SAS® Forecast Server 14.3 Time Series Packages

The correct bibliographic citation for this manual is as follows: SAS Institute Inc. 2017. *SAS® Forecast Server 14.3: Time Series Packages*. Cary, NC: SAS Institute Inc.

SAS® Forecast Server 14.3: Time Series Packages

Copyright © 2017, SAS Institute Inc., Cary, NC, USA

All Rights Reserved. Produced in the United States of America.

For a hard-copy book: No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, or otherwise, without the prior written permission of the publisher, SAS Institute Inc.

For a web download or e-book: Your use of this publication shall be governed by the terms established by the vendor at the time you acquire this publication.

The scanning, uploading, and distribution of this book via the Internet or any other means without the permission of the publisher is illegal and punishable by law. Please purchase only authorized electronic editions and do not participate in or encourage electronic piracy of copyrighted materials. Your support of others' rights is appreciated.

U.S. Government License Rights; Restricted Rights: The Software and its documentation is commercial computer software developed at private expense and is provided with RESTRICTED RIGHTS to the United States Government. Use, duplication, or disclosure of the Software by the United States Government is subject to the license terms of this Agreement pursuant to, as applicable, FAR 12.212, DFAR 227.7202-1(a), DFAR 227.7202-3(a), and DFAR 227.7202-4, and, to the extent required under U.S. federal law, the minimum restricted rights as set out in FAR 52.227-19 (DEC 2007). If FAR 52.227-19 is applicable, this provision serves as notice under clause (c) thereof and no other notice is required to be affixed to the Software or documentation. The Government's rights in Software and documentation shall be only those set forth in this Agreement.

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

September 2017

SAS® and all other SAS Institute Inc. product or service names are registered trademarks or trademarks of SAS Institute Inc. in the USA and other countries. ® indicates USA registration.

Other brand and product names are trademarks of their respective companies.

SAS software may be provided with certain third-party software, including but not limited to open-source software, which is licensed under its applicable third-party software license agreement. For license information about third-party software distributed with SAS software, refer to http://support.sas.com/thirdpartylicenses.

Contents

Chapter 1
Introduction to Packages for the TIMEDATA Procedure

Contents

Overview

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

Table 1.1 shows the packages that you can use in the TIMEDATA procedure. Each package has a unique abbreviation that contains up to five alphabetic characters.

Table 1.1 Packages for the TIMEDATA Procedure

Package Name	Abbreviation	Description
Time Frequency Analysis	TFA	Tools for efficient analysis of time series in both the time domain and the frequency domain
Time Series Analysis	TSA	Tools for efficient statistical analysis of time series (transformations, decompositions, statistical tests for intermittency, seasonality, stationarity, and forecast bias, and so on)

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

How Do You Use Packages?

You use a package by specifying its name in a REGISTER statement in the TIMEDATA procedure. The REGISTER statement enables your custom SAS code to access the functions in the package that is specified in the statement. You can specify multiple REGISTER statements in a single TIMEDATA procedure call. However, you can specify only a single package per REGISTER statement, as follows.

REGISTER *package-1* **;**

REGISTER *package-2* **;**

REGISTER *package-n* **;**

For more information, see the REGISTER statement in *SAS/ETS User's Guide*.

Chapter Organization

This book is organized as follows.

Chapter 1, this chapter, provides an overview of packages that you can use in the TIMEDATA procedure.

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

- The "Overview" section describes the broad purpose of the package; it describes which specific facet of a time series analysis problem the package is designed to tackle.
- The "Functional Summary" section provides a table that lists all the functions that are offered by the package along with a brief description of their functionalities. You can use the contents of this table to quickly locate a function that offers the functionalities you need.
- Each of the subsequent sections then details a specific function. Each of these sections is divided into four parts:
  - The "Required Arguments" section describes the function arguments that must always be specified for the function to work properly. A function returns a negative status code when any of its required arguments is not specified.
  - The "Optional Arguments" section describes the function arguments that are auxiliary. Optional arguments provide more control over the behavior of a function. If you choose not to specify an optional argument, simply type the comma that separates it from its surrounding arguments.
  - The "Returned Values" section describes the status codes that are returned by a function and defines their meanings. You can use the contents of this section to quickly uncover the source of any failures in your SAS code.
  - The "Example" section provides a brief example of the use of the function in the TIMEDATA procedure.

- The “Details” section provides detailed descriptions of the mathematical techniques that are used internally by a function. Refer to this section in order to gain a better understanding of the inner workings of a function.
- The “Examples” section follows the detailed description of all the functions in package. This section includes various examples that depict how the various functions provided by a package interact with each other and with functions from other packages to solve complex problems. These examples can provide a good starting point for your own custom SAS code implementations.
- The “References” section contains references for the methodology and for examples of the procedure.

Status Codes

All functions in the TIMEDATA procedure packages return a status code upon their execution. Negative status codes indicate the occurrence of computational failures. Positive return codes, including 0, are customized to the needs of individual functions. Generally, a status code of 0 indicates unconditional success, and a status code greater than 0 indicates conditional success.

Chapter 2
Time-Frequency Analysis Package for the TIMEDATA Procedure

Contents

Overview

Time-frequency analysis refers to techniques that analyze a time series in both time and frequency domains. The time-frequency analysis (TFA) package provides functions that enable you to perform time-frequency analysis within SAS as part of the programming statements in the TIMEDATA procedure in SAS/ETS software.

NOTE: Each function in this chapter is a method of the TFA class object.

Functional Summary

Table 2.1 summarizes the functions in the TFA package.

Table 2.1 Functions in the Time-Frequency Analysis Package

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

FFT Function

rc = **TFA.FFT** *(y, 'sign', output)* **;**

The FFT function computes the discrete Fourier transform of a real time series. Given an input array $x = \begin{pmatrix} x[0] & x[1] & \dots & x[n-1] \end{pmatrix}$ and $s = \pm 1$, the output is $y = \begin{pmatrix} y[0] & y[1] & \dots & y[n-1] \end{pmatrix}$, where

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

When $s = -1$, the output y is the *forward* discrete Fourier transform of x; when $s = 1$, the output y is the *backward* discrete Fourier transform of x.

Required Arguments

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

y specifies an array of real-valued time series. Any missing value in this array is replaced by 0 before computation.

'*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.

Returned Values

The FFT function returns the following values:

rc returns one of the following scalar return codes:

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

output returns the discrete Fourier transform of the input time series as an array with two columns: the first column represents the real part of the output time series, and the second column represents the imaginary part of the output time series.

Example

The following statements read and plot the yearly sunspot count data since 1900:

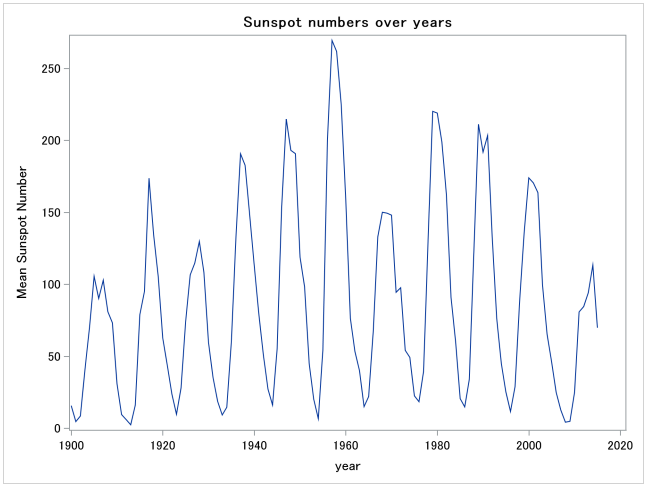
```
data sunspot;
   input x@@;
   year = _N_ + 1900 - 1;
   datalines;
15.7 4.6 8.5 40.8 70.1 105.5 90.1 102.8 80.9 73.2 30.9 9.5 6.0 2.4
16.1 79.0 95.0 173.6 134.6 105.7 62.7 43.5 23.7 9.7 27.9 74.0 106.5
114.7 129.7 108.2 59.4 35.1 18.6 9.2 14.6 60.2 132.8 190.6 182.6
148.0 113.0 79.2 50.8 27.1 16.1 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
;
```

The following statements call the FFT function to compute the forward discrete Fourier transform of the sunspot count data and store the result in the work.fft_x data set.

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

Output 2.1 shows the results.

Figure 2.1 Sunspot Numbers over Years



The following statements compute the discrete Fourier transform of the sunspot data:

```
proc timedata data=sunspot out=_null_;
   var x;
   register tfa;
   declare object obj(tfa);
   array fft_x[1]/nosymbols;
   rc = obj.fft(x, 'forward', fft_x); if rc then stop;
   rc=write_array('fft_x', fft_x); if rc then stop;
run;
```

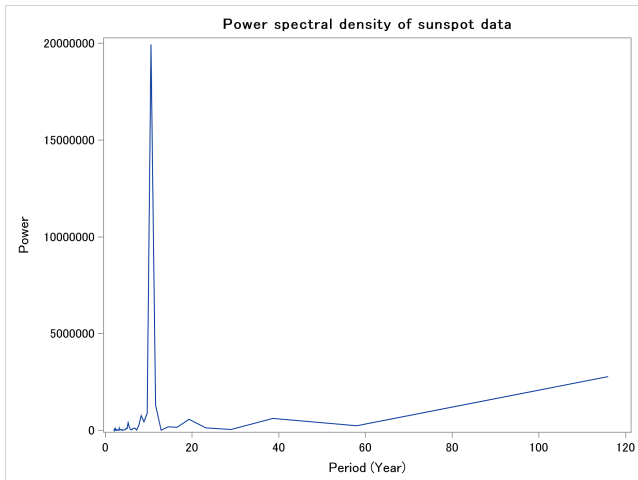
The following statements plot the power spectral density. You can see a spike around 10 years.

```
data power;
   keep Freq Period Power;
   set fft_x nobs=n;
   Power = fft_x1 ** 2 + fft_x2 ** 2;
   Freq = (_n_-1)/n; /* Unit Hz */
   if Freq > 0;
   Period = 1/Freq;
   output;
   if Freq > 0.5 then stop;
run;
```

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

Output 2.2 shows the results.

Figure 2.2 Power Spectral Density of Sunspot Data



FFTC Function

rc = **TFA.FFTC** *(z_re, z_im, 'sign', output)* **;**

The FFT function computes the discrete Fourier transform of a complex time series. Given an input array $z = \begin{pmatrix} z[0] & z[1] & \ldots & z[n-1] \end{pmatrix}$ and $s = \pm 1$, the output is $y = \begin{pmatrix} y[0] & y[1] & \ldots & y[n-1] \end{pmatrix}$, where

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

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

Required Arguments

You must specify the following arguments, separated by commas:

z_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.

z_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.

'*sign*' specifies the sign of the discrete Fourier transform. You can specify the following values within single or double quotation marks:

FORWARD calculates a forward discrete Fourier Transform ($s = -1$).

BACKWARD calculates a backward discrete Fourier Transform ($s = 1$).

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

Returned Values

The FFTC function returns the following values:

rc returns one of the following scalar return codes:

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

output returns the discrete Fourier transform of the input time series as an array with two columns: the first column represents the real part of the discrete Fourier transform, and the second column represents the imaginary part of the discrete Fourier transform.

Example

The following example runs the FFTC function once on the columns named x_re and x_im of the SAS table x and then runs the function again on two input arrays named y_re and y_im:

```
data x;
   input x_re x_im;
   datalines;
1 -1
2 -2
3 -3
4 -4
;
run;


proc timedata data=x out=_null_;
   var x_re x_im;
   register tfa;
   declare object obj(tfa);
   /* Call FFTC on x_re, x_im column of SAS table x */
   array z1[1]/nosymbols;
```

```
    rc = obj.fftc(x_re, x_im, 'forward', z1); if rc then stop;
    rc = write_array('z1', z1); if rc then stop;
    /* Call FFTC on user defined input arrays y_re, y_im */
    array y_re[4]/nosymbols ( 1  2  3  4);
    array y_im[4]/nosymbols (-1 -2 -3 -4);
    array z2[1]/nosymbols;
    rc = obj.fftc(y_re, y_im, 'forward', z2); if rc then stop;
    rc = write_array('z2', z2); if rc then stop;
  run;
```

HILBERT Function

rc = **TFA.HILBERT** *(y, output)* **;**

The HILBERT function 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.

Required Arguments

You must specify the following argument:

y specifies an array that represents an input time series. Any missing value is replaced with 0.

Returned Values

The HILBERT function returns the following values:

rc returns one of the following scalar return codes:

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

output returns the analytic signal that corresponds to the input time series as an array with two columns: the first column represents the real part of the analytic signal time series and the second column represents the imaginary part of the analytic signal, which is the Hilbert transform of the input time series.

Example

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

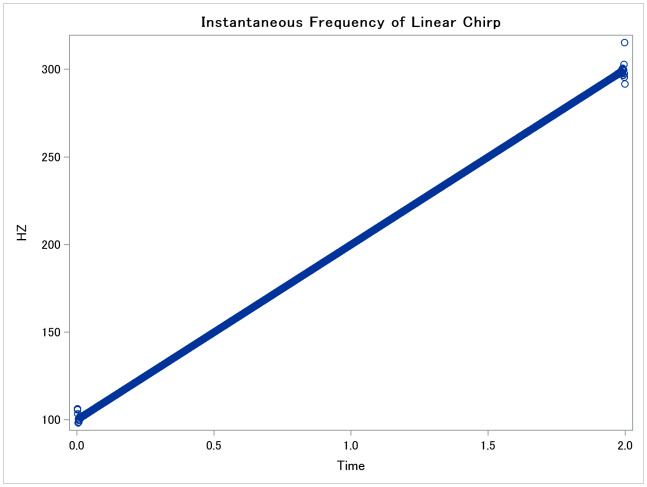
to interpret $\dot{\theta}(t)$ as the "instantaneous frequency" of $x(t)$, and the Hilbert transform can be used to extract the instantaneous frequency of the linear chirp (Boashash 1992a, b). The following example runs the HILBERT function to compute the instantaneous frequency of a linear chirp:

```
/* Generate a chirp with linear instantaneous frequency.
   The chirp is sampled at 1 kHz for 2 seconds. The instantaneous
   frequency is 100 at t = 0 and 200 Hz at t = 1. */
data 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;
run;


/* Compute the Hilbert transform of the chirp data and use it
   compute the instantaneous frequency */
proc timedata data=chirp out=_null_;
   var x;
   register tfa;
   declare object tfa(tfa);
   array analytic[1,1] / nosymbols;
   rc = tfa.hilbert(x, analytic); if rc then stop;
   rc = write_array('analytic', analytic, 'real', 'imaginary');
   array angle[2000] / nosymbols;
   do i = 1 to 2000;
      angle[i] = atan2(analytic[i,2], analytic[i,1]);
   end;
   /* Delete boundary values angle[2] - angle[1] */
   array diff[1998] /nosymbols;
   do i = 1 to 1998;
      diff[i] = angle[i+2] - angle[i+1];
      /* Corrects the radian phase angles by adding 2*pi when diff's absolute
         value is larger than pi */
      if diff[i] < - constant('pi') then diff[i] = diff[i] + 2*constant('pi');
      if diff[i] > constant('pi') then diff[i] = diff[i] - 2*constant('pi');
   end;
   array instfreq[1998,2] /nosymbols;
   do i = 1 to 1998;
      instfreq[i,1] = (i+1)*(1/1000);
      instfreq[i,2] = 1000/(2*constant('pi'))*diff[i];
   end;
   rc = write_array('instfreq', instfreq, 'time', 'instfreq');
run;

proc sgplot data=instfreq;
   title "Instantaneous Frequency of Linear Chirp";
   scatter x=time y=instfreq;
   xaxis label="Time";
```

```
    yaxis label="HZ";
  run;
```

Output 2.3 shows the results.

Figure 2.3 Instantaneous Frequency of Linear Chirp



PWV Function

rc = **TFA.PWV(***y*, *window*, < *overlap* >, < *fftlen* >, < *center* >, < *nthreads* >, < *hilbert_tsf* >, < *fade* >, *output***)** ;

The PWV function computes the pseudo-Wigner-Ville distribution of a real time series.

Required Arguments

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

y specifies an array of input time series. The length of this series is denoted in the rest of this section as series_length. Any missing value in this array is replaced with 0 before computation.

window specifies an array of numbers that contains the window values to be used for computation of 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, separated by commas. If you want to use the default value for an argument, enter a space for it.

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 = window_length.

fftlen specifies the length of the vector on which the discrete Fourier transform will 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 you set *fftlen* to a power of two in order to speed up the computation. The default value is the maximum of 256 and window_length.

center specifies whether the input time series is centered. You can specify the following values for *center*:

0 does not center the input time series.

1 subtracts the mean of the entire series from each term of the input series before calculating the PWV distribution. The missing values of the input time series are excluded in the calculation of the mean.

The default value of *center* is 0.

nthreads specifies the number of threads to use. The value of *nthreads* must be a nonnegative integer and must not be larger than 128. This argument along with the value of k (see the description of the *output* return argument) determines the number of threads that are used as follows:

- If the value *nthreads* is strictly positive, then the PWV function attempts to perform the computation using *nthreads* threads.
- If the value of *nthreads* is larger than k, the computation uses only one thread.
- If the value of *nthreads* is 0, then the number of threads that are used is equal to the number of available CPUs.

The default value of *nthreads* is 1.

hilbert_tsf specifies whether to replace the input signal by an analytical signal. You can specify the following values for *hilbert_tsf*:

0 does not replace the input series.

1 replaces the input series by its analytic signal before further computations if the value of *center* is 0; otherwise (the value of *center* is 1), replaces the original signal by the analytic signal that corresponds to the centered signal.

The default value of *hilbert_tsf* is 1.

fade determines the size and alignment of the output. For a description of the impact of this parameter on computation and output size, see section "Pseudo-Wigner-Ville Distribution" on page 24. The default value of *fade* is 1.

Returned Values

The PWV function returns the following values:

rc returns one of the following scalar return codes:

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

output returns the pseudo-Wigner-Ville transform of the input time series. The output is a two-dimensional array of dimension (*fftlen* × k) × 3, where

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

The columns of *output* from left to right are as follows:

1. The first column stores the normalized frequency index. The frequency index starts from *0* and increments in multiples of $\dfrac{1}{2 \times \textit{fftlen}}$.
2. The second column stores the time index. The time index starts from *0* and increments in multiples of window_length – *overlap*.
3. The third column stores the value of the pseudo-Wigner-Ville distribution that corresponds to the current time index and the current normalized frequency index.

Example

The following example calls the PWV function on column y of the data set a:

```
 data a;
   input  y;
   datalines;
   3
   1
   4
   2
;


proc timedata data=a out=_null_;
   var y;
   array window[1]/nosymbols;
   array pwv[1,1]/nosymbols;
   register TFA;
   declare object TFA(TFA);
   /*--- create a hamming window of size 3 ---*/
   rc = TFA.WINDOW('hamming',3,,window); if rc then stop;
   rc = TFA.PWV(y,window,2,4,0,1,1,1,pwv); if rc then stop;
   /*--- write the output to a sas dataset, give appropriate name to the columns ---*/
```

```
   rc = write_array('pwv', pwv, 'frequency', 'time', 'pwv');
run;
```

STFT Function

rc = **TFA.STFT** *(y, window, overlap, fftlen, center, nthreads, output)* **;**

The STFT function 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 25.

Required Arguments

You must specify the following argument:

y specifies an array of input time series. The length of this series is denoted in the rest of this section as series_length. Any missing value in this array is replaced with 0 before computation.

Optional Arguments

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

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 minimum of 256 and the series length. The length of the window is denoted as window_length in the rest of this section.

overlap specifies the extent of overlap between two consecutive windows, where *overlap* must be an integer that is strictly less window_length. The default value is $\lfloor \frac{n}{2} \rfloor$, where n = window_length.

fftlen specifies the length of the vector on which the discrete Fourier transformation will be done, where *fftlen* must be a positive integer and must be at least as large as window_length. The default value is the maximum of 256 and window_length. It is recommended that *fftlen* be set to 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 larger than 128. The number of threads that are used is determined by the values of *nthreads* and *k* (see the description of the *output* return argument).

- If the value of *nthreads* is strictly positive, then the function 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.

Returned Values

The STFT function returns the following values:

rc returns one of the following scalar return codes:

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

output returns the short-time Fourier transform of input time series as a two-dimensional array of dimension (*fftlen* × k) × 7, where $\mathsf{k} = \left\lfloor \frac{\mathsf{series\_length} - \mathit{overlap}}{\mathsf{window\_length} - \mathit{overlap}} \right\rfloor$. The columns of *output* from left to right are as follows:

1. The first column contains values of the normalized frequency index. The frequency index starts from *0* and increases in increments of $\frac{1}{\mathit{fftlen}}$.
2. The second column contains the time index. The time index starts from *0* and increases in increments of window_length – *overlap*.
3. The third column contains the power of the time series that corresponds to the current normalized frequency index and the current time index. Its value is given 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).
4. The fourth column contains the amplitude that corresponds to the current normalized frequency index and the current time index. It is given by $\sqrt{\text{power}}$, where power is the value stored in the third column.
5. The fifth column stores the phase of the time series that corresponds to the current normalized frequency index and the current time index. It is computed as $\mathrm{atan2}(y, x)$ where x is the real part of the time series (stored in the sixth column) and y is the imaginary part of the time-series (stored in the seventh column).
6. The sixth column contains the real part of the Fourier coefficient.
7. The seventh column contains the imaginary part of the Fourier coefficient.

Example

The following example calls the STFT function on the column *y* of data a:

```
 data a;
   input  y;
   datalines;
   3
   1
   4
   2
;


proc timedata data=a out=_null_;
   var y;
   array window[1]/nosymbols;
   array stft[1,1]/nosymbols;
   register TFA;
   declare object TFA(TFA);
   /*--- create a hamming window of size 3 ---*/
   rc = TFA.WINDOW('hamming',3,,window); if rc then stop;
   rc = TFA.STFT(y,window,2,4,0,1,stft); if rc then stop;
   /*--- write the output to a sas dataset, give appropriate name to the columns ---*/
   rc = write_array('stft', stft, 'frequency', 'time', 'power',
            'amplitude', 'phase', 'coef_re', 'coef_im');
run;
```

WINDOW Function

rc = **TFA.WINDOW** *(<'name'>, length, <params>, output)* **;**

The WINDOW function creates a window of a requested type and length. The WINDOW function is useful for smoothing spectra. For more information about the window functions that are implemented in the time-frequency analysis package, see Harris (1978).

Required Arguments

You must specify the following argument:

length specifies the length of the requested window.

Optional Arguments

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

'*name*' specifies the type of window. The default is a HANNING window.

In the description of the following window functions, N denotes the length of the window, and the N values that define a window are given by $w[0], \ldots, w[N-1]$. Some windows need additional parameters.

You can specify the following values of *name* within single or double quotation marks:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

The last half of the window is defined by symmetry.

params specifies an array of numbers that contain parameters to be used for creating windows. Not all windows require parameters. When *params* is missing, the default value of *params* is used for any window type that needs a parameter.

Returned Values

The WINDOW function returns the following values:

rc returns one of the following scalar return codes:

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

output returns an array of numbers that contain the window.

Example

The following DATA step creates a dummy data set:

```
data dummy;
dummy=1;
run;
```

The following statements create a Hanning window of length 5 and write it to output:

```
/*-----------*/
proc timedata data=dummy out=_null_;
var dummy;
array window[1]/nosymbols;
register TFA;
declare object TFA(TFA);
rc = TFA.WINDOW('hanning',5,,window);
put window[*]=;
run;
```

Details

Discrete Fourier Transforms

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

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

That is, $y = Wx$, where $\omega = \exp -\frac{2\pi \mathbf{i}}{n}$ and $W = (\omega^{(i-1)(j-1)})_{i,j=0}^{n-1,n-1}$. The backward discrete transform of x can be obtained from the matrix vector product W^*x, where W^* denotes the conjugate transpose of W. W satisfies the relation $WW^* = W^*W = nI$, where I is the identity matrix, thus explaining the inverse relation between the forward and the backward transforms. Even though naive computation of Wx takes $\mathcal{O}(n^2)$ operations, the time-frequency analysis package implementation belongs to class of algorithms known as fast Fourier transforms, which exploit the structure of W to compute the discrete Fourier transform of x in $\mathcal{O}(n \log n)$ operations. For more information, see Van Loan (1992).

Hilbert Transformation

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

For a continuous time series $x(t)$ with Fourier transform $X(f) = \int x(t)\exp(-2\pi\mathbf{i}ft)\,dt$, 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 $\mathrm{E} = \int_{-\infty}^{\infty} |x(t)|^2\,dt = \int_0^{2\pi} |X(f)|^2\,df$ as the "total energy" in $x(t)$, and consequently you can interpret $\int_{t_0}^{t_1} |x(t)|^2\,dt$ as the total energy of $x(t)$ between the time points t_0 and t_1. Similarly you can interpret $\int_{f_0}^{f_1} |X(f)|^2\,df$ to be the energy of $x(t)$ between the frequencies f_0 and f_1. This also implies that $|x(t)|^2 = \lim_{h\to 0}\frac{1}{h}\int_t^{t+h} |x(u)|^2\,du$ can be considered to be the instantaneous energy per unit time at time t and $|X(f)|^2$ can be considered to be the instantaneous energy per unit frequency at f.

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

Because energy is positive, you should expect that $P_x(t, f) \geq 0$ for all t and f; this condition is known as positivity. Similarly you should expect $\int_0^{2\pi} P_x(t, f)\,df$ to yield the instantaneous energy at time t, so you should have $\int_0^{2\pi} P_x(t, f)\,df = |x(t)|^2$. Similarly you should expect $\int_{-\infty}^{\infty} P_x(t, f)\,dt = |X(f)|^2$. These two conditions are known as the marginal conditions. Other desirable properties of time-frequency distributions are discussed in Cohen (1995, chapter 6). However, it is not possible for a distribution to simultaneously satisfy positivity and the marginal conditions, and most time-frequency distributions that are used in practice satisfy these conditions only approximately.

The time-frequency analysis package implements the discrete versions of two widely used time-frequency distributions: the pseudo-Wigner-Ville distribution and the short-time Fourier transform. The continuous

version of pseudo Wigner-Ville distribution satisfies the marginal property in the special case when it reduces to the Wigner-Ville distribution, but it does not satisfy the positivity condition. The short-time-Fourier transform satisfies the positivity condition, but it does not satisfy the marginal conditions.

Pseudo-Wigner-Ville Distribution

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

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

It can be shown that the $W_x(t,f)$ is real even when $x(t)$ takes complex values.

The pseudo-Wigner-Ville distribution is a modification of Wigner-Ville distribution, that is obtained by an additional term in the defining integral. The pseudo-Wigner-Ville distribution of a continuous time series $x(t)$ is given by

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

where $w(\tau)$ is a window function.

The Wigner distribution of discrete time series $x[k]$ is defined as follows (Claasen and Mecklenbräuker 1980b, a; Debnath 2002):

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

From the preceding formula, it follows that $W_n(n,f/2)$ is the discrete-time Fourier transform of $x[n+k]\overline{x[n-k]}$ and provides the basis for the computation here. This also explains why the normalized frequency in the PWV output varies from *0* to $1/2$.

Given an input time series, $x(t)$, the computation can be considered as the evaluation of a function $U_x(n,f)$, which measures the value of the pseudo-Wigner-Ville distribution at time n and frequency $f/2$ for different values of n and f. Now $U_x(n,f)$ can be defined: given a possibly complex-valued time series $x=(x[0],x[2],\ldots,x[L-1])$ and a window of odd length $2m+1$, where *window*=$(w[0],\ldots,w[2m])$, define

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

The preceding summation is performed with the following convention: any term in the summation for which both $n+k$ and $n-k$ do not lie between *0* and series_length – 1 is replaced with 0.

Define the following:

- $S =$ window_length $-$ *overlap*

- Let

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

- Let

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

where the window_length = 2L + 1.

The output of the PWV function consists of the evaluation of $U_x(n, f)$ for $n = c, c+S, c+2S, \ldots, c+kS$ and for $f = 0, \frac{1}{\textit{fftlen}}, \ldots, \frac{\textit{fftlen}-1}{\textit{fftlen}}$.

When *fade* is 0, the output has more observations, but some of the observations correspond to windows where only a portion of the data is available. When *fade* is 1, the output is restricted to windows in which all the data are used.

The pseudo-Wigner-Ville distribution has some undesirable properties. It displays annoying artifacts for multicomponent time series (Cohen 1995), and replacing the input with the analytic signal that corresponds to the input yields better results (Boashash 1988). For this reason, the TFA package provides the *hilbert_tsf* option, which replaces the original series with its analytic signal before computation, and the *center* option, which removes the mean from the series so that an overall mean effect does not show up in the output. Before any computation, the input series is first transformed depending on the value of the *hilbert_tsf* and *center* parameters: First the value of the *center* parameter is checked; if it is 1, then the input series that is replaced by the centered series that is obtained by subtracting the series mean from each term of the series. If the value of the *hilbert_tsf* parameter is also 1, this possibly centered series is replaced by the analytic signal that corresponds to the centered input.

Short-Time Fourier Transform

The short-time Fourier transform (STFT) computations consist of multiple "local" discrete Fourier transform computations. The input time series is divided into multiple contiguous blocks, and their discrete Fourier transforms are computed in succession. The use of window functions makes the spectra smooth.

Given a time series $x[0], x[1], \ldots, x[L-1]$ and a window $w[0], \ldots, w[m-1]$, the computation of STFT can be considered to be the evaluation of a function $S_x(n, f)$, which measures the strength of the frequency f at time n for different values of n and f, where $S_x(n, f)$ is defined as

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

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

Examples

Example 2.1: Plotting All Supported Windows

This example plots all supported windows of length 20. The following DATA step creates a dummy data set:

```
/*------  ----*/
data dummy;
length dummy 8;
dummy = 1;
run;
```

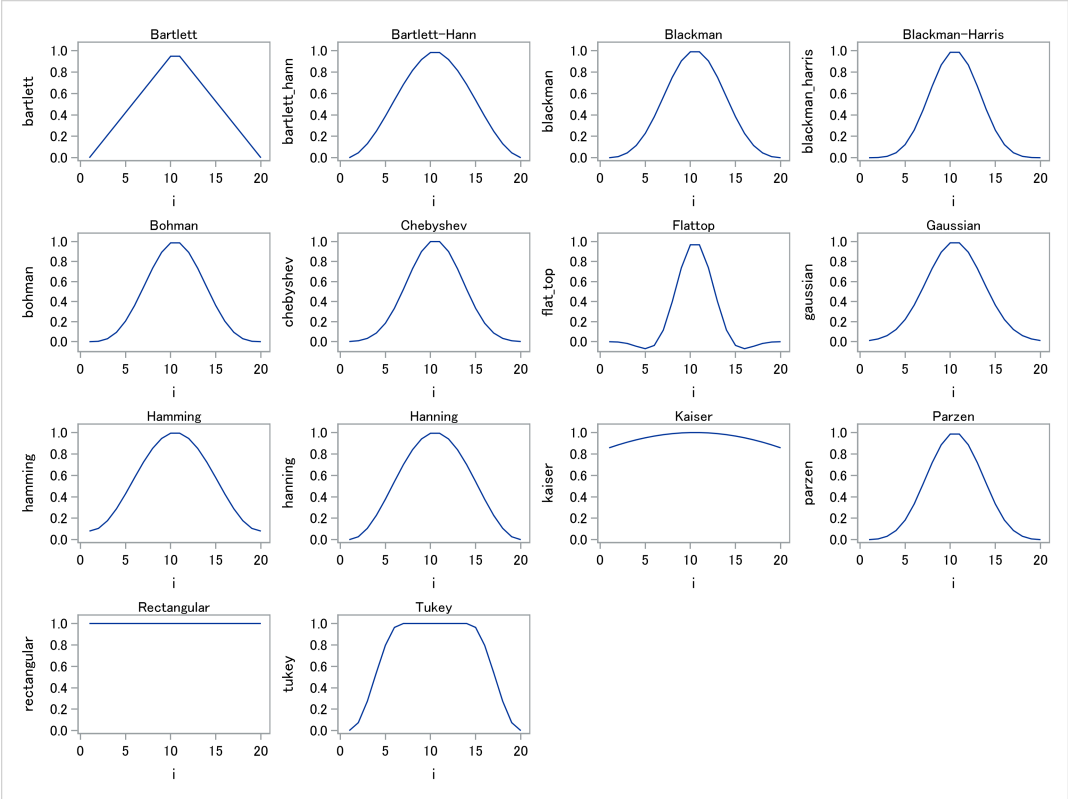
The following statements create and plot windows:

```
proc timedata data=dummy out=_null_;
var dummy;
register tfa;
declare object obj(tfa);

/*---- temporary array storing all the window names ----*/
array window_names[14] $ 24/nosymbols (
         "bartlett"      "bartlett_hann"   "blackman"   "blackman_harris"
         "bohman"        "chebyshev"       "flat_top"   "gaussian"
         "hamming"       "hanning"         "kaiser"     "parzen"
         "rectangular"   "tukey");

/*---parameter list for windows that accept parameters----*/
array cheb_parm[1]/nosymbols (50);
array gaus_parm[1]/nosymbols (3);
array kais_parm[1]/nosymbols (0.8);
array tuke_parm[1]/nosymbols (0.6);

/*-----output will be stored in this temporary array---*/
array window[1]/nosymbols;

/*----all windows will be of this length -----*/
window_length = 20;

do i = 1 to dim(window_names);
    /*---- create the window ----*/
    select(window_names[i]);
       when ("chebysev") rc = obj.window(window_names[i],window_length,cheb_parm,window);
       when ("gaussian") rc = obj.window(window_names[i],window_length,gaus_parm,window);
       when ("kaiser")   rc = obj.window(window_names[i],window_length,kais_parm,window);
       when ("tukey")    rc = obj.window(window_names[i],window_length,tuke_parm,window);
       otherwise         rc = obj.window(window_names[i],window_length,         ,window);
    end;
    if rc then stop;
    /*---- save the windows in a dataset ----*/
    output_ds = strip(window_names[i]) || "_ds";
    rc = write_array(output_ds,window,strip(window_names[i]));
```

```
end;

run;

data window_ds;
   merge bartlett_ds bartlett_hann_ds blackman_ds blackman_harris_ds bohman_ds chebyshev_ds
   flat_top_ds gaussian_ds hamming_ds hanning_ds kaiser_ds parzen_ds
   rectangular_ds tukey_ds;
   obs=_N_;
   label window_bartlett='w[i]' window_bartlett_hann='w[i]'
   window_blackman='w[i]' window_bohman='w[i]' window_chebyshev='w[i]'
   window_flat_top='w[i]' window_gaussian='w[i]' window_hamming='w[i]'
   window_hanning='w[i]'  window_kaiser='w[i]' window_parzen='w[i]'
   window_rectangular='w[i]' window_tukey='w[i]' obs='i';
run;

%macro create_cell_window(window_name,window_col);
      cell;
      cellheader;
      entry "&window_name";
      endcellheader;
      seriesplot x=obs y=&window_col;
   endcell;
%mend;
%macro create_template;
%local win_list win_data wname wcol i;

%let win_list = Bartlett Bartlett-Hann Blackman Blackman-Harris
                Bohman Chebyshev Flattop Gaussian Hamming Hanning
                Kaiser Parzen Rectangular Tukey;
%let win_cols = bartlett bartlett_hann blackman blackman_harris bohman chebyshev
                flat_top gaussian hamming hanning kaiser parzen rectangular tukey;
ods path(prepend) work.templat(update);
proc template;
   define statgraph templat.window;
   begingraph;
   entrytitle '';
   layout lattice / columns=4 columngutter=5 rowgutter=5  columndatarange=unionall
   rowdatarange=union  skipemptycells=true ;
   %do i = 1 %to %sysfunc(countw(&win_list,%str( )));
        %let wname = %scan(&win_list,&i,%str( ));
        %let wcol = %scan(&win_cols,&i,%str( ));
        %create_cell_window(&wname,&wcol);
   %end;
    endlayout;
endgraph;
end;
run;
%mend;
%create_template;

ods graphics / height=800px;
proc sgrender data=window_ds template=templat.window;
run;
```

Output 2.1.1 shows the various windows that are obtained from the preceding code.

Output 2.1.1 Window Functions



References

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

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

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

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

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

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

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

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

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

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

Chapter 3

Time Series Analysis Package for TIMEDATA Procedure

Contents

Overview

The time series analysis (TSA) package contains a set of time series analysis functions that can be used as part of the programming statements in the TIMEDATA 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.

Functional Summary

Table 3.1 summarizes the functions in the TSA package.

Table 3.1 Functions in Time Series Analysis Package

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

ACCUMULATE Function

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

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

Required Arguments

You must specify the following arguments, separated by commas:

time specifies the time ID array for the time series.

y specifies the times series array to accumulate.

'*interval*' specifies the time interval.

You can specify the following values within single quotation marks:

DAY specifies a seasonal cycle of length 7.

MONTH specifies a seasonal cycle of length 12.

QTR specifies a seasonal cycle of length 4.

Optional Arguments

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

'*accumulate*' specifies the accumulation statistic.

You can specify the following values within single quotation marks:

AVERAGE | AVG specifies the average of the values in the time series.

CSS specifies the corrected sum of squares of the values in the time series.

FIRST specifies the first value of the time series.

LAST specifies the last value of the time series.

MAXIMUM | MAX specifies the maximum value in the time series.

MEDIAN | MED specifies the median of the values in the time series.

MINIMUM | MIN specifies the minimum value in the time series.

N specifies the number of nonmissing observations.

NMISS specifies the number of missing observations.

NOBS specifies the number of observations.

STDDEV | STD specifies the standard deviation of the values in the time series.

TOTAL specifies the total sum of the values in the time series.

USS specifies the uncorrected sum of squares of the values in the time series.

The default is TOTAL.

'*setmiss*' specifies the missing value interpretation.

You can specify the following values within single quotation marks:

AVERAGE | AVG specifies the accumulated average value.

FIRST specifies the accumulated first nonmissing value.

MAXIMUM | MAX specifies the accumulated maximum value.

MEDIAN | MED specifies the accumulated median value.

MINIMUM | MIN specifies the accumulated minimum value.

MISSING specifies a missing value.

NEXT specifies the next period's accumulated nonmissing value. Missing values at the end of the accumulated series remain missing.

PREVIOUS | PREV specifies the previous period's accumulated nonmissing value. Missing values at the beginning of the accumulated series remain missing.

The default is MISSING.

'*zeromiss*' specifies the zero value interpretation.

You can specify the following values within single quotation marks:

BOTH sets both beginning and ending zeros to missing.

LEFT sets beginning zeros to missing.

NONE leaves beginning and ending zeros unchanged.

RIGHT sets ending zeros to missing.

The default is NONE.

Returned Values

The ACCUMULATE function returns the following values:

rc returns one of the following scalar return codes:

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

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

z returns the accumulated time series array.

Example

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

```
proc timedata data=sashelp.air out=out outarray=outarray;
   id date interval=month;
   var air;
   outarray qtravg new_id;
   register tsa;
   declare object TSA(tsa);
   rc=TSA.ACCUMULATE(date, air, 'QTR', new_id, qtravg, 'AVERAGE', , );
run;
```

ACCUMULATE2 Function

rc = **TSA.ACCUMULATE2** *(time, y, 'interval', id, z, <'accumulate'>, <'setmiss'>, <'zeromiss'>)* **;**

The ACCUMULATE2 function accumulates a high-frequency time series to a lower frequency and expands the lower-frequency time series to the same length as the high-frequency series.

Required Arguments

You must specify the following arguments, separated by commas:

time specifies the time ID array for the time series.

y specifies the times series array to accumulate.

'*interval*' specifies the time interval.

You can specify the following values within single quotation marks:

DAY specifies a seasonal cycle of length 7.

MONTH specifies a seasonal cycle of length 12.

QTR specifies a seasonal cycle of length 4.

Optional Arguments

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

'*accumulate*' specifies the accumulation statistic.

You can specify the following values within single quotation marks:

AVERAGE | AVG specifies the average of the values in the time series.

CSS specifies the corrected sum of squares of the values in the time series.

FIRST specifies the first value of the time series.

LAST specifies the last value of the time series.

MAXIMUM | MAX specifies the maximum value in the time series.

MEDIAN | MED specifies the median of the values in the time series.

MINIMUM | MIN specifies the minimum value in the time series.

N specifies the number of nonmissing observations.

NMISS specifies the number of missing observations.

NOBS specifies the number of observations.

STDDEV | STD specifies the standard deviation of the values in the time series.

TOTAL specifies the total sum of the values in the time series.

USS specifies the uncorrected sum of squares of the values in the time series.

The default is TOTAL.

'*setmiss*' specifies the missing value interpretation.

You can specify the following values within single quotation marks:

AVERAGE | AVG specifies the accumulated average value.

FIRST specifies the accumulated first nonmissing value.

LAST specifies the accumulated last nonmissing value.

MAXIMUM | MAX specifies the accumulated maximum value.

MEDIAN | MED specifies the accumulated median value.

MINIMUM | MIN specifies the accumulated minimum value.

MISSING specifies a missing value.

NEXT specifies the next period's accumulated nonmissing value. Missing values at the end of the accumulated series remain missing.

PREVIOUS | PREV specifies the previous period's accumulated nonmissing value. Missing values at the beginning of the accumulated series remain missing.

The default is MISSING.

'*zeromiss*' specifies the zero value interpretation.

You can specify the following values within single quotation marks:

BOTH sets both beginning and ending zeros to missing.

LEFT sets beginning zeros to missing.

NONE leaves beginning and ending zeros unchanged.

RIGHT sets ending zeros to missing.

The default is NONE.

Returned Values

The ACCUMULATE2 function returns the following values:

rc returns one of the following scalar return codes:

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

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

z returns the accumulated time series array.

Example

This example uses the TIMEDATA procedure to accumulate a monthly time series into a yearly time series:

```
proc timedata data=sashelp.air out=out outarray=outarray;
   id date interval=month;
   var air;
   outarray yearavg_expand yearid_expand;
   register tsa;
   declare object TSA(tsa);
   rc=TSA.ACCUMULATE2(date, air, 'YEAR', yearid_expand, yearavg_expand,
                      'AVERAGE', , );
run;
```

ACF Function

rc **= TSA.ACF** *(y, nlag, lags, df, < mu >, < acov >, < acf >, < acfstd >, < acf2std >, < acfnorm >, < acfprob > < acflprob >)* **;**

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

Required Arguments

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

y specifies the times series array.

nlag specifies the number of the lag to use in the calculation.

Returned Values

The ACF function returns the following values:

rc returns one of the following scalar return codes:

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

lags returns the number of the lags that were used in the calculation.

df returns the number of observations used to compute *acov* and *acf*.

Optional Returned Values

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

mu returns the mean estimate.

acov returns an array of covariance estimates, with *nlag*+1 entries.

acf returns an array of autocorrelation estimates, with *nlag*+1 entries.

acfstd returns an array of standard errors, with *nlag*+1 entries.

acf2std returns an array of twice standard errors, with *nlag*+1 entries.

acfnorm returns an array of normalized autocorrelation, with *nlag*+1 entries.

acfprob returns an array of autocorrelation probabilities, with *nlag*+1 entries.

acflprob returns an array of autocorrelation log probabilities, with *nlag*+1 entries.

Example

This example uses the TIMEDATA procedure to compute the autocorrelation of lag 3 of the time series air:

```
proc timedata data=sashelp.air out=out outscalars=outscalars
              outarray=outarray print=scalars;
   id date interval=month;
   var air;
   outscalars mu;
   outarrays acf acov lags df acfstd;
   register tsa;
   declare object TSA(tsa);
   rc=TSA.ACF(air, 3, lags, df, mu, acov, acf, acfstd, , , , );
run;
```

ARMAORDERS Function

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

The ARMAORDERS function performs tests to tentatively identify the autoregressive and moving average orders of mixed autoregressive moving average models.

Required Arguments

You must specify the following argument:

y specifies the times series array to test.

Optional Arguments

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

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

'*method*' specifies the method of tentative order selection.

You can specify the following values within single quotation marks:

ESACF specifies the extended sample autocorrelation function.

MINIC specifies the minimum information criterion.

SCAN specifies the squared canonical correlations.

The default value is MINIC.

p specifies the autoregressive order range, where *p* is an array of two nonnegative integers that define the minimum and maximum values.

By default, *p* is the array [0,5].

q specifies the moving average order range, where *q* is an array of two nonnegative integers that define the minimum and maximum values.

By default, *q* is the array [0,5].

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

By default, *perror* is the array [$\max(p)$, $\max(p)+\max(q)$].

Returned Values

The ARMAORDERS function returns the following values:

rc returns one of the following scalar return codes:

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

porders returns the recommended autoregressive orders.

qorders returns the recommended moving average orders.

Example

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

```
proc timedata data=sashelp.air out=out outscalars=outscalars print=scalars;
   id date interval=month;
   var air;
   outscalars porders qorders;
   register tsa;
   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);
run;
```

CCF Function

rc = **TSA.CCF** *(y, x, nlag, lags, df, < ymu > , < xmu >, < ccov >, < ccf >, < ccfstd >, < ccf2std >, < ccfnorm >, < ccfprob >, < ccflprob >)* **;**

The CCF function computes the cross-correlation and cross-covariance for two time series arrays.

Required Arguments

You must specify the following arguments, separated by commas:

y specifies one times series array.

x specifies the other time series array.

nlag specifies the number of the lag to compute.

Returned Values

The CCF function returns the following values:

rc returns one of the following scalar return codes:

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

lags returns an array of lags that were computed, with *nlag*+1 entries.

df returns an array of number of products for which to compute the cross-correlation, with *nlag*+1 entries.

Optional Returned Values

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

ymu returns the mean estimate of input time series *y*.

xmu returns the mean estimate of input time series *x*.

ccov returns an array of cross-covariance estimates, with $2 \times nlag + 1$ entries.

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

ccfstd returns an array of standard errors, with $2 \times nlag + 1$ entries.

ccf2std returns an array of double standard errors, with $2 \times nlag + 1$ entries.

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

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

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

Example

This example uses the TIMEDATA procedure to compute the cross-correlation and cross-covariance of two time series arrays (Price and Sale) with lag 20:

```
proc timedata data=sashelp.pricedata outarray=ccf_array
              outscalar=ccf_scalar;
   id date interval=month;
   var price sale;
   by region line product;
   outscalars ymu xmu;
   outarrays lags df ccov ccf ccfstd ccf2std ccfnorm ccfprob ccflprob;
   register tsa;
   declare object TSA(tsa);
   rc=TSA.CCF(price, sale, 20, lags, df, ymu, xmu, ccov, ccf,
              ccfstd, ccf2std, ccfnorm, ccfprob, ccflprob);
run;
```

COUNTDISTRIBUTION Function

rc = **TSA.COUNTDISTRIBUTION** *(counts, values, <'dist'>, <'select'>, distribution, estimates, <prob>, <predict>, <error>, <lower>, <upper>, <chisq>, <chisqprob>)* **;**

The COUNTDISTRIBUTION function performs count distribution analysis for a time series.

Required Arguments

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

counts specifies the input count series array to analyze.

values specifies the input values series array to analyze.

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.

'*dist*' specifies the type of 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.

'*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.

Returned Values

The COUNTDISTRIBUTION function returns the following values:

rc returns one of the following scalar return codes:

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

distribution returns the selected frequency count distribution.

estimates returns an array of distribution parameters estimates.

The following table shows the returned distributions and their parameters:

Returned Value of *distribution*	**Distribution**	**Parameters Whose Estimates Are Returned in** *estimates*
1	Binomial	*q*
2	Zero-modified binomial	*q*, *zm*
3	Geometric	*beta*
4	Zero-modified geometric	*beta*, *zm*
5	Poisson	*lambda*
6	Zero-modified Poisson	*q*, *zm*
7	Negative binomial	*r*, *beta*
8	Zero-modified negative binomial	*r*, *beta*, *zm*

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.

prob returns an array of the probability density.

predict returns an array of predictions of the frequency series counts.

error returns an array of residuals. Residuals are computed as the difference between the actual counts and predicted values.

lower returns an array of the lower confidence limit.

upper returns an array of the upper confidence limit.

chisq returns an array of the chi-square statistics.

chisqprob returns an array of the chi-square probability.

Example

This example uses the TIMEDATA procedure to find the best fit distribution for the count series nHome:

```
proc timedata data=sashelp.baseball out=out outscalars=outscalars
              outarray=outarray print=scalars;
   var nHome;
   register tsa;
   declare object TSA(tsa);
   outscalar nvalues distribution;
   outarrays values counts estimates lower upper;
   array prob[1]/nosymbols;
   array predict[1]/nosymbols;
   array error[1]/nosymbols;
   array chisq[1]/nosymbols;
   array chisqprob[1]/nosymbols;
   rc = tsa.frequency(nHome, nvalues, values, counts);
   rc1=TSA.COUNTDISTRIBUTION(counts, values, 'POISSON', 'AIC', distribution,
                             estimates, prob, predict ,error , lower, upper,
                             chisq, chisqprob);
   rc3=append_array('prob', prob,'prob');
   rc3=append_array('predict', predict,'predict');
   rc3=append_array('error', error, 'error');
   rc3=append_array('chisq', chisq, 'chisq');
   rc3=append_array('chisqprob', chisqprob, 'chisqprob');
run;
```

FREQUENCY Function

rc **= TSA.FREQUENCY** *(y, < nvalues >, < values >, < counts >)* **;**

The FREQUENCY function analyzes the frequency of a time series: it outputs all unique values and corresponding counts for the time series.

Required Arguments

You must specify the following argument:

y specifies the times series array to analyze.

Returned Values

The FREQUENCY function returns the following values:

rc returns one of the following scalar return codes:

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

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.

nvalues returns the number of unique values.

values returns an array of the unique values.

counts returns an array of count values that correspond to the values returned in *values*.

Example

This example uses the TIMEDATA procedure to analyze the frequency of the time series nHome:

```
proc timedata data=sashelp.baseball out=out outscalars=outscalars
              outarray=outarray print=scalars;
   var nHome;
   register tsa;
   declare object TSA(tsa);
```

```
    outscalar nvalues;
    outarrays values counts;
    rc=TSA.FREQUENCY(nHome, nvalues, values, counts);
run;
```

IACF Function

rc = **TSA.IACF** *(y, nlag, lags, df, < mu >, < iacf >, < iacfstd >, < iacf2std >, < iacfnorm >, < iacfprob >, < iacflprob >)* **;**

The IACF function computes the inverse autocorrelation for a time series array.

Required Arguments

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

y specifies the times series array.

nlag specifies the number of the lag to use in the calculation.

Returned Values

The IACF function returns the following values:

rc returns one of the following scalar return codes:

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

lags returns the number of the lag that was used in the calculation.

df returns the number of observations used to compute *iacf*.

Optional Returned Values

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

mu returns the mean estimate.

iacf returns an array of inverse autocorrelation estimates, with *nlag*+1 entries.

iacfstd returns an array of inverse autocorrelation standard errors, with *nlag*+1 entries.

iacf2std returns an array of twice standard errors, with *nlag*+1 entries.

iacfnorm returns an array of normalized inverse autocorrelation, with *nlag*+1 entries.

iacfprob returns an array of inverse autocorrelation probabilities, with *nlag*+1 entries.

iacflprob returns an array of inverse autocorrelation log probabilities, with *nlag*+1 entries.

Example

This example uses the TIMEDATA procedure to compute the inverse autocorrelation of lag 3 of the time series air:

```
proc timedata data=sashelp.air out=out outscalars=outscalars
              outarray=outarray print=scalars;
   id date interval=month;
   var air;
   outscalars mu;
   outarrays iacf lags df iacfstd;
   register tsa;
   declare object TSA(tsa);
   rc=TSA.IACF(air, 3, lags, df, mu, iacf, iacfstd, , , , );
run;
```

INTERMITTENCYTEST Function

rc = **TSA.INTERMITTENCYTEST** *(y, base, threshold, med)* **;**

The INTERMITTENCYTEST function tests for intermittency of a univariate time series by computing the median of the length of contiguous constant periods (demand intervals).

Required Arguments

You must specify the following arguments, separated by commas:

y specifies the times series array to test. The test is applied to the last 100 values.

base specifies the base value to test. The value is typically 0.

threshold specifies the threshold value for intermittency. The value is typically greater than 2.

Returned Values

The INTERMITTENCYTEST function returns the following values:

rc returns one of the following scalar return codes:

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

med returns the median length of the contiguous constant periods.

Example

This example uses the TIMEDATA procedure to test the intermittency on the time series array Air:

```
proc timedata data=sashelp.air out=out outscalars=outscalars print=scalars;
   id date interval=month;
   var air;
   outscalars intermittent;
   register tsa;
   declare object TSA(tsa);
   intermittent=0;
   rc=TSA.INTERMITTENCYTEST(air, 0, 2, med);
   if rc>0 then intermittent= 1;
run;
```

MOVINGSUMMARY Function

rc = **TSA.MOVINGSUMMARY** *(y, 'method', k, <lead>, <w>, <'setmiss'>, <'abs'>, x, <p>, <nmiss>)* **;**

The MOVINGSUMMARY function computes statistics for a set of values within a moving time window.

Required Arguments

You must specify the following arguments, separated by commas:

y specifies the input time series array.

'*method*' specifies the statistic to calculate for each output array, x_t, based on the elements of the *y* input array in the *t* window.

You can specify the following methods within single quotation marks:

EWMA calculates the exponentially weighted moving average.

GMEAN calculates the moving geometric mean.

MAX calculates the maximum value.

MEAN calculates the moving average.

MED calculates the median value.

MIN calculates the minimum value.

PROD calculates the moving product.

RANGE calculates the maximum value minus minimum value.

SUM calculates the moving sum.

TVALUE calculates the standard deviation divided by mean.

VAR calculates the variance of the sample defined by the window around *t*.

k specifies the window size, where *k* is a positive integer. When the method is EWMA, *k* is set to 1 and defaults are used for all other arguments.

Optional Arguments

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

lead specifies the number of leading terms, where *lead* is a nonnegative integer less than *k*. You can specify the following values:

0 specifies the backward moving summary.

k/2 specifies the centered moving summary.

k–1 specifies the forward moving summary.

The default value is 0. When the method is EWMA, *lead* is set to 0.

w specifies an array of weights that has *k* elements (a scalar when *k*=1). This argument is required for the EWMA method, and it must be a scalar between 0 and 1, inclusive. This argument is optional for the MEAN, PROD, TVALUE, and VAR methods and is not supported for all other methods.

'*setmiss*' specifies how missing values are interpreted.

You can specify the following values within single quotation marks:

IGNORE specifies that missing values have no effect on the summary.

MEAN specifies that missing values are replaced with the mean of the remaining nonmissing values in the window. This value is supported only for the method SUM.

MISSING specifies that if the input window contains a missing value, the output value is also missing.

The default value is IGNORE.

'*abs*' specifies how the series is transformed into nonnegative values prior to performing the moving summary.

You can specify the following values within single quotation marks:

OFF specifies no modification. This value is not supported for the GMEAN method, because the geometric mean is undefined for negative values in the series.

ON transforms each member of the series into its absolute value.

SQUARE transforms each member of the series into its square

The default value is ON when the *method* is GMEAN and is OFF for all other *methods*.

Returned Values

The MOVINGSUMMARY function returns the following values:

rc returns one of the following scalar return codes:

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

x returns the transformed series.

Optional Returned Values

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

p returns an array in which element *t* is the number of products that contributed to element *t* of *x*. The *p* argument is supported only when *method* is PROD or GMEAN.

nmiss returns the number of missing values that are generated.

Examples

This example uses the TIMEDATA procedure to compute the five-period moving average of the time series array air:

```
proc timedata data=sashelp.air out=out outscalar=scalars outarray=arrays;
   id date interval=month;
   var air;
   outarrays x p;
```

```
   outscalars rc nmiss;
   register tsa;
   declare object TSA(tsa);
   rc=TSA.MOVINGSUMMARY(air, 'MEAN' , 5, 0 , , , , x, p, nmiss);
run;
```

This example uses the TIMEDATA procedure to compute the five-period centered weighted moving product of the time series array air:

```
proc timedata data=sashelp.air out=out outscalar = scalars outarray=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;
   register tsa;
   declare object TSA(tsa);
   rc=TSA.MOVINGSUMMARY(air, 'PROD' , 5, 2.5 , w, , , x, , );
run;
```

PACF Function

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

The PACF function computes the partial autocorrelation for a time series array.

Required Arguments

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

y specifies the times series array.

nlag specifies the number of the lag to use in the calculation.

Returned Values

The PACF function returns the following values:

rc returns one of the following scalar return codes:

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

lags returns the number of the lag that was used in the calculation.

df returns the number of observations used to compute *pacf*.

Optional Returned Values

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

mu returns the mean estimate.

pacf returns an array of partial autocorrelation estimates, with *nlag*+1 entries.

pacfstd returns an array of partial autocorrelation standard errors, with *nlag*+1 entries.

pacf2std returns an array of twice standard errors, with *nlag*+1 entries.

pacfnorm returns an array of normalized partial autocorrelation, with *nlag*+1 entries.

pacfprob returns an array of partial autocorrelation probabilities, with *nlag*+1 entries.

pacflprob returns an array of partial autocorrelation log probabilities, with *nlag*+1 entries.

Example

This example uses the TIMEDATA procedure to compute the autocorrelation of lag 3 of the time series air:

```
proc timedata data=sashelp.air out=out outscalars=outscalars
              outarray=outarray print=scalars;
   id date interval=month;
   var air;
   outscalars mu;
   outarrays pacf lags df pacfstd;
   register tsa;
   declare object TSA(tsa);
   rc=TSA.PACF(air, 3, lags, df, mu, pacf, pacfstd, , , , );
run;
```

SCALE Function

rc = **TSA.SCALE** *(y, min, max, nomiss, x, < nmiss >)* **;**

The SCALE function scales a time series between a specified minimum value and a specified maximum value.

Required Arguments

You must specify the following arguments, separated by commas:

y specifies the input time series array.

min specifies the minimum value in the output array.

max specifies the maximum value in the output array.

nomiss specifies how missing values are treated. You can specify the following values:

0 allows missing values in the input array.

1 does not allow missing values in the input array. If missing values exist, the output array x_t becomes missing for all values of t.

The default is 0.

Returned Values

The SCALE function returns the following values:

rc returns one of the following scalar return codes:

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

x returns the transformed series.

Optional Returned Values

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

nmiss returns the number of missing values that are generated.

Example

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

```
proc timedata data=sashelp.air out=out outarray=trans_array;
   var air;
   register tsa;
   declare object TSA(tsa);
   outarrays t1;
   rc=TSA.SCALE(air, 0, 100, , t1, );
run;
```

SEASONALDECOMP Function

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

The SEASONALDECOMP function computes the seasonal indices of a univariate time series by using classical decomposition.

Required Arguments

You must specify the following arguments, separated by commas:

y specifies the times series array to decompose.

s specifies the seasonality to test, where *s* must be either a positive integer or _SEASONALITY_, which is the length of the seasonal cycle as specified by the SEASONALITY= option in the PROC TIMEDATA statement or implied by the INTERVAL= option in the ID statement.

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

You can specify the following values within single quotation marks:

ADD | ADDITIVE specifies additive decomposition.

LOGADD | LOGADDITIVE specifies log-additive decomposition.

MULT | MULTIPLICATIVE specifies multiplicative decomposition.

MULTORADD specifies multiplicative or additive decomposition, depending on data.

PSEUDOADD | PSEUDOADDITIVE specifies pseudo-additive decomposition.

Optional Arguments

You can also specify the following argument, separated by a comma from arguments that precede it. If you want to use a default value for this argument, enter a space for it.

lambda specifies the Hodrick-Prescott filter parameter for trend-cycle decomposition. The default value is 1,600. Filtering applies when the trend component or the cycle component is requested. If filtering is not specified, this option is ignored.

Returned Values

The SEASONALDECOMP function returns the following values:

rc returns one of the following scalar return codes:

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

Optional Returned Values

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

tcc specifies the trend-cycle component.

sic specifies the seasonal-irregular component.

sc specifies the seasonal component.

scstd specifies the seasonal component standard errors.

tcs specifies the trend-cycle-seasonal component.

ic specifies the irregular component.

sa specifies the seasonally adjusted series.

pcsa specifies the percentage of change in seasonally adjusted series.

tc specifies the trend component.

cc specifies the cycle component.

Example

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

```
proc timedata data=sashelp.air out=out outarray=outarray;
   id date interval=month;
   var air;
   outarray ADJUSTED;
   register tsa;
   declare object TSA(tsa);
   rc=TSA.SEASONALDECOMP(air, _SEASONALITY_, 'ADD', , , , , , , , ADJUSTED, , , );
run;
```

SEASONALINDICES Function

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

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

Required Arguments

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

y specifies the times series array.

s specifies the seasonality to test, where *s* must be either a positive integer or _SEASONALITY_, which is the length of the seasonal cycle as specified by the SEASONALITY= option in the PROC TIMEDATA statement or implied by the INTERVAL= option in the ID statement.

Optional Arguments

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

'*mode*' specifies the type of model to be used in the regression.

You can specify the following values within single quotation marks:

ADD | ADDITIVE uses an additive model.

MULT | MULTIPLICATIVE uses a multiplicative model.

The default method is ADD.

'*term*' specifies the type of terms to be used in the regression.

You can specify the following values within single quotation marks:

S uses only seasonal dummies terms.

SC uses only seasonal dummies and constant terms.

ST uses only seasonal dummies and trend terms.

STC uses seasonal dummies, trend, and constant terms.

STQ uses seasonal dummies, trend, and quadratic terms.

STQC uses seasonal dummies, trend, quadratic, and constant terms.

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

Returned Values

The SEASONALINDICES function returns the following values:

rc returns one of the following scalar return codes:

rc	Termination Reason
0	Success
< 0	Computational failure

indices returns an array of seasonal indices.

Example

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

```
proc timedata data=sashelp.air out=out outarray=outarray print=arrays;
   id date interval=month;
   var air;
   outarrays sindices;
   register tsa;
   declare object TSA(tsa);
   rc=TSA.SEASONALINDICES(air, _SEASONALITY_,  'ADD', 'STQC', sindices);
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 TIMEDATA statement or implied by the INTERVAL= option in the ID statement.

Optional Arguments

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

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

p specifies the autoregressive order (0 or 1). The default value is 0.

alpha specifies the significance level. The default value is 0.01.

Returned Values

The SEASONTEST function returns the following values:

rc returns one of the following scalar return codes:

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

Optional Returned Values

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

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

Example

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

```
proc timedata data=sashelp.air out=out outscalars=outscalars
              outarray=outarray print=scalars;
   id date interval=month;
   var air;
   outscalars seasonal;
   outarray aic;
   register tsa;
   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;
  run;
```

SEGMENTATION Function

rc = **TSA.SEGMENTATION** *(y, <'method'>, <'criterion'>, <minnseg>, <maxnseg>, <maxerror>, <maxpcterror>, segindex, segvalue)* **;**

The SEGMENTATION function segments a univariate time series.

Required Arguments

You must specify the following argument:

y specifies the times 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.

'*method*' specifies the type of segment model.

You can specify the following values within single quotation marks:

CUBIC uses a cubic curve as the segment model. For segments that have fewer than four values, a quadratic curve is used.

LANDSCAPE uses a linear regression as the segment model.

LINEAR uses a linear curve as the segment model.

MEAN uses the mean of the series as the segment model.

MEDIAN uses the median of the series as the segment model.

MODE uses the largest mode of the series as the segment model. If there is more than one mode of the same size, the mode closest to the median is used.

QUADRATIC uses a quadratic curve as the segment model. For segments that have fewer than three values, a linear curve is used.

The default value is LANDSCAPE.

'*criterion*' specifies the method for determining the stopping criterion.

You can specify the following values within single quotation marks:

ABSOLUTE bases the stopping criterion on the absolute value.

SQUARED bases the stopping criterion on the squared value.

The default value is ABSOLUTE.

minnseg specifies the minimum number of segments. The number must be a positive integer less than 10,000. The default value is 1, which implies there is no constraint.

maxnseg specifies the maximum number of segments. The number must be a positive integer less than 10,000. The default is the length of the series minus one. The value of this argument cannot be less than the value of the *minnseg* option. By default, there is no constraint.

maxerror specifies the maximum error for determining the stopping criterion. By default, there are no limits on the stopping criterion.

maxpcterror specifies the maximum percentage error for determining the stopping criterion. The default value is 10%.

Returned Values

The SEGMENTATION function returns the following values:

rc returns one of the following scalar return codes:

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

segindex returns an array that indicates the segmentation index that corresponds to each observation in the original time series.

segvalue returns an array that indicates the time series representation value that corresponds to each observation in the original time series.

Example

This example uses the TIMEDATA procedure to compute five segments of the time series air:

```
proc timedata data=sashelp.air out=out outarray=outarray;
   id date interval=month;
   var air;
   outarray segindex segvalue;
   register tsa;
   declare object TSA(tsa);
   rc=0;
   rc=TSA.SEGMENTATION(air, 'MEAN', 'SQUARED', 5, 5, , , segindex, segvalue);
run;
```

SIMILARITY Function

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

The SIMILARITY function analyzes the similarity between two time series.

Required Arguments

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

x specifies the input time series array to be compared to the target time series.

y specifies the target time series array to be compared to the input time series.

Optional Arguments

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

'*type*' specifies the similarity measure.

You can specify the following values within single quotation marks:

ABSDEV specifies the absolute deviation.

MABSDEV specifies the mean absolute deviation.

MABSDEVINP specifies the mean absolute deviation relative to the length of the input sequence.

MABSDEVMAX specifies the mean absolute deviation relative to the maximum valid path length.

MABSDEVMIN specifies the mean absolute deviation relative to the minimum valid path length.

MABSDEVTAR specifies the mean absolute deviation relative to the length of the target sequence.

MSQRDEV specifies the mean squared deviation.

MSQRDEVINP specifies the mean squared deviation relative to the length of the input sequence.

MSQRDEVMAX specifies the mean squared deviation relative to the maximum valid path length.

MSQRDEVMIN specifies the mean squared deviation relative to the minimum valid path length.

MSQRDEVTAR specifies the mean squared deviation relative to the length of the target sequence.

SQRDEV specifies the squared deviation.

The default value is SQRDEV.

'*scale*' specifies how the working input sequence is scaled with respect to the working target sequence. Scaling is performed after normalization.

You can specify the following values within single quotation marks:

ABS applies absolute scaling.

NONE applies no scaling.

STD applies standard scaling.

The default value is NONE.

expandpct specifies the warping expansion as a percentage of the length of the target sequence, where *expandpct* ranges from 0 to 100, 0 implies no compression, and 100 implies maximum allowable compression. The default value is 100.

expandabs specifies the absolute warping expansion, where *expandabs* is an integer that ranges from 0 to 10,000. The default is the maximum allowable absolute expansion.

compresspct specifies the warping compression as a percentage of the length of the target sequence, where *compresspct* ranges from 0 to 100, 0 implies no compression, and 100 implies maximum allowable compression. The default value is 100.

compressabs specifies the absolute warping compression, where *compressabs* is an integer that ranges from 0 to 10,000. The default is the maximum allowable absolute compression.

Returned Values

The SIMILARITY function returns the following values:

rc returns one of the following scalar return codes:

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

measure returns the similarity measure.

Example

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

```
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  .
;

proc timedata data=test out=out2 outscalars=tsa_scalar2;
   register tsa;
   declare object TSA(tsa);
   var x y;
   outscalar measure;
   rc=TSA.SIMILARITY(x, y, 'ABSDEV', 'NONE', 20, 0 , , , measure);
run;
```

STATIONARITYTEST Function

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

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

Required Arguments

You must specify the following argument:

y specifies the times series array to test.

Optional Arguments

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

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

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

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

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

You can specify the following values within single quotation marks:

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

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

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

The default value of *type* is SZM.

Returned Values

The STATIONARITYTEST function returns the following values:

rc returns one of the following scalar return codes:

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

pvalue returns the probability value associated with the test.

Example

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

```
proc timedata data=sashelp.air out=out outscalars=outscalars print=scalars;
   id date interval=month;
   var air;
   outscalars stationary1 stationary2;
   register tsa;
   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;
  run;
```

TRANSFORM Function

rc = **TSA.TRANSFORM** *(y, <'type'>, <inverse>, <c>, x)* **;**

The TRANSFORM function transforms a time series to another form.

Required Arguments

You must specify the following arguments:

y specifies an input time series array.

Optional Arguments

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

type specifies the type of transformation. You can specify the following values within single quotation marks:

LOG specifies logarithmic transformation.

SQRT specifies square root transformation.

LOGIT specifies logit transformation.

BOXCOX specifies Box-Cox transformation.

NONE requests that no transformation be performed.

The default value is NONE.

inverse specifies whether to perform an inverse transformation. You can specify the following values:

0 does not perform an inverse transformation.

1 returns the inverse of the specified transformation method.

The default value is 0.

c specifies a parameter to be used in the transformation. Its use depends on the transformation method as follows:

- For log transformation, c is bias: $x = \log(y + c)$. The default value is 0.

- For square root transformation, c is bias: $x = \log(y + c)$. 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 λ: $x = c^2 + (y^c - 1)/c$. If c is not specified, x=y.

Returned Values

The TRANSFORM function returns the following values:

rc returns one of the following scalar return codes:

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

x returns the transformed series.

Example

This example uses the TIMEDATA procedure to take the log transform of the time series array air:

```
proc timedata data=sashelp.air out=out outarray=trans_array;
   var air;
   register tsa;
   declare object TSA(tsa);
   outarrays t1;
   rc=TSA.TRANSFORM(air, 'LOG', 0, 0, t1);
run;
```

UNBIASEDNESS Function

rc = **TSA.UNBIASEDNESS** *(y, predict, < siglevel >, intercept, scale, fvalue, pvalue)* **;**

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

Required Arguments

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

y specifies the input time series array.

predict specifies an input array of predicted time series.

Optional Arguments

You can also specify the following argument. If you want to use a default value for this argument, enter a space for it.

siglevel specifies the significance level.

Returned Values

The UNBIASEDNESS function returns the following values:

rc returns one of the following scalar return codes:

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

intercept returns the constant parameter.

scale returns the scale parameter.

fvalue returns the test statistic for the F test.

pvalue returns the p-value for the F test.

Example

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

```
proc hpf data=sashelp.air out=_null_ outfor=outfor;
   id date interval=month;
   forecast air;
run;
```

```
proc reg data=outfor;
   model actual=predict;
   test intercept=0, predict=1;
run;
quit;

proc timedata data=outfor out=out outscalars=bias_scalar outarray=bias_array;
   id date interval=month;
   var ACTUAL PREDICT;
   outscalars intercept scale fvalue pvalue;
   register tsa;
   declare object TSA(tsa);
   rc=TSA.UNBIASEDNESS(ACTUAL, PREDICT, 0.05, intercept, scale, fvalue, pvalue);
run;
```

WHITENOISE Function

rc = **TSA.WHITENOISE** *(y, nlag, lags, df, wn, < wnprob >, < wnlprob >)* **;**

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

Required Arguments

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

y specifies the times series array to compute.

nlag specifies the number of the lag to use in the calculation.

Returned Values

The WHITENOISE function returns the following values:

rc returns one of the following scalar return codes:

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

lags returns the number of the lag that was used in the calculation.

df returns the number of observations that were used to test white noise.

wn returns an array of Ljung-Box white noise tests, with *nlag*+1 entries.

Optional Returned Values

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

wnprob returns white noise probabilities.

wnlprob returns white noise log probabilities.

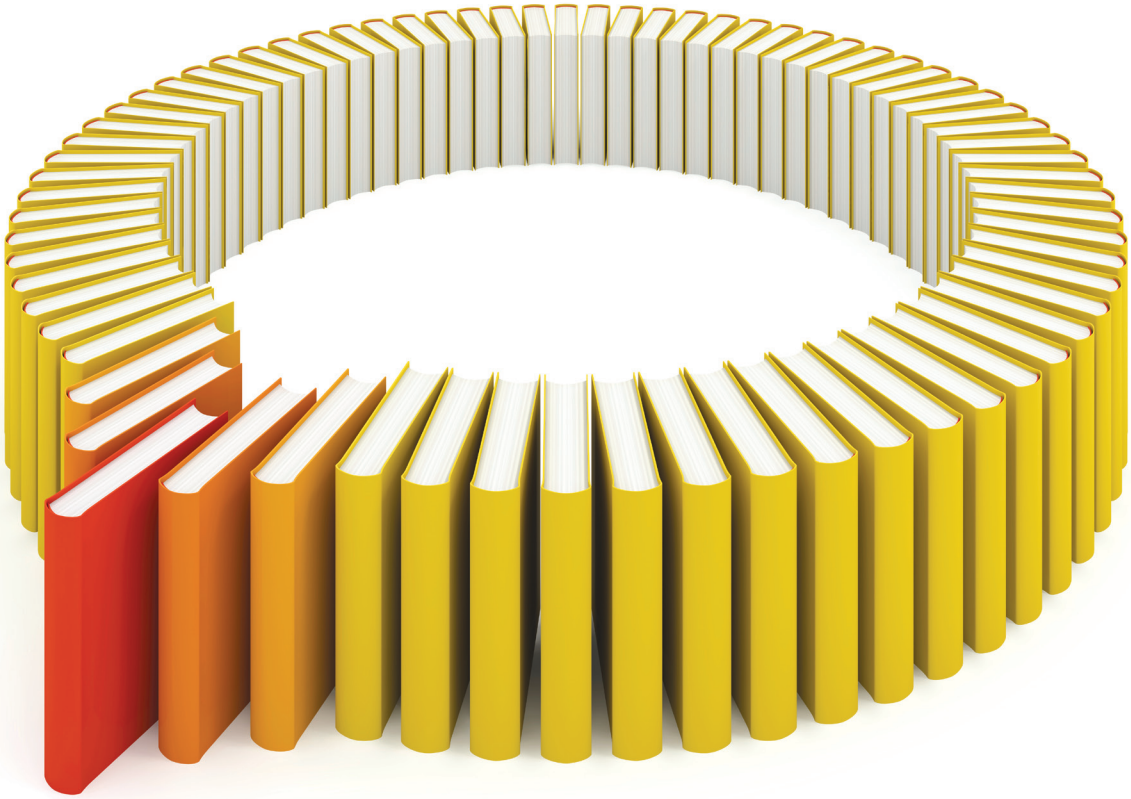
Example

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

```
proc timedata data=sashelp.air out=out outarray=outarray;
   id date interval=month;
   var air;
   outarrays lags df wn wnprob wnlprob;
   register tsa;
   declare object TSA(tsa);
   rc=TSA.WHITENOISE(air, 3, lags, df, wn, wnprob, wnlprob);
run;
```

Subject Index

Syntax Index



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.

sas
THE POWER TO KNOW.®

SAS and all other SAS Institute Inc. product or service names are registered trademarks or trademarks of SAS Institute Inc. in the USA and other countries. ® indicates USA registration. Other brand and product names are trademarks of their respective companies. © 2013 SAS Institute Inc. All rights reserved. S107969US.0613