time series $x(t)$ with Fourier transform $X(f) = \int x(t) \exp(-2\pi ift) \, dt$, the Hilbert transform is defined by inverting $X(f)$ over the positive frequencies (Cohen 1995, pg 30):

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

The spectrum of $\hat{x}(t)$ is identical to the spectrum of $x(t)$ for positive frequencies, and the spectrum of $\hat{x}(t)$ is 0 for negative frequencies.

The Hilbert transform of a discrete time series is similarly constructed as a time series whose discrete Fourier transform coincides with that of the input time series for positive spectra and vanishes otherwise. The time-frequency analysis package implementation is based on the method described in Marple (1999).
Time-Frequency Distributions

Time-frequency distributions are standard tools for studying a time series whose frequency behavior varies with time. The discussion here follows the treatment given in Cohen (1995).

An important concept in time frequency analysis is the “energy” of a signal. Let $x(t)$ be a continuous time series with Fourier transform $X(f)$. Then for well-behaved $x(t)$ and $X(f)$, you can consider $E = \int_{-\infty}^{\infty} |x(t)|^2 \, dt = \int_{0}^{2\pi} |X(f)|^2 \, df$ as the “total energy” in $x(t)$, and consequently you can interpret $\int_{t_0}^{t_1} |x(t)|^2 \, dt$ as the total energy of $x(t)$ between the time points $t_0$ and $t_1$. Similarly you can interpret $\int_{f_0}^{f_1} |X(f)|^2 \, df$ to be the energy of $x(t)$ between the frequencies $f_0$ and $f_1$. This also implies that $|x(t)|^2 = \lim_{h \to 0} \frac{1}{h} \int_{t-h}^{t+h} |x(u)|^2 \, du$ can be considered to be the instantaneous energy per unit time at time $t$ and $|X(f)|^2$ can be considered to be the instantaneous energy per unit frequency at $f$.

A time-frequency distribution of a time series $x(t)$ is a function $P_x(t, f)$ of the time index $t$ and frequency $f$ such that $P_x(t, f)$ is a measure of the intensity of energy of $x(t)$ at time $t$ and frequency $f$. That is, given a small $\Delta t$ and small $\Delta f$, you should be reasonably able to interpret $P_x(t, f) \Delta t \Delta f$ as the energy of $x(t)$ that can be attributed to $[x, x + \Delta t] \times [f, f + \Delta f]$.

Because energy is positive, you should expect that $P_x(t, f) \geq 0$ for all $t$ and $f$; this condition is known as positivity. Similarly you should expect $\int_{0}^{2\pi} P_x(t, f) \, df$ to yield the instantaneous energy at time $t$, so you should have $\int_{0}^{2\pi} P_x(t, f) \, df = |x(t)|^2$. Similarly you should expect $\int_{-\infty}^{\infty} P_x(t, f) \, dt = |X(f)|^2$. These two conditions are known as the marginal conditions. Other desirable properties of time-frequency distributions are discussed in Cohen (1995, chapter 6). However, it is not possible for a distribution to simultaneously satisfy positivity and the marginal conditions, and most time-frequency distributions that are used in practice satisfy these conditions only approximately.

The time-frequency analysis package implements the discrete versions of two widely used time-frequency distributions: the pseudo-Wigner-Ville distribution and the short-time Fourier transform. The continuous version of pseudo Wigner-Ville distribution satisfies the marginal property in the special case when it reduces to the Wigner-Ville distribution, but it does not satisfy the positivity condition. The short-time-Fourier transform satisfies the positivity condition, but it does not satisfy the marginal conditions.

Pseudo-Wigner-Ville Distribution

The pseudo-Wigner-Ville distribution is a generalization of the Wigner-Ville distribution. The Wigner-Ville distribution of a continuous time series $x(t)$ is obtained by computing the Fourier transform of $x(t + \tau/2)x(t - \tau/2)$ for fixed $t$ as $\tau$ varies. So the Wigner-Ville distribution of a continuous, possibly complex-valued, time series $x(t)$ is given by $W_x(t, f)$:

$$W_x(t, f) = \int_{-\infty}^{\infty} x(t + \tau/2)x(t - \tau/2) \exp(-2\pi i f \tau) \, d\tau$$

$$= 2 \int_{-\infty}^{\infty} x(t + \tau)x(t - \tau) \exp(-4\pi i f \tau) \, d\tau.$$

It can be shown that the $W_x(t, f)$ is real even when $x(t)$ takes complex values.
The pseudo-Wigner-Ville distribution is a modification of Wigner-Ville distribution that is obtained by an additional term in the defining integral. The pseudo-Wigner-Ville distribution of a continuous time series \( x(t) \) is given by

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

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):

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

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 \( \text{window} = (w[0], \ldots, w[2m]) \), define

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

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 \( \text{series\_length} - 1 \) is replaced with 0.

Define the following:

- \( S = \text{window\_length} - \text{overlap} \)
- Let

\[
k = \begin{cases} 
\left\lfloor \frac{\text{series\_length} - 1}{S} \right\rfloor, & \text{when } \text{fade} = 1 \\
\left\lfloor \frac{\text{series\_length} - \text{overlap}}{S} \right\rfloor, & \text{when } \text{fade} = 0
\end{cases}
\]
- Let

\[
c = \begin{cases} 
0, & \text{when } \text{fade} = 0 \\
L, & \text{when } \text{fade} = 1
\end{cases}
\]

where the \( \text{window\_length} = 2L + 1 \).

The output of the PWV function consists of the evaluation of \( U_x(n, f) \) for \( n = c, c + S, c + 2S, \ldots, c + kS \) and for \( f = 0, \frac{1}{\text{fftlen}}, \ldots, \frac{\text{fftlen} - 1}{\text{fftlen}} \).
When \textit{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 \textit{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 \texttt{hilbert/tsf} option, which replaces the original series with its analytic signal before computation, and the \texttt{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 \texttt{hilbert/tsf} and \texttt{center} parameters: First the value of the \texttt{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 \texttt{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 $w[0], \ldots, w[m]$, the computation of STFT can be considered to be the evaluation of a function $S_x(n, f)$, which measures the strength of the frequency $f$ at time $n$ for different values of $n$ and $f$, where $S_x(n, f)$ is defined as

$$S_x(n, f) = \sum_{k=0}^{m-1} x[n + k]w[k]\exp(-i2\pi kf)$$

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

**References**


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 4.1 summarizes the functions in the TSA package.

<table>
<thead>
<tr>
<th>TSA Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACCUMULATE</td>
<td>Accumulates a univariate time series to a particular frequency</td>
</tr>
<tr>
<td>ACCUMULATE2</td>
<td>Accumulates a high-frequency time series to a lower frequency and expands the lower-frequency series to have the same length as the high-frequency series</td>
</tr>
<tr>
<td>ACF</td>
<td>Computes autocorrelation and autocovariance for a time series array</td>
</tr>
<tr>
<td>ARMAORDERS</td>
<td>Performs tests to tentatively identify the autoregressive and moving average orders of mixed autoregressive moving average models</td>
</tr>
<tr>
<td>CCF</td>
<td>Computes the cross-correlation and cross-covariance for two time series arrays</td>
</tr>
<tr>
<td>DPF Class Object</td>
<td>Performs count distribution analysis for time series</td>
</tr>
<tr>
<td>FREQ Class Object</td>
<td>Performs frequency analysis of a time series</td>
</tr>
<tr>
<td>INTERMITTENCYTEST</td>
<td>Tests for intermittency of a univariate time series</td>
</tr>
<tr>
<td>IACF</td>
<td>Computes the inverse autocorrelation for a time series array</td>
</tr>
<tr>
<td>MOVINGSUMMARY</td>
<td>Computes statistics for a set of values within a moving time window</td>
</tr>
<tr>
<td>PACF</td>
<td>Computes the partial autocorrelation for a time series array</td>
</tr>
<tr>
<td>SCALE</td>
<td>Scales a time series between the minimum value and the maximum value of the original time series</td>
</tr>
<tr>
<td>SEASONALDECOMP</td>
<td>Computes the seasonal indices of a univariate time series using classical decomposition</td>
</tr>
<tr>
<td>SEASONALINDICES</td>
<td>Computes the seasonal indices of a univariate time series by using regression seasonal dummies</td>
</tr>
<tr>
<td>SEASONTEST</td>
<td>Tests for seasonality of a univariate time series</td>
</tr>
<tr>
<td>SIMILARITY</td>
<td>Performs similarity analysis for time series</td>
</tr>
<tr>
<td>STATIONARITYTEST</td>
<td>Tests for stationarity of a univariate time series</td>
</tr>
<tr>
<td>TRANSFORM</td>
<td>Transforms time series according to the specified transformation type</td>
</tr>
</tbody>
</table>
### ACCUMULATE Function

\[ rc = \text{TSA.ACCUMULATE} \left( \text{time}, \ y, \ 'interval', \ id, \ z, \ '<accumulate>', \ '<setmiss>', \ '<zeromiss>' \right) ; \]

The ACCUMULATE function accumulates a univariate time series to a particular frequency.

#### Required Arguments
You must specify the following arguments, separated by commas:

- **time** specifies the time ID array for the time series.
- **y** specifies the times series array to accumulate.
- **'interval'** specifies the time interval.
  
  You can specify the following values within single quotation marks:
  
  - **DAY** specifies a seasonal cycle of length 7.
  - **MONTH** specifies a seasonal cycle of length 12.
  - **QTR** specifies a seasonal cycle of length 4.

#### Optional Arguments
You can also specify the following arguments, separated by commas. If you want to use a default value for any of these arguments, enter a space for it.

- **'accumulate'** specifies the accumulation statistic.
  
  You can specify the following values within single quotation marks:
  
  - **AVERAGE | AVG** specifies the average of the values in the time series.
  - **CSS** specifies the corrected sum of squares of the values in the time series.
  - **FIRST** specifies the first value of the time series.
  - **LAST** specifies the last value of the time series.
  - **MAXIMUM | MAX** specifies the maximum value in the time series.

---

**Table 4.1  continued**

<table>
<thead>
<tr>
<th>TSA Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>UNBIASEDNESS</td>
<td>Tests whether a univariate time series is unbiased</td>
</tr>
<tr>
<td>WHITENOISE</td>
<td>Tests for white noise of a time series array</td>
</tr>
</tbody>
</table>
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:

<table>
<thead>
<tr>
<th>rc</th>
<th>Termination Reason</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Success</td>
</tr>
<tr>
<td>&lt; 0</td>
<td>Computational failure</td>
</tr>
</tbody>
</table>

id returns the time ID array for the accumulated time series.

z returns the accumulated time series array.

---

**Example**

This example uses the TSMODEL procedure to accumulate a time series:

```plaintext
proc tsmodel data=mycas.air outarray=mycas.outarray;
  id date interval=month;
  var air;
  outarray qtravg new_id;
  require tsa;
  submit;
  declare object TSA(tsa);
  rc=TSA.ACCUMULATE(date, air, 'QTR', new_id, qtravg, 'AVERAGE', , );
  endsubmit;
run;
```

---

**ACCUMULATE2 Function**

\[ rc = TSA.ACCUMULATE2 \left( \text{time}, \text{y}, \text{interval'}, \text{id, z,<'accumulate'>, <'setmiss'>, <'zeromiss'>} \right) ; \]

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.
Chapter 4: Time Series Analysis Package for the TSMODEL Procedure

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:
ACF Function

**Returned Values**

The ACCUMULATE2 function returns the following values:

- **rc**
  - returns one of the following scalar return codes:
    - 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:

```r
proc tsmodel data=mycas.air outarray=mycas.outarray;
    id date interval=month;
    var air;
    outarray 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**

```r
rc = TSA.ACF (y, nlag, lags, df, < mu>, < acov>, < acf>, < acfstd>, < acl2std>, < acfnorm>, < acfprob>
              < aclfprob>);
```

The ACF function computes autocorrelation and autocovariance for a time series array.
Required Arguments

You must specify the following arguments, separated by a comma:

- \( y \) specifies the times series array.
- \( nlag \) specifies the number of the lag to use in the calculation.

Returned Values

The ACF function returns the following values:

- \( rc \) returns one of the following scalar return codes:
  
<table>
<thead>
<tr>
<th>( rc )</th>
<th>Termination Reason</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Success</td>
</tr>
<tr>
<td>&lt; 0</td>
<td>Computational failure</td>
</tr>
</tbody>
</table>

- \( lags \) returns the number of the lags that were used in the calculation.
- \( df \) returns the number of observations used to compute \( acov \) and \( acf \).

Optional Returned Values

You can also specify the following arguments, separated by commas to request additional returned values. If you do not want the value to be returned, enter a space for it.

- \( mu \) returns the mean estimate.
- \( acov \) returns an array of covariance estimates, with \( nlag+1 \) entries.
- \( acf \) returns an array of autocorrelation estimates, with \( nlag+1 \) entries.
- \( acfstd \) returns an array of standard errors, with \( nlag+1 \) entries.
- \( acf2std \) returns an array of twice standard errors, with \( nlag+1 \) entries.
- \( acfnorm \) returns an array of normalized autocorrelation, with \( nlag+1 \) entries.
- \( acfprob \) returns an array of autocorrelation probabilities, with \( nlag+1 \) entries.
- \( acflprob \) 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 \( \text{Air} \):
proc tsmode data=mycas.air outscalars=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 \( y, < \text{dif}>, <'\text{method'}>, <\text{p}>, <\text{q}>, <\text{perror}>, \text{porders, qorders} > \); \]

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:

\[ y \quad \text{specifies the times series array to test.} \]

Optional Arguments

You can also specify the following arguments, separated by commas. If you want to use a default value for any of these arguments, enter a space for it.

\[ \text{dif} \quad \text{specifies either an array of positive integers or a positive integer that is used for differencing. The default value is 0.} \]

\[ '\text{method'} \quad \text{specifies the method of tentative order selection.} \]

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.
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:
  
<table>
<thead>
<tr>
<th>( rc )</th>
<th>Termination Reason</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Success</td>
</tr>
<tr>
<td>&lt; 0</td>
<td>Computational failure</td>
</tr>
</tbody>
</table>

- \( porders \) returns the recommended autoregressive orders.

- \( qorders \) returns the recommended moving average orders.

## Example

This example uses the TSMODEL procedure to tentatively identify the autoregressive and moving average orders for the time series Air:

```plaintext
proc tsmode data=mycas.air outscalars=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

\[ rc = \text{TSA.CCF}(y, x, nlag, lags, df, < ymu >, < xmu >, < ccov >, < ccf >, < ccfstd >, < ccfnorm >, < ccfprob >, < ccf1prob >); \]

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:
  
  \[
  \begin{array}{c|c}
  rc & \text{Termination Reason} \\
  \hline
  0 & \text{Success} \\
  < 0 & \text{Computational failure} \\
  \end{array}
  \]

- \( lags \) 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.

cfnorm returns an array of normalized cross-correlation, with $2 \times nlag + 1$ entries.

cflprob returns an array of probabilities, with $2 \times nlag + 1$ entries.

cfllprob 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:

```plaintext
proc tsmodel data=mycas.pricedata outarray=mycas.ccf_array
   outscalar=mycas.ccf_scalar;
   id date interval=month;
   var price sale;
   by region line product;
   outscalars ymu xmu;
   outarrays lags df ccov ccf ccf2std ccfnorm ccfprob ccfllprob;
   require tsa;
   submit;
   declare object TSA(tsa);
   rc=TSA.CCF(price, sale, 20, lags, df, ymu, xmu, ccov, ccf,
      ccf2std, ccfnorm, ccfprob, ccfllprob);
   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

```plaintext
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:

<table>
<thead>
<tr>
<th>Value</th>
<th>Termination Reason</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Successful</td>
</tr>
<tr>
<td>&lt; 0</td>
<td>Computational failure</td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Column Content</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>NAME</em></td>
<td>Name of target variable</td>
</tr>
<tr>
<td>Distribution</td>
<td>Distribution</td>
</tr>
<tr>
<td>Parameter</td>
<td>Name of parameter</td>
</tr>
<tr>
<td>Estimate</td>
<td>Parameter estimate</td>
</tr>
<tr>
<td>StdErr</td>
<td>Standard error</td>
</tr>
<tr>
<td>Tvalue</td>
<td>( t ) value</td>
</tr>
<tr>
<td>Probt</td>
<td>Approximate probability &gt;</td>
</tr>
<tr>
<td>Lower</td>
<td>Lower 95% of parameter estimate</td>
</tr>
<tr>
<td>Upper</td>
<td>Upper 95% of parameter estimate</td>
</tr>
</tbody>
</table>

OUTDPROB collects predictions using the count distribution that has the best fit. The output table contains the following columns:
<table>
<thead>
<tr>
<th>Column Name</th>
<th>Column Content</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>NAME</em></td>
<td>Name of target variable</td>
</tr>
<tr>
<td>Index</td>
<td>Index</td>
</tr>
<tr>
<td>Value</td>
<td>Values</td>
</tr>
<tr>
<td>Observedzeros</td>
<td>Observed zeros</td>
</tr>
<tr>
<td>Expectedzeros</td>
<td>Expected zeros</td>
</tr>
<tr>
<td>Observed</td>
<td>Observed counts</td>
</tr>
<tr>
<td>Expected</td>
<td>Expected counts</td>
</tr>
<tr>
<td>Expectedlower</td>
<td>Expected lower confidence</td>
</tr>
<tr>
<td>Expectedupper</td>
<td>Expected upper confidence</td>
</tr>
<tr>
<td>Probability</td>
<td>Probability</td>
</tr>
<tr>
<td>Probabilitylower</td>
<td>Probability lower confidence</td>
</tr>
<tr>
<td>Probabilityupper</td>
<td>Probability upper confidence</td>
</tr>
<tr>
<td>Chisquare</td>
<td>Chi-square statistic</td>
</tr>
<tr>
<td>Chisquareprob</td>
<td>Chi-square probability</td>
</tr>
<tr>
<td>Chisquarelogprob</td>
<td>Chi-square log probability</td>
</tr>
</tbody>
</table>

**Example**

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

```plaintext
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

```cpp
declare object f(FREQ);
declare object of(OUTFREQ);
rc = f.Initialize();
rc = f.SetY(claims);
rc = f.SetOption(<'name',value, 'name', value, ...>);
rc = f.Run();
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:

\[ \text{seasonality} \]

specifies the seasonality of the time series, where seasonality is an integer. The default value is 0.

Returned Values

The Run() statement returns the following values:

\[ rc \]

returns one of the following scalar return codes:

<table>
<thead>
<tr>
<th>rc</th>
<th>Termination Reason</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Successful</td>
</tr>
<tr>
<td>1</td>
<td>Binary series</td>
</tr>
<tr>
<td>2</td>
<td>Nonnegative integer series</td>
</tr>
<tr>
<td>3</td>
<td>Integer series</td>
</tr>
<tr>
<td>4</td>
<td>Noninteger series</td>
</tr>
<tr>
<td>&lt; 0</td>
<td>Computational failure</td>
</tr>
</tbody>
</table>
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:

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Column Content</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>NAME</em></td>
<td>Name of target variable</td>
</tr>
<tr>
<td>Values</td>
<td>Array of series values</td>
</tr>
<tr>
<td>Counts</td>
<td>Array of frequency series counts</td>
</tr>
<tr>
<td>Percent</td>
<td>Percentage of total frequency</td>
</tr>
</tbody>
</table>

Example

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

```plaintext
proc tsmodel data=mycas.air outobj=(of=mycas.outfreq(replace=YES));
   var air;
   id date interval=month;
   require tsa;
   submit;
     declare object f(FREQ);
     declare object of(OUTFREQ);
     rc = f.Initialize();
     rc = f.SetOption("SEASONALITY",12);
     rc = f.SetY(air);
     rc = f.Run();
     rc = of.Collect(f);
  endsubmit;
run;
```

IACF Function

\[ rc = \text{TSA.IACF}\left(y, nlag, lags, df, \langle \mu \rangle, \langle \text{iacf} \rangle, \langle \text{iacfstd} \rangle, \langle \text{iacf2std} \rangle, \langle \text{iacfnorm} \rangle, \langle \text{iactprob} \rangle, \langle \text{iactfprob} \rangle\right); \]

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:
  - \( 0 \): 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.

- \( iacf2norm \): returns an array of normalized inverse autocorrelation, with \( nlag+1 \) entries.

- \( iactprob \): returns an array of inverse autocorrelation probabilities, with \( nlag+1 \) entries.

- \( iacfprob \): 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 \( \text{Air} \):

```plaintext
proc tsmode l data=mycas.air outscalars=mycas.outscalars
  outarray=mycas.outarray;
  id date interval=month;
  var air;
  outscalars mu;
  outarrays iacf lags df iacfstd;
  require tsa;
```
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 outscalars=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

\[ rc = \text{TSA.MOVINGSUMMARY}(y, \text{'method'}, k, <\text{lead}>, <\text{w}>, <\text{setmiss}>, <\text{abs}>, x, <p>, <nmiss>) \]

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.
- \( \text{'method'} \) 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 mean divided by the standard deviation.
- **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, TV ALUE, 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:
returns one of the following scalar return codes:

<table>
<thead>
<tr>
<th>rc</th>
<th>Termination Reason</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>Warping limits relaxed</td>
</tr>
<tr>
<td>3</td>
<td>Expansion limits relaxed</td>
</tr>
<tr>
<td>2</td>
<td>Compression limits relaxed</td>
</tr>
<tr>
<td>1</td>
<td>Warping limits imposed</td>
</tr>
<tr>
<td>0</td>
<td>Success</td>
</tr>
<tr>
<td>&lt; 0</td>
<td>Computational failure</td>
</tr>
</tbody>
</table>

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.

\( n\text{miss} \) 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 \( \text{Air} \):

```plaintext
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 \( \text{Air} \):

```plaintext
proc tsmodel data=mycas.air outscalar = mycas.scalars outarray=mycas.arrays;
  id date interval=month;
  var air;
  Array w[5]/nosymbols; w[1]=0.3; w[2]=0.2; w[3]=0.25; w[4]=0.1; w[5]=0.15;
  outarrays x;
  outscalars rc;
```
PACF Function

```plaintext
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:
  
<table>
<thead>
<tr>
<th>rc</th>
<th>Termination Reason</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Success</td>
</tr>
<tr>
<td>&lt; 0</td>
<td>Computational failure</td>
</tr>
</tbody>
</table>

- **lags** returns the number of the lag that was used in the calculation.
- **df** returns the number of observations used to compute `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.
**Example**

This example uses the TSMODEL procedure to compute the autocorrelation of lag 3 of the time series Air:

```plaintext
proc tsmodel data=mycas.air outscalars=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**

```plaintext
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:

<table>
<thead>
<tr>
<th>( rc )</th>
<th>Termination Reason</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>The input series is nearly constant</td>
</tr>
<tr>
<td>4</td>
<td>Missing values were found when the value 1 was specified for ( nomiss )</td>
</tr>
<tr>
<td>3</td>
<td>One or more arguments are ignored</td>
</tr>
<tr>
<td>2</td>
<td>One or more arguments are set to the default value</td>
</tr>
<tr>
<td>1</td>
<td>The input series is all missing</td>
</tr>
<tr>
<td>-1</td>
<td>One or more arguments are not supported</td>
</tr>
<tr>
<td>-2</td>
<td>The minimum value of the transformed series is greater than its maximum value</td>
</tr>
<tr>
<td>-4</td>
<td>Extreme slope</td>
</tr>
<tr>
<td>-99</td>
<td>Bad arguments</td>
</tr>
</tbody>
</table>

\( x \) returns the transformed series.

Optional Returned Values

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

\( nmiss \) 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:

```plaintext
proc tsmodel data=mycas.air outarray=mycas.trans_array;
  var air;
  require tsa;
  submit;
  declare object TSA(tsa);
  outarrays t1;
  rc=TSA.SCALE(air, 0, 100, , t1, );
  endsubmit;
run;
```
SEASONALDECOMP Function

\[ rc = \text{TSA.SEASONALDECOMP} (y, s, 'mode', \langle \text{lambda} \rangle, \langle \text{tcc} \rangle, \langle \text{sic} \rangle, \langle \text{sc} \rangle, \langle \text{scstd} \rangle, \langle \text{tcs} \rangle, \langle \text{ic} \rangle, \langle \text{sa} \rangle, \langle \text{pcsa} \rangle, \langle \text{tc} \rangle, \langle \text{cc} \rangle) ; \]

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.

- \( \text{lambda} \) 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:
returns one of the following scalar return codes:

<table>
<thead>
<tr>
<th>rc</th>
<th>Termination Reason</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Success</td>
</tr>
<tr>
<td>&lt; 0</td>
<td>Computational failure</td>
</tr>
</tbody>
</table>

### Optional Returned Values

You can also specify the following arguments, separated by commas to request additional returned values. If you do not want the value to be returned, enter a space for it.

- `tcc` specifies the trend-cycle component.
- `sic` specifies the seasonal-irregular component.
- `sc` specifies the seasonal component.
- `scstd` specifies the seasonal component standard errors.
- `tcs` specifies the trend-cycle-seasonal component.
- `ic` specifies the irregular component.
- `sa` specifies the seasonally adjusted series.
- `psa` specifies the percentage of change in seasonally adjusted series.
- `tc` specifies the trend component.
- `cc` specifies the cycle component.

### Example

This example uses the TSMODEL procedure to compute the seasonal indices on the time series array `Air`:

```plaintext
proc tsmodel data=mycas.air outarray=mycas.outarray;
   id date interval=month;
   var air;
   outarray ADJUSTED;
   require tsa;
   submit;
   declare object TSA(tsa);
   rc=TSA.SEASONALDECOMP(air, _SEASONALITY_, 'ADD', , , , , , , , ADJUSTED, , , );
   endsubmit;
run;
```

### SEASONALINDICES Function

```plaintext
rc = TSA.SEASONALINDICES (y, s, <'mode'>, <'term'>, indices);
```

The SEASONALINDICES function computes the seasonal indices of a univariate time series by using regression seasonal dummies.
Required Arguments

You must specify the following arguments, separated by a comma:

- \( y \): specifies the times series array.
- \( s \): specifies the seasonality to test, where \( s \) must be either a positive integer or \_SEASONALITY_, which is the length of the seasonal cycle as specified by the SEASONALITY= option in the PROC TSMODEL statement or implied by the INTERVAL= option in the ID statement.

Optional Arguments

You can also specify the following arguments, separated by commas. If you want to use a default value for any of these arguments, enter a space for it.

- \('mode'\): specifies the type of model to be used in the regression. You can specify the following values within single quotation marks:
  - ADD | ADDITIVE: uses an additive model.
  - MULT | MULTIPLICATIVE: uses a multiplicative model.

  The default method is ADD.

- \('term'\): specifies the type of terms to be used in the regression. You can specify the following values within single quotation marks:
  - S: uses only seasonal dummies terms.
  - SC: uses only seasonal dummies and constant terms.
  - ST: uses only seasonal dummies and trend terms.
  - STC: uses seasonal dummies, trend, and constant terms.
  - STQ: uses seasonal dummies, trend, and quadratic terms.
  - STQC: uses seasonal dummies, trend, quadratic, and constant terms.

  The default value is S. Quadratic values can be used only in the additive model.

Returned Values

The SEASONALINDICES function returns the following values:

- \( rc \): returns one of the following scalar return codes:
### SEASONTEST Function

<table>
<thead>
<tr>
<th>rc</th>
<th>Termination Reason</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Success</td>
</tr>
<tr>
<td>&lt; 0</td>
<td>Computational failure</td>
</tr>
</tbody>
</table>

(indices) returns an array of seasonal indices.

---

### Example

This example uses the TSMODEL procedure to compute the seasonal indices of the time series Air:

```plaintext
proc tsmode 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.
Chapter 4: Time Series Analysis Package for the TSMODEL Procedure

- **dif** specifies an array of positive integers or a positive integer that is used for differencing. The default value is 0.
- **p** specifies the autoregressive order (0 or 1). The default value is 0.
- **alpha** specifies the significance level. The default value is 0.01.

### Returned Values

The SEASONTEST function returns the following values:

- **rc** returns one of the following scalar return codes:

<table>
<thead>
<tr>
<th>rc</th>
<th>Termination Reason</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Time series is not seasonal</td>
</tr>
<tr>
<td>1</td>
<td>Time series is seasonal</td>
</tr>
<tr>
<td>&lt; 0</td>
<td>Computational failure</td>
</tr>
</tbody>
</table>

### Optional Returned Values

You can also specify the following arguments, separated by commas to request additional returned values. If you do not want the value to be returned, enter a space for it.

- **aic** returns an array of three values: Akaike’s information criterion (AIC) for the nonseasonal model, AIC for seasonal model, and the p-value for the F test.

### Example

The following example uses the TSMODEL procedure to test the seasonality of the time series array Air:

```plaintext
proc tsmodel data=mycas.air outscalars=mycas.outscalars
  outarray=mycas.outarray;
    id date interval=month;
    var air;
    outscalars seasonal;
    outarray 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:

<table>
<thead>
<tr>
<th>rc</th>
<th>Termination Reason</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>Warping limits relaxed</td>
</tr>
<tr>
<td>3</td>
<td>Expansion limits relaxed</td>
</tr>
<tr>
<td>2</td>
<td>Compression limits relaxed</td>
</tr>
<tr>
<td>1</td>
<td>Warping limits imposed</td>
</tr>
<tr>
<td>0</td>
<td>Success</td>
</tr>
<tr>
<td>&lt; 0</td>
<td>Computational failure</td>
</tr>
</tbody>
</table>

- **measure** returns the similarity measure.
Example

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

```sas
data test;
    input i y x;
datalines;
1 2 3
2 4 5
3 6 3
4 7 3
5 3 3
6 8 6
7 9 3
8 3 8
9 10 .
10 11 .
;

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 outscalars=mycas.tsa_scalar2;
    require tsa;
    submit;
    declare object TSA(tsa);
    var x y;
    outscalar measure;
    rc=TSA.SIMILARITY(x, y, 'ABSDEV', 'NONE', 20, 0 , , , measure);
endsubmit;
run;
```
STATIONARITYTEST Function

\[ rc = \text{TSA.STATIONARITYTEST} \left( y, < \text{dif} >, < d >, < p >, < 'type' >, pvalue \right); \]

The STATIONARITYTEST function tests for stationarity of a univariate time series.

Required Arguments

You must specify the following argument:

- \( y \) specifies the times series array to test.

Optional Arguments

You can also specify the following arguments, separated by commas. If you want to use a default value for any of these arguments, enter a space for it.

- \( \text{dif} \) specifies an array of positive integers or a positive integer that is used for differencing. The default value is 0.
- \( d \) specifies the order of unit root (\( d = 1, \ldots, 12 \)). If the \( \text{type} \) is SSM, then \( d = 1 \). The default value is 1.
- \( p \) 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:
  
  - \( \text{SSM} \) specifies the studentized test statistic for the single mean (intercept) case.
  - \( \text{STR} \) specifies the studentized test statistic for the deterministic time trend case.
  - \( \text{SZM} \) specifies the studentized test statistic for the zero mean (no intercept) case. This value is allowed only when \( d=1 \).

  The default value of \( \text{type} \) is SZM.

Returned Values

The STATIONARITYTEST function returns the following values:

- \( rc \) returns one of the following scalar return codes:
### TRANSFORM Function

<table>
<thead>
<tr>
<th>rc</th>
<th>Termination Reason</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Time series is stationary with the default significance level of 0.05</td>
</tr>
<tr>
<td>1</td>
<td>Time series is not stationary with the default significance level of 0.05</td>
</tr>
<tr>
<td>&lt; 0</td>
<td>Computational failure</td>
</tr>
</tbody>
</table>

\[ pvalue \] returns the probability value associated with the test.

### Example

This example uses the TSMODEL procedure to test the stationarity on the time series array Air:

```plaintext
proc tsmode data=mycas.air outscalars=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

\[ rc \text{ = TSA.TRANSFORM (} y, \langle \text{'type'} \rangle, \langle \text{inverse} \rangle, \langle c \rangle, x) ; \]

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: $x = \log(y + c)$. The default value is 0.
- For logit transformation, $c$ is scaling: $x = \log(c \times y / (1 - (c \times y)))$. The default value is 1.
- For the Box-Cox transformation, $c$ is $\lambda$: $x = c^2 + (y^c - 1)/c$. If $c$ is not specified, $x=y$.

Returned Values

The TRANSFORM function returns the following values:

$rc$ returns one of the following scalar return codes:

<table>
<thead>
<tr>
<th>$rc$</th>
<th>Termination Reason</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Success</td>
</tr>
<tr>
<td>&lt; 0</td>
<td>Computational failure</td>
</tr>
</tbody>
</table>

$x$ returns the transformed series.
Example

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

```r
proc tsmodel data=mycas.air outarray=mycas.trans_array;
  var air;
  require tsa;
  submit;
  declare object TSA(tsa);
  outarrays t1;
  rc=TSA.TRANSFORM(air, 'LOG', 0, 0, t1);
  endsubmit;
run;
```

**UNBIASEDNESS Function**

\[ rc = \text{TSA.UNBIASEDNESS}(y, \text{predict}, \text{<siglevel>}, \text{intercept}, \text{scale}, \text{fvalue}, \text{pvalue}); \]

The UNBIASEDNESS function tests whether a univariate time series is unbiased.

**Required Arguments**

You must specify the following arguments, separated by a comma:

- \textit{y} specifies the input time series array.
- \textit{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.

- \textit{siglevel} specifies the significance level.

**Returned Values**

The UNBIASEDNESS function returns the following values:

- \textit{rc} returns one of the following scalar return codes:
### rc Termination Reason

<table>
<thead>
<tr>
<th>rc</th>
<th>Termination Reason</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Biased predictions</td>
</tr>
<tr>
<td>0</td>
<td>Unbiased predictions</td>
</tr>
<tr>
<td>-1</td>
<td>Degree of freedom error</td>
</tr>
<tr>
<td>-2</td>
<td>Singular system</td>
</tr>
<tr>
<td>-3</td>
<td>Extreme value</td>
</tr>
</tbody>
</table>

*intercept* returns the constant parameter.

*scale* returns the scale parameter.

*fvalue* returns the test statistic for the \( F \) test.

*pvalue* returns the \( p \)-value for the \( F \) test.

---

**Example**

This example uses the TSMODEL procedure to test whether the series Actual is unbiased:

```sas
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 outscalars=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

\[ rc = \text{TSA.WHITENOISE}(y, nlag, lags, df, wn, <\text{wnprob}>, <\text{wnlprob}>) ; \]

The WHITENOISE function tests for white noise in a time series array.

**Required Arguments**

You must specify the following arguments, separated by a comma:

- \( y \) specifies the times series array to compute.
- \( nlag \) specifies the number of the lag to use in the calculation.

**Returned Values**

The WHITENOISE function returns the following values:

- \( rc \) returns one of the following scalar return codes:

<table>
<thead>
<tr>
<th>\text{rc}</th>
<th>\text{Termination Reason}</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Success</td>
</tr>
<tr>
<td>&lt; 0</td>
<td>Computational failure</td>
</tr>
</tbody>
</table>

- \( lags \) returns the number of the lag that was used in the calculation.
- \( df \) returns the number of observations that were used to test white noise.
- \( wn \) returns an array of Ljung-Box white noise tests, with \( nlag + 1 \) entries.

**Optional Returned Values**

You can also specify the following arguments, separated by commas to request additional returned values. If you do not want the value to be returned, enter a space for it.

- \( \text{wnprob} \) returns white noise probabilities.
- \( \text{wnlprob} \) returns white noise log probabilities.
Example

This example uses the TSMODEL procedure to perform the white noise test of lag 3 of the time series Air:

```plaintext
proc 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 5

**Time Series Model Package for the TSMODEL Procedure**

## Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Overview</td>
<td>146</td>
</tr>
<tr>
<td>TSM Package Summary</td>
<td>147</td>
</tr>
<tr>
<td>Using the TSM Package</td>
<td>149</td>
</tr>
<tr>
<td>TSM Object Summary</td>
<td>151</td>
</tr>
<tr>
<td>TSM Synopsis</td>
<td>152</td>
</tr>
<tr>
<td>TSM Methods</td>
<td>152</td>
</tr>
<tr>
<td>ESM Spec Object Summary</td>
<td>156</td>
</tr>
<tr>
<td>ESM Spec Synopsis</td>
<td>157</td>
</tr>
<tr>
<td>ESM Spec Methods</td>
<td>157</td>
</tr>
<tr>
<td>ARIMA Spec Object Summary</td>
<td>160</td>
</tr>
<tr>
<td>ARIMA Spec Synopsis</td>
<td>163</td>
</tr>
<tr>
<td>ARIMA Spec Methods</td>
<td>163</td>
</tr>
<tr>
<td>TSMPEst Object Summary</td>
<td>169</td>
</tr>
<tr>
<td>TSMPEst Synopsis</td>
<td>169</td>
</tr>
<tr>
<td>TSMPEst Methods</td>
<td>170</td>
</tr>
<tr>
<td>TS MSpec Object Summary</td>
<td>170</td>
</tr>
<tr>
<td>TS MSpec Synopsis</td>
<td>171</td>
</tr>
<tr>
<td>TS MSpec Methods</td>
<td>171</td>
</tr>
<tr>
<td>TSM For Object Summary</td>
<td>171</td>
</tr>
<tr>
<td>TSM For Synopsis</td>
<td>172</td>
</tr>
<tr>
<td>TSM For Methods</td>
<td>173</td>
</tr>
<tr>
<td>TSMINEst Object Summary</td>
<td>174</td>
</tr>
<tr>
<td>TSMINEst Synopsis</td>
<td>174</td>
</tr>
<tr>
<td>TSMINEst Methods</td>
<td>174</td>
</tr>
<tr>
<td>TSMINSpec Object Summary</td>
<td>174</td>
</tr>
<tr>
<td>TSMINSpec Synopsis</td>
<td>175</td>
</tr>
<tr>
<td>TSMINSpec Methods</td>
<td>175</td>
</tr>
<tr>
<td>Examples: TSM Package</td>
<td>175</td>
</tr>
<tr>
<td>Example 5.1: Fitting and Forecasting with ARIMA and ESM Models</td>
<td>175</td>
</tr>
<tr>
<td>Example 5.2: Fitting a Transfer Function Model</td>
<td>179</td>
</tr>
<tr>
<td>Example 5.3: Replaying a Previously Fitted Model</td>
<td>180</td>
</tr>
<tr>
<td>References</td>
<td>183</td>
</tr>
</tbody>
</table>
Overview

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*. This document describes the following objects, which are contained in the TSM package and designed to provide access to various univariate time series model families:

- 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.

- The following time series model specification objects enable you to specify the characteristics of a time series model to be executed by a TSM object:
  - ESMSpec (exponential smoothing model specification)
  - ARIMASpec (autoregressive integrated moving average specification)

- The following 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):
  - TSMPEst (time series model parameter estimates collector)
  - TSMSpec (time series model specification collector)
  - TSMFor (time series model forecast collector)

- The following time series model repeater objects act as conduits to replay collected model specifications and model parameter estimates as input to other TSM objects:
  - TSMINEst (time series model input parameter estimates repeater)
  - TSMINSpec (time series model input specification repeater)
TSM Package Summary

All of the objects in the TSM package are summarized in Table 5.1.

<table>
<thead>
<tr>
<th>TSM Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TSM</td>
<td>Time series model object</td>
</tr>
<tr>
<td>ESMSpec</td>
<td>Exponential smoothing model specification object</td>
</tr>
<tr>
<td>ARIMASpec</td>
<td>Autoregressive integrated moving average model specification object</td>
</tr>
<tr>
<td>TSMPEst</td>
<td>Time series model parameter estimates collector object</td>
</tr>
<tr>
<td>TSMSpec</td>
<td>Time series model specification collector object</td>
</tr>
<tr>
<td>TSMFor</td>
<td>Time series model forecast collector object</td>
</tr>
<tr>
<td>TSMINEst</td>
<td>Time series model input parameter estimates repeater object</td>
</tr>
<tr>
<td>TSMINSpec</td>
<td>Time series model input specification repeater object</td>
</tr>
</tbody>
</table>

The data flow between the TSM objects is illustrated in Figure 5.1.
Chapter 5: Time Series Model Package for the TSMODEL Procedure

Figure 5.1 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 a 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 are used to 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 model specification object-specific 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 AR or MA 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 on this use, see Chapter 1, “Automatic Time Series Analysis and Forecasting Package for the TSMODEL Procedure.”

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 (Y) series and any independent (X) series 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 completed 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 as desired.

**Step 3: Use the TSM Collector Objects**

Collector objects enable you to create a snapshot of results from TSM objects and store them to 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 method 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 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 corresponding BY variable values. 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 restore rows from a CAS table to make them available for use 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 the repeater object’s design. Repeater objects must be bound to 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 and CAS table passed as arguments to execute a new time series model. The data can then be collected by using a collector object as described in step 3.
Status Codes

Status return codes (designated rc in method usage statements) are numeric values that are returned when a method associated with an object is called; they are defined in Table 5.2. These codes can help determine whether the method executed successfully.

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

TSM Object Summary

The time series model (TSM) object generates forecasts of univariate time series. Table 5.3 lists the time series model families that are supported.

<table>
<thead>
<tr>
<th>Family</th>
<th>Specification Object</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ESM</td>
<td>ESMSpec</td>
<td>Exponential smoothing models</td>
</tr>
<tr>
<td>ARIMA</td>
<td>ARIMASpec</td>
<td>ARIMAX models</td>
</tr>
</tbody>
</table>

The basic execution pattern for using a TSM object follows this sequence of operations:

1 **Declare:** The object declaration statement creates a new TSM object.

2 **Initialize:** The TSM.Initialize method takes a model specification object as its argument and initializes the TSM object for that specified time series model. If no model specification object is provided, the TSM object is initialized by the default ESM specification.

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 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.
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**

The object declaration statement creates a new object, `obj`, of type `TSM`.

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

Upon declaration, the TSM objects have a default model—ESM/method=BEST. Table 5.4 summarizes the methods and attributes that are associated with the TSM Object.

<table>
<thead>
<tr>
<th>TSM Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initialize</td>
<td>Initialize the TSM object.</td>
</tr>
<tr>
<td>SetY</td>
<td>Specify the dependent time series array (Y) for the TSM object.</td>
</tr>
<tr>
<td>AddX</td>
<td>Add an independent time series array (XSeries) for the TSM object.</td>
</tr>
<tr>
<td>SetOption</td>
<td>Specify the named option for the TSM object</td>
</tr>
<tr>
<td>Run</td>
<td>Run the TSM object.</td>
</tr>
<tr>
<td>Replay</td>
<td>Replay the restored model and parameter estimates.</td>
</tr>
<tr>
<td>GetForecast</td>
<td>Get the forecast series.</td>
</tr>
<tr>
<td>NFor</td>
<td>Return the forecast series length.</td>
</tr>
<tr>
<td>Criterion</td>
<td>Create the final forecast fit statistic.</td>
</tr>
</tbody>
</table>

**TSM Methods**

This section describes the methods of a TSM object.

**TSM.Initialize Method**

The Initialize method initializes a TSM object to use the specified `ModelSpec`. This method must be called before specifying the time series arrays (X and Y series) and other attributes for the TSM object. If no `ModelSpec` object is specified, the TSM object is initialized to use the default ESM specification.

```plaintext
Usage: rc = obj.Initialize ( <ModelSpec> ) ;
```
Parameters
[in] ModelSpec Optional name of a TSM model specification object that is used to configure the TSM object.

TSM.SetY Method
The SetY method specifies the dependent time series array, YSeries, for the TSM object.
Usage:
rc = obj.SetY ( YSeries );

Parameters
[in] YSeries Numeric array that contains the dependent series for the TSM object.

TSM.AddX Method
The AddX method adds an independent time series array, XSeries, for the TSM object. Each call to the AddX method adds the specified X 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.
Usage:
rc = obj.AddX ( XSeries < , Required, NoDiff, ModelSymbol > );

Parameters
[in] XSeries Numeric array that specifies an independent series for the TSM object.
[in] Required Optional binary variable (0 or 1). A value of 1 specifies that the X-variable is required to be in the model. This might cause the model estimation to fail if it is deemed inadmissible for inclusion into the underlying ARIMA model. The default value is 0.
[in] NoDiff Optional numeric variable (0 or 1). A value of 1 specifies that the X-variable does not automatically follow the Y-variable differencing. The default value is 0.
[in] ModelSymbol Optional character variable that specifies the name of an input symbol in the model specification to be bound to the X variable when the model is run. By default, TSM.AddX adds X 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 X variable if the ARIMASpec object includes a symbol that matches the ModelSymbol argument.

TSM.SetOption Method
The SetOption method specifies named options for the TSM object. Name is a character literal or variable. The type of Value depends on the value of Name. Table 5.5 shows the allowed options.
Usage:
rc = obj.SetOption ( 'Name', Value < , 'Name', Value,... > );
Table 5.5  'Name' Arguments for the TSM.SetOption Method

<table>
<thead>
<tr>
<th>'Name'</th>
<th>Value Type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>CRITERION</td>
<td>String</td>
<td>Fit statistic mnemonic. For more information, see the CRITERION= option in the PROC HPFDIAGNOSE statement in the HPFDIAGNOSE chapter in SAS Forecast Server Procedures: User’s Guide</td>
</tr>
<tr>
<td>LEAD</td>
<td>Numeric</td>
<td>Forecast lead (nonnegative integer)</td>
</tr>
<tr>
<td>BACK</td>
<td>Numeric</td>
<td>BACK region (nonnegative integer)</td>
</tr>
<tr>
<td>HOLDOUT</td>
<td>Numeric</td>
<td>HOLDOUT region (nonnegative integer)</td>
</tr>
<tr>
<td>ALPHA</td>
<td>Numeric</td>
<td>CI band significance level (0 \leq \text{Value} \leq 1)</td>
</tr>
</tbody>
</table>

**Parameters**

[in] **Name**  Character variable that names the option to specify.

[in] **Value**  Specifies the value for the named option.

**TSM.Replay Method**

The Replay method enables the restoration of 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.

Usage: \[ rc = \text{obj.Replay( } TSMINSpecObj <, TSMINEstObj>\)); \]

**Parameters**

[in] **TSMINSpecObj**  Object to supply the model specification.

[in] **TSMINEstObj**  Optional object to supply the model’s parameter estimates.

**TSM.Run Method**

The Run method runs the TSM object to estimate and forecast the time series model by using the dependent (Y) and independent (X) series that have been specified for it. Upon successful completion, various results can be extracted from the TSM object.

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

**Parameters**

No parameters are associated with this method.
TSM.GetForecast Method

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

Usage: \( rc = obj.GetForecast ( 'Which', Result) ; \)

**Parameters**

- **Which** [in] Character string whose value specifies the forecast series to return. Supported series are described in Table 5.6.
- **Result** [out] Numeric array to receive the forecast series.

<table>
<thead>
<tr>
<th>Which</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>PREDICT</td>
<td>Forecast series</td>
</tr>
<tr>
<td>ERROR</td>
<td>Residual series</td>
</tr>
<tr>
<td>STDERR</td>
<td>Forecast standard error series</td>
</tr>
<tr>
<td>UPPER</td>
<td>Upper confidence limit series</td>
</tr>
<tr>
<td>LOWER</td>
<td>Lower confidence limit series</td>
</tr>
</tbody>
</table>

TSM Attributes

This section describes TSM methods that return some useful attributes of the TSM object.

TSM.NFor Method

The NFor method returns the length (number of observations) of the forecast series for the TSM object. A missing value indicates that the TSM object has not produced a successful forecast.

Usage: \( nfor = obj.nfor ( ) ; \)

TSM.Criterion Method

The Criterion method returns the fit statistic value for the final forecast for the TSM object. The criterion is specified via TSM.SetOption method. A missing value indicates that the TSM object has not produced a successful forecast.

Usage: \( criterion = obj.criterion ( ) ; \)
ESMSpec Object Summary

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.

You can also store the XML representation of the ESMSpec to a CAS table. For more information, see the TSMSpec object. Data flow through the ESMSpec object is illustrated in Figure 5.2.

**Figure 5.2** ESMSpec Object Data flow
ESMSpec Synopsis

The object declaration statement creates a new object, \( \textit{obj} \), of type \texttt{ESMSpec}.

\texttt{DECLARE OBJECT \textit{obj} (ESMSpec) ;}

Table 5.7 summarizes the methods that are associated with the ESMSpec method.

<table>
<thead>
<tr>
<th>ESMSpec Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Open</td>
<td>Open ESM model specification.</td>
</tr>
<tr>
<td>Close</td>
<td>Close ESM model specification.</td>
</tr>
<tr>
<td>SetTransform</td>
<td>Set transform for ESM model.</td>
</tr>
<tr>
<td>SetParm</td>
<td>Set parameters for ESM model.</td>
</tr>
<tr>
<td>SetOption</td>
<td>Set option for ESM model.</td>
</tr>
</tbody>
</table>

ESMSpec Methods

This section describes the methods of a ESMSpec object.

ESMSpec.Open Method

The Open method initializes the ESMSpec object for a BEST ESM model with no transformation.

Usage: \( rc = \textit{obj}.\texttt{Open} () ; \)

Parameters
There are no parameters associated with this method.

ESMSpec.Close Method

The Close method 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.

Usage: \( rc = \textit{obj}.\texttt{Close} () ; \)

Parameters
There are no parameters associated with this method.
ESMSpec.SetTransform Method

The SetTransform method specifies the functional transform, ‘Type’, to be used by the ESM model. Optional arguments Option and Parm offer greater control over the transform.

Usage: \( rc = \text{obj.SetTransform} ( \text{'Type'}, \text{'Option'}, \text{Parm} ) \);

**Parameters**

- **Type** [in] Character string that specifies the functional transform to use. Table 5.8 describes the transformations available. The default value is NONE.

- **Option** [in] Optional character string that specifies prediction semantics for the inverse transform. Values are 'MEAN' or 'MEDIAN'. The default is 'MEAN'.

- **Parm** [in] Optional numeric variable that specifies a control parameter (a number between –5 and 5) for the functional transform. This parameter is allowed only for Box-Cox transforms.

**Table 5.8** ‘Type’ Arguments for ESMSpec.SetTransform

<table>
<thead>
<tr>
<th>Transform</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NONE</td>
<td>No transformation</td>
</tr>
<tr>
<td>LOG</td>
<td>Log transformation (positive series)</td>
</tr>
<tr>
<td>AUTO</td>
<td>Select between NONE and LOG</td>
</tr>
<tr>
<td>SQRT</td>
<td>Square root transformation (nonnegative series)</td>
</tr>
<tr>
<td>LOGISTIC</td>
<td>Logistic transformation</td>
</tr>
<tr>
<td>BOXCOX</td>
<td>Box-Cox transformation ((-5 \leq \text{Parm} \leq 5))</td>
</tr>
</tbody>
</table>

ESMSpec.SetParm Method

The SetParm method 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 to be accepted.

Usage: \( rc = \text{obj.SetParm} ( \text{'CompName'}, \text{Parm}, \text{LRest}, \text{URest} ) \);
Parameters

[in] **CompName**  Character variable that specifies the ESM component context. Table 5.9 describes the options for 'CompName'.

[in] **Parm**  Numeric variable that specifies smoothing component weight (initial or fixed).

[in] **LRest**  Optional numeric variable that specifies a lower bound restriction on the smoothing weight.

[in] **URest**  Optional numeric variable that specifies an upper bound restriction on the smoothing weight.

Table 5.9  ‘CompName’ Arguments for ESMSpec.SetParm

<table>
<thead>
<tr>
<th>CompName</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Level</td>
<td>Level-state smoothing weight</td>
</tr>
<tr>
<td>Trend</td>
<td>Trend-state smoothing weight</td>
</tr>
<tr>
<td>Damp</td>
<td>Trend-state damping weight</td>
</tr>
<tr>
<td>Season</td>
<td>Seasonal-states smoothing weight</td>
</tr>
</tbody>
</table>

ESMSpec.SetOption Method

The SetOption method specifies ESM options. Options are (’Name’, Value) pairs where ’Name’ is a case-insensitive character string and Value is dependent on the named option.

Usage:  \(rc = obj.SetOption(’Name’, Value < , ’Name’, Value, ... > ) ;\)

Parameters

[in] **Name**  Character variable that specifies the ESM option name. Table 5.10 lists valid option names and corresponding value description. Default values are in parentheses.

[in] **Value**  Variable that specifies the option value.

Table 5.10  ‘Name’ Arguments for ESMSpec.SetOption Method

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>METHOD</td>
<td>String</td>
<td>Smoothing method (SIMPLE)</td>
</tr>
<tr>
<td>CRITERION</td>
<td>String</td>
<td>Fit statistic mnemonic (RMSE)</td>
</tr>
<tr>
<td>NOEST</td>
<td>Numeric</td>
<td>No parameter estimation (0)</td>
</tr>
</tbody>
</table>
Notes:

1. Values for the METHOD option include:
   - SIMPLE
   - LINEAR
   - DAMPTREND
   - WINTERS
   - ADDWINTERS
   - SEASONAL
   - MULTISEASONAL
   - BESTN
   - BESTS
   - BEST

2. See the CRITERION= option in the PROC HPFDIAGNOSE statement in the HPFDIAGNOSE chapter in SAS Forecast Server Procedures: User’s Guide. CRITERION applies only to the ESM uber models BEST, BESTN, and BESTS.

3. NOEST requires that all parameters for the associated ESM method be explicitly specified; otherwise, it is silently ignored.

---

**ARIMASpec Object Summary**

The ARIMASpec object generates autoregressive integrated moving average (ARIMA) model specifications for use with a TSM object. 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**: Add the ARIMASpec to a TSM object using the TSM.Initialize method.
You can also store the XML representation of the ARIMA model to a CAS table. For more information, see the TSMSpec object. Data flow through the ARIMASpec object is illustrated in Figure 5.3

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 notes summarize where these features can be used. More details can be found in the method descriptions.

1. ARIMASpec.SetDiff supports a seasonal wildcard lag in the DiffArray.
2. ARIMASpec.AddARPoly and ARIMASpec.AddMAPoly support addition of seasonal ARIMA polynomial factors.
3. ARIMASpec.AddTF supports a seasonal wildcard lag in the DiffArray.
4. ARIMASpec.AddTFNumPoly and ARIMASpec.AddTFDenPoly support addition of seasonal ARIMA polynomial factors for transfer function numerator and denominator, respectively.
5. 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 without regard to the NOEST option setting. 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.

Orders specified for differencing and polynomial factors must be integers greater than or equal to 1.
Figure 5.3 ARIMASpec Object Data flow
ARIMASpec Synopsis

The object declaration statement creates a new object, \textit{obj}, of type \texttt{ARIMASpec}.

\begin{verbatim}
DECLARE OBJECT obj (ARIMASpec) ;
\end{verbatim}

Table 5.11 summarizes the methods that are associated with the ARIMASpec method.

<table>
<thead>
<tr>
<th>ARIMASpec Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Open</td>
<td>Open ARIMA model specification.</td>
</tr>
<tr>
<td>Close</td>
<td>Close ARIMA model specification.</td>
</tr>
<tr>
<td>SetTransform</td>
<td>Specify transform.</td>
</tr>
<tr>
<td>SetDiff</td>
<td>Add differencing to the ARIMA model.</td>
</tr>
<tr>
<td>AddARPoly</td>
<td>Add autoregressive polynomial factor to the ARIMA model.</td>
</tr>
<tr>
<td>AddMAPoly</td>
<td>Add moving average polynomial factor to the ARIMA model.</td>
</tr>
<tr>
<td>AddTF</td>
<td>Add transfer function to the ARIMA model.</td>
</tr>
<tr>
<td>AddTFNumPoly</td>
<td>Add transfer function numerator factors to the ARIMA model.</td>
</tr>
<tr>
<td>AddTFDenPoly</td>
<td>Add transfer function denominator factors to the ARIMA model.</td>
</tr>
<tr>
<td>SetTFTransform</td>
<td>Specify the transfer function transform.</td>
</tr>
<tr>
<td>SetOption</td>
<td>Specify the ARIMA option.</td>
</tr>
</tbody>
</table>

ARIMASpec Methods

This section describes the methods of a ARIMASpec object.

ARIMASpec.\texttt{Open} Method

The \texttt{Open} method initializes an empty ARIMASpec object for configuration.

Usage: \begin{verbatim}
rc = obj.Open () ;
\end{verbatim}

\textbf{Parameters}
There are no parameters associated with this method.
ARIMASpec.Close Method

The Close method 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 catalog.

Usage: \( rc = obj.\text{Close}() \);

Parameters
There are no parameters associated with this method.

ARIMASpec.SetTransform Method

The SetTransform method specifies the functional transform ‘Type’ to be used by the ARIMA model. Optional arguments \textit{Option} and \textit{Parm} offer greater control over the transform.

Usage: \( rc = obj.\text{SetTransform}('Type' < , 'Option', Parm>) \);

Parameters
- [in] \textit{Type} Character string that specifies the functional transform to use. Table 5.12 describes the transformations available. The default value is NONE.
- [in] \textit{Option} Optional character string that specifies prediction semantics for the inverse transform. Values are 'MEAN' or 'MEDIAN'. The default is 'MEAN'.
- [in] \textit{Parm} Optional numeric variable that specifies a control parameter (a number between \(-5\) and \(5\)) for the functional transform. This parameter is allowed only for Box-Cox transforms.

\textbf{Table 5.12} ‘Type’ Arguments for ARIMASpec.SetTransform

<table>
<thead>
<tr>
<th>Transform</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NONE</td>
<td>No transformation</td>
</tr>
<tr>
<td>LOG</td>
<td>Log transformation (positive series)</td>
</tr>
<tr>
<td>SQRT</td>
<td>Square root transformation (nonnegative series)</td>
</tr>
<tr>
<td>LOGISTIC</td>
<td>Logistic transformation</td>
</tr>
<tr>
<td>BOXCOX</td>
<td>Box-Cox transformation ((-5 \leq Parm \leq 5))</td>
</tr>
</tbody>
</table>

ARIMASpec.SetDiff Method

The SetDiff method adds differencing to the ARIMA model.

Usage: \( rc = obj.\text{SetDiff}(\text{DiffArray} <, \text{NDiff}>) \);
Parameters

[in] **DiffArray**  Numeric array that specifies differencing orders, which must be an integer greater than or equal to 1.

[in] **NDiff**  Optional numeric variable that specifies the number of DiffArray values to use. The default is to use all elements of DiffArray.

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.

**ARIMASpec.AddARPoly Method**

The AddARPoly method adds an autoregressive (AR) polynomial factor to ARIMA model. Additional AR polynomial factors can be added to the ARIMA model with subsequent calls to the AddARPoly method.

Usage:

```
rc = obj.AddARPoly ( OrderArray <, NOrder, Seasonal, CoeffArray> );
```

Parameters

[in] **OrderArray**  Numeric array that specifies AR polynomial lags. Valid values are integers greater than or equal to 1.

[in] **NOrder**  Optional numeric variable that specifies the number of OrderArray values to use. By default, all OrderArray values are used.

[in] **Seasonal**  Optional numeric variable (0 or 1). A value of 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.

[in] **CoeffArray**  Optional numeric array that specifies the AR polynomial coefficients. If specified, it must be of the same cardinality as the OrderArray.

**ARIMASpec.AddMAPoly Method**

The AddMAPoly method adds a moving average (MA) polynomial factor to ARIMA model. Additional MA polynomial factors can be added to the ARIMA model with subsequent calls to the AddMAPoly method.

Usage:

```
rct = obj.AddMAPoly ( OrderArray <, NOrder, Seasonal, CoeffArray> );
```
Parameters

- **OrderArray**: Numeric array that specifies MA polynomial lags, which must be greater than or equal to 1.
- **NOrder**: Optional numeric variable that specifies the number of OrderArray values to use. By default, all OrderArray values are used.
- **Seasonal**: Optional numeric variable (0 or 1). A value of 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**: Optional numeric array that specifies the MA polynomial coefficients. If specified, it must be of the same cardinality as the OrderArray.

ARIMASpec.AddTF Method

The AddTF method adds a transfer function to the ARIMA model for the specified XName variable. This adds the variable as a simple scale effect subject to any specified lag and differencing that might be applied.

Usage: \( rc = obj.AddTF( XName, < Delay, DifArray, NDiff > ) ; \)

Parameters

- **XName**: Character string that specifies the name of the X variable.
- **Delay**: Optional numeric variable that specifies the simple delay for the predictor. If not specified, the default is 0.
- **DifArray**: Optional 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.
- **NDiff**: Optional numeric variable that specifies the number of DifArray values to use. If not specified, then the size of DifArray is assumed.

Values in DifArray are interpreted as follows:

1. Negative values are not allowed and result in an error condition.
2. Nonnegative values represent differencing orders.
3. An .S missing value is interpreted to include seasonal difference order.
4. Any other missing value is ignored.

ARIMASpec.AddTFNumPoly Method

The AddTFNumPoly method adds a transfer function numerator polynomial factor for the specified XName variable. Additional polynomials can be added with subsequent calls to the AddTFNumPoly method.

Usage: \( rc = obj.AddTFNumPoly( XName, NumArray <, NNum, Seasonal, CoeffArray> ) ; \)
Parameters

[in] XName    Character string that specifies the name of the X variable.
[in] NumArray Numeric array that specifies numerator polynomial lags for the X variable. Valid values are integers greater than 0.
[in] NNum     Optional numeric variable that specifies the number of NumArray values to use. By default, all NumArray values are used.
[in] Seasonal Optional numeric variable (0 or 1). A value of 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.
[in] CoeffArray Optional numeric array that specifies the numerator polynomial coefficients.

ARIMASpec.AddTFDenPoly Method

The AddTFDenPoly method adds a transfer function denominator polynomial factor for the specified XName variable. Additional polynomials can be added with subsequent calls to the AddTFNumPoly method.

Usage: \[rc = obj.AddTFDenPoly ( XName, DenArray, NDen, Seasonal, CoeffArray )\];

Parameters

[in] XName    Character string that specifies the name of the X variable.
[in] DenArray Numeric array that specifies denominator polynomial lags for the X variable. Valid values are integers greater than 0.
[in] NDen     Optional numeric variable that specifies the number of DenArray values to use. By default, all DenArray values are interpreted.
[in] Seasonal Optional numeric variable (0 or 1). A value of 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.
[in] CoeffArray Optional numeric array that specifies the denominator polynomial coefficients.

ARIMASpec.SetTFTransform Method

The SetTFTransform method specifies a functional transform for specified XName variable.

Usage: \[rc = obj.SetTFTransform ( XName, 'Type', Parm )\);

Parameters

[in] XName    Character string that specifies the name of the X variable.
[in] Type     Character string that specifies the functional transform to use. Table 5.13 summarizes the supported transforms. The default is no transform.
[in] Parm     Optional numeric variable that specifies a control parameter (a number between –5 and 5) for the functional transform. This parameter is used only for Box-Cox transforms.
Table 5.13  ‘Type’ Arguments for ARIMASpec.SetTFTransform

<table>
<thead>
<tr>
<th>Transform</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NONE</td>
<td>No transformation</td>
</tr>
<tr>
<td>LOG</td>
<td>Log transformation (positive series)</td>
</tr>
<tr>
<td>SQRT</td>
<td>Square root transformation (non-negative series)</td>
</tr>
<tr>
<td>LOGISTIC</td>
<td>Logistic transformation</td>
</tr>
<tr>
<td>BOXCOX</td>
<td>Box-Cox transformation ((-5 \leq Parm \leq 5))</td>
</tr>
</tbody>
</table>

ARIMASpec.SetOption Method

The SetOption method specifies ARIMA model options. Options are \(\langle \text{Name}, \text{Value} \rangle\) pairs where \(\text{Name}\) is a case-insensitive character string and \(\text{Value}\) is dependent on the named option.

Usage: \(\text{rc} = \text{obj.SetOption}(\langle \text{Name}, \text{Value} <, \langle \text{Name}, \text{Value}, ... > \rangle)\);

Parameters

[in] Name  Character string that specifies the ARIMA option name. Table 5.14 lists valid option names and corresponding value description. Default values are in parentheses.

[in] Value  variable that specifies the option value.

Table 5.14  ‘Name’ Arguments for ARIMASpec.SetOption

<table>
<thead>
<tr>
<th>Name</th>
<th>Allowed Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>CONVERGE</td>
<td>(0 &lt; \text{Value} &lt; 1.0) ((0.001))</td>
</tr>
<tr>
<td>DELTA</td>
<td>(0 &lt; \text{Value} &lt; 1.0) ((0.001))</td>
</tr>
<tr>
<td>MAXITER</td>
<td>Positive integer (50)</td>
</tr>
<tr>
<td>METHOD</td>
<td>‘CLS’, ‘ML’, ‘ULS’ (\langle \text{CLS} \rangle)</td>
</tr>
<tr>
<td>MU</td>
<td>Constant for ARIMA model</td>
</tr>
<tr>
<td>NOEST</td>
<td>0=Estimate, 1=No estimation (0)</td>
</tr>
<tr>
<td>NOINT</td>
<td>0=Intercept, 1=No intercept (0)</td>
</tr>
<tr>
<td>NOSTABLE</td>
<td>0=Check stability, 1=Do not check stability (0)</td>
</tr>
<tr>
<td>SINGULAR</td>
<td>(0 &lt; \text{Value} &lt; 1.0) ((1E-7))</td>
</tr>
</tbody>
</table>
**TSMPEst Object Summary**

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. Data flow through the TSMPEst object is illustrated in Figure 5.4.

![Figure 5.4 TSMPEst Object Data flow](image)

Table 5.15 summarizes the methods that are associated with the TSMPEst object.

<table>
<thead>
<tr>
<th>TSMPEst Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Collect</td>
<td>Collects parameter estimates from TSM objects.</td>
</tr>
<tr>
<td>nrows</td>
<td>Returns the number of rows that are collected.</td>
</tr>
</tbody>
</table>
TSMPEst Methods

This section describes the methods of a TSMPEst object.

TSMPEst.Collect Method

The Collect method collects time series model parameter estimates from a TSM object, TSMObj, and stores them in a CAS table.

Usage: \[ rc = obj.Collect ( TSMObj ) ; \]

Parameters

[in] TSMObj
The TSM object to use as the source of time series model parameter estimates.

TSMPEst.Nrows Attribute

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

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

TSMSpec Object Summary

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. Data flow through the TSMSpec object is illustrated in Figure 5.5

Figure 5.5 TSMSpec Object Data flow
**TSMSpec Synopsis**

The object declaration statement creates a new object, `obj`, of type `TSMSpec`.

```
DECLARE OBJECT obj (TSMSpec) ;
```

Table 5.16 summarizes the methods of the TSMSpec object:

<table>
<thead>
<tr>
<th>TSMSpec Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Collect</td>
<td>Collects model specification from TSM objects.</td>
</tr>
</tbody>
</table>

**TSMSpec Methods**

This section describes the methods of a TSMSpec object.

**TSMSpec.Collect Method**

The Collect method collects time series model specifications from a TSM object, `TSMObj`, and stores them in a CAS table.

Usage: 

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

*Parameters*

- `[in]` `TSMObj` The TSM object to use as the source of time series model specification.

**TSMFor Object Summary**

The Time Series Model Forecast (TSMFor) collector 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 that used by HPFENGINE procedure for its OUTFOR data set and is described in Table 5.17. Data flow through the TSMFor object is illustrated in Figure 5.6.

<table>
<thead>
<tr>
<th>Table 5.17 Forecast Series Variables</th>
</tr>
</thead>
<tbody>
<tr>
<td>Column</td>
</tr>
<tr>
<td><em>NAME</em></td>
</tr>
<tr>
<td><em>TIMEID</em></td>
</tr>
<tr>
<td>ACTUAL</td>
</tr>
<tr>
<td>PREDICT</td>
</tr>
</tbody>
</table>
Table 5.17  continued

<table>
<thead>
<tr>
<th>Column</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ERROR</td>
<td>Numeric</td>
<td>Residuals</td>
</tr>
<tr>
<td>STD</td>
<td>Numeric</td>
<td>Prediction standard error</td>
</tr>
<tr>
<td>UPPER</td>
<td>Numeric</td>
<td>Upper confidence limit</td>
</tr>
<tr>
<td>LOWER</td>
<td>Numeric</td>
<td>Lower confidence limit</td>
</tr>
</tbody>
</table>

Figure 5.6  TSMFor Object Data flow

TSMFor Synopsis

The object declaration statement creates a new object, *obj*, of type *TSMFor*.

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

Table 5.18 summarizes the methods of the TSMFor object:
### Table 5.18 Methods of the TSMFor Object

<table>
<thead>
<tr>
<th>TSMFor Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Collect</td>
<td>Collects forecasts estimates from TSM objects.</td>
</tr>
<tr>
<td>nrows</td>
<td>Returns the number of rows collected.</td>
</tr>
</tbody>
</table>

---

**TSMFor Methods**

This section describes the methods of a TSMFor object.

**TSMFor.Collect Method**

The collect method collects the forecast series from a TSM object, `ModelObj`. An optional `Region` argument can be specified to indicate the forecast region to be collected.

**Usage:**

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

**Parameters**

- **ModelObj** [in] Character string name of the TSM object to use as the source of time series model forecasts.
- **Region** [in] Optional character string or numeric array to specify the time region to be collected. The default is ALL. Data types for Region are described in Table 5.19

**Table 5.19 Region Argument Data Types**

<table>
<thead>
<tr>
<th>Type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>String</td>
<td>Collection region (ALL</td>
</tr>
</tbody>
</table>

**TSMFor Attributes**

The TSMFor object contains a single attribute, which is describe next.

**TSMFor.nrows Method**

The nrows method returns the number of rows—an attribute of the TSMFor object—that have been collected.

**Usage:**

```r
rc = obj.nrows ( ) ;
```
TSMINEst Object Summary

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. TSMINEst object must be used in conjunction with a TSMINSpec spec object and a TSM model object to replay a model with its saved parameters (for an example, see Example 5.3).

TSMINEst Synopsis

The object declaration statement creates a new object, obj, of type TSMINEst.

```
DECLARE OBJECT obj (TSMINEst) ;
```

Table 5.20 summarizes the methods of the TSMINEst object.

<table>
<thead>
<tr>
<th>TSMINEst Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>nrows</td>
<td>Returns number of rows in the TSMINEst object</td>
</tr>
</tbody>
</table>

TSMINEst Methods

There are no methods associated with this object.

TSMINEst Attributes

The TSMINEst object contains a single attribute, which is describe next.

TSMINEst.nrows Attribute

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

Usage: 

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

TSMINspec Object Summary

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.
TSMINSpec Synopsis

The object declaration statement creates a new object, obj, of type TSMINSpec.

DECLARE OBJECT obj (TSMINSpec) ;

Table 5.21 summarizes the methods of the TSMINSpec object.

Table 5.21 Methods of the TSMINSpec Object

<table>
<thead>
<tr>
<th>TSMINSpec Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>nrows</td>
<td>Number of rows in TSMINSpec object</td>
</tr>
</tbody>
</table>

TSMINSpec Methods

There are no methods associated with this object.

TSMINSpec Attributes

The TSMINSpec object contains a single attribute, which is describe next.

TSMINSpec.nrows Attribute

The nrows attribute returns the number of rows in the TSMINSpec object. A returned missing value indicates that the TSMINSpec object has not been successfully configured.

Usage:  
rc = obj.nrows ( ) ;

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 in mycas, a CAS library that you have necessary permissions to work with. In all the examples of this section, the functionality of the TSM package is illustrated using the TSMODEL procedure. This section assumes that you are familiar with the general workings of the TSMODEL procedure; for more information, see Chapter 2, “The TSMODEL Procedure” (SAS Visual Forecasting: Forecasting Procedures).

Example 5.1: 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 on ARIMA
modeling of Series G, see “Example 7.2 Seasonal Model for the Airline Series” in the ARIMA chapter of SAS/ETS User’s Guide. This example shows how you can use the objects in the TSM package to fit the airline model, ARIMA(0,1,1)(0,1,1)_{12} NOINT, and the Winters exponential smoothing model to the airline series. In particular, it shows how you can do the following:

- Create ARIMA and ESM specifications by using the ARIMASpec and ESMSpec specification objects.
- Use these specification objects to initialize the TSM model objects.
- 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.
- Postprocess the forecast results to compute an ad hoc statistic.
- Output the computed results to CAS tables.

The broad outline of this program is as follows:

- 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).
- The ID statement specifies date as the time index variable, and the INTERVAL= option indicates that the data are monthly.
- The VAR statement specifies the input data set variable, air, which contains the airline series.
- 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.
- The REQUIRE statement specifies the TSM package, which is needed for the analysis.
- The statements between the SUBMIT and ENDSUBMIT statements use the TSM package objects to perform the actual analysis in your CAS session.
- 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 merely to illustrate how you can write your own custom postprocessing code to analyze the results that are produced by the TSM objects.
Example 5.1: Fitting and Forecasting with ARIMA and ESM Models

```sas
proc tsmmodel 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 Airline Model ***;
    declare object airModel(tsm);
    declare object airSpec(arimaspec);

    *** --setup 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 MA orders: q = (1)(12) ***;
    *** Use AddMAPoly twice for the two factors ***;
    ma[1] = 1;
    ma[1] = 12;
    rc = airSpec.AddMAPoly(ma);
    *** Specify NOINT ***;
    rc = airSpec.SetOption('noint',1);
    *** Specify log transform ***;
    rc = airSpec.SetTransform('log');
    *** done setting up ARIMA model ***;
    rc = airSpec.Close( );

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

    *** output Airline Model forecasts and estimates ***;
    declare object airFor(tsmfor);
    declare object airEst(tsmpest);
    rc = airFor.Collect(airModel);
    rc = airEst.Collect(airModel);

    **put 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( );

*** setup and run the TSM object ***;
rc = esmModel.Initialize(esmspec);
rc = esmModel.SetY(Air);
rc = esmModel.SetOption('lead',12);
rc = esmModel.Run( );

**put 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 5.1.1 shows the predictions, and Output 5.1.2 shows the parameter estimates for the airline model.

Output 5.1.1  Airline Model Predictions (Partial Output)

<table>
<thead>
<tr>
<th>DATE</th>
<th>PREDICT</th>
<th>STD</th>
<th>UPPER</th>
<th>LOWER</th>
</tr>
</thead>
<tbody>
<tr>
<td>JAN1961</td>
<td>450.4</td>
<td>16.9215</td>
<td>484.5</td>
<td>418.2</td>
</tr>
<tr>
<td>FEB1961</td>
<td>426.1</td>
<td>18.8590</td>
<td>464.2</td>
<td>390.3</td>
</tr>
<tr>
<td>MAR1961</td>
<td>480.1</td>
<td>24.0408</td>
<td>528.9</td>
<td>434.7</td>
</tr>
<tr>
<td>APR1961</td>
<td>492.8</td>
<td>27.2405</td>
<td>548.3</td>
<td>441.6</td>
</tr>
<tr>
<td>MAY1961</td>
<td>509.5</td>
<td>30.5863</td>
<td>572.0</td>
<td>452.1</td>
</tr>
<tr>
<td>JUN1961</td>
<td>584.2</td>
<td>37.6514</td>
<td>661.4</td>
<td>513.9</td>
</tr>
<tr>
<td>JUL1961</td>
<td>670.7</td>
<td>45.9957</td>
<td>765.3</td>
<td>585.1</td>
</tr>
<tr>
<td>AUG1961</td>
<td>668.2</td>
<td>48.4237</td>
<td>768.0</td>
<td>578.3</td>
</tr>
<tr>
<td>SEP1961</td>
<td>559.6</td>
<td>42.6271</td>
<td>647.7</td>
<td>480.7</td>
</tr>
<tr>
<td>OCT1961</td>
<td>498.3</td>
<td>39.7181</td>
<td>580.6</td>
<td>425.0</td>
</tr>
<tr>
<td>NOV1961</td>
<td>431.2</td>
<td>35.8240</td>
<td>505.5</td>
<td>365.2</td>
</tr>
<tr>
<td>DEC1961</td>
<td>478.9</td>
<td>41.3484</td>
<td>565.0</td>
<td>403.0</td>
</tr>
</tbody>
</table>
Example 5.2: 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, time has been assigned second as the time interval (which is one of the simplest interval types for sequential indexing). As shown in “Example 7.3 Model for Series J Data from Box and Jenkins” in the ARIMA chapter of SAS/ETS User’s Guide, a reasonable ARIMAX model for y turns out to be $y = \text{TFinput}(x) + \text{AR}(2)$, where TFinput(x) is a transfer function term in x with a delay of 3, numerator polynomial of order 2, denominator polynomial of order 1; and AR(2) is an error term of autoregressive order 2.

The following statements show you how to fit this model by using the objects in the TSM package:

```sas
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;

*** setup the Transfer Function Model spec: ***;
rc = jSpec.Open( );

*** Specify AR orders: p = (1 2) ***;
```

Output 5.1.2 Airline Model Parameter Estimates (Partial Output)

<table>
<thead>
<tr>
<th>EST</th>
<th>STDERR</th>
<th>TVALUE</th>
<th>PVALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.3773</td>
<td>0.0820</td>
<td>4.6033</td>
<td>9.828E-6</td>
</tr>
<tr>
<td>0.5724</td>
<td>0.0780</td>
<td>7.3361</td>
<td>2.17E-11</td>
</tr>
</tbody>
</table>

Output 5.1.3 shows the number of times the airline model residuals are smaller than the Winters model residuals.

Output 5.1.3 Number of Times the Airline Model Residuals Are Smaller Than the Winters Model Residuals

<table>
<thead>
<tr>
<th>nbetter</th>
</tr>
</thead>
<tbody>
<tr>
<td>71.0000</td>
</tr>
</tbody>
</table>
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.AddTFTNumPoly('x', num);
den[1] = 1;
rc = jSpec.AddTTFDenPoly('x', den);
*** done setting up ARIMA model ***;
rc = jSpec.Close( );

*** setup and run the TSM object ***;
rc = jModel.Initialize(jSpec);
rc = jModel.SetY(y);
rc = jModel.AddX(x);
rc = jModel.Run( );

*** output Airline Model forecasts and estimates ***;
rc = jEst.Collect(jModel);
endsubmit;
quit;

Output 5.2.1 shows the parameter estimates for the transfer function model.

**Output 5.2.1** Parameter Estimates for the Transfer Function Model (Partial Output)

<table>
<thead>
<tr>
<th>EST</th>
<th>STDERR</th>
<th>TVALUE</th>
<th>PVALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>53.2630</td>
<td>0.1193</td>
<td>446.5</td>
<td>0</td>
</tr>
<tr>
<td>1.5329</td>
<td>0.0475</td>
<td>32.2472</td>
<td>6.25E-97</td>
</tr>
<tr>
<td>-0.6330</td>
<td>0.0501</td>
<td>-12.6434</td>
<td>2.27E-29</td>
</tr>
<tr>
<td>-0.5352</td>
<td>0.0748</td>
<td>-7.1534</td>
<td>7.21E-12</td>
</tr>
<tr>
<td>0.3760</td>
<td>0.1029</td>
<td>3.6553</td>
<td>0.000306</td>
</tr>
<tr>
<td>0.5189</td>
<td>0.1078</td>
<td>4.8124</td>
<td>2.425E-6</td>
</tr>
<tr>
<td>0.5484</td>
<td>0.0382</td>
<td>14.3499</td>
<td>1.72E-35</td>
</tr>
</tbody>
</table>

**Example 5.3: 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 set). This example shows how you can do the following:

- Save the model specification and parameter estimates for later use by using the Collect method of the TSMSpec and TSMPest objects, respectively.
- Reuse the previously saved model specification and parameter estimates to configure a TSM object by using the Replay method.
- Produce the model forecasts by using this TSM object.
Example 5.3: Replaying a Previously Fitted Model

The following statements fit the airline model to the airline series (see Example 5.1 for more information about the airline series and the airline model). The model specification and parameter estimates are stored in CAS tables mycas.airOSpec and mycas.airEst, respectively.

```plaintext
proc tsmoodel 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;

  *** setup 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 MA 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 log transform ***;
  rc = airSpec.SetTransform('log');
  *** done setting up ARIMA model ***;
  rc = airSpec.Close();

  *** setup 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 5.3.1 shows the parameter estimates that are saved in mycas.airEst.
Output 5.3.1 Parameter Estimates for the Airline Model (Partial Output)

<table>
<thead>
<tr>
<th><em>EST</em></th>
<th><em>STDERR</em></th>
<th><em>TVALUE</em></th>
<th><em>PVALUE</em></th>
</tr>
</thead>
<tbody>
<tr>
<td>0.3773</td>
<td>0.0820</td>
<td>4.6033</td>
<td>9.828E-6</td>
</tr>
<tr>
<td>0.5724</td>
<td>0.0780</td>
<td>7.3361</td>
<td>2.17E-11</td>
</tr>
</tbody>
</table>

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

```plaintext
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 Airline Model ***;
declare object airModel(tsm);
declare object airSpec(tsminspec);
declare object airEst(tsminest);
declare object airFor(tsmfor);

*** setup 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 Airline Model forecasts ***;
rc = airFor.Collect(airModel);
endsubmit;
quit;
```

Output 5.3.2 shows the forecasts that are produced according to the fitted model.
**Output 5.3.2** Replayed Forecasts (Partial Output)

**Airline Model Predictions**

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

**References**

Chapter 6
Utility Package for the TSMODEL Procedure

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 6.1 summarizes the object classes that are contained in the UTL package.

<table>
<thead>
<tr>
<th>UTL Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLIMITS</td>
<td>Compute prediction standard errors and confidence limits for specified actual and predicted time series.</td>
</tr>
<tr>
<td>OUTNVP</td>
<td>Ad hoc name,value pair collector object for storing numeric scalar or array variables that are found in a user program into a CAS table.</td>
</tr>
<tr>
<td>UTLSTAT</td>
<td>Collector object for computing forecast fit statistics for specified actual and predicted time series and storing those statistics in a CAS table.</td>
</tr>
</tbody>
</table>

### Using the UTL Package

The objects in the UTL package are subdivided into two different categories:

1. Stateful computational objects (the CLIMITS object)
2. Collector objects (the OUTNVP and UTLSTAT objects)

Collector objects provide a mechanism to create a snapshot of results (either from stateful objects or from plain program variables) and store those results into CAS tables. Each collector object defines a CAS table schema that is determined by the collector object’s design. The collector objects in the UTL package (OUTNVP and UTLSTAT) follow a common method pattern. The basic execution follows this sequence of operations:

1. **Declare**: Create the collector object by using the object declaration statement.
2. **Collect**: Use the Collect method to store results into a CAS table. The input arguments of the Collect method are specific to the collector object. For example, the UTLSTAT collector object’s Collect method requires an actual and predicted time series as arguments. It then uses the specified series to compute forecast fit statistics and stores the results in a CAS table. In contrast, the OUTNVP
collector object’s Collect method takes in an ad hoc numerical scalar or array variable from the user program and stores it into a CAS table. Rows that are collected are automatically appended to the collector’s associated CAS table at the end of each BY group, and the collector object’s saved row set is automatically reset. The Nrows attribute returns the current row count in the collector. A missing value is returned if nothing has been collected. The data, now stored in CAS tables, can then be used to produce reports or be used in further computations.

### Common Argument Types

Table 6.2 defines the common argument types that are used in this chapter.

**Table 6.2  Common Argument Types**

<table>
<thead>
<tr>
<th>SAS Data Type</th>
<th>Declaration Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>String</td>
<td>LENGTH x $n;</td>
</tr>
<tr>
<td>Numeric</td>
<td>x or LENGTH x 8;</td>
</tr>
<tr>
<td>Numeric array</td>
<td>ARRAY x[n]/NOSYMBOLS;</td>
</tr>
<tr>
<td>Status</td>
<td>x or LENGTH x 8;</td>
</tr>
</tbody>
</table>

### Status Code Convention

Table 6.3 shows the return code (rc) values that are used by all objects in this chapter.

**Table 6.3  Status Codes**

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

Upon returning a negative status code, most methods in the 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 2, “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:

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

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

<table>
<thead>
<tr>
<th>CLIMIT Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compute</td>
<td>Compute the prediction standard errors and confidence limits for specified</td>
</tr>
<tr>
<td></td>
<td>actual and predicted time series.</td>
</tr>
<tr>
<td>GetForecast</td>
<td>Retrieve a computed forecast series by name.</td>
</tr>
</tbody>
</table>

Figure 6.1 outlines the programmatic data flow through the CLIMITS object.
**Figure 6.1** 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

- **Actual** (in) Numeric array that corresponds to the actual time series.
- **Predicted** (in) Numeric array that corresponds to the predicted time series.
- **Alpha** (in) 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

- **Which** (in) Character literal or string whose value specifies the forecast series to return. Table 6.5 shows the series types that are supported.

<table>
<thead>
<tr>
<th>Which</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>STDERR</td>
<td>Forecast standard error series</td>
</tr>
<tr>
<td>UPPER</td>
<td>Upper confidence limit series</td>
</tr>
<tr>
<td>LOWER</td>
<td>Lower confidence limit series</td>
</tr>
</tbody>
</table>

- **Result** (out) 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 6.6.

<table>
<thead>
<tr>
<th>Column</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>NAME</em></td>
<td>String</td>
<td>Name of the variable</td>
</tr>
<tr>
<td><em>CALL</em></td>
<td>Numeric</td>
<td>Call count within the BY group</td>
</tr>
<tr>
<td><em>UTAG</em></td>
<td>Numeric</td>
<td>User-defined numeric tag</td>
</tr>
<tr>
<td><em>VIX</em></td>
<td>Numeric</td>
<td>Value index (1-based) for the row</td>
</tr>
<tr>
<td><em>VALUE</em></td>
<td>Numeric</td>
<td>Actual value for variable’s row</td>
</tr>
</tbody>
</table>

OUTNVP Synopsis

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

```
DECLARE OBJECT obj (OUTNVP) ;
```

Table 6.7 summarizes the methods and attributes that are associated with the OUTNVP object.

<table>
<thead>
<tr>
<th>OUTNVP Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Collect</td>
<td>Collect value for numeric data type (scalar literal or variable or array variable).</td>
</tr>
<tr>
<td>nrows</td>
<td>Get the OUTNVP instance row count.</td>
</tr>
</tbody>
</table>

Figure 6.2 outlines the programmatic data flow through the OUTNVP object.
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 6.8. For more information about the HPFENGINE procedure, see *SAS Forecast Server Procedures: User’s Guide*.

<table>
<thead>
<tr>
<th>Column</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>NAME</em></td>
<td>String</td>
<td>Variable name of actual series</td>
</tr>
<tr>
<td><em>MODEL</em></td>
<td>String</td>
<td>Variable name of predicted series</td>
</tr>
<tr>
<td>DFE</td>
<td>Numeric</td>
<td>Degrees of freedom error</td>
</tr>
<tr>
<td>N</td>
<td>Numeric</td>
<td>Number of observations</td>
</tr>
<tr>
<td>NOBS</td>
<td>Numeric</td>
<td>Number of observations used</td>
</tr>
<tr>
<td>NMISSA</td>
<td>Numeric</td>
<td>Number of missing actuals</td>
</tr>
<tr>
<td>NMISSP</td>
<td>Numeric</td>
<td>Number of missing predicted values</td>
</tr>
<tr>
<td>NPARMS</td>
<td>Numeric</td>
<td>Number of model parameters</td>
</tr>
<tr>
<td>TSS</td>
<td>Numeric</td>
<td>Total sum of squares</td>
</tr>
<tr>
<td>SST</td>
<td>Numeric</td>
<td>Corrected total sum of squares</td>
</tr>
</tbody>
</table>
### Table 6.8 continued

<table>
<thead>
<tr>
<th>Column</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SSE</td>
<td>Numeric</td>
<td>Sum of square error</td>
</tr>
<tr>
<td>MSE</td>
<td>Numeric</td>
<td>Mean square error</td>
</tr>
<tr>
<td>RMSE</td>
<td>Numeric</td>
<td>Root mean square error</td>
</tr>
<tr>
<td>UMSE</td>
<td>Numeric</td>
<td>Unbiased mean square error</td>
</tr>
<tr>
<td>URMSE</td>
<td>Numeric</td>
<td>Unbiased root mean square error</td>
</tr>
<tr>
<td>MAPE</td>
<td>Numeric</td>
<td>Mean absolute percentage of error</td>
</tr>
<tr>
<td>MAE</td>
<td>Numeric</td>
<td>Mean absolute error</td>
</tr>
<tr>
<td>RSQUARE</td>
<td>Numeric</td>
<td>R-square</td>
</tr>
<tr>
<td>ADJRSQ</td>
<td>Numeric</td>
<td>Adjusted R-square</td>
</tr>
<tr>
<td>AADJRSQ</td>
<td>Numeric</td>
<td>Amemiya’s adjusted R-square</td>
</tr>
<tr>
<td>RWRSQ</td>
<td>Numeric</td>
<td>Random walk R-square</td>
</tr>
<tr>
<td>AIC</td>
<td>Numeric</td>
<td>Akaike’s information criterion</td>
</tr>
<tr>
<td>AICC</td>
<td>Numeric</td>
<td>Finite sample corrected Akaike’s information criterion</td>
</tr>
<tr>
<td>SBC</td>
<td>Numeric</td>
<td>Schwarz Bayesian information criterion</td>
</tr>
<tr>
<td>APC</td>
<td>Numeric</td>
<td>Amemiya’s prediction criterion</td>
</tr>
<tr>
<td>MAXERR</td>
<td>Numeric</td>
<td>Maximum error</td>
</tr>
<tr>
<td>MINERR</td>
<td>Numeric</td>
<td>Minimum error</td>
</tr>
<tr>
<td>MAXPE</td>
<td>Numeric</td>
<td>Maximum percentage of error</td>
</tr>
<tr>
<td>MINPE</td>
<td>Numeric</td>
<td>Minimum percentage of error</td>
</tr>
<tr>
<td>ME</td>
<td>Numeric</td>
<td>Mean error</td>
</tr>
<tr>
<td>MPE</td>
<td>Numeric</td>
<td>Mean percentage of error</td>
</tr>
<tr>
<td>MDAPE</td>
<td>Numeric</td>
<td>Median absolute percentage of error</td>
</tr>
<tr>
<td>GMAPE</td>
<td>Numeric</td>
<td>Geometric mean absolute percentage of error</td>
</tr>
<tr>
<td>MINPPE</td>
<td>Numeric</td>
<td>Minimum predicted percentage of error</td>
</tr>
<tr>
<td>MAXPPE</td>
<td>Numeric</td>
<td>Maximum predicted percentage of error</td>
</tr>
<tr>
<td>MPPE</td>
<td>Numeric</td>
<td>Mean predicted percentage of error</td>
</tr>
<tr>
<td>MAPPE</td>
<td>Numeric</td>
<td>Mean absolute predicted percentage of error</td>
</tr>
<tr>
<td>MDAPPE</td>
<td>Numeric</td>
<td>Median absolute predicted percentage of error</td>
</tr>
<tr>
<td>GMAPPE</td>
<td>Numeric</td>
<td>Geometric mean absolute predicted percentage of error</td>
</tr>
</tbody>
</table>
Table 6.8  continued

<table>
<thead>
<tr>
<th>Column</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MINSPE</td>
<td>Numeric</td>
<td>Minimum symmetric percentage of error</td>
</tr>
<tr>
<td>MAXSPE</td>
<td>Numeric</td>
<td>Maximum symmetric percentage of error</td>
</tr>
<tr>
<td>MSPE</td>
<td>Numeric</td>
<td>Mean symmetric percentage of error</td>
</tr>
<tr>
<td>SMAPE</td>
<td>Numeric</td>
<td>Mean absolute symmetric percentage of error</td>
</tr>
<tr>
<td>MDASPE</td>
<td>Numeric</td>
<td>Median absolute symmetric percentage of error</td>
</tr>
<tr>
<td>GMASPE</td>
<td>Numeric</td>
<td>Geometric mean absolute symmetric percentage of error</td>
</tr>
<tr>
<td>MINRE</td>
<td>Numeric</td>
<td>Minimum relative error</td>
</tr>
<tr>
<td>MAXRE</td>
<td>Numeric</td>
<td>Maximum relative error</td>
</tr>
<tr>
<td>MRE</td>
<td>Numeric</td>
<td>Mean relative error</td>
</tr>
<tr>
<td>MRAE</td>
<td>Numeric</td>
<td>Mean relative absolute error</td>
</tr>
<tr>
<td>MDRAE</td>
<td>Numeric</td>
<td>Median relative absolute error</td>
</tr>
<tr>
<td>GMRAE</td>
<td>Numeric</td>
<td>Geometric mean relative absolute error</td>
</tr>
<tr>
<td>MASE</td>
<td>Numeric</td>
<td>Mean absolute scaled error</td>
</tr>
<tr>
<td>MINAPES</td>
<td>Numeric</td>
<td>Minimum absolute error percentage of standard deviation</td>
</tr>
<tr>
<td>MAXAPES</td>
<td>Numeric</td>
<td>Maximum absolute error percentage of standard deviation</td>
</tr>
<tr>
<td>MAPES</td>
<td>Numeric</td>
<td>Mean absolute error percentage of standard deviation</td>
</tr>
<tr>
<td>MDAPEPS</td>
<td>Numeric</td>
<td>Median absolute error percentage of standard deviation</td>
</tr>
<tr>
<td>GMAPES</td>
<td>Numeric</td>
<td>Geometric mean absolute error percentage of standard deviation</td>
</tr>
</tbody>
</table>

**UTLSTAT Synopsis**

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

```
DECLARE OBJECT obj (UTLSTAT) ;
```

Table 6.9 summarizes the methods of the UTLSTAT object:
Table 6.9  Methods of the UTLSTAT Object

<table>
<thead>
<tr>
<th>UTLSTAT Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Collect</td>
<td>Compute and collect forecast fit statistics for a specified ad hoc pair of actual and predicted time series.</td>
</tr>
<tr>
<td>nrows</td>
<td>Get the UTLSTAT instance row count.</td>
</tr>
</tbody>
</table>

Figure 6.3 outlines the programmatic data flow through the UTLSTAT object.

Figure 6.3  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 6.8. 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 6.1: 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 6.8) 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.

```plaintext
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.

```plaintext
class 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;
```
Example 6.1: Collecting Forecast Fit Statistics and Ad Hoc Numeric Variables into CAS Tables

/* 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 6.1.1 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 6.1.1 Sample of the Forecast Fit Statistics Computed and Collected by the UTLSTAT Object

<table>
<thead>
<tr>
<th>Obs</th>
<th><em>NAME</em></th>
<th><em>MODEL</em></th>
<th>NOBS</th>
<th>RMSE</th>
<th>MAPE</th>
<th>MAE</th>
<th>RSQUARE</th>
<th>AIC</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>AIR</td>
<td>predict</td>
<td>144</td>
<td>10.579085435</td>
<td>3.0845016398</td>
<td>8.0064787209</td>
<td>0.9921692375</td>
<td>679.35714621</td>
</tr>
</tbody>
</table>

You can use the PRINT procedure again to display a small subset of observations from the following six different series that were collected by the OUTNVP object into the mycas.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:

```sas
/* Sort OUTNVP table by "row index" and "BY group Collect() call count" */ /* (that is, _VIX_ and _CALL_ columns). Transfer the sorted table rows */ /* back from CAS and into a local data set. */ proc sort data=mycas.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 6.1.2** Sample of the Six Ad Hoc Series Collected by the OUTNVP Object

<table>
<thead>
<tr>
<th>Obs</th>
<th>VAR_</th>
<th>CALL_</th>
<th>UTAG_</th>
<th><em>VIX</em></th>
<th>VALUE_</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>AIR</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>112</td>
</tr>
<tr>
<td>2</td>
<td>predict</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>111.44700541</td>
</tr>
<tr>
<td>3</td>
<td>error</td>
<td>3</td>
<td>0</td>
<td>1</td>
<td>0.5529945856</td>
</tr>
<tr>
<td>4</td>
<td>stderr</td>
<td>4</td>
<td>0</td>
<td>1</td>
<td>10.691036547</td>
</tr>
<tr>
<td>5</td>
<td>lcl</td>
<td>5</td>
<td>0</td>
<td>1</td>
<td>90.492958825</td>
</tr>
<tr>
<td>6</td>
<td>ucl</td>
<td>6</td>
<td>0</td>
<td>1</td>
<td>132.401052</td>
</tr>
<tr>
<td>7</td>
<td>AIR</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>118</td>
</tr>
<tr>
<td>8</td>
<td>predict</td>
<td>2</td>
<td>0</td>
<td>2</td>
<td>119.94581467</td>
</tr>
<tr>
<td>9</td>
<td>error</td>
<td>3</td>
<td>0</td>
<td>2</td>
<td>-1.945814668</td>
</tr>
<tr>
<td>10</td>
<td>stderr</td>
<td>4</td>
<td>0</td>
<td>2</td>
<td>10.691036547</td>
</tr>
<tr>
<td>11</td>
<td>lcl</td>
<td>5</td>
<td>0</td>
<td>2</td>
<td>98.991768079</td>
</tr>
<tr>
<td>12</td>
<td>ucl</td>
<td>6</td>
<td>0</td>
<td>2</td>
<td>140.89986126</td>
</tr>
<tr>
<td>13</td>
<td>AIR</td>
<td>1</td>
<td>0</td>
<td>3</td>
<td>132</td>
</tr>
<tr>
<td>14</td>
<td>predict</td>
<td>2</td>
<td>0</td>
<td>3</td>
<td>135.17351431</td>
</tr>
<tr>
<td>15</td>
<td>error</td>
<td>3</td>
<td>0</td>
<td>3</td>
<td>-3.17351431</td>
</tr>
<tr>
<td>16</td>
<td>stderr</td>
<td>4</td>
<td>0</td>
<td>3</td>
<td>10.691036547</td>
</tr>
<tr>
<td>17</td>
<td>lcl</td>
<td>5</td>
<td>0</td>
<td>3</td>
<td>114.21946772</td>
</tr>
<tr>
<td>18</td>
<td>ucl</td>
<td>6</td>
<td>0</td>
<td>3</td>
<td>156.1275609</td>
</tr>
</tbody>
</table>

The sequence that is displayed in **Output 6.1.2** 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
Example 6.2: 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 external 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 6.1 to store into a CAS table the actual series plus the five forecast series that were retrieved from the TSM object via its own GetForecast method. Finally, some results are retrieved from CAS tables, sorted, and printed for further inspection:

The following DATA step creates a synthetic data set that contains actual and simulated predicted time series:

```plaintext
data ExternalModel (replace=yes);
  set Sashelp.Air; /* The actual series: Sashelp.Air */
  error = floor(10*ranuni(246)); /* Simulated forecast error series */
  predict = air + error; /* Simulated predicted series */
run;
```

The following DATA step loads the ExternalModel data set onto the CAS server. This DATA step assumes that your CAS engine libref is named mycas, but you can substitute any appropriately defined CAS engine libref.

```plaintext
data mycas.ExternalModel;
  set ExternalModel;
run;
```

The following statements use the TSMODEL procedure to compute the prediction standard errors and confidence limits of the ad hoc external forecast that resides in the synthetic data set:
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 */
  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 6.1, 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.outnvpmode1 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:

```plaintext
/* 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.outnvpmode1 out=outnvpmode1;
   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=outnvpmode1(obs=18);
run;
```

**Output 6.2.1** Sample of the Six Ad Hoc Series Collected by the OUTNVP Object

<table>
<thead>
<tr>
<th>Obs</th>
<th><em>VAR</em></th>
<th><em>CALL</em></th>
<th><em>UTAG</em></th>
<th><em>VIX</em></th>
<th><em>VALUE</em></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>AIR</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>112</td>
</tr>
<tr>
<td>2</td>
<td>predict</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>117</td>
</tr>
<tr>
<td>3</td>
<td>error</td>
<td>3</td>
<td>0</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>stderr</td>
<td>4</td>
<td>0</td>
<td>1</td>
<td>5.0408057116</td>
</tr>
<tr>
<td>5</td>
<td>lcl</td>
<td>5</td>
<td>0</td>
<td>1</td>
<td>107.12020235</td>
</tr>
<tr>
<td>6</td>
<td>ucl</td>
<td>6</td>
<td>0</td>
<td>1</td>
<td>126.87979765</td>
</tr>
<tr>
<td>7</td>
<td>AIR</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>118</td>
</tr>
<tr>
<td>8</td>
<td>predict</td>
<td>2</td>
<td>0</td>
<td>2</td>
<td>124</td>
</tr>
<tr>
<td>9</td>
<td>error</td>
<td>3</td>
<td>0</td>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td>10</td>
<td>stderr</td>
<td>4</td>
<td>0</td>
<td>2</td>
<td>5.0408057116</td>
</tr>
<tr>
<td>11</td>
<td>lcl</td>
<td>5</td>
<td>0</td>
<td>2</td>
<td>114.12020235</td>
</tr>
<tr>
<td>12</td>
<td>ucl</td>
<td>6</td>
<td>0</td>
<td>2</td>
<td>133.87979765</td>
</tr>
<tr>
<td>13</td>
<td>AIR</td>
<td>1</td>
<td>0</td>
<td>3</td>
<td>132</td>
</tr>
<tr>
<td>14</td>
<td>predict</td>
<td>2</td>
<td>0</td>
<td>3</td>
<td>135</td>
</tr>
<tr>
<td>15</td>
<td>error</td>
<td>3</td>
<td>0</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>16</td>
<td>stderr</td>
<td>4</td>
<td>0</td>
<td>3</td>
<td>5.0408057116</td>
</tr>
<tr>
<td>17</td>
<td>lcl</td>
<td>5</td>
<td>0</td>
<td>3</td>
<td>125.12020235</td>
</tr>
<tr>
<td>18</td>
<td>ucl</td>
<td>6</td>
<td>0</td>
<td>3</td>
<td>144.87979765</td>
</tr>
</tbody>
</table>

The sequence displayed in **Output 6.2.1** 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 6.2.1 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

classes summary
  TFA package, 78
Syntax Index

GetXSeries method, 18

ACCUMULATE function, 107
ACCUMULATE2 function, 109
ACF function, 111
AddARPoly method, 165
AddFrom method, 33
AddMAPoly method, 165
AddSeries method, 9
AddTF method, 166
AddTFDenPoly method, 167
AddTFNumPoly method, 166
AddX method, 7, 153
AddY method, 7
ARIMASpec object, 160
ARMAORDERS function, 113

ACF function, 111
ACF function, 111
AddARPoly method, 165
AddFrom method, 33
AddMAPoly method, 165
AddSeries method, 9
AddTF method, 166
AddTFDenPoly method, 167
AddTFNumPoly method, 166
AddX method, 7, 153
AddY method, 7
ARIMASpec object, 160
ARMAORDERS function, 113

CCF function, 115
CLIMITS object, 188
Close method, 23, 34, 157, 164
Collect method, 36, 39, 40
Criterion method, 155
criterion method, 19

DIAGNOSE object, 11
DIAGSPEC object, 20
DPF Class Object, 116

ESMSpec object, 156

FORENG object, 14
FREQ Class Object, 120

GetForecast method, 17, 155
GetModel method, 13
GetSeries method, 10

IACF function, 121
INDIAG object, 50
INESTobject, 52
INFMSG object, 51
Initialize method, 7, 12, 15, 16, 152
INTERMITTENCYTEST function, 123

MOVINGSUMMARY function, 124

NFor method, 19, 155
NModels method, 14
nrows method, 37, 39, 41, 42, 44, 46–48, 50–52
NTid method, 10

Open method, 23, 34, 42, 44, 45, 47–49, 157, 163
OUTCOMP object, 41
OUTDIAG object, 36
OUTTEST object, 37
OUTFMSG object, 48
OUTFOR object, 39
OUTINDEPEND object, 43
OUTMODELINFO object, 44
OUTNV object, 191
OUTSELECT object, 46
OUTSTAT object, 47

PACF function, 127

Replay method, 12, 17, 154
Run method, 12, 16, 17, 154

SCALE function, 128
SEASONALDECOMP function, 130
SEASONALINDICES function, 131
SEASONTEST function, 133
SELSPEC object, 31
SetARIMAX method, 25
SetARIMAXRefine method, 25
SetDiagnose method, 34
SetDiff method, 164
SetESM method, 26
SetIDM method, 27
SetOption method, 10, 13, 18, 24, 35, 153, 159, 168
SetParm method, 158
SetSpec method, 12
SetTFTransform method, 167
SetTransform method, 28, 158, 164
SetTrend method, 28
SetUCM method, 29
SetUCMRefine method, 30
SetY method, 153
SIMILARITY function, 135
STATIONARITYTEST function, 138

TRANSFORM function, 139
TSDF object, 6
TSM object, 151
TSMFor object, 171
TSMINEST object, 174
TSMINSPEC object, 174
TSMPEST object, 169
TSMSpec object, 170
UNBIASEDNESS function, 141
UTLSTAT object, 193

WHITENOISE function, 143
Gain Greater Insight into Your SAS® Software with SAS Books.

Discover all that you need on your journey to knowledge and empowerment.

support.sas.com/bookstore for additional books and resources.